CHAPITRE II

Diviser pour régner

II.1 Présentation de la méthode

La méthode de diviser pour régner est une méthode qui permet, parfois de trouver des solutions efficaces à des problèmes algorithmiques. L'idée est de découper le problème initial, de taille n, en plusieurs sous-problèmes de taille sensiblement inférieure, puis de recombiner les solutions partielles.

L'exemple typique est l'algorithme de *tri fusion* : pour trier un tableau de taille n, on le découpe en deux tableaux taille $\frac{n}{2}$ et l'étape de *fusion* permet de recombiner les deux solutions en $n-1$ opérations. On peut l'écrire ainsi :

Algorithm 1 Tri fusion

```
1: procedure TriFusion(T)
2:     if n ≤ 1 then
3:         return T
4:     else
5:         n = |T|
6:         T_1 = TriFusion(T[0 ... n/2])
7:         T_2 = TriFusion(T[n/2 + 1 ... n − 1])
8:         return Fusion(T_1,T_2)
9:     end if
10: end procedure
```

On va estimer la complexité en comptant le nombre $T(n)$ de comparaisons effectuées par l'algorithme. On a vu qu'on obtient directement que

$$\begin{cases} T(0) = 0, \\ T(1) = 0, \\ T(n) \approx 2\,T(n/2) + n - 1. \end{cases}$$

le $\approx$ est là car il y a des parties entières à considérer pour être rigoureux.

II.2 Forme générale et théorème maître

La forme générale considérée dans ce cours va être :
- **Diviser :** on découpe le problème en a sous-problèmes de tailles $\frac{n}{b}$, qui sont de même nature, avec $a \geq 1$ et $b > 1$.
- **Régner :** les sous-problèmes sont résolus récursivement.
- **Recombiner :** on utilise les solutions aux sous-problèmes pour reconstruire la solution au problème initial en temps $\mathcal{O}(n^d)$, avec $d \geq 0$.

L'équation qu'on aura à résoudre quand on traduit le programme en équation sur la complexité est :

$$\begin{cases} T(1) = \text{constante}, \\ T(n) \approx a\,T\left(\frac{n}{b}\right) + O(n^d). \end{cases}$$

Le théorème maître permet de résoudre ce type d'équations.

Théorème II.1 (Théorème Maître) *On considère l'équation $T(n) = a\,T\left(\frac{n}{b}\right) + \mathcal{O}(n^d)$. Soit $\lambda = \log_b a$. On a les trois cas suivants :*

1. *si $\lambda > d$, alors $T(n) = O(n^\lambda)$;*
2. *si $\lambda = d$, alors $T(n) = O(n^d \log n)$;*
3. *si $\lambda < d$, alors $T(n) = O(n^d)$.*

► Par exemple, pour le tri fusion, on a $a = 2$, $b = 2$, $\lambda = d = 1$ et donc une complexité de $O(n \log n)$.

► En pratique, seuls les cas 1. et 2. peuvent mener à des solutions algorithmiques intéressantes. Dans le cas 3., tout le coût est concentré dans la phase "recombiner", ce qui signifie souvent qu'il y a des solutions plus efficaces.

II.3 Exemples

II.3.1 Dichotomie

Si T est un tableau trié de taille n, on s'intéresse à l'algorithme qui recherche si $x \in T$ au moyen d'une dichotomie. Pour l'algorithme récursif, on spécifie un indice de début d et de fin f, et on recherche si x est dans T entre les positions d et f. L'appel initial se fait avec $d = 0$ et $f = n - 1$. Voir l'algorithme 2 pour la description.

On identifie les paramètres : $a = 1$ car on appelle soit à gauche, soit à droite (ou on a fini, mais on se place dans le pire des cas), $b = 2$ car les sous-problèmes sont de taille $n/2$ et $d = 0$ car on se contente de renvoyer la solution, donc en temps constant. La complexité de la dichotomie est donc $\mathcal{O}(\log n)$.

II.3.2 Exponentiation rapide

Il s'agit de calculer x^n pour x et n donnés, en calculant la complexité par rapport à n. La méthode naïve (multiplier n fois 1 par x) donne une complexité linéaire. On peut faire mieux

Algorithm 2 Dichotomie

```
1: procedure RECHERCHE(T,x,d,f)
2:     if f < d then
3:         return Faux
4:     else
5:         m = ⌊(b+a)/2⌋
6:         if T[m] = x then
7:             return Vrai
8:         else if T[m] < x then
9:             return Recherche(T,x,m + 1,f)
10:        else
11:            return Recherche(T,x,d,m − 1)
12:        end if
13:    end if
14: end procedure
```

en utilisant le fait que

$$\begin{cases} x^0 = 1, & \\ x^n = (x^2)^{\frac{n}{2}} & \text{si n est pair et strictement positif,} \\ x^n = x(x^2)^{\frac{n-1}{2}} & \text{si n est impair.} \end{cases}$$

On peut directement traduire cette constatation en algorithme. Et on retrouve les mêmes pa-

Algorithm 3 Exponentiation rapide

```
1: procedure PUISSANCE(x,n)
2:     if n = 0 then
3:         return 1
4:     else
5:         if n est pair then
6:             return Puissance(x ∗ x, n/2)
7:         else
8:             return x∗Puissance(x ∗ x, (n−1)/2)
9:         end if
10:    end if
11: end procedure
```

ramètres pour le théorème maître que dans le cas de la dichotomie. La complexité de l'exponentiation rapide est donc en $\mathcal{O}(n \log n)$.

► L'exponentiation rapide peut être utilisée pour des "multiplications" plus compliquées, comme la multiplication de matrices, la composition de fonctions, ... Dans ces cas, il ne faut pas oublier de compter le coût de la multiplication dans les calculs, qui n'est pas toujours constante.

II.3.3 Algorithme de Karatsuba

► On rappelle qu'un polynôme P est de la forme

$$P(X) = a_0 + a_1X + a_2X^2 + a_3X^3 + \ldots + a_nX^n = \sum_{i=0}^{n} a_iX^i,$$

où les a_i sont appelés les coefficients de P. Attention, il y a $n+1$ coefficients. On peut naturellement représenter P en machine par un tableau de taille $n+1$ et avec $P[0] = a_0$, $P[1] = a_1$,...

► Les polynômes sont abondamment utilisés en informatique. Ils sont par exemple de bons outils pour approximer des fonctions plus complexes.

► Si $P = \sum_{i=0}^{n} a_iX^i$ et $Q = \sum_{i=0}^{n} b_iX^i$, calculer le polynôme $R = P+Q$ est facile, car l'addition des polynômes revient à l'addition deux à deux des coefficients de même rang. On a ainsi

$$P + Q = (a_0 + b_0) + (a_1 + b_1)X + (a_2 + b_2)X^2 + \ldots (a_n + b_n)X^n.$$

On peut donc le calculer en temps $\mathcal{O}(n)$ en faisant une simple boucle.

► La multiplication des polynômes est plus compliquée, si on développe les premiers termes, on a

$$PQ = a_0b_0 + (a_0b_1 + a_1b_0)X + (a_0b_2 + a_1b_1 + a_2b_0)X^2 + \ldots + a_nb_nX^{2n}.$$

La formule général pour le k-ème coefficient c_k de PQ c'est

$$c_k = \sum_{i+j=k} a_ibj.$$

Si on implémente cette règle en algorithme, on obtient une multiplication de polynômes de complexité $\mathcal{O}(n^2)$.

► L'objectif est d'obtenir une multiplication plus rapide. Pour cela on commence à décomposer P et Q en deux polynômes. On écrit[1]

$$P = R \cdot X^{n/2} + S, \qquad Q = T \cdot X^{n/2} + U,$$

où R, S, T et U sont des polynômes de taille $\frac{n}{2}$.

On peut multiplier les deux expressions et on obtient

$$PQ = RT \cdot X^n + (RU + ST) \cdot X^{n/2} + SU.$$

On peut effectuer les 4 produits RT, RU, ST et SU récursivement, puis recombiner en temps linéaire (on a juste à faire des sommes et des décalage (multiplier par X^i c'est décaler de i cases les coefficients). On est dans un cas typique de diviser pour régner, avec les paramètres $a = 4$, $b = 2$ et $d = 1$. Le théorème maître nous donne une complexité de $\mathcal{O}(n^2)$: on n'a rien gagné.

► Pour améliorer la complexité il faut introduire une nouvelle idée. C'est ce qu'a fait Karatsuba en remarquant qu'on peut aussi écrire le produit comme :

$$PQ = RT \cdot X^n + ((R + S)(T + U) - (RT + SU)) \cdot X^{n/2} + SU.$$

Cela semble plus compliqué, mais on remarque qu'on a plus que trois produits plus petits à effectuer (RT, SU et $(R+S)(T+U)$). Le reste se fait en temps linéaire, on a donc les paramètres $a = 3$, $b = 2$ et $d = 1$. Le théorème maître donne une complexité de $\mathcal{O}(n^{\log_2 3}) \approx \mathcal{O}(n^{1.585})$: on a gagné significativement en efficacité.

[1] Pour cet algorithme on ne s'occupe pas de bien faire les $\frac{n}{2}$ selon la parité de n : on s'autorise à écrire $\frac{n}{2}$ partout. Le traitement rigoureux avec les parties entières ne change pas le résultat.

II.4 Théorème d'Akra-Bazzi (1998)

▶ Dans tous les exemples de ce chapitre on a approximé les formules pour ne pas faire apparaître les parties entières et les légers décalages. Par exemple, si on écrit la formule exacte pour le tri fusion, on obtient que

$$T(n) = T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + \mathcal{O}(n).$$

Les autres algorithmes peuvent donner lieu à des formules encore plus compliquées, avec des -1 en plus dans les appels récursifs.

▶ Heureusement, Akra et Bazzi ont montré une extension du théorème maître qui permet de travailler avec les approximations :

Théorème II.2 *Si on a*

$$T(n) = a\,T\left(\frac{n}{b} + h(n)\right) + O(n^d), \qquad h(n) = O\left(\frac{n}{(\log n)^2}\right)$$

alors le résultat du théorème maître est encore valable.

▶ En particulier, on peut remplacer des quantités comme $\lfloor \frac{n+1}{b} \rfloor - 3$ par $\frac{n}{b}$ et appliquer le théorème maître avec le bon résultat, comme on l'a fait jusqu'ici. Le théorème d'Akra-Bazzi permet de valider mathématiquement les approximations que l'on faisait.

▶ Le vrai théorème d'Akra-Bazzi est encore plus général, puisqu'il permet de résoudre des récurrences avec des a et des b différents, comme par exemple

$$T(n) = 3T\left(\frac{n}{2}\right) + 2T\left(\frac{n}{3}\right) + T\left(\frac{n}{5}\right) + \mathcal{O}(n).$$

Cela dépasse largement le cadre de ce cours, les curieux trouveront plus d'information sur wikipedia.