Introduction à l'algorithmique et la complexité (et un peu de CAML)

Diviser pour Régner

Nicolas Nisse

Université Côte d'Azur, Inria, CNRS, I3S, France

Cours dispensés en MPSI (option Info) au CIV, depuis 2011-

http://www-sop.inria.fr/members/Nicolas.Nisse/lectures/prepa/

Outline

Diviser pour Régner ("Divide and Conquer" en anglais)

Nous entrons maintenant dans le cœur de ce cours, à savoir, concevoir des algorithmes efficaces.

Une méthode pour cela est celle dite de Diviser pour régner.

Étant donnés un problème Π et une instance $\mathscr{I}$ pour ce problème, il s'agit de déterminer des instances $\mathscr{I}_1, \cdots, \mathscr{I}_k$ **plus petites et "indépendantes/disjointes"** les unes des autres, telles que, pour résoudre (récursivement) Π pour $\mathscr{I}$, il est suffisant de résoudre Π pour chacune des instances $\mathscr{I}_1, \cdots, \mathscr{I}_k$ puis combiner les solutions obtenues.

Nous avons déjà vu des exemples de tels algorithmes :

Exponentiation rapide : (slide 14, Chap 3). Il s'agit de calculer x^c, et pour cela, l'algorithme résout (récursivement) le "sous-problème" $x^{c/2}$ (de "taille" deux fois plus petite) et utilise la solution obtenue pour calculer x^c.

Recherche dichotomique : (slide 17, Chap 3). Il s'agit de trouver un élément x dans un tableau *tab* trié. Pour cela, on détermine dans quelle moitié de *tab* se trouve x et on cherche (récursivement) dans un tableau (trié) deux fois plus petit.

Tri Fusion : (slides 9-10, Chap 4). On veut trier un tableau *tab*, pour cela, on divise *tab* en 2 tableaux (disjoints) deux fois plus petits, que l'on trie récursivement, et on fusionne les résultats.

Outline

Résolution de $u_k = a \cdot u_{k-1} + O(b^k)$

Lors de l'étude de la complexité d'un algorithme récursif de type "diviser pour régner", on rencontre souvent des suites de la forme $u_0 = O(1)$ et $u_k = a \cdot u_{k-1} + O(b^k)$ (rappelez vous le Tri_Fusion avec $a = b = 2$, slide 10 Chap. 4). Nous apprenons ici à déterminer u_k.

Th : Soient $a, b \in \mathbb{R}^{+*}$ et $(u_k)_{k\mathbb{N}}$ telle que $u_0 = O(1)$ et $u_k = a \cdot u_{k-1} + O(b^k)$

- $u_k = O(a^k)$ si $a > b$;
- $u_k = O(b^k)$ si $b > a$;
- $u_k = O(k \cdot a^k)$ si $a = b$.

Preuve : Posons $v_k = u_k / a^k$. Alors $v_k = v_{k-1} + O((\frac{b}{a})^k)$. Si $a = b$, alors $v_k = v_{k-1} + O(1)$, d'où $v_k = O(k)$ et $u_k = O(k \cdot a^k)$. Sinon, $v_k = O(\sum_{0 \leq j \leq k} (\frac{b}{a})^k) = O(\frac{1-(b/a)^{k+1}}{1-b/a})$.
Si $a > b$, $v_k = O(1)$ et $u_k = O(a^k)$. Si $b > a$, $v_k = O((b/a)^k)$ et $u_k = O(b^k)$.

La propriété suivante sera aussi souvent utilisée. Soit $n = 2^k$ et supposons que $c(n) = O(x^k)$ alors, $c(n) = O(n^{\log_2 x})$, en effet, $k = \log_2 n$ et donc
$x^k = (2^{\log_2 x})^k = (2^{\log_2 x})^{\log_2 n} = (2^{\log_2 n})^{\log_2 x} = n^{\log_2 x}$. (rappel : $y = 2^{\log_2 y}$ par définition de $\log_2$)

Outline

Multiplication de matrices

Une matrice est repésentée ici par un tableau de tableaux (un tableau de “lignes").
Étant données deux matrices $n \times n$: $A = [a_{i,j}]_{1 \leq i,j \leq n}$ et $B = [b_{i,j}]_{1 \leq i,j \leq n}$, on désire calculer leur produit $C = A \cdot B = [c_{i,j}]_{1 \leq i,j \leq n} = [\sum_{1 \leq k \leq n} a_{i,k} b_{k,j}]_{1 \leq i,j \leq n}$.

```
>       Caml Light version 0.80

#let mult a b =
        let n = vect_length a in
        let c = make_matrix n n 0 in
        for i=0 to n-1 do
                for j=0 to n-1 do
                        let temp = ref 0 in
                        for k=0 to n-1 do
                                temp := !temp + a.(i).(k) * b.(k).(j);
                        done;
                        c.(i).(j) <- ! temp;
                done;
        done;
        c;;
mult : int vect vect -> int vect vect -> int vect vect = <fun>
#
```

Terminaison : triple boucles imbriquées

Correction : laissée au lecteur

Complexité : triple boucles imbriquées.
$c(mult) = O(n^3)$.

Il est possible de faire mieux que le $O(n^3)$ de cet algorithme “naïf" comme nous allons le voir dans la suite.

Exercice : Étant données deux matrices $n \times q$: $A = [a_{i,j}]$ et $B = [b_{i,j}]$, donnez un algorithme pour calculer $C = A + B = [a_{i,j} + b_{i,j}]_{1 \leq i \leq n, 1 \leq j \leq q}$ en temps $O(nq)$.

Multiplication de matrices

Aparté de programmation. Tester les 2 algorithmes ci-dessous et vérifier qu'ils "buggent"

```
#let mult a b =
        let n = vect_length a in
        let ligne = make_vect n 0 in
        let c = make_vect n ligne in
        for i = 0 to n-1 do
                for j = 0 to n-1 do
                        let tmp = ref 0 in
                        for k =0 to n-1 do
                                tmp := !tmp + a.(i).(k) * b.(k).
                        done;
                        c.(i).(j) <- !tmp;
                done;
        done;
        c;;
mult : int vect vect -> int vect vect -> int vect vect = <fun>
```

```
>       Caml Light version 0.80

#let mult a b =
        let n = vect_length a in
        let c = make_vect n (make_vect n 0) in
        for i=0 to n-1 do
                for j=0 to n-1 do
                        let temp = ref 0 in
                        for k=0 to n-1 do
                                temp := !temp + a.(i).(k) * b.(k).(
                        done;
                        c.(i).(j) <- ! temp;
                done;
        done;
        c;;
mult : int vect vect -> int vect vect -> int vect vect = <fun>
```

Cela est dû au fait que *let c = make_vect n init*, si *init* est mutable (comme c'est le cas de *init = ligne = make_vect n* 0 et de *init = make_vect n* 0), "met" dans chaque "case" de *c* une référence à *init* (et non sa valeur). Ainsi, si la valeur de *init* est modifié(e), elle l'est dans chaque "occurrence de sa référence" (donc, ici, dans chaque "case").

Algorithme de Strassen (1969)

Soit $p \in \mathbb{N}^*$. Pour simplifier la présentation, supposons que N et M (dont on veut faire le produit) sont des matrices $2^p \times 2^p$. Soient A, B, C, D, E, F, G et H les huit matrices $2^{p-1} \times 2^{p-1}$ telles que $M = \begin{bmatrix} A & B \\ C & D \end{bmatrix}$ et $N = \begin{bmatrix} E & F \\ G & H \end{bmatrix}$. Soient $P_1 = A*(F-H)$, $P_2 = (A+B)*H$, $P_3 = (C+D)*E$, $P_4 = D*(G-E)$, $P_5 = (A+D)*(E+H)$, $P_6 = (B-D)*(G+H)$ et $P_7 = (A-C)*(E+F)$.

L'algorithme de Strassen repose sur le fait suivant.

Prouver que $M*N = \begin{bmatrix} -P_2+P_4+P_5+P_6 & P_1+P_2 \\ P_3+P_4 & P_1-P_3+P_5-P_7 \end{bmatrix}$

Algorithme de Strassen — complexité $c(p)$

- Si $p=0$, $MN = [m_{0,0} * n_{0,0}]$, sinon — $O(1)$ opérations
- Calcul de $F-H$, $A+B$, $C+D$, $G-E$, $A+D$, $E+H$, $B-D$, $G+H$, $A-C$ et $E+F$, soit 10 additions de matrices $2^{p-1} \times 2^{p-1}$. — $O((2^{p-1})^2)$ opérations.
- Calcul des P_i avec 7 multiplications de matrices $2^{p-1} \times 2^{p-1}$. — $7c(p-1)$ opérations.
- Calcul de $-P_2+P_4+P_5+P_6$; P_1+P_2; P_3+P_4 et $P_1-P_3+P_5-P_7$: quelques additions/soustractions de matrices $2^{p-1} \times 2^{p-1}$ — $O((2^{p-1})^2)$ opérations.

Donc, $c(0) = O(1)$ et $c(p) = 7c(p-1) + O(4^p)$. Donc $c(p) = 7^p$ (cf. Th. Slide 5). Si $2^{p-1} < n \leq 2^p$, la multiplication de 2 matrices $n \times n$ est réalisée en $O(7^p) \approx O(n^{2,8})$ opérations.

Rmq: *Le meilleur algorithme connu a une complexité de $O(n^{2.3728639})$ [François Le Gall (2014)].*

Outline

Multiplication de polynômes

Soient $P(X) = \sum_{i=0}^{n} a_i X^i$ et $Q(X) = \sum_{i=0}^{n} b_i X^i$ $(a_n, b_n \neq 0)$ deux polynômes de $\mathbb{R}[X]$ de degré n représentés par les tableaux de leurs coefficients (les algorithmes ci-dessous supposent que P et Q ont même degré, le cas général est laissé au lecteur).

On veut déterminer le tableau des coefficients de $P \cdot Q = \sum_{i=0}^{2n} (\sum_{j=0}^{i} a_j b_{i-j}) X^i$.

```
#let multPol p q =
        let n = vect_length p in
        let res = make_vect (2*n-1) 0 in
        for i = 0 to (2*n-2) do
                for j=0 to i do
                        res.(i) <-  res.(i) + p.(j)*q.(i-j);
                done;
        done;
        res;;
multPol : int vect -> int vect -> int vect = <fun>
```

Terminaison : double boucles imbriquées

Correction : laissée au lecteur

Complexité : double boucles imbriquées.
$c(multPol) = O(n^2)$.

Il est possible de faire mieux que le $O(n^2)$ de cet algorithme “naïf" comme nous allons le voir dans la suite.

Multiplication de polynômes, fonctions auxiliaires

Continuons avec de “simples" fonctions sur les polynômes.

Addition de 2 polynômes

Exercice : Ecrire un algorithme qui prends (les coefficients de) 2 polynômes $P(X) = \sum_{i=0}^{p} a_i X^i$ et $Q(X) = \sum_{i=0}^{q} b_i X^i$ et calcule les coefficients du polynôme $(P+Q)(X) = \sum_{i=0}^{\max\{p,q\}} (a_i + b_i) X^i$ en temps $O(\max\{p,q\})$.

Multiplication par un monôme

L'algorithme ci-dessous calcule les coefficients du polynôme $X^q P(X)$ en temps $O(q + degre(P))$.

```
#let multMonome p q =
        let n = vect_length p in
        let res = make_vect (n+q) 0 in
        for i=q to n-1+q do
                res.(i) <- p.(i-q)
        done;
        res;;
multMonome : int vect -> int -> int vect = <fun>
```

Division euclidienne d'un polynôme par un monôme

Rappel : Étant donné un polynôme $P(X) = \sum_{i=0}^{p} a_i X^i$ et $q \leq n$, il existe deux uniques polynômes $A(X) = \sum_{i=0}^{n-q} a_{i+q} X^i$ et $B(X) = \sum_{i=0}^{q-1} a_i X^i$ tels que $P(X) = X^q A(X) + B(X)$.

Exercice : Ecrire un algorithme qui prends $P(X)$ et $q \leq n$ et calcule (les coefficients de) $A(X)$ et $B(X)$ tels que $P(X) = X^q A(X) + B(X)$ en temps $O(n)$.

Multiplication de polynômes, méthode de Karatsuba (1962)

Soit $p \in \mathbb{N}^*$. Pour simplifier la présentation, supposons que P et Q (dont on veut faire le produit) sont des polynômes de degré $n = 2^p$. Soient A, B, C et D les quatre polynômes de degré $\leq n/2$ ($< n/2$ pour B et D) tels que $P(X) = X^{n/2}A(X) + B(X)$ et $Q(X) = X^{n/2}C(X) + D(X)$.

L'algorithme de Karatsuba repose sur le fait suivant.

Prouver que
$P \cdot Q(X) = X^n A(X)C(X) + X^{n/2}((A(X)+B(X))(C(X)+D(X)) - A(X)C(X) - B(X)D(X))$

Algorithme de Strassen — complexité $c(p)$

- Si $p = 0$, on conclut facilement, sinon — $O(1)$ opérations
- Calcul de $A+B$ et $C+D$: 2 additions de polynômes de degré 2^{p-1}. $O(2^{p-1})$ opérations.
- Calcul de $A \cdot C$, $(A+B)(C+D)$ et $B \cdot D$, soit 3 multiplications de polynômes de degré 2^{p-1}. $3c(p-1)$ opérations.
- Multiplication par X^n, multiplication par $X^{n/2}$ et quelques additions/soustractions de polynômes de degré $n = 2^p$ — $O(2^p)$ opérations.

Donc, $c(0) = O(1)$ et $c(p) = 3c(p-1) + O(2^p)$. Donc $c(p) = 3^p = n^{\log 3} \approx n^{1.585}$ (cf. Th. Slide 5).