09/01
2015

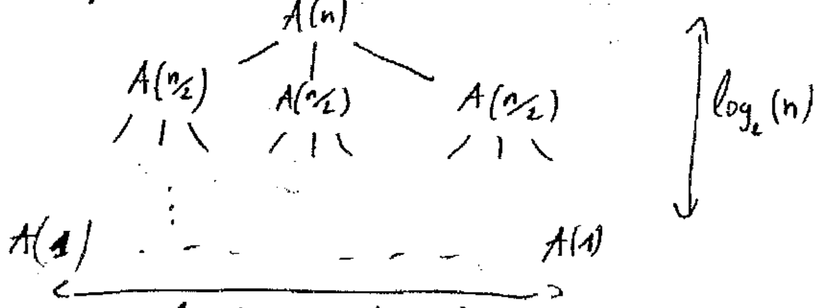
902 | DIVISER POUR REGNER

I Le principe de la méthode

Il s'agit d'une méthode récursive pour donner une solution à un problème, en trois étapes

1. On divise le problème en sous problèmes de tailles inférieures
2. On traite recursivement les sous problème, et on "règne" directement sur les cas de base
3. On combine de manière appropriée les résultats

Remarque : La structure d'execution de l'algorithme est un arbre. Par exemple dans le cas d'une division en 3 sous problèmes de taille 2, on a



$$3^{\log_2(n)} = n^{\log_2(3)}$$

Exemple d'algorithme : recherche d'un element dans un arbre

appartient (x, T)

- Si $T =$ feuille(y)
  - renvoyer oui ssi $y = x$
- Sinon
  - $\bigvee_{T' \text{ fils de } T}$ appartient(x, T')

II Correction d'algorithme

Toutes les preuves de correction se font par induction sur l'arbre de calcul.

Exemple d'algorithme : Recherche de distance minimale dans un ensemble de points

PPP $(A\,[(x_1, y_1), \dots (x_n, y_n)])$

- Si $n \leq 3$, $(n \geq 2)$
  - $d = \min(|(x_1,y_1)-(x_2,y_2)|, |(x_1,y_1)-(x_3,y_3)|, |(x_2,y_2)-(x_3,y_3)|)$
- Sinon
  - Séparer A en A_G et A_D, de même taille et tels que les points de A_G soient à gauche des points de A_D
  - Soit $d = \inf(PPP(A_G), PPP(A_D))$
  - Pour $(x_i, y_i) \in A_D$, $(x_j, y_j) \in A_G$
    - $d = \min(d, |(x_i, y_i) - (x_j, y_j)|)$
  - Retourner d

Preuve de correction

Cas $n \leq 3$: la distance est bien calculée.

Autres cas : • En divisant, on se retrouve avec au moins 2 points par ensembles

- La distance sur ces sous ensembles est calculée
- La distance minimale est atteinte, soit dans les parties droite et gauche, soit en chevauchement.

La correction des algorithmes de type diviser pour régner est assez simple généralement, l'étude de la complexité est des plus intéressants.

III Calcul de complexité

1) Un exemple : multiplication d'entiers

Soient x, y deux entiers codés sur $n = 2^k$ bits

$$\begin{cases} x = 2^{n/2} x_G + x_D \\ y = 2^{n/2} y_G + y_D \end{cases} \quad \text{où } x_G, y_G, x_D, y_D \text{ sont codés sur } 2^{k-1} \text{ bits}$$

Alors $x \times y = 2^n x_G y_G + 2^{n/2}(x_G y_D + x_D y_G) + x_D y_D$

Les additions sont de complexité linéaire

$$T(n) = 4\, T(n/2) + O(n)$$

$$\Rightarrow \frac{T(n)}{n^2} = \frac{T(n/2)}{(n/2)^2} + O\left(\frac{1}{n}\right) \Rightarrow \underline{T(n) = O(n^2)}$$

Amélioration : $x_G y_D + x_D y_G = (x_D + x_G)(y_G + y_D) - x_G y_G - x_D y_D$

On peut économiser une multiplication en ajoutant des additions : $T'(n) = 3\, T'(n/2) + O(n)$

$$\Rightarrow \frac{T'(n)}{n^{\log_2(3)}} = \frac{T'(n/2)}{(n/2)^{\log_2(3)}} + O\left(n^{1-\log_2(3)}\right)$$

$$\Rightarrow T'(n) = O\left(n^{\log_2(3)}\right)$$

On retrouve cette idée pour la multiplication de matrices

2) Théorème général

> Soit $a \geq 1$, $b > 1$ et $T(n)$ défini par la relation de récurrence $\begin{cases} T(n) = a\,T(\lceil n/b \rceil) + f(n) \\ T(1) = 1 \end{cases}$
>
> et si $f(n) = O(n^d)$ Alors
>
> $$T(n) = \begin{cases} O(n^d) & \text{si } d > \log_b(a) \\ O(n^d \log(n)) & \text{si } d = \log_b(a) \\ O(n^{\log_b(a)}) & \text{si } d < \log_b(a) \end{cases}$$

remarque L'algorithme pour la multiplication d'entiers est un exemple du troisième cas.

3) Application du deuxième cas : Algorithme du Tri fusion

Tri fusion $a[1, \dots n]$
- Si $n \geq 1$
  - retourner fusion (tri fusion $a[1 .. \lfloor n/2 \rfloor]$, tri fusion $a[\lfloor n/2 \rfloor + 1, \dots n]$)
- Sinon
  - retourner a

avec fusion $(a[1, \dots k], b[1, \dots \ell])$
- Si $k = 0$, retourner b
- si $\ell = 0$ retourner a
- Sinon
  - Si $a[1] \leq b[1]$
    - retourner $a[1] \circ$ fusion $(a[2, .. k], b[1, .. \ell])$
  - Sinon
    - retourner $b[1] \circ$ fusion $(a[1 \dots k], b[2, \dots \ell])$

- fusion s'exécute en $O(n)$

$$T(n) = 2\,T(n/2) + O(n) \quad \text{or} \quad 1 = \log_2(2)$$

$$\Rightarrow T(n) = O(n \log n)$$

On peut adapter le tri fusion à d'autres modèles de calcul : TRI POLYPHASÉ (Développement)

4) Division en sous problèmes de taille aléatoire

L'algorithme suivant calcule le k-ième élément par ordre croissant dans une liste non triée

Recherche $(K, S[1,\dots n])$

pivot $= S[1]$

On sépare S en $\left.\begin{array}{l} S_e : \text{éléments égaux au pivot} \\ S_i : \text{élément inférieurs au pivot} \\ S_s : \text{éléments supérieurs au pivot.} \end{array}\right\} O(n)$

Si $K \leq |S_i|$, Recherche (K, S_i)

Si $K > |S_i| + |S_e|$ Recherche $(K - |S_i| - |S_e|, S_s)$

Sinon retourne pivot

Complexité : Dans le pire des cas, on enlève un élément à chaque fois : $n + (n-1) + \dots + 1 = O(n^2)$

en moyenne, Si le pivot est dans le 2^{eme} ou 3^{eme} quart du tableau (50% des cas), on réduit le tableau d'au moins $3/4$

L'espérance du nb d'itération avant de trouver un élément dans le 2^{eme} ou 3^{eme} quart du tableau est 2.

Donc $E[T(n)] \leq E[T(3n/4)] + O(n) + O(n)$

$\log_{4/3}(1) < 1 \implies E[T(n)] = O(n)$

IV Applications en analyse numérique

1) Transformée de Fourier rapide

La transformée de Fourier rapide permet la multiplication de Polynômes de degré $\leq n$ en temps $O(n \log n)$ plutôt que $O(n^2)$ (Développement)

2) Multiplications de matrices

$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$

$$AB = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{21}B_{11} + A_{22}B_{21} \\ A_{11}B_{12} + A_{12}B_{22} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}$$

On a $T(n) = 8T(n/2) + O(n^2) \Rightarrow T(n) = O(n^{\log_2(8)}) = O(n^3)$

L'algorithme de Strassen propose une méthode pour avoir à effectuer plus d'additions mais moins de multiplications de sous matrices (7) à la manière de la multiplication de grands entiers. On obtient une complexité

$$T(n) = 7\,T(n/2) + O(n^2) \Rightarrow T(n) = O(n^{\log_2(7)}).$$

De meilleurs algorithmes existent, avec des complexités jusqu'à $O(n^{2,376})$, en découpant la matrice en plus de sous problèmes plus petits.

Références:

- Algorithmique, Cormen, Leiserson, Rivest, Stein
  DUNOD
- Algorithms, Dasgupta, Papadimitriou, Vazirani, McGraw-Hill Higher Ed.

Tri polyphasé

Mathias Millet

Froidevaux.

9 janvier 2015

Modèle de calcul

On se place dans le modèle suivant : ?
- Un processeur avec un une mémoire centrale de taille M
- Des bandes magnétiques pour stoquer les données, chacune équipée d'une tête de lecture qui peut lire ou écrire une case mémoire à la fois. On supposera les bandes de longueur non bornée.

On dispose donc des opérations supplémentaires suivantes :
- `var` ← `lis`$_i$: donne à la variable `var` la valeur lue sur la bande i
- `écris`$_i$(`var`) : écris la valeur de `var` sur la bande i
- `avance`$_i$: avance d'une case mémoire sur la bande i
- `recule`$_i$: recule d'une case mémoire sur la bande i

Compléxité Entrées/Sorties Dans ce modèle de calcul, on peut définir la compléxité entrées/sorties d'un algorithme comme le nombre d'opération *avance* et *recule* effectuées pendant une exécution.

↳ plutôt écrire et lire ne sont pas dedans. parce que bouger sur le ruban coûte cher !

Tri polyphasé

Définition : Monotonie Une monotonie est tableau trié par ordre croissant, placé sur une bande magnétique

Définition : Fusion On définit une fusion de deux monotonies de taille m et n placées chacune sur une bande différente comme la création d'une nouvelle monotonie, sur une autre bande, obtenue en interclassant les éléments de chacune des monotonies de départ. La compléxité est en $O(m+n)$ (en supposant que les têtes de lecture sont initialement placées aux bons endroits).

On suppose en général que les données lues sont effacées au fur et à mesure.

Algorithme du tri polyphasé On suppose que les données à trier se trouvent sur une seule bande. La taille de ces données est N. On effectue les étapes suivantes :
- On trouve n tel que $\frac{N}{F_n} > M \geq \frac{N}{F_{n+1}}$ [1]
- On découpe les données en paquets de taille $t = \left\lceil \frac{N}{F_{n+1}} \right\rceil$ (on obtient donc F_{n+1} paquets), ← on ajoute des éléments dans N. qu'on trie en mémoire, puis qu'on replace sur les autres bandes, obtenant ainsi des monotonies. On en place F_n sur une bande, F_{n-1} sur l'autre.
- On fusionne alors les paquets selon la façon explicitée ci-dessous.

1. F_n représente n$^{\text{ème}}$ élément de la suite de Fibonacci, où l'on a choisi $F_0 = 0$ et $F_1 = 1$

Fusion des monotonies Au début de la $i^{\text{ème}}$ étape de fusion, on a la situation suivant

1. Une bande avec F_{n-i} monotonies de taille $t \cdot F_i$
2. Une bande avec F_{n-i+1} monotonies de taille $t \cdot F_{i+1}$
3. Une bande vide

On fusionne alors 2 à 2 les monotonies des deux premières bandes, en plaçant les nouvel monotonies obtenues sur la troisième bande. On effectue ainsi F_{n-i} fusions, de complexité $F_i + t \cdot F_{i+1}$.

On obtient à la fin de la fusion :

1. Une bande vide
2. Une bande avec F_{n-i-1} monotonies de taille $t \cdot F_{i+1}$
3. Une bande avec F_{n-i} monotonies de taille $t \cdot F_{i+2}$

On obtient une unique monotonie à la fin de la $(n-1)^{\text{ème}}$ fusion. Les données sont al triées !

Calcul de complexité On obtient donc pour la complexité globale de l'algorithme :

$$O\left(\sum_{i=1}^{n-1} F_{n-i} \cdot t \cdot F_{i+2}\right)$$

On dispose de la formule (magique) : $\displaystyle\sum_{k=0}^{n} F_{n-k}F_k = \frac{n-1}{5}F_n + \frac{2n}{5}F_{n-1}$

On a alors :

$$\begin{aligned}
C &= t \cdot \sum_{i=1}^{n-1} F_{n-i}F_{i+2} = t \cdot \sum_{i=1}^{n-1} F_{n+2-(i+2)}F_{i+2} = t \cdot \sum_{k=3}^{n+1} F_{n+2-k}F_k \\
&= t \cdot \left(\sum_{k=0}^{n+2} F_{n+2-k}F_k \quad - \quad (2 \cdot F_0F_{n+2} + F_1F_{n+1} + F_2F_n)\right) \\
&= t \cdot \left(\frac{n+1}{5}F_{n+2} + \frac{2(n+2)}{5}F_{n+1} - F_{n+2}\right) = t \cdot \left(\frac{n-4}{5}F_{n+2} + \frac{2(n+2)}{5}F_{n+1}\right)
\end{aligned}$$

Or on sait que $t = \dfrac{N}{F_{n+1}}$, on a donc $C = N \cdot \left(\dfrac{n-4}{5}\dfrac{F_{n+2}}{F_{n+1}} + \dfrac{2(n+2)}{5}\right)$

On utilise maintenant la deuxième formule magique : $F_n = \frac{1}{\sqrt{5}}(\phi^n + \bar{\phi}^n)$ avec $|\phi| > 1$, $|\bar{\phi}|$

de laquelle on tire $\dfrac{F_{n+2}}{F_{n+1}} = O(1)$, et

$\log(F_n) = -\dfrac{1}{2}\log(5) + n\log(\phi) + \log(1 + o(1)) \Longleftrightarrow n = O(\log(F_n))$

On obtient donc que $C = O(N\log(F_n))$

Finalement, comme $\dfrac{N}{F_n} > M$, $F_n = O(\frac{N}{M})$, et la complexité de l'algorithme est

$$\boxed{O\left(N\log(\frac{N}{M})\right)}$$

Transformée de Fourier rapide

Référence: Cormen strict

Il existe deux méthodes pour représenter les polynômes de degré inférieur à n:

1. par la liste des coefficients du polynôme;
2. par n couples (point,valeur).

Les deux représentations sont équivalentes, le passage de la première à la deuxième par évaluation, la transformation inverse par interpolation e Lagrange . Voici le cout des opérations élémentaires dans les deux représentations:

représentation	addition	multiplication
coefficients	$O(n)$	$O(n^2)$
couples (point, valeur)	$O(n)$	$O(n)$ mais necessité de $2n$ couples

La transformée de Fourier rapide donne une conversion en complexité $O(n \log n)$ et donc une multiplication en $O(n \log n)$ pour la représentation classique par coefficients.

Idée de l'algorithme

Soit P un polynôme de degré inférieur à n donné par ses coefficients. A priori, l'évaluation de $P(x)$ est une opération en $O(n)$, donc pour obtenir n couples (point, valeur), une complexité $O(n^2)$. Cependant, on peut mutualiser des opérations pour l'évaluation, lorsqu'on en fait plusieurs en même temps (on suppose que n est pair):

$$\begin{aligned}
P(X) &= a_0 + a_1 X + \ldots + a_{n-1} X^{n-1} \\
P_0(X) &= a_0 + a_2 X + \ldots + a_{n-2} X^{n/2-1} \\
P_1(X) &= a_1 + a_3 X + \ldots + a_{n-1} X^{n/2-1}
\end{aligned}$$

Alors $P(x) = P_0(x^2) + xP_1(x^2)$ et $P(-x) = P_0(x^2) - xP_1(x^2)$. On s'est ramené à l'évaluation de 2 polynômes de degrés inférieurs à $n/2$. Si on suppose que n est une puissance de 2 et qu'on utilise les nombres complexes, en particulier les racines de l'unité, on peut itérer le procédé . L'algorithme suivant fourni une évaluation d'un polynôme P de degré inférieur ou égal à $n = 2^p$ donné par ses coefficients sur les racines n-ièmes de l'unité.

```
FFT(P[a_0...a_{n-1}])
  Si n=1
    Retourner a_0
  Sinon
    ω_n = e^{2iπ/n}
    ω = 1
    P_0 = [a_0; a_2...a_{n-2}]
    P_1 = [a_1; a_3...a_{n-1}]
    y^0 = FFT(P_0)                        Complexité 2T(n/2)
    y^1 = FFT(P_1)
    Pour k de 0 à n/2-1                   Complexité O(n)
      y_k = y^0_k + ω y^1_k
      y_{k+n/2} = y^0_k - ω y^1_k
      ω = ω ω_n
    Retourner y                           Complexité finale O(n log n)
```

Transformée de Fourier inverse

Le calcul précédent est également la multiplication du vecteur des coefficients par la matrice de dermonde V_n:

$$\begin{pmatrix} y_0 \\ \vdots \\ \vdots \\ \vdots \\ y_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 & \dots & 1 \\ 1 & \omega_n & \dots & \omega_n^{n-1} \\ 1 & \omega_n^2 & \dots & \omega_n^{2(n-1)} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{n-1} & \dots & \omega_n^{(n-1)^2} \end{pmatrix} \begin{pmatrix} a_0 \\ \vdots \\ \vdots \\ \vdots \\ a_{n-1} \end{pmatrix}$$

Pour effectuer la transformée de Fourier inverse, il suffit d'inverser la matrice V_n. Or $[V_n^{-1}]_{j,k} =$
En effet, si $W_n = (\frac{\omega_n^{-kj}}{n})_{j,k}$

$$\begin{aligned} [V_n W_n]_{i,j} &= \sum_{k=0}^{n-1} \omega_n^{ki} \frac{\omega_n^{-kj}}{n} \\ &= \frac{1}{n} \sum_{k=0}^{n-1} \omega_n^{k(i-j)} \\ &= \delta_j^i \end{aligned}$$

L'algorithme pour la transformée de Fourier rapide inverse s'en déduit, en modifiant légèrement pour la transformée de Fourier rapide:

FFTI($Y[y_0 ... y_{n-1}]$)

  Si n=1

    Retourner y_0

  Sinon

    $\omega_n = e^{-2i\pi/n}$

    $\omega = 1/n$

    $Y_0 = [y_0; y_2 ... y_{n-2}]$

    $Y_1 = [y_1; y_3 ... y_{n-1}]$

    $P^0 = \mathrm{FFTI}(Y_0)$ — Complexité $2T(n/2)$

    $P^1 = \mathrm{FFTI}(Y_1)$

    Pour k de 0 à n/2-1 — Complexité $O(n)$

      $P_k = P_k^0 + \omega P_k^1$

      $P_{k+n/2} = P_k^0 - \omega P_k^1$

      $\omega = \omega\omega_n$

    Retourner P — Complexité finale $O(n \log n)$

Conclusion

On dispose d'un algorithme asymptotiquement plus rapide que la multiplication classique pour tiplier entre eux deux polynômes donnés par leurs coefficients:

Multiplication

$$
\begin{tikzcd}
\boxed{\text{Coefficients}} \arrow[rr, "O(n^2)"'] \arrow[d, "O(n\log n)"'] \arrow[rr, bend right=40, "O(n\log n)"] & & \boxed{\text{Coefficients}} \\
\boxed{\text{Point, Valeur}} \arrow[rr, "O(n)"'] & & \boxed{\text{Point, Valeur}} \arrow[u, "O(n\log n)"']
\end{tikzcd}
$$

Cet algo et le tri fusion sont-ils implémentés ainsi en réalité ?