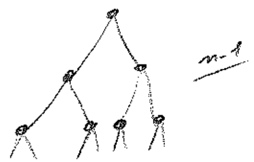
902 — Diviser pour régner : Exemples et applications.

Introduction : Apparition naturelle du "diviser pour régner"

Problème : Déterminer la meilleure équipe de Rugby parmis n équipes. On propose un algorithme naïf :

Vainqueur $j[1,\dots,n]$ =



```
    gagnant ← j[1];
    i ← 2;
    tant que i ≤ n faire
        si j[i] est plus fort que gagnant alors
            gagnant ← j[i].
        fin si
        i ← i+1
    fin tant que.
retourner gagnant.
```

Cet algorithme n'est pas très équitable pour l'équipe en position "gagnant" durant la compétition, car elle se fatigue. C'est pourquoi en pratique, on utilise un arbre tournoi.

L'idée est que si on, par exemple, 2^n équipes, alors on fait une compétition entre les 2^{n-1} premières, une autre entre les 2^{n-1} dernières, et on fait les gagnants de chaque compétition s'affronter à la fin.

I. Diviser pour régner

1) Un paradigme stratégique.

Nous pouvons nous inspirer de notre situation réelle en introduction pour dégager un paradigme récursif, utilisable en algorithmique. Soit $P(n)$ un problème de type P et de taille n :

1) On divise $P(n)$ en sous-problèmes de Type P mais de taille inférieure.
2) Régner : on résoud récursivement les sous-problèmes avec la même approche, jusqu'à ce que la taille du problème permette de le résoudre directement.
3) Combiner : On combine les solutions des sous-problèmes de $P(n)$ pour le résoudre.

2) L'intérêt d'une telle stratégie

Lors de la conception d'un algorithme, il survient aussitôt la question de la complexité.

Un algorithme qui utilise le paradigme "diviser pour régner", vérifiera généralement une relation de récurrence de la forme :

$$T(n) = a\, T\left(\frac{n}{b}\right) + f(n)$$

où T est la complexité de la résolution de $P(n)$.

Théorème général Cormen p. 86

Soient $a \geq 1$ et $b > 1$ deux constantes, soit $f(n)$ une fonction et soit $T(n)$ définie pour les entiers positifs par la récurrence :

$$T(n) = a\,T(n/b) + f(n)$$

où n/b désigne $\lfloor \frac{n}{b} \rfloor$ ou $\lceil \frac{n}{b} \rceil$. Alors $T(n)$ peut-être borné asymptotiquement de la façon suivante.

1) Si $f(n) = O(n^{\log_b(a) - \varepsilon})$ pour un certain $\varepsilon > 0$, alors
$$T(n) = O(n^{\log_b(a)})$$

2) Si $f(n) = \Theta(n^{\log_b(a)})$ alors $T(n) = (n^{\log_b(a)} \lg(n))$

3) Si $f(n) = \Omega(n^{\log_b(a)+\varepsilon})$, pour un certain $\varepsilon > 0$, et si $a\,f(n/b) \leq c\,f(n)$ pour un certain $c > 1$ et pour n suffisamment grand, alors $T(n) = \Theta(f(n))$

Ex : Pour un algorithme vérifiant : $T(n) = 3\,T(\frac{n}{2}) + O(n)$ on obtient une complexité $T(n) = O(n^{1.59})$

L'avantage du paradigme "diviser pour régner" est que l'on sait bien classifier les complexités qui en ressortent.

II Exemples types : Les tris

1. Tri-fusion $T[1,\dots,n]$ =
2. $(T_1[1,\dots,\frac{n}{2}], T_2[1,\dots,\frac{n}{2}])$ = divise $T[1,\dots,n]$
3 ~~Tree~~ fusion $T_1[1,\dots,\frac{n}{2}]$
4. Tri-fusion $T_2[1,\dots,\frac{n}{2}]$
5. $T[1,\dots,n] \leftarrow$ Fusion$(T_1[1,\dots,\frac{n}{2}], T_2[1,\dots,\frac{n}{2}])$

Le tri fusion utilise le paradigme diviser pour régner :
- ligne 2 : diviser
- ligne 3 et 4 : régner
- ligne 5 : combiner.

```
Tri-rapide (A,p,q) =
  si p<q faire
     r = partitionner (A,p,q)
     Tri-rapide (A,p,r)
     Tri-rapide (A,r+1,q)
```

Il y a plusieurs choix possibles pour la fonction partitionner, mais l'idée est qu'on effectue un plus gros travail au niveau de la partition, pour que la fusion soit triviale.

Les deux exemples de tri on une complexité moyenne en $O(n \ln(n))$, Ceci "montre" que la complexité globale dépend de la somme du coût des étapes diviser et combiner. Ce coût est représenté par la fonction $f(n)$.

III - Autres Exemples

1) Multiplication des entiers bit-à-bit.

Papa p46

Soit x, y deux nombres binaires à n-bits. On peut multiplier ces deux entiers grâce à un algorithme récursif :
$$\begin{cases} x = 2^{\lceil \frac{n}{2} \rceil} x_G + x_D \\ y = 2^{\lceil \frac{n}{2} \rceil} y_G + y_D \end{cases}$$

$$\longrightarrow x\,y = 2^n x_G y_G + 2^{\lceil \frac{n}{2} \rceil}(x_G y_D + x_D y_G) + x_D y_D$$

Nous avons donc besoin de 4 multiplications. Comme les addition se font en temps linéaire, nous avons donc une relat° de récurrence de la forme :
$$T(n) = 4\,T\left(\frac{n}{2}\right) + O(n)$$
ce qui donne une complexité de la forme
$$T(n) = O(n^2).$$
Mais on peut faire plus malin en remarquant que :
$$bc + ad = (a+b)(c+d) - ac - bc$$

c'est-à-dire :
$$x\,y = 2^n x_G y_G + 2^{\lceil \frac{n}{2} \rceil}\left[(x_G + x_D)(y_G + y_D) - x_G y_G - x_D y_D\right] + x_D y_D$$
cet algorithme ne demande que 3 multiplication.
on a donc : $T(n) = 3\,T\left(\frac{n}{2}\right) + O(n)$.
ce qui donne $T(n) = O\left(n^{\log_2(3)}\right) = O\left(n^{1{,}59}\right)$.

n log n ac Fourier Papa p 48

Dev 1 : Recherche de 2 points les + proches dans un ens de n points

2) Algorithme de Strassen pour la multiplicat° des matrices

Cor p69

Pour multiplier deux matrices carrées de taille n, A et B, on peut effectuer un calcul récursif en les divisant en 4 blocs :
$$A\,B = \left(\begin{array}{c|c} A_1 & A_2 \\ \hline A_3 & A_4 \end{array}\right)\left(\begin{array}{c|c} B_1 & B_2 \\ \hline B_3 & B_4 \end{array}\right) = C$$
puis on doit calculer
$$C_1 = A_1 B_1 + A_2 B_3$$
$$C_2 = A_1 B_2 + A_2 B_4$$
$$C_3 = A_3 B_1 + A_4 B_3$$
$$C_4 = A_3 B_2 + A_4 B_4$$
Ce qui nous donne 8 multiplication à chaque étape. Mais l'algorithme de Strassen n'utilise que 7 multiplication à chaque étape en posant des matrices intermédiaires comme dans la multiplication bit-à-bit.
On a donc : $T(n) = 7\,T\left(\frac{n}{2}\right) + O(n)$
d'où $T(n) = O\left(n^{2{,}81}\right)$
au lieu de $O(n^3)$

Dev 2 : Transformée de Fourier Rapide.

Ref : Cormen et Papadimitriou.

Cormen
956

Recherche des 2 points les plus proches dans le plan

<u>Données:</u> Q, ensemble de n points. $(n \geq 2)$

<u>Objectif:</u> Trouver le couple de points de Q minimisant la distance euclidienne.

Algorithme naïf: Recherche d'un min sur toutes les paires de points: $O(n^2)$.

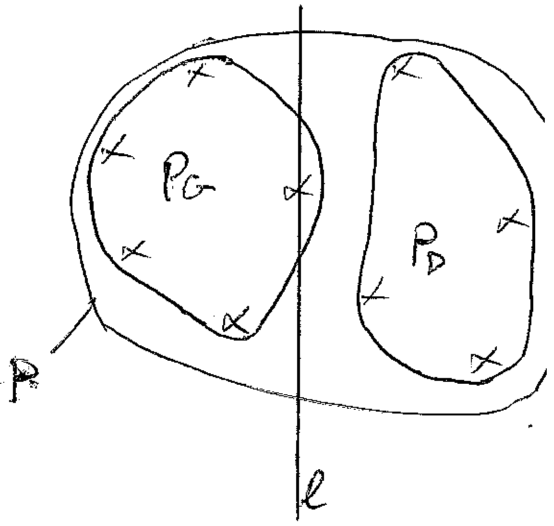
→ Amélioration: diviser pour régner

L'algorithme récursif reçoit en entrée $P \subseteq Q$ et deux tableaux X et Y, chacun contient les points de P triés par ordre d'abscisses (resp. d'ordonnées).

<u>Cas de base:</u> $|P| = 2$ ou 3, auquel cas on calcule directement le résultat

<u>Diviser:</u> On cherche une droite l vertical qui coupe P en deux parties "égales", P_G et P_D, $|P_G| = \left\lceil \frac{|P|}{2} \right\rceil$, $|P_D| = \left\lfloor \frac{|P|}{2} \right\rfloor$, et on divise alors X en X_G et X_D, et Y en Y_G et Y_D (X_G et X_D triés par abscisses, Y_G et Y_D par ordonnées)

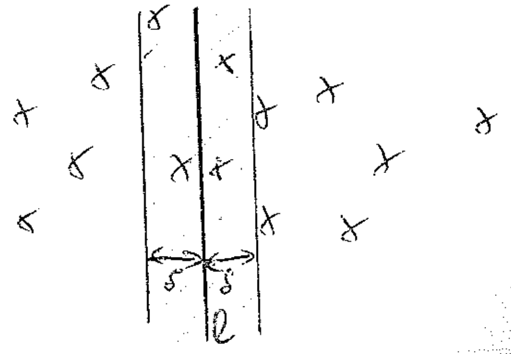
→ Temps linéaire



<u>Régner:</u>
On appelle l'algorithme récursif sur (P_G, X_G, Y_G) et (P_D, X_D, Y_D), obtenant ainsi la distance minimale à gauche δ_G, et à droite δ_D

<u>Combiner:</u>
On pose $\delta = \min(\delta_D, \delta_G)$.

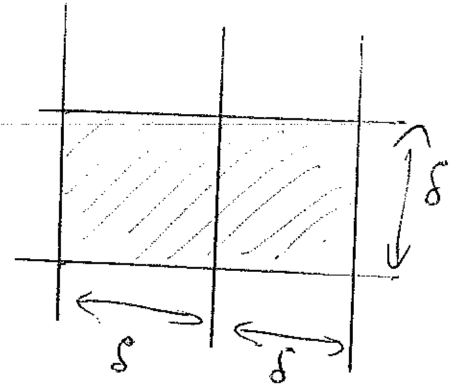
Les deux points les plus proches peuvent vérifier deux configurations différentes:
* Ils sont tous deux dans P_G ou P_D
* Il y en a un de chaque coté. Dans ce cas, on les note p_G et p_D.
puisque $d(p_G, p_D) \leq \delta$, ils sont situé dans un ruban de largeur 2δ, centré en l



On calcule alors Y', tableau correspondant à Y privé des points hors du ruban.

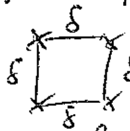
Comme $d(p_G, p_D) \le \delta$, p_G et p_D sont verticalement distants d'au plus δ.

p_G et p_D sont donc contenus dans un rectangle de côté $\delta \times 2\delta$



Lemme : Un tel rectangle ne peut contenir plus de 8 points.

En effet, chacun des deux carrés (gauche et droite) ne peut contenir que 4 points par minimalité de δ. (C'est la configuration



), et donc le rectangle n'en contient pas plus de 8.

Ainsi, pour un point donné x du ruban, seuls ses 7 prédécesseurs et ses 7 successeurs dans Y' peuvent être candidats pour vérifier $d(x,y) < \delta$.

Cela implique que pour chaque point de Y', on peut vérifier en temps constant si il appartient à une paire minimale de P

Rassemblons les morceaux :

Recherche_2points(Q) =

- Calculer X_Q tableau trié par abscisses
- Calculer Y_Q ———— ordonnées.
- Retourner Recherche_rec(Q, X_Q, Y_Q).

Recherche_rec(P, X, Y)

Si $|P| = 2$ retourner $(X(1), X(2))$

Si $|P| = 3$ retourner (q_1, q_2) minimisant la distance

— Cas de base en $\Theta(1)$.

Diviser P en P_G et P_D par la droite $x = l$.

Calculer X_G, Y_G, X_D, Y_D

— Division en $\Theta(n)$.

$(p_{G1}, p_{G2}, \delta_G) \leftarrow$ Recherche_rec(P_G, X_G, Y_G)

$(p_{D1}, p_{D2}, \delta_D) \leftarrow$ Recherche_rec(P_D, X_D, Y_D).

— Règne.

$\delta := \min(\delta_1, \delta_2)$.

Si $\delta = \delta_G$, $q_1 := p_{G1}$ et $q_2 := p_{G2}$

   sinon $q_1 := p_{D1}$ et $q_2 := p_{D2}$

Calculer Y' (éléments de Y d'abscisse $\in [l - \delta, l + \delta]$)

Pour $i = 1$ à $|Y'| - 1$ faire.

   Pour $j = 1$ à $\min(7, |Y'| - i)$ faire.

      si $d(Y'[i], Y'[j]) < \delta$ alors

         $\delta \leftarrow d(Y'[i], Y'[j])$.

         $q_1 \leftarrow Y'[i]$

         $q_2 \leftarrow Y'[j]$

— Combinaison en $\Theta(n)$

Retourner (q_1, q_2, δ)

Complexité :

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n).$$

$$\rightarrow T(n) = \Theta(n \ln n)$$

Algorithmique

Transformée de Fourier rapide (FFT)

But : Calculer le produit de deux polynomes A et B : $C = AB$

- → Peut se faire en $O(n^2)$ par calcul direct des coefficients.
- → La FFT permet une amélioration en $O(n \ln n)$.

Equivalence des représentations :

Un polynome de degré d est caractérisé de manière unique par ses valeurs en $d+1$ points distincts.

Un polynome A peut donc être représenté soit :

- Par ses coefficients $(a_0, \dots, a_d)$.
- Par les valeurs $(A(x_0), A(x_1), \dots, A(x_d))$. (éventuellement plus de points).

→ Dans la représentation par valeurs, le produit de deux polynomes se fait en temps linéaire.

Dans la suite, on supposera que l'on cherche à calculer le produit de A et B de même degré d (non restrictif quitte à autoriser des 0 commes coefficients de plus haut degré).

La FFT suit alors le schéma suivant :

* Choisir des points $x_0, \dots, x_{n-1}$, avec $n \geq 2d+1$
* Evaluer A et B en tous ces points → $A(x_0), \dots, A(x_{n-1}), B(x_0), \dots B(x_{n-1})$
* Calculer pour tout k $C(x_k) = A(x_k)B(x_k)$
* Interpoler $C = A \cdot B$ à partir des $C(x_k)$

Choix des points :

On prend n la plus petite puissance de 2 supérieure à $2d+1$, on pose $\omega = e^{\frac{2i\pi}{n}}$, et $x_k = \omega^k$ pour $k \in [\![0; n-1]\!]$.

$\{x_k \mid k\}$ est donc l'ensemble des racines n-ièmes de l'unité.

Evaluation par diviser pour régner :

l'évaluation peut se voir comme le calcul de

$$\begin{pmatrix} A(x_0) \\ \vdots \\ A(x_{n-1}) \end{pmatrix} = \begin{pmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ \vdots & \vdots & \vdots & & \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{pmatrix} \cdot \begin{pmatrix} a_0 \\ \vdots \\ a_{n-1} \end{pmatrix}$$

Le calcul naïf est en $\mathcal{O}(n^2)$ multiplications

→ On peut faire mieux.

A se décompose en parties paires et impaires comme suit :

$$\boxed{A(x) = A_p(x^2) + x A_i(x^2)}$$

et notament, cela simplifie l'évaluation en des points opposés :

$$A(x) = A_p(x^2) + x A_i(x^2)$$
$$A(-x) = A_p(x^2) - x A_i(x^2).$$

Ainsi si l'on connait les valeurs de A_p et A_i en $(w_i^k)^2$, on a en temps constant $A(w^k)$ et $A(-w^k)$.

Le problème évaluer A de degré d sur $\{w^k \mid k \in [\![0; n-1]\!]\}$ se réduit à : évaluer $\left| \begin{array}{l} A_p \text{ de degré } \left\lfloor \frac{d}{2} \right\rfloor \text{ sur } \{w^{2k} \mid k \in [\![0; \frac{n}{2}-1]\!]\}, \\ A_i \text{ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \end{array} \right.$

ce que l'on fait de la même méthode jusqu'au cas de base, celui de de l'évaluation d'un polynome de degré 0 et $w = 1$.

```
FFT(A, n)
  si |n| = 1, retourner A(1)
  sinon  w := e^(2iπ/n + α) ; y := créer tableau de taille n
      Ap := (a0, a2, ...)                          ] Diviser en O(n)
      Ai := (a1, a3, ...)
      yp := FFT(Ap, n/2)                           ] Régner
      yi := FFT(Ai, n/2)
      Pour k = 0 à (n/2 - 1) faire                 ] Combiner en O(n)
          y[k] ← yp[k] + w^k × yi[k]
          y[k + n/2] ← yp[k] − w^k × yi[k]
      retourner y
```

La complexité de l'algorithme est donnée par :

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$$

Par théorème sur les récurrences, $T(n) = \Theta(n \ln n)$

Multiplication :

→ Multiplication des évaluations point par point en $\Theta(n)$ par une procédure Produit.

Interpolation :

On sait que

$$\begin{pmatrix} C(x_0) \\ \vdots \\ \vdots \\ C(x_{n-1}) \end{pmatrix} = \underbrace{\begin{pmatrix} 1 & w & w^2 & \cdots & w^{n-1} \\ 1 & w^2 & w^4 & \cdots & w^{2(n-1)} \\ \vdots & & & & \\ \vdots & & & & \\ 1 & w^{(n-1)} & \cdots & \cdots & w^{(n-1)(n-1)} \end{pmatrix}}_{M_n(w)} \times \begin{pmatrix} c_0 \\ \vdots \\ \vdots \\ c_{n-1} \end{pmatrix}$$

On reconnait ici une matrice de Vandermonde, qui est donc inversible, et on remarque que $M_n(w)^{-1} = \frac{1}{n} M_n(w^{-1})$

Comme w^{-1} est également une racine de l'unité, on peut appliquer $FFT([C(x_0), \ldots, C(x_{n-1})], n)$ et diviser les coefficients par n pour trouver C.

L'interpolation se fait aussi en $\Theta(n \ln n)$.

Au final : On calcule $n = 2^k \geq 2d+1$.

$C = \frac{1}{n} FFT[\text{Produit}(FFT(A,n), FFT(B,n)), n]$, et le calcul s'effectue en $\Theta(n \ln n)$