Diviser pour régner

Jean-Pierre Becirspahic
Lycée Louis-Le-Grand

Exponentiation rapide

$(X, *)$ est un monoïde multiplicatif, $x \in X$ et $n \in \mathbb{N}$.
But : calculer x^n avec le moins de multiplications possibles.

Exponentiation rapide

$(X, *)$ est un monoïde multiplicatif, $x \in X$ et $n \in \mathbb{N}$.
But : calculer x^n avec le moins de multiplications possibles.

Algorithme naïf :

```
let puissance x = function
  | 0 -> 1
  | 1 -> x
  | n -> x * puissance x (n-1) ;;
```

Cette méthode nécessite $n - 1$ multiplications.

Exponentiation rapide

$(X, *)$ est un monoïde multiplicatif, $x \in X$ et $n \in \mathbb{N}$.
But : calculer x^n avec le moins de multiplications possibles.

Méthode binaire : les formules $x^{2k} = (x^2)^k$ et $x^{2k+1} = x(x^2)^k$ conduisent à l'algorithme :

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

Exponentiation rapide

$(X, *)$ est un monoïde multiplicatif, $x \in X$ et $n \in \mathbb{N}$.
But : calculer x^n avec le moins de multiplications possibles.

Méthode binaire : les formules $x^{2k} = (x^2)^k$ et $x^{2k+1} = x(x^2)^k$ conduisent à l'algorithme :

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

Si $n = [b_p, b_{p-1}, \ldots, b_0]_2$, alors $c_n = c_k + 1 + b_0$ avec $k = [b_p, b_{p-1}, \ldots, b_1]_2$

donc $c_n = p + \sum_{k=0}^{p-1} b_k$, et : $p \leqslant c_n \leqslant 2p \iff \lfloor \log n \rfloor \leqslant c_n \leqslant 2\lfloor \log n \rfloor$.

Exponentiation rapide

$(X,*)$ est un monoïde multiplicatif, $x \in X$ et $n \in \mathbb{N}$.
But : calculer x^n avec le moins de multiplications possibles.

Méthode binaire : les formules $x^{2k} = (x^2)^k$ et $x^{2k+1} = x(x^2)^k$ conduisent à l'algorithme :

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

Si $n = [b_p, b_{p-1}, \ldots, b_0]_2$, alors $c_n = c_k + 1 + b_0$ avec $k = [b_p, b_{p-1}, \ldots, b_1]_2$

donc $c_n = p + \sum_{k=0}^{p-1} b_k$, et : $p \leqslant c_n \leqslant 2p \iff \lfloor \log n \rfloor \leqslant c_n \leqslant 2\lfloor \log n \rfloor$.

$c_n = \lfloor \log n \rfloor$ lorsque $n = 2^p$; $c_n = 2\lfloor \log n \rfloor$ lorsque $n = 2^{p+1}$.

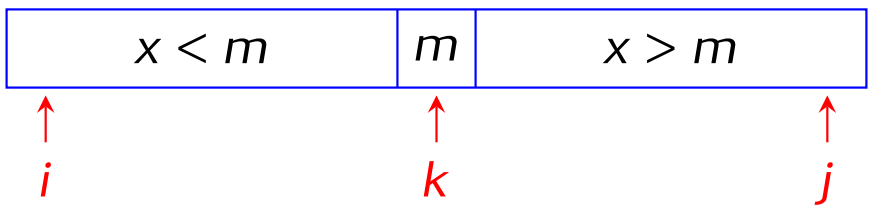
Recherche dichotomique

Diviser pour régner : 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 $n' < n$ (avec en général $n' \approx n/2$).

Recherche dichotomique

Diviser pour régner : 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 $n' < n$ (avec en général $n' \approx n/2$).

Exemple : recherche dichotomique dans un tableau trié.

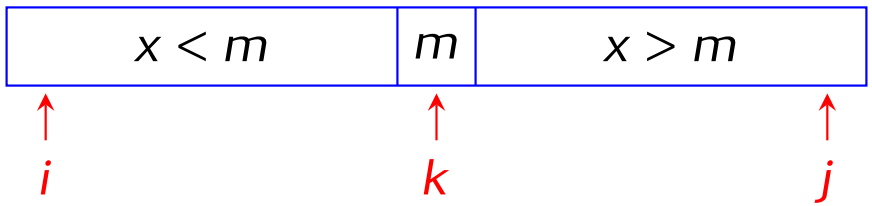



```
let recherche_dicho x t =
  let rec f i j =
        if j < i then false
          else match (i + j) / 2 with
            | k when x = t.(k) -> true
            | k when x < t.(k) -> f i (k-1)
            | k                -> f (k+1) j
  in f 0 (vect_length t - 1) ;;
```

Recherche dichotomique

Diviser pour régner : 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 $n' < n$ (avec en général $n' \approx n/2$).

Exemple : recherche dichotomique dans un tableau trié.




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

c_n : nombre de comparaisons dans le pire des cas. Alors $c_0 = 0$ et $c_n = 2 + c_{\lfloor n/2 \rfloor}$ pour $n \geqslant 1$, ce qui conduit à $c_n = 2\lfloor \log n \rfloor + 2 = \Theta(\log n)$.

Le théorème maître

En général, le coût est solution de : $\boxed{c_n = ac_{\lfloor n/2 \rfloor} + bc_{\lceil n/2 \rceil} + d_n.}$

Le théorème maître

En général, le coût est solution de : $\boxed{c_n = ac_{\lfloor n/2 \rfloor} + bc_{\lceil n/2 \rceil} + d_n.}$

- Si $n = 2^p$, $u_p = c_{2^p}$ vérifie : $u_p = (a+b)u_{p-1} + d_{2^p}$, soit par télescopage : $u_p = (a+b)^p \left(u_0 + \sum_{j=1}^{p} \frac{d_{2^j}}{(a+b)^j} \right)$.

Le théorème maître

En général, le coût est solution de : $\boxed{c_n = ac_{\lfloor n/2 \rfloor} + bc_{\lceil n/2 \rceil} + d_n.}$

- Si $n = 2^p$, $u_p = c_{2^p}$ vérifie : $u_p = (a+b)u_{p-1} + d_{2^p}$, soit par télescopage : $u_p = (a+b)^p \left(u_0 + \sum_{j=1}^{p} \frac{d_{2^j}}{(a+b)^j} \right)$.

  Lorsque $d_n = \lambda n^k$, $u_p = \begin{cases} \alpha 2^{kp} + \beta(a+b)^p & \text{si } a+b \neq 2^k \\ (u_0 + \lambda p)(a+b)^p & \text{si } a+b = 2^k \end{cases}$ et :

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

Le théorème maître

En général, le coût est solution de : $\boxed{c_n = ac_{\lfloor n/2 \rfloor} + bc_{\lceil n/2 \rceil} + d_n.}$

- Si $n = 2^p$, $u_p = c_{2^p}$ vérifie : $u_p = (a+b)u_{p-1} + d_{2^p}$, soit par télescopage : $u_p = (a+b)^p \left(u_0 + \sum_{j=1}^{p} \frac{d_{2^j}}{(a+b)^j} \right)$.

  Lorsque $d_n = \lambda n^k$, $u_p = \begin{cases} \alpha 2^{kp} + \beta(a+b)^p & \text{si } a+b \neq 2^k \\ (u_0 + \lambda p)(a+b)^p & \text{si } a+b = 2^k \end{cases}$ et :

  - 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 n quelconque, si (d_n) est croissante, alors (c_n) aussi et :

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

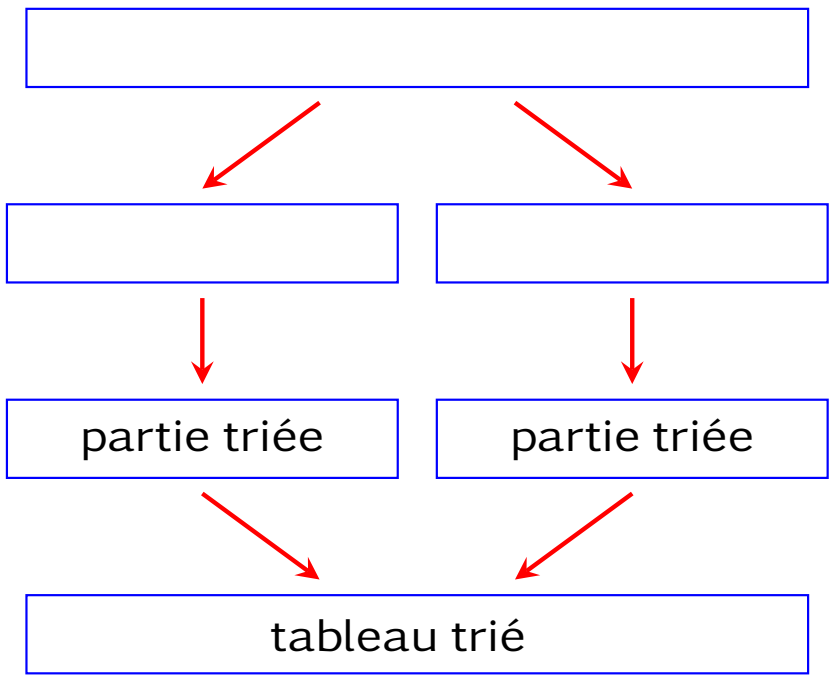
Tri fusion

But : trier des éléments en minimisant le nombre de comparaisons.
On adopte une stratégie « diviser pour régner » :

– scission du tableau en deux parties :

– tri des deux moitiés du tableau :

– fusion des parties triées :



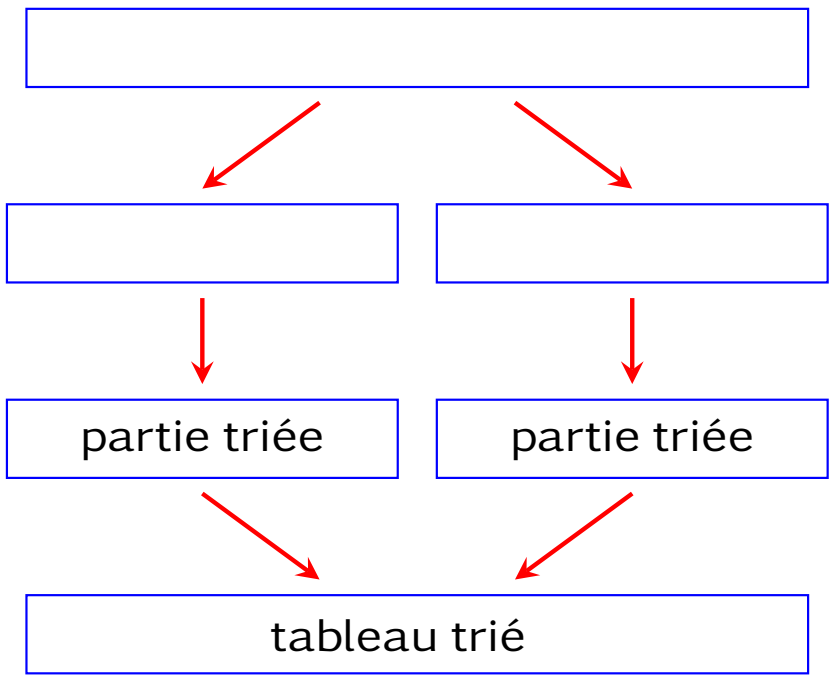
Tri fusion

But : trier des éléments en minimisant le nombre de comparaisons.
On adopte une stratégie « diviser pour régner » :

– scission du tableau en deux parties :

– tri des deux moitiés du tableau :

– fusion des parties triées :



Si la scission et la fusion ont un coût linéaire, $c_n = c_{\lfloor n/2 \rfloor} + c_{\lceil n/2 \rceil} + \Theta(n)$ et alors $c_n = \Theta(n \log n)$ (théorème maître).

Tri fusion

On choisit d'implémenter cet algorithme à l'aide des listes :
Scission d'une liste en deux sous-listes :

```
let rec scission = function
  | []      -> [], []
  | [a]     -> [a], []
  | a::b::q -> let (l1, l2) = scission q in a::l1, b::l2 ;;
```

Fusion de deux listes triées :

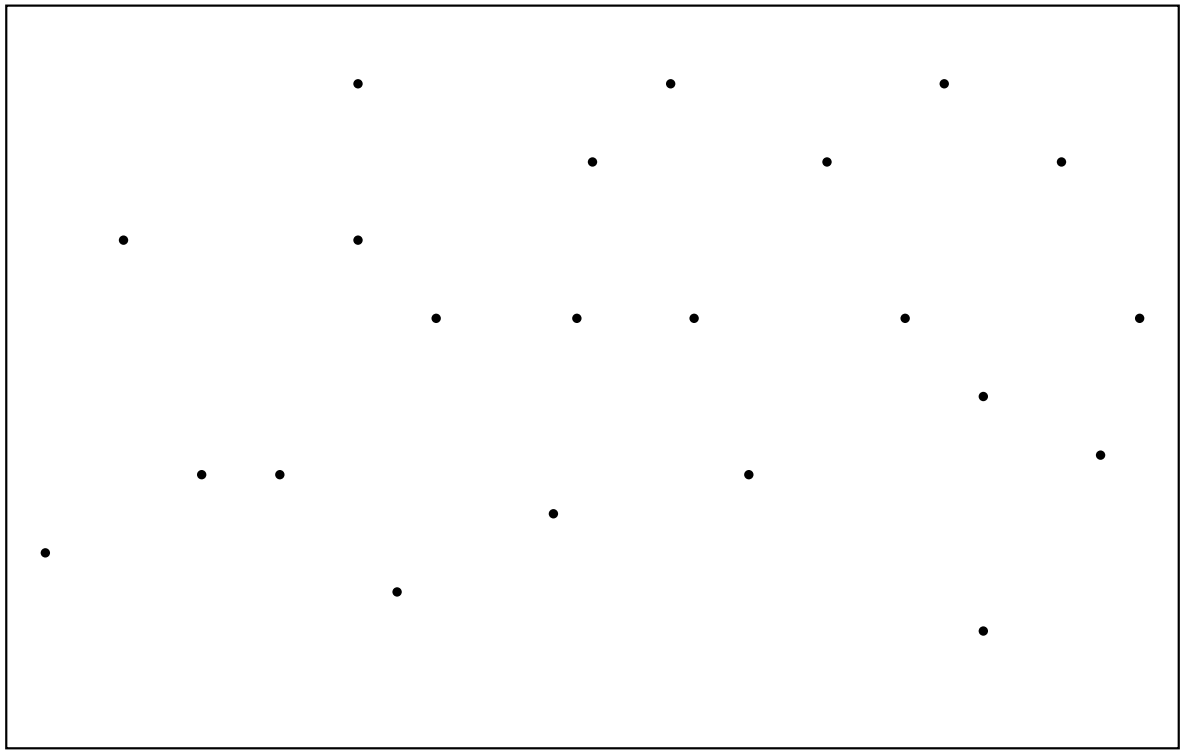
```
let rec fusion = fun
  | [] l2                          -> l2
  | l1 []                          -> l1
  | (t1::q1) l2 when t1 < hd l2 -> t1::(fusion q1 l2)
  | l1 (t2::q2)                    -> t2::(fusion l1 q2) ;;
```

Algorithme de tri fusion :

```
let rec merge_sort = function
  | []  -> []
  | [a] -> [a]
  | l   -> let (l1, l2) = scission l in
           fusion (merge_sort l1) (merge_sort l2) ;;
```

Distance minimale

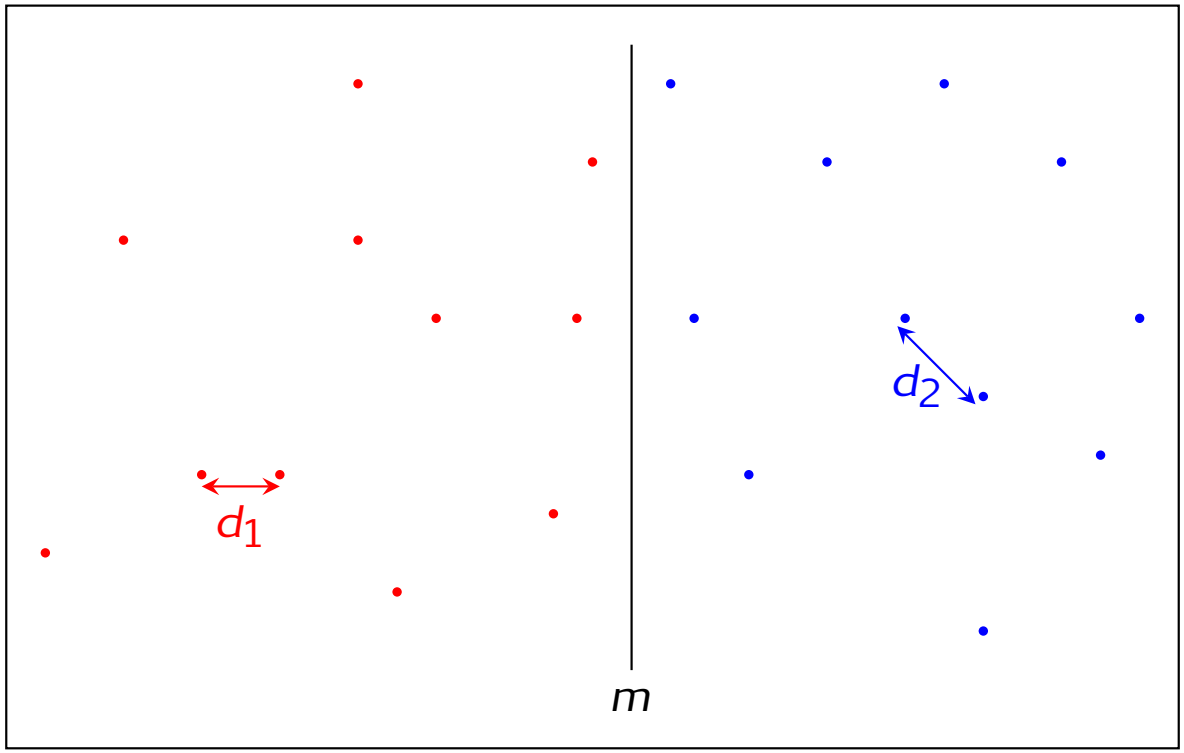
But : trouver les deux points les plus proches dans un nuage de points.



Algorithme naïf : calculer la distance entre les $\binom{n}{2} \sim \frac{n^2}{2}$ paires possibles.

Distance minimale

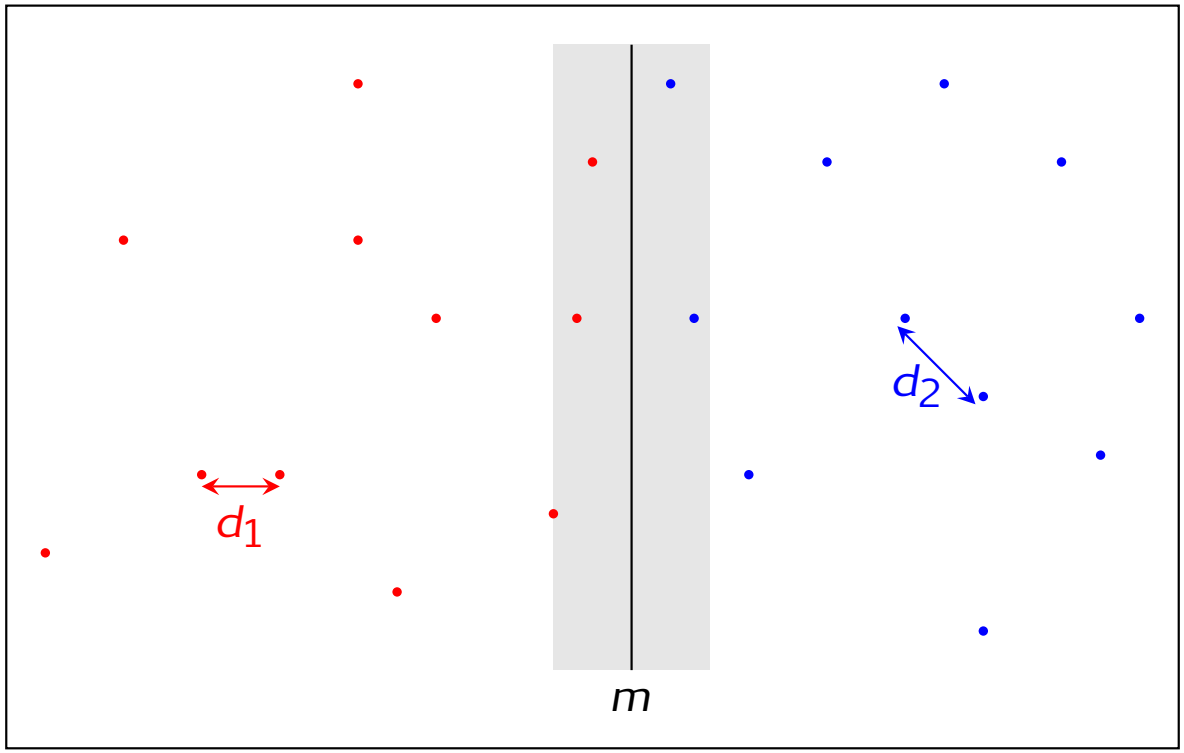
But : trouver les deux points les plus proches dans un nuage de points.



Diviser pour régner : on sépare le nuage de points $\mathcal{P}$ en deux parties triées par abscisses croissantes $\mathcal{P}_1$ et $\mathcal{P}_2$ et on calcule par appel récursif les distances minimales d_1 et d_2.

Distance minimale

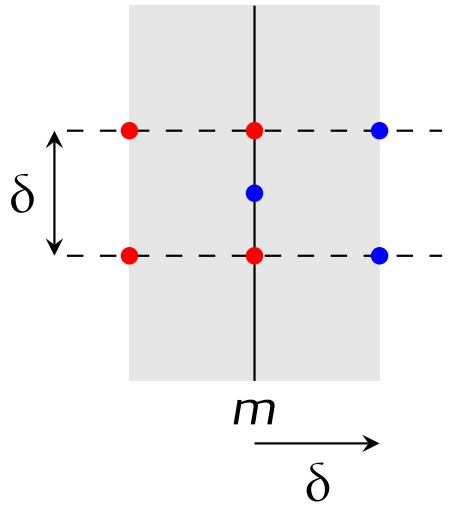
But : trouver les deux points les plus proches dans un nuage de points.



On calcule $\delta = \min(d_1, d_2)$ et on restreint la recherche à la bande verticale délimitée par les abscisses $m - \delta$ et $m + \delta$.

Distance minimale

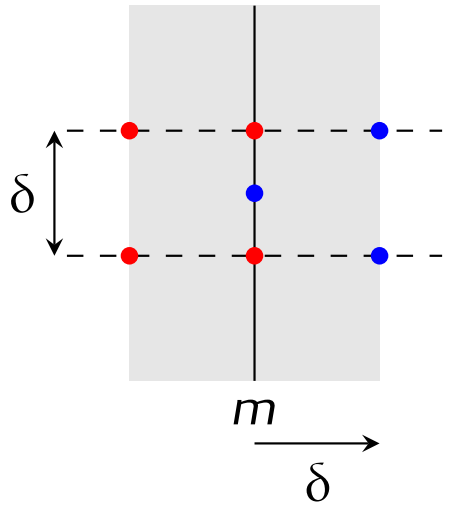
But : trouver les deux points les plus proches dans un nuage de points.



Dans chaque tranche de hauteur δ de cette bande ne peuvent se trouver qu'*au plus* sept points. Ceci permet pour chaque point de cette bande de ne calculer leur distance qu'aux six points suivants par ordre croissant d'ordonnée $\longrightarrow \Theta(n)$.

Distance minimale

But : trouver les deux points les plus proches dans un nuage de points.



Dans chaque tranche de hauteur δ de cette bande ne peuvent se trouver qu'*au plus* sept points. Ceci permet pour chaque point de cette bande de ne calculer leur distance qu'aux six points suivants par ordre croissant d'ordonnée $\longrightarrow \Theta(n)$.

Le coût de cet algorithme vérifie une relation de la forme :
$c_n = c_{\lfloor n/2 \rfloor} + c_{\lceil n/2 \rceil} + \Theta(n)$, donc $c_n = \Theta(n \log n)$.

Multiplication de deux polynômes

- Addition de deux polynômes de même degré :

$$\sum_{k=0}^{n} a_k X^k + \sum_{k=0}^{n} b_k X^k = \sum_{k=0}^{n} (a_k + b_k) X^k$$

  Utilise $n+1$ additions, le coût est linéaire.

- Produit de deux polynômes de même degré :

$$\left(\sum_{i=0}^{n} a_i X^i\right) \times \left(\sum_{j=0}^{n} b_j X^j\right) = \sum_{i=0}^{n} \sum_{j=0}^{n} a_i b_j X^{i+j}$$

  Utilise $(n+1)^2$ additions et $(n+1)^2$ multiplications, le coût est quadratique.

Multiplication de deux polynômes

- Addition de deux polynômes de même degré :

$$\sum_{k=0}^{n} a_k X^k + \sum_{k=0}^{n} b_k X^k = \sum_{k=0}^{n} (a_k + b_k) X^k$$

  Utilise $n+1$ additions, le coût est linéaire.

- Produit de deux polynômes de même degré :

$$\left(\sum_{i=0}^{n} a_i X^i\right) \times \left(\sum_{j=0}^{n} b_j X^j\right) = \sum_{i=0}^{n} \sum_{j=0}^{n} a_i b_j X^{i+j}$$

  Utilise $(n+1)^2$ additions et $(n+1)^2$ multiplications, le coût est quadratique.

On cherche à minimiser le nombre de multiplications, quitte à augmenter le nombre d'additions.

Multiplication de deux polynômes

Algorithme de Karatsuba

On pose $m = \left\lceil \frac{n}{2} \right\rceil$ puis $P = X^m P_1 + P_2$ et $Q = X^m Q_1 + Q_2$.

P_1, P_2, Q_1, Q_2 sont de degrés inférieurs ou égaux à $\left\lfloor \frac{n}{2} \right\rfloor$.

On note c_n le nombre de multiplications utilisées pour le calcul de PQ.

Multiplication de deux polynômes

Algorithme de Karatsuba

On pose $m = \left\lceil \frac{n}{2} \right\rceil$ puis $P = X^m P_1 + P_2$ et $Q = X^m Q_1 + Q_2$.

P_1, P_2, Q_1, Q_2 sont de degrés inférieurs ou égaux à $\left\lfloor \frac{n}{2} \right\rfloor$.

On note c_n le nombre de multiplications utilisées pour le calcul de PQ.

- Si $PQ = X^{2m} P_1 Q_1 + X^m (P_1 Q_2 + P_2 Q_1) + P_2 Q_2$, alors

  $c_n = 4c_{\lfloor n/2 \rfloor} + \Theta(n)$, ce qui conduit à $c_n = \Theta(n^2)$.

Multiplication de deux polynômes

Algorithme de Karatsuba

On pose $m = \left\lceil \frac{n}{2} \right\rceil$ puis $P = X^m P_1 + P_2$ et $Q = X^m Q_1 + Q_2$.

P_1, P_2, Q_1, Q_2 sont de degrés inférieurs ou égaux à $\left\lfloor \frac{n}{2} \right\rfloor$.

On note c_n le nombre de multiplications utilisées pour le calcul de PQ.

- Si $PQ = X^{2m} P_1 Q_1 + X^m (P_1 Q_2 + P_2 Q_1) + P_2 Q_2$, alors

  $c_n = 4c_{\lfloor n/2 \rfloor} + \Theta(n)$, ce qui conduit à $c_n = \Theta(n^2)$.

- Si $PQ = X^{2m} R_1 + X^m (R_2 - R_1 - R_3) + R_3$ avec :

  $R_1 = P_1 Q_1$, $R_2 = (P_1 + P_2)(Q_1 + Q_2)$ et $R_3 = P_2 Q_2$,

  alors $c_n = 3c_{\lfloor n/2 \rfloor} + \Theta(n)$, ce qui conduit à $c_n = \Theta(n^{\log 3})$.

Multiplication de deux entiers

Si $x = \sum_{i=0}^{n-1} x_i 2^i$ et $y = \sum_{i=0}^{n-1} y_i 2^i$, $s = x + y$ est calculé à l'aide des relations :

- $r_0 = 0$;
- $\forall i \in [\![0, n-1]\!], \quad s_i = (r_i + x_i + y_i) \bmod 2$;
- $\forall i \in [\![0, n-1]\!], \quad r_{i+1} = (r_i + x_i + y_i)/2$;
- $s_n = r_n$.

où r_i est la retenue à la position i. Le coût d'une addition est un $\Theta(n)$.

Multiplication de deux entiers

Si $x = \sum_{i=0}^{n-1} x_i 2^i$ et $y = \sum_{i=0}^{n-1} y_i 2^i$, $s = x + y$ est calculé à l'aide des relations :

- $r_0 = 0$;
- $\forall i \in [\![0, n-1]\!], \quad s_i = (r_i + x_i + y_i) \bmod 2$;
- $\forall i \in [\![0, n-1]\!], \quad r_{i+1} = (r_i + x_i + y_i)/2$;
- $s_n = r_n$.

où r_i est la retenue à la position i. Le coût d'une addition est un $\Theta(n)$.

Multiplication par l'algorithme naïf : $xy = \sum_{i=0}^{n-1} (x2^i) y_i$, qui se ramène à n additions d'entiers de longueur $2n$. Ceci donne un coût en $\Theta(n^2)$.

Algorithme de multiplication rapide : on pose $x = a2^m + b$ et $y = c2^m + d$ avec $m = \lceil n/2 \rceil$, et : $xy = ac2^{2m} + \Big((a+b)(c+d) - ac - bd\Big)2^m + bd$.

Le coût est un $\Theta\left(n^{\log 3}\right)$.

Multiplication de deux matrices

Si M et N sont deux matrices $n \times n$, le coût d'une addition est un $\Theta(n^2)$ et la coût d'une multiplication par l'algorithme naïf un $\Theta(n^3)$.

Multiplication de deux matrices

Si M et N sont deux matrices $n \times n$, le coût d'une addition est un $\Theta(n^2)$ et la coût d'une multiplication par l'algorithme naïf un $\Theta(n^3)$.

Si $n = 2m$, on pose $M = \begin{pmatrix} A_1 & B_1 \\ C_1 & D_1 \end{pmatrix}$ et $N = \begin{pmatrix} A_2 & B_2 \\ C_2 & D_2 \end{pmatrix}$.

Alors $MN = \begin{pmatrix} A_1A_2 + B_1C_2 & A_1B_2 + B_1D_2 \\ C_1A_2 + D_1C_2 & C_1B_2 + D_1D_2 \end{pmatrix}$.

Ceci conduit à : $c_n = 8c_{\lceil n/2 \rceil} + \Theta(n^2)$, soit $c_n = \Theta(n^{\log 8}) = \Theta(n^3)$.

Multiplication de deux matrices

Si M et N sont deux matrices $n \times n$, le coût d'une addition est un $\Theta(n^2)$ et la coût d'une multiplication par l'algorithme naïf un $\Theta(n^3)$.

Si $n = 2m$, on pose $M = \begin{pmatrix} A_1 & B_1 \\ C_1 & D_1 \end{pmatrix}$ et $N = \begin{pmatrix} A_2 & B_2 \\ C_2 & D_2 \end{pmatrix}$.

Alors $MN = \begin{pmatrix} A_1A_2 + B_1C_2 & A_1B_2 + B_1D_2 \\ C_1A_2 + D_1C_2 & C_1B_2 + D_1D_2 \end{pmatrix}$.

Ceci conduit à : $c_n = 8c_{\lceil n/2 \rceil} + \Theta(n^2)$, soit $c_n = \Theta(n^{\log 8}) = \Theta(n^3)$.

Formules de Strassen :

$$M_1 = (B_1 - D_1)(C_2 + D_2) \qquad M_5 = A_1(B_2 - D_2) \qquad X = M_1 + M_2 - M_4 + M_6$$
$$M_2 = (A_1 + D_1)(A_2 + D_2) \qquad M_6 = D_1(C_2 - A_2) \qquad Y = M_4 + M_5$$
$$M_3 = (A_1 - C_1)(A_2 + B_2) \qquad M_7 = (C_1 + D_1)A_2 \qquad Z = M_6 + M_7$$
$$M_4 = (A_1 + B_1)D_2 \qquad\qquad T = M_2 - M_3 + M_5 - M_7$$

avec $MN = \begin{pmatrix} X & Y \\ Z & T \end{pmatrix}$.

Cette fois, $c_n = 7c_{\lceil n/2 \rceil} + \Theta(n^2)$, et $c_n = \Theta(n^{\log 7})$.