ALGO1 – Diviser pour régner

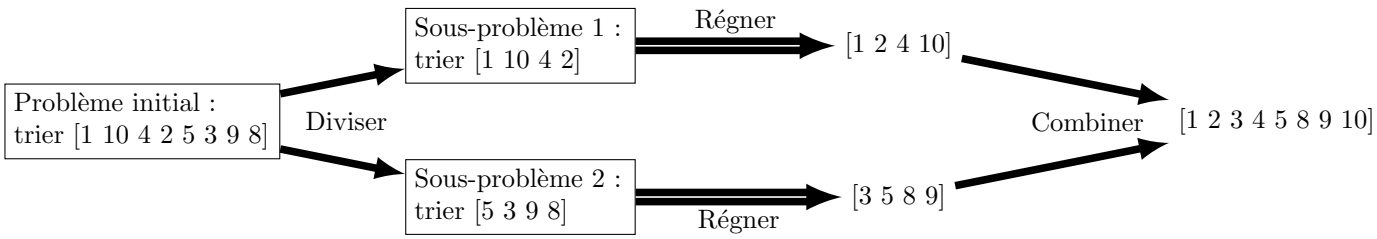
François Schwarzentruber

September 28, 2020

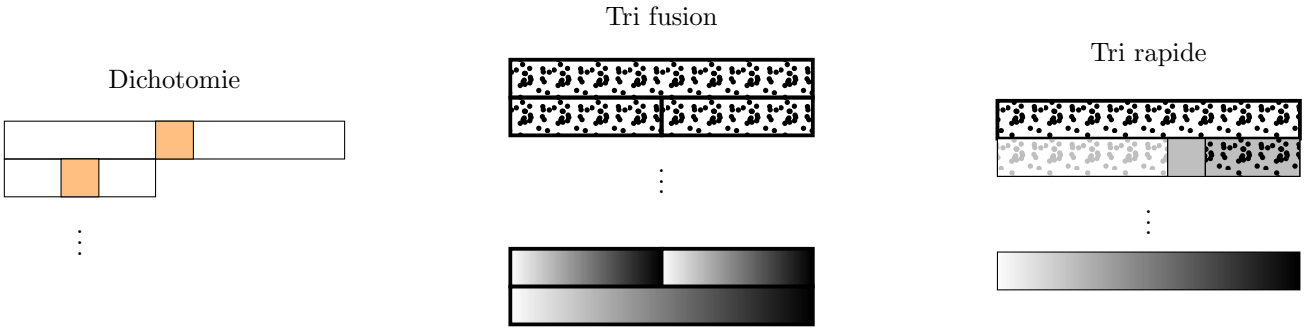
1 Motivation

	Complexité d'un algo naïf	Complexité d'un algo efficace obtenu via diviser pour régner
Chercher un élément dans un tableau trié à n éléments		
Tri par comparaison d'un tableau de taille n		
Calcul du k-ième élément d'un tableau à n éléments		
Multiplication de nombres à n bits chacun		
Multiplication de matrices de taille $n \times n$		
Calcul de la transformée de Fourier d'un signal de n échantillons		

2 Principe



3 Exemples

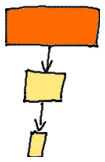


4 Théorème fondamental sur la complexité

Théorème 1 *Soit $a \geq 1$, $d \geq 2$ et $b \geq 0$. Considérons une suite $(\tau(n))_{n \in \mathbb{N}}$ qui vérifie, pour tout $n \geq 2$,*

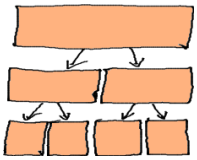
$$\tau(n) = a\tau\left(\lceil\frac{n}{d}\rceil\right) + O(n^b).$$

Alors :



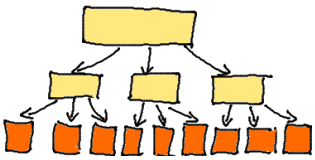
si $a < d^b$,

$\tau(n) = O(n^b)$



si $a = d^b$,

$\tau(n) = O(n^b \times \textit{hauteur})$
$= O(n^b \log_d n)$



si $d^b < a$,

$\tau(n) = O(\textit{nb feuilles})$
$= O(a^{\log_d n}) = O(n^{\log_d a})$

DÉMONSTRATION. On travaille à constante près. Donc on s'intéresse en fait à la suite suivante :

$$\tau(n) = a\tau\left(\lceil\frac{n}{d}\rceil\right) + n^b$$

et $\tau(1) = 1$.

Quand $n = d^h$ Démontrons d'abord le résultat lorsque n est une puissance de d, autrement dit que $n = d^h$ pour un certain h, qui est la hauteur de l'arbre des appels. A la profondeur ℓ, le coût dans un nœud est $\frac{n^b}{d^{\ell b}}$ et il y a a^ℓ nœuds.

Le tableau suivant résume les complexités à chaque niveau d'appels :

Niveau ℓ	Complexité dans un nœud	Complexité totale sur le niveau
0 (racine)	n^b	$1 \times n^b$
1	$\frac{n^b}{d^b}$	$a \times \frac{n^b}{d^b}$
$\vdots$	$\vdots$	$\vdots$
ℓ	$\frac{n^b}{d^{\ell b}}$	$a^\ell \times \frac{n^b}{d^{\ell b}}$
$\vdots$	$\vdots$	$\vdots$
h (feuilles)	1	a^h

Autrement dit,

$$\tau(n) := \sum_{\ell=0}^{h} a^\ell \times \frac{n^b}{d^{\ell b}} = n^b \sum_{\ell=0}^{h} \left(\frac{a}{d^b}\right)^\ell.$$

1. Si $a < d^b$, alors la raison de la série est plus petite que 1 et donc la série converge. Donc $\tau(n) = O(n^b)$.
2. Si $a = d^b$, alors la raison vaut 1 et la somme vaut $h = O(\log n)$. Donc $\tau(n) = O(n^b \log n)$.
3. Si $b^d < a$, alors la raison est plus grande que 1 et la série diverge. Donc $\tau(n) = O(n^b \left(\frac{a}{d^b}\right)^{\log_d n}) = O(n^{\log_d a})$.

Cela termine la preuve pour le comportement asymptotique quand n tend vers l'infini en prenant pour valeur des puissances de d.

Remarque 2 *Il faut bien être à l'aise avec les logarithmes. Voici les points clefs :*

- $x = d^{log_d x}$ *pour toute base* $d > 1$ *et pour tout nombre* $x > 0$*;*
- *tout se passe bien avec un exposant :* $x^t = \left(d^{log_d x}\right)^t = d^{t log_d x}$.

n quelconque Maintenant si n est quelconque, on a l'encadrement $d^{log_d n} \leq n < d \times d^{log_d n}$.

Lemme 3 *La fonction* $\tau : \mathbb{N} \to \mathbb{R}$ *est croissante.*

DÉMONSTRATION. Pour le montrer, on montre par récurrence sur n que τ est croissante sur $\{1, \ldots, n\}$.

- Pour $n = 1$, la propriété est triviale.

- Supposons pour un n donné que τ est croissante sur $\{1,\dots,n\}$ et montrons que τ est croissante sur $\{1,\dots,n+1\}$. On a :

  $\tau(n+1) = a\tau\left(\lceil\frac{n+1}{d}\rceil\right) + (n+1)^b \geq a\tau\left(\lceil\frac{n}{d}\rceil\right) + n^b = \tau(n)$ par hypothèse de récurrence (là on utilise que $d \geq 2$ pour être sûr que $\lceil\frac{n+1}{d}\rceil \leq n$).

- Par récurrence, on a montré que τ est croissante sur $\mathbb{N}$.

■

On conclut de la manière suivante. Par exemple, si $b > log_d a$, on a $\tau(n) \leq \tau(d^{\lceil log_d n\rceil})$ car τ est croissante. Mais on a $\tau(d^{\lceil log_d n\rceil}) = O(d^{\lceil log_d n\rceil b}) = O((dd^{\lfloor log_d n\rfloor})^b) \leq O(dn^b) = O(n^b)$.

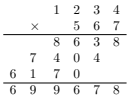
■

5 Algorithme de Karatsuba

Définition 4 (problème de la multiplication de nombres)

- *Entrée : deux entiers x, y donnés en base 2 avec n chiffres ;*
- *Sortie : le produit de x et y, $x \times y$, donnée en base 2 avec $2n$ chiffres.*

Théorème 5 *L'algorithme de multiplication appris à l'école primaire est en $\Theta(n^2)$.*



Théorème 6 *L'algorithme de Karatsuba est en $O(n^{\log_2 3})$.*

pre : deux entiers x, y de n-bits
post : le produit xy
fonction *karatsuba*(x, y)
  si $n = 1$ **alors**
    retourner xy
  sinon
    $x_G, x_D :=$ les $\lceil\frac{n}{2}\rceil$ bits les plus à gauche, les $\lfloor\frac{n}{2}\rfloor$ bits les plus à droite de x;
    $y_G, y_D :=$ les $\lceil\frac{n}{2}\rceil$ bits les plus à gauche, les $\lfloor\frac{n}{2}\rfloor$ bits les plus à droite de y;
    $P_1 =$*produit*(x_G, y_G)
    $P_2 =$*produit*(x_D, y_D)
    $P_3 =$*produit*$(x_G + x_D, y_G + y_D)$
    retourner $2^n P_1 + 2^{\frac{n}{2}}(P_3 - P_1 - P_2) + P_2$

5.0.1 Premier tentative de diviser pour régner

Appliquons 'diviser pour régner' pour créer un algorithme plus performant. Découpons les deux nombres x et y comme suit :

$$x = 2^{\lfloor\frac{n}{2}\rfloor}x_G + x_D \text{ et } y = 2^{\lfloor\frac{n}{2}\rfloor}y_G + y_D$$

où x_G, y_G, x_D, y_D sont des nombres entre 0 et $2^{\lceil\frac{n}{2}\rceil} - 1$. On écrit :

$$xy = 2^{2\lfloor\frac{n}{2}\rfloor}x_G y_G + 2^{\lfloor\frac{n}{2}\rfloor}(x_G y_D + y_G x_D) + x_D y_D.$$

La multiplication xy se ramène à 4 multiplications avec des nombres avec 2 fois moins de bits, puis des additions, et multiplication par des puissances de deux soit du $O(n)$. Le cas de base correspond à une multiplication avec des nombres avec 1 bit chacun. Cela donne

$$\tau(n) = 4\tau\left(\lceil\frac{n}{2}\rceil\right) + O(n).,$$ donc du $O(n^2)$, pas mieux qu'à l'école primaire.

5.0.2 Deuxième tentative

Voici l'astuce de Gauss (ou Karatsuba) :

$$x_G y_D + y_G x_D = (x_G + x_D)(y_G + y_D) - x_G y_G - x_D y_D.$$

Ainsi on se ramène à uniquement 3 multiplications, certes avec des additions supplémentaires mais qui ne coûtent pas si chères. On a :

$$xy = 2^n P_1 + 2^{\frac{n}{2}}(P_2 - P_1 - P_3) + P_3$$

avec $P_1 = x_G y_G$ et $P_2 = (x_G + x_D)(y_G + y_D)$ et $P_3 = x_D y_D$.

Chaque addition coûte $O(n)$. Chaque multiplication par une puissance de 2 coûte $O(n)$ (il faut juste décaler les bits). La relation de récurrence donne :

$\tau(n) = 3\tau\left(\lceil \frac{n}{2} \rceil\right) + O(n)$, donc $O(n^{\log_2 3})$, soit environ $O(n^{1.59})$.

Remarque 7 *Remarque technique concernant le produit P_2. Les représentations binaires des nombres $x_G + x_D$ et $y_G + y_D$ peuvent avoir $1 + \lceil \frac{n}{2} \rceil$ bits, i.e. dépasser $\lceil \frac{n}{2} \rceil$. Mais ce n'est pas grave. En effet, en toute rigueur on devrait écrire ce produit sous la forme*

$$(a2^{\lceil \frac{n}{2} \rceil + 1} + A) \times (b2^{\lceil \frac{n}{2} \rceil + 1} + B) = ab2^{2\lceil \frac{n}{2} \rceil + 2} + 2^{\lceil \frac{n}{2} \rceil + 1}(aB + bA) + AB$$

où a et b sont des bits. Il y a un traitement en $O(n)$ plus l'appel récursif correspondant au produit AB. On a caché ce détail dans le pseudo-code ci-dessous.

```
pre : deux entiers x, y de n-bits
post : le produit xy
fonction produit(x, y)
    si n = 1 alors
        retourner xy
    sinon
        x_G, x_D := les ⌈n/2⌉ bits les plus à gauche, les ⌊n/2⌋ bits les plus à droite de x;
        y_G, y_D := les ⌈n/2⌉ bits les plus à gauche, les ⌊n/2⌋ bits les plus à droite de y;
        P_1 = produit(x_G, y_G);
        P_2 = produit(x_D, y_D);
        retourner 2^n P_1 + 2^{n/2}(produit(x_G + x_D, y_G + y_D) − P_1 − P_2) + P_2
```

6 Algorithme de Strassen

Définition 8 (problème de la multiplication de matrices)

- *Entrée : deux matrices X et Y de taille n ;*
- *Sortie : le produit XY.*

Théorème 9 *L'algorithme naïf est en $\Theta(n^3)$.*

Théorème 10 *L'algorithme de Strassen est en $\Theta(n^{2.81})$.*

L'objectif est de créer un algorithme efficace pour calculer le produit XY de deux matrices X et Y de taille $n \times n$. L'algorithme naïf calcule chaque terme de XY en $O(n)$ et il y a n^2 termes, d'où une complexité totale de $O(n^3)$.

6.0.1 Premier essai diviser pour régner

L'idée consiste à découper les matrices en sous-blocs de la façon suivante : content...
$X = \begin{pmatrix} A & B \\ C & D \end{pmatrix}$ et $Y = \begin{pmatrix} E & F \\ G & H \end{pmatrix}$. On a alors : $XY = \begin{pmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{pmatrix}$.
D'où $T(n) = 8T(\frac{n}{2}) + O(n^2)$, i.e. $T(n) = O(n^{log_2 8}) = O(n^3)$.

6.0.2 Algorithme de Strassen

L'algorithme de Strassen calcule XY à partir de seulement 7 produits de matrices plus petites et non pas 8 ! En fait on a

Algorithme de Strassen

En écrivant $X = \begin{pmatrix} A & B \\ C & D \end{pmatrix}$ et $Y = \begin{pmatrix} E & F \\ G & H \end{pmatrix}$, calculer

$$XY = \begin{pmatrix} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_1 + P_5 - P_3 - P_7 \end{pmatrix}$$

où ces produits sont calculés récursivement :

- $P_1 = A(F - H)$
- $P_2 = (A + B)H$
- $P_3 = (C + D)E$
- $P_4 = D(G - E)$
- $P_5 = (A + D)(E + H)$
- $P_6 = (B - D)(G + H)$
- $P_7 = (A - C)(E + F)$

On a donc un algorithme de type diviser pour régner vérifiant la relation de récurrence :

$T(n) = 7T(\frac{n}{2}) + O(n^2)$ d'où d'après le théorème fondamental, $T(n) = O(n^{log_2 7}) = O(n^{2.81})$.

Le meilleur algorithme connu à ce jour, dû à Coppersmith-Winograd [**?**], est en $O(n^{2.376})$.

7 Transformation de Fourier Discrète Rapide (FFT)

Définition 11 *La transformée de Fourier Discrète est le problème algorithmique :*

- ***entrée :***
  - *un polynôme* $A = \sum_{i=0}^{n-1} a_i X^i$*, décrit avec un vecteur* $\vec{a} = (a_0, a_1, \ldots, a_{n-1})$*;*
  - *une racine* $n^{ème}$ *de l'unité* ω *;* *par exemple* $\omega = e^{\frac{2\pi i}{n}}$
- ***sortie :*** $A(\omega^0), A(\omega), \ldots, A(\omega^{n-1})$.

Théorème 12 *L'algorithme naïf est en* $\Theta(n^2)$ *.*

7.1 Principe

On décompose le polynôme A en sa partie paire P, et impaire I : $A = P(X^2) + X \times I(X^2)$.

Exemple 13

A	$=$	1	$+6X$	$-3X^2$	$+2X^3$	$+X^4$	$-7X^5$
P	$=$	1		$-3X$		$+X^2$	
I	$=$		6		$+2X$		$-7X^2$

entrée : un polynôme A, une racine de n-ème de l'unité ω
sortie : $A(\omega^0), A(\omega), \ldots, A(\omega^{n-1})$
fonction FFT(A, ω)
  si A est constant
    retourner $A(1)$
  sinon
    soit P la partie paire et I la partie impaire de A
    calculer $P(1), P(\omega^2) \ldots, P(\omega^{n-2})$ via $FFT(P, \omega^2)$
    calculer $I(1), I(\omega^2) \ldots, I(\omega^{n-2})$ via $FFT(I, \omega^2)$
    pour $k = 0$ à $n - 1$
      $A(\omega^k) := P(\omega^{2k}) + \omega^k I(\omega^{2k})$
  retourner $A(\omega^0), A(\omega), \ldots, A(\omega^{n-1})$

Théorème 14 *La complexité est en* $O(n \log n)$ *.*

Démonstration.

$$\tau(n) = 2\tau\left(\lceil \frac{n}{2} \rceil\right) + O(n).$$

■

7.2 Pseudo-code détaillé

On considère ici une racine n-ème de l'unité ω . Ainsi $\omega^{n/2} = -1$.

Posons $(s_0, \ldots, s_{n/2-1}) = (P(1), P(\omega^2) \ldots, P(\omega^{n-2})$ et $(s'_0, \ldots, s'_{n/2-1}) = (I(1), I(\omega^2) \ldots, I(\omega^{n-2})$. On a pour tout k,

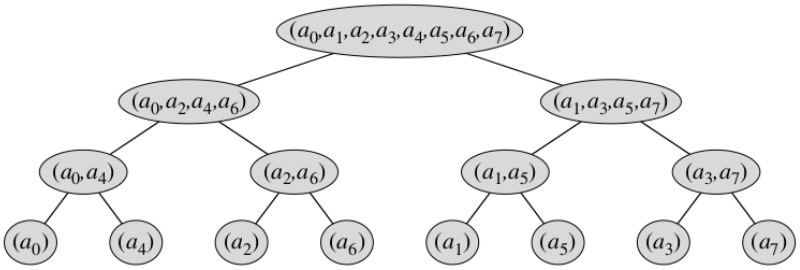
$$A(\omega^k) = P(\omega^{2k}) + \omega^k I(\omega^{2k}).$$

- pour tout $k \leq \frac{n}{2} - 1$, on a $A(\omega^k) = P(\omega^{2k}) + \omega^k I(\omega^{2k}) = s_k + \omega^k s'_k$;
- pour tout $k \leq \frac{n}{2} - 1$, on a $A(\omega^{k+n/2}) = P(\omega^{2(k+n/2)}) + \omega^{k+n/2} I(\omega^{2(k+n/2)}) = P(\omega^{2k}) - \omega^k I(\omega^{2k}) = s_k - \omega^k s'_k$.

entrée : des nombres $a_0, \ldots, a_{n-1}$, et une racine n-ème de l'unité primitive ω
sortie : $A(\omega^0), A(\omega), \ldots, A(\omega^{n-1})$ où $A = \sum_{i=0}^{n-1} a_i X^i$
fonction FFT$(a_0, \ldots, a_{n-1}, \omega)$
  si $n = 1$
    retourner a_0
  sinon
    $(s_0, \ldots, s_{n/2-1}) := FFT(a_0, a_2, \ldots, a_{n-2}, \omega^2)$
    $(s'_0, \ldots, s'_{n/2-1}) := FFT(a_1, a_3, \ldots, a_{n-1}, \omega^2)$
    $\alpha := 1$
    pour $k = 0$ à $n/2 - 1$
      $y_k = s_k + \alpha s'_k$
      $y_{k+n/2} = s_k - \alpha s'_k$ opération papillon
      $\alpha := \alpha \times \omega$
  retourner $y_0, \ldots, y_{n-1}$

7.3 Circuit pour la FFT

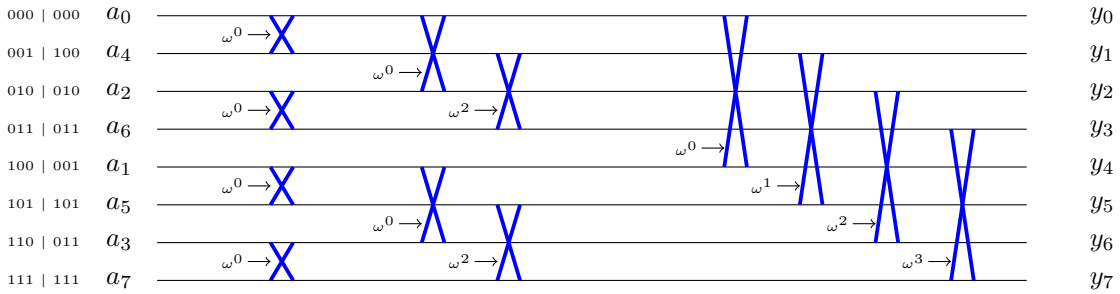
Voici l'arbre des appels récursifs de la FFT :



L'ordre 0, 4, 2, 6, 1, 5, 3, 7 s'obtient en prenant les miroirs des écritures binaires de 0, 1, 2, 3, 4, 5, 6, 7.

Le circuit suivant représente le calcul de la FFT, en lisant l'arbre ci-dessous. Les feuilles sont les entrées.

Le calcul en bleu dans l'algorithme s'appelle une *opération papillon* : on la représente par une croix bleue, agrémentée par la valeur de α.



Théorème 15 *Il y a* $\Theta(n \log n)$ *opération papillons.*

Théorème 16 *Le circuit est de profondeur* $\Theta(\log n)$ *. L'algorithme s'exécute en* $\Theta(\log n)$ *sur une architecture parallèle.*

7.4 Algorithme itératif

entrée : des nombres $a_0, \ldots, a_{n-1}$, et une racine n-ème de l'unité primitive ω
sortie : $A(\omega^0), A(\omega), \ldots, A(\omega^{n-1})$ où $A = \sum_{i=0}^{n-1} a_i X^i$

fonction FFT$(a_0, \ldots, a_{n-1}, \omega)$
  pour $k = 0$ à $n-1$ **faire**
    $s_k := a_{miroir(k)}$

  pour $m := 2, 4, 8, \ldots, n$ **faire**
    $\omega := e^{2i\pi/m}$
    pour $k = 0, m, 2m, \ldots, n-m$ **faire**
      $\alpha := 1$
      pour $j = 0$ à $m/2 - 1$ **faire**
        $s_{k+j}, s_{k+j+m/2} = s_{k+j} + \alpha s_{k+j+m/2}, s_{k+j} - \alpha s_{k+j+m/2}$
        $\alpha := \alpha \times \omega$

  retourner $s_0, \ldots, s_{n-1}$

7.5 Transformation de Fourier inverse

Définition 17 (matrice de Vandermonde) $V(\omega) = \begin{pmatrix} 1 & 1 & 1 & \ldots & 1 \\ 1 & \omega & \omega^2 & \ldots & \omega^{n-1} \\ 1 & \omega^2 & \omega^{2\times 2} & \ldots & \omega^{2\times(n-1)} \\ \vdots & \ldots & \ldots & \ldots & \vdots \\ 1 & \omega^{n-1} & \omega^{(n-1)\times 2} & \ldots & \omega^{(n-1)(n-1)} \end{pmatrix}$.

Proposition 18 *La transformée de Fourier Discrète revient à calculer $V(\omega)\vec{a}$ où ω racine n-ème primitive de l'unité.*

Théorème 19 *Si ω est une racine n-ème primitive de l'unité, $V(\omega)^{-1} = \frac{1}{n}V(\omega^{-1})$.*

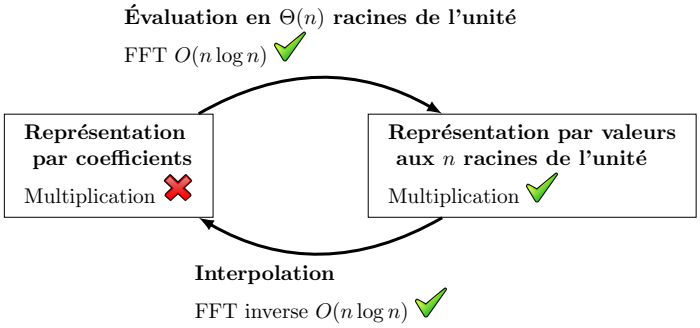
La transformée de Fourier est le vecteur $\vec{y} = V(\omega)\vec{a}$. La transformée inverse est : $\vec{a} := V(\omega)^{-1}\vec{y}$..

Corollaire 20 *On peut calculer la FFT inverse en appelant le même algo : $\frac{1}{n}FFT(\vec{y}, {\omega_n}^{-1})$.*

7.6 Multiplication de polynômes

Algorithme pour multiplier deux polynômes représentées par coefficients

1. On évalue les polynômes A, B de degré $n/2$ en les points $1, \omega, \ldots, \omega^{n-1}$ où $\omega = e^{\frac{2\pi i}{n}}$; (FFT)
2. On effectue la multiplication par valeur, on calcule $C(1) = A(1)B(1), C(\omega) = A(\omega)B(\omega)$, etc.
3. On interpole C pour obtenir ses coefficients. (FFT inverse)



Corollaire 21 *On peut multiplier deux polynômes de degré $O(n)$ en $O(n \log n)$.*

8 Type abstrait 'Polynôme'

Voici le type abstrait d'un polynôme :

- Évaluer le polynôme en un point ;
- Addition de deux polynômes ;
- Multiplier deux polynômes ; ← là, on se concentre sur cette opération
- Calculer une racine ;
- Dériver, etc.

8.1 Implémentations par coefficients

$$A = \sum_{i=0}^{n-1} a_i X^i$$

On stocke les coefficients $a_0, \ldots, a_{n-1}$ dans un tableau.

Évaluation On utilise la règle de Horner en $\Theta(n)$:

$$A(8) = a_0 + 8 \times (a_1 + 8 \times (a_2 + ... + 8 \times (a_{n-2} + 8 \times a_{n-1}))...))$$

Addition En $\Theta(n)$: on somme coefficients par coefficients.

Multiplication On cherche le produit de deux polynômes A et B de degré $n-1$ définis par :

$$A = \sum_{i=0}^{n-1} a_i X^i, B = \sum_{i=0}^{n-1} b_i X^i.$$

Le résultat est le polynôme $C = \sum_{k=0}^{2n-2} c_k$ de degré $2n-2$, avec :

$$c_k = \sum_{i=0}^{k} a_i b_{k-i}.$$

Remarque 22 *En théorie du signal, on peut avoir envie de construire un signal c à partir d'une superposition de signaux b décallés, pondérés par des coefficients a. C'est exactement le calcul ci-dessus !*

L'implémentation directe du produit de 2 polynômes est donc de complexité $\Theta(n^2)$.

8.2 Implémentation par paires point-valeur

Un polynôme A de degré $\leq n-1$ est représenté par un ensemble de n paires point-valeur :

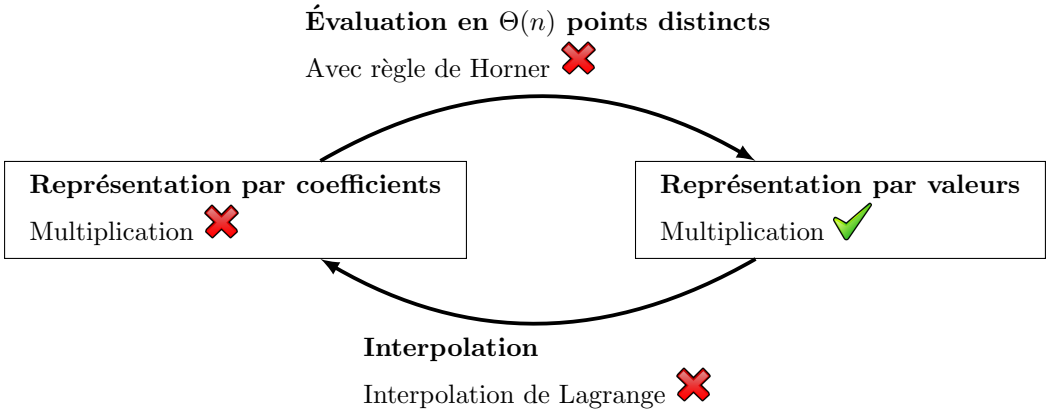
$$\{(x_0, y_0), \ldots, (x_{n-1}, y_{n-1})\}$$

Exemple 23 *Le polynôme $A = 2 + X^2 + 5X^3$ est représentée par l'ensemble de 4 points $\{(-1,-2),(0,2),(1,8),(2,46)\}$ (car $P(-1) = -2$, $P(0) = 2$, etc.).*

Addition. En $\Theta(n)$.

Multiplication. En $\Theta(n)$, mais il faut utiliser une représentation étendue.
A de degré $\leq n-1$, représenté par $\{(x_0, y_0), \ldots, (x_{2n-1}, y_{2n-1})\}$
B de degré $\leq n-1$, représenté par $\{(x_0, y'_0), \ldots, (x_{2n-1}, y'_{2n-1})\}$
alors AB de degré $\leq 2n-1$ représenté par $\{(x_0, y_0 y'_0), \ldots, (x_{2n-1}, y_{2n-1} y'_{2n-1})\}$

8.3 Conversion d'une représentation à une autre



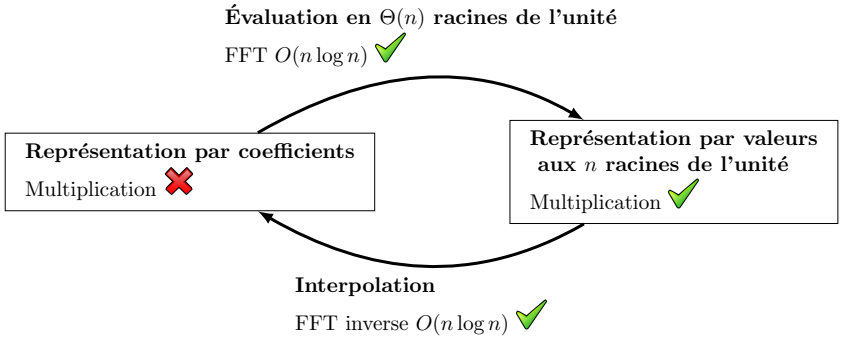
	par coefficients	par valeurs
Évaluation en un point	$O(n)$	?
Somme	$O(n)$	$O(n)$
Multiplication	$O(n^2)$	$O(n)$

Évaluation en $\Theta(n)$ points distincts. En $\Theta(n^2)$ en appliquant $\Theta(n)$ fois la règle de Horner.

Interpolation Trouver les coefficients du polynômes à partir des points s'appellent une *interpolation*. Cela requiert $\Theta(n^2)$ opérations élémentaires si on utilise l'interpolation de Lagrange qui permet de retrouver les coefficients du polynôme à partir de couples (x_i, y_i) :

$$\sum_i y_i \ell_i(X) \text{ où } \ell_i = \frac{\Pi_{j\neq i}(X-x_j)}{\Pi_{j\neq i}(x_i-x_j)}.$$

8.4 Avoir le meilleur des deux mondes avec la FFT



9 Sélection

Définition 24 (Problème de sélection)

- *Entrée : un tableau T d'éléments comparables, un entier k ;*
- *Sortie : le k-ème élément le plus petit de T.*

fonction *selection*(T, k)
  Choisir un élément x de T comme pivot
  Découper T en trois catégories : $T_<$, $T_=$ et $T_>$ (resp. les éléments inférieurs $< x$, égaux à x et $> x$)
  si $k \leq |T_<|$ **alors retourner** *selection*$(T_<, k)$
  sinon si $|T_<| < k \leq |T_<| + |T_=|$ **alors retourner** x
  sinon retourner *selection*$(T_>, k - |T_=| - |T_<|)$

Théorème 25 *La complexité pire cas est $\Theta(n^2)$.*

9.1 Choix aléatoire du pivot

Choisir aléatoirement uniformément le choix du pivot jusqu'à ce que $|T_<| \leq 3n/4$ et $|T_>| \leq 3n/4$.

Théorème 26 *L'espérance du temps passé pour choisir un tel pivot est $O(n)$.*

Démonstration. On choisit un indice i du tableau T entre 1 et n. Tester que $|T_<| \leq 3n/4$ et $|T_>| \leq 3n/4$ se fait en temps $O(n)$.

On a $|T_<| \leq 3n/4$ et $|T_>| \leq 3n/4$ dès lors que i est entre $n/4$ et $3n/4$. Ainsi :

$$\mathbb{P}(|T_<| \leq 3n/4 \text{ et } |T_>| \leq 3n/4) = 1/2$$

L'espérance E du nombre de lancers jusqu'à avoir un bon choix vérifie :

$$E = \sum_{j=1}^{\infty} j(1/2)^j$$

Or $E = \sum_{j=1}^{\infty} j(1/2)^j = 1 + \sum_{j=1}^{\infty}(j-1)(1/2)^j = 1 + \sum_{j=0}^{\infty}(j)(1/2)^j = 1 + E/2$
Donc $E = 2$.
Donc l'espérance du temps passé est $2O(n) = O(n)$.
■

Théorème 27 *L'espérance du temps passé de l'algorithme sélection avec ce choix aléatoire est $O(n)$.*

DÉMONSTRATION. On a une espérance de $O(n)$ à chaque appel récursif. En notant $T(n)$ l'espérance du temps pour l'algorithme, on a :

$$T(n) \leq T(3n/4) + O(n)$$

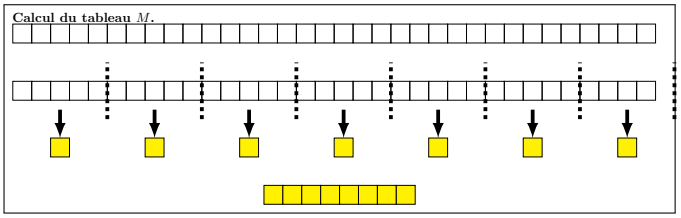
D'où du $T(n) = O(n)$. ■

9.2 Algorithme linéaire déterministe

fonction *selection*(T, k)

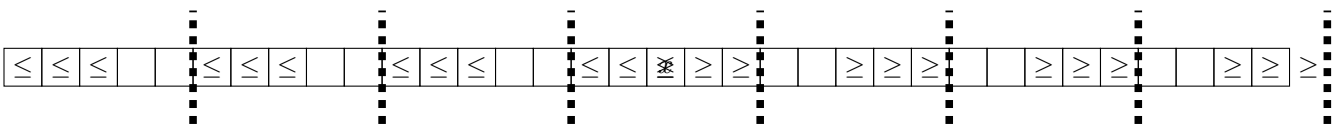
Découper T en blocs de taille 5 (sauf éventuellement le dernier de taille ≤ 5)
Calculer les médians de chaque blocs et les stocker dans un tableau M

$x :=$ *selection*$(M, \lfloor |M|/2 \rfloor)$
Découper T en trois catégories : $T_<$, $T_=$ et $T_>$
si $k \leq |T_<|$ **alors retourner** *selection*$(T_<, k)$
sinon si $|T_<| < k \leq |T_<| + |T_=|$ **alors retourner** x
sinon retourner *selection*$(T_>, k - |T_=| - |T_<|)$



Théorème 28 *L'algorithme de sélection est en $O(n)$ dans le pire cas.*

Pourquoi le pivot est-il bon ?
Trions (par la pensée) les blocs par médians croissants et dessinons les chacun triés.



DÉMONSTRATION. Le calcul du tableau M est en $O(n)$. Le calcul du médian de M est en $T(n/5)$.
Il y a 3/10 des éléments qui sont plus petits que ou égaux à x. Il y a 3/10 des éléments qui sont plus grands que ou égaux à x. Ainsi, nous avons $|T_<| \leq 7n/10$ et $|T_>| \leq 7n/10$. Ainsi la relation de récurrence est :

$$T(n) = T(n/5) + T(7n/10) + O(n).$$

Avec la méthode de l'arbre récursif (série convergente), on voit que $T(n) = O(n)$. ■

10 Notes bibliographiques

Diviser pour régner est au programme de Terminale ! Ce cours a été réalisé avec les deux ouvrages phares [DPV08] et [CLRS09]. La terminologie *diviser pour régner* remonte à Karatsuba et Ofman [KO63] en 1963. Le tri fusion aurait été inventé en 1945 par John von Neumann. Le tri rapide a été inventé en 1962 par Hoare [Hoa62].

L'algorithme de Strassen a fait des émois en 1969 [Str69] alors que tout le monde pensait que tout algorithme était forcément cubique dans le pire cas. Blum, Floyd, Pratt, Rivest, and Tarjan ont inventé l'algorithme de sélection en pire cas $O(n)$ [BFP+73]. Pour la FFT, on cite souvent Cooley et Tukey [CT65], mais l'algorithme était connu et on attribue sa première invention à Gauss.

Le papier original pour le master theorem est [BHS80]. Il y a des méthodes plus puissantes comme celle de Akra-Bazzi [AB98].

References

[AB98] Mohamad A. Akra and Louay Bazzi. On the solution of linear recurrence equations. *Comput. Optim. Appl.*, 10(2):195–210, 1998.

[BFP+73] Manuel Blum, Robert W. Floyd, Vaughan R. Pratt, Ronald L. Rivest, and Robert Endre Tarjan. Time bounds for selection. *J. Comput. Syst. Sci.*, 7(4):448–461, 1973.

[BHS80] Jon Louis Bentley, Dorothea Haken, and James B Saxe. A general method for solving divide-and-conquer recurrences. *ACM SIGACT News*, 12(3):36–44, 1980.

[CLRS09] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2009.

[CT65] James W Cooley and John W Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of computation*, 19(90):297–301, 1965.

[DPV08] Sanjoy Dasgupta, Christos H. Papadimitriou, and Umesh V. Vazirani. *Algorithms*. McGraw-Hill, 2008.

[Hoa62] C. A. R. Hoare. Quicksort. *Comput. J.*, 5(1):10–15, 1962.

[KO63] Anatolii Karatsuba and Yu Ofman. Multiplication of multidigit numbers on automata. In *Soviet physics doklady*, volume 7, pages 595–596, 1963.

[Str69] Volker Strassen. Gaussian elimination is not optimal. *Numerische mathematik*, 13(4):354–356, 1969.