Arithmétique multi-précision

- grands entiers ;
- réels représentés avec une grande précision ;
- arithmétique modulaire avec un grand modulo, éléments de grands corps finis ;
- polynômes de grand degré (à coefficients dans $\mathbb{Z}$, $\mathbb{Q}$, un corps fini, etc.).

Entiers et polynômes : cas très semblables, car manipuler $\sum_{i=0}^{n} a_i \beta^i$ (représentation en base β, typiquement une grande puissance de 2) ou

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

n'est pas très différent, l'arithmétique polynomiale étant souvent plus simple (pas de retenues).

Taille : de quelques centaines à plusieurs milliards de bits

- analyse numérique, lorsque les formats VF usuels ne suffisent pas ; vérification et mise au point d'algorithmes (ex. approximations polynomiales, valeurs tabulées, calcul de points difficiles à arrondir) : centaines de bits

Taille : de quelques centaines à plusieurs milliards de bits

- analyse numérique, lorsque les formats VF usuels ne suffisent pas ; vérification et mise au point d'algorithmes (ex. approximations polynomiales, valeurs tabulées, calcul de points difficiles à arrondir) : centaines de bits
- cryptographie : RSA, Diffie-Hellman. Besoin d'entiers de grande taille pour résister aux attaques : millers de bits

Taille : de quelques centaines à plusieurs milliards de bits

- analyse numérique, lorsque les formats VF usuels ne suffisent pas ; vérification et mise au point d'algorithmes (ex. approximations polynomiales, valeurs tabulées, calcul de points difficiles à arrondir) : centaines de bits
- cryptographie : RSA, Diffie-Hellman. Besoin d'entiers de grande taille pour résister aux attaques : millers de bits
- "mathématiques expérimentales" : recherches de propriétés, de contre-exemples, etc. : millions de bits et au-delà.

Á quoi ça sert ?

D'abord, c'est beau... mais parfois ça sert :

- Physique :
  - Frolov : calculs à précision > 100 chiffres décimaux en 2003 pour le calcul d'interactions électromagnétiques ;
  - Laskar (Observatoire Paris) : stabilité à long terme ($> 10^8$ années) du système solaire.

Á quoi ça sert ?

D'abord, c'est beau... mais parfois ça sert :

- Physique :
  - Frolov : calculs à précision > 100 chiffres décimaux en 2003 pour le calcul d'interactions électromagnétiques ;
  - Laskar (Observatoire Paris) : stabilité à long terme ($> 10^8$ années) du système solaire.
- Théorie des nombres :
  - certains problèmes nécessitent la manipulation de polynômes de degré ≈ 100000 ;
  - plus grand nombre premier explicitement connu

$$2^{24036583} - 1$$

(environ 7 millions de chiffres décimaux), prouvé en utilisant le test de Lucas-Lehmer ;

Á quoi ça sert ?

- découverte “expérimentale” de la formule de Bailey-Borwein-Plouffe

$$\pi = \sum_{k=0}^{\infty} \frac{1}{16^k} \left(\frac{4}{8k+1} - \frac{2}{8k+4} - \frac{1}{8k+5} - \frac{1}{8k+6} \right),$$

qui permet d’avoir “directement” le $j$ème chiffre binaire de π.

Á quoi ça sert ?

- découverte “expérimentale” de la formule de Bailey-Borwein-Plouffe

$$\pi = \sum_{k=0}^{\infty} \frac{1}{16^k} \left(\frac{4}{8k+1} - \frac{2}{8k+4} - \frac{1}{8k+5} - \frac{1}{8k+6} \right),$$

  qui permet d’avoir “directement” le $j$ème chiffre binaire de π.
- contre-exemples à des conjectures, ou renforcement de la confiance qu’on leur accorde.

Á quoi ça sert ?

- découverte "expérimentale" de la formule de Bailey-Borwein-Plouffe

$$\pi = \sum_{k=0}^{\infty} \frac{1}{16^k} \left(\frac{4}{8k+1} - \frac{2}{8k+4} - \frac{1}{8k+5} - \frac{1}{8k+6} \right),$$

  qui permet d'avoir "directement" le $j$ème chiffre binaire de π.
- contre-exemples à des conjectures, ou renforcement de la confiance qu'on leur accorde.
  Exemple (théorème de Robin). Notons $\sigma(n) = \sum_{d|n} d$.
  L'hypothèse de Riemann est vraie si et seulement si

$$\forall n > 5040, \frac{\sigma(n)}{n \ln \ln n} < e^{\gamma},$$

  où γ est la constante d'Euler.

Multiplication entière et réelle : algorithmes simples

La multiplication est l'opération la plus importante en multi-précision : la division et la racine carrées sont construites à partir de la multiplication (et sans coûter beaucoup plus cher, grâce à la méthode de Newton).

- base β : entier assez grand (les "chiffres" sont des entiers représentables sur le processeur utilisé, appelés "petits entiers") ;
- d'abord : multiplication et division d'un grand entier par un petit entier.

Multiplication par un “petit” entier, ou “scalaire”

Produit $a \times b$, où b est multiprécision, et a est “atomique”

$$b = b_{n-1}\beta^{n-1} + b_{n-2}\beta^{n-2} + \cdots + b_0$$

Multiplication par un “petit” entier, ou “scalaire”

Produit $a \times b$, où b est multiprécision, et a est “atomique”

$$b = b_{n-1}\beta^{n-1} + b_{n-2}\beta^{n-2} + \cdots + b_0$$

et on suppose $0 \leq a, b_i \leq \beta - 1$. L'algorithme est élémentaire :

$c_0 \leftarrow 0$
for $i = 0$ to $n - 1$ **do**
  $d \leftarrow a \cdot b_i + c_i$
  $p_i \leftarrow d \bmod \beta$
  $c_{i+1} \leftarrow d \operatorname{div} \beta$
end for
$p_n \leftarrow c_n$

Multiplication par un “petit” entier, ou “scalaire”

Produit $a \times b$, où b est multiprécision, et a est “atomique”

$$b = b_{n-1}\beta^{n-1} + b_{n-2}\beta^{n-2} + \cdots + b_0$$

et on suppose $0 \leq a, b_i \leq \beta - 1$. L'algorithme est élémentaire :

$c_0 \leftarrow 0$
for $i = 0$ to $n - 1$ **do**
  $d \leftarrow a \cdot b_i + c_i$
  $p_i \leftarrow d \bmod \beta$
  $c_{i+1} \leftarrow d \operatorname{div} \beta$
end for
$p_n \leftarrow c_n$

On montre par récurrence l'invariant :

$$(c_i p_{i-1} p_{i-2} \cdots p_0)_\beta = a \times (b_{i-1} b_{i-2} \cdots b_0)_\beta$$

et que les p_i et les c_i sont $\leq \beta - 1$.

Division par un “petit” entier

Divisons b par a, où a et b sont écrits comme précédemment. On obtient :

$q_{n-1} \leftarrow b_{n-1}$ div a
$r_{n-1} \leftarrow b_{n-1}$ mod a
for $i = n-2$ to 0 **do**
  $t_i \leftarrow \beta r_{i+1} + b_i$
  $q_i \leftarrow t_i$ div a
  $r_i \leftarrow t_i$ mod a
end for
$r \leftarrow r_0$

Division par un “petit” entier

Divisons b par a, où a et b sont écrits comme précédemment. On obtient :

$q_{n-1} \leftarrow b_{n-1}$ div a
$r_{n-1} \leftarrow b_{n-1}$ mod a
for $i = n-2$ to 0 **do**
  $t_i \leftarrow \beta r_{i+1} + b_i$
  $q_i \leftarrow t_i$ div a
  $r_i \leftarrow t_i$ mod a
end for
$r \leftarrow r_0$

et on montre par récurrence sur p que

$$\begin{aligned} a \times \left[\beta^{p-1} q_{n-1} + \beta^{p-2} q_{n-2} + \cdots + q_{n-p}\right] + r_{n-p} \\ = \beta^{p-1} b_{n-1} + \beta^{p-2} b_{n-2} + \cdots + b_{n-p} \end{aligned}$$

et que $0 \leq r_{n-p} \leq a$.

Multiplication naïve

- multiplier deux nombres multi-précision de même taille n, a et b
- se généralise sans problème à des tailles différentes
- la méthode naïve consiste à multiplier a par chacun des chiffres b_i à l'aide de la "multiplication scalaire" et à accumuler les résultats obtenus, avec le décalage adéquat
- Temps $O(n^2)$ (que Kolmogorov pensait optimal).

Multiplication de Karatsuba

A et B sur $2k$ "chiffres" $\longrightarrow$ on les décompose en $A = A_1\beta^k + A_0$ et $B = B_1\beta^k + B_0$, où A_1, A_0, B_1 et B_0 s'écrivent sur k chiffres.

Multiplication de Karatsuba

A et B sur $2k$ "chiffres" $\rightarrow$ on les décompose en $A = A_1\beta^k + A_0$ et $B = B_1\beta^k + B_0$, où A_1, A_0, B_1 et B_0 s'écrivent sur k chiffres.

- On définit $C = (A_1 - A_0)(B_1 - B_0)$, on obtient facilement :

$$AB = \beta^{2k} A_1 B_1 + \beta^k (A_1 B_1 + A_0 B_0 - C) + A_0 B_0,$$

Multiplication de Karatsuba

A et B sur $2k$ "chiffres" $\longrightarrow$ on les décompose en $A = A_1\beta^k + A_0$ et $B = B_1\beta^k + B_0$, où A_1, A_0, B_1 et B_0 s'écrivent sur k chiffres.

- On définit $C = (A_1 - A_0)(B_1 - B_0)$, on obtient facilement :

$$AB = \beta^{2k} A_1 B_1 + \beta^k (A_1 B_1 + A_0 B_0 - C) + A_0 B_0,$$

- $\longrightarrow$ calcule AB en n'effectuant que 3 multiplications de nombres de k chiffres : $A_1 B_1$, $A_0 B_0$ et $(A_1 - A_0)(B_1 - B_0)$.

Multiplication de Karatsuba

A et B sur $2k$ "chiffres" $\longrightarrow$ on les décompose en $A = A_1\beta^k + A_0$ et $B = B_1\beta^k + B_0$, où A_1, A_0, B_1 et B_0 s'écrivent sur k chiffres.

- On définit $C = (A_1 - A_0)(B_1 - B_0)$, on obtient facilement :

$$AB = \beta^{2k} A_1 B_1 + \beta^k (A_1 B_1 + A_0 B_0 - C) + A_0 B_0,$$

- $\longrightarrow$ calcule AB en n'effectuant que 3 multiplications de nombres de k chiffres : $A_1 B_1$, $A_0 B_0$ et $(A_1 - A_0)(B_1 - B_0)$.

Chaque fois que l'on double la taille n des nombres manipulés, on triple le temps de calcul $Kar(n)$. On trouve facilement

$$Kar(n) = O\left(n^{\log(3)/\log(2)}\right).$$

Multiplication de Karatsuba

A et B sur $2k$ "chiffres" $\longrightarrow$ on les décompose en $A = A_1\beta^k + A_0$ et $B = B_1\beta^k + B_0$, où A_1, A_0, B_1 et B_0 s'écrivent sur k chiffres.

- On définit $C = (A_1 - A_0)(B_1 - B_0)$, on obtient facilement :

$$AB = \beta^{2k} A_1 B_1 + \beta^k (A_1 B_1 + A_0 B_0 - C) + A_0 B_0,$$

- $\longrightarrow$ calcule AB en n'effectuant que 3 multiplications de nombres de k chiffres : $A_1 B_1$, $A_0 B_0$ et $(A_1 - A_0)(B_1 - B_0)$.

Chaque fois que l'on double la taille n des nombres manipulés, on triple le temps de calcul $Kar(n)$. On trouve facilement

$$Kar(n) = O\left(n^{\log(3)/\log(2)}\right).$$

$\log(3)/\log(2) \approx 1.585 \longrightarrow$ on gagne presque un facteur $\sqrt{n}$ par rapport à la méthode naïve.

Multiplication de Karatsuba

- En pratique devient intéressant dès que la taille des nombres manipulés dépasse quelques dizaines de mots machine.
- S'adapte trivialement pour effectuer le produit de 2 nombres complexes en ne faisant que 3 multiplications réelles.

Digression : produit matriciel de Strassen

- La méthode de Karatsuba a très certainement inspiré Strassen pour la mise au point de son algorithme de produit de matrices $n \times n$ en temps $n^{\log(7)/\log(2)}$.

Digression : produit matriciel de Strassen

- La méthode de Karatsuba a très certainement inspiré Strassen pour la mise au point de son algorithme de produit de matrices $n \times n$ en temps $n^{\log(7)/\log(2)}$.
- Supposons que l'on veuille multiplier deux matrices d'ordre 2n, A et B, décomposées en blocs de taille n comme suit :

$$A = \begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix}, \text{ et } B = \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix},$$

pour en calculer le produit

$$C = \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}.$$

si on note :

$$\begin{cases} M_1 &= (A_{1,1} + A_{2,2})(B_{1,1} + B_{2,2}) \\ M_2 &= (A_{2,1} + A_{2,2})B_{1,1} \\ M_3 &= A_{1,1}(B_{1,2} - B_{2,2}) \\ M_4 &= A_{2,2}(B_{2,1} - B_{1,1}) \\ M_5 &= (A_{1,1} + A_{1,2})B_{2,2} \\ M_6 &= (A_{2,1} - A_{1,1})(B_{1,1} + B_{1,2}) \\ M_7 &= (A_{1,2} - A_{2,2})(B_{2,1} + B_{2,2}) \end{cases},$$

si on note :

$$\begin{cases} M_1 &= (A_{1,1}+A_{2,2})(B_{1,1}+B_{2,2}) \\ M_2 &= (A_{2,1}+A_{2,2})B_{1,1} \\ M_3 &= A_{1,1}(B_{1,2}-B_{2,2}) \\ M_4 &= A_{2,2}(B_{2,1}-B_{1,1}) \\ M_5 &= (A_{1,1}+A_{1,2})B_{2,2} \\ M_6 &= (A_{2,1}-A_{1,1})(B_{1,1}+B_{1,2}) \\ M_7 &= (A_{1,2}-A_{2,2})(B_{2,1}+B_{2,2}) \end{cases},$$

alors

$$\begin{cases} C_{1,1} &= M_1+M_4-M_5+M_7 \\ C_{1,2} &= M_3+M_5 \\ C_{2,1} &= M_2+M_4 \\ C_{2,2} &= M_1-M_2+M_3+M_6 \end{cases}.$$

Multiplication de Toom-Cook

- Andrei Toom, 1963 — Stephen Cook, 1966 ;
- Pour simplifier, on supposera être en base 2, mais se généralise sans peine dans une autre base.
- On ramène le produit d'entiers au produit de polynômes. Idée :
- entiers $\longrightarrow$ polynômes : un entier a de n bits est découpé en k parties de longueur ℓ
  $\longrightarrow a = A_{k-1}2^{\ell(k-1)} + A_{k-2}2^{\ell(k-2)} + \cdots + A_0$, avec $0 \leq A_i \leq 2^\ell - 1$. Le nombre a est donc la valeur en 2^ℓ du polynôme

$$A(X) = A_{k-1}X^{k-1} + A_{k-2}X^{k-2} + \cdots + A_0.$$

Multiplication de Toom-Cook

- a et $b \longrightarrow$ polynômes A et B. Si on connait la valeur de $A(X)$ et $B(X)$ en $2k-1$ points choisis de sorte que ces valeurs soient simples à calculer et soient des entiers $\ll a$ et b, en multipliant ces valeurs 2 à 2, on connait la valeur du polynôme $R(X) = A(X)B(X)$ en ces $2k-1$ points ;

Multiplication de Toom-Cook

- a et $b \longrightarrow$ polynômes A et B. Si on connait la valeur de $A(X)$ et $B(X)$ en $2k-1$ points choisis de sorte que ces valeurs soient simples à calculer et soient des entiers $\ll a$ et b, en multipliant ces valeurs 2 à 2, on connait la valeur du polynôme $R(X) = A(X)B(X)$ en ces $2k-1$ points ;
- R est de degré $2k-2 \longrightarrow$ connaitre sa valeur en $2k-1$ points suffit pour le "reconstruire" (i.e., connaître ses coefficients—là encore, les points doivent être choisis pour que ce soit facile) ;

Multiplication de Toom-Cook

- a et $b \longrightarrow$ polynômes A et B. Si on connait la valeur de $A(X)$ et $B(X)$ en $2k-1$ points choisis de sorte que ces valeurs soient simples à calculer et soient des entiers $\ll a$ et b, en multipliant ces valeurs 2 à 2, on connait la valeur du polynôme $R(X) = A(X)B(X)$ en ces $2k-1$ points ;
- R est de degré $2k-2 \longrightarrow$ connaitre sa valeur en $2k-1$ points suffit pour le "reconstruire" (i.e., connaître ses coefficients—là encore, les points doivent être choisis pour que ce soit facile) ;
- la calcul de $AB(2^\ell)$, qui ne pose aucune difficulté particulière donne le résultat.

Multiplication de Toom-Cook

- a et $b \longrightarrow$ polynômes A et B. Si on connait la valeur de $A(X)$ et $B(X)$ en $2k-1$ points choisis de sorte que ces valeurs soient simples à calculer et soient des entiers $\ll a$ et b, en multipliant ces valeurs 2 à 2, on connait la valeur du polynôme $R(X) = A(X)B(X)$ en ces $2k-1$ points ;
- R est de degré $2k-2 \longrightarrow$ connaitre sa valeur en $2k-1$ points suffit pour le "reconstruire" (i.e., connaître ses coefficients—là encore, les points doivent être choisis pour que ce soit facile) ;
- la calcul de $AB(2^\ell)$, qui ne pose aucune difficulté particulière donne le résultat.

Ce mécanisme peut s'utiliser une fois ou de manière récursive (pour calculer AB aux $2k-1$ points choisis).

Multiplication de Toom-Cook

- Les points choisis en pratique sont de tous petits entiers $(\ldots, -2, -1, 0, 1, 2, \ldots)$, ou des inverses de toutes petites puissances de 2 (1/2, 1/4) ainsi que $+\infty$ (abus de langage : on considère le terme de tête du polynôme).
- Par exemple, GMP utilise ∞, 2, -1, 1, 0. L'idée qui sous-tend ceci est que les valeurs de A et B en ces points sont d'une taille juste légèrement plus grande que ℓ.
- avec des petites modifications de l'algorithme, des valeurs comme $\pm i$ ou $\pm\sqrt{2}$ seraient certainements intéressantes à utiliser.

Multiplication de Toom-Cook

- Ici : cas le plus utile en pratique : $k = 3$ (algorithme "Toom-3").
- Polynômes A et B : $A_2X^2 + A_1X + A_0$ et $B_2X^2 + B_1X + B_0$, où $A_2, A_1, A_0, B_2, B_1 et B_0$ s'écrivent sur au plus $\ell = \lceil n/3 \rceil$ chiffres.

Multiplication de Toom-Cook

- Ici : cas le plus utile en pratique : $k = 3$ (algorithme "Toom-3").
- Polynômes A et B : $A_2X^2 + A_1X + A_0$ et $B_2X^2 + B_1X + B_0$, où $A_2, A_1, A_0, B_2, B_1 et B_0$ s'écrivent sur au plus $\ell = \lceil n/3 \rceil$ chiffres.
- Nous devons choisir $2k - 1 = 5$ points. Nous choisirons ∞, 2, -1, 1, 0. La valeur de A en ces points est donc :
  - en ∞ : $A(\infty) = A_2$;
  - en 2 : $A(2) = 4A_2 + 2A_1 + A_0$;
  - en -1 : $A(-1) = A_2 - A_1 + A_0$;
  - en 1 : $A(1) = A_2 + A_1 + A_0$;
  - en 0 : $A(0) = A_0$.

Multiplication de Toom-Cook

- Ici : cas le plus utile en pratique : $k = 3$ (algorithme "Toom-3").
- Polynômes A et B : $A_2X^2 + A_1X + A_0$ et $B_2X^2 + B_1X + B_0$, où $A_2, A_1, A_0, B_2, B_1 et B_0$ s'écrivent sur au plus $\ell = \lceil n/3 \rceil$ chiffres.
- Nous devons choisir $2k - 1 = 5$ points. Nous choisirons ∞, 2, -1, 1, 0. La valeur de A en ces points est donc :
  - en ∞ : $A(\infty) = A_2$;
  - en 2 : $A(2) = 4A_2 + 2A_1 + A_0$;
  - en -1 : $A(-1) = A_2 - A_1 + A_0$;
  - en 1 : $A(1) = A_2 + A_1 + A_0$;
  - en 0 : $A(0) = A_0$.
- même chose pour B.

Multiplication de Toom-Cook

- En base 2, calculer les termes $A(i)$ ou $B(i)$ $(i = \infty, 2, \pm 1, 0)$: additions et décalages ;

Multiplication de Toom-Cook

- En base 2, calculer les termes $A(i)$ ou $B(i)$ $(i = \infty, 2, \pm 1, 0)$: additions et décalages ;
- la plus grande valeur que puissent avoir ces termes est $(4 + 2 + 1)(2^\ell - 1) = 3 \cdot 2^{\ell+1} - 6 \rightarrow$ ils sont donc représentables sur au plus $\ell + 3$ bits.

Multiplication de Toom-Cook

- En base 2, calculer les termes $A(i)$ ou $B(i)$ $(i = \infty, 2, \pm 1, 0)$: additions et décalages ;
- la plus grande valeur que puissent avoir ces termes est $(4 + 2 + 1)(2^\ell - 1) = 3 \cdot 2^{\ell+1} - 6 \rightarrow$ ils sont donc représentables sur au plus $\ell + 3$ bits.
- le calcul des produits $A(i) \cdot B(i)$ se fait donc par des multiplications de nombres d'au plus $\lceil n/3 \rceil + 3$ bits.

Multiplication de Toom-Cook

Passons maintenant au calcul des coefficients du polynôme $P = AB$, à partir des 5 valeurs $P(i) = A(i) \cdot B(i)$. Ce polynôme est de degré 4, et si on appelle P_k son coefficient de degré k, on a bien évidemment :

- en ∞ : $P(\infty) = P_4$;
- en 2 : $P(2) = 16P_4 + 8P_3 + 4P_2 + 2P_1 + P_0$;
- en -1 : $P(-1) = P_4 - P_3 + P_2 - P_1 + P_0$;
- en 1 : $P(1) = P_4 + P_3 + P_2 + P_1 + P_0$;
- en 0 : $P(0) = P_0$.

Multiplication de Toom-Cook

Passons maintenant au calcul des coefficients du polynôme $P = AB$, à partir des 5 valeurs $P(i) = A(i) \cdot B(i)$. Ce polynôme est de degré 4, et si on appelle P_k son coefficient de degré k, on a bien évidemment :

- en ∞ : $P(\infty) = P_4$;
- en 2 : $P(2) = 16P_4 + 8P_3 + 4P_2 + 2P_1 + P_0$;
- en -1 : $P(-1) = P_4 - P_3 + P_2 - P_1 + P_0$;
- en 1 : $P(1) = P_4 + P_3 + P_2 + P_1 + P_0$;
- en 0 : $P(0) = P_0$.

On connaît les $P(i)$, on cherche les P_k.

Multiplication de Toom-Cook

- c'est un système linéaire de matrice

$$\mathcal{M} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 16 & 8 & 4 & 2 & 1 \\ 1 & -1 & 1 & -1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}.$$

- Cette matrice est inversible (Vandermonde), et

$$\mathcal{M}^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ -2 & 1/6 & -1/6 & -1/2 & 1/2 \\ -1 & 0 & 1/2 & 1/2 & -1 \\ 2 & -1/6 & -1/3 & 1 & -1/2 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}.$$

Multiplication de Toom-Cook

- On a donc

$$\begin{cases} P_4 &= P(\infty) \\ P_3 &= -2P(\infty) + \frac{1}{6}P(2) - \frac{1}{6}P(-1) - \frac{1}{2}P(1) + \frac{1}{2}P(0) \\ P_2 &= -P(\infty) + \frac{1}{2}P(-1) + \frac{1}{2}P(1) - P(0) \\ P_1 &= 2P(\infty) - \frac{1}{6}P(2) - \frac{1}{3}P(-1) + P(1) - \frac{1}{2}P(0) \\ P_0 &= P(0) \end{cases}$$

- divisions par 2 : des décalages.
- divisions par 3 ou 6 : divisions par petits entiers (linéaires—pas besoin de nombreux chiffres fractionnaires du quotient : on sait que le résultat final p_i est un entier !).
- Il est possible de montrer qu'on ne peut pas éviter ces divisions.

Multiplication de Toom-Cook

- produit de 2 nombres de n bits : 5 produits de nombres de $\lceil n/3 \rceil + 3$ bits. En oubliant lâchement le $\lceil . \rceil$ et le $+3$, on trouve alors que la complexité est en

$$O\left(n^{\log 5/\log 3}\right) = O\left(n^{1.46\cdots}\right).$$

- Le cas général de l'algorithme de Toom-Cook (découpage en k entiers de longueur ℓ), appelé parfois "Toom-k" donnera une complexité en

$$O\left(n^{\log(2k-1)/\log k}\right).$$

Multiplication de Toom-Cook

Complexité en

$$O\left(n^{\log 5/\log 3}\right) = O\left(n^{1.46\cdots}\right).$$

Multiplication de Toom-Cook

Complexité en

$$O\left(n^{\log 5/\log 3}\right) = O\left(n^{1.46\cdots}\right).$$

- on peut donc pour tout $\epsilon > 0$ trouver un algorithme de multiplication en $0(n^{1+\epsilon})$.

Multiplication de Toom-Cook

Complexité en

$$O\left(n^{\log 5/\log 3}\right) = O\left(n^{1.46\cdots}\right).$$

- on peut donc pour tout $\epsilon > 0$ trouver un algorithme de multiplication en $0(n^{1+\epsilon})$.
- la "constante cachée" croît très rapidement avec k, ce qui fait que seuls Toom-3 et Toom-5 ont un intérêt pratique

Multiplication de Toom-Cook

Complexité en

$$O\left(n^{\log 5/\log 3}\right) = O\left(n^{1.46\cdots}\right).$$

- on peut donc pour tout $\epsilon > 0$ trouver un algorithme de multiplication en $0(n^{1+\epsilon})$.
- la "constante cachée" croît très rapidement avec k, ce qui fait que seuls Toom-3 et Toom-5 ont un intérêt pratique
- Raison : on n'a pas un nombre arbitrairement grand de points d'interpolation "simples" (i.e., pour lesquels les matrices A et A^{-1} sont toutes les deux très simples).

Multiplication de Toom-Cook

Complexité en

$$O\left(n^{\log 5/\log 3}\right) = O\left(n^{1.46\cdots}\right).$$

- on peut donc pour tout $\epsilon > 0$ trouver un algorithme de multiplication en $0(n^{1+\epsilon})$.
- la "constante cachée" croît très rapidement avec k, ce qui fait que seuls Toom-3 et Toom-5 ont un intérêt pratique
- Raison : on n'a pas un nombre arbitrairement grand de points d'interpolation "simples" (i.e., pour lesquels les matrices A et A^{-1} sont toutes les deux très simples).

Faire mieux ? prendre comme points les racines n^e de l'unité (dans $\mathbb{C}$ ou dans un anneau) $\longrightarrow$ FFT.

La transformée de Fourier discrète

- très utilisée en traitement du signal
- Au départ : séries de Fourier, représentation d'une fonction périodique f — ici de période 2π — comme somme d'une série trigonométrique : $a_0 + \sum_{n=1}^{+\infty}(a_n \cos nx + b_n \sin nx)$
- A soulevé de nombreuses questions intéressantes.
  - premières idées : Bernoulli (cordes vibrantes) ;
  - Fourier (travaillant sur l'équation de la chaleur) donne les relations

$$\begin{aligned} a_0 &= \frac{1}{2\pi}\int_0^{2\pi} f(t)dt \\ a_p &= \frac{1}{\pi}\int_0^{2\pi} f(t)\cos(pt)dt \\ b_p &= \frac{1}{\pi}\int_0^{2\pi} f(t)\sin(pt)dt \end{aligned}$$

    Plusieurs questions se sont posées : pour quelles fonctions f la série obtenue converge-t-elle, et a-t-elle pour somme f ?
  - Problème étudié par Dirichlet, Riemann, Lebesgue...

La transformée de Fourier discrète dans $\mathbb{C}$

Soit $\omega = e^{-\frac{2i\pi}{n}}$, et définissons la matrice

$$F_\omega = \begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \cdots & \omega^{2(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & \omega^i & \omega^{2i} & \cdots & \omega^{i(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \cdots & \omega^{(n-1)(n-1)} \end{pmatrix}$$

i.e., $F_{\omega;ij} = \omega^{(i-1)(j-1)}$, alors la TFD d'ordre n est l'application $x \in \mathbb{C}^n \to X \in \mathbb{C}^n$ définie par

$$X = F_\omega x.$$

On note parfois $X = \mathcal{F}_n(x)$.

TFD inverse

$$F_{\omega^{-1}} = \begin{pmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega^{-1} & \omega^{-2} & \dots & \omega^{-(n-1)} \\ 1 & \omega^{-2} & \omega^{-4} & \dots & \omega^{-2(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & \omega^{-i} & \omega^{-2i} & \dots & \omega^{-i(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & \omega^{-(n-1)} & \omega^{-2(n-1)} & \dots & \omega^{-(n-1)(n-1)} \end{pmatrix}$$

vérifie

$$F_{\omega} F_{\omega^{-1}} = n I_n.$$

On a donc :

$$(F_{\omega})^{-1} = \frac{1}{n} F_{\omega^{-1}}.$$

TFD inverse

D'où la définition de la transformée de Fourier inverse :

$$x_\ell = \frac{1}{n} \sum_{k=0}^{n-1} X_k (\omega^{-1})^{k\ell}.$$

On notera $x = \mathcal{F}_n^{-1}(X)$.

Algorithme FFT (Fast Fourier Transform) de Cooley-Tukey

- probablement inventé vers 1805 par Gauss ;

Algorithme FFT (Fast Fourier Transform) de Cooley-Tukey

- probablement inventé vers 1805 par Gauss ;
- redécouvert (puis oublié) en 1914 ;

Algorithme FFT (Fast Fourier Transform) de Cooley-Tukey

- probablement inventé vers 1805 par Gauss ;
- redécouvert (puis oublié) en 1914 ;
- formulation moderne : article de 1965 de Cooley et Tukey

Algorithme FFT (Fast Fourier Transform) de Cooley-Tukey

- probablement inventé vers 1805 par Gauss ;
- redécouvert (puis oublié) en 1914 ;
- formulation moderne : article de 1965 de Cooley et Tukey
- TFD, en supposant que n est une puissance de 2 (sinon, $\exists$ algorithmes basés sur factorisation de n).

Algorithme FFT (Fast Fourier Transform) de Cooley-Tukey

- probablement inventé vers 1805 par Gauss ;
- redécouvert (puis oublié) en 1914 ;
- formulation moderne : article de 1965 de Cooley et Tukey
- TFD, en supposant que n est une puissance de 2 (sinon, $\exists$ algorithmes basés sur factorisation de n).

On veut calculer, pour $k = 0, \ldots, n-1$,

$$X_k = \sum_{\ell=0}^{n-1} x_\ell \omega^{k\ell}.$$

Idée : se ramener à des TFD d'ordre $n/2$.

Algorithme FFT (Fast Fourier Transform) de Cooley-Tukey

- n est une puissance de 2,
- ω^2 est une racine $(n/2)$ième de l'unité

On s'intéresse aux TFD d'ordre $n/2$ (avec ω^2 comme racine de l'unité) des vecteurs

$$(x_0, x_2, x_4, \ldots, x_{n-2}) \to (P_0, P_1, \ldots, P_{n/2-1})$$

et

$$(x_1, x_3, x_5, \ldots, x_{n-1}) \to (I_0, I_1, \ldots, I_{n/2-1}).$$

On a :

$$\begin{aligned} X_k &= \sum_{\ell=0}^{n-1} x_\ell \omega^{k\ell} \\ &= \sum_{m=0}^{n/2-1} \left(x_{2m} \omega^{k\cdot(2m)} + x_{2m+1} \omega^{k\cdot(2m+1)} \right) \\ &= \sum_{m=0}^{n/2-1} x_{2m} (\omega^2)^{km} + \omega^k \sum_{m=0}^{n/2-1} x_{2m+1} (\omega^2)^{km}. \end{aligned}$$

Algorithme FFT (Fast Fourier Transform) de Cooley-Tukey

- pour $k \leq n/2 - 1$, on retombe immédiatement sur la FFT d'ordre $n/2$:

$$X_k = P_k + \omega^k I_k$$

- pour k compris entre $n/2$ et $n-1$, notons $k' = k - \frac{n}{2}$. Il vient

$$(\omega^2)^{km} = (\omega^2)^{(k'+n/2)m} = (\omega^2)^{k'm}$$

et $\omega^k = \omega^{k'+\frac{n}{2}} = -\omega^{k'}$, d'où l'on tire

$$X_k = P_{k-n/2} - \omega^{k-n/2} I_{k-n/2}.$$

On ramène donc le calcul d'une TFD d'ordre n à celui de 2 TFD d'ordre $n/2$. L'algorithme FFT est constitué par ce calcul récursif.

Algorithme FFT (Fast Fourier Transform) de Cooley-Tukey

- on suppose que les ω^i (pour $i < n$) sont précalculés,
- la FFT d'ordre 2 consomme 2 opérations ($\omega^{n/2} = -1$ est racine carrée de l'unité, donc $X_0 = x_0 + x_1$ et $X_1 = x_0 - x_1$),
- la FFT d'ordre n demande 2 FFT d'ordre $n/2$ et $2n$ opérations dans $\mathbb{C}$.

Conclusion : Par récurrence, la FFT d'ordre n consomme $2n \log_2 n - n = O(n \log n)$ opérations dans $\mathbb{C}$.

Algorithme FFT (Fast Fourier Transform) de Cooley-Tukey

- on suppose que les ω^i (pour $i < n$) sont précalculés,
- la FFT d'ordre 2 consomme 2 opérations ($\omega^{n/2} = -1$ est racine carrée de l'unité, donc $X_0 = x_0 + x_1$ et $X_1 = x_0 - x_1$),
- la FFT d'ordre n demande 2 FFT d'ordre $n/2$ et $2n$ opérations dans $\mathbb{C}$.

Conclusion : Par récurrence, la FFT d'ordre n consomme $2n \log_2 n - n = O(n \log n)$ opérations dans $\mathbb{C}$.
S'adapte trivialement à la TFD inverse ($\omega \longrightarrow \omega^{-1}$, et division par n).

Remarque importante

de

$$F_\omega \cdot x = \begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \cdots & \omega^{2(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & \omega^i & \omega^{2i} & \cdots & \omega^{i(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \cdots & \omega^{(n-1)(n-1)} \end{pmatrix} \cdot \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_i \\ \vdots \\ x_{n-1} \end{pmatrix},$$

on tire que X_k (i.e., la $k+1$ième composante de $F.x$) est la valeur du polynôme

$$P_x(t) = x_0 + x_1 t + x_2 t^2 + \cdots + x_{n-1} t^{n-1}$$

au point ω^k.

Remarque importante

- FFT : algorithme de calcul de la valeur d'un polynôme aux points $1, \omega, \omega^2, \ldots, \omega^{n-1}$ en ne consommant que $O(n \log n)$ opérations dans $\mathbb{C}$.

Remarque importante

- FFT : algorithme de calcul de la valeur d'un polynôme aux points $1, \omega, \omega^2, \ldots, \omega^{n-1}$ en ne consommant que $O(n \log n)$ opérations dans $\mathbb{C}$.
- FFT inverse : moyen rapide d'interpoler les coefficients d'un polynôme de degré $n-1$ à partir de sa valeur aux points $1, \omega, \omega^2, \ldots, \omega^{n-1}$.

Remarque importante

- FFT : algorithme de calcul de la valeur d'un polynôme aux points $1, \omega, \omega^2, \ldots, \omega^{n-1}$ en ne consommant que $O(n \log n)$ opérations dans $\mathbb{C}$.
- FFT inverse : moyen rapide d'interpoler les coefficients d'un polynôme de degré $n-1$ à partir de sa valeur aux points $1, \omega, \omega^2, \ldots, \omega^{n-1}$.

$\longrightarrow$ même principe évaluation-interpolation que l'algorithme de Toom-Cook.

Application à la multiplication

Idée générale d'algorithme pour multiplier deux grands entiers.

- on veut multiplier deux entiers (représentés en base β) :

$$x = x_{p-1}\beta^{p-1} + x_{p-2}\beta^{p-2} + \cdots + x_0$$

et

$$y = y_{p-1}\beta^{p-1} + y_{p-2}\beta^{p-2} + \cdots + y_0$$

- on leur associe les deux polynômes

$$x(t) = x_{p-1}t^{p-1} + x_{p-2}t^{p-2} + \cdots + x_0$$

et

$$y(t) = y_{p-1}t^{p-1} + y_{p-2}t^{p-2} + \cdots + y_0$$

Application à la multiplication

- on construit des vecteurs de dimension $2p$, en complétant les coefficients de $x(t)$ et $y(t)$ avec des zéros $(\underbrace{0, \cdots, 0}_{p \text{ termes}} x_{p-1}, x_{p-2}, \ldots, x_0)$ et $(\underbrace{0, \cdots, 0}_{p \text{ termes}} y_{p-1}, y_{p-2}, \ldots, y_0)$
- on calcule par l'algorithme FFT la TFD d'ordre $2p$ de ces deux vecteurs, on obtient $(X_{2p-1}, \ldots, X_0)$ et $(Y_{2p-1}, \ldots, Y_0)$
- on calcule les produits termes à termes des X_i et des Y_i, i.e., on calcule

$$C_i = X_i \cdot Y_i$$

pour $i = 0, 1, \ldots, 2p-1$. Suivant la taille de X_i et Y_i pourra se faire récursivement ou par un produit de deux petits entiers.

Application à la multiplication

- Coefficients C_i : valeurs du polynôme $x(t)y(t)$ en les racines $2p$ièmes de l'unité.
- on calcule la TFD inverse d'ordre $2p$ de $(C_{2p-1}, C_{2p-2}, \cdots, C_0)$, qui donne un vecteur $(c_{2p-1}, c_{2p-2}, \ldots, c_0)$;
- le polynôme $x(t)y(t)$ vaut $c_{2p-1}t^{2p-1} + c_{2p-2}t^{2p-2} + \cdots + c_0$, et l'entier xy vaut $c_{2p-1}\beta^{2p-1} + c_{2p-2}\beta^{2p-2} + \cdots + c_0$

Application à la multiplication

Attention :

- Il ne faut pas en déduire que la complexité de la multiplication d'entiers est en $O(n \log n)$, ce sera un peu plus, car dans $\mathbb{C}$, les calculs devront être menés avec une précision qui dépendra de n.
- Pour que ceci soit intéressant, il faut que les produits $X_i \cdot Y_i$ n'aient besoin de se faire qu'avec une précision très petite devant la taille de x et y (on verra que c'est de l'ordre du logarithme de cette précision)

Algorithme de Schönhage et Strassen (il y en a 2)

- calculs approchés en utilisant des nombres VF ;
- ϵ : erreur relative d'une opération complexe ;
- pour simplifier, on suppose que tous les arrondis se font dans le sens qui minimise le module du résultat.

En partant de $(x_{p-1}, x_{p-2}, \ldots, x_0)$ et $(y_{p-1}y_{p-2}, \ldots, y_0)$ où $|x_i|, |y_i| \leq \beta - 1$ sont les chiffres des entiers que l'on veut multiplier, on va calculer comme expliqué plus haut les coefficients $(c_{2p-1}, c_{2p-2}, \ldots, c_0)$. Ils sont calculés de manière approchée mais ce sont des entiers $\longrightarrow$ si on s'arrange pour que l'erreur absolue sur chaque c_i soit $< 1/2$, on les retrouvera sans difficulté

Algorithme de Schönhage et Strassen (il y en a 2)

Une opération élémentaire de l'algorithme FFT :

- opération exacte : $a \leftarrow b + \rho c$, où ρ est une puissance de ω ;
- calcul effectivement réalisé : $a' \leftarrow \circ(b' + \rho' c')$, où $\circ$ désigne l'arrondi.

On a $|\rho| = 1$, et on suppose que les valeurs précalculées des ρ' sont telles que $|\rho'| \leq 1$ (arrondis dans le "bon sens" lors du précalcul), et avec une erreur relative $\leq \epsilon$, i.e.,

$$\left| \frac{\rho' - \rho}{\rho} \right| \leq \epsilon.$$

On note $\ell = \log_2(2p) = \log_2(p) + 1)$.

Algorithme de Schönhage et Strassen (il y en a 2)

L'étude d'erreur n'est pas difficile, mais fastidieuse. À la fin, on trouve une erreur finale (sur les c_i) majorée par

$$6 \cdot \ell \cdot 2^{2\ell} \cdot \epsilon \cdot (\beta - 1)^2.$$

On veut que cette erreur soit $< 1/2$, ce qui donne la condition

$$\epsilon < \frac{1}{12 \cdot \ell \cdot 2^{2\ell} \cdot (\beta - 1)^2} \tag{1}$$

Avoir une erreur relative ϵ demande à manipuler des nombres d'au moins $1 - \log_2(\epsilon)$ bits (vous verrez partout $-\log_2(\epsilon)$, mais il est nécessaire d'avoir un bit de plus pour arrondir d'une façon qui minimise le module : on n'arrondit pas forcément au plus près). Il faut donc que l'on représente les nombres sur

$$2\log(\beta - 1) + 2\ell + \log \ell + \log 12 + 1 \text{ bits}$$

Algorithme de Schönhage et Strassen (il y en a 2)

Donc on se ramène de la manipulation de nombres de

$$N = p \log_2(\beta) \text{ bits}$$

à celle de nombres de (on rappelle : $\ell = \log_2(p) + 1$))

$$2 \log(\beta - 1) + 2 \log_2 p + 2 + \log_2(\log_2(p) + 1) + \log_2 3 + 3 \text{ bits}$$

Par exemple, avec $N = 2^{20} \approx 10^6$ bits, et $p = 32768$ (donc $\beta = 2^{N/p} = 2^{32}$), on se retrouve à manipuler des nombres de 104 bits.
De manière générale, on se retrouve donc de la manipulation de nombres de N bits à celle de nombres d'environ

$$2\frac{N}{p} + 2 \log_2(p) \text{ bits.}$$

Algorithme de Schönhage et Strassen (il y en a 2)

Si on ne fait qu'une seule étape de l'algorithme, en faisant les multiplications élémentaires des "petits" nombres avec l'algorithme en $O(n^2)$, on fait $O(p \log p)$ opérations de coût $O\left(\frac{N}{p} + \log p\right)^2$, ce qui donne un coût total (en temps) en

$$O\left(p \cdot \log p \cdot \left(\frac{N}{p} + \log p\right)^2\right).$$

Algorithme de Schönhage et Strassen (il y en a 2)

Si on ne fait qu'une seule étape de l'algorithme, en faisant les multiplications élémentaires des "petits" nombres avec l'algorithme en $O(n^2)$, on fait $O(p \log p)$ opérations de coût $O\left(\frac{N}{p} + \log p\right)^2$, ce qui donne un coût total (en temps) en

$$O\left(p \cdot \log p \cdot \left(\frac{N}{p} + \log p\right)^2\right).$$

En prenant $p = N/\log_2(N)$ pour équilibrer les termes (en pratique, recherche systématique de l'optimum autour de cette valeur), on obtient un coût en temps en

$$O\left(N \log^2 N\right).$$

Algorithme de Schönhage et Strassen (il y en a 2)

Si on ne fait qu'une seule étape de l'algorithme, en faisant les multiplications élémentaires des "petits" nombres avec l'algorithme en $O(n^2)$, on fait $O(p \log p)$ opérations de coût $O\left(\frac{N}{p} + \log p\right)^2$, ce qui donne un coût total (en temps) en

$$O\left(p \cdot \log p \cdot \left(\frac{N}{p} + \log p\right)^2\right).$$

En prenant $p = N/\log_2(N)$ pour équilibrer les termes (en pratique, recherche systématique de l'optimum autour de cette valeur), on obtient un coût en temps en

$$O\left(N \log^2 N\right).$$

On s'est ramené d'une taille N à une taille $\log N$: pas de cas réels où ceci demande à nouveau une très grande précision $\longrightarrow$ dans la vraie vie cet algorithme ne s'utilise pas récursivement

Algorithme de Schönhage et Strassen (il y en a 2)

En théorie, si on réutilise cet algorithme récursivement pour calculer les produits terme à terme $X_i \cdot Y_i$ et ceux qui apparaissent dans les calculs élémentaires $a \leftarrow b + \rho c$ de le FFT, on arrive à un coût en

$$N \log N \log\log N \log\log\log N \cdots \times 2^{O(\log^* N)}.$$

En effet, si $\tau(N)$ est le temps de calcul d'un produit de nombres de N bits, alors en choisissant à nouveau de découper nos nombres en $p = N/\log_2 N$ blocs, on effectue $\lambda(p \log p)$ opérations sur des nombres de taille $O(\log N)$, par conséquent

$$\tau(N) = \lambda \frac{N}{\log N} \log\left(\frac{N}{\log N}\right) \times \tau(\log N) \approx \lambda(N \times \tau(\log N)),$$

ce qui donne

$$\tau(N) = N \log(N) \log\log(N) \log\log\log(N) \cdots \times 2^{O(\log^* N)}.$$

L'autre algorithme de Schönhage et Strassen

- utilise la TFD dans l'anneau

$$\mathbb{Z}/(2^{2^k}+1)\mathbb{Z};$$

- complexité en

$$O(N\log(N)\log\log(N));$$

L'autre algorithme de Schönhage et Strassen

- utilise la TFD dans l'anneau

$$\mathbb{Z}/(2^{2^k}+1)\mathbb{Z};$$

- complexité en

$$O(N\log(N)\log\log(N));$$

- a été implanté. . . pour multiplier 2 nombres de 784141 mots de 64 bits, $\approx$ 1 seconde sur un opteron 4GHz ;

L'autre algorithme de Schönhage et Strassen

- utilise la TFD dans l'anneau

$$\mathbb{Z}/(2^{2^k}+1)\mathbb{Z};$$

- complexité en

$$O(N\log(N)\log\log(N));$$

- a été implanté. . . pour multiplier 2 nombres de 784141 mots de 64 bits, $\approx$ 1 seconde sur un opteron 4GHz ;
- a été l'algorithme de meilleure complexité connue jusqu'en 2007 ;

L'autre algorithme de Schönhage et Strassen

- utilise la TFD dans l'anneau

$$\mathbb{Z}/(2^{2^k}+1)\mathbb{Z};$$

- complexité en

$$O(N\log(N)\log\log(N));$$

- a été implanté. . . pour multiplier 2 nombres de 784141 mots de 64 bits, $\approx$ 1 seconde sur un opteron 4GHz ;
- a été l'algorithme de meilleure complexité connue jusqu'en 2007 ;
- algorithme de Martin Fürer en

$$N\log(N)\cdot 2^{O(\log^*(N))}.$$