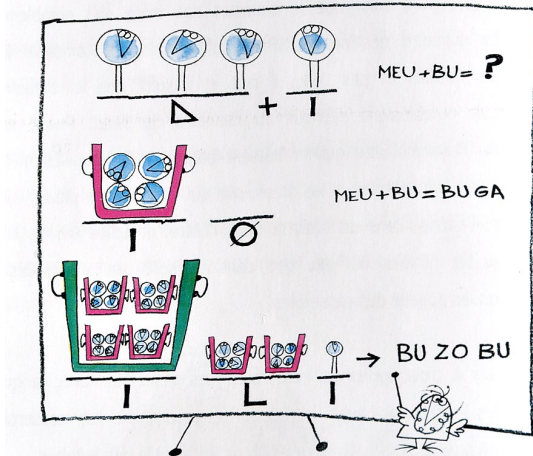


Algorithmes arithmétiques : J'apprends à compter aux ordinateurs



Jean-Michel Muller
CNRS – Laboratoire LIP

<http://perso.ens-lyon.fr/>

jean-michel.muller/

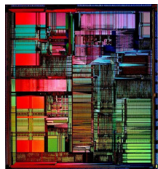
Contraintes lorsqu'on implante une arithmétique en machine

- **Vitesse** : la météo de demain doit être calculée en moins de 24h ;
- **Précision** :
 - certaines prédictions de la relativité générale vérifiées avec erreur relative $\approx 10^{-14}$;
 - calcul formel, cryptographie, «expériences mathématiques» : quelques milliers de chiffres ;
 - record : 5000 milliards de décimales de π (Yee et Kodo, 2010) ;
- **«taille»** : surface des circuits, consommation mémoire ;
- **Energie consommée** : autonomie, chauffe des circuits ;
- **Simplicité d'implantation et d'utilisation** : si une arithmétique est trop ésotérique, personne ne l'utilisera.

On ne vit pas dans un monde parfait !

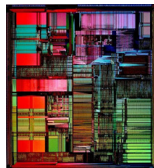
Certes, on fait des calculs impensables il y a 60 ans, mais l'informatique est victime de son succès :

- programmes et circuits de plus en plus gros et complexes : plus grande probabilité que des erreurs restent → programmes et circuits faux, mauvaises «spécifications» ;
- milliards d'opérations consécutives : comment se comportent les erreurs d'arrondi ?



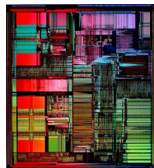
On peut faire du très mauvais travail...

- 1994 : «bug» de la division du processeur Pentium d'Intel, 8391667/12582905 donnait 0.666869... au lieu de 0.666910... ;



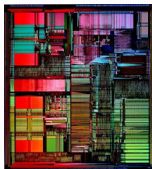
On peut faire du très mauvais travail...

- 1994 : «bug» de la division du processeur Pentium d'Intel, 8391667/12582905 donnait 0.666869... au lieu de 0.666910... ;
- Sur certains ordinateurs Cray on pouvait déclencher un overflow en multipliant par 1 ;



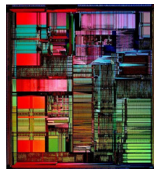
On peut faire du très mauvais travail...

- 1994 : «bug» de la division du processeur Pentium d'Intel, 8391667/12582905 donnait 0.666869... au lieu de 0.666910... ;
- Sur certains ordinateurs Cray on pouvait déclencher un overflow en multipliant par 1 ;
- Maple, version 6.0. Entrez 214748364810, vous obtiendrez 10.



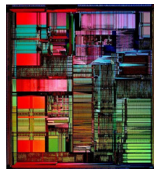
On peut faire du très mauvais travail...

- 1994 : «bug» de la division du processeur Pentium d'Intel, $8391667/12582905$ donnait $0.666869\dots$ au lieu de $0.666910\dots$;
- Sur certains ordinateurs Cray on pouvait déclencher un overflow en multipliant par 1 ;
- Maple, version 6.0. Entrez 214748364810, vous obtiendrez 10.
- Maple, version 7.0, si l'on calcule $\frac{5001!}{5000!}$ on obtient 1 au lieu de 5001 ;



On peut faire du très mauvais travail...

- 1994 : «bug» de la division du processeur Pentium d'Intel, $8391667/12582905$ donnait $0.666869\dots$ au lieu de $0.666910\dots$;
- Sur certains ordinateurs Cray on pouvait déclencher un overflow en multipliant par 1 ;
- Maple, version 6.0. Entrez 214748364810, vous obtiendrez 10.
- Maple, version 7.0, si l'on calcule $\frac{5001!}{5000!}$ on obtient 1 au lieu de 5001 ;
- Excel'2007 (premières versions), calculez $65535 - 2^{-37}$, vous obtiendrez 100000 ;



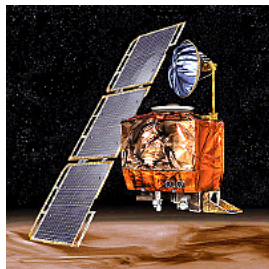
Un petit problème de spécification...

- la sonde Mars Climate Orbiter s'est écrasée sur Mars en 1999 ;



Un petit problème de spécification...

- la sonde Mars Climate Orbiter s'est écrasée sur Mars en 1999 ;
- une partie des développeurs des logiciels supposait que l'unité de longueur était le mètre ;



Un petit problème de spécification...

- la sonde Mars Climate Orbiter s'est écrasée sur Mars en 1999 ;
- une partie des développeurs des logiciels supposait que l'unité de longueur était le mètre ;
- l'autre partie croyait que c'était le pied.



L'arithmétique des ordinateurs est une fusée à deux étages

Premier étage :

- entiers **bornés** (p. ex. compris entre -2^{63} et $2^{63} - 1$ avec un processeur 64 bits) ;
- réels «virgule flottante» de précision bornée ;
- opérateurs arithmétiques **matériels** (i.e., dessinés sur silicium) ;
- algorithmes avec contrainte géométrique ;
- l'addition est encore du domaine de la recherche.

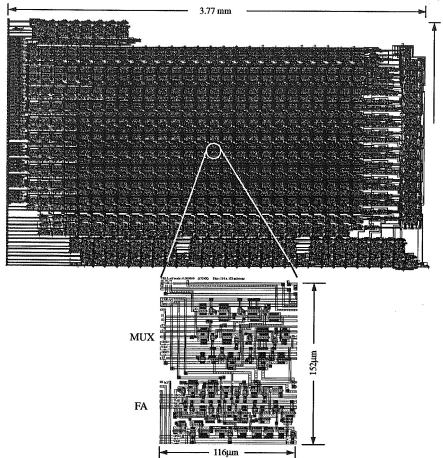


Figure 15. The layout of the 32-bit multiplier.

L'arithmétique des ordinateurs est une fusée à deux étages

Deuxième étage :

- on dispose des 4 opérations arithmétiques sur les «entiers machine» (en général, 32 ou 64 bits) ;
- on veut manipuler des nombres de taille arbitraire, en utilisant les «briques de base» du premier étage ;
- opérateurs **logiciels** (des programmes) ;
- il faut (un petit peu !) oublier ce qu'on a appris à l'école.

Ce soir : on va étudier ce 2ème étage, en se concentrant surtout sur la **multiplication**.

On va s'intéresser à la «complexité» des algorithmes

- notation $\mathcal{O}$: soient 2 fonctions f et g de $\mathbb{N}$ dans $\mathbb{R}^+$. On dit que

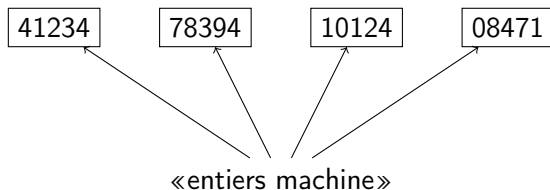
$$f = \mathcal{O}(g)$$

s'il existe $n_0 \in \mathbb{N}$ et $c > 0$ tels que pour tout $n \geq n_0$, $f(n) \leq c \cdot g(n)$;

- par exemple on sait trier un tableau de n nombres en $\mathcal{O}(n \log n)$ comparaisons ;
- quand on veut faire des implantations pratiques, on ne peut pas négliger la constante «cachée» dans le $\mathcal{O}$, mais regarder la complexité permet de :
 - s'abstraire du matériel sous-jacent ;
 - voir si nos solutions «passent l'échelle».

Représentation des «grands entiers»

Exemple (base 10) : on veut représenter $a = 41234783941012408471$.



$$a = 41234 \times \beta^3 + 78394 \times \beta^2 + 10124 \times \beta + 8471.$$

- Ici : $\beta = 10^5$. En pratique, on prend la plus grande puissance de 2 représentable dans un entier machine ;
- numération classique, dans une grande base.

Addition de grands entiers (n chiffres de base β)

Exemple dans le cas $\beta = 10^5$:

$$\begin{array}{r} 12435\ 32729\ 82667\ 34237 \\ +\ 23895\ 79335\ 31512\ 20925 \\ \hline \end{array}$$

Temps $\mathcal{O}(n)$.

Addition de grands entiers (n chiffres de base β)

Exemple dans le cas $\beta = 10^5$:

$$\begin{array}{r} 1 0 \\ 12435 \ 32729 \ 82667 \ 34237 \\ + \ 23895 \ 79335 \ 31512 \ 20925 \\ \hline 14179 \ 55162 \end{array}$$

Temps $\mathcal{O}(n)$.

Addition de grands entiers (n chiffres de base β)

Exemple dans le cas $\beta = 10^5$:

$$\begin{array}{r} 12435 32729 82667 34237 \\ + 23895 79335 31512 20925 \\ \hline 12065 14179 55162 \end{array}$$

Temps $\mathcal{O}(n)$.

Addition de grands entiers (n chiffres de base β)

Exemple dans le cas $\beta = 10^5$:

$$\begin{array}{r} \quad \quad \quad \textcolor{red}{1} \quad \quad \textcolor{red}{1} \quad \quad \textcolor{red}{0} \\ 12435 \ 32729 \ 82667 \ 34237 \\ + \ 23895 \ 79335 \ 31512 \ 20925 \\ \hline 36331 \ 12065 \ 14179 \ 55162 \end{array}$$

Temps $\mathcal{O}(n)$.

Multiplication d'un grand entier par un entier machine

Exemple dans le cas $\beta = 10^5$:

12456 33829 82967 54222

×

38236

Temps $\mathcal{O}(n)$.

Multiplication d'un grand entier par un entier machine

Exemple dans le cas $\beta = 10^5$:

$$\begin{array}{r} 20732 \\ 12456\ 33829\ 82967\ 54222 \\ \times 38236 \\ \hline 32392 \end{array}$$

Temps $\mathcal{O}(n)$.

Multiplication d'un grand entier par un entier machine

Exemple dans le cas $\beta = 10^5$:

$$\begin{array}{r} 31723\ 20732 \\ 12456\ 33829\ 82967\ 54222 \\ \times \qquad \qquad \qquad 38236 \\ \hline 46944\ 32392 \end{array}$$

Temps $\mathcal{O}(n)$.

Multiplication d'un grand entier par un entier machine

Exemple dans le cas $\beta = 10^5$:

$$\begin{array}{r} 12935\ 31723\ 20732 \\ 12456\ 33829\ 82967\ 54222 \\ \times \hspace{10em} 38236 \\ \hline 17367\ 46944\ 32392 \end{array}$$

Temps $\mathcal{O}(n)$.

Multiplication d'un grand entier par un entier machine

Exemple dans le cas $\beta = 10^5$:

$$\begin{array}{r} 12935 \ 31723 \ 20732 \\ 12456 \ 33829 \ 82967 \ 54222 \\ \times \qquad \qquad \qquad 38236 \\ \hline 4762 \ 80551 \ 17367 \ 46944 \ 32392 \end{array}$$

Temps $\mathcal{O}(n)$.

Division Euclidienne d'un grand entier par un entier machine

Exemple dans le cas $\beta = 10^5$:

37829 23456 35222

12034

Temps $\mathcal{O}(n)$.

Division Euclidienne d'un grand entier par un entier machine

Exemple dans le cas $\beta = 10^5$:

37829 23456 35222	12034
1727 23456	3

Temps $\mathcal{O}(n)$.

Division Euclidienne d'un grand entier par un entier machine

Exemple dans le cas $\beta = 10^5$:

37829 23456 35222	12034
1727 23456	3 14352
11488 35222	

Temps $\mathcal{O}(n)$.

Division Euclidienne d'un grand entier par un entier machine

Exemple dans le cas $\beta = 10^5$:

37829 23456 35222	12034
1727 23456	3 14352 95465
11488 35222	
9412	

Temps $\mathcal{O}(n)$.

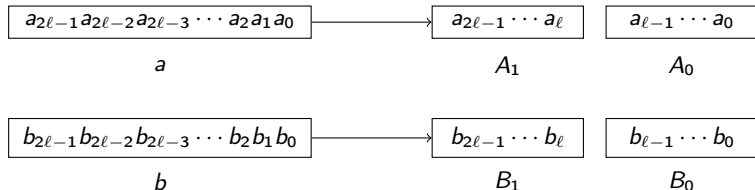
Multiplication de grands entiers (n chiffres de base β)

- on a envie de dire que c'est n multiplications d'un grand entier par un entier machine et $n - 1$ additions $\rightarrow$ temps $\mathcal{O}(n^2)$;
- on est conforté dans cette impression quand on voit le tableau :

$$\begin{array}{r} 1278395184 \\ \times 8453871031 \\ \hline 1278395184 \\ 3835185552 \\ 0000000000 \\ 1278395184 \\ 8948766288 \\ 10227161472 \\ 3835185552 \\ 6391975920 \\ 5113580736 \\ 10227161472 \\ \hline 10807388012187514704 \end{array}$$

Une autre façon d'écrire la multiplication en temps $\mathcal{O}(n^2)$

Si a et b s'écrivent sur $n = 2\ell$ «chiffres» de base β on les coupe en deux :



→ décomposition

$$a = A_1\beta^\ell + A_0$$

et

$$b = B_1\beta^\ell + B_0,$$

où A_1, A_0, B_1 et B_0 s'écrivent sur ℓ chiffres.

Une autre façon d'écrire la multiplication en temps $\mathcal{O}(n^2)$

$$a = A_1\beta^\ell + A_0$$

et

$$b = B_1\beta^\ell + B_0,$$

On a donc :

$$ab = \beta^{2\ell} A_1 B_1 + \beta^\ell (A_1 B_0 + A_0 B_1) + A_0 B_0$$

→ quand on double la taille des nombres on multiplie par 4 le temps de calcul ;

→ algorithme de temps $\mathcal{O}(n^2)$.

Et pourtant... multiplication de Karatsuba (1962)

- On avait

$$ab = \beta^{2\ell} A_1 B_1 + \beta^\ell (A_1 B_0 + A_0 B_1) + A_0 B_0;$$

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

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

→ au prix de 3 add/sous supplémentaires, on calcule ab en n'effectuant que 3 multiplications de nombres de ℓ chiffres :

- $A_1 B_1$,
- $A_0 B_0$, et
- $(A_1 - A_0)(B_1 - B_0)$.

Et pourtant... multiplication de Karatsuba (1962)

Temps de calcul ?

$$Kar(n) = 3Kar(n/2) + \gamma n$$

└─ additions, multiplications
par puissances de β

En supposant que n est une puissance de 2, donne

$$Kar(n) = (Kar(1) + 2\gamma) \cdot n^{\frac{\log 3}{\log 2}} - 2\gamma n.$$

On a donc

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

- le gain est important : $\log(3)/\log(2) \approx 1.585$;
- en pratique : à partir d'environ 100 chiffres décimaux ;
- Il est surprenant que personne n'ait trouvé cela avant 1962 !

A donné des idées...

- une autre «évidence» était qu'il fallait $\mathcal{O}(n^3)$ opérations pour multiplier 2 matrices de taille $n \times n$;
- Algorithme de Strassen : produit de matrices $n \times n$ en $\mathcal{O}(n^{\log(7)/\log(2)})$ opérations ($\log(7)/\log(2) \approx 2.807$).

Supposons que l'on veuille multiplier deux matrices $2n \times 2n$, A et B , décomposées en blocs $n \times n$ 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

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

A donné des idées...

Si on note :

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

alors

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

- doubler la taille des matrices $\rightarrow$ multiplier par 7 le temps de calcul ;
- amélioré depuis (Coppersmith-Winograd : $\mathcal{O}(n^{2.376})$). On ne connaît pas la complexité optimale (au moins n^2).

Faire mieux que l'algorithme de Karatsuba ?

Rappel : nombre a de 2ℓ chiffres coupé en deux parties A_1 et A_0 de sorte que $A = \beta^\ell A_1 + A_0$. Même chose pour b .

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

avec $C = (A_1 - A_0)(B_1 - B_0)$.

Si on définit $A(X) = A_1 X + A_0$ et $B(X) = B_1 X + B_0$ alors :

- $a = A(\beta^\ell)$ et $b = B(\beta^\ell)$;
- $(A_0 - A_1) = -A(-1)$, même chose pour B ,

→ on calcule la valeur de $AB(X)$ aux points 0 (terme $A_0 B_0$), -1 (terme C), et «à l'infini» (terme $A_1 B_1$), puis on en déduit ses coefficients.

Généraliser cette approche : couper a et b en k parties au lieu de 2.

Multiplication de Toom (1963) et Cook (1966) – en base 2

- entiers $\rightarrow$ polynômes : un entier a de n bits est découpé en k parties de longueur ℓ :

$$\begin{array}{ccccccc} \boxed{a_{n-1} \cdots a_{n-\ell}} & \boxed{a_{n-\ell-1} \cdots a_{n-2\ell}} & \boxed{\cdots} & \boxed{a_{\ell-1} \cdots a_0} \\ A_{k-1} & A_{k-2} & & A_0 \end{array}$$

On a donc :

$$a = A_{k-1}2^{\ell(k-1)} + A_{k-2}2^{\ell(k-2)} + \cdots + A_0,$$

avec $0 \leq A_i \leq 2^\ell - 1$. Le nombre a est donc la valeur en 2^ℓ du polynôme

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

- aux 2 grands entiers a et b que l'on veut multiplier, on associe donc 2 polynômes $A(X)$ et $B(X)$.

Algorithme de Toom et Cook

- On calcule la valeur de A et B en $2k - 1$ points choisis de sorte que ces valeurs soient
 - simples à calculer ;
 - soient des entiers bien plus petits que a et b ;
- on multiplie ces valeurs 2 à 2 $\rightarrow$ valeur du polynôme $P = AB$ en ces $2k - 1$ points ;
- comme P est de degré $2k - 2$, connaître sa valeur en $2k - 1$ points suffit pour le «reconstruire» (les points doivent être tels que cette reconstruction soit facile) ;
- le calcul de $ab = P(2^\ell)$ ne pose aucune difficulté particulière (des décalages et une «grande» addition).

Quels points choisir ?

- de tous petits entiers : $-2, -1, 0, 1, 2, \dots$;
- des inverses de toutes petites puissances de 2 ;
- ∞ (abus de langage : coefficient dominant) ;
- on pourrait facilement prendre aussi $\pm i, \pm\sqrt{2}$, etc.

Le logiciel GMP utilise $k = 3$, et les $2k - 1 = 5$ points $\infty, 2, -1, 1, 0$. On va détailler cette solution.

On a donc :

$$\begin{aligned} A &= A_2 X^2 + A_1 X + A_0, \\ B &= B_2 X^2 + B_1 X + B_0, \end{aligned}$$

où A_2, A_1, A_0, B_2, B_1 et B_0 s'écrivent sur au plus $\ell = \lceil n/3 \rceil$ bits (i.e., ils sont inférieurs ou égaux à $2^\ell - 1$).

Evaluation des polynômes A et B aux points $\infty, 2, -1, 1, 0$

- 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 .

Notons que la plus grande valeur absolue possible d'un des $A(i)$ ou $B(i)$ est

$$(4 + 2 + 1) \cdot (2^\ell - 1) < 2^{\ell+3}$$

→ les $A(i)$ et $B(i)$ s'écrivent sur au plus $\ell + 3$ bits.

Etape suivante : pour chaque point i on calcule
 $P(i) = A(i) \cdot B(i)$

À partir des 5 valeurs $P(i)$ on veut reconstruire le polynôme P . P est de degré 4, et si P_j est son coefficient de degré j , on a :

- 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 : $P(0) = P_0$.

→ système linéaire de matrice $\mathcal{M} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 16 & 8 & 4 & 2 & 1 \\ 1 & -1 & 1 & -1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$.

Reconstruire P ?

$\mathcal{M}$ est inversible (c'est—presque—une matrice de Vandermonde), et d'inverse

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

on a donc

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

Résultat final ?

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

- divisions par 2 : décalages ;
- divisions par 3 ou 6 (donc en pratique par 3) : à l'aide de la division par un entier machine présentée auparavant ;
- elles n'ont pas besoin d'être précises : P_i est entier $\rightarrow$ il suffit de le calculer avec erreur $< 1/2$;
- produit (entier) final :

$$P_0 + P_1 \cdot 2^\ell + P_2 \cdot 2^{2\ell} + P_3 \cdot 2^{3\ell} + P_4 \cdot 2^{4\ell}$$

Combien ça coûte ?

- produit de 2 nombres de n bits $\rightarrow$ calcul de 5 produits de nombres de $\lceil n/3 \rceil + 3$ bits plus des opérations (additions, décalages) de coût linéaire en n .
- en travaillant un peu, complexité en

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

Gain relativement léger par rapport à l'algorithme de Karatsuba ($1.585 \rightarrow 1.465$) : il faut des valeurs de n assez grandes pour que cet algorithme prenne le pas (la constante cachée dans le $\mathcal{O}$ est plus importante).

Se généralise ?

- l'algorithme «général» de Toom-Cook (on coupe a et b en k entiers de ℓ bits) est de complexité

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

→ pour tout $\epsilon > 0$ il y a un algorithme de complexité $\mathcal{O}(n^{1+\epsilon})$.

- en pratique on ne va pas vraiment au delà de $k = 5$: la «constante cachée» dans le $\mathcal{O}$ devient trop importante.

Raison : il n'y a pas tant de «points simples» $-2, -1, 0, 1, 2, \infty$ que cela.

Mais : on peut garder l'idée de multiplier des entiers en se ramenant à des polynômes, et en effectuant le cycle

évaluation → produits points à points → interpolation

Multiplication basée sur la transformée de Fourier rapide

Deux algorithmes dus à Schönhage et Strassen : l'un utilise la FFT dans $\mathbb{C}$, l'autre la FFT dans $\mathbb{Z}/(2^{2^k} + 1)\mathbb{Z}$.

- ici : dans $\mathbb{C}$.
- Si $\omega = e^{2i\pi/n}$, la **transformée de Fourier discrète** d'ordre n du vecteur $a = (a_0, \dots, a_{n-1})$ est le vecteur $\hat{a} = F_\omega \cdot a$, où

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

$\hat{a}_i$ est la **valeur du polynôme** $a_0 + a_1X + \dots + a_{n-1}X^{n-1}$ en ω^i .

Multiplication basée sur la transformée de Fourier rapide

- F_ω est d'inverse

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

- il existe un algorithme (FFT : *Fast Fourier Transform*) pour calculer la transformée de Fourier discrète (et son inverse) en effectuant $\mathcal{O}(n \log n)$ opérations dans $\mathbb{C}$;

Multiplication basée sur la transformée de Fourier rapide

- Transformée de Fourier : valeurs d'un polynôme aux points ω^j ;
- Transformée de Fourier inverse : coefficients d'un polynôme à partir de ses valeurs aux points ω^j .

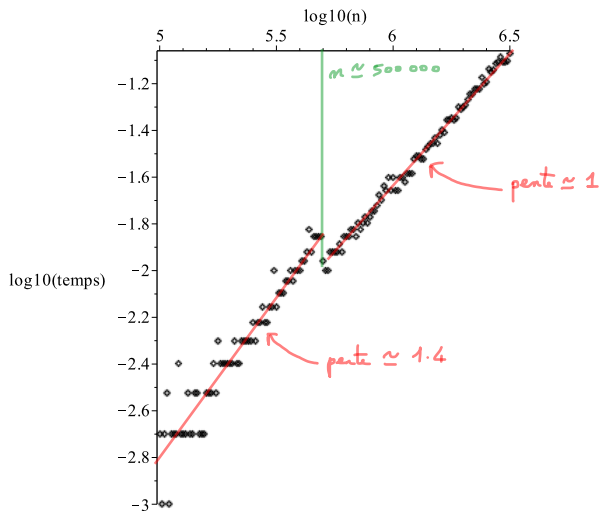
On utilise ces algorithmes : résultats approchés pour lesquels il faut une erreur $< 1/2$.

Etude d'erreur très soigneuse $\rightarrow$ précision avec laquelle doivent être faites les opérations dans $\mathbb{C}$.

Temps de la multiplication de nombres de N bits :

$$\underbrace{N \log(N) \log \log(N) \log \log \log(N) \dots}_{\log^* N \text{ termes}} \times 2^{O(\log^* N)}.$$

Maple sur mon Mac, nombres de 10000 à 3000000 de chiffres



Et la division ne coûte pas vraiment plus cher...

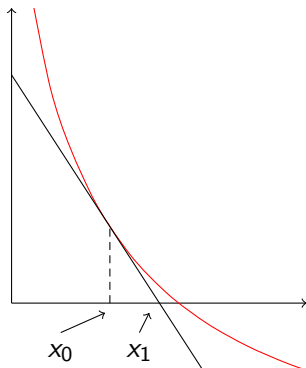
- Calcul de $b/a \rightarrow$ calcul de $1/a$ puis multiplication par b ;
- itération de Newton-Raphson

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

pour calculer une racine simple d'une fonction f ;

- on choisit $f(x) = 1/x - a$, ce qui donne

$$x_{k+1} = x_k(2 - ax_k).$$



Division par la méthode de Newton-Raphson

Itération

$$x_{k+1} = x_k(2 - ax_k).$$

- on montre facilement que

$$x_{k+1} - \frac{1}{a} = -a \left(x_k - \frac{1}{a} \right)^2$$

→ convergence **quadratique**.

- x_0 : division de «nombres machine» du processeur.

Un petit exemple

$$x_{k+1} = x_k(2 - ax_k).$$

Prenons $a = 3.141592718182845$, et $x_0 = 0.318$. On trouve

- $x_1 = 0.318309577966477982220$;
- $x_2 = 0.318309879638860441509973682150434139851 \dots$
- $x_3 = 0.3183098796391463465498711025172722970905142827638 \dots$
- $x_4 = 0.318309879639146346549871359316376150100879245428068 \dots$

Pour avoir n chiffres du quotient : il faut $\mathcal{O}(\log(n))$ itérations... et on a bien envie de dire que chaque itération a un coût $\mathcal{O}(M(n))$, où $M(n)$ est le coût de la multiplication de nombres de n chiffres.

Alors ?

Un petit tour de passe-passe...

Rappel :

$$x_{k+1} - \frac{1}{a} = -a \left(x_k - \frac{1}{a} \right)^2$$

- pour connaître x_k avec une précision de n chiffres, il suffit de connaître x_{k-1} avec une précision de $n/2$ chiffres ;
- pour connaître x_{k-1} avec une précision de $n/2$ chiffres, il suffit de connaître x_{k-2} avec une précision de $n/4$ chiffres ;
- etc. (en fait il faut être un peu plus soigneux que cela)

Seule la **dernière** itération se fait avec la précision «cible», celle d'avant se fait avec précision moitié, celle d'avant avec précision quart, etc.

Un petit tour de passe-passe...

- Temps $T_{inv}(n)$ du calcul de n chiffres d'un inverse :

$$T_{inv}(n) = 2 \left(M(n) + M\left(\frac{n}{2}\right) + M\left(\frac{n}{4}\right) + M\left(\frac{n}{8}\right) + \dots \right);$$

- nos algorithmes de multiplication sont à croissance au moins linéaire :
 $M(2p) \geq 2M(p)$

$$T_{inv}(n) \leq 2M(n) \left(1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \dots \right) = 4M(n),$$

et donc, pour la division

$$T_{div}(n) \leq 5M(n).$$

Et pour terminer...

- racine carrée : même truc que pour la division $\rightarrow$ coût $\mathcal{O}(M(n))$;
- exp, log, etc. : séries jusqu'à une certaine valeur de n , et au-delà
itération AGM :

$$\begin{aligned}a_{n+1} &= \frac{a_n + b_n}{2}, \\b_{n+1} &= \sqrt{a_n b_n}.\end{aligned}$$

Convergence très rapide vers $\pi/(2I(a_0, b_0))$, où

$$I(a, b) = \int_0^{\pi/2} \frac{dt}{\sqrt{a^2 \cos^2 t + b^2 \sin^2 t}},$$

puis utilisation du fait que, lorsque $s \rightarrow +\infty$

$$I(1, 4/s) = \ln(s) + \frac{4 \ln(s) - 4}{s^2} + o\left(\frac{1}{s^2}\right)$$

Et pour terminer...

- calcul de π : actuellement, records battus à l'aide de formules du type

$$\frac{1}{\pi} = 12 \sum_{k=0}^{+\infty} \frac{(-1)^k (6k)! (13591409 + 545140134k)}{(3k)! (k!)^3 640320^{3k+3/2}}.$$

- mais là encore on peut utiliser l'itération AGM...

Merci de votre attention !