INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Arithmétique en précision arbitraire

Paul Zimmermann

N° 4272

29 septembre 2001

THÈME 2

Rapport de recherche

ISSN 0249-6399 ISRN INRIA/RR--4272--FR

Arithmétique en précision arbitraire

Paul Zimmermann*

Thème 2 — Génie logiciel
et calcul symbolique
Projet SPACES, www-spaces.lip6.fr

Rapport de recherche n° 4272 — 29 septembre 2001 — 22 pages

Résumé : Cet article dresse un panorama des différents algorithmes disponibles pour effectuer des calculs arithmétiques sur des nombres entiers ou flottants. Après un bref rappel des diverses représentations possibles pour les nombres entiers en précision arbitraire, les différents algorithmes connus de multiplication, division, racine carrée, pgcd, lecture et écriture sont présentés, ainsi que leur complexité et leur domaine d'application. Pour chaque opération, sont décrits l'algorithme « naïf » et celui de meilleure complexité asymptotique connue, ainsi que des algorithmes intermédiaires de type « diviser pour régner » ayant souvent une large plage d'utilisation. Pour les calculs flottants, outre les opérations de base (multiplication, division, racine carrée), des méthodes générales sont décrites pour le calcul des fonctions algébriques, élémentaires, hypergéométriques, holonomes et spéciales.

Mots-clés : nombre entier, nombre flottant, multi-précision, algorithme de Karatsuba, Toom-Cook, transformée de Fourier rapide, méthode de Newton, arrondi exact.

* LORIA/INRIA Lorraine, zimmerma@loria.fr

Cet article a été écrit pour un numéro spécial de la revue « Calculateurs parallèles » sur l'arithmétique des ordinateurs, numéro édité par Arnaud Tisserand.

Arbitrary Precision Arithmetic

Abstract: This paper surveys the available algorithms for integer or floating-point arbitrary precision calculations. After a brief discussion about possible memory representations, known algorithms for multiplication, division, square root, greatest common divisor, input and output, are presented, together with their complexity and usage. For each operation, we present the naïve algorithm, the asymptotically optimal one, and also intermediate "divide and conquer" algorithms, which often are very useful. For floating-points computations, some general-purpose methods are presented for algebraic, elementary, hypergeometric and special functions.

Key-words: integer number, floating-point number, arbitrary precision, Karatsuba's algorithm, Toom-Cook's algorithm, Fast Fourier Transform, Newton's method, exact rounding.

Les calculs en précision arbitraire ont été rendus populaires par la chasse aux décimales de π. À part ces « calculs de l'extrême », on est souvent amené à effectuer des calculs avec une précision dépassant la capacité des processeurs actuels (53 bits soit environ 16 chiffres décimaux pour les calculs en virgule flottante, 32 à 64 bits soit 10 à 20 chiffres pour les calculs entiers).

Suivant la taille des nombres à manipuler, la connaissance *a priori* ou l'utilisation répétée d'un des opérandes, la taille mémoire disponible, plusieurs algorithmes peuvent être utilisés. Nous nous proposons ici de faire le tour des différents algorithmes connus, en indiquant pour chacun ses particularités et son domaine d'utilisation. Nous évoquerons uniquement les algorithmes *séquentiels*, la parallélisation de ces algorithmes étant un sujet de recherche en soi.

Parmi les critères considérés, le premier qui vient à l'esprit est la vitesse d'exécution. Celle-ci se mesure habituellement par la *complexité asymptotique* de l'algorithme considéré. Par exemple, on dira que l'algorithme naïf de multiplication est en $\mathcal{O}(n^2)$, ce qui signifie qu'il existe une constante C telle que le nombre d'opérations élémentaires pour multiplier deux nombres de n chiffres est au plus Cn^2 pour n assez grand. Il est ici inutile de préciser de quel type d'opération élémentaire il s'agit, et on peut remplacer « chiffres » par « bits », puisque cela revient à changer la constante C.

La notation $\mathcal{O}(\cdot)$ ne suffit pas toujours pour départager deux méthodes concurrentes. Nous verrons ainsi que la méthode de Newton et la division récursive, utilisées avec la multiplication de Karatsuba, sont toutes deux en $\mathcal{O}(n^{\log_2 3})$. Déterminer la meilleure méthode nécessite d'expliciter la constante se « cachant » sous la notation $\mathcal{O}(\cdot)$. Cela impose de préciser les opérations élémentaires considérées et ce que représente n — chiffres ou bits — ou bien de calculer des constantes relatives par rapport à une opération de référence.

Dans tout cet article, nous considérons qu'une opération élémentaire correspond à une instruction sur un mot-machine [1] et a donc un coût unitaire borné. Cette hypothèse, relativement bien vérifiée pour de petits calculs, ne l'est plus pour ceux utilisant beaucoup de mémoire, où des phénomènes de mémoire cache (L1, L2, ...) ou de mémoire externe (*swap*) viennent perturber les opérations. Même pour un petit calcul, le fonctionnement en pipeline des processeurs actuels rend difficilement prévisible le nombre exact de cycles utilisés par une suite donnée d'instructions en langage assembleur.

1. Calcul entier

1.1. Représentations logique et physique.

1.1.1. *Représentation logique.* On distingue essentiellement deux types de représentation des nombres (entiers ou flottants) en précision arbitraire. La représentation *dense* représente tous les chiffres d'un nombre dans une base β : l'entier 830538 est codé $[8,3,0,5,3,8]$ en base 10. La représentation *creuse* ne représente que les chiffres non nuls : cet entier sera codé $[(8,5),(3,4),(5,2),(3,1),(8,0)]$.

Dans une représentation *redondante*, on tolère des chiffres dans un intervalle de longueur supérieure à $\beta - 1$, par exemple $[0, 2\beta - 2]$. Cette représentation est principalement utilisée dans les processeurs ou pour les calculs parallèles, puisqu'elle retarde le problème de la propagation des retenues. Ainsi en binaire, l'addition de $59 = (111011)_2$ et de $37 = (100101)_2$ donne $(211112)_2$, qui se simplifie en $(1100000)_2$. Cependant, la comparaison et la division deviennent non triviales. On utilise aussi les chiffres 0, 1 et -1 en base $\beta = 2$ (codage de Booth).

Au lieu de représenter un entier dans la base $\beta^0, \beta^1, \beta^2, \ldots$, un *système modulaire* stocke les restes de sa division euclidienne par des entiers premiers entre eux $\beta_0, \beta_1, \beta_2, \ldots$ La reconstruction se fait via le théorème des restes chinois. J.-C. Bajard et L.-S. Didier font un tour d'horizon de ce type de représentation dans [2].

1. Addition, soustraction, multiplication, division, « et » logique, « ou » logique, décalage, « ou » exclusif, complément à deux.

1.1.2. *Représentation physique.* En ce qui concerne le stockage physique des chiffres en machine, de nombreuses variantes sont possibles. On peut stocker les chiffres dans des registres entiers (comme les bibliothèques Pari [3] et GNU MP [22]) ou dans des registres flottants (comme Arithmos [17] par défaut, ou la bibliothèque LIP). Les registres flottants permettent des opérations plus rapides sur certains processeurs ; les registres entiers quant à eux offrent un stockage plus compact. On peut même envisager un stockage physique dans des registres entiers, et des calculs intermédiaires utilisant des registres flottants. Pour simplifier, on parlera de *mot-machine*, désignant soit un registre entier, soit un registre flottant.

Le choix de la base β est également important. Pour une utilisation optimale de la mémoire, on prendra la plus grande base possible. Ce n'est pas le choix fait par le système de calcul formel Maple, qui utilise $\beta = 10^4$ sur un processeur 32 bits et $\beta = 10^9$ sur un processeur 64 bits ; cela double l'espace mémoire utilisé, ce qui s'avère très pénalisant pour de gros calculs.

Nous nous intéressons par la suite uniquement aux représentations denses non redondantes, indépendamment du mode de stockage physique choisi. Sauf mention explicite, la base β correspond au mot-machine, c'est-à-dire que le plus grand nombre pouvant être stocké dans un mot-machine égale $\beta - 1$.

1.2. **Multiplication.** Lorsque l'un des opérandes d'une multiplication est connu *a priori*, des méthodes spécifiques existent à base d'additions, soustractions et décalages ; on parle alors de « multiplication par une constante ». Par exemple, pour calculer $17x$, on commence par calculer $16x$ par un décalage de 4 bits, puis on ajoute x. Plusieurs auteurs ont mis au point des algorithmes efficaces de multiplication par une constante [38].

Lorsque les opérandes ne sont pas constants, les principaux algorithmes sont la multiplication naïve en $\mathcal{O}(n^2)$, l'algorithme de Karatsuba en $\mathcal{O}(n^{\log_2 3})$, celui de Toom-Cook en $\mathcal{O}(n^{\log_3 5})$ et ses variantes, enfin l'algorithme de Schönhage-Strassen en $\mathcal{O}(n \log n \log \log n)$. Il est crucial de toujours choisir le meilleur algorithme de multiplication entière en fonction de la taille des opérandes, car comme nous le verrons par la suite, les autres opérations, aussi bien entières que flottantes, s'y ramènent.

Certains algorithmes rapides de multiplication — comme celui de Karatsuba et de Toom-Cook — nécessitent pour une bonne efficacité que les deux opérandes soient de même taille. Dans le cas contraire, on découpe usuellement le plus grand nombre en « morceaux » de la taille du plus petit, afin d'effectuer des produits équilibrés. Ainsi le cas le pire pour l'algorithme de Karatsuba est lorsque les deux opérandes ont des tailles qui sont deux nombres de Fibonacci consécutifs. Les algorithmes à base de transformée de Fourier rapide souffrent dans une moindre mesure d'un déséquilibre des entrées, le paramètre essentiel étant pour eux la taille du produit. En pratique, la plupart des implantations se contentent de déterminer les frontières entre les différents algorithmes de multiplication pour des entrées de même taille.

1.2.1. *Multiplication naïve.* La multiplication naïve consiste simplement à multiplier chiffre par chiffre : $A = \sum_i a_i \beta^i$ et $B = \sum_j b_j \beta^j$ étant les deux multiplicandes, leur produit est :

$$A \cdot B = \sum_k \left(\sum_{i+j=k} a_i b_j \right) \beta^k. \tag{1}$$

Si A et B ont respectivement n et m chiffres en base β, le coût est nm multiplications entre deux mots-machine, et quasiment autant d'additions. La bonne façon de considérer la multiplication naïve est : $A \cdot B = \sum_i (a_i B) \beta^i$. Ainsi l'opération de base est la multiplication d'un grand entier (B) par un mot-machine (a_i), avec accumulation du résultat dans une troisième variable :

Multiplication naïve.
Entrée : $A = [a_{n-1}, \ldots, a_0]$, $B = [b_{m-1}, \ldots, b_0]$

Sortie : $[c_{n+m-1}, \ldots, c_0]$ contient $A \cdot B$
$[c_m, \ldots, c_0] \leftarrow a_0 \cdot B$
for j **from** 1 **to** $n-1$ **do**
    $[c_{j+m}, \ldots, c_j] \leftarrow [0, c_{j-1+m}, \ldots, c_j] + a_j \cdot B$

1.2.2. *Algorithme de Karatsuba.* L'algorithme de Karatsuba [31] se comprend mieux sur les polynômes. Si $a_1X + a_0$ et $b_1X + b_0$ sont deux polynômes de degré 1, leur produit se calcule habituellement en 4 multiplications de coefficients, alors que la formule

$$a_1b_1X^2 + [(a_1 + a_0)(b_1 + b_0) - a_1b_1 - a_0b_0]X + a_0b_0$$

n'utilise que 3 produits : a_1b_1, a_0b_0 et $(a_1 + a_0)(b_1 + b_0)$. En remplaçant X par $\beta^{n'}$, et en utilisant récursivement cette identité, on obtient l'algorithme suivant :

Algorithme de Karatsuba.
Entrée : deux entiers $0 \le A, B < \beta^n$
Sortie : le produit $A \cdot B$
1 si $n < n_0$ utiliser la multiplication naïve
2 $n' \leftarrow \lceil n/2 \rceil$
3 écrire $A = a_1\beta^{n'} + a_0, B = b_1\beta^{n'} + b_0$ avec $0 \le a_i, b_i < \beta^{n'}$
4 $H \leftarrow$ `Karatsuba`(a_1, b_1)
5 $L \leftarrow$ `Karatsuba`(a_0, b_0)
6 $M \leftarrow$ `Karatsuba`$(a_0 + a_1, b_0 + b_1)$
7 $M' \leftarrow M - L - H$
8 $A \cdot B = H \cdot \beta^{2n'} + M' \cdot \beta^{n'} + L$

(Les divisions par $\beta^{n'}$ à l'étape 3 se font directement en lisant les valeurs de a_1 et a_0 sur la représentation dense de A ; idem pour les multiplications par $\beta^{2n'}$ et $\beta^{n'}$ à l'étape 8.)

La complexité asymptotique de cet algorithme est $\mathcal{O}(n^{\log_2 3})$, avec $\log_2 3 \sim 1.585$. La limite n_0 à partir de laquelle l'algorithme de Karatsuba est plus rapide que la méthode naïve dépend de la vitesse relative de l'addition par rapport à la multiplication de mots-machine ; n_0 sera d'autant plus basse que l'addition est plus rapide.

La description ci-dessus ne précise pas où sont stockés les résultats intermédiaires ; de plus, lorsque n est impair, la mémoire nécessaire pour L et H n'est pas la même. Pour une implantation efficace, il est primordial de veiller à la gestion de la mémoire. Si A et B sont stockés sur n mots-machine, et qu'un espace mémoire de $2n$ mots est pré-alloué pour le résultat $A \cdot B$, on peut montrer qu'un espace mémoire auxiliaire de $2n + \mathcal{O}(1)$ mots suffit pour tout l'algorithme [39].

Dans le cas entier, les sommes $a_0 + a_1$ et $b_0 + b_1$ peuvent excéder $\beta^{n'}$ et produire une retenue, ce qui impose des additions supplémentaires. Une solution est d'utiliser l'identité ci-dessous, où $s_a = |a_1 - a_0|$, $s_b = |b_1 - b_0|$ et $\epsilon s_a s_b = (a_1 - a_0)(b_1 - b_0)$. Ainsi, les différences s_a et s_b ne produisent pas de retenue :

$$(a_1X + a_0)(b_1X + b_0) = a_1b_1X^2 + (-\epsilon s_a s_b + a_1b_1 + a_0b_0)X + a_0b_0.$$

Une autre amélioration possible concerne le nombre d'additions (ou soustractions) effectuées. La variante présentée ci-dessus effectue après les trois appels récursifs $6n'$ additions de mots-machine ($2n'$ pour retrancher L à M, $2n'$ pour retrancher H, enfin $2n'$ à nouveau pour ajouter $M'\beta^{n'}$ à $H\beta^{2n'} + L$). Le même calcul s'effectue avec $5n'$ additions [46] : si l'on décompose $L = l_1\beta^{n'} + l_0$, $H = h_1\beta^{n'} + h_0$, $M = m_1\beta^{n'} + m_0$, alors le résultat final s'écrit :

$$h_1\beta^{3n'} + h_0 + m_1 - l_1 - h_1)\beta^{2n'} + (l_1 + m_0 - l_0 - h_0)\beta^{n'} + l_0.$$

Les deux termes intermédiaires s'obtiennent en 5 additions ou soustractions de nombres de taille n' de la façon suivante : calculer $\alpha = l_1 - h_0$, puis $m_0 - l_0 + \alpha$ et $m_1 - h_1 - \alpha$.

1.2.3. *Algorithme de Toom-Cook.* L'algorithme de Karatsuba découpe les opérandes en deux ; une meilleure efficacité est obtenue en les découpant en trois : c'est l'algorithme de Toom-Cook, de complexité asymptotique $\mathcal{O}(n^{1.465})$. Considérons deux polynômes $A = a_2X^2 + a_1X + a_0$ et $B = b_2X^2 + b_1X + b_0$, de produit

$$C = c_4X^4 + c_3X^3 + c_2X^2 + c_1X + c_0.$$

Grâce à la formule d'interpolation de Lagrange, il suffit de connaître la valeur de C en 5 points distincts pour le déterminer entièrement. Or la valeur de C en un point n'est autre que le produit des valeurs de A et B. On peut choisir ces points de sorte que les formules donnant les c_i soient les plus simples possibles ; ces formules s'obtiennent facilement à partir de n'importe quel système de calcul formel en résolvant le système linéaire donnant les valeurs de C en fonction de celles de A et B. Par exemple, pour les points d'interpolation $0, \infty, 1, -1, 2$, on obtient avec Maple :

```
> C := x -> c4*x^4 + c3*x^3 + c2*x^2 + c1*x + c0:
> solve({C(0) = AB(0), c4 = a2*b2, C(1) = AB(1), C(-1) = AB(-1), C(2) = AB(2)},
        {c0, c1, c2, c3, c4});
{c4 = a2 b2, c0 = AB(0), c2 = -a2 b2 - AB(0) + 1/2 AB(-1) + 1/2 AB(1),

    c3 = -2 a2 b2 + 1/2 AB(0) - 1/6 AB(-1) - 1/2 AB(1) + 1/6 AB(2),

    c1 = 2 a2 b2 - 1/2 AB(0) - 1/3 AB(-1) + AB(1) - 1/6 AB(2)}
```

Une fois déterminés le produit a_2b_2 des coefficients dominants de A et B — correspondant à l'évaluation à l'infini — et les produits des valeurs en $0, 1, -1, 2$, quelques additions ainsi que des divisions — exactes — par 2 et 3 suffisent.

La méthode de Toom-Cook s'étend à la découpe en n'importe quel nombre de parties : si l'on découpe en r parties, cela correspond au produit de deux polynômes de degré $r-1$, dont le produit de degré $2r-2$ peut être déterminé par interpolation en $2r-1$ points. La complexité asymptotique est donc $\mathcal{O}(n^{\frac{\log(2r-1)}{\log r}})$. Les algorithmes de Karatsuba et de Toom-Cook correspondent respectivement à $r = 2$ et $r = 3$. Le lecteur trouvera une discussion détaillée des cas $r = 2, 3, 4$ dans [57].

1.2.4. *Algorithmes utilisant la transformée de Fourier rapide.* Les variantes de l'algorithme de Toom-Cook permettent d'atteindre une complexité en $\mathcal{O}(n^{1+\epsilon})$ pour ϵ aussi petit que voulu ; néanmoins la mise en œuvre devient plus difficile lorsque r augmente, car il faut implanter « en dur » la résolution du système linéaire à $2r-1$ équations. La constante devant le $\mathcal{O}(\cdot)$ devient ainsi importante, et la plupart des implantations se limitent à $r = 2$ ou 3.

Une meilleure complexité s'obtient grâce à la transformée de Fourier rapide (*Fast Fourier Transform* ou FFT). Deux variantes sont principalement utilisées, toutes deux dues à Schönhage et Strassen [49] :

- l'algorithme sur le corps des complexes de complexité asymptotique $\mathcal{O}(n \log n^2)$ ou pour la version récursive $\mathcal{O}(n \log n \log\log n \log\log\log n \ldots)$. L'implantation utilise les nombres flottants de la machine, en simple ou double précision, qui procurent une bonne efficacité. L'inconvénient majeur est que, basées sur une arithmétique inexacte, peu d'implantations sont réellement prouvées.
- l'algorithme calculant modulo $2^m + 1$, de complexité asymptotique $\mathcal{O}(n \log n \log\log n)$. Son inconvénient est un coût plus élevé pour de petites tailles, les processeurs actuels étant moins optimisés pour les calculs entiers, mais il n'est pas limité par la précision des flottants machines, et en outre nécessite moins de mémoire.

Le second algorithme fournit en fait une famille de méthodes : FFT-k ramène une multiplication de taille n à 2^k multiplications de taille $n/2^{k-2}$, soit une complexité asymptotique $\mathcal{O}(n^{k/(k-2)})$. Cet algorithme ne peut donc supplanter la méthode naïve que pour $k \geq 5$, l'algorithme de Karatsuba pour $k \geq 6$, et celui de Toom-Cook pour $k \geq 7$. Avec GMP sur Pentium II, FFT-5 est plus rapide que la méthode naïve à partir de 480 mots-machine, soit environ 4600 chiffres décimaux, FFT-7 supplante l'algorithme de Karatsuba à partir de 2624 mots, soit environ 25000 chiffres décimaux, et il faut aller jusqu'à 5888 mots soit environ 57000 chiffres pour battre l'algorithme de Toom-Cook avec FFT-8.

La multiplication via FFT ne sert pas qu'à calculer des décimales de π : G. Woltman l'utilise dans son programme de recherche de nombres premiers de la forme $2^p - 1$ [55], et dans son programme de factorisation par la méthode des courbes elliptiques, qui effectue une multiplication modulo $2^{727} - 1$ (entier de 219 chiffres sans facteur connu) en environ $4.3\mu s$ sur un Pentium II à 366Mhz.

1.2.5. *Carré.* Quel facteur peut-on gagner dans le calcul de A^2 par rapport à celui de AB ? Pour la multiplication naïve, on peut espérer au mieux un facteur 2, en remplaçant les n^2 produits de mots-machine de [1] par $n(n+1)/2$ produits.

Supposons que l'on passe d'un algorithme de multiplication en cn^α à l'algorithme de Toom-Cook d'ordre r, supposé plus rapide. La frontière entre les deux méthodes est définie par le fait que le coût cn^α de la première devient aussi grand que celui de la méthode de Toom-Cook, soit $2r - 1$ multiplications de nombres de taille n/r plus un coût proportionnel à n pour la résolution du système linéaire — essentiellement des additions —, d'où $cn^\alpha \sim (2r-1)c(n/r)^\alpha + kn$. Cela donne comme valeur de la frontière :

$$n_0 = \left(\frac{k/c}{(1-(2r-1)/r^\alpha)}\right)^{\frac{1}{\alpha-1}}. \tag{2}$$

Pour $\alpha = 2$, cette formule montre que la frontière entre la méthode naïve et un algorithme plus rapide, quel qu'il soit, est directement proportionnelle au rapport k/c entre addition et multiplication de mots-machine.

Une autre conséquence de la formule [2] est que si l'on améliore en c' la constante c de la première méthode, la valeur de n_0 sera multipliée par un facteur $(c/c')^{1/(\alpha-1)}$. Cette remarque vaut aussi si on substitue la constante c de la multiplication par celle c' du carré, car le coût linéaire k ne varie pas. Par conséquent, la frontière n_1 entre carré naïf et par l'algorithme de Karatsuba est deux fois plus grande que celle pour la multiplication, en théorie du moins car on a négligé les termes constants. Cela est relativement bien vérifié en pratique, puisqu'on obtient avec GMP un rapport 2.26 sur Pentium II, et 2.13 sur Alpha EV6 (cf. Tableau 2 p. 18).

Une fois connues les frontières pour la multiplication (n_0) et le carré (n_1), on peut déterminer le rapport des constantes correspondantes. On a en effet $cn_0^\alpha = ln_0^\beta$ et $c'n_1^\alpha = l'n_1^\beta$, où ln^β est le coût d'une multiplication pour $n \geq n_0$, et $l'n^\beta$ celui d'un carré pour $n \geq n_1$, donc :

$$\frac{l'}{l} = \frac{c'}{c}\left(\frac{n_1}{n_0}\right)^{\alpha-\beta} = \left(\frac{c'}{c}\right)^{\frac{\beta-1}{\alpha-1}}. \tag{3}$$

Pour l'algorithme de Karatsuba utilisé après la méthode naïve ($c'/c = 1/2$), on obtient $k'/k = 2/3$. Cela corrobore le rapport expérimental $\frac{26.64}{46.2} = 0.58$ trouvé par Zuras [57], et les rapports compris entre 0.60 et 0.75 obtenus avec GMP de 52 à 138 mots sur Pentium II (frontières naïf-Karatsuba pour le carré et Karatsuba-Toom-Cook pour la multiplication). Pour l'algorithme de Toom-Cook utilisé après Karatsuba ($c'/c = 2/3$), la formule [3] donne un rapport théorique ~ 0.724, là encore proche de celui trouvé expérimentalement par Zuras, soit $\frac{64.59}{90.15} \sim 0.716$. Avec GMP sur Pentium II, le rapport varie entre 0.64 et 0.77 de 200 à 5000 mots.

Pour la FFT, il n'est plus possible de déterminer de rapport théorique entre carré et produit, car cela dépend à la fois du rapport des transformées de Fourier elles-mêmes, et de celui des multiplications terme-à-terme. Cependant, le premier rapport valant $2/3$ — deux transformées de Fourier pour un carré, contre trois pour une multiplication — et le second étant proche de $2/3$ pour Karatsuba et Toom-Cook, il en sera de même pour le rapport pondéré.

1.3. **Division.** Comme la multiplication, la division par une constante bénéficie de méthodes spécifiques (travaux de J. Vuillemin et J.-M. Muller), dont nous ne parlerons pas ici.

Soit à diviser un entier A de $n+m$ mots-machine par un diviseur B de n mots-machine, donnant un quotient Q de taille m ou[2] $m+1$, et un reste de taille n. Si $m > n$, on commence par diviser les $2n$ mots de poids fort du dividende par le diviseur, ce qui donne les n mots de poids fort du quotient, et un nouveau dividende de taille inférieure ; on continue jusqu'à ce que le dividende soit de taille $\leq 2n$. Si au contraire $m < n$, on divise les $2m$ mots de poids fort du dividende par les m mots de poids fort du diviseur, ce qui donne une approximation Q' du quotient, que l'on corrige éventuellement si le reste $A - Q'B$ est négatif.

L'opération de base est donc la division d'un entier de taille $2n$ par un diviseur de taille n, donnant un quotient et un reste de taille n. Nous présentons dans cette section trois classes de méthodes : la division naïve et une de ses variantes en $\mathcal{O}(n^2)$, la division récursive, combinée avantageusement à la multiplication de Karatsuba ou de Toom-Cook, et la méthode de Newton de même complexité asymptotique $\mathcal{O}(n \log n \log\log n)$ que la multiplication. Comme on le verra, les constantes cachées sous la notation $\mathcal{O}(\cdot)$ sont plus importantes que pour la multiplication, ce qui explique que la division de grands nombres coûte plus cher qu'une multiplication, surtout lorsque seule la division naïve est utilisée, comme c'est le cas pour Maple !

1.3.1. *Division naïve classique.* Soit à diviser $A = a_{2n-1}\beta^{2n-1} + \cdots + a_1\beta + a_0$ par $B = b_{n-1}\beta^{n-1} + \cdots + b_1\beta + b_0$ avec $0 \leq a_i, b_j < \beta$ où β est la base-machine. On suppose en général que B est *normalisé*, c'est-à-dire que le mot de poids fort b_{n-1} vérifie $\beta/2 \leq b_{n-1} < \beta$. Si ce n'est pas le cas, on multiplie B par la puissance de 2 idoine pour qu'il soit normalisé, ce qui donne $A = Q\cdot(2^k B) + R$, et il suffit ensuite de diviser R par B.

On suppose aussi que le quotient $\lfloor A/B \rfloor$ est représentable sur n mots-machine, soit $A < \beta^n B$. B étant normalisé, on a $A < \beta^{2n} \leq 2B\beta^n$, donc si $A < \beta^n B$ n'est pas réalisé, il suffit de retrancher $\beta^n B$ à A pour qu'elle le devienne ; on ajoute alors β^n au quotient final, qui doit donc pouvoir stocker un bit « de retenue » en plus. Cette variante est nécessaire pour la division récursive présentée plus loin. On procède alors de la façon suivante :

Division naïve.
Entrée : $A = \sum_0^{2n-1} a_i\beta^i$, $B = \sum_0^{n-1} b_j\beta^j$ normalisé, $A < \beta^n B$
Sortie : quotient $Q = \sum_0^{n-1} q_j\beta^j$, reste dans A

1 **for** j **from** $n-1$ **downto** 0 **do** *Invariant :* $A < \beta^{j+1}B$
2 $\quad q_j^* \leftarrow \lfloor (a_{n+j}\beta + a_{n+j-1})/b_{n-1} \rfloor$
3 $\quad q_j \leftarrow \min(q_j^*, \beta - 1)$
4 $\quad A \leftarrow A - q_j B\beta^j$
5 $\quad$ **while** $A < 0$ **do**
6 $\quad\quad q_j \leftarrow q_j - 1$
7 $\quad\quad A \leftarrow A + B\beta^j$

(Les coefficients a_{n+j} et a_{n+j-1} pour le calcul de q_j^* à l'étape 2 sont ceux correspondant à la valeur courante de $A = a_{n+j}\beta^{n+j} + a_{n+j-1}\beta^{n+j-1} + \cdots$). Considérons d'abord le cas où $q_j = q_j^* \leq \beta - 1$. Comme $q_j b_{n-1} \geq a_{n+j}\beta + a_{n+j-1} - b_{n-1} + 1$,

$$A - q_j B\beta^j \leq (b_{n-1} - 1)\beta^{n+j-1} + (A \bmod \beta^{n+j-1}),$$

ce qui assure que la nouvelle valeur de a_{n+j} est nulle, et $a_{n+j-1} < b_{n-1}$. Par contre il se peut que A devienne négatif, mais comme $q_j b_{n-1} \leq a_{n+j}\beta + a_{n+j-1}$:

$$A - q_j B\beta^j > (a_{n+j}\beta + a_{n+j-1})\beta^{n+j-1} - q_j(b_{n-1}\beta^{n-1} + \beta^{n-1})\beta^j \geq -q_j\beta^{n+j-1}.$$

2. La division de l'entier 3 sur $n+m = 2$ bits par l'entier 1 sur $n = 1$ bit donne un quotient sur $m+1 = 2$ bits.

Par conséquent, $A - q_j B\beta^j + 2B\beta^j \geq (2b_{n-1} - q_j)\beta^{n+j-1} > 0$, ce qui prouve que la boucle **while** est exécutée au plus deux fois [34, Théorème 4.3.1.B]. (C'est ici que le fait que B soit normalisé est crucial ; si on suppose uniquement que b_{n-1} est non nul, le nombre de passages dans la boucle **while** peut aller jusqu'à β !) Lorsque la correction $A \leftarrow A + B\beta^j$ est effectuée, *i.e.* lorsque A est négatif à l'étape 5, on a après la boucle **while** $A < \beta^j B$, donc l'invariant de la boucle sur j est bien respecté.

Si le quotient estimé q_j^*, qui vaut au plus $\beta+1$, est supérieur ou égal à β, on choisit $q_j = \beta - 1$. Il suffit de vérifier dans ce cas l'invariant $A < \beta^j B$ lorsqu'on ne passe pas par la boucle **while**, sinon le même argument que ci-dessus s'applique. Avant la boucle **while**, on a $A < \beta^{j+1}B - (\beta-1)B\beta^j = \beta^j B$, donc l'invariant est respecté.

L'algorithme **Division naïve** nécessite n divisions de deux mots par un mot, et n produits d'un nombre de n mots par un mot, produits qui sont soustraits au dividende A. Il est clair que pour n grand, les divisions de coût $\mathcal{O}(n)$ sont négligeables par rapport aux produits de coût total $\Omega(n^2)$. Cependant pour les processeurs ayant une division entière lente, et pour de petites tailles, le calcul des quotients approchés q_j^* peut s'avérer plus coûteux que celui des produits $q_j B$! Une solution est alors d'utiliser la méthode suivante, également en $\mathcal{O}(n^2)$, mais avec moins de divisions entières.

1.3.2. *Division naïve avec prétraitement du diviseur.* Cette méthode s'explique plus facilement sur un exemple. Soit à diviser $A = 766970544842443844$ par $B = 862664913$, chaque mot-machine pouvant stocker trois chiffres décimaux, c'est-à-dire $\beta = 1000$, $n = 3$ avec les notations ci-dessus. L'algorithme classique opère de la façon suivante :

j	A	q_j	$A - q_j B\beta^j$	après correction
2	766 970 544 842 443 844	889	61 437 185 443 844	idem
1	61 437 185 443 844	071	187 976 620 844	idem
0	187 976 620 844	218	−84 330 190	778 334 723

ce qui donne le quotient $Q = 889071217$ et le reste $R = 778334723$.

La division naïve avec prétraitement du diviseur [51] commence par calculer le plus petit multiple de B supérieur ou égal à β^{n+1}. Dans notre exemple c'est $1160 \cdot B = 1000691299080$. Ce multiple s'écrit $B' = \beta^{n+1} + B''$ avec $B'' < B < \beta^n$. Cela revient à fixer $b_{n-1} = \beta$ dans la version classique, cas où le calcul du quotient estimé q_j est trivial, puisqu'il suffit de lire le mot a_{n+j} de poids fort de A :

j	A	q_j	$A - q_j B'\beta^j$	après correction
2	766 970 544 842 443 844	766	441 009 747 163 844	idem
1	441 009 747 163 844	441	−295 115 730 436	705 575 568 644

Cela donne $A = 766440 \cdot (1160B) + 705575568644$. Il ne reste plus qu'à diviser le dernier reste 705575568644 par B, et à multiplier 766440 par 1160, ce qui donne $705575568644 = 817 \cdot B + 778334723$, et $Q = 766440 \cdot 1160 + 817 = 889071217$.

Cette variante n'effectue que deux divisions de deux mots par un mot au lieu de n : une pour déterminer B' et une pour diviser le dernier reste par B. Elle s'avère encore plus intéressante si le même diviseur B est utilisé plusieurs fois, auquel cas B' peut être sauvegardé.

1.3.3. *Division récursive.* L'algorithme **Division naïve** s'écrit pour $n = 2$ de la façon suivante, où la fonction `DivRem` retourne quotient et reste de la division euclidienne d'un dividende de deux mots par un diviseur normalisé d'un mot (les quotients q_1, q_0 et Q peuvent contenir un bit de retenue, voir la remarque à propos de la division naïve).

Entrée : $A = a_3\beta^3 + a_2\beta^2 + a_1\beta + a_0$, $B = b_1\beta + b_0$ avec $b_1 \geq \beta/2$
Sortie : quotient Q et reste R de la division de A par B
$(q_1, r_1) \leftarrow \texttt{DivRem}(a_3\beta + a_2, b_1)$
$A \leftarrow r_1\beta^2 + a_1\beta + a_0 - q_1 b_0 \beta$
while $A < 0$ **do** $q_1 \leftarrow q_1 - 1, A \leftarrow A + B\beta$ **end**
$(q_0, r_0) \leftarrow \texttt{DivRem}(a_2'\beta + a_1', b_1)$ où $A := a_2'\beta^2 + a_1'\beta + a_0'$

5 $A \leftarrow r_0\beta + a'_0 - q_0 b_0$
6 **while** $A < 0$ **do** $q_0 \leftarrow q_0 - 1, A \leftarrow A + B$ **end**
7 $Q = q_1\beta + q_0, R = A$

Cet algorithme fonctionne récursivement en remplaçant β par β^n. La division de $4n$ mots par $2n$ mots nécessite alors deux divisions récursives de $2n$ mots par n mots, et deux multiplications $n \times n$, à savoir $q_1 b_0$ à l'étape 2 et $q_0 b_0$ à l'étape 5.

Si $D(n)$ désigne la complexité de la division $2n$ par n, et $M(n)$ celle de la multiplication $n \times n$, on obtient la récurrence suivante :

$$D(2n) \sim 2D(n) + 2M(n),$$

dont la solution est $D(n) \sim \frac{1}{2^{\alpha-1}-1} M(n)$ si $M(n) = \Theta(n^\alpha)$ avec $\alpha > 1$. Pour $\alpha = \log_2 3$ (Karatsuba), cela donne $D(n) \sim 2M(n)$, pour $\alpha = \log_3 5$ (Toom-Cook), cela donne $D(n) \sim 2.63M(n)$. Si par contre $M(n) = \mathcal{O}(n \log^k n)$, un facteur multiplicatif de $\log n$ rend la division récursive plus lente que la méthode de Newton.

Cet algorithme de division récursive a été évoqué par Moenck et Borodin [41], mais uniquement en combinaison avec une multiplication par FFT, cas où il n'est pas intéressant. Jebelean a montré dans [30] son intérêt avec l'algorithme de Karatsuba, enfin Burnikel et Ziegler ont exprimé dans [13] cet algorithme sous une forme très simple, similaire à celle ci-dessus. M. Quercia en donne dans [47] une description détaillée.

1.3.4. *Méthode de Newton.* La méthode de Newton est un procédé général pour trouver une racine d'une fonction $f(x)$. Partant d'une valeur initiale x_0, on calcule les valeurs successives $x_{k+1} = x_k - f(x_k)/f'(x_k)$, qui convergent vers une racine de f si x_0 est bien choisi. La convergence est quadratique, c'est-à-dire que le nombre de chiffres corrects double à chaque étape. De plus, la méthode de Newton est auto-correctrice : pour obtenir $2n$ chiffres corrects pour x_{k+1}, il suffit que les n chiffres les plus significatifs de x_k soient corrects, par conséquent le calcul de x_k peut s'effectuer avec n chiffres seulement, celui de x_{k-1} avec $n/2$ chiffres, ... Si n chiffres de $f(x_k)/f'(x_k)$ se calculent en $\mathcal{O}(T(n))$, le calcul total est également en $\mathcal{O}(T(n))$, dès lors que T est au moins linéaire en n. C'est en particulier le cas des fonctions algébriques, où f et f' sont des polynômes, et s'évaluent donc en $\mathcal{O}(M(n))$ opérations.

Pour la division, on utilise la méthode de Newton pour calculer une approximation X de β^n/B, puis une approximation Q du quotient est obtenue via XA div β^n, enfin le reste est déterminé par une multiplication supplémentaire via $A - QB$.

Partant d'une approximation x_k des k mots de poids fort de β^n/B, l'itération :

$$x_{2k} = x_k\beta^k + x_k(\beta^{2k} - B_{2k}x_k \text{ div } \beta^k) \text{ div } \beta^k$$

fournit une approximation sur $2k$ mots, B_{2k} représentant les $2k$ mots de poids fort de B. Les deux opérations coûteuses sont le produit $B_{2k}x_k$ et celui de x_k par $\beta^{2k} - B_{2k}x_k$ div β^k, ce dernier terme ayant au plus k mots non nuls, car x_k est une approximation d'ordre k de β^n/B. La complexité $I(n)$ de calcul d'un inverse sur n mots vérifie donc :

$$I(n) \sim I(n/2) + 3M(n/2),$$

soit $I(n) \sim 3M(n)$. Le calcul du quotient Q coûte donc $4M(n)$, et la division avec reste $5M(n)$.

Karp et Markstein ont publié dans [33] une astuce permettant de calculer le quotient en $\frac{7}{2}M(n)$ opérations, et donc le reste en $\frac{9}{2}M(n)$. L'idée est d'incorporer le dividende lors de la dernière itération de Newton : étant donnée une approximation $x_{n/2}$ sur $n/2$ mots de β^n/B, on calcule $y_{n/2} = A_{n/2}x_{n/2} \text{ div} \beta^{n/2}$, puis

$$y_n = y_{n/2}\beta^{n/2} + x_{n/2}(A_n - B_n y_{n/2} \text{ div } \beta^{n/2}) \text{ div } \beta^{n/2}.$$

Le meilleur algorithme actuellement connu pour la division avec reste est donc asymptotiquement 4.5 fois plus lent qu'une multiplication ! Peut-on faire mieux ? Lorsque seul le reste est voulu, est-il possible de le calculer plus rapidement ? Ces questions demeurent ouvertes.

On retrouve la formule de Newton — aussi appelée méthode de la tangente — en cherchant l'intersection de la tangente au point x avec l'axe des abscisses : $f(x) + hf'(x) = 0$ d'où $h = -f(x)/f'(x)$. On obtient une convergence encore plus rapide en considérant des développements de Taylor d'ordre plus élevé, par exemple l'itération cubique :

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

Le lecteur désireux d'en savoir plus se référera à l'essai *Newton's iteration* du site de Gourdon et Sebah [21], qui dresse un historique de l'itération de Newton, et présente une méthode originale pour obtenir d'autres formules. Gourdon et Sebah donnent notamment l'itération d'ordre 5 suivante pour l'inverse :

$$h_k = 1 - Ax_k, \quad x_{k+1} = x_k + x_k(1 + h_k^2)(h_k + h_k^2),$$

qui grâce à la factorisation $1 + h + h^2 + \cdots + h^{2^m-1} = (1+h)(1+h^2)\cdots(1+h^{2^{m-1}})$, se généralise à des itérations d'ordre $2^m + 1$.

Si l'on autorise une utilisation mémoire plus que linéaire, des méthodes plus rapides existent ; par exemple un algorithme dû à J. van der Hoeven [52] permet d'effectuer une division de $2n$ mots par n mots avec la même complexité qu'une multiplication $n \times n$ par Karatsuba, moyennant un espace mémoire en $\mathcal{O}(n \log n)$.

1.3.5. *Division exacte.* Lorsque le reste d'une division est *a priori* nul, on parle de « division exacte ». Des algorithmes spécifiques existent alors, plus rapides que la division avec reste :

- si le quotient est exact, il n'est pas nécessaire de mettre à jour les chiffres de poids faible du dividende qui donneront le reste final, puisqu'on sait qu'il sera nul ;
- si le quotient $Q = A/B$ est exact, alors $A/B \bmod \beta^n$ donne les n mots de poids faible de Q [28].

La première remarque revient à effectuer une division flottante, c'est-à-dire à tronquer les n mots de poids faible de A dans l'algorithme **Division naïve**, et permet de calculer Q en environ $n^2/2$ opérations-machine. Il faut néanmoins faire attention à ne pas sur- ou sous-estimer Q ; en pratique on gardera a_{n-1}, d'ailleurs nécessaire pour le calcul du dernier quotient estimé q_0^*.

La méthode de Jebelean, à l'inverse, calcule Q de proche en proche en commençant par les mots de poids faible. Illustrons-la sur la division de $A = 766970544064109121$ par $B = 862664913$, avec $\beta = 10$: les derniers chiffres sont respectivement 1 et 3, donc $Q \equiv A/B \equiv 1/3 \equiv 7 \bmod 10$. Si q_1 est le chiffre des dizaines de Q, on doit avoir l'égalité $A \equiv 21 \equiv BQ = 13(10q_1 + 7) \equiv 30q_1 + 91 \bmod 100$. Donc $30q_1 \equiv -70 \bmod 100$, et en divisant par 10 on obtient $q_1 \equiv -7/3 \equiv 1 \bmod 10$. En poursuivant on trouve finalement $A/B \equiv 889071217 \bmod 10^9$, ce qui donne le quotient exact. Ce procédé nécessite que B et β soient premiers entre eux ; si ce n'est pas le cas, il suffit de diviser au préalable A et B par $\mathrm{pgcd}(B, \beta)$, facile à calculer si β est une puissance d'un petit entier. La méthode de Jebelean utilise $n^2/2$ multiplications de mots-machine — comme la première — plus n divisions modulaires $r_j/b_0 \bmod \beta$. Ces dernières sont plus coûteuses que les divisions flottantes de la première méthode (sauf si $1/b_0 \bmod \beta$ est précalculé), en revanche aucune erreur sur le quotient final n'est ici possible.

Krandick et Jebelean obtiennent dans [35] un nouvel algorithme en $n^2/4$ multiplications, en combinant la première méthode pour déterminer les $n/2$ mots de poids fort du quotient, et la seconde pour calculer les $n/2$ mots de poids faible :

```
> A:=766970544064109121: B:=862664913:
> evalf(A/B, 5), A/B mod 10^5;
```

66 90 81	817
-64	16*1*
2 90	× *1*
1 61	161
1 29 81	162*7*
-1 13 89	× *7*
15 92	11389

FIG. 1. Calcul de la racine carrée de 669081 par la méthode « naïve »

```
          9
.88907 10  , 71217
```

La fonction `mpz_divexact` de GMP implante l'algorithme de Jebelean, et permet donc de gagner un facteur 2 sur la division avec reste. Cependant cette fonction — du moins en GMP 3.1.1 — reste de complexité quadratique quand la taille des données croît, contrairement à la division avec reste, ce qui fait que cette dernière est plus rapide pour de grandes tailles ! En pratique, sur un Pentium II, ce phénomène se produit à partir de 14000 chiffres environ.

1.4. **Racine carrée.** La racine carrée entière consiste, étant donné un entier positif ou nul N, à trouver le plus grand carré S^2 inférieur ou égal à N, et le reste correspondant. En d'autres termes, S correspond à l'arrondi vers zéro de la racine carrée de N :

$$N = S^2 + R, \quad 0 \leq R \leq 2S.$$

Comme pour la division, il existe trois classes d'algorithmes : la méthode naïve, un algorithme récursif de type « diviser pour régner », et un algorithme asymptotiquement optimal, basé sur la transformée de Fourier rapide et la méthode de Newton.

1.4.1. *Méthode naïve.* C'est l'algorithme que l'on apprend — ou apprenait — à l'école. Soit à calculer la racine carrée de $N = 669081$ (Fig. 1). On regroupe les chiffres par deux en partant des unités. Les deux chiffres les plus significatifs sont 66, dont la racine carrée entière est 8 ; on retranche $8^2 = 64$ de 66, et on « descend » les deux chiffres suivants, soit 90. On multiplie alors la racine carrée courante par 2, ce qui donne 16, que l'on complète par le plus grand chiffre x tel que $(160 + x) \cdot x \leq 290$. Ici c'est $x = 1$, on retranche ce produit à 290, et on « descend » le dernier groupe de deux chiffres, 81. Il s'agit alors de trouver le plus grand chiffre y tel que $(1620 + y) \cdot y \leq 12981$: c'est $y = 7$. La racine carrée est donc $S = 817$ et le reste $R = 1592$.

Cet algorithme décrit ici en base 10 s'étend à n'importe quelle base β. Il suffit de savoir au départ calculer la racine carrée entière d'un nombre inférieur à β^2 ; ensuite à l'étape i, étant donnés un reste r_i et une racine carrée s_i, on doit déterminer le plus grand « chiffre » x tel que $(2s_i\beta + x)x \leq r_i$. Soit x_0 le quotient de r_i par $2s_i\beta$, alors on peut montrer que dès que $s_i \geq \beta/2$, $x = x_0$ ou $x = x_0 - 1$; donc si x_0 est trop grand, une seule correction suffit.

1.4.2. *Racine carrée récursive.* Cet algorithme est l'équivalent de la division récursive pour la racine carrée. Au lieu de traiter seulement 2 chiffres à la fois, on considère la moitié des chiffres donnés, en tirant parti d'une procédure récursive donnant à la fois racine carrée et reste, et on fait appel à la division avec reste pour calculer une approximation de la partie basse de la racine carrée. Comme ci-dessus, cette approximation peut se révéler trop grande, mais d'au plus une unité [56].

Algorithme **SqrtRem**.
Entrée : $n = a_3\beta^3 + a_2\beta^2 + a_1\beta + a_0$ avec $0 \leq a_i < \beta$ et $a_3 \geq \beta/4$
Sortie : (s, r) tels que $s^2 \leq n = s^2 + r < (s+1)^2$
$(s', r') \leftarrow$ `SqrtRem`$(a_3\beta + a_2)$

$(q, u) \leftarrow \texttt{DivRem}(r'\beta + a_1, 2s')$
$s \leftarrow s'\beta + q$
$r \leftarrow u\beta + a_0 - q^2$
if $r < 0$ **then**
    $r \leftarrow r + 2s - 1$
    $s \leftarrow s - 1$
return (s, r)

La complexité $S(n)$ pour une entrée sur $2n$ mots-machine, c'est-à-dire une racine carrée et un reste sur n mots-machine, vérifie :

$$S(2n) \sim S(n) + D(n) + C(n),$$

où $C(n)$ est le coût d'un carré sur n mots, soit $S(n) \sim (D(n)+C(n))/(2^\alpha - 1)$ si $M(n) = \Theta(n^\alpha)$ avec $\alpha > 1$. Pour $\alpha = 2$, on retrouve $S(n) \sim \frac{1}{2}M(n)$ de la méthode naïve ; pour $\alpha = \log_2 3$ (Karatsuba), cela donne $S(n) \sim \frac{4}{3}M(n)$, pour $\alpha = \log_3 5$ (Toom-Cook), cela donne $S(n) \sim 1.905M(n)$. Lorsque $M(2n) \sim 2M(n)$, contrairement à la division récursive, cet algorithme reste compétitif, avec un coût $S(n) \sim D(n) + C(n)$, soit $\frac{31}{6}M(n)$ avec le meilleur algorithme de division connu, en supposant qu'un carré coûte $2/3$ d'un produit.

Asymptotiquement, la méthode de Newton permet de faire mieux, à savoir calculer la racine carrée en $\frac{7}{2}M(n)$ opérations, donc le reste en $\frac{9}{2}M(n)$.

1.4.3. *Méthode de Newton.* Habituellement, on préfère l'itération de Newton donnant $1/\sqrt{A}$,

$$x_{k+1} = x_k + \frac{x_k}{2}(1 - Ax_k^2),$$

à celle pour $\sqrt{A}$ qui nécessite une division. L'astuce de Karp et Markstein consistant à incorporer A lors de la dernière étape s'applique ici aussi [33] : x_k étant une approximation sur $n/2$ mots de la racine carrée inverse, $y_k = Ax_k$ est une approximation sur $n/2$ mots de la racine carrée, et on remplace la dernière itération par :

$$y_{k+1} = y_k + \frac{x_k}{2}(A - y_k^2).$$

Le coût total est de $\frac{7}{2}M(n)$ — $\frac{19}{6}M(n)$ si un carré coûte $2/3$ d'un produit — pour calculer une approximation sur n mots de la racine carrée d'un entier de $2n$ mots, et donc de $\frac{9}{2}M(n)$ — respectivement $\frac{23}{6}M(n)$ — pour déterminer également le reste. (Comme pour la division, on ne sait pas calculer le reste tout seul.)

Si la méthode de Newton est asymptotiquement plus efficace, la méthode récursive est environ deux fois plus rapide pour quelques centaines de chiffres, et ne devient plus lente qu'au-delà du million de chiffres [56].

Comme pour l'inverse, des itérations d'ordre plus élevé existent pour la racine carrée inverse, comme celle-ci d'ordre 3 issue de l'essai *Computation of inverse and square root in high precision* de Gourdon et Sebah [21] :

$$h_k = 1 - Ax_k^2, \quad x_{k+1} = x_k + \frac{x_k}{8}(4h_k + 3h_k^2).$$

1.5. **Plus grand commun diviseur.** Plusieurs améliorations ont été apportées par différents auteurs — notamment Lehmer, Jebelean, Weber — au calcul de plus grand commun diviseur. L'idée de Jebelean est la suivante [29]. Traditionnellement, pour accélérer l'algorithme d'Euclide, on utilise le mot de poids fort de chacune des entrées a et b pour trouver une combinaison linéaire $ua + vb$ petite. Or u et v étant alors de taille moitié de celle d'un mot-machine, les multiplications ua et vb utilisent de façon inefficace la multiplication d'un grand entier par un mot-machine. Jebelean propose de considérer *deux mots* de poids fort de chacun des opérandes, ce qui conduit à u et v de la taille d'un mot-machine, et permet de gagner un facteur deux sur le nombre de multiplications d'un grand entier par un mot-machine. La méthode de Weber [54] est basée sur la k-réduction de Sorenson, qui remplace a par $|c_1 a - c_2 b|/2^k$. Ces améliorations n'améliorent néanmoins pas la complexité asymptotique $\mathcal{O}(n^2)$ de l'algorithme d'Euclide.

La meilleure complexité théorique connue pour le calcul de pgcd d'entiers est $\mathcal{O}(M(n)\log n)$. La plupart des auteurs se contentent de traiter le cas du pgcd de polynômes, affirmant que le cas entier est similaire modulo les problèmes de retenue [1, 53]. Il semble selon les quelques auteurs qui ont implanté un pgcd entier sous-quadratique — Cesari [15], Crandall, Woltman — que celui-ci ne concurrence le pgcd classique qu'au-delà de 10000 chiffres décimaux pour un pgcd étendu, voire plus pour un pgcd simple.

1.6. **Lecture et écriture.** Un calcul donné comporte en général peu de lectures ou d'écritures, par rapport aux multiplications ou divisions. C'est sans doute la raison pour laquelle les opérations de lecture et d'écriture sont habituellement passées sous silence, et parfois non optimisées, même dans les bibliothèques les plus efficaces comme GMP.

Les deux procédures ci-dessous effectuent la lecture et l'écriture d'un entier multiprécision, qui correspondent à un changement de base, via un algorithme de type « diviser pour régner ». La base interne de représentation des entiers est β, l'interaction avec l'utilisateur se faisant en base γ. En pratique on précalcule les $\mathcal{O}(\log n)$ puissances $\gamma^{n'}$, soit un coût initial de $\mathcal{O}(M(n))$.

Algorithme **Lecture**.
Entrée : un entier $A = \sum_{i=0}^{n-1} a_i \gamma^i$ en base γ
Sortie : sa représentation interne $\sum_{i=0}^{m-1} b_i \beta^i$ en base β
si $n = 1$, retourner la représentation de a_0 en base β
$n' \leftarrow \lceil n/2 \rceil$
$A_0 \leftarrow \texttt{Lecture}(\sum_{i=0}^{n'-1} a_i \gamma^i), \quad A_1 \leftarrow \texttt{Lecture}(\sum_{i=n'}^{n-1} a_i \gamma^{i-n'})$
retourner $A_0 + A_1 \gamma^{n'}$

Algorithme **Écriture**.
Entrée : un entier $A = \sum_{i=0}^{m-1} b_i \beta^i$ en base interne β
Sortie : sa représentation $\sum_{i=0}^{n-1} a_i \gamma^i$ en base γ
$n' \leftarrow \lceil \frac{m}{2} \frac{\log \beta}{\log \gamma} \rceil, \quad (Q, R) \leftarrow \texttt{DivRem}(A, \gamma^{n'})$
si $Q = 0$, retourner la représentation de R en base γ
$[a_{n'-1}, \ldots, a_0] \leftarrow \texttt{Écriture}(R), \quad [a_{n-1}, \ldots, a_{n'}] \leftarrow \texttt{Écriture}(Q)$
retourner $[a_{n-1}, \ldots, a_{n'}, a_{n'-1}, \ldots, a_0]$

Si $L(n)$ et $E(n)$ désignent les coûts de lecture et d'écriture d'entiers de taille n, on obtient $L(2n) \sim 2L(n) + M(n)$ et $E(2n) \sim 2E(n) + D(n)$. C'est le même type d'équation que pour la division récursive p. 10, par conséquent $L(n) \sim \frac{1}{2^\alpha - 2} M(n)$ et $E(n) \sim \frac{1}{2^\alpha - 2} D(n)$, la fraction étant remplacée par $\frac{1}{2}\log_2 n$ pour la multiplication via FFT. La lecture et l'écriture sont donc, comme le pgcd, intrinsèquement plus complexes que la multiplication [34]. Pour de petites tailles, on préférera aux algorithmes de type « diviser pour régner » ci-dessus des méthodes classiques à la Horner, de complexité $n^2/2$, ce qui revient à prendre $n' = 1$.

Dans l'algorithme **Écriture**, pour éviter la division de A par $X = \gamma^{n'}$, on peut calculer une approximation flottante Y de $1/X$ et en déduire une estimation de Q via AY, puis $R = A - QX$ via une multiplication supplémentaire, ce qui équivaut en coût à $2L(n)$. Le calcul des $Y \sim \gamma^{-n'}$ s'effectue par itération de Newton parallèlement au calcul des $\gamma^{n'}$, avec une précision finale $n/2$; cela donne un surcoût de $4M(\frac{n}{4}) + 4M(\frac{n}{8}) + \cdots$. Cette méthode donne une meilleure complexité dans le cas de la FFT, où ce surcoût est négligeable.

Le Tableau 1 récapitule les meilleures complexités obtenues pour les différentes opérations entières en fonction de la multiplication (la division et la racine carrée s'entendent avec reste). Les entrées

Opération	naïf	Karatsuba	Toom-Cook	FFT
Multiplication	n^2	$n^{\log_2 3}$	$n^{\log_3 5}$	$n \log n \log\log n$
Carré	1/2	2/3	0.72	$\sim 2/3$
Division	1	2	2.63	9/2
Racine carrée	1/2	4/3	1.91	$\sim 23/6$
Lecture	1/2	1	1.31	$\frac{1}{2}\log_2 n$
Écriture	1/2	2	3.45	$\log_2 n$

TAB. 1. Meilleures complexités connues pour différentes opérations entières

du tableau donnent les facteurs multiplicatifs par rapport au coût de la multiplication indiqué en deuxième ligne.

2. Calcul flottant

Comme pour les entiers, plusieurs représentations sont possibles pour les réels : dense ou creuse, en virgule fixe ou flottante.[3] Les « expansions » (cf. travaux de Priest, Daumas et *al.* [44, 19]) sont une représentation creuse tirant parti de l'efficacité des processeurs pour les opérations sur des registres flottants. On ne parlera pas ici des représentations à base de fractions continues, où par exemple $3+1/(7+1/15)$ correspond à l'intervalle $[\frac{333}{106}, \frac{355}{113}[\sim [3.141509\ldots 3.141593[$.

Les calculs en virgule fixe suffisent quand on sait borner *a priori* la croissance des données, par exemple pour des transactions financières, ou pour des calculs flottants devant donner *in fine* un entier. Lorsqu'il s'agit de calculer les n premiers chiffres d'une constante donnée, un calcul en virgule fixe terminera toujours, alors qu'un calcul en virgule flottante pourra boucler si cette constante est nulle, comme par exemple : $\sqrt{1+\sqrt{3}}+\sqrt{3+3\sqrt{3}}-\sqrt{10+6\sqrt{3}}$.

Nous considérons ici une représentation dense en virgule flottante, de la forme :

$$\pm m \cdot 2^e, m \in \mathbb{N}, e \in \mathbb{Z}, E_{\min} \le e \le E_{\max}.$$

Cette représentation n'est pas unique, mais a l'avantage de coder ± 0 ; avec $E_{\min} = E_{\max}$ on retrouve une représentation en virgule fixe. Notons que si la mantisse m est de précision arbitraire, l'exposant e reste borné, comme dans la norme IEEE 754 [25]. On peut distinguer deux groupes d'implantations suivant la sémantique donnée aux calculs flottants :

- celles où le résultat d'un calcul a une erreur relative correspondant *à peu près* à la précision courante, mais sans garantie formelle,
- celles où le résultat d'un calcul est le nombre flottant le plus proche du résultat exact avec la précision et le mode d'arrondi donnés. On parle alors d'« arrondi exact ».

MAPLE et PARI/GP font partie du premier groupe, comme MATHEMATICA, qui cependant fournit à l'utilisateur une indication — sans garantie formelle toutefois — du nombre de chiffres corrects d'un calcul, via le concept de *significance arithmetic* [50]. Les implantations du second groupe sont peu nombreuses en précision arbitraire : on peut citer Arithmos [17] et MPFR [18], qui fournissent les quatre modes d'arrondi classiques. Une particularité d'Arithmos est que la base interne β peut être choisie par l'utilisateur dans l'intervalle $[2, 2^{24}]$.

Du fait de la représentation choisie, la plupart des opérations flottantes en précision arbitraire se ramènent à des opérations entières sur les mantisses et sur les exposants, la majeure partie du coût portant sur les mantisses. La plupart des algorithmes sur les entiers sont donc utiles — et utilisés

3. Dans un calcul en virgule fixe, on tronque toujours au même endroit, par exemple, après les centimes pour des sommes d'argent, alors qu'en virgule flottante, la quantité négligée dépend de la valeur absolue des opérandes : 1.00/3 donne 0.33 en virgule fixe et .333 en virgule flottante.

— pour les calculs flottants. Cependant certains algorithmes spécifiques aux flottants existent, que nous présentons ci-dessous.

Nous considérons dans la suite que les données et le résultat d'une opération flottante ont la même précision, les méthodes devant être adaptées lorsque ce n'est pas le cas. Si $X(n)$ désigne le coût d'une opération sur des entiers de n bits (ou n mots-machine), nous noterons $X^*(n)$ le coût de l'opération correspondante sur les flottants ; par exemple $M^*(n)$ est le coût de la multiplication flottante.

2.1. **Multiplication.** La multiplication de deux flottants $x = m \cdot 2^e$ et $y = m' \cdot 2^{e'}$ revient à déterminer le flottant $m'' \cdot 2^{e''}$ le plus proche de xy. Si x et y sont non nuls, m et m' ayant n bits, il s'agit de trouver l'entier m'' sur n bits le plus proche de mm', à une puissance de 2 près. On peut bien sûr calculer le produit entier de m et m', et arrondir ensuite le résultat suivant le mode donné, donc $M^*(n) \leq M(n) + \mathcal{O}(n)$.

Dans le cas le pire, on ne peut pas faire mieux que le produit entier. En effet, on peut avoir besoin de tous les bits du produit mm' pour pouvoir décider de l'arrondi correct ; c'est le cas lorsque $m = 2^n - 2$ et $m' = 2^{n-1} + 1$. Néanmoins, dans la plupart des cas il suffit de connaître les n premiers bits de mm' plus quelques bits supplémentaires. Avec la multiplication naïve, on peut obtenir une approximation de ces $n + o(n)$ bits en $\sim n^2/2$ opérations [36].

Lorsqu'on dispose d'un algorithme de multiplication entière en $\mathcal{O}(n^\alpha)$ avec $\alpha < 2$, T. Mulders montre dans [43] comment effectuer un produit tronqué plus rapidement qu'un produit entier. Pour l'algorithme de Karatsuba, le gain théorique est de l'ordre de 20%, et diminue avec α pour devenir nul lorsque α approche 1. Cependant garantir un arrondi exact avec cet algorithme reste non trivial.

Pour les algorithmes utilisant la transformée de Fourier rapide, on ne sait pas calculer un produit flottant plus efficacement qu'en passant par le produit entier. L'équivalence $M^*(n) \sim M(n)$ reste donc une question ouverte.

2.2. **Division.** La première variante de division exacte (§ 1.3.5) conduit à un algorithme de division flottante naïve de coût moyen $n^2/2$, comme pour la multiplication flottante.

L'algorithme de division récursive (§ 1.3.3) donne directement l'arrondi exact, grâce au calcul simultané du reste. En effet, pour obtenir l'arrondi du quotient m/m' de deux mantisses positives, on divise $2^k m$ par m', où k est pris de telle sorte que le quotient q ait exactement n bits. Si $0 \leq r < m'$ est le reste, on a alors :

$$\frac{2^k m}{m'} = q + \frac{r}{m'}.$$

L'arrondi vers 0 et vers $-\infty$ est q ; celui vers $+\infty$ est q si $r = 0$, $q+1$ sinon ; l'arrondi au plus proche est q si $r < m'/2$, $q + 1$ si $r > m'/2$, avec la règle de l'arrondi pair si $r = m'/2$.

Lorsque la multiplication est en $\mathcal{O}(n^\alpha)$ avec $1 < \alpha < 2$ — ce qui correspond aux méthodes de Karatsuba et Toom-Cook — Mulders propose dans [43] un algorithme de division tronquée (*short division*) théoriquement meilleur que la division récursive, de complexité $\sim 1.397M(n)$ pour Karatsuba. Ce résultat a été amélioré dans [24], où un algorithme de complexité $\sim M(n)$ est présenté, toujours pour Karatsuba. Ces différents algorithmes, qui sont comparés dans [46], sont néanmoins de comportement plus chaotique que la division récursive, et l'obtention de l'arrondi exact est plus difficile.

Avec la multiplication via FFT, la division flottante est une opération de base, puisque c'est le quotient calculé par la méthode de Newton. On a donc $D^*(n) \sim D(n) - M(n) \sim \frac{7}{2}M(n)$, puisque $D(n)$ tient compte du calcul du reste, coûtant $M(n)$ opérations supplémentaires. Pour déterminer l'arrondi exact, la méthode de Newton avec une précision cible de $n+o(n)$ bits suffit dans la plupart des cas, sinon il faut calculer le reste correspondant en $M(n)$ opérations supplémentaires, afin de se ramener à la discussion ci-dessus.

2.3. **Racine carrée.** La racine carrée flottante naïve s'obtient en $n^2/4$ opérations en moyenne. Comme pour la division, l'algorithme de racine carrée récursive (§ 1.4.2) donne directement l'arrondi exact. Soit une mantisse m positive d'au plus n bits. On choisit k de sorte que $2^{2k}m = s^2 + r$ avec $0 \le r \le 2s$ et s ayant exactement n bits. Alors l'arrondi vers 0 ou vers $-\infty$ de $2^k\sqrt{m}$ est s ; l'arrondi vers $+\infty$ est s si $r = 0$, $s+1$ sinon ; l'arrondi au plus proche est s si $r \le s$, $s+1$ sinon.

Avec la multiplication via FFT, on se retrouve dans un cas similaire à la division, avec $S^*(n) \sim S(n) - M(n) \sim \frac{7}{2}M(n)$. Là encore, le calcul du reste peut s'avérer nécessaire, notamment dans le cas d'une racine carrée exacte.

2.4. **Fonctions algébriques.** Soit une fonction algébrique $y(x)$, c'est-à-dire solution de $P(x,y) = 0$ où P est un polynôme. La méthode de Newton conduit à l'itération

$$y_{k+1} = y_k - \frac{P(x, y_k)}{P'_y(x, y_k)}.$$

Brent montre dans [10] que si $P(x,y)$ s'évalue en $\mathcal{O}(M(n))$, le calcul de $y(x)$ a la même complexité. Une conséquence importante est que les fonctions algébriques s'évaluent en $\mathcal{O}(M(n))$ comme la multiplication. Brent compare dans [8] plusieurs méthodes (sécante, Newton) pour trouver les racines de $P(x,y) = 0$.

Il arrive couramment dans la méthode de Newton qu'on ait à calculer un produit de $2n$ chiffres par n chiffres, donnant un résultat de $3n$ chiffres, dont seuls les n chiffres médians sont utiles. L'algorithme *Recursive Middle Product* de [24] permet d'accélérer ce type de calcul, en fournissant directement ces n chiffres médians.

Pour garantir l'arrondi exact du résultat, on recourra à des bornes sur le nombre de 0 ou 1 consécutifs dans la valeur d'une fonction algébrique [37].

2.5. **Fonctions élémentaires.** Lorsque la précision voulue est faible et fixée *a priori*, comme c'est le cas pour la double précision (mantisse de 53 bits), on utilise en général des approximations polynomiales, après une première étape appelée « réduction d'argument » ramenant l'étude à un petit domaine. La réduction d'argument, les approximations polynomiales et le problème de l'arrondi exact sont traités par D. Defour et J.-M. Muller dans [20], avec le calcul de l'exponentielle comme exemple.

On se restreindra ici au cas de la grande précision, disons 1000 bits ou plus. Brent montre dans [9] que les fonctions exp, log, sinh, cosh, tanh, arcsinh, arccosh, arctanh sont toutes de même complexité asymptotique, à un facteur constant près. De même, il montre que les fonctions $\sin, \cos, \tan, \arcsin$, arccos, arctan sont de même complexité. L'équivalence de exp et sin n'est par contre pas certaine, malgré l'identité $\exp ix = \cos x + i \sin x$. L'exponentielle et l'arc-tangente se calculant en $\mathcal{O}(M(n)\log n)$ opérations via la moyenne arithmético-géométrique [10], il en est de même pour les fonctions sus-citées. Par exemple le logarithme, implanté par S. Boldo dans la bibliothèque MPFR [18], est donné par :

$$\log x = \frac{\pi}{2\ \mathrm{agm}(1, \frac{4}{2^m x})} - m \log 2 + o(2^{-p} \log x)$$

où $\mathrm{agm}(u,v)$ désigne la moyenne arithmético-géométrique de u et v.

Plus précisément, Brent montre dans [10] que la constante π peut être calculée en $\frac{15}{2}M(n)\log_2 n$ opérations, le logarithme et l'exponentielle en $13M(n)\log_2 n$ opérations, l'arc-tangente et le sinus en $34M(n)\log_2 n$ opérations.

2.6. **Fonctions hypergéométriques et holonomes.** Les fonctions hypergéométriques sont les séries convergentes $\sum_n a_n x^n$ dont les coefficients vérifient $\frac{a_{n+1}}{a_n} = \frac{p(n)}{q(n)}$ où p et q sont des polynômes.

C'est un sous-ensemble de la classe des fonctions holonomes, définies par le fait que, outre la convergence de la série $\sum_n a_n x^n$, la suite (a_n) vérifie une récurrence linéaire à coefficients polynomiaux en n :

$$p_0(n)a_n + p_1(n)a_{n+1} + \cdots + p_k(n)a_{n+k} = 0.$$

Les fonctions holonomes s'évaluent en $\mathcal{O}(M(n)\log^2 n)$ opérations par scindage binaire (*binary splitting*) [6, 23, 32, 21], sous réserve d'une relation fonctionnelle simple donnant $f(x+y)$ en fonction de $f(x)$ et $f(y)$. Brent l'a montré dès 1976 dans le cas de l'exponentielle et du sinus [9]. E. Jeandel a implanté cette méthode de façon générique pour les fonctions hypergéométriques ${}_2F_1(a,b,c;x)$,

$$f(x) = \sum_{n=0}^{\infty} \frac{a(a+1)\cdots(a+n-1)b(b+1)\cdots(b+n-1)}{c(c+1)\cdots(c+n-1)} \frac{x^n}{n!},$$

où a, b, c sont des rationnels quelconques, et où chacune des factorielles montantes peut être omise [27]. Ceci couvre la plupart des fonctions usuelles : $\exp x = {}_0F_0(x)$, $\log x = (x-1)\cdot {}_2F_1(1,1,2;1-x)$, $\sinh x = x \cdot {}_0F_1(\frac{3}{2}; \frac{x^2}{4})$, $\cos x = {}_0F_1(\frac{1}{2}; -\frac{x^2}{4})$, $\arctan x = x \cdot {}_2F_1(1, \frac{1}{2}, \frac{3}{2}; -x^2)$.

2.7. **Fonctions spéciales.** Certaines fonctions spéciales relèvent également du scindage binaire [12] :

$$\operatorname{erf}(x) = \frac{2x}{\sqrt{\pi}} \sum_{n=0}^{\infty} \frac{(-x^2)^n}{n!(2n+1)} = \frac{2x}{\sqrt{\pi}} {}_1F_1(\frac{1}{2}, \frac{3}{2}, -x^2).$$

Le même article montre comment évaluer la fonction zêta de Riemann via la formule d'Euler Maclaurin :

$$\zeta(s) = \sum_{j=1}^{p-1} j^{-s} + \frac{1}{2}p^{-s} + \frac{p^{1-s}}{s-1} + \sum_{k=1}^{m} T_{k,p}(s) + E_{m,p}(s),$$

où $T_{k,p}(s)$ dépend du nombre de Bernoulli B_{2k}, et $E_{m,p}(s)$ est un terme d'erreur majoré en fonction de $T_{m+1,p}$. Schönhage montre aussi dans [48] que $\log t$ bits de $\zeta(\frac{1}{2} + it)$ s'obtiennent en temps $t^{3/8+o(1)}$. On trouvera dans [7] un florilège de méthodes pour évaluer $\zeta(s)$.

3. Quelques chiffres

La table 2 indique le nombre de mots-machine à partir duquel la bibliothèque GMP passe d'un algorithme à un autre asymptotiquement plus efficace, sur deux architectures différentes. Par exemple,

borne	Pentium II	Alpha 21264
`KARATSUBA_MUL_THRESHOLD`	23 (220)	47 (900)
`KARATSUBA_SQR_THRESHOLD`	52 (500)	100 (1920)
`TOOM3_MUL_THRESHOLD`	138 (1320)	71 (1360)
`TOOM3_SQR_THRESHOLD`	172 (1650)	105 (2020)
`BZ_THRESHOLD`	102 (980)	32 (610)
`FFT_MODF_MUL_THRESHOLD`	672 (6470)	816 (15720)
`FFT_MUL_THRESHOLD`	5888 (56710)	1504 (28975)

TAB. 2. Frontières entre différents algorithmes avec GMP 3.1.1, sur Pentium II sous Linux (mot-machine de 32 bits) et Alpha 21264 sous OSF (64 bits). Les frontières sont en mots-machine, les valeurs entre parenthèses en chiffres décimaux

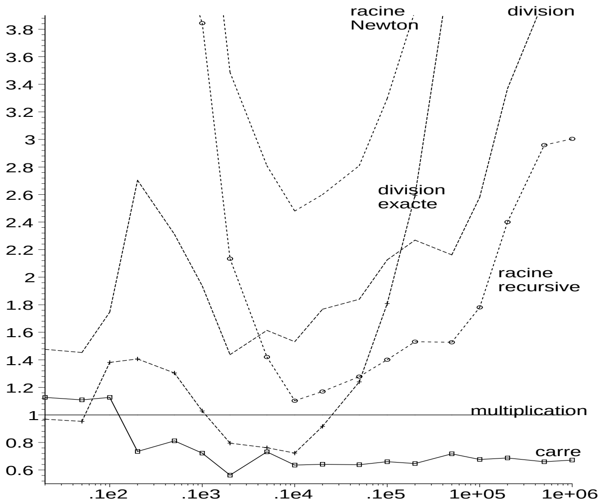

FIG. 2. Comparaison de différentes opérations sur les entiers en GMP 3.1.1 sur un Pentium II. L'abscisse indique le nombre n de chiffres décimaux des opérandes, l'ordonnée le rapport des temps par rapport à la multiplication $n \times n$

mots	8	16	32	64	128	256	512	1024	2048
chiffres	154	308	616	1233	2466	4932	9864	19728	39k
temps	0.9μs	2.7μs	8.9μs	30μs	96μs	.29ms	.98ms	2.0ms	13ms
mots	4096	8192	16k	32k	66k	131k	262k	524k	
chiffres	79k	158k	316k	631k	1.3M	2.5M	5.1M	10M	
temps	30ms	55ms	0.13s	0.35s	0.68s	1.4s	3.6s	7.4s	

TAB. 3. Temps de multiplication de deux entiers de même taille avec GMP 3.1.1 sur Alpha 21264 à 500 Mhz (machine `cosimo` de l'UMS Médicis)

la constante `KARATSUBA_MUL_THRESHOLD` correspond au passage de la multiplication en $\mathcal{O}(n^2)$ à l'algorithme de Karatsuba. Il est à noter que ces valeurs, déterminées automatiquement par le programme `tuneup` de GMP, sont optimales pour l'architecture considérée. (Configurer GMP avec `--enable-fft` pour avoir accès à l'algorithme de Schönhage-Strassen de multiplication modulo $2^m + 1$.) La table 3 donne les temps de multiplication correspondant.

4. Applications

Un champ privilégié d'application pour l'arithmétique en précision arbitraire est naturellement la théorie des nombres, en particulier la factorisation d'entier. La plupart des algorithmes probabilistes de factorisation effectuent essentiellement des multiplications modulo l'entier n à « casser », plus quelques pgcds mais dont le coût relatif est négligeable. C'est le cas de la méthode ρ de Pollard, qui calcule $x_{k+1} = x_k^2 + a \bmod n$, et de la méthode des courbes elliptiques (ECM pour *Elliptic Curve Method*), qui calcule des multiples d'un point sur une courbe elliptique, se ramenant à une suite d'additions ou de duplications de points, chacune de ces opérations s'effectuant via un nombre fini de multiplications modulaires. Crandall et Fagin ont ainsi trouvé en 1991 deux facteurs de 19 chiffres du treizième nombre de Fermat $2^{2^{13}} + 1$, ce dernier ayant 2467 chiffres décimaux [16]. Plusieurs optimisations ont été proposées pour ECM, soit en diminuant le nombre de multiplications modulaires, soit en augmentant la probabilité de succès de la méthode. Ces améliorations restent vaines si les multiplications modulaires sont inefficaces. Plusieurs logiciels efficaces de factorisation par ECM existent : `ecmfft` de Peter Montgomery, `mprime` de George Woltman, et `gmp-ecm`, qui détenait début 2001 le record de factorisation par ECM, avec un facteur de 54 chiffres.

L'arithmétique en précision arbitraire peut aussi accélérer les calculs polynomiaux. Pour multiplier $9x^5 + 4x^4 + 5x^3 + 10x^2 + 6x + 10$ par $3x^5 + 8x^4 + 6x + 10$, substituons la variable x par la valeur 1000, ce qui donne respectivement 9 004 005 010 006 010 et 3 008 000 000 006 010, dont le produit vaut 27084047070152192150110136120100. Il ne reste plus qu'à « lire » les coefficients du produit par blocs de trois chiffres :

$$27x^{10} + 84x^9 + 47x^8 + 70x^7 + 152x^6 + 192x^5 + 150x^4 + 110x^3 + 136x^2 + 120x + 100.$$

Quant aux calculs flottants, dans la plupart des applications la double précision suffit, soit environ 16 chiffres significatifs. Dans certains cas cependant la quadruple précision est nécessaire [26]. Davantage de précision est également utile lorsqu'on désire identifier des constantes algébriques : par exemple, Pari/GP a besoin de 86 décimales de $\sqrt{2}+\sqrt{3}+\sqrt{5}$ pour trouver son polynôme minimal :

```
? \p 78
   realprecision = 86 significant digits (78 digits displayed)
? algdep(sqrt(2)+sqrt(3)+sqrt(5), 8)
%1 = x^8 - 40*x^6 + 352*x^4 - 960*x^2 + 576
```

Voir également à ce sujet la page web *Inverse Symbolic Calculator* [14].

5. Pour en savoir plus

Le tome 2 de *The Art of Computer Programming* [34] reste une référence, même si de nouveaux algorithmes ont été inventés depuis. Le livre plus récent *Modern Computer Algebra* [53] décrit très clairement la plupart des algorithmes utilisés en calcul formel, y compris ceux sur les nombres et les polynômes, qui sont étroitement liés. Sur le web, le site *Mathematical constants and computation* [21] propose des pages très bien faites sur la méthode de Newton et les itérations quadratiques ou d'ordre plus élevé pour le calcul de l'inverse ou de la racine carrée, et présente aussi les meilleurs algorithmes connus pour le calcul en grande précision de constantes telles que π, $\log 2$, la constante d'Euler γ, ou $\zeta(3)$.

Le lecteur désirant implanter les algorithmes récursifs de division et racine carrée consultera avec profit [46], où sont décrits avec minutie certains détails obscurs passés sous silence ici.

Dans [5], Bernstein décrit en termes d'homomorphisme d'anneau de nombreux algorithmes de multiplication, notamment ceux à base de FFT. Il a également effectué une comparaison des différents logiciels de multiplication rapide [4], et maintient une liste électronique pour les développeurs de logiciels sur les grands entiers ; utiliser l'adresse `bnis-subscribe@list.cr.yp.to` pour s'abonner.

Nous avons omis les calculs modulaires, pour lesquels une bonne référence est [40, ch. 14]. La méthode REDC de Montgomery permet, après un calcul préliminaire, d'effectuer une multiplication modulaire sans division [42] ; pour l'exponentiation modulaire, la méthode 2^k-aire et sa variante à « fenêtre glissante » (*sliding window*) diminuent d'un facteur k et $k+1$ respectivement le nombre de multiplications modulaires. Quant à l'inversion modulaire, elle se ramène à un calcul de pgcd étendu ; lorsque plusieurs inversions modulo le même entier sont à calculer, une astuce due encore à Montgomery permet de n'effectuer qu'un seul pgcd : par exemple pour deux inversions, on commence par calculer $g = \frac{1}{ab} \bmod n$, puis $\frac{1}{a} = bg \bmod n$, et $\frac{1}{b} = ag \bmod n$.

Remerciements.
Pour leurs remarques pertinentes sur une pré-version de cet article, ou pour le temps passé à m'expliquer les mystères cachés de l'arithmétique en précision arbitraire, je tiens à remercier par ordre alphabétique : Dan Bernstein, Richard Brent, Mathieu Dutour, Torbjörn Granlund, Guillaume Hanrot, Vincent Lefèvre, Claire Moreau-Finot, Michel Quercia, Jean-Luc Rémy, Kevin Ryde, Bruno Salvy, Pascal Sebah, Jean Vuillemin.

Références

[1] Aho, A. V., Hopcroft, J. E., and Ullman, J. D. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[2] Bajard, J.-C., and Didier, L.-S. Les systèmes modulaires de représentation des nombres : un tour d'horizon. *Calculateurs parallèles Numéro spécial sur l'arithmétique des ordinateurs* (2001).

[3] Batut, C., Bernardi, D., Cohen, H., and Olivier, M. *User's Guide to PARI-GP*, 1995. `ftp://megrez.math.u-bordeaux.fr/pub/pari`.

[4] Bernstein, D. J. Integer multiplication benchmarks. `http://cr.yp.to/speed/mult.html`.

[5] Bernstein, D. J. Multidigit multiplication for mathematicians. *Advances in Applied Mathematics*. À paraître. Cf `http://cr.yp.to/papers.html`.

[6] Borwein, J. M., and Borwein, P. B. *Pi and the AGM.* Wiley-Interscience, 1987.

[7] Borwein, J. M., Bradley, D. M., and Crandall, R. E. Computational strategies for the Riemann zeta function. *Journal of Computational and Applied Mathematics 121*, 1–2 (2000), 247–296.

[8] Brent, R. P. Multiple-precision zero-finding methods and the complexity of elementary function evaluation. In *Analytic Computational Complexity* (New York, 1975), J. F. Traub, Ed., Academic Press, pp. 151–176. `web.comlab.ox.ac.uk/oucl/work/richard.brent/pub/pub028.html`.

[9] Brent, R. P. The complexity of multiple-precision arithmetic. In *The Complexity of Computational Problem Solving* (1976), R. S. Anderssen and R. P. Brent, Eds., University of Queensland Press, pp. 126–165. `web.comlab.ox.ac.uk/oucl/work/richard.brent/pub/pub032.html`.

[10] Brent, R. P. Fast multiple-precision evaluation of elementary functions. *J. ACM 23*, 2 (1976), 242–251.

[11] Brent, R. P. A Fortran multiple-precision arithmetic package. *ACM Trans. Math. Softw. 4*, 1 (1978), 57–70.

[12] Brent, R. P. Unrestricted algorithms for elementary and special functions. In *Information Processing* (1980), S. H. Lavington, Ed., vol. 80, pp. 613–619. Cf `web.comlab.ox.ac.uk/oucl/work/richard.brent/pub/pub052.html`.

[13] Burnikel, C., and Ziegler, J. Fast recursive division. Rapport de recherche MPI-I-98-1-022, MPI Saarbrücken, 1998.

[14] Centre for Experimental and Constructive Mathematics. Inverse symbolic calculator. `http://www.cecm.sfu.ca/projects/ISC/`.

[15] Cesari, G. Parallel implementation of Schönhage's integer GCD algorithm. In *Proceedings of the Algorithmic Number Theory Symposium* (1998), J. P. Buhler, Ed., no. 1423 in Lecture Notes in Computer Science, pp. 65–76.

[16] Crandall, R., and Fagin, B. Discrete weighted transforms and large-integer arithmetic. *Mathematics of Computation 62*, 205 (1994), 305–324.

[17] Cuyt, A., Kuterna, P., Verdonk, B., and Vervloet, J. Arithmos: a reliable integrated computational environment. `http://win-www.uia.ac.be/u/cant/arithmos/index.html`, 2001.

[18] Daney, D., Hanrot, G., Lefèvre, V., Rouillier, F., and Zimmermann, P. The MPFR library. `http://www.mpfr.org/`.

[19] Daumas, M., and Finot, C. Division of floating point expansions with an application to the computation of a determinant. *Journal of Universal Computer Science 5*, 6 (1999), 323–338.

[20] Defour, D., and Muller, J.-M. évaluation des fonctions élémentaires. *Calculateurs parallèles Numéro spécial sur l'arithmétique des ordinateurs* (2001).

[21] Gourdon, X., and Sebah, P. Mathematical constants and computation. `http://numbers.computation.free.fr/Constants/constants.html`, 2001.

[22] Granlund, T. *GNU MP: The GNU Multiple Precision Arithmetic Library*, 2000. Version 3.1.1, `http://www.swox.se/gmp/`.

[23] Haible, B., and Papanikolaou, T. Fast multiprecision evaluation of series of rational numbers. Rapport Technique TI-7/97, Darmstadt University of Technology, 1997.

[24] Hanrot, G., Quercia, M., and Zimmermann, P. Speeding up the division and square root of power series. Rapport de recherche 3973, Institut National de Recherche en Informatique et en Automatique, 2000.

[25] IEEE standard for binary floating-point arithmetic. ANSI-IEEE Standard 754, New York, 1985. 18 pages.

[26] Indelicato, P. De l'utilité de la quadruple précision. Lettre d'information du CNUSC, numéro 68, 1999.

[27] Jeandel, E. Évaluation rapide de fonctions hypergéométriques. Rapport technique 242, Institut National de Recherche en Informatique et en Automatique, 2000. 17 pages.

[28] Jebelean, T. An algorithm for exact division. *Journal of Symbolic Computation 15* (1993), 169–180.

[29] Jebelean, T. A double-digit Lehmer-Euclid algorithm for finding the GCD of long integers. *Journal of Symbolic Computation 19* (1995), 145–157.

[30] Jebelean, T. Practical integer division with Karatsuba complexity. In *Proc. ISSAC'97* (Maui, Hawaii, 1997), W. W. Küchlin, Ed., pp. 339–341.

[31] Karatsuba, A. A., and Ofman, Y. P. Multiplication of multiplace numbers by automata. *Dokl. Akad. Nauk SSSR 145*, 2 (1962), 293–294.

[32] Karatsuba, E. A. Fast evaluation of hypergeometric functions by FEE. *Computational Methods and Function Theory* (1997).

[33] Karp, A. H., and Markstein, P. High-precision division and square root. *ACM Trans. Math. Softw. 23*, 4 (1997), 561–589.

[34] Knuth, D. E. *The Art of Computer Programming*, vol. 2 : Seminumerical Algorithms. Addison-Wesley, 1981. 2e édition.

[35] Krandick, W., and Jebelean, T. Bidirectional exact integer division. *Journal of Symbolic Computation 21*, 4–6 (1996), 441–456.

[36] Krandick, W., and Johnson, J. R. Efficient multiprecision floating point multiplication with exact rounding. Rapport technique 93-76, RISC, Johannes Kepler University, Linz, Austria, 1993.

[37] Lang, T., and Muller, J.-M. Bound on run of zeros and ones for images of floating-point numbers by algebraic functions. Rapport de recherche 4045, Institut National de Recherche en Informatique et en Automatique, 2000.

[38] Lefèvre, V. Multiplication par une constante. *Calculateurs parallèles Numéro spécial sur l'arithmétique des ordinateurs* (2001). 15 pages.

[39] Maeder, R. Storage allocation for the karatsuba integer multiplication algorithm. DISCO, 1993. preprint.

[40] Menezes, A. J., van Oorschot, P. C., and Vanstone, S. A. *Handbook of Applied Cryptography.* CRC Press, 1997. `http://cacr.math.uwaterloo.ca/hac/`.

[41] Moenck, R., and Borodin, A. Fast modular transforms via division. In *Proceedings of the 13th Annual IEEE Symposium on Switching and Automata Theory* (1972), pp. 90–96.

[42] Montgomery, P. L. Modular multiplication without trial division. *Mathematics of Computation 44*, 170 (1985), 519–521.

[43] Mulders, T. On short multiplications and divisions. *AAECC 11*, 1 (2000), 69–88.

[44] Priest, D. M. Algorithms for arbitrary precision floating point arithmetic. In *Proceedings of the 10th Symposium on Computer Arithmetic* (Grenoble, France, 1991), P. Kornerup and D. Matula, Eds., IEEE Computer Society Press, pp. 132–144.

[45] Quercia, M. Bibliothèque numerix. `http://pauillac.inria.fr/~quercia/cdrom/bibs/numerix.tar.gz`, 2000. version 0.13.

[46] Quercia, M. Calcul multiprécision. `http://pauillac.inria.fr/~quercia/papers/multiprecision.ps.gz`, 2000. 19 pages.

[47] Quercia, M. Division et racine carrée rapides. `http://pauillac.inria.fr/~quercia/papers/divrap.ps.gz`, 2000. 10 pages.

[48] Schönhage, A. Numerik analytischer Funktionen und Komplexität. *Jahresberichts der Deutschen Mathematiker-Vereinigung 92* (1990), 1–20. Teubner, Stuttgart.

[49] Schönhage, A., and Strassen, V. Schnelle Multiplikation großer Zahlen. *Computing 7* (1971), 281–292.

[50] Sofroniou, M., and Spaletta, G. Precise numerical computation, 1999. 18 pages.

[51] Svoboda, A. An algorithm for division. *Information Processing Machines 9* (1963), 25–34.

[52] van der Hoeven, J. Lazy multiplication of formal power series. In *Proceedings of ISSAC'97* (Maui, Hawaii, 1997), W. W. Küchlin, Ed., pp. 17–20.

[53] von zur Gathen, J., and Gerhard, J. *Modern Computer Algebra.* Cambridge University Press, 1999.

[54] Weber, K. The accelerated integer GCD algorithm. *ACM Trans. Math. Softw. 21* (1995), 111–122.

[55] Woltman, G. The great internet mersenne prime search. `http://www.mersenne.org`.

[56] Zimmermann, P. Karatsuba square root. Rapport de recherche 3805, Institut National de Recherche en Informatique et en Automatique, 1999.

[57] Zuras, D. More on squaring and multiplying large integers. *IEEE Trans. Comput. 43*, 8 (1994), 899–908.

Unité de recherche INRIA Lorraine
LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)

Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38330 Montbonnot-St-Martin (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399