IE4a - Projet 2008

Arithmétique sur de grands entiers en C et en assembleur

ACEITUNO Jonathan
ALBAR Boris

4 Juin 2008

Table des matières

Introduction

L'objectif du projet d'IE4a est de construire un ensemble de fonctions réalisant des opérations primaires sur des grands nombres entiers sans perte de précision. Ces opérations (addition et multiplication) sont réalisées en C et en assembleur. En tant que mise en œuvre pratique de cette bibliothèque, une fonction de calcul de la factorielle d'un nombre doit être implémentée. Celle-ci permettra en plus de juger de la rapidité des opérations arithmétiques de base. C'est selon ce critère qu'est choisie une représentation adéquate pour le calcul sur des grands entiers.

Ce projet permet de mettre en application les bases de programmation C et assembleur vues dans le cadre de l'UE, les structures de données dynamiques et, pour notre part, l'optimisation par l'utilisation de jeux d'instructions spécifiques aux processeurs compatibles Pentium 4 (MMX, SSE...). La recherche dans le domaine du calcul en précision arbitraire a éclairé nos choix et a permis, outre la réalisation du projet, de découvrir une vaste étendue d'algorithmes et de théories mathématiques.

La première partie du rapport est centrée sur la description de notre réalisation sous toutes ses facettes. On y trouvera les justifications de nos choix de développement. La deuxième partie s'oriente sur tout ce qui tourne autour de la réalisation, comme les stratégies de test, la répartition des tâches ou un mode d'emploi succint.

Première partie

Description détaillée de la réalisation

Le choix de la plateforme de développement s'est porté sur un système compatible UNIX, en utilisant les outils de développement GNU (sauf Emacs). La justification de ce choix est évidente au vu de la concurrence[1].

Nous avons volontairement omis de détailler les fonctions telles qu'elles sont écrites car la plupart ne sont que des outils pour des fonctions plus générales.

Interprétation personnelle des objectifs

- Réaliser des opérations de base (addition, multiplication, soustraction, division, décalage, lecture et écriture dans une chaîne) sur des nombres arbitrairement grands.
- Rapidité d'exécution sur des nombres de taille moyenne (jusqu'à 200000 chiffres).
- Implémentation d'algorithmes non naïfs pour les opérations les plus coûteuses (multiplication, écriture).
- Utilisation des possibilités particulières des processeurs i686.

1 Représentation des données

Un nombre peut être considéré comme un polynôme P dans l'espace des polynômes $\mathbb{R}_p[X]$, c'est-à-dire sous la forme :

$$P = \sum_i a_i X^i$$

Ou comme une série nulle à partir d'un certain rang : $P = \sum_{i=0}^{\infty} a_i X^i$ tel qu'il existe $N \in \mathbb{N}$ avec : $\forall n > N,\ a_n = 0$.

L'indéterminé X d'un polynôme représente la base dans lequel le nombre s'exprime. Rappelons que les coefficients du polynôme sont fonction de la base choisie !

1.1 Représentation des grands entiers

Il existe un nombre infini de bases, néanmoins nous pouvons dégager deux types de bases possibles simplifiant les calculs sur ces polynômes dans l'optique du projet :

[1] Microsoft(TM) Windows(R)

D'une part, les bases du type 10^n : celles-ci ont l'avantage de simplifier et de rendre intuitifs l'écriture des polynômes, qui se rapproche de notre système de numération, mais ont pour inconvénient de demander des calculs plus lourds (par exemple au niveau de la gestion des retenues pour une addition). De plus, la mémoire étant scindée en élements de taille de type 2^n, une partie de cet espace mémoire reste inutilisée.

D'autre part, les bases du type 2^n : elles ont l'avantage de simplifier les calculs (par exemple, les additions avec retenue qui directement câblées dans le processeur) et d'utiliser toute la mémoire disponible. Néanmoins, les fonctions de conversion ne sont pas triviales.

Au vu des objectifs et de la disponibilité d'ordinateurs à architecture 32 bits (ceux qui seront utilisés pour la présentation), nous avons choisi la base 2^{32}.

1.2 Mise en œuvre de la structure de données en C

La mise en oeuvre de cette base dans une représentation en C est telle que nous la trouvons d'une part l'`union umot` (cf *ARPL.h*), composée de plusieurs structures donnant chacune une représentation différente (par exemple 32x1 bits, 8x4 bits, ou 1x32 bits sont des représentation possibles). Ceci étant rendu possible par l'utilisation d'une astuce peu connue du compilateur GCC (présente aussi sur d'autres compilateurs comme celui d'Intel), consistant à spécifier à la fin de chaque élément sa part en nombre de bits dans l'union (par exemple un `unsigned long a :8` comptera pour 8 bits dans l'union). L'`union` etant caractérisé en C par le partage d'un espace mémoire entre différentes structures de données, chaque représentation est accessible simultanément. Autrement dit, on pourra accéder aux 8 bits de poids fort sans masque de bits et de façon complètement triviale !

La deuxième structure entrant en jeu dans le programme est celle représentant un nombre en précision arbitraire : elle comporte un pointeur vers un tableau de `umot`, des entiers donnant le signe, la base, et la longueur du tableau précédent, c'est-à-dire le nombre de coefficients du polynôme. En voici le code :

```
1  struct arpn {
2      //* pointeur vers un tableau de n mots de (base) ↩
           bits *//
3      umot *mots;
4      //* nombre éléments alloués (n) *//
5      unsigned long maxdeg;
6      //* signe = +-1 *//
7      long signe;
8      //* base : 2^base *//
9      unsigned long base;
10 };
```

Listing 1 – Structure de données pour un grand nombre.

Des `typedef` permettent d'utiliser les structures décrites précédemment à la manière de types de variable, ce qui allège grandement la notation.

Notons au passage que la taille des éléments de la structure ci-dessus n'a pas été choisie au hasard. Elle a été choisie pour être alignée sur 16 bits. Autrement dit, la somme des tailles des éléments est égale à 16. Si la structure n'était pas alignée, le compilateur l'alignerait automatiquement en ajoutant des bits non-utilisés.

2 Conversion des données

Les nombres entrés par l'utilisateurs sont représentés sous forme de chaînes de caractère, que l'on nommera *représentation utilisateur*. Il a fallu construire et optimiser des fonctions de conversion de la représentation utilisateur à la *représentation interne*. Ces opérations sont très coûteuses en temps du fait de divisions sucessives par 10 nécessaires au changement de base. De plus, la complexité de l'opération est en $\mathcal{O}(n^2)$ pour l'algorithme naïf. Cela a motivé la recherche d'algorithmes de conversion possédant une complexité moindre.

2.1 Passage de la représentation utilisateur à la représentation interne (lecture)

On suppose la transformation d'une chaîne de caractères à un entier de précision arbitraire en base 10 triviale.

L'algorithme de lecture naïf est donc le suivant :

Algorithm 1 Lecture d'une chaîne de caractères en représentation interne : complexité $\mathcal{O}(n^2)$

Entrée : un entier $A = \sum_i a_i.10^i$
Sortie : un grand nombre $B = \sum_i b_i.2^{32.i}$
Variables : $total = \sum b_i$
Pour tout a_i **faire** :
  $total \leftarrow total + a_i * 10$
Déduire les b_i en connaissant la somme (total) et la taille des b_i par divisions successives.
Renvoyer B

On aurait pu implémenter un algorithme de lecture de complexité moindre, de type *diviser-pour-régner*[PZ], néanmoins, la rapidité des multiplications par 10 a rendu cette tâche moins urgente.

2.2 Passage de la représentation interne à la représentation utilisateur (écriture)

Il faut donc ici passer d'une base, par exemple 2^{32}, à la base 10 en chaînes de caractères. L'algorithme de conversion naïf est le suivant :

Algorithm 2 Écriture d'un grand nombre en représentation utilisateur : complexité $\mathcal{O}(n^2)$

Entrée : un grand nombre $A = \sum_{i=1}^{m} a_i.2^{32.i}$
Sortie : un entier $B = \sum_{i=1}^{n} b_i.10^i$
Variable : $reste = \sum a_i$
Pour i de 1 à n **faire** :
  $b_i \leftarrow (reste) \, mod \, 10$
  $reste \leftarrow reste/10$
Renvoyer B

Cet algorithme est limité du fait des divisions successives par 10, qui sont coûteuses en temps processeur.

Nous avons donc implémenté un algorithme récursif d'écriture de type *diviser-pour-régner* :

Algorithm 3 Écriture d'un grand nombre en représentation utilisateur : complexité $\mathcal{O}(\frac{1}{2^\alpha - 2} D(n))$ avec α et $D(n)$ dépendant du coût de la division

Entrée : un grand nombre $A = \sum_{i=1}^{m} a_i.2^{32.i}$
Sortie : un entier $B = \sum_{i=1}^{n} b_i.10^i$
$n' \leftarrow m/2$
$Q \leftarrow A/10^{n'}$
$R \leftarrow A \, mod \, 10^{n'}$
$(b_{n'-1}, \ldots, b_0) \leftarrow Ecriture(R)$
$(b_n, \ldots, b_{n'}) \leftarrow Ecriture(Q)$
Renvoyer B.

Des précisions concernant la division utilisée viendront plus tard.

3 Opérations arithmétiques sur les grands nombres

3.1 Addition

3.1.1 Algorithme

L'algorithme est celui utilisé depuis l'école primaire :

Algorithm 4 Addition de deux grands nombres

Entrée : deux grands nombres $A = \sum_{i=1}^{m} a_i.2^{32.i}$ et $B = \sum_{i=1}^{n} b_i.2^{32.i}$
Sortie : un grand nombre $C = \sum_{i=1}^{max(n,m)} c_i.2^{32.i}$
Variable : $retenue = 0$
$n' \leftarrow min(n, m)$
Pour i de 1 à n' **faire :**
  $c_i \leftarrow a_i + b_i + retenue$
  $retenue \leftarrow (a_i + b_i)/2^{32}$
Retourner C

3.1.2 Note pour l'addition en assembleur avec MMX

L'addition avec MMX comporte une particularité : il n'y a pas de modification de drapeau pour l'addition MMX. On calcule la retenue à partir des bits les plus significatifs des deux mots et du bit le moins significatif du résultat correspondant par la formule : `retenue = (`a_i`&`b_i`) | ((!`c_i`) & (`a_i`|`b_i`))`.

3.2 Multiplication

Nous avons à disposition une ribambelle d'algorithmes différents : le naïf, l'algorithme de Karatsuba, celui de Toom-Cook, celui utilisant les NTT (Numerical Theoretic Transforms, *ie* des FFT sur un corps fini avec une arithmétique modulaire). Nous avons implémenté les deux premiers.

L'algorithme de Karatsuba n'étant plus rapide que l'algorithme naïf qu'au dessus d'un certain seuil (défini empiriquement suivant le processeur), l'algorithme est choisi suivant la taille des opérandes.

3.2.1 Algorithme naïf de multiplication

C'est aussi le même qu'à l'école. Rappelons-nous :

Algorithm 5 Multiplication naïve de deux grands nombres en complexité $\mathcal{O}(n^2)$

Entrée : deux grands nombres $A = \sum_{i=1}^{m} a_i.2^{32.i}$ et $B = \sum_{i=1}^{n} b_i.2^{32.i}$
Sortie : un grand nombre $C = \sum_{i=1}^{n+m} c_i.2^{32.i}$
$(c_n, \ldots, c_1) \leftarrow (a_1 * B)$
Pour i de 1 à m **faire :**
  $(c_{i+m}, \ldots, c_1) \leftarrow (0, c_{i-1+m}, \ldots, c_1) + a_i * B$
Retourner C

3.2.2 Algorithme de multiplication de Karatsuba

L'algorithme de Karatsuba se base sur une jolie astuce. Avec $P = a_1X + a_0$ et $Q = b_1X + b_0$, on a le produit :

$$P.Q = (a_1X + a_0)(b_1X + b_0) = a_1b_1X^2 + a_0b_1X + b_0a_1X + a_0b_0$$

Avec cette formule, on calcule quatre produits. C'est beaucoup trop. En bridant, on peut réduire le produit à trois multiplications seulement :

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

De manière récursive, on parvient à étendre cette propriété à n'importe quel polynôme, donc à l'écriture d'un nombre sur n'importe quelle base. La mise en œuvre de cet algorithme est la suivante :

Algorithm 6 Multiplication de deux grands nombres par l'algorithme de Karatsuba en complexité $\mathcal{O}(n^{\log_3 2})$

Entrée : deux grands nombres $A = \sum_{i=1}^{m} \alpha_i.2^{32.i}$ et $B = \sum_{i=1}^{n} \beta_i.2^{32.i}$
Sortie : un grand nombre $C = \sum_{i=1}^{n+m} \gamma_i.2^{32.i}$
On pose : $A = a_1.2^{\frac{min(m,n)}{2}*32} + a_0$ **et** $B = b_1.2^{\frac{min(m,n)}{2}*32} + b_0$
$gros \leftarrow Karatsuba(a_1, b_1)$
$petit \leftarrow Karatsuba(a_0, b_0)$
$moyen \leftarrow Karatsuba(a_0 + a_1, b_0 + b_1)$
$moyen \leftarrow moyen - petit - gros$
Retourner $C = gros.2^{min(m,n)*32} + moyen.2^{\frac{min(m,n)}{2}*32} + petit$

3.2.3 Performances

Nous avons testé les deux algorithmes sur des nombres équilibrés (de même taille) de différents calibres sur un PC *Pentium 4 3,00 GHz* :

Nombre de chiffres	Naïf (sec)	Karatsuba (sec)
1000	0.000078	0.000038
5000	0.001931	0.001213
8000	0.004718	0.002644
12000	0.018077	0.008590
20000	0.031245	0.010562
100000	0.767737	0.167462

On remarque que l'algorithme de Karatsuba devient réellement efficace pour des assez grands nombres, mais pas pour 1000! qui n'utilise que des multiplications par des petits nombres.

Cette efficacité se perd lorsque les nombres deviennent très déséquilibrés.

4 Division

Il existe plusieurs algorithmes de division. Néanmoins, ceux-ci sont nettement plus complexes à implémenter que ceux pour la multiplication. On peut citer parmi eux l'algorithme[1] de *Knuth*[2][DEK], celui de *Burnikel-Zieger* et la *méthode de Newton* (méthode itérative à convergence quadratique qui donne une approximation de l'inverse d'une série).

[1] Algorithme D. Partie 4.3.1.
[2] Qui est aussi l'inventeur de TEX!

Nous avons implémenté l'algorithme de Knuth dans le cas le plus favorable (division d'un nombre A de $2n$ mots par un nombre B de n mots tel que B soit normalisé, autrement dit, que le reste de la division tienne sur n mots). Cette division sert pour l'algorithme d'écriture en *diviser-pour-régner*.

Voici l'algorithme :

Algorithm 7 Division de deux grands nombres avec B normalisé

Entrée : deux grands nombres $A = \sum_{i=1}^{2n} a_i.2^{32.i}$ et $B = \sum_{i=1}^{n} b_i.2^{32.i}$ normalisé
Sortie : un grand nombre $C = \sum_{i=1}^{n} c_i.2^{32.i}$ pour le quotient et le reste dans A
Pour i de n-1 à 0 **faire :**
  $c_i \leftarrow (a_{n+i} * 2^{32} + a_{n+i-1})/\ b_n$
  $A \leftarrow A - c_i B * 2^{32*i}$
Si A<0
  $c_i \leftarrow c_i - 1$
  $A \leftarrow A + B * 2^{32*i}$
  Si A<0
    $c_i \leftarrow c_i - 1$
    $A \leftarrow A + B * 2^{32*i}$
Retourner C

On pourrait remplacer les deux *Si* ci-dessus par une boucle néanmoins le théorème B partie 4.3.1 de l'ouvrage de Knuth[DEK] nous indique qu'elle n'est parcourue au maximum que deux fois dans la condition que B soit normalisé.

Deuxième partie

Autour de la réalisation

1 Stratégie et modalités de validation

Nous avons utilisé la meilleure librairie C de calcul en précision arbitraire, *GMP*, afin de vérifier nos calculs (sur des nombres comme 100 000!).

Les tests sur des nombres de "moyenne importance" (20000 chiffres) ont été effectués sur des puissances de 10 car, en base 2^{32}, ces nombres s'écrivent de manière non-triviale et génèrent une multitude de cas particuliers. De plus, leur résultat est très simple à vérifier.

Le programme donné en exemple a aussi permis de vérifier la bonne marche de la librairie et d'effectuer des corrections de dernière minute.

Voici un ordre de grandeur des résultats obtenus avec le programme sur un *Pentium-M* cadencé à 1,70 GHz :

1000 ! Sans conversion : 0.402296 ms, avec conversion : 1.918406 ms.

10000 ! Sans conversion : 52.178376 ms, avec conversion : 205.057436 ms.

100000 ! Sans conversion : 6.123453 sec, avec conversion : 28,773915 sec.

2 Répartition des tâches

- Recherche bibliographique sur papier : Jonathan ACEITUNO
- Recherche bibliographique sur le plan algorithmique (rapports de recherche de différents instituts) : Boris ALBAR
- Mise au point de la structure de données (et de la représentation) : Jonathan ACEITUNO et Boris ALBAR
- Ecriture des fonctions "naïves" (C et ASM) : Jonathan ACEITUNO
- Ecriture des fonctions "non-naïves" (C et ASM avec MMX/SSE/SSE2) : Boris ALBAR
- Rédaction du rapport : Jonathan ACEITUNO et Boris ALBAR
- Rédaction de l'annexe : Boris ALBAR (et Jonathan ACEITUNO pour la mise en page !)
- Ecriture de la fonction `main()` et de l'outil `tools/cpuspeed` : Jonathan ACEITUNO

3 Mode d'emploi

3.1 Utiliser le programme d'exemple

Ce projet est fait pour être exécuté et étudié sous environnement compatible UNIX. Si vous avez une distribution Linux, tant mieux, et si vous êtes allergique, essayez tout de même. Sur Windows, un shell Cygwin fera peut-être l'affaire.

Placez-vous dans le répertoire de l'archive du projet. Décompressez l'archive :

```
1 tar xvjf projet_ie4a-albar_aceituno.tar.bz2
```

Ensuite, placez-vous dans le répertoire du projet :

```
1 cd projet_ie4a
```

Vous devez maintenant *compiler* le projet. En effet, il est fourni avec un fichier exécutable (`arpcalc`) mais celui-ci n'est certainement pas optimisé pour votre architecture (les calculs de temps seront erronnés). Ce fichier exécutable est uniquement là au cas où vous ne parvenez pas à compiler le projet.

Essayons tout de même ! Pour commencer, nettoyons tout cela :

```
1 make clean
```

Ensuite, il faut compiler toutes les sources. Attention, des avertissements vont apparaître sous vos yeux ébahis. Ceux-ci ne dénoncent pas une fragilité du code, nous avons activé tous les avertissements possibles dans GCC. Compilez avec :

```
1 make
```

Après une attente, voici un moment bien mérité. Vous allez pouvoir jouer avec le programme si tout s'est bien passé (ce qui est le cas, bien entendu). Lancez le programme en tapant :

```
1 ./arpcalc
```

Vous verrez que le programme ne calcule rien et se termine après quelques explications. Ce sont celles-ci que vous pouvez suivre afin de mener à bien des expérimentations diaboliques.

La syntaxe du programme est la suivante :

```
1 ./arpcalc <operation> [<arg1> [<arg2>]]
```

Les opérations possibles sont les suivantes :

factorial Factorielle classique. Utilisable de la façon suivante : `./arpcalc factorial 1000`

add Addition naïve en Assembleur. Utilisable de la façon suivante : `./arpcalc add 1 2`

add_c Addition naïve en C. Utilisable de la façon suivante : `./arpcalc add_c 1 2`

multiply Multiplication naïve en Assembleur. Utilisable de la façon suivante : `./arpcalc multiply 1 2`

multiply_c Multiplication naïve en C. Utilisable de la façon suivante : `./arpcalc multiply_c 1 2`

test Le test ultime (calculer 1000 fois 1000 ! puis mettre le dernier résultat dans un fichier resultat.txt). Utilisable de la façon suivante : `./arpcalc test`

Le programme donne le résultat et les temps (temps avec conversion en représentation utilisateur, et temps sans cette conversion).

Nous n'avons pas cru bon de mettre à disposition les multiplications avec l'algorithme de Karatsuba pour des raisons de convivialité (pour tester l'algorithme efficacement, des nombres d'au moins quelques milliers de chiffres sont nécessaires).

Bon amusement !

3.2 Utiliser la librairie

Vous souhaitez utiliser cette librairie dans votre programme ? Voici la liste des fonctions à utiliser avec leurs paramètres :

- `allocnb(arpn *nombre, unsigned long taille)` : Alloue un grand nombre de taille donnée.
- `freenb(arpn *nombre)` : Libère un grand nombre. A utiliser sans modération !
- `arpn ConvToArpn(char *nombreUtilisateur)` : Convertit une chaîne de caractères en grand nombre en base 2^{32} (arrête le programme s'il y a une erreur).
- `unsigned char *ConvToString_predc(arpn nombre)` : Convertit un grand nombre en chaîne de caractères.
- `arpn addition_naive(arpn n1, arpn n2)` : Additionne deux grands nombres.

- `arpn multiplication_naive(arpn n1, arpn n2)` : Multiplie deux grands nombres.
- `arpn factorial_naive(unsigned long nombre)` : Calcule la factorielle d'un (petit[1]) nombre.

La liste ne s'arrête pas là, mais le reste est à explorer seul (ou entre ami(e)s).

Un petit programme d'exemple, supposant qu'on le place dans le répertoire de la librairie (dans ce cas, enlevez le fichier main.c, sinon il y aura deux fonctions `main`) :

```
1  #include "ARPL.h"
2
3  // Addition de 123 et de 1000!
4  int main() {
5      // On prend le premier nombre...
6      arpn nombre1 = ConvToArpn("123");
7      // On prend le deuxième...
8      arpn nombre2 = factorial_naive(1000);
9      // On fait l'addition...
10     arpn resultat = addition_naive(nombre1, nombre2);
11
12     // On convertit le résultat en chaîne de ↩
           caractères.
13     unsigned char *resultatString = ↩
           ConvToString_predc(resultat);
14
15     // On imprime ce résultat à l'écran.
16     printf("%s\n", resultatString);
17
18     // Important, il faut tout libérer.
19     freenb(&nombre1);
20     freenb(&nombre2);
21     freenb(&resultat);
22
23     // Tout s'est bien passé.
24     return 0;
25 }
```

Sympathique, n'est-ce pas ?

4 Perspectives d'améliorations

D'autres optimisations auraient pu être implémentées, tant au niveau de l'utilisation du matériel (utilisation des extensions *SSE3*, utilisation des capacités des *GPU* des cartes graphiques récentes sous réserve d'une implémentation des opérations en précision arbitraire sur des nombres flottants) que de l'occupation mémoire (supprimer l'utilisation de mémoire de manière récursive dans la fonction d'écriture en *divide-and-conquer*) ou des algorithmes utilisés (par

[1] (mais tout de même `long`)

exemple, division de Burnikel et Zieger[CBJZ], trouvée en 1998 ; ou encore le tout dernier[2] algorithme de multiplication de Fürer).

Le programme d'exemple aurait pu être un vrai invite de commandes dynamique, mais les semaines sont courtes...

Au niveau de l'utilisation de la librairie, des efforts pour en faire un vrai module sont encore à faire (mais cela dépasse le cadre du projet)...

Conclusion

Il a été bénéfique de réaliser ce projet, tant au niveau mathématique et théorique que technique. L'utilisation de l'assembleur AT&T a été, pour certains, une découverte, et pour d'autres, un retour aux sources. La recherche effrénée de l'optimisation et de l'efficacité fut une entreprise intéressante, car elle est trop souvent négligée de nos jours au vu de la puissance des ordinateurs actuels. Cela nous a permis de nous prendre pour des John Carmack en herbe.

L'écriture d'un rapport avec LaTeX nous a permis de le maîtriser encore un peu plus cet outil fantastique. Ce projet montre vraiment à quel point le logiciel libre et les systèmes UNIX sont puissants et agréables à utiliser. Toute alternative est vouée, à terme, à l'échec (sauf les systèmes BSD (sauf Mac OS X (sauf Darwin (sauf XNU (sauf Mach)))))).

Troisième partie

Annexe

A Bibliographie

Références

[DEK] D.E. Knuth, Art of Computer Programming vol. 2. Addison-Wesley.

[PZ] P. Zimmermann, Arithmétique en précision arbitraire. Rapport INRIA n°2372.

[MQ] Michel Quercia, Calcul multiprécision.

[TGPLM] T. Granlund, P.L. Mongomery, Division by Invariant Integers using Multiplication.

[DJB] D.J. Bernstein, Scaled Remainder Trees.

[CBJZ] C. Burnikel, J. Ziegler, Fast Recursive Division.

[RBPZ] R. Brent, P. Zimmermann, Modern Computer Arithmetic.

[2] Le nec-plus-ultra en complexité !

B Code source

Le code source est imprimé séparément.