Département de formation doctorale en informatique **École doctorale IAEM Lorraine**
UFR STMIA

Speeding up Integer Multiplication and Factorization

Améliorations de la multiplication et de la factorisation d'entier

THÈSE

présentée et soutenue publiquement le 28 janvier 2010

pour l'obtention du

Doctorat de l'université Henri Poincaré – Nancy 1
(spécialité informatique)

par

Alexander Kruppa

Composition du jury

Rapporteurs : Karim Belabas (Professeur, Université Bordeaux 1)
Antoine Joux (Professeur associé (P.A.S.T.), Université de Versailles)

Membres du jury : Isabelle Debled-Rennesson (Maître de conférences, Université Henri Poincaré)
Marc Joye (Cryptographer, Thomson R&D)
Arjen Lenstra (Professeur, EPFL)
Herman te Riele (Directeur de Recherche, CWI)
Paul Zimmermann (Directeur de Recherche, INRIA)

Laboratoire Lorrain de Recherche en Informatique et ses Applications — UMR 7503

Mis en page avec la classe thloria.

Contents

List of Figures

List of Tables

Acknowledgments

I have profited enormously from working in an outstanding group of researchers for the duration of my thesis, and I'd like to thank all the members of the CACAO/CARAMEL team at LORIA whom I've had the pleasure of meeting during my stay. I thank especially Jérémy Detrey, Antonio Vera and Sylvain Chevillard for their help with writing the French part of the thesis. The countless chats with Pierrick Gaudry were a delightful pastime and provided invaluable discussion of the algorithms presented here.

I've had the honor and pleasure of working with Peter Lawrence Montgomery and thank him for offering me this opportunity. Few have shaped modern integer factoring as much as he did, and reading his earlier work is a large part of what originally got me interested in the subject.

Above all I am indebted to my thesis advisor Paul Zimmermann for accepting me as his student and his continual support and expert advice on all aspects of this thesis.

Introduction

Integer multiplication is used in practically every arithmetic algorithm, and problems in algorithmic number theory in particular often require rapid multiplication of very large integers. Factorization of integers is one of the fundamental problems in number theory and gained significant practical importance with the advent of the RSA public-key cryptographic system whose security relies on the difficulty of factoring. This thesis presents improvements to the Schönhage-Strassen algorithm for multiplication of large integers, to the P–1 and P+1 methods of factorization which quickly find prime factors p where $p-1$ or $p+1$ have themselves no large prime factors, and to the Number Field Sieve which is the fastest algorithm for factoring composite integers which have no easy to find prime factors, such as occur in cryptographic applications.

Integer multiplication is ubiquitous, and multiplication of large integers occurs frequently enough in scientific computation that it is somewhat surprising that the first algorithm faster than the $O(n^2)$ bit operations required by the trivial grammar-school multiplication was discovered only in 1962. In that year, Karatsuba and Ofman [51] showed how to reduce the problem of multiplying n-bit numbers to three multiplications of $n/2$-bit numbers, achieving asymptotic complexity $O(n^{1.585})$. A year later, Toom [97] generalized Karatsuba and Ofman's algorithm by expressing integer multiplication by polynomial multiplication and using polynomial evaluation, point-wise multiplication, and interpolation to compute the product polynomial. This allows reducing the computation to $2k-1$ pieces of n/k bits each, for asymptotic cost $O(n^{\log_k(2k-1)})$ for k fixed. In principle, this permits any exponent $1+\epsilon$ in the asymptotic cost function, however, large k are not efficient for input numbers of realistic size as the cost of evaluation and interpolation would dominate. In 1971, Schönhage and Strassen [90] essentially solved the problem of fast integer multiplication by using the Fast Fourier Transform (FFT), discovered by Cooley and Tukey in 1965 [29], to perform the required convolution product. Their method uses time only $O(n\log(n)\log(\log(n)))$ to multiply n-bit numbers. Many programming libraries for multiple precision arithmetic offer a fast integer multiplication algorithm. One such library is the Gnu Multiple Precision arithmetic library (GMP) [49], developed mainly by Torbjörn Granlund. It is widely used and enjoys a reputation for being exceptionally fast, both due to careful choice of algorithms and highly optimized implementation. It uses the Schönhage-Strassen algorithm for multiplication of very large integers. One application where the large-integer multiplication of GMP is used extensively is GMP-ECM [103], an implementation of the P–1, P+1, and Elliptic Curve methods of factorization developed mainly by Paul Zimmermann.

Integer factoring is an ancient problem in number theory. It ceased to be a question of purely academic interest and turned into a matter of significant economic relevance with the publication of the now widely used Rivest-Shamir-Adleman (RSA) public-key cryptographic system [85] which relies on the intractability of factoring large integers. Fermat's Little Theorem states

$$a^{p-1} \equiv 1 \pmod{p}$$

for any prime p and $p \nmid a$, which Euler extended to composite moduli by

$$a^{\phi(N)} \equiv 1 \pmod N$$

for $\gcd(a, N) = 1$, where $\phi(N)$ is the Euler totient function, defined by

$$\phi(p_1^{\nu_1} \cdots p_k^{\nu_k}) = (p_1 - 1)p_1^{\nu_1 - 1} \cdots (p_k - 1)p_k^{\nu_k - 1}$$

with $p_1, \ldots, p_k$ distinct primes and $\nu_1, \ldots, \nu_k$ positive integers. Thus, given the prime factorization of N, it is easy to compute $\phi(N)$. RSA uses a public key for encryption, consisting of an odd composite modulus $N = pq$ with p, q primes of roughly equal size, and a public exponent e. For decryption it uses a private key, consisting of N again and an integer d such that

$$de \equiv 1 \pmod{\phi(N)},$$

i.e., $de = k\phi(N) + 1$ for some integer k. It encrypts a message expressed as an integer $1 < m < N$ by

$$c = m^e \bmod N,$$

and decrypts c to recover the original message by

$$c^d \bmod N = m^{de} \bmod N = m^{k\phi(N)+1} \bmod N = m.$$

Factoring N reveals d and so breaks the encryption. The keys for RSA are therefore chosen to be as difficult as possible to factor with known algorithms, and of a size that is expected to be out of reach for computing resources available during the key's intended lifetime.

Factoring algorithms can be divided into two classes: special-purpose algorithms, and general-purpose algorithms. The former make use of certain properties of the prime factors, most commonly their size, and their run time hardly depends on the size of the input number (usually only like the complexity of integer multiplication), but greatly on whether its factors have the desired property. The run time of general-purpose algorithms depends almost exclusively on the size of the input number, and not on any special properties of the factors. RSA keys are chosen to be resistant to special-purpose methods so that only general-purpose algorithms are relevant to their security. The best currently known factoring algorithm for attacking RSA is the Number Field Sieve (NFS) with time complexity conjectured to be in $L_N[1/3, (64/9)^{1/3}]$, where

$$L_x[\alpha, c] = e^{(c+o(1)) \log(x)^{\alpha} \log(\log(x))^{1-\alpha}}$$

for $x \to \infty$, and the cost of factoring with the NFS is the major criterion for rating the security of RSA key sizes. In spite of being useless for breaking RSA directly, special-purpose factoring algorithms are still of great interest, on one hand for factoring numbers that aren't RSA keys and may have easy-to-find factors, and as a sub-routine for the NFS.

A central concept to modern integer factoring algorithms is that of *smoothness*: an integer is called B-smooth if no prime factor exceeding B divides it, and B-powersmooth if no prime or prime power exceeding B divides it.

The P–1 algorithm published by Pollard in 1974 [80] was the first of a class of special-purpose factoring algorithms that find a prime factor p of N quickly if the order of a finite group defined over $\mathbb{F}_p$ is smooth. In the case of the P–1 algorithm, the group is simply the group of units of $\mathbb{F}_p$ and has order $p-1$. Stage 1 of his algorithm chooses some integer x_0 coprime to N and computes $x_0^e \bmod N$, with e including all primes and prime powers up to a chosen bound B_1. If $p - 1$ is B_1-powersmooth and thus divides e, then $x_0^e \equiv 1 \pmod p$, and $\gcd(x_0^e - 1, N)$ usually reveals p

except when this gcd is composite. Pollard further proposes a stage 2 which looks for a collision modulo p in selected powers of x_0^e which allows him to discover prime factors p where $p-1$ contains a single prime between B_1 and a second bound B_2, but is otherwise B_1-powersmooth. He shows that a prime factor p can be found in time $O\big(\sqrt{p}M(\log(N))\big)$ if the collision detection is done by polynomial multipoint evaluation with a fast FFT-based multiplication routine. Williams [101] extends Pollard's P–1 idea to the P+1 method which finds a prime p quickly if one of $p-1$ or $p+1$ is smooth. The P–1 and P+1 algorithms sometimes find surprisingly large factors quickly if $p-1$ or $p+1$ happens to be smooth enough, but if both group orders contain a large prime factor, then these methods are powerless. For example, a fraction of about $1-\log(2)$ of integers up to N is $N^{1/2}$-smooth (see Section 5.3.1), so roughly half of the 40-digit primes p have a prime factor exceeding 20 digits in both $p-1$ and $p+1$, which makes these primes impractical to find with either method.

Pollard's idea for an asymptotically fast FFT stage 2 to the P–1 algorithm was first implemented by Montgomery and Silverman [74]. The authors suggest several ideas to speed up their algorithm further, and to adapt it to the P+1 factoring method.

The asymptotically fastest special-purpose factoring algorithm is the Elliptic Curve Method (ECM) by H. W. Lenstra Jr. [62] which can be viewed as a generalization of P–1 and P+1 in that it works in a group of points on an elliptic curve over $\mathbb{F}_p$ with group order in $[p-2\sqrt{p}+1, p+2\sqrt{p}+1]$, depending on the curve parameters. It has the major advantage that it can keep trying different curves until a lucky curve with smooth group order is found. With optimal choice of parameters, ECM has conjectured complexity of $L_p[1/2, 2]$ to find a prime factor p which, in the worst case of $p \approx \sqrt{N}$, leads to $L_N[1/2, 1]$, making it the currently only special-purpose factoring algorithm with sub-exponential running time. Brent [15] and Montgomery [65] present stage 2 extensions for ECM, and Montgomery [74] develops an FFT stage 2.

Even though ECM has far superior asymptotic complexity and the P–1 and P+1 methods act, in a way, merely as two particular attempts at a smooth group order among the nearly endless number of such trials offered by ECM, the older methods have some advantages that still keep them useful. One advantage is sheer speed. The arithmetic during stage 1 is much simpler for P–1 and P+1 than for ECM so that with comparable parameters, less CPU time is spent. Another advantage is that for P–1 and P+1, a much faster FFT stage 2 algorithm is possible, due to the fact that $\mathbb{Z}/N\mathbb{Z}$ (or a quadratic extension thereof for P+1) has ring structure, which is not the case for the group of points on an elliptic curve. The ring structure permits a particularly efficient polynomial multipoint evaluation algorithm, allowing stage 2 to run with much less CPU and memory usage than is possible for the FFT extension for ECM. Finally, for some input numbers $p-1$ is known to have a divisor n (e.g., cyclotomic numbers $\phi_n(x)$ for n, $x \in \mathbb{N}$, unless $p \mid n$), which increases the probability that $(p-1)/n$ is smooth. These advantages make it quite reasonable to give P–1 and P+1 a try before moving on to the fundamentally more powerful, but also computationally more expensive ECM algorithm.

Currently the best factoring algorithm for attacking RSA moduli (and other numbers without small or otherwise easy to find prime factors) is the Number Field Sieve. It was first proposed by Pollard in 1988 [82] and originally required that the integer to be factored has a simple algebraic form such as $a^n \pm c$ with small a and c, but in the following years was extended to factoring general integers [20]. It factors an integer N in conjectured time $L_N[1/3, c]$ with $c = (32/9)^{1/3}$ for input numbers of simple enough form, in which case the algorithm is called the Special Number Field Sieve (SNFS), or with $c = (64/9)^{1/3}$ for general integers, then called General Number Field Sieve (GNFS). An early success for the SNFS was the factorization of the 9th Fermat number $F_9 = 2^{2^9} + 1$ in 1990 [61] and for the GNFS that of a 130-digit RSA factoring challenge number in 1996. In January 2010, the largest SNFS factorization is that of $2^{1039} - 1$ in 2007 [2], whereas the GNFS record is the factorization of a 768-bit (232-digit) RSA challenge number in 2009 [55].

Like other general-purpose factoring algorithms, the NFS factors an integer N by finding a congruence of squares, $x^2 \equiv y^2 \pmod{N}$ with $x \neq \pm y \pmod{N}$, from which a non-trivial factor of N is obtained by taking $\gcd(x+y, N)$. The values of x and y are found by collecting a large set of "relations," which are essentially pairs of smooth integers, from which a subset can be chosen so that in their product each prime occurs in even exponent, resulting in squares. The most time consuming part of the NFS (in both Special and General variant) is the relation collection phase which examines a very large number of polynomial values to look for smooth values by use of sieving techniques and other factoring algorithms. Here, an integer n is considered smooth if it contains only prime factors up to a sieving bound B, except for up to k integers up to a large prime bound L. The sieving routine reports such n where the cofactor after dividing out the primes up B is small enough, say below L^k. These cofactors need to be factored to test if any prime factor exceeds L. The memory consumption of the sieving increases with B, and for large-scale factorizations the available memory frequently limits B so that L and perhaps k need to be increased to allow sufficiently many values to pass as smooth. This way, a very large number of such cofactors occur during the sieving for an NFS factorization, and algorithms optimized for high-throughput factorization of small integers need to be used to avoid the cofactorization becoming the bottleneck of the sieving process. The Elliptic Curve Method is frequently suggested for this purpose [7] [41], and the P–1 and P+1 methods are likewise good candidates.

Contributions

In joint work with Pierrick Gaudry and Paul Zimmermann, we developed an improved implementation of the Schönhage-Strassen integer multiplication algorithm, based on the code in GMP version 4.1.4 which was written by Paul Zimmermann. The new implementation improves cache locality during the FFT phase, increases the possible convolution length for given input size, and uses fine-grained choice of convolution length and other parameters depending on the size of the input numbers. It is described in Chapter 1. These improvements resulted in a factor 2 speedup over the code in GMP 4.1.4.

In joint work with Montgomery, we have implemented an improved version of the P–1 and P+1 stage 2 algorithm that implements the ideas mentioned in [74] and other improvements. The implementation is based on GMP-ECM and is described in Chapter 2.

A library for high-throughput factorization of integers up to 38 digits, using the P–1, P+1, and ECM algorithms, has been written for use in the NFS siever program developed in the context of the CADO project (*Crible algébrique: distribution, optimisation*). Chapter 3 contains an overview of the NFS algorithm. The details of the small-integer factoring implementation are found in Chapter 4, and its cost-efficiency is compared to proposed hardware implementations of ECM for NFS. An outline of methods to estimate the success probability of finding factors with the P–1, P+1 and ECM algorithms in cofactors produced by the NFS sieving step is given in Chapter 5.

Notation

An overview of mathematical notation used throughout the thesis. Most of it follows common usage, but is listed here for reference.

Sets:

$\mathbb{C}$	The complex numbers
$\mathbb{N}$	The non-negative integers
$\mathbb{P}$	The rational primes
$\mathbb{Q}$	The rational numbers
$\mathbb{Z}$	The integers

Relations:

$a \mid b$	a divides b, there is an integer k such that $b = ka$
$a \nmid b$	a does not divide b
$a \perp b$	a is coprime to b, $\gcd(a, b) = 1$
$a \mid\mid b$	a divides b exactly, $a \mid b$ and $b/a \perp a$

Functions:

$\log(x)$	The natural logarithm of x
$\log_b(x)$	The logarithm of x to base b
$\phi(n)$	The Euler totient function, the number of integers $1 \leq k < n$ with $k \perp n$
$\left(\frac{a}{p}\right)$	The Legendre symbol for $a \pmod p$
$\pi(n)$	The prime counting function, the number of primes not exceeding n
$\lfloor n \rfloor$	The floor function, the largest integer k with $k \leq n$
$\lceil n \rceil$	The ceiling function, the smallest integer k with $k \geq n$
$\lfloor n \rceil$	The nearest integer function, $\lfloor n + \frac{1}{2} \rfloor$
$\mathrm{Val}_p(n)$	The p-adic valuation of n

Other frequently used symbols:

β	The machine word base, typically $\beta = 2^{32}$ or $\beta = 2^{64}$

Chapter 1

Integer Multiplication with Schönhage-Strassen's Algorithm

1.1 Introduction

The text in Sections 1.2 and 1.3 of this chapter is based on joint work with P. Gaudry and P. Zimmermann which was published in [46].

Multiplication of integers is one of the most basic operations in arithmetic and as such plays a vital role in computational arithmetic. For many algorithms the time spent performing multiplications dominates. Numerous other operations can be reduced to integer multiplication: modular multiplication (by Barrett reduction [5] or Montgomery's REDC [64]), polynomial multiplication, multi-point evaluation and factorization, or root-finding by iterative methods.

In several applications, the integers to be multiplied are large, in particular when reducing polynomial arithmetic to integer multiplication [99, 8.4], for high-precision evaluation of constants, primality testing or integer factorization. Allan Steel [94] gives an overview of algorithms that can be implemented efficiently by reduction to multiplication. For these, a multiplication algorithm with low asymptotic complexity is required to make large operand sizes practical.

Given two multiple precision non-negative integers $a = \sum_{i=0}^{m} a_i w^i$ and $b = \sum_{j=0}^{n} b_j w^j$ with word base w and $0 \leq a_i, b_j < w$, we would like to compute the integer $c = \sum_{k=0}^{m+n+1} c_k w^k = ab$ with $0 \leq c_k < w$. The convolution product of the sums for a and b yields

$$\begin{aligned} ab &= \sum_{i=0}^{m} a_i w^i \sum_{j=0}^{n} b_j w^j \qquad (1.1) \\ &= \sum_{k=0}^{m+n} w^k \sum_{j=\max(0,k-m)}^{\min(k,n)} a_{k-j} b_j . \end{aligned}$$

Hence we can set

$$\hat{c}_k := \sum_{j=\max(0,k-m)}^{\min(k,n)} a_{k-j} b_j \qquad (1.2)$$

and have $c = \sum_{k=0}^{m+n+1} c_k w^k = \sum_{k=0}^{m+n} \hat{c}_k w^k$, however $c_k \neq \hat{c}_k$ in general since the $\hat{c}_k$ may be larger than w (but they do not exceed $\min(m+1, n+1) \cdot (w-1)^2$). The desired c_k values can be obtained by an additional step commonly called "carry propagation:" set $\hat{c}_{m+n+1} := 0$ and then, for $k = 0, \ldots, m+n$ in sequence,

$$\begin{aligned} \hat{c}_{k+1} &:= \hat{c}_{k+1} + \left\lfloor \frac{\hat{c}_k}{w} \right\rfloor \\ \hat{c}_k &:= \hat{c}_k \bmod w. \end{aligned}$$

The sum $\sum_{k=0}^{m+n+1} \hat{c}_k w^k$ is invariant under this process, and finally all $\hat{c}_k < w$ so we can set $c_k := \hat{c}_k$.

The steps of decomposing the input integers into a sequence of digits in a convenient word base w, performing a convolution product on these sequences, and obtaining the correct sequence of digits of the product by carry propagation is common to multiple precision integer multiplication algorithms. With suitable choice of w, e.g., a power of 2 on a binary computer, the two steps of decomposition and carry propagation are inexpensive, requiring only $O(n+m)$ additions or assignments of integers of size $O(\log(w))$. The main difference between multiplication algorithms is how the convolution product is computed, and this is where they greatly differ in speed.

The most simple convolution algorithm, the "grammar-school" method, computes each $\hat{c}_k$ individually by the sum (1.2). This involves $(m+1)(n+1)$ multiplications of single digit (in base w) integers a_i and b_j and about as many additions; assuming constant cost for these operations, the algorithm has complexity in $O(mn)$, or for m and n of equal size, $O(n^2)$.

1.1.1 The Karatsuba Algorithm

The first algorithm to offer better asymptotic complexity than the grammar-school method was introduced in 1962 by A. Karatsuba and Yu. Ofman [51] (English translation in [52]), commonly called Karatsuba's method. The idea is to compute a product of two $2n$-word inputs by three products of n-word values (whereas the grammar-school method would require four such products). Writing $a = a_1 w + a_0$ and $b = b_1 w + b_0$, where $0 \leq a_0, a_1, b_0, b_1 < w$, we can compute the convolution product $\hat{c}_2 w^2 + \hat{c}_1 w + \hat{c}_0 = a_1 b_1 w^2 + (a_0 b_1 + a_1 b_0) w + a_0 b_0$ via

$$\begin{aligned} \hat{c}_2 &= a_1 b_1 \\ \hat{c}_0 &= a_0 b_0 \\ \hat{c}_1 &= (a_1 + a_0)(b_1 + b_0) - \hat{c}_2 - \hat{c}_0. \end{aligned}$$

This method can be applied recursively, where the size of the numbers to be multiplied is about halved in each recursive step, until they are small enough for the final multiplications to be carried out by elementary means, such as one-word multiplication or the grammar-school method. Assuming a threshold of one machine word for these small multiplications so that they have constant cost, Karatsuba's method performs multiplication of 2^n-word inputs in $O(3^n)$ one-word multiplications and $O(3^n)$ additions, for a complexity of $O(n^{\log_2(3)}) \subset O(n^{1.585})$.

The underlying principle of Karasuba's short-cut is that of *evaluation* and *interpolation* of polynomials to obtain the coefficients of the product. Multiplication of multi-digit integers is intimately related to multiplication of polynomials with integer coefficients. Given the integers a and b in base w notation, we can write $A(x) = \sum_{i=0}^{m} a_i x^i$ and $B(x) = \sum_{j=0}^{n} b_j x^j$ so that $a = A(w)$ and $b = B(w)$. Now we can compute the product polynomial $C(x) = A(x)B(x)$ and find $c = ab = C(w)$. The step of breaking down the input integers into digits in a certain word base amounts to obtaining the coefficients of a polynomial, the convolution product computes the product polynomial $C(x)$, and the carry propagation step evaluates $C(w)$.

In Karatsuba's method, the product polynomial $C(x)$ is computed by evaluating $C(x) = A(x)B(x)$ at sufficiently many points x so that the coefficients of $C(x)$ can be determined uniquely. In the description given above (Karatsuba and Ofman's original description reduces multiplication to two squarings first), we compute $A(0) = a_0$, $B(0) = b_0$, $A(1) = a_1 + a_0$, $B(1) = b_1 + b_0$ and (formally[1]) $A(\infty) = a_1$, $B(\infty) = b_1$ in what constitutes the *evaluation* phase of the algorithm.

The values of the product polynomial $C(x)$ are the products of the values of $A(x)$ and $B(x)$: $C(0) = A(0)B(0)$, $C(1) = A(1)B(1)$, and $C(\infty) = A(\infty)B(\infty)$. In this step, the results of the evaluation phase are multiplied pair-wise. In Karatsuba's method, three products are computed, each multiplying numbers half the size of the input integers.

A polynomial of degree d is uniquely determined given $d+1$ values at distinct points, so these three values suffice to obtain the coefficients of $C(x) = \hat{c}_2x^2 + \hat{c}_1x + \hat{c}_0$ in the *interpolation* phase of the algorithm. In the case of Karatsuba's method this is particularly simple, since $\hat{c}_0 = C(0)$, $\hat{c}_2 = C(\infty)$ and $\hat{c}_1 = C(1) - \hat{c}_2 - \hat{c}_0$.

1.1.2 The Toom-Cook Algorithm

In 1963, A. L. Toom [97] (English translation in [98]) suggested a method which implements a generalization of Karatsuba's method that allows splitting the input numbers into more than two pieces each, leading to polynomials of larger degree but smaller coefficients that must be multiplied. Cook's thesis [28] translates the method to an algorithm, it is now commonly called Toom-Cook method. Given two $(r+1)$-word integers a and b, we can compute their product by writing $a = \sum_{i=0}^{r} a_iw^i$, $b = \sum_{i=0}^{r} b_iw^i$, $0 \le a_i, b_i < w$, and multiplying the polynomials $A(x) = \sum_{i=0}^{r} a_ix^i$ and $B(x) = \sum_{i=0}^{r} b_ix^i$ to obtain the product polynomial $C(x) = \sum_{i=0}^{2r} \hat{c}_ix^i$ of degree $2r$ by evaluating $A(x)$ and $B(x)$ at $2r+1$ distinct points, pair-wise multiplication of the values (each about $1/(r+1)$ the size of the input numbers) and interpolating $C(x)$. For example, for $r = 2$, the points of evaluation $x = 0, \infty, \pm 1$, and 2 could be chosen, so that $A(0) = a_0$, $A(\infty) = a_2$, $A(1) = a_2 + a_1 + a_0$, $A(-1) = a_2 - a_1 + a_0$, and $A(2) = 4a_2 + 2a_1 + a_0$ (likewise for $B(x)$). After the pair-wise products to obtain $C(0)$, $C(\infty)$, $C(1)$, $C(-1)$, and $C(2)$, the interpolation determines the coefficients of $C(x) = \sum_{i=0}^{4} c_ix^i$ by, e.g.,

$$\begin{aligned}
\hat{c}_0 &= C(0)\\
\hat{c}_4 &= C(\infty)\\
2\hat{c}_2 &= C(1) + C(-1) - 2\hat{c}_4 - 2\hat{c}_0\\
6\hat{c}_3 &= C(2) - 2C(1) - 14\hat{c}_4 - 2\hat{c}_2 + \hat{c}_0\\
\hat{c}_1 &= C(1) - \hat{c}_4 - \hat{c}_3 - \hat{c}_2 - \hat{c}_0.
\end{aligned}$$

Toom-Cook with $r = 2$ computes a product of two $3n$-word integers with five products of two n-word integers each; applied recursively, the asymptotic cost of this method is $O(n^{\log_3(5)}) \subset O(n^{1.47})$ and in general, for a fixed r, $O(n^{\log_{r+1}(2r+1)})$.

Even for $r = 2$ the evaluation phase and especially the interpolation phase are noticeably more involved than for Karatsuba's method. This complexity quickly grows with r; if carried out in a straight-forward manner, the evaluation and interpolation performs $O(r^2)$ multiplications of $O(n/r)$-bit integers with $O(\log(r))$ bit integers which yields a complexity of $O(rn\log(r))$, and selection of optimal sets of points of evaluation and of interpolation sequences is non-trivial [13]. Hence for a given n we cannot increase r arbitrarily: the increase of cost of evaluation and interpolation quickly exceeds the saving due to smaller pair-wise products. Toom shows that by

[1] Evaluating a polynomial $f(x)$ at $x = \infty$ should be interpreted as evaluating the homogenized polynomial $F(x,y) = y^{\deg(f)}f(x/y)$ at $(x,y) = (1,0)$; the point $(1,0)$ of the projective line corresponds to the point at infinity of the affine line.

choice of $r = c^{\sqrt{\log(n/r)}}$ with a suitable constant c, an algorithm with complexity in $O(n^{1+\epsilon})$ for any positive ϵ can be obtained; however to reach small ϵ, unreasonably large n are required.

An advantage of Karatsuba's method over Toom-Cook with $r > 1$ is that no division is required in the interpolation stage which makes it applicable over finite fields of small characteristic. Montgomery [71], extending work by Weimerskirch and Paar [100], gives division-free Karatsuba-like formulas that split the input into more than two parts and obtain the product coefficients (in a manner that does not adhere to the evaluation/interpolation principle) with a number of multiplications closer to $n^{\log_2(3)}$ than plain Karatsuba does when n is not a power of 2.

1.1.3 FFT-based Multiplication

The problem of costly evaluation and interpolation can be overcome by use of the Fast Fourier Transform (FFT). An FFT of length ℓ computes from $a_0, \ldots, a_{\ell-1} \in R$ with R a suitable ring

$$\mathfrak{a}_j = \sum_{i=0}^{\ell-1} a_i \omega^{ij}, \qquad j = 0, \ldots, \ell - 1 \tag{1.3}$$

where $\omega \in R$ is an ℓ-th primitive root of unity (which must exist for R to be suitable).

When the a_i are interpreted as coefficients of the polynomial $A(x) = \sum_{i=0}^{\ell-1} a_i x^i$, the FFT can be viewed as a polynomial multi-point evaluation scheme

$$\mathfrak{a}_j = A(\omega^j)$$

which evaluates $A(x)$ at the ℓ distinct points ω^j. Likewise, the inverse FFT computes the polynomial coefficients a_i from the FFT coefficients $\mathfrak{a}_j$ by

$$a_i = \frac{1}{\ell} \sum_{j=0}^{\ell-1} \mathfrak{a}_j \omega^{-ij}, \qquad i = 0, \ldots, \ell - 1. \tag{1.4}$$

The division by ℓ requires that ℓ is a unit in R. To see that (1.4) is the inverse operation of (1.3), we can substitute $\mathfrak{a}_j$ in (1.4) as defined in (1.3) and note that $\sum_{j=0}^{\ell-1} (\omega^k)^j$ is zero for $\ell \nmid k$, and is ℓ for $\ell \mid k$.

In practice the FFT is fastest to compute if ℓ is a power of 2. Assume $\ell = 2^k$ and rewrite (1.3) as

$$\begin{aligned} \mathfrak{a}_j = \sum_{i=0}^{\ell-1} a_i \omega^{ij} &= \sum_{i=0}^{\ell/2-1} a_{2i} \omega^{2ij} + \sum_{i=0}^{\ell/2-1} a_{2i+1} \omega^{(2i+1)j} \\ &= \sum_{i=0}^{\ell/2-1} a_{2i} \omega^{2ij} + \omega^j \sum_{i=0}^{\ell/2-1} a_{2i+1} \omega^{2ij} \end{aligned}$$

for $j = 0, \ldots, \ell - 1$. Since $\omega^\ell = 1$, we have $\omega^{2j} = \omega^{2(j-\ell/2)}$ so each of the two sums takes only $\ell/2$ distinct values. These values $\mathfrak{a}_j^{\text{even}} = \sum_{i=0}^{\ell/2-1} a_{2i} (\omega^2)^{ij}$ and $\mathfrak{a}_j^{\text{odd}} = \sum_{i=0}^{\ell/2-1} a_{2i+1} (\omega^2)^{ij}$ for $j = 0, \ldots, \ell/2-1$ are the FFT of length $\ell/2$ of the coefficients of even and odd index, respectively. Hence an FFT of length ℓ can be computed by two FFTs of length $\ell/2$ and $O(\ell)$ additional ring operations. An FFT of length $\ell = 1$ is just the identity function. Thus the arithmetic complexity $F(\ell)$ of the FFT satisfies $F(1) = 1$, $F(2\ell) = 2F(\ell) + O(\ell)$ and so $F(\ell) \in O(\ell \log(\ell))$ ring operations.

Also, since $\omega^{j+\ell/2} = -\omega^j$, we can compute

$$\begin{aligned} \mathfrak{a}_j &= \mathfrak{a}_j^{\text{even}} + \omega^j \mathfrak{a}_j^{\text{odd}} \\ \mathfrak{a}_{j+\ell} &= \mathfrak{a}_j^{\text{even}} - \omega^j \mathfrak{a}_j^{\text{odd}} \end{aligned} \tag{1.5}$$

where the product $\omega^j \mathfrak{a}_j^{\text{odd}}$ needs to be computed only once, for $j = 0, \ldots, \ell/2$. This operation can be performed in-place, with $\mathfrak{a}_j$ and $\mathfrak{a}_{j+\ell}$ overwriting $\mathfrak{a}_j^{\text{even}}$ and $\mathfrak{a}_j^{\text{odd}}$, respectively. Hence an FFT algorithm can be formulated as in Algorithm 1. This recursive algorithm was first published by Cooley and Tukey [29], although the relevant identities were known to previous authors [30], including Gauss.

This algorithm operates in-place, and at each recursion level the input coefficients a_i of even index i are expected in the lower half of the data array and those of odd index in the upper half, for the sub-FFT being computed. Over the entire recursion, this requires coefficients a_i where the least significant bit (LSB) of i is zero to be located in the low half of the input array and those where the LSB of i is one in the upper half. Within these two halves, coefficients with the second LSB of i equal zero must be in the lower half again, etc. This leads to a storage location for a_i that is the *bit-reverse* of the index i. Let $\text{bitrev}_k(i), 0 \leq i < 2^k$ denote the bit-reverse of the k-bit integer i (extended to k bits with leading zero bits if necessary): if $i = \sum_{n=0}^{k-1} i_n 2^n$, $i_n \in \{0, 1\}$, then $\text{bitrev}_k(i) = \sum_{n=0}^{k-1} i_{k-1-n} 2^n$. Hence for a length $\ell = 2^k$ in-place FFT, the input coefficient a_i must be placed in location $\text{bitrev}_k(i)$ in the data array. The output coefficient $\mathfrak{a}_j$ is stored in location j. The reordering process due to in-place FFT algorithms is called "scrambling".

```
Procedure FFT_DIT ($\ell$, $a$, $\omega$)
Input: Transform length $\ell = 2^k, k \in \mathbb{N}$
       Input coefficients $a_{0,\ldots,\ell-1} \in R$, stored in bit-reversed order
       Root of unity $\omega \in R, \omega^\ell = 1$, $\omega^i \neq 1$ for $0 < i < \ell$
Output: The FFT coefficients $\mathfrak{a}_j = \sum_{i=0}^{\ell-1} a_i \omega^{ij}$ stored in normal order, replacing the
        input
if $\ell > 1$ then
    FFT_DIT($\ell/2$, $a_{0,\ldots,\ell/2-1}$, $\omega^2$) ;
    FFT_DIT($\ell/2$, $a_{\ell/2,\ldots,\ell-1}$, $\omega^2$) ;
    for $i := 0$ to $\ell/2 - 1$ do
        $(a_i, a_{i+\ell/2}) := (a_i + \omega^i a_{i+\ell/2}, a_i - \omega^i a_{i+\ell/2})$ ;
```

Algorithm 1: Recursive Cooley-Tukey algorithm for Fast Fourier Transform

The Cooley-Tukey in-place FFT can be used by passing the input coefficients in bit-reverse order, and fast algorithms exist to perform this permutation [86]. However, for computing convolution products with the FFT, a more elegant solution is available. Gentleman and Sande [47] presented an alternative algorithm for computing the FFT that can be derived by splitting the computation of $\mathfrak{a}_j$ in (1.3) by even and odd j rather than even and odd i:

$$\begin{aligned} \mathfrak{a}_{2j} &= \sum_{i=0}^{\ell-1} a_i \omega^{2ij} = \sum_{i=0}^{\ell/2-1} (a_i + a_{i+\ell/2}) \omega^{2ij} \\ \mathfrak{a}_{2j+1} &= \sum_{i=0}^{\ell-1} a_i \omega^{i(2j+1)} = \sum_{i=0}^{\ell/2-1} \omega^i (a_i - a_{i+\ell/2}) \omega^{2ij} \end{aligned}$$

Hence we can compute the length-ℓ FFT from the two length-$\ell/2$ FFTs of the coefficients

$$\begin{aligned} a_i^{\text{even}} &= a_i + a_{i+\ell/2} \\ a_i^{\text{odd}} &= \omega^i (a_i - a_{i+\ell/2}), \end{aligned} \tag{1.6}$$

for $0 \leq i < \ell/2$. This leads to Algorithm 2 which takes unscrambled input (coefficient a_i in array location i) and produces scrambled output (coefficient $\mathfrak{a}_j$ in array location $\text{bitrev}_k(j)$). Additionally, the Cooley-Tukey algorithm can be used for an inverse FFT simply by replacing ω by ω^{-1} everywhere and dividing each output coefficient by ℓ. For the pair-wise multiplication, whether the FFT coefficients are scrambled or not does not matter, so long as both sequences of FFT coefficients are in the same order. Hence we can use the Gentleman-Sande algorithm for computing forward FFTs producing scrambled output, do point-wise multiplication of the FFT coefficients, and use the Cooley-Tukey algorithm for the inverse FFT taking scrambled input and producing output in normal order. This way, explicit re-ordering with a bit-reversal algorithm is avoided completely.

```
Procedure FFT_DIF ($\ell$, $a$, $\omega$)
Input: Transform length $\ell = 2^k, k \in \mathbb{N}$
       Input coefficients $a_{0,\ldots,\ell-1} \in R$ stored in normal order
       Root of unity $\omega \in R, \omega^\ell = 1$, $\omega^i \neq 1$ for $0 < i < \ell$
Output: The FFT coefficients $\mathfrak{a}_j = \sum_{i=0}^{\ell-1} a_i \omega^{ij}$, stored in bit-reversed order, replacing
        the input
if $\ell > 1$ then
    for $i := 0$ to $\ell/2 - 1$ do
        $(a_i, a_{i+\ell/2}) := (a_i + a_{i+\ell/2}, \omega^i(a_i - a_{i+\ell/2}))$ ;
    FFT_DIF($\ell/2$, $a_{0,\ldots,\ell/2-1}$, $\omega^2$) ;
    FFT_DIF ($\ell/2$, $a_{\ell/2,\ldots,\ell-1}$, $\omega^2$) ;
```

Algorithm 2: Recursive Gentleman-Sande algorithm for Fast Fourier Transform

When multiplying two polynomials $A(x)$ of degree m and $B(x)$ of degree n, the product polynomial $C(x)$ has degree $m+n$, and an FFT of length $m+n+1 \leq \ell$ is required to determine the coefficients of $C(x)$ uniquely. The same transform length must be used for the forward transforms (evaluating $A(x)$ and $B(x)$) and for the inverse transform (interpolating $C(x)$); in the forward transform, the coefficients of $A(x)$ and $B(x)$ are padded with zeros to fill the input of the FFT up to the transform length.

If the degrees of the input polynomials are too large so that $m + n + 1 > \ell$, the product polynomial $C(x)$ cannot be interpolated correctly. Let $C(x) = x^\ell C_1(x) + C_0(x)$, $\deg(C_0(x)) < \ell$. Since $\omega^\ell = 1$, $C(\omega^j) = C_1(\omega^j) + C_0(\omega^j)$ for all $j \in \mathbb{N}$, so polynomials $C(x)$ and $C(x) \bmod (x^\ell - 1)$ have the same FFT coefficients, and the interpolation of polynomials with an inverse FFT of length ℓ from a given set of FFT coefficients is unique only modulo $x^\ell - 1$. In other words, the coefficients of too high powers x^i, $i \geq \ell$, in the product polynomial are *wrapped around* and added to the coefficients of $x^{i \bmod \ell}$, also called cyclic wrap-around or a *cyclic convolution*. Hence for inputs $A(x)$, $B(x)$, a convolution product with a length-ℓ FFT produces the ℓ coefficients of $A(x)B(x) \bmod (x^\ell - 1)$; if the product polynomial has degree less than ℓ, its coefficients are not disturbed by this modulus. Algorithm 3 shows a cyclic convolution product using the Fast Fourier Transform.

The implicit polynomial modulus in an FFT convolution can sometimes be used profitably, but often a modulus other than $x^\ell - 1$ is desired. This is accomplished with a *weighted* transform.

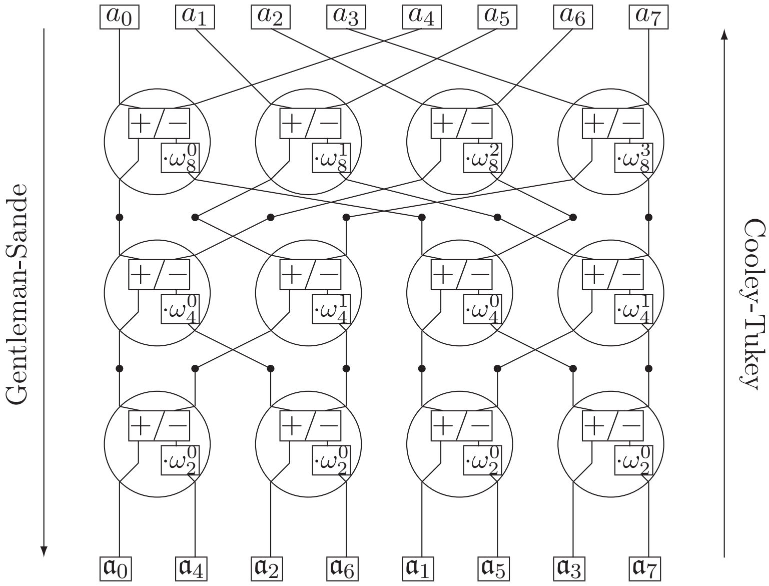

Figure 1.1: Data flow in a length-8 FFT. Here, ω_n denotes an n-th primitive root of unity. Read from top to bottom, the diagram shows the data flow of the Gentleman-Sande algorithm with a_i as input coefficients and $\mathfrak{a}_i$ as output. Read from bottom to top, it shows the data flow of the Cooley-Tukey algorithm, with $\mathfrak{a}_i$ as input and a_i as output.

The FFT input coefficients a_i and b_i are multiplied by weights w^i, for $0 \le i < \ell$; after the pairwise multiplication and inverse FFT, the output coefficients c_i are multiplied by w^{-i}. In effect, the FFT computes $C(x)$ so that $C(wx) = A(wx)B(wx) \bmod (x^\ell - 1)$. With $A(wx)B(wx) = x^\ell w^\ell C_1(wx) + C_0(wx)$, we have $C(wx) = w^\ell C_1(wx) + C_0(wx)$ and thus $C(x) = w^\ell C_1(x) + C_0(x)$, which causes the wrapped-around part to be multiplied by w^ℓ and corresponds to a multiplication $C(x) = A(x)B(x) \bmod (x^\ell - w^\ell)$. To allow a particular polynomial modulus $x^\ell - r$, the ring R over which the FFT is performed must contain a w so that $w^\ell = r$, i.e., a (not necessarily primitive) ℓ-th root of r. For example if a modulus of $x^\ell + 1$ is desired, we require that $w = \sqrt[\ell]{-1}$, which is a (2ℓ)-th root of unity (and necessarily primitive if ℓ is a power of 2), exists in R. In this case the wrapped-around coefficients of x^i, $\ell \le i < 2\ell$, are subtracted from the coefficients of $x^{i \bmod \ell}$, which leads to a *negacyclic convolution*.

Unfortunately we cannot apply the FFT directly to the problem of multiplying integers (via polynomials with integer coefficients) as the ring of integers $\mathbb{Z}$ does not offer any primitive roots of unity of order greater than 2. In order to carry out the FFT, we must map the coefficients of the polynomials $A(x)$ and $B(x)$ to some other ring first which has a primitive root of unity of order ℓ and the appropriate weight w if a weighted transform is desired, where ℓ is a unit, and where the coefficients c_i of the product polynomial can be identified uniquely. This mapping is what distinguishes different FFT based integer multiplication algorithms.

```
Input: Convolution length $\ell = 2^k$
       Input coefficients $a_{0,\ldots,\ell-1}, b_{0,\ldots,\ell-1} \in R$, where $R$ supports a length-$\ell$ FFT
Output: Coefficients $c_{0,\ldots,\ell-1} \in R$ of cyclic convolution product of $a$ and $b$
Data: $t_{0,\ldots,\ell-1}$, temporary storage
/* Compute primitive $\ell$-th root of unity in $R$                */
$\omega := \sqrt[\ell]{\mathbf{1}}, \mathbf{1} \in R$;
/* Copy $a$ to $c$, compute forward FFT in-place                    */
$c_{0,\ldots,\ell-1} := a_{0,\ldots,\ell-1}$;
FFT_DIF($\ell$, $c_{0,\ldots,\ell-1}$, $\omega$) ;
/* Copy $b$ to $t$, compute forward FFT in-place                    */
$t_{0,\ldots,\ell-1} := b_{0,\ldots,\ell-1}$;
FFT_DIF($\ell$, $t_{0,\ldots,\ell-1}$, $\omega$) ;
/* Compute pair-wise products                                       */
for $i := 0$ to $\ell - 1$ do
    $c_i := c_i \cdot t_i$;
/* Compute inverse FFT in-place                                     */
FFT_DIT($\ell$, $c_{0,\ldots,\ell-1}$, $\omega^{-1}$);
for $i := 0$ to $\ell - 1$ do
    $c_i := c_i/\ell$;
```

Algorithm 3: Cyclic convolution product with the Fast Fourier Transform

1.2 An Efficient Implementation of Schönhage-Strassen's Algorithm

Schönhage and Strassen [90] were the first to present practical algorithms for integer multiplication based on the FFT. They gave two possibilities for performing the convolution product via the FFT: one over the complex numbers $\mathbb{C}$ which has complexity $O\big(N \log(N) \log(\log(N))^{1+\epsilon}\big)$ for any positive ϵ and with input numbers of N bits, and one over a residue class ring $R = \mathbb{Z}/\left(2^n + 1\right)\mathbb{Z}$ with complexity $O(N \log(N) \log(\log(N)))$. Even though both methods were published in the same paper and are both used in practice, "Schönhage-Strassen's algorithm" usually refers only to the latter.

Since Fourier transforms over $\mathbb{C}$ are ubiquitous in signal processing, data compression, and many other fields of scientific computation, a wealth of publications on and countless implementations of the complex FFT exist. General-purpose complex FFT implementations (e.g., [42]) can be readily used for large-integer multiplication, although specialized implementations for fast convolution products with purely real input offer further opportunities for optimization. The field of complex FFTs and their efficient implementation is vast, and we do not explore it here any further, but focus on convolution products using FFTs over the ring $\mathbb{Z}/(2^n + 1)\mathbb{Z}$.

1.2.1 Overview

Schönhage-Strassen's algorithm (SSA) reduces the multiplication of two input integers a and b to ℓ multiplications in $R_n = \mathbb{Z}/\left(2^n + 1\right)\mathbb{Z}$, for suitably chosen ℓ and n. In order to achieve the stated complexity, these multiplications must be performed efficiently, and SSA calls itself recursively for this purpose, until the numbers to be multiplied are small enough for simpler algorithms such as the grammar-school, Karatsuba, or Toom-Cook methods. In order to facilitate this recursive use of SSA, it is formulated to accept any two integers $0 \leq a, b < 2^N + 1$ for a given N as input and compute the product $ab \bmod (2^N + 1)$. If the inputs are such that $ab < 2^N + 1$, then this SSA of course lets us compute the correct integer product ab, so we can use this algorithm (by choosing N suitably) to perform integer multiplication. If the product of the inputs is known not to exceed $2^N + 1$, optimizations are possible that allow performing the top-most recursion level of SSA more efficiently, see 1.2.6. The multiplication modulo $2^N + 1$ is done with a negacyclic convolution of length ℓ using an FFT over the ring R_n, so that arithmetic modulo $2^N + 1$ can be performed via arithmetic modulo $x^\ell + 1$, and the pair-wise products in R_n can use SSA again.

Thus SSA maps the computation of ab in $\mathbb{N}$ to the computation of $ab \bmod (2^N + 1)$ in R_N, maps this product in R_N to a product of polynomials $A(x)B(x)$ in $\mathbb{Z}[x]/(x^\ell + 1)$, and maps this convolution product to $R_n[x]/(x^\ell + 1)$ so that R_n supports a length-ℓ weighted FFT for a negacyclic convolution and allows lifting the coefficients of $A(x)B(x) \bmod (x^\ell + 1)$ uniquely. Figure 1.2 shows the sequence of mappings.

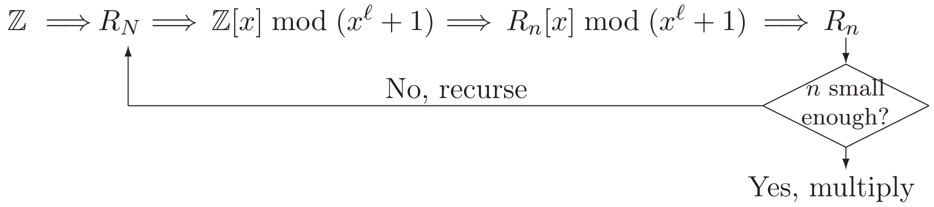
$\mathbb{Z} \Longrightarrow R_N \Longrightarrow \mathbb{Z}[x] \bmod (x^\ell + 1) \Longrightarrow R_n[x] \bmod (x^\ell + 1) \Longrightarrow R_n$
No, recurse
n small enough?
Yes, multiply

Figure 1.2: Diagram of mappings in the Schönhage-Strassen algorithm.

1.2.2 Description of SSA

For given a, b whose product modulo $2^N + 1$, $4 \mid N$, is sought, write $N = \ell M$ where $\ell = 2^k$, $k \geq 2$, and choose an n so that

$$\begin{aligned} n &= \ell m, \\ n &\geq 2M + k. \end{aligned} \tag{1.7}$$

The two conditions imply $n > \sqrt{2N}$. The choice of good values for N, ℓ, and n is of great importance for the performance of SSA; how to select these is described in 1.2.7.

Let $R_n = \mathbb{Z}/(2^n + 1)\mathbb{Z}$. Since $2^n = (2^m)^\ell \equiv -1 \pmod{2^n + 1}$, 2^m is an (2ℓ)-th primitive root of unity in R_n, so that it supports FFTs for cyclic convolutions of power-of-two length up to 2ℓ, or weighted FFTs for negacyclic convolutions up to length ℓ.

From the input integers $0 \leq a, b < 2^N + 1$ we form polynomials

$$A(x) = \sum_{i=0}^{\ell-1} a_i x^i, \qquad \begin{array}{l} 0 \leq a_i < 2^M \text{ for } 0 \leq i < \ell - 1 \\ \\ 0 \leq a_{\ell-1} \leq 2^M, \end{array} \tag{1.8}$$

that is, we cut a into ℓ pieces of M bits each, except the last piece may be equal to 2^M. Doing likewise for b, we have $a = A(2^M)$ and $b = B(2^M)$. To obtain the product $c = ab \bmod (2^N + 1)$, we can use a negacyclic convolution to compute the product polynomial $C(x) = A(x)B(x) \bmod (x^\ell + 1)$ so that, when the polynomials are evaluated at $x = 2^M$, the polynomial modulus $x^\ell + 1$ preserves the residue class modulo $2^N + 1 = \left(2^M\right)^\ell + 1$ of the integer product.

The coefficients of the product polynomial $C(x) = \sum_{i=0}^{\ell-1} c_i x^i$ can be lifted exactly from R_n if $c_i \bmod (2^n+1)$ is unique for all possible values of each c_i. Due to the negacyclic wrap-around, each c_i can be defined by

$$c_i = c_i^{\text{low}} - c_i^{\text{high}}$$

with

$$\begin{aligned} c_i^{\text{low}} &= \sum_{0 \le j \le i} a_j b_{i-j} \\ c_i^{\text{high}} &= \sum_{i < j < \ell} a_j b_{i-j+\ell}, \end{aligned}$$

where all c_i^{low} and c_i^{high} are non-negative and, applying the bounds from (1.8), satisfy

$$\begin{array}{rcll} c_i^{\text{low}} & < & (i+1)2^{2M}, & 0 \le i < \ell \\ c_i^{\text{high}} & < & (\ell-1-i)2^{2M}, & 0 \le i < \ell - 2 \\ c_{\ell-2}^{\text{high}} & \le & 2^{2M}. & \end{array}$$

Thus, for all $0 \le i < \ell$,

$$((i+1)-\ell)2^{2M} \le c_i < (i+1)2^{2M},$$

so each c_i is confined to an interval of length $2^{2M}\ell$ and it suffices to choose $2^n + 1 > 2^{2M}\ell$, or $n \ge 2M + k$ with $\ell = 2^k$. This minimal choice of $n = 2M + k$ requires that the lifting algorithm adjusts the range of each c_i depending on i; if an algorithm is desired that works independently of i, we must choose $n \ge 2M + k + 1$.

Hence the conditions on n given in (1.7) are sufficient to allow the computation of $C(x) = A(x)B(x) \bmod x^\ell + 1$ with a negacyclic convolution by a weighted FFT over the ring R_n.

Given the coefficients of the product polynomial $C(x)$, the integer $c = C(2^M) \bmod (2^N+1) = ab \bmod (2^N+1)$ can be computed in the final carry propagation step. The c_i may be negative or greater than $2^M - 1$, so $0 \le C(2^M) < 2^N + 1$ does not necessarily hold and the carry propagation needs to take the modulus $2^N + 1$ into account.

The SSA thus consists of five consecutive steps, shown below. In this example, the coefficients c_i of the product polynomial overwrite the coefficients a_i.

1. **Decompose a and b and apply weights**
   Allocate memory for array $a[i]$, $0 \le i < \ell$, with at least $n+1$ bits of storage per array entry
   Store in $a[i]$ the i-th M-bit part of a
   Apply weights by setting $a[i] := w^i \cdot a[i] \bmod (2^n+1)$, $w = 2^{n/\ell}$
   Do likewise for array $b[]$

2. **Forward transform of a and b**
   Perform length-ℓ FFT in-place on array $a[]$ working modulo 2^n+1, with root of unity $\omega = 2^{2n/\ell}$
   Do likewise for array $b[]$

3. **Pair-wise multiply**
   Set $a[i] := a[i] \cdot b[i] \bmod (2^n+1)$ for $0 \le i < \ell$

4. **Inverse transform**
   Perform length-ℓ inverse FFT in-place on array $a[]$ working modulo 2^n+1, including division by ℓ

5. **Apply inverse weights, carry propagation**
   Un-apply weights by setting $a[i] := w^{-i} \cdot a[i] \bmod (2^n + 1)$, $w = 2^{n/\ell}$
   Compute sum $c = \left(\sum_{i=0}^{\ell-1} a_i 2^{iM}\right) \bmod \left(2^N + 1\right)$

Most of the ideas for the SSA presented so far were already present in the papers by Schönhage and Strassen [90], or in follow-up papers by Schönhage [87]. We describe now several practical improvements that allow SSA to multiply very large integers rapidly on contemporary computers. The implementation is based on the implementation of SSA in the GNU Multiple Precision arithmetic library (GMP) [49], version 4.2.1.

1.2.3 Arithmetic Modulo $2^n + 1$ with GMP

Arithmetic operations modulo $2^n + 1$ have to be performed during the forward and inverse transforms, the point-wise products, when applying the weight signal, and when unapplying it. Thanks to the fact that the primitive roots of unity are powers of two, the only needed operations outside the point-wise products are additions, subtractions, and multiplications by a power of two which can be performed by bit-wise shifts on a binary computer. Since $2^{2n} \equiv 1 \pmod{2^n+1}$, division by 2^k can be reduced to multiplication by 2^{2n-k}. Reduction modulo 2^n+1 is inexpensive as well, since $a_1 2^n + a_0 \equiv a_0 - a_1 \pmod{2^n + 1}$, so the reduction again requires only a bit-wise shift and a subtraction.

To simplify arithmetic modulo $2^n + 1$, we require n to be a multiple of β, the number of bits per machine word. Since n must also be a multiple of $\ell = 2^k$, this usually is not an additional constraint unless $k < 5$ on a 32-bit computer or $k < 6$ on a 64-bit computer, and SSA is typically used for numbers that are large enough so that the transform length is at least 64. Let $m = n/\beta$ be the number of computer words corresponding to an n-bit number. A residue mod $2^n + 1$ has a semi-normalized representation with m full words and one carry of weight 2^n:

$$a = (a_m, a_{m-1}, \ldots, a_0),$$

with $0 \le a_i < 2^\beta$ for $0 \le i < m$, and $0 \le a_m \le 1$.

The addition of two such representations is done as follows (we give here the C code using GMP functions):

```
c = a[m] + b[m] + mpn_add_n (r, a, b, m);
r[m] = (r[0] < c);
MPN_DECR_U (r, m + 1, c - r[m]);
```

The first line of this algorithm adds $(a_{m-1}, \ldots, a_0)$ and $(b_{m-1}, \ldots, b_0)$, puts the low m words of the result in $(r_{m-1}, \ldots, r_0)$, and adds the out carry to $a_m + b_m$; we thus have $0 \le c \le 3$. The second line yields $r_m = 0$ if $r_0 \ge c$, in which case we simply subtract c from r_0 at the third line. Otherwise $r_m = 1$, and we subtract $c - 1$ from r_0: a borrow may propagate, but at most to r_m. In all cases $r = a + b \bmod (2^n + 1)$, and r is semi-normalized.

The subtraction is done in a similar manner:

```
c = a[m] - b[m] - mpn_sub_n (r, a, b, m);
r[m] = (c == 1);
MPN_INCR_U (r, m + 1, r[m] - c);
```

After the first line, we have $-2 \le c \le 1$. If $c = 1$, then $r_m = 1$ at the second line, and the third line does nothing. Otherwise, $r_m = 0$ at the second line, and we add $-c$ to r_0, where the carry may propagate up to r_m. In all cases $r = a - b \bmod (2^n + 1)$, and r is semi-normalized.

The multiplication by 2^e is more tricky to implement. However this operation mainly appears in the butterflies — see below — $[a, t] \leftarrow [a+b, (a-b)2^e]$ of the forward and inverse transforms, which may be performed as follows:

1. Write $e = d \cdot \beta + s$ with $0 \leq s < \beta$, where β is the number of bits per word
2. Decompose $a = (a_1, a_0)$, where a_1 contains the upper d words
3. Idem for b
4. $t := (a_0 - b_0, b_1 - a_1)$
5. $a := a + b$
6. $t := t \cdot 2^s$

Step 4 means that the most $(m-d)$ significant words from t are formed with $a_0 - b_0$, and the least d significant words with $b_1 - a_1$, where we assume that borrows are propagated, so that t is semi-normalized. Thus the only real multiplication by a power of two is that of step 6, which may be efficiently performed with GMP's `mpn_lshift` routine.

If one has a combined `mpn_addsub` routine which simultaneously computes $x + y$ and $x - y$ faster than separate `mpn_add` and `mpn_sub` calls, then step 5 can be written $a := (b_1 + a_1, a_0 + b_0)$ which shows that t and a may be computed with two `mpn_addsub` calls.

1.2.4 Improved FFT Length Using $\sqrt{2}$

Since all prime factors of $2^n + 1$ are $p \equiv 1 \pmod 8$ if $4 \mid n$, 2 is a quadratic residue in R_n, and it turns out that $\sqrt{2}$ is of a simple enough form to make it useful as a root of unity with power-of-two order. Specifically, $\left(\pm 2^{3n/4} \mp 2^{n/4}\right)^2 \equiv 2 \pmod{2^n + 1}$, which is easily checked by expanding the square. Hence for a given $n = 2^k m$, $k \geq 2$, we can use, e.g., $\sqrt{2} = 2^{3n/4} - 2^{n/4}$ as a root of unity of order 2^{k+2} to double the possible transform length. In the case of the negacyclic convolution, this allows a length 2^{k+1} transform, and $\sqrt{2}$ is used only in the weight signal. For a cyclic convolution, $\sqrt{2}$ is used normally as a root of unity during the transform, allowing a transform length of 2^{k+2}. This idea is mentioned in [8, §9] where it is credited to Schönhage, who later pointed out [88] that he was aware of this trick from the start, but published it only "encoded" in [89, exercise 18].

Multiplication by an odd power of $\sqrt{2}$ involves two binary shifts and a subtraction which requires more arithmetic than multiplication by a power of 2, but is still inexpensive enough that the smaller pair-wise products in the convolution due to larger transform length lead to a net gain. In our implementation, this $\sqrt{2}$ trick saved roughly 10% on the total time of integer multiplication.

Unfortunately, using higher roots of unity for the transform is not feasible as prime divisors of $2^n + 1$ are not necessarily congruent to $1 \pmod{2^{k+3}}$. Deciding whether they are or not requires factoring $2^n + 1$, and even if they are as in the case of the eighth Fermat number $F_8 = 2^{256} + 1$ [17], there does not seem to be a simple form for $\sqrt[4]{2}$ which would make it useful as a root of unity in the transform.

1.2.5 Improved Cache Locality

When multiplying large integers with SSA, the time spent in accessing data for performing the Fourier transforms is non-negligible; especially since the operations performed on the data are so inexpensive, the relative cost of memory access is quite high. The literature is rich with papers dealing with the organization of the computations in order to improve the data locality and thus cache efficiency during an FFT. However, most of these papers are concerned with contexts which are different from ours: usually the coefficients are small and most often they are complex floating-point numbers represented as a pair of `double`'s. Also there is a variety of target platforms, from embedded hardware implementations to super-scalar computers.

We have tried to apply several of these approaches in our context where the coefficients are integers modulo $2^n + 1$ that each occupy at least a few cache lines and where the target platform is a standard PC workstation.

In this work, we concentrate on multiplying large, but not huge integers. By this we mean that we consider only 3 levels of memory for our data: level 1 cache, level 2 cache, and standard RAM. In the future one might also want to consider the case where we have to use the hard disk as a 4th level of storage.

Here are the orders of magnitude for these memories, to fix ideas: on a typical Opteron, a cache line is 64 bytes; the L1 data cache is 64 KB; the L2 cache is 1 MB; the RAM is 8 GB. The smallest coefficient size (i.e., n-bit residues) we consider is about 50 machine words, that is 400 bytes. For very large integers, a single coefficient barely fits into the L1 cache. For instance, in our implementation, when multiplying two integers of $105, 000, 000$ words each, a transform of length 2^{15} is used with coefficients of size 52 KB.

In an FFT computation, the main operation is the butterfly operation as described in equations (1.5) and (1.6). This is an operation in a ring R_n that, for a Gentleman-Sande FFT, computes $a + b$ and $(a - b)\omega$, where a and b are coefficients in R_n and ω is some root of unity. In SSA this root of unity is a power of 2.

The very first FFT algorithm is the iterative one. In our context this is a really bad idea. The main advantage of it is that the data is accessed in a sequential way. In the case where the coefficients are small enough so that several of them fit in a cache line, this saves many cache misses. But in our case, contiguity is irrelevant due to the size of the coefficients compared to cache lines.

The next very classical FFT algorithm is the recursive one. In this algorithm, at a certain level of recursion, we work on a small set of coefficients, so that they must fit in the cache. This version (or a variant of it) was implemented in GMP up to version 4.2.1. This behaves well for moderate sizes, but when multiplying large numbers, everything fits in the cache only at the tail of the recursion, so that most of the transform is already done when we are at last in the cache. The problem is that before getting to the appropriate recursion level, the accesses are very cache unfriendly.

In order to improve the locality for large transforms, we have tried three strategies found in the literature: the Belgian approach, the radix-2^k transform, and Bailey's 4-step algorithm.

The Belgian transform

Brockmeyer et *al.* [18] propose a way of organizing the transform that reduces cache misses. In order to explain it, let us first define a tree of butterflies as follows (we don't mention the root of unity for simplicity):

```
TreeBfy(A, index, depth, stride)
    Bfy(A[index], A[index+stride])
    if depth > 1
        TreeBfy(A, index-stride/2, depth-1, stride/2)
        TreeBfy(A, index+stride/2, depth-1, stride/2)
```

An example of a tree of depth 3 is given on the right of Figure 1.3. Now, the depth of a butterfly tree is bounded by a value that is not the same for every tree. For instance, on Figure 1.3, the butterfly tree that starts with the butterfly between a_0 and a_4 has depth 1: one

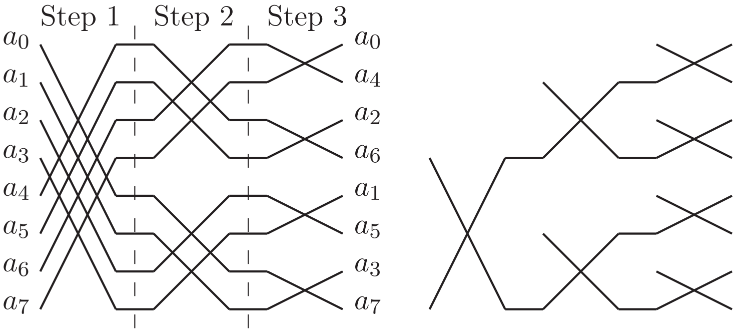

Figure 1.3: The FFT circuit of length 8 and a butterfly tree of depth 3.

can not continue the tree on step 2. Similarly, the tree starting with the butterfly between a_1 and a_5 has depth 1, the tree starting between a_2 and a_6 has depth 2 and the tree starting between a_3 and a_7 has depth 3. More generally, the depth can be computed by a simple formula.

One can check that by considering all the trees of butterflies starting with an operation at step 1, we cover the complete FFT circuit. It remains to find the right ordering for computing those trees of butterflies. For instance, in the example of Figure 1.3, it is important to do the tree that starts between a_3 and a_7 in the end, since it requires data from all the other trees.

One solution is to perform the trees of butterflies following the `bitrev` order. One obtains the following algorithm, where `ord_2` stands for the number of trailing zeros in the binary representation of an integer (together with the 4-line `TreeBfy` routine, this is a recursive description of the 36-line routine from [18, Code 6.1]):

```
BelgianFFT(A, k)
    l = 2^{k-1}
    for i := 0 to l-1
        TreeBfy(A, bitrev(i, k-1), 1+ord_2(i+1), l)
```

Inside a tree of butterflies, we see that most of the time, the butterfly operation will involve a coefficient that has been used just before, so that it should still be in the cache. Therefore an approximate 50% cache-hit is provided by construction, and we can hope for more if the data is not too large compared to the cache size.

We have implemented this in GMP, and this saved a few percent for large sizes, thus confirming the fact that this approach is better than the classical recursive transform.

Higher radix transforms

The principle of higher radix transforms is to use an atomic operation which groups several butterflies. In Arndt's book [3] the reader will find a description of several variants in this spirit. The classical FFT can be viewed as a radix-2 transform. The next step is a radix-4 transform, where the atomic operation has 4 inputs and 4 outputs (without counting roots of unity) and groups 4 butterflies of 2 consecutive steps of the FFT.

We can then build a recursive algorithm upon this atomic operation. Of course, since we perform 2 radix-2 steps at a time, the number of levels in the recursion is reduced by a factor of up to 2 from $\log_2(\ell)$ to $\lceil \log_4(\ell) \rceil$ (we have to handle the last recursion level by a radix-2 transform if the number k of radix-2 FFT levels is odd).

In the literature, the main interest for higher radix transforms comes from the fact that the number of operations is reduced for a transform of complex numbers (this is done by exhibiting a free multiplication by i). In our case, the number of operations remains the same. However, in the atomic block each input is used in two butterflies, so that the number of cache misses is less than 50%, just as for the Belgian approach. Furthermore, with the recursive structure, just as for the classical recursive FFT, at some point we deal with a number of inputs which is small enough so that everything fits in the cache.

We have tested this approach, and this was faster than the Belgian transform by a few percent.

The next step after radix-4 is radix-8 which works in the same spirit, but grouping 3 levels at a time. We have also implemented it, but this saved nothing, and was even sometimes slower than the radix-4 approach. Our explanation is that for small numbers, a radix of 4 is close to optimal with respect to cache locality, and for large numbers, the number of coefficients that fit in the cache is rather small and we have misses inside the atomic block of 12 butterflies.

More generally, radix-2^t groups t levels together, with a total of $t2^{t-1}$ butterflies, over 2^t residues. If all those residues fit in the cache, the cache miss rate is less than $1/t$. Thus the optimal strategy seems to choose for t the largest integer such that $2^t n$ bits fit in the cache (either L1 or L2, whichever is the fastest cache where a single radix-2 butterfly fits).

Bailey's 4-step algorithm

The algorithm we describe here can be found in a paper by Bailey [4]. In it, the reader will find earlier references tracing back the original idea, which was in fact already mentioned in [47]. For simplicity we stick to the "Bailey's algorithm" denomination.

A way of seeing Bailey's 4-step algorithm is as a radix-$\sqrt{\ell}$ transform, where $\ell = 2^k$ is the length of the input sequence. In other words, instead of grouping 2 steps as in radix-4, we group $k/2$ steps. To be more general, let us write $k = k_1 + k_2$, where k_1 and k_2 are to be thought as close to $k/2$, but this is not really necessary. Then Bailey's 4-step algorithm consists in the following phases:

1. Perform 2^{k_2} transforms of length 2^{k_1};

2. Multiply the data by weights;

3. Perform 2^{k_1} transforms of length 2^{k_2}.

There are only three phases in this description. The fourth phase is usually some matrix transposition[2], but this is irrelevant in our case: the coefficients are large so that we keep a table of pointers to them, and this transposition is just pointer exchanges which are basically for free, and fit very well in the cache.

The second step involving weights is due to the fact that in the usual description of Bailey's 4-step algorithm, the transforms of length 2^{k_1} are exactly Fourier transforms, whereas the needed operation is a twisted Fourier transform where the roots of unity involved in the butterflies are different (since they involve a (2^k)-th root of unity, whereas the classical transform of length 2^{k_1} involves a (2^{k_1})-th root of unity). In the classical FFT setting this is very interesting, since we can then reuse some small-dimension implementation that has been very well optimized. In our

[2] Indeed, Bailey's algorithm might be viewed as a two-dimensional transform of a matrix with 2^{k_1} rows and 2^{k_2} columns, where Phase 1 performs 2^{k_2} one-dimensional transforms on the columns, and Phase 3 performs 2^{k_1} one-dimensional transforms on the rows.

case, we have found it better to write separate code for this twisted FFT, so that we merge the first and second phases.

The interest of this way of organizing the computation is again not due to a reduction of the number of operations, since they are exactly the same as with the other FFT approaches mentioned above. The goal is to help locality. Indeed, assume that $\sqrt{\ell}$ coefficients fit in the cache, then the number of cache misses is at most 2ℓ, since each call to the internal FFT or twisted FFT operates on $\sqrt{\ell}$ coefficients.

Of course we are interested in numbers for which $\sqrt{\ell}$ coefficients do not fit in the L1 cache, but for all numbers we might want to multiply, they do fit in the L2 cache. Therefore the structure of the code follows the memory hierarchy: at the top level of Bailey's algorithm, we deal with the RAM vs L2 cache locality question, then in each internal FFT or twisted FFT, we can take care of the L2 vs L1 cache locality question. This is done by using the radix-4 variant inside our Bailey-algorithm implementation.

We have implemented this approach (with a threshold for activating Bailey's algorithm only for large sizes), and combined with radix-4, this gave us our best timings. We have also tried a higher dimensional transform, in particular 3 steps of size $\sqrt[3]{\ell}$. This did not help for the sizes we considered.

Mixing several phases

Another way to improve locality is to mix different phases of the algorithm in order to do as much work as possible on the data while they are in the cache. An easy improvement in this spirit is to mix the pointwise multiplication and the inverse transform, in particular when Bailey's algorithm is used. Indeed, after the two forward transforms have been computed, one can load the data corresponding to the first column, do the pointwise multiplication of its elements, and readily perform the small transform of this column. Then the data corresponding to the second column is loaded, multiplied and transformed, and so on. In this way, one saves one full pass on the data. Taking the idea one step further, assuming that the forward transform for the first input number has been done already (or that we are squaring one number), after performing the column-wise forward transform on the second number we can immediately do the point-wise multiply and the inverse transform on the column, so saving another pass over memory.

Following this idea, we can also merge the "decompose" and "recompose" steps with the transforms, again to save a pass on the data. In the case of the "decompose" step, there is more to it since one can also save unnecessary copies by merging it with the first step of the forward transform.

The "decompose" step consists of cutting parts of M bits from the input numbers, then multiplying each part a_i by w^i modulo $2^n + 1$, giving a'_i. If one closely looks at the first FFT level, it will perform a butterfly between a'_i and $a'_{i+\ell/2}$ with w^{2i} as multiplier. This will compute $a'_i + a'_{i+\ell/2}$ and $a'_i - a'_{i+\ell/2}$, and multiply the latter by w^{2i}. It can be seen that the M non-zero bits from a'_i and $a'_{i+\ell/2}$ do not overlap, thus no real addition or subtraction is required: the results $a'_i + a'_{i+\ell/2}$ and $a'_i - a'_{i+\ell/2}$ can be obtained with just copies and ones' complements. As a consequence, it should be possible to completely avoid the "decompose" step and the first FFT level, by directly starting from the second FFT level, which for instance will add $a'_i + a'_{i+\ell/2}$ to $(a'_j - a'_{j+\ell/2})w^{2j}$; here the four operands $a'_i, a'_{i+\ell/2}, a'_j, a'_{j+\ell/2}$ will be directly taken from the input integer a, and the implicit multiplier w^{2j} will be used to know where to add or subtract a'_j and $a'_{j+\ell/2}$. This example illustrates the kind of savings obtained by avoiding trivial operations like copies and ones' complements, and furthermore improving the locality. This idea was not used in the results in §1.3.

1.2.6 Multiplication without Reduction Modulo $2^N + 1$

The reason why SSA uses a negacyclic convolution is that it allows the algorithm to be used recursively: the "pair-wise products" modulo $2^n + 1$ can in turn be performed using the same algorithm, each one giving rise to ℓ' smaller pair-wise products modulo $2^{n'} + 1$ (where n' must be divisible by ℓ'). A drawback of this approach is that it requires a weighted transform, i.e., additional operations before the forward transforms and after the inverse transform, and an (2ℓ)-th root of unity for the weights which halves the possible transform length for a given n.

The negacyclic transform is needed only to facilitate a modulus of $x^\ell + 1$ in the polynomial multiplication which is compatible with the modulus of $2^N + 1$ of the integer multiplication. But at the top-most recursion level, we choose N so that the integer product $c = ab$ is not affected by any modulo reduction, and no particular modulus for the integer and hence for the polynomial multiplication needs to be enforced.

Therefore one can replace $R_N = \mathbb{Z}/(2^N + 1)\mathbb{Z}$ by $\mathbb{Z}/(2^N - 1)\mathbb{Z}$ in the top-most recursion level of SSA, and replace the negacyclic by a simple cyclic convolution (without any weights in the transform), to compute an integer product mod $2^N - 1$, provided that $c = ab < 2^N - 1$. We call this a "Mersenne transform," whereas the original SSA performs a "Fermat transform"[3]. This idea of using a Mersenne transform is already mentioned by Bernstein [8] where it is called "cyclic Schönhage-Strassen trick".

Despite the fact that it can be used at the top level only, the Mersenne transform is nevertheless very interesting for the following reasons:

- a Mersenne transform modulo $2^N - 1$, combined with a Fermat transform modulo $2^N + 1$ and CRT reconstruction, can be used to compute a product of $2N$ bits;
- as mentioned, a Mersenne transform can use a larger FFT length $\ell = 2^k$ than the corresponding Fermat transform. While ℓ must divide N for the Fermat transform so that the weight $w = 2^{N/\ell}$ is a power of two, it only needs to divide $2N$ for the Mersenne transform so that $\omega = 2^{2N/\ell}$ is a power of two. This improves the efficiency for ℓ near $\sqrt{N}$, and enables one to use a value of ℓ closer to optimal. (The FFT length can be doubled again by using $\sqrt{2}$ as a root of unity in the transform as described in §1.2.4.)

The above idea can be generalized to a Fermat transform mod $2^{aN} + 1$ and a Mersenne transform mod $2^{bN} - 1$ for small integers a, b.

Lemma 1. *Let a, b be two positive integers. Then at least one of* $\gcd(2^a + 1, 2^b - 1)$ *and* $\gcd(2^a - 1, 2^b + 1)$ *is* 1.

Proof. Both gcds are obviously odd. Let $g = \gcd(a, b)$, $r = 2^g$, $a' = a/g$, $b' = b/g$. Denote by $\mathrm{ord}_p(r)$ the multiplicative order of $r \pmod p$ for an odd prime p. In the case of b' odd, $p \mid r^{b'} - 1 \Rightarrow \mathrm{ord}_p(r) \mid b' \Rightarrow 2 \nmid \mathrm{ord}_p(r)$, and $p \mid r^{a'} + 1 \Rightarrow \mathrm{ord}_p(r) \mid 2a'$ and $\mathrm{ord}_p(r) \nmid a' \Rightarrow 2 \mid \mathrm{ord}_p(r)$, hence no prime can divide both $r^{b'} - 1$ and $r^{a'} + 1$. In the other case of b' even, a' must be odd, and the same argument holds with the roles of a' an b' exchanged, so no prime can divide both $r^{a'} - 1$ and $r^{b'} + 1$. □

It follows from Lemma 1 that for two positive integers a and b, either $2^{aN} + 1$ and $2^{bN} - 1$ are coprime, or $2^{aN} - 1$ and $2^{bN} + 1$ are coprime, thus we can use one Fermat transform of size aN (respectively bN) and one Mersenne transform of size bN (respectively aN). However this

[3] Here, a "Fermat transform" is meant modulo $2^N + 1$, without N being necessarily a power of two as in Fermat numbers.

does not imply that the reconstruction is easy: in practice we used $b = 1$ and made only a vary (see §1.2.7).

1.2.7 Parameter Selection and Automatic Tuning

SSA takes for n a multiple of ℓ, so that $\omega = 2^{2n/\ell}$ is a primitive ℓ-th root of unity, and $w = 2^{n/\ell}$ is used for the weight signal (or, if $\sqrt{2}$ is used as described in 1.2.4, $\omega = 2^{n/\ell}$ and $w = (\sqrt{2})^{n/\ell}$. The following omits the use of $\sqrt{2}$ for simplicity). As shown in 1.2.3, this ensures that all FFT butterflies involve only additions/subtractions and shifts on a radix 2 computer. In practice one may additionally require n to be a multiple of the machine word size β to simplify arithmetic modulo $2^n + 1$.

For a given size N divisible by $\ell = 2^k$, we define the *efficiency* of the FFT-ℓ scheme:

$$\frac{2N/\ell + k}{n},$$

where n is the smallest multiple of ℓ larger than or equal to $2N/\ell + k$. For example for $N = 1,000,448$ and $\ell = 2^{10}$, we have $2N/\ell + k = 1964$, and the next multiple of ℓ is $n = 2\ell = 2048$, therefore the efficiency is $\frac{1964}{2048} \approx 96\%$. For $N = 1,044,480$ with the same value of ℓ, we have $2N/\ell + k = 2050$, and the next multiple of ℓ is $n = 3\ell = 3072$, with an efficiency of about 67%. The FFT scheme is close to optimal when its efficiency is near 100%.

Note that a scheme with efficiency below 50% does not need to be considered. Indeed, this means that $2N/\ell + k \leq \frac{1}{2}n$, which necessarily implies that $n = \ell$ (as n has to be divisible by ℓ). Then the FFT scheme of length $\ell/2$ can be performed with the same value of n, since $2(N/(\ell/2)) + (k-1) < 4N/\ell + 2k \leq n$, and n is a multiple of $\ell/2$.

From this last remark, we can assume $2N/\ell \geq \frac{1}{2}n$ — neglecting the small k term —, which together with $n \geq \ell$ gives:

$$\ell \leq 2\sqrt{N}. \tag{1.9}$$

It should be noted that choosing n minimal according to the conditions $\ell \geq 2N/\ell + k$ and $\ell \mid 2n$ (e.g., for a Fermat transform with use of $\sqrt{2}$) is not always optimal. At the $j+1$-st recursive level of a length-ℓ FFT, we multiply by powers of an $\ell/2^j$-th root of unity, i.e., by $2^{i2^j 2n/\ell}$ for successive i, by performing suitable bit-shifts. When $2^j 2n/\ell$ is a multiple of the word size, no actual bit-shifts are performed any more, since the shift can be done by word-by-word copies. On system where bit-shifting is much more expensive than mere word-copying, e.g., if no well-optimized multiple precision shift code is available, it can be advantageous to choose n larger to make $2n/\ell$ divisible by a small power of 2. This way the number of FFT levels that perform bit-shifts is reduced. In our code, for transform lengths below the threshold for Bailey's algorithm and n that are small enough not to use SSA recursively, we ensure $\ell \mid n$ (even when $\sqrt{2}$ is used and $\ell \mid 2n$ would suffice for a Fermat transform). If the resulting n satisfies $n/l \equiv 3 \pmod 4$, we round up some more to make $4l \mid n$. The comparative timings of length 128 ($k = 7$) and length 256 ($k = 8$) can be seen in figure 1.4

Automatic Tuning

We found that significant speedups could be obtained with better tuning schemes, which we describe here. All examples given in this section are related to an Opteron.

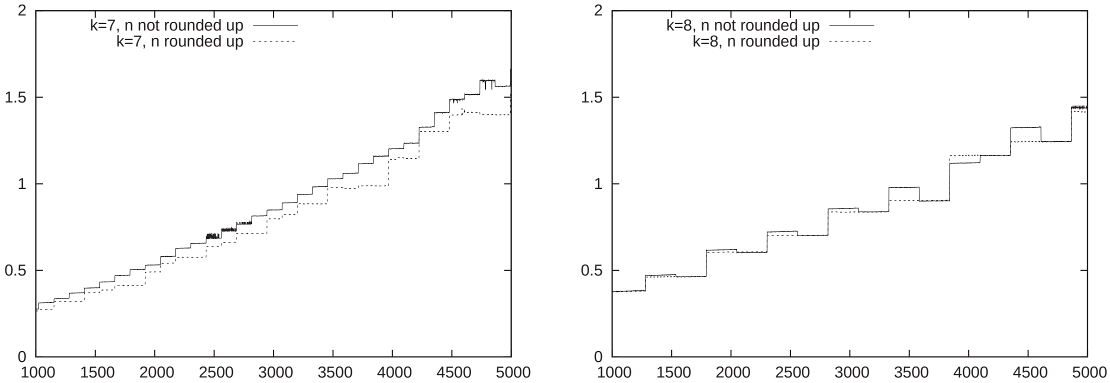

Figure 1.4: Time in milliseconds for a length 128 ($k = 7$) and length 256 ($k = 8$) FFT, for input sizes of 1000 to 5000 words, with and without rounding up n to avoid bit-shifts

Tuning the Fermat and Mersenne Transforms

Until version 4.2.1, GMP used a naive tuning scheme for the FFT multiplication. For the Fermat transforms modulo $2^N + 1$, an FFT of length 2^k was used for $t_k \leq N < t_{k+1}$, where t_k is the smallest bit-size for which FFT-2^k is faster than FFT-2^{k-1}. For example on an Opteron, the default `gmp-mparam.h` file uses $k = 4$ for a size less than 528 machine words, then $k = 5$ for less than 1184 words, and so on:

```
#define MUL_FFT_TABLE  { 528, 1184, 2880, 5376, 11264, 36864, 114688, 327680, 1310720,
  3145728, 12582912, 0 }
```

A special rule is used for the last entry: here $k = 14$ is used for less than $m = 12582912$ words, $k = 15$ is used for less than $4m = 50331648$ words, and then $k = 16$ is used. An additional single threshold determines from which size upward — still in words — a Fermat transform mod 2^n+1 is faster than a full product of two n-bit integers:

```
#define MUL_FFT_MODF_THRESHOLD          544
```

For a product mod 2^n+1 of at least 544 words, GMP 4.2.1 therefore uses a Fermat transform, with $k = 5$ until 1183 words according to the above `MUL_FFT_TABLE`. Below the 544 words threshold, the algorithm used is the 3-way Toom-Cook algorithm, followed by a reduction mod $2^n + 1$.

This scheme is not optimal since the FFT-2^k curves intersect several times, as shown by Figure 1.5.

To take into account those multiple crossings, the new tuning scheme determines word-intervals $[m_1, m_2]$ where the FFT of length 2^k is preferred for Fermat transforms:

```
#define MUL_FFT_TABLE2 {{1, 4 /*66*/}, {401, 5 /*96*/}, {417, 4 /*98*/},
  {433, 5 /*96*/}, {865, 6 /*96*/}, ...
```

The entry `{433, 5 /*96*/}` means that from 433 words — and up to the next size of 865 words — FFT-2^5 is preferred, with an efficiency of 96%. A similar table is used for Mersenne transforms.

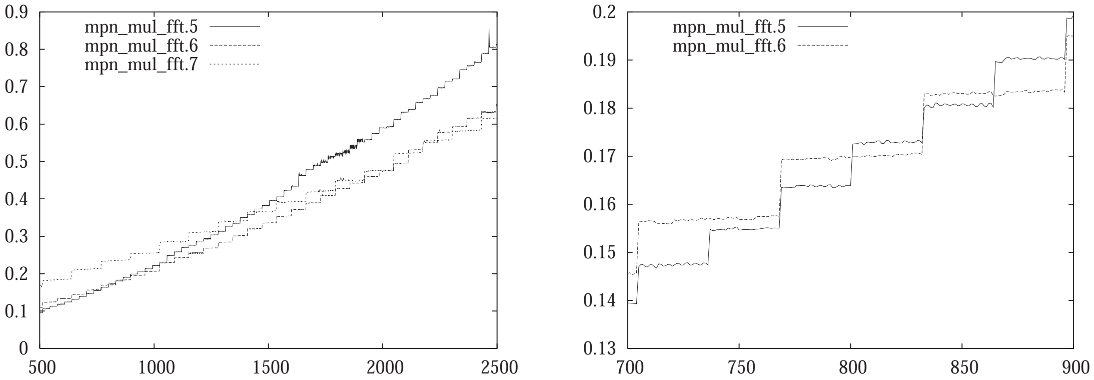

Figure 1.5: Time in milliseconds needed to multiply numbers modulo $2^n + 1$ with an FFT of length 2^k for $k = 5, 6, 7$. On the right, the zoom (with only $k = 5, 6$) illustrates that two curves can intersect several times.

Tuning the Plain Integer Multiplication

Up to GMP 4.2.1, a single threshold controls the plain integer multiplication:

```
#define MUL_FFT_THRESHOLD               7680
```

This means that SSA is used for a product of two integers of at least 7680 words, which corresponds to about $148,000$ decimal digits, and the Toom-Cook 3-way algorithm is used below that threshold.

We now use the generalized Fermat-Mersenne scheme described in 1.2.6 with $b = 1$ (in our implementation we found $1 \leq a \leq 7$ was enough). Again, for each size, the best value of a is determined by our tuning program:

```
#define MUL_FFT_FULL_TABLE2 {{16, 1}, {4224, 2}, {4416, 6}, {4480, 2},
  {4608, 4}, {4640, 2}, ...
```

For example, the entry `{4608, 4}` means that to multiply two numbers of 4608 words each — or whose product has 2×4608 words — and up to numbers of of 4639 words each, the new algorithm uses one Mersenne transform modulo $2^N - 1$ and one Fermat transform modulo $2^{4N} + 1$. Reconstruction is easy since $2^{aN} + 1 \equiv 2 \bmod (2^N - 1)$.

1.3 Results

On July 1st, 2005, Allan Steel wrote a web page [93] entitled "*Magma V2.12-1 is up to 2.3 times faster than GMP 4.1.4 for large integer multiplication*," which was a motivation for working on improving GMP's implementation and we compare our results to Magma's timings. We have also tested other freely available packages providing an implementation for large integer arithmetic. Among them, some (`OpenSSL/BN`, `LiDiA/libI`) do not go beyond Karatsuba algorithm, some do have some kind of FFT, but are not really made for really large integers: `arprec`, `Miracl`. Two useful implementations we have tested are `apfloat` and `CLN`. They take about 4 to 5 seconds on our test machine to multiply one million-word integers, whereas we need about 1 second. Bernstein mentions some partial implementation `Zmult` of Schönhage-Strassen's algorithm, with

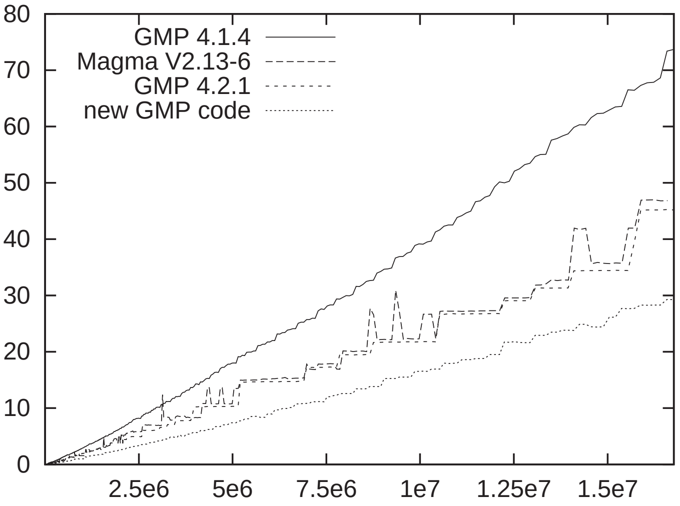

Figure 1.6: Comparison of GMP 4.1.4, GMP 4.2.1, Magma V2.13-6 and our new code for the plain integer multiplication on a 2.4GHz Opteron (horizontal axis in 64-bit words, vertical axis in seconds).

good timings, but right now, only very few sizes are handled, so that the comparison with our software is not really possible.

A program that implements a complex floating-point FFT for integer multiplication is George Woltman's `Prime95`. It is written mainly for the purpose of performing the Lucas-Lehmer algorithm for testing large Mersenne numbers $2^p - 1$ for primality in the Great Internet Mersenne Prime Search [102], and since its inception has found 10 new such primes, each one a new record at the time of its discovery. It uses a DWT for multiplication mod $a2^n \pm c$, with a and c not too large, see [79]. We compared multiplication modulo $2^{2wn} - 1$ in `Prime95` version 24.14.2 with multiplication of n-word integers using our SSA implementation on a Pentium 4 at 3.2 GHz, and on an Opteron 250 at 2.4 GHz, see Figure 1.7. It is plain that on a Pentium 4, `Prime95` beats our implementation by a wide margin, in fact usually by more than a factor of 10. On the Opteron, the difference is a bit less pronounced, where it is by a factor between 2.5 and 3. The reasons for this architecture dependency of the relative performance is that `Prime95` uses an SSE2 implementation of floating point FFT, which performs slightly better on the Pentium 4 than on the Opteron at a given clock rate, but more importantly that all-integer arithmetic as in SSA performs poorly on the Pentium 4, but excellently on the Opteron, due to both native 64 bit arithmetic and a very efficient integer ALU. Some other differences between `Prime95` and our implementation need to be pointed out in this context: due to the floating point nature of `Prime95`'s FFT, rounding errors can build up for particular input data to the point where the result will be incorrectly rounded to integers. While occurring with only low probability, this trait may be undesirable in scientific computation. In particular, the specification of GMP requires a correct multiplication algorithm for all input values, and when the first version of an FFT multiplication for GMP was written around 1998, it was not known how to choose parameters for a complex floating-point FFT so that correct rounding could be guaranteed in the convolution product. Therefore the preference was given to an all-integer algorithm such as Schönhage-Strassens where the problem of rounding errors does not occur. As it turns out, multiplication with the floating point FFT can be made provably correct, see again [79], but at the cost of using larger FFT lengths, thus giving up some performance. Figure 1.8 shows the maximum number of bits that can be stored per FFT element of type double so that provably correct rounding is possible. `Prime95`'s default choice uses between 1.3 and 2 times as many, so for multiplication of large integers, demanding provably correct rounding would about double the run time. Also, the DWT in `Prime95` needs to be initialized for a given modulus, and this initialization incurs overhead which becomes very costly if numbers of constantly varying sizes are to be multiplied. Finally, the implementation of the FFT in `Prime95` is done entirely in hand-optimized assembly for the x86 family of processors, and will not run on other architectures.

Another implementation of complex floating point FFT is Guillermo Ballester Valor's `Glucas`. The algorithm it uses is similar to that in `Prime95`, but it is written portably in C. This makes it slower than `Prime95`, but still faster than our code on both the Pentium 4 and the Opteron, as shown in Figure 1.7.

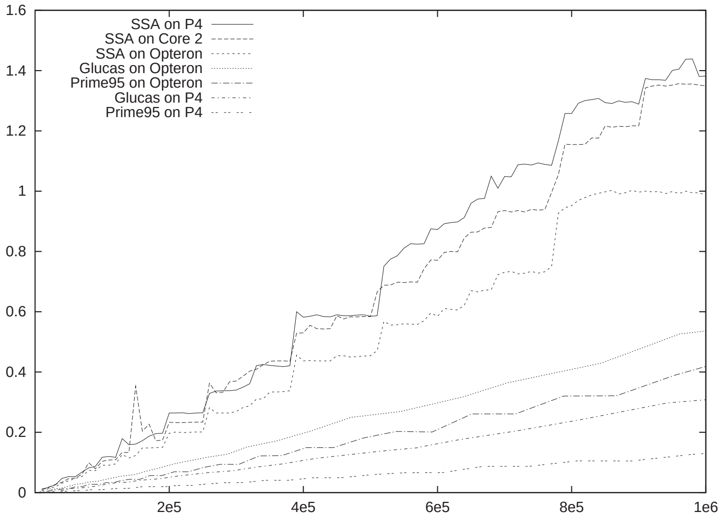

Figure 1.7: Time in seconds for multiplication of different word lengths with our implementation, Prime95 and Glucas on a 3.2 GHz Pentium 4, a 2.4 GHz Core 2, and a 2.4 GHz Opteron.

$K = 2^k$	2^{12}	2^{14}	2^{16}	2^{18}	2^{20}	2^{21}	2^{23}	2^{25}
bits/dbl	16	15	14	13	12	11	10	9
N	3.27e5	1.22e5	4.58e5	1.7e6	6.29e6	1.15e7	4.1e7	1.5e8
`Prime95`	21.37	21.08	20.78	20.49	20.22	19.94	18.29	17.76

Figure 1.8: Number of bits which can be stored in an IEEE 754 double-precision floating point number for provably correct multiplication of integers of bit-size N with an FFT of length K, and number of bits used in `Prime95` for FFT length K.

Fürer [43] proposed an integer multiplication algorithm with complexity $O\big(n\log(n)2^{\log^*(n)}\big)$, where $\log^*(n)$ is the minimum number of logarithms one needs to stack, starting from n, to get a result of at most 1. It is therefore asymptotically faster than Schönhage-Strassen's algorithm with complexity $O(n\log(n)\log(\log(n)))$, although the difference of the two asymptotic functions is small for n in the range of interest. We are not aware of a well-optimized implementation of Fürer algorithm, so no comparison of their speed in practice is possible at the moment.

Chapter 2

An Improved Stage 2 to P±1 Factoring Algorithms

2.1 Introduction

This chapter is joint work with Peter Lawrence Montgomery and was published in [73].

It extends the work of [74], a fast stage 2 for the P–1 algorithm based on polynomial multipoint evaluation where the points of evaluation lie in a geometric progression. The previous paper mentioned several ideas how the method could be improved by using patterns in the roots of the polynomial to build it more quickly, using symmetry in the resulting polynomial to reduce storage space and speed up polynomial arithmetic, and to adapt the method to the P+1 factoring algorithm.

These ideas are implemented in the current work, making efficient use of today's computers with large memory and multi-processing capability. Several large factors were found with the new implementation, including a 60-digit factor of the Lucas number L_{2366} by the P+1 method which still (at the end of 2009) stands as the record for this method. Some large factors were found with the P–1 method, listed in Section 2.13, but sadly no new record was set in spite of our best efforts.

2.2 The P–1 Algorithm

In 1974 John Pollard [80, §4] introduced the P–1 algorithm for factoring an odd composite integer N. It hopes that some prime factor p of N has smooth $p-1$. An integer is B-smooth if it has no prime factors exceeding B. It picks $b_0 \not\equiv \pm 1 \pmod{N}$ and coprime to N and outputs $b_1 = b_0^e \bmod N$ for some positive exponent e. This exponent might be divisible by all prime powers below a bound B_1. Stage 1 succeeds if $(p-1) \mid e$, in which case $b_1 \equiv 1 \pmod{p}$ by Fermat's little theorem. The algorithm recovers p by computing $\gcd(b_1 - 1, N)$ (except in rare cases when this gcd is composite). When this gcd is 1, we hope that $p - 1 = qn$ where n divides e and q is not too large. Then

$$b_1^q \equiv (b_0^e)^q = b_0^{eq} = (b_0^{nq})^{e/n} = \left(b_0^{p-1}\right)^{e/n} \equiv 1^{e/n} = 1 \pmod{p}, \tag{2.1}$$

so p divides $\gcd(b_1^q - 1,\, N)$. Stage 2 of P–1 tries to find p when $q > 1$ but q does not exceed the stage 2 search bound B_2.

Pollard tests each prime q in $]B_1,\, B_2]$ individually. If q_n and q_{n+1} are successive primes, he looks up $b_1^{q_{n+1}-q_n}$ mod N in a small table. It is conjectured that $q_{n+1} - q_n < \log(q_{n+1})^2$, related to Cramér's conjecture which states

$$\limsup_{n\to\infty} \frac{q_{n+1} - q_n}{\log(q_n)^2} = 1.$$

The prime gap that follows $q_n = 1693182318746371$ has length 1132, giving the largest quotient $(q_{n+1} - q_n)/\log(q_{n+1})^2 = 0.92\ldots$ currently known. This prime is greater than any that will be used in way of Pollard's stage 2, so in practice the size of the table can be bounded by $\log(B_2)^2/2$, as only even differences need to be stored if $B_1 > 2$. Given $b_1^{q_n}$ mod N, we can form $b_1^{q_{n+1}}$ mod $N = b_1^{q_n} b_1^{q_{n+1}-q_n}$ mod N with $b_1^{q_{n+1}-q_n}$ mod N taken from the precomputed table, and test $\gcd(b_1^{q_{n+1}} - 1,\, N)$. Pollard observes that one can combine gcd tests: if $p \mid \gcd(x,\, N)$ or $p \mid \gcd(y,\, N)$, then $p \mid \gcd(xy \bmod N,\, N)$. His stage 2 cost is two modular multiplications per q: one to compute $b_1^{q_n}$ and one to multiply $b_1^{q_n} - 1$ to an accumulator A; plus $O\Big(\log(B_2)^2\Big)$ multiplications to build the table and taking $\gcd(A, N)$ at the end, but these cost are asymptotically negligible.

Montgomery [65] uses two sets S_1 and S_2, such that each prime q in $]B_1,\, B_2]$ divides a nonzero difference $s_1 - s_2$ where $s_1 \in S_1$ and $s_2 \in S_2$. He forms $b_1^{s_1} - b_1^{s_2}$ using two table look-ups, saving one modular multiplication per q. Sometimes one $s_1 - s_2$ works for multiple q. Montgomery adapts his scheme to Hugh Williams's P+1 method and Hendrik Lenstra's Elliptic Curve Method (ECM). These changes lower the constant of proportionality, but stage 2 still uses $O(\pi(B_2) - \pi(B_1))$ operations modulo N.

The end of Pollard's original P–1 paper [80] suggests an FFT continuation to P–1. Montgomery and Silverman [74, p. 844] implement it, using a circular convolution to evaluate a polynomial along a geometric progression. It costs $O\big(\sqrt{B_2}\log(B_2)\big)$ operations to build and multiply two polynomials of degree $O\big(\sqrt{B_2}\big)$, compared to $O(B_2/\log(B_2))$ primes below B_2, so the FFT stage 2 beats Pollard's original stage 2 and Montgomery's variant from [65] when B_2 is large.

Montgomery's dissertation [67] describes an FFT continuation to ECM. He takes either the gcd of two polynomials, or uses a general multipoint evaluation method for polynomials with arbitrary points of evaluation. These cost an extra factor of $\log(B_2)$ compared to when the points are along a geometric progression. Zimmermann [103] implements these FFT continuations to ECM and uses them for the P±1 methods as well.

2.2.1 New Stage 2 Algorithm

Like in [74], in this chapter we evaluate a polynomial along geometric progressions. We exploit patterns in its roots to generate its coefficients quickly. We aim for low memory overhead, saving it for convolution inputs and outputs (which are elements of $\mathbb{Z}/N\mathbb{Z}$). Using memory efficiently lets us raise the convolution length ℓ. Many intermediate results are reciprocal polynomials, which need about half the storage and can be multiplied efficiently using weighted convolutions.

Doubling ℓ costs slightly over twice as much time per convolution, but each longer convolution extends the search for q (and effective B_2) fourfold. Silverman's 1989 implementation used 42 megabytes and allowed 250-digit inputs. It repeatedly evaluated a polynomial of degree 15360 at $8 \cdot 17408$ points in geometric progression, using $\ell = 32768$. This enabled him to achieve $B_2 \approx 10^{10}$.

Today's (2008) PC memories are 100 or more times as large as those used in [74]. With this extra memory, we achieve $\ell = 2^{23}$ or more, a growth factor of 256. With the same number of convolutions (individually longer lengths but running on faster hardware) our B_2 advances by a factor of $256^2 \approx 6.6 \cdot 10^4$. We make use of multi-processor systems by parallel execution of the convolution products. Supercomputers with many CPUs and huge shared memories do spectacularly.

Most techniques herein adapt to P+1, but some computations take place in an extension ring, raising memory usage if we use the same convolution sizes.

Section 2.13 gives some new results, including a record 60-digit P+1 factor.

The new algorithm to build a polynomial from its roots and the algorithm to evaluate that polynomial on points along a geometric progression make use of the ring structure of $\mathbb{Z}/NZ$ for P–1, or a quadratic extension thereof for P+1. This ring structure is not present in elliptic curves used for ECM, so these techniques do not apply to it. The method of choosing sets S_1 and S_2 as in Section 2.6 to determine the roots and evaluation points for the polynomial can be used for ECM, however, together with general algorithms of building a polynomial from its roots and evaluating it on an arbitrary set of points, like those described in [67] or [103].

2.3 The P+1 Algorithm

Hugh Williams [101] introduced a P+1 factoring algorithm in 1982. It finds a prime factor p of N when $p+1$ (rather than $p-1$) is smooth. It is modeled after P–1.

One variant of the P+1 algorithm chooses $P_0 \in \mathbb{Z}/N\mathbb{Z}$ and lets the indeterminate α_0 be a zero of the quadratic $f(x) = x^2 - P_0 x + 1$. The product of the two roots of this quadratic is the constant coefficient 1, hence they are α_0 and α_0^{-1}, and their sum is P_0. We hope that this quadratic is irreducible modulo the unknown factor p, i.e., that the discriminant $\Delta = P_0^2 - 4$ is a quadratic non-residue modulo p. If so, α_0 lies in $\mathbb{F}_{p^2}/\mathbb{F}_p$, i.e., has degree 2. By the Frobenius endomorphism, $\alpha_0^p \neq \alpha_0$ is the second root in $\mathbb{F}_{p^2}$. Hence $\alpha_0\alpha_1 \equiv \alpha_0\alpha_0^p = \alpha_0^{p+1} \equiv 1 \pmod{p}$ and the order of α_0 divides $p+1$.

On the other hand, if P_0^2-4 is a quadratic residue modulo p, then α_0 lies in $\mathbb{F}_p$ and $\alpha_0^p = \alpha_0$, so that $\alpha_0^{p-1} = 1$ and the order of α_0 divides $p-1$. In this case, the P+1 algorithm behaves like the P–1 algorithm. Since whether α_0 has order dividing $p-1$ or $p+1$ depends on the unknown prime p (and can vary for different prime factors p of one composite number), it generally is impossible to tell which order results from a particular choice P_0, unless, say, the resulting determinant is a rational square or known to be a quadratic residue modulo all candidate prime factors of the input number. Williams suggests testing three different values for P_0 to reach a confidence of 87.5% that an element of order dividing $p+1$ has been tried for a given prime factor p.

Stage 1 of the P+1 algorithm computes $P_1 = \alpha_1 + \alpha_1^{-1}$ where $\alpha_1 \equiv \alpha_0^e \pmod{N}$ for some exponent e, starting from $P_0 = \alpha_0 + \alpha_0^{-1}$ and using Chebyshev polynomials to simplify the computation. If $\mathrm{ord}(\alpha_0) \mid e$, regardless of whether $\mathrm{ord}(\alpha_0) \mid p-1$ or $\mathrm{ord}(\alpha_0) \mid p+1$, then $P_1 \equiv \alpha_0^e + \alpha_0^{-e} \equiv 1+1 \equiv 2 \pmod{N}$ and $\gcd(P_1 - 2, N) > 1$; if this gcd is also less than N, the algorithm succeeds. Stage 2 of P+1 hopes that $\alpha_1^q \equiv 1 \pmod{p}$ for some prime q, not too large, and some prime p dividing N.

2.3.1 Chebyshev Polynomials

Although the theory behind P+1 mentions α_0 and $\alpha_1 = \alpha_0^e$, an implementation manipulates primarily values of $\alpha_0^n + \alpha_0^{-n}$ and $\alpha_1^n + \alpha_1^{-n}$ for various integers n rather than the corresponding values (in an extension ring) of α_0^n and α_1^n.

For integer n, the Chebyshev polynomials V_n of degree n and U_n of degree $n-1$ are determined by $V_n(X+X^{-1}) = X^n + X^{-n}$ and $(X - X^{-1})U_n(X+X^{-1}) = X^n - X^{-n}$. The use of these polynomials shortens many formulas, such as

$$P_1 \equiv \alpha_1 + \alpha_1^{-1} \equiv \alpha_0^e + \alpha_0^{-e} = V_e(\alpha_0 + \alpha_0^{-1}) = V_e(P_0) \pmod{N}.$$

These polynomials have integer coefficients, so $P_1 \equiv V_e(P_0) \pmod{N}$ is in the base ring $\mathbb{Z}/N\mathbb{Z}$ even when α_0 and α_1 are not.

The Chebyshev polynomials satisfy many identities, including

$$\begin{aligned}
V_{mn}(X) &= V_m(V_n(X)), \\
U_{m+n}(X) &= U_m(X)\,V_n(X) - U_{m-n}(X), && (2.2)\\
U_{m+n}(X) &= V_m(X)\,U_n(X) + U_{m-n}(X), \\
V_{m+n}(X) &= V_m(X)\,V_n(X) - V_{m-n}(X), && (2.3)\\
V_{m+n}(X) &= (X^2-4)\,U_m(X)\,U_n(X) + V_{m-n}(X).
\end{aligned}$$

For given integers n and P_0, the value of the Chebyshev polynomial $V_n(P_0)$ can be evaluated by the methods of Montgomery [66].

2.4 Overview of Stage 2 Algorithm

Our algorithm performs multipoint evaluation of polynomials by convolutions. Its inputs are the output of stage 1 (b_1 for P–1 or P_1 for P+1), and the desired stage 2 interval $]B_1, B_2]$.

The algorithm chooses a highly composite odd integer P. It checks for q in arithmetic progressions with common difference $2P$. There are $\phi(P)$ such progressions to check when $\gcd(q, 2P) = 1$.

We need an even convolution length $\ell_{\max}$ (determined primarily by memory constraints) and a factorization $\phi(P) = s_1 s_2$ where s_1 is even and $0 < s_1 < \ell_{\max}$. Sections 2.6, 2.10.1 and 2.12 have sample values.

Our polynomial evaluations will need approximately

$$s_2 \left\lceil \frac{B_2}{2P(\ell_{\max} - s_1)} \right\rceil \approx \frac{\phi(P)}{2P} \frac{B_2}{s_1(\ell_{\max} - s_1)} \tag{2.4}$$

convolutions of length $\ell_{\max}$. We prefer a small $\phi(P)/P$ to keep (2.4) low. We also prefer s_1 near $\ell_{\max}/2$, say $0.3 \le s_1/\ell_{\max} \le 0.7$.

Using a factorization of $(\mathbb{Z}/P\mathbb{Z})^*$ as described in Section 2.6, we construct two sets S_1 and S_2 of integers such that

(a) $|S_1| = s_1$ and $|S_2| = s_2$.

(b) S_1 is symmetric around 0: if $k \in S_1$, then $-k \in S_1$.

(c) If $k \in \mathbb{Z}$ and $\gcd(k, P) = 1$, then there exist unique $k_1 \in S_1$ and $k_2 \in S_2$ such that $k \equiv k_1 + k_2 \pmod{P}$.

Once S_1 and S_2 are chosen, for the P–1 method we compute the coefficients of

$$f(X) = X^{-s_1/2} \prod_{k_1 \in S_1} (X - b_1^{2k_1}) \bmod N \tag{2.5}$$

by the method in Section 2.8. Since S_1 is symmetric around zero, this $f(X)$ is symmetric in X and $1/X$.

For each $k_2 \in S_2$ we evaluate (the numerators of) all

$$f(b_1^{2k_2+(2m+1)P}) \bmod N \tag{2.6}$$

for $\ell_{\max} - s_1$ consecutive values of m as described in Section 2.9, and check the product of these outputs for a nontrivial gcd with N. This checks $s_1(\ell_{\max} - s_1)$ candidates, not necessarily prime but including all primes in $]B_1, B_2]$, hoping to find q.

For the P+1 method, replace b_1 by α_1 in (2.5) and (2.6). The polynomial f is still over $\mathbb{Z}/N\mathbb{Z}$ since each product $(X - \alpha_1^{2k_1})(X - \alpha_1^{-2k_1}) = X^2 - V_{2k_1}(P_1) + 1$ is in $(\mathbb{Z}/N\mathbb{Z})[X]$, but the multipoint evaluation works in an extension ring. See Section 2.9.1.

2.5 Justification

Let p be an unknown prime factor of N. As in (2.1), assume $b_1^q \equiv 1 \pmod{p}$ where q is not too large, and $\gcd(q,\, 2P) = 1$.

The selection of S_1 and S_2 ensures there exist $k_1 \in S_1$ and $k_2 \in S_2$ such that $(q - P)/2 \equiv k_1 + k_2 \pmod{P}$. That is,

$$q = P + 2k_1 + 2k_2 + 2mP = 2k_1 + 2k_2 + (2m+1)P \tag{2.7}$$

for some integer m. We can bound m knowing bounds on q, k_1, k_2, detailed in Section 2.6. Both $b_1^{\pm 2k_1}$ are roots of $f \pmod{p}$ since S_1 is symmetric around 0 and by (2.5). Hence

$$f(b_1^{2k_2+(2m+1)P}) = f(b_1^{q-2k_1}) \equiv f(b_1^{-2k_1}) \equiv 0 \pmod{p}. \tag{2.8}$$

For the P+1 method, if $\alpha_1^q \equiv 1 \pmod{p}$, then (2.8) evaluates f at $X = \alpha_1^{2k_2+(2m+1)P} = \alpha_1^{q-2k_1}$. The factor $X - \alpha_1^{-2k_1}$ of $f(X)$ evaluates to $\alpha^{-2k_1}(\alpha_1^q - 1)$, which is zero modulo p even in the extension ring.

2.6 Selection of S_1 and S_2

Let "+" of two sets denote the set of sums. By the Chinese Remainder Theorem,

$$(\mathbb{Z}/(mn)\mathbb{Z})^* = n(\mathbb{Z}/m\mathbb{Z})^* + m(\mathbb{Z}/n\mathbb{Z})^* \text{ if } \gcd(m,n) = 1. \tag{2.9}$$

This is independent of the representatives: if $S \equiv (\mathbb{Z}/m\mathbb{Z})^* \pmod{m}$ and $T \equiv (\mathbb{Z}/n\mathbb{Z})^* \pmod{n}$, then $nS + mT \equiv (\mathbb{Z}/(mn)\mathbb{Z})^* \pmod{mn}$. For prime powers, we have $(\mathbb{Z}/p^k\mathbb{Z})^* = (\mathbb{Z}/p\mathbb{Z})^* + \sum_{i=1}^{k-1} p^i(\mathbb{Z}/p\mathbb{Z})$.

We choose S_1 and S_2 such that $S_1 + S_2 \equiv (\mathbb{Z}/P\mathbb{Z})^* \pmod{P}$ which ensures that all values coprime to P, in particular all primes, in the stage 2 interval are covered. One way uses a factorization $mn = P$ and (2.9). Other choices are available by factoring individual $(\mathbb{Z}/p\mathbb{Z})^*$, $p \mid P$, into smaller sets of sums.

Let $R_n = \{2i - n - 1 : 1 \le i \le n\}$ be an arithmetic progression centered at 0 of length n and common difference 2. For odd primes p, a set of representatives of $(\mathbb{Z}/p\mathbb{Z})^*$ is R_{p-1}. Its cardinality is composite for $p \ne 3$ and the set can be factored into arithmetic progressions of prime length by

$$R_{mn} = R_m + mR_n. \tag{2.10}$$

If $p \equiv 3 \pmod 4$, alternatively $\frac{p+1}{4}R_2 + \frac{1}{2}R_{(p-1)/2}$ can be chosen as a set of representatives with smaller absolute values. For example, for $p = 7$ we may use $\{-2, 2\} + \{-1, 0, 1\}$.

Example. For $P = 3 \cdot 5 \cdot 7 = 105$, we could use

$$(\mathbb{Z}/105\mathbb{Z})^* = 35(\mathbb{Z}/3\mathbb{Z})^* + 21(\mathbb{Z}/5\mathbb{Z})^* + 15(\mathbb{Z}/7\mathbb{Z})^*$$

by (2.9) and choose

$$S_1 + S_2 = 35\{-1, 1\} + 21\{-3, -1, 1, 3\} + 15\{-5, -3, -1, 1, 3, 5\}.$$

However, we can use (2.10) to write

$$\begin{aligned} \{-3, -1, 1, 3\} &= 2\{-1, 1\} + \{-1, 1\} \qquad \text{and} \\ \{-5, -3, -1, 1, 3, 5\} &= 3\{-1, 1\} + \{-2, 0, 2\}. \end{aligned}$$

Now we can choose $S_1 + S_2 = 35\{-1, 1\} + 42\{-1, 1\} + 21\{-1, 1\} + 45\{-1, 1\} + 15\{-2, 0, 2\}$, and let for example $S_1 = 35\{-1, 1\} + 42\{-1, 1\} + 21\{-1, 1\} + 45\{-1, 1\}$ and $S_2 = 15\{-2, 0, 2\}$ to make $s_1 = |S_1| = 16$ (close to) a power of 2 and $s_2 = |S_2|$ small.

When evaluating (2.6) for all $m_1 \le m < m_2$ and $k_2 \in S_2$, the highest exponent coprime to P that is *not* covered at the low end of the stage 2 range will be $2\max(S_1 + S_2) + (2m_1 - 1)P$. Similarly, the smallest value at the high end of the stage 2 range not covered is $2\min(S_1 + S_2) + (2m_2 + 1)P$. Hence, for a given choice of P, S_1, S_2, m_1 and m_2, all primes in $[(2m_1 - 1)P + 2\max(S_1 + S_2) + 1, (2m_2 + 1)P + 2\min(S_1 + S_2) - 1]$ are covered.

Example. To cover the interval $[1000, 500000]$ with $\ell_{\max} = 512$, we might choose $P = 1155$, $s_1 = 240$, $s_2 = 2$, $m_1 = -1$, and $m_2 = \ell_{\max} - s_1 + m_1 = 271$. With $S_1 = 231(\{-1, 1\} + \{-2, 2\}) + 165(\{-2, 2\} + \{-1, 0, 1\}) + 105(\{-3, 3\} + \{-2, -1, 0, 1, 2\})$ and $S_2 = 385\{-1, 1\}$, we have $\max(S_1 + S_2) = -\min(S_1 + S_2) = 2098$ and thus cover all primes in $[-3 \cdot 1155 + 4196 + 1,\ 543 \cdot 1155 - 4196 - 1] = [732, 622968]$.

For choosing a value of P which covers a desired $]B_1, B_2]$ interval, we can test candidate P from a table. This table could contain values so that P and $\phi(P)$ are increasing, and each P is maximal for its $\phi(P)$. We can select those P which, in order, cover the desired $]B_1, B_2]$ interval with $\ell_{\max}$ (limited by memory), minimize the cost of stage 2 and maximize $(2m_2 + 1)P + 2\min(S_1 + S_2)$. The table of P values may contain a large number of candidate values so that a near-optimal choice can be found for various $\ell_{\max}$ and B_2 parameters. To speed up selection of the optimal value of P, some restrictions on which P to test are desirable.

Assume S_1 and S_2 are symmetric around 0, so that $M = \max(S_1 + S_2) = -\min(S_1 + S_2)$. Then the effective start of the stage 2 interval is $2M + (2m_1 - 1)P + 1$, the effective end is $-2M + (2m_2 + 1)P - 1$, and their difference $-4M + 2P(m_2 - m_1 + 1) - 2$. Hence we require $B_2 - B_1 \le 2(m_2 - m_1 + 1)P$. Since $m_2 - m_1 + 1 \le \ell_{\max}$, this implies $B_2 - B_1 \le 2\ell_{\max}P$ or $P \ge (B_2 - B_1)/(2\ell_{\max})$, which together with an upper bound on $\ell_{\max}$ by available memory provides a lower bound on P.

The cost of stage 2 is essentially that of initialising the multi-point evaluation once per stage 2 by building the reciprocal Laurent polynomials $f(x)$ and $h(x)$ (see Section 2.9) and computing the discrete Fourier transform of $h(x)$, and that of performing the multi-point evaluation s_2 times per stage 2 by computing the polynomial $g(x)$, its product with $h(x)$ and the gcd of the coefficients of the product polynomial and N. The cost of polynomial multiplication is in $O(\ell_{\max} \log(\ell_{\max}))$, but for the sake of parameter selection can be approximated by just $\ell_{\max}$ — good parameters will use an $\ell_{\max}$ close to the largest possible, and for small changes of $\ell_{\max}$ (say, by up to a factor of 2), the effect of the $\log(\ell_{\max})$ term is small. The cost of building the polynomial $f(x)$ is proportional to the cost of polynomial multiplication so that we may take the cost of initialisation and of evaluation to be proportional with a positive real constant c. Hence the cost of stage 2 can be approximated roughly but usefully as $(c+s_2)\ell_{\max}$. Since $\ell_{\max} > s_1$ and $s_1 s_2 = \phi(P)$, we have $(c+s_2)\ell_{\max} > \phi(P)$ for any valid choice of stage 2 parameters, so that once a set of acceptable parameters has been found, its cost can serve as an upper bound on $\phi(P)$ when looking for better parameters. Since the entries in the table of P values are in order of increasing P and $\phi(P)$, the bound on $\phi(P)$ implies a bound on P.

For a given candidate P value within these bounds and for possible transform lengths $\ell_{\max}$ for the multi-point evaluation, choose s_1 and s_2 so that $s_1 s_2 = \phi(P)$, s_1 is even, $s_1 < \ell_{\max}$, s_2 is minimal and under these conditions, $|\ell_{\max}/2 - s_1|$ is minimal. For positive integers $n < 10^{10}$, the number of divisors of n does not exceed 4032 (attained for the highly composite number 97772875200) so that even exhaustive search of s_1 values from the prime factorization of $\phi(P)$ is sufficiently fast. If the multiplication routine for reciprocal Laurent polynomials (such as the one in Section 2.7.2) rounds up transform lengths to a power of 2, it is preferable to choose s_1 slightly below rather than slightly above a power of 2, so that having to round up transforms lengths by almost a factor of 2 is avoided when building f as described in Section 2.8. The resulting choice of P, $\ell_{\max}$, s_1, and s_2 is acceptable if the resulting m_1 and m_2 values allow covering the desired stage 2 interval $]B_1, B_2]$. Each such choice has an associated cost, and the acceptable choice with the smallest cost wins. If several have the same cost, we use the one with the largest effective B_2.

2.7 Cyclic Convolutions and Polynomial Multiplication with the NTT

Most of the CPU time in this algorithm is spent performing multiplication of polynomials with coefficients modulo N, the number to be factored. The Karatsuba (see Section 1.1.1) and Toom-Cook (see Section 1.1.2) algorithms could work directly over $R = \mathbb{Z}/N\mathbb{Z}$, so long as the interpolation phase does not involve division by a zero divisor of the ring, and since N is assumed not to have very small prime factors, this is not a problem in practice. However, the FFT stage 2 gains its speed by fast arithmetic on polynomials of very large degree, in which case FFT based multiplication algorithms (see Section 1.1.3) far exceed Karatsuba's or Toom-Cook's methods.

Unfortunately, the FFT for a large transform length ℓ cannot be used directly when $R = \mathbb{Z}/N\mathbb{Z}$, since we don't know a suitable ℓ-th primitive root of unity. Instead, we need to map coefficients of the polynomials to be multiplied to $\mathbb{Z}$ first, then to a ring that supports an FFT of the desired length, back to $\mathbb{Z}$ and to $\mathbb{Z}/N\mathbb{Z}$ again by reducing modulo N.

The Schönhage-Strassen algorithm described in Chapter 1 uses the ring $R = \mathbb{Z}/(2^n+1)\mathbb{Z}$ with $\ell \mid 2n$ (or $\ell \mid 4n$ if the $\sqrt{2}$ trick is used) and the ℓ-th root of unity $2^{2n/\ell} \in R$. It could be used for our purpose, but the condition $\ell \mid 2n$ often makes it impracticable: most frequently we want to factor input number of not too great size, say less than a few thousands bits, but use polynomials of degrees in the millions. For the Schönhage-Strassen algorithm, in that case we'd have to choose n in the millions also, too large by about three orders of magnitude. This would make the multiplication unacceptably slow and memory use prohibitive. The problem can be alleviated by the Kronecker-Schönhage segmentation trick, which reduces polynomial multiplication to integer multiplication, see [99] or [103, p. 534] and Section 2.12. However for larger numbers N and polynomials of smaller degree, say in the ten-thousands, using SSA directly is a viable option.

A very attractive approach to the problem of multiplying polynomials with relatively small coefficients and very large degree is the use of a number theoretic transform (NTT), described in the following.

Nussbaumer [77] gives other convolution algorithms.

2.7.1 Convolutions over $\mathbb{Z}/N\mathbb{Z}$ with the NTT

Montgomery and Silverman [74, Section 4] suggest a number theoretic transform (NTT). They treat the input polynomial coefficients as integers in $[0,\ N-1]$ and multiply the polynomials over $\mathbb{Z}$. The product polynomial, reduced modulo $X^\ell - 1$, has coefficients in $[0,\ \ell(N-1)^2]$. Select distinct NTT primes p_j that each fit into one machine word such that $\prod_j p_j > \ell(N-1)^2$. Require each $p_j \equiv 1 \pmod{\ell}$, so a primitive ℓ-th root of unity exists. Do the convolution modulo each p_j and use the Chinese Remainder Theorem (CRT) to determine the product over $\mathbb{Z}$ modulo $X^\ell - 1$. Reduce this product modulo N. Montgomery's dissertation [67, Chapter 8] describes these computations in detail.

The convolution codes need interfaces to (1) zero a Discrete Fourier Transform (DFT) buffer, (2) insert an entry modulo N in a DFT buffer, reducing it modulo the NTT primes, (3) perform a forward, in-place, DFT on a buffer, (4) multiply two DFT buffers point-wise, overwriting an input, and perform an in-place inverse DFT on the product, and (5) extract a product coefficient modulo N via a CRT computation and reduction modulo N.

2.7.2 Reciprocal Laurent Polynomials and Weighted NTT

Define a *reciprocal Laurent polynomial* (RLP) in x to be an expansion

$$a_0 + \sum_{j=1}^{d} a_j \left(x^j + x^{-j}\right) = a_0 + \sum_{j=1}^{d} a_j V_j \left(x + x^{-1}\right)$$

for scalars a_j in a ring. It is *monic* if $a_d = 1$. It is said to have degree $2d$ if $a_d \neq 0$. The degree is always even. A monic RLP of degree $2d$ fits in d coefficients (excluding the leading 1). While manipulating RLPs of degree at most $2d$, the *standard basis* is $\{1\} \cup \{x^j + x^{-j} : 1 \le j \le d\} = \{1\} \cup \{V_j\left(x + x^{-1}\right) : 1 \le j \le d\}$.

Let $Q(x) = q_0 + \sum_{j=1}^{d_q} q_j \left(x^j + x^{-j}\right)$ be an RLP of degree at most $2d_q$ and likewise $R(x)$ an RLP of degree at most $2d_r$. To obtain the product RLP $S(x) = s_0 + \sum_{j=1}^{d_s} s_j \left(x^j + x^{-j}\right) = Q(x)R(x)$ of degree at most $2d_s = 2(d_q + d_r)$, choose a convolution length $\ell > d_s$ and perform a weighted convolution product (as in Section 1.1.3) by computing $\tilde{S}(wx) = Q(wx)R(wx) \bmod \left(x^\ell - 1\right)$ for a suitable weight $w \neq 0$.

Suppose $\tilde{S}(x) = \sum_{j=0}^{\ell-1} \tilde{s} x^j$ and $\tilde{S}(wx) = S(wx) \bmod \left(x^\ell - 1\right)$. We have

$$\begin{aligned}
\tilde{S}(wx) &= s_0 + \sum_{j=1}^{d_s} \left(w^j s_j x^j + w^{-j} s_j x^{\ell-j}\right) \\
&= \sum_{j=0}^{d_s} w^j s_j x^j + \sum_{j=\ell-d_s}^{\ell-1} w^{j-\ell} s_{\ell-j} x^j \\
&= \sum_{j=0}^{\ell-d_s-1} w^j s_j x^j + \sum_{j=\ell-d_s}^{d_s} w^j \left(s_j + w^{-\ell} s_{\ell-j}\right) x^j + \sum_{j=d_s+1}^{\ell-1} w^{j-\ell} s_{\ell-j} x^j
\end{aligned}$$

and so $\tilde{s}_j = s_j$ for $0 \le j < \ell - d_s$, and $\tilde{s}_j = s_j + w^{-\ell} s_{l-j}$ for $\ell - d_s \le j \le d_s$. From the latter, we can solve for s_j and s_{l-j} by the linear system

$$\begin{pmatrix} 1 & w^{-\ell} \\ w^{-\ell} & 1 \end{pmatrix} \begin{pmatrix} s_j \\ s_{\ell-j} \end{pmatrix} = \begin{pmatrix} \tilde{s}_j \\ \tilde{s}_{\ell-j} \end{pmatrix}.$$

When the matrix is invertible, i.e., $w^\ell \neq \pm 1$, there is a unique solution which can be computed by

$$\begin{aligned} s_j &= \frac{w^{-l}\tilde{s}_j - \tilde{s}_{\ell-j}}{w^{-2\ell} - 1} \\ s_{l-j} &= \tilde{s}_{\ell-j} - w^{-\ell} s_j. \end{aligned}$$

This leads to Algorithm 4. It flows like the interface in Section 2.7.1.

Our implementation chooses the NTT primes $p_j \equiv 1 \pmod{3\ell}$. We require $3 \nmid \ell$. Our w is a primitive cube root of unity. Multiplications by 1 are omitted. When $3 \nmid i$, we use $w_j^i q_i + w_j^{-i} q_i \equiv -q_i \pmod{p_j}$ to save a multiply.

Substituting $x = e^{i\theta}$ where $i^2 = -1$ gives

$$Q(e^{i\theta})R(e^{i\theta}) = \left(q_0 + 2\sum_{j=1}^{d_q} q_j \cos(j\theta)\right)\left(r_0 + 2\sum_{j=1}^{d_r} r_j \cos(j\theta)\right).$$

These cosine series can be multiplied using discrete cosine transforms, in approximately the same auxiliary space needed by the weighted convolutions. We did not implement that approach.

2.7.3 Multiplying General Polynomials by RLPs

In Section 2.9 we will construct an RLP $h(x)$ which will later be multiplied by various $g(x)$. The length-ℓ DFT of $h(x)$ evaluates $h(\omega^i)$ for $0 \le i < \ell$, where ω is an ℓ-th primitive root of unity. However since $h(x)$ is reciprocal, $h(\omega^i) = h(\omega^{\ell-i})$ and the DFT has only $\ell/2+1$ distinct coefficients. In signal processing, the DFT of a signal extended symmetrically around the center of each endpoint is called a Discrete Cosine Transform of type I. Using a DCT–I algorithm [6], we could compute the coefficients $h(\omega^i)$ for $0 \le i \le \ell/2$ with a length $\ell/2+1$ transform. We have not implemented this.

Instead we compute the full DFT of the RLP (using $x^\ell = 1$ to avoid negative exponents). To conserve memory, we store only the $\ell/2+1$ possibly distinct DFT output coefficients for later use.

In the scrambled output of a decimation-in-frequency FFT of length $\ell = 2^r$, the possibly distinct DFT coefficients $h(\omega^i)$ for $0 \le i \le \ell/2$ are stored at even indices and at index 1. When we multiply $h(x)$ and one $g(x)$ via the FFT, each $h(\omega^i)$ for $0 < 2i < \ell$ must be multiplied to *two* coefficients of the FFT output of $g(x)$, which again will be in scrambled order. Rather than un-scrambling the transform coefficients for the point-wise multiplication, the correct index pairs to use can be computed directly.

For $0 < 2i < \ell$, the FFT coefficients of $h(x)$ stored at index $2i$ and index $m_i - 2i$, where $m_i = 2^{\lfloor \log_2(i) \rfloor + 3} - 2^{\lfloor \log_2(i) \rfloor + 1} - 1$, correspond to $h(\omega^{\mathrm{bitrev}_r(2i)})$ and $h(\omega^{\ell - \mathrm{bitrev}_r(2i)})$ and thus are equal. For the point-wise product with the scrambled FFT output of one $g(x)$, we can multiply the FFT coefficients of $g(x)$ stored at index $2i$ and $m_i - 2i$ by the FFT coefficient of h that was stored at index $2i$.

```
Procedure MUL_RLP ($s_{0\ldots d_q+d_r}$, $d_q$, $q_{0\ldots d_q}$, $d_r$, $r_{0\ldots d_r}$)
Input: RLP $Q(x) = q_0 + \sum_{j=1}^{d_q} q_j \left(x^j + x^{-j}\right)$ of degree at most $2d_q$
    RLP $R(x) = r_0 + \sum_{j=1}^{d_r} r_j \left(x^j + x^{-j}\right)$ of degree at most $2d_r$
    Convolution length $\ell > d_q + d_r$
    CRT primes $p_1, \ldots, p_k$
Output: RLP $S(x) = s_0 + \sum_{j=1}^{d_s} s_j \left(x^j + x^{-j}\right) = Q(x)R(x)$ of degree at most
        $2d_s = 2d_q + 2d_r$
    (Output may overlap input)
Data: NTT arrays $M$ and $M'$, each with $\ell$ elements per $p_j$ for auxiliary storage
    (A squaring uses only $M$)
Zero $M$ and $M'$
for $1 \le j \le k$ do
    Choose $w_j$ with $w_j^\ell \not\equiv 0, \pm 1 \pmod{p_j}$ ;
    $M_{j,0} := q_0 \bmod p_j$;
    $M'_{j,0} := r_0 \bmod p_j$;
for $1 \le i \le d_q$   (in any order) do
    for $1 \le j \le k$ do                                 /* Store $Q(wx) \bmod p_j$ in $M_j$ */
        $M_{j,i} := w_j^i q_i \bmod p_j$;
        $M_{j,\ell-i} := w_j^{-i} q_i \bmod p_j$;
if $Q(x) \ne R(x)$ then
    for $1 \le i \le d_r$   (in any order) do
        for $1 \le j \le k$ do                             /* Store $R(wx) \bmod p_j$ in $M'_j$ */
            $M'_{j,i} := w_j^i r_i \bmod p_j$;
            $M'_{j,\ell-i} := w_j^{-i} r_i \bmod p_j$;
for $1 \le j \le k$ do
    NTT_DIF($M_{j,0\ldots\ell-1}$, $\ell$, $p_j$);           /* Forward transform of $Q(wx) \bmod p_j$ */
    if $Q(x) \ne R(x)$ then
        NTT_DIF($M'_{j,0\ldots\ell-1}$, $\ell$, $p_j$);      /* Forward transform of $R(wx) \bmod p_j$ */
        $M_{j,0\ldots\ell-1} := M_{j,0\ldots\ell-1} \cdot M'_{j,0\ldots\ell-1} \bmod p_j$;      /* Point-wise product */
    else
        $M_{j,0\ldots\ell-1} := (M_{j,0\ldots\ell-1})^2 \bmod p_j$;                  /* Point-wise squaring */
    INTT_DIT($M_{j,0\ldots\ell-1}$, $\ell$, $p_j$);                              /* Inverse transform */
    for $1 \le i \le \ell - d_s - 1$ do
        $M_{j,i} := w_j^{-i} M_{j,i} \pmod{p_j}$;                                    /* Un-weighting */
    for $\ell - d_s \le i \le \lfloor \ell/2 \rfloor$ do
        $t := (w^{-l} M_j - M_{\ell-j})/(w^{-2\ell} - 1)$;
        $M_{l-j} := M_{\ell-j} - w^{-\ell} t$;
        $M_j := t$;
for $0 \le i \le d_s$ do
    $s_i :=$ CRT($M_{1\ldots j,i}$, $p_{1\ldots k}$) mod$N$
```

Algorithm 4: NTT-Based Multiplication Algorithm for reciprocal Laurent polynomials.

2.7.4 Multiplying RLPs without NTT

If no suitable transform-based multiplication algorithm is available for the weighted convolution of Section 2.7.2, RLPs can be multiplied with a regular polynomial multiplication routine, although less efficiently. Given an RLP $f(x) = f_0 + \sum_{i=1}^{d_f} f_i(x^i + x^{-i})$ of degree at most $2d_f$ in standard basis, we can write a polynomial $\tilde{f}(x) = f_0/2 + \sum_{i=1}^{d_f} f_i x^i$ of degree at most d_f so that $f(x) = \tilde{f}(x) + \tilde{f}(1/x)$. Likewise for $g(x)$ and $\tilde{g}(x)$.

Let $\operatorname{rev}\bigl(\tilde{f}(x)\bigr) = x^{d_f}\tilde{f}(1/x)$ denote the polynomial with reversed sequence of coefficients. We have $\operatorname{rev}\bigl(\operatorname{rev}\bigl(\tilde{f}(x)\bigr)\bigr) = \tilde{f}(x)$ and $\operatorname{rev}\bigl(\tilde{f}(x)\tilde{g}(x)\bigr) = \operatorname{rev}\bigl(\tilde{f}(x)\bigr)\operatorname{rev}\bigl(\tilde{g}(x)\bigr)$. Let $\lfloor f(x)\rfloor$ denote the polynomial whose coefficients at non-negative exponents of x are equal to those in $f(x)$, and whose coefficients at negative exponents of x are 0. We have $\lfloor f(x) + g(x)\rfloor = \lfloor f(x)\rfloor + \lfloor g(x)\rfloor$.

Now we can compute the product

$$\begin{aligned}
& f(x)g(x) \\
= \; & (\tilde{f}(x) + \tilde{f}(1/x))(\tilde{g}(x) + \tilde{g}(1/x)) \\
= \; & \tilde{f}(x)\tilde{g}(x) + \tilde{f}(x)\tilde{g}(1/x) + \tilde{f}(1/x)\tilde{g}(x) + \tilde{f}(1/x)\tilde{g}(1/x) \\
= \; & \tilde{f}(x)\tilde{g}(x) + x^{-d_g}\tilde{f}(x)\operatorname{rev}\bigl(\tilde{g}(x)\bigr) + x^{-d_f}\operatorname{rev}\bigl(\tilde{f}(x)\operatorname{rev}(\tilde{g}(x))\bigr) + \tilde{f}(1/x)\tilde{g}(1/x),
\end{aligned}$$

but we want to store only the coefficients at non-negative exponents in the product, so

$$\lfloor f(x)g(x)\rfloor = \tilde{f}(x)\tilde{g}(x) + \lfloor x^{-d_g}\tilde{f}(x)\operatorname{rev}\bigl(\tilde{g}(x)\bigr)\rfloor + \lfloor x^{-d_f}\operatorname{rev}\bigl(\tilde{f}(x)\operatorname{rev}(\tilde{g}(x))\bigr)\rfloor + \tilde{f}_0\tilde{g}_0$$

produces a polynomial whose coefficients in monomial basis are equal to those of the RLP $f(x)g(x)$ in standard basis. This computation uses two multiplications of polynomials of degrees at most d_f and d_g, respectively, whereas the algorithm in Section 2.7.2 has cost essentially equivalent to one such multiplication.

2.8 Computing Coefficients of f

Assume the P+1 algorithm. The monic RLP $f(X)$ in (2.5), with roots α_1^{2k} where $k \in S_1$, can be constructed using the decomposition of S_1. The coefficients of f will always be in the base ring since $P_1 \in \mathbb{Z}/N\mathbb{Z}$.

For the P–1 algorithm, set $\alpha_1 = b_1$ and $P_1 = b_1 + b_1^{-1}$. The rest of the construction of f for P–1 is identical to that for P+1.

Assume S_1 and S_2 are built as in Section 2.6, say $S_1 = T_1 + T_2 + \cdots + T_m$ where each T_j has an arithmetic progression of prime length, centered at zero. At least one of these has cardinality 2 since $s_1 = |S_1| = \prod_j |T_j|$ is even. Renumber the T_j so $|T_1| = 2$ and $|T_2| \ge |T_3| \ge \cdots \ge |T_m|$.

If $T_1 = \{-k_1, k_1\}$, then initialize $F_1(X) = X + X^{-1} - \alpha_1^{2k_1} - \alpha_1^{-2k_1} = X + X^{-1} - V_{2k_1}(P_1)$, a monic RLP in X of degree 2.

Suppose $1 \le j < m$. Given the coefficients of the monic RLP $F_j(X)$ with roots $\alpha_1^{2k_1}$ for $k_1 \in T_1 + \cdots + T_j$, we want to construct

$$F_{j+1}(X) = \prod_{k_2 \in T_{j+1}} F_j(\alpha_1^{2k_2}X). \tag{2.11}$$

The set T_{j+1} is assumed to be an arithmetic progression of prime length $t = |T_{j+1}|$ centered at zero with common difference k, say $T_{j+1} = \{(-1-t)k/2 + ik : 1 \le i \le t\}$. If t is even, k is even to ensure integer elements. On the right of (2.11), group pairs $\pm k_2$ when $k_2 \neq 0$. We need the coefficients of

$$F_{j+1}(X) = \begin{cases} F_j(\alpha_1^{-k}X)\, F_j(\alpha_1^{k}X), & \text{if } t = 2 \\ F_j(X) \prod_{i=1}^{(t-1)/2} \Big(F_j(\alpha_1^{2ki}X)\, F_j(\alpha_1^{-2ki}X) \Big), & \text{if } t \text{ is odd.} \end{cases} \tag{2.12}$$

Let $d = \deg(F_j)$, an even number. The monic input F_j has $d/2$ coefficients in $\mathbb{Z}/N\mathbb{Z}$ (plus the leading 1). The output F_{j+1} will have $td/2 = \deg(F_{j+1})/2$ such coefficients.

Products such as $F_j(\alpha_1^{2ki}X)\, F_j(\alpha_1^{-2ki}X)$ can be formed by the method in Section 2.8.1, using d coefficients to store each product. The interface can pass $\alpha_1^{2ki} + \alpha_1^{-2ki} = V_{2ki}(P_1) \in \mathbb{Z}/N\mathbb{Z}$ as a parameter instead of $\alpha_1^{\pm 2ki}$.

For odd t, the algorithm in Section 2.8.1 forms $(t-1)/2$ such monic products each with d output coefficients. We still need to multiply by the input F_j. Overall we store $(d/2) + \frac{t-1}{2}d = td/2$ coefficients. Later these $(t+1)/2$ monic RLPs can be multiplied in pairs, with products overwriting the inputs, until F_{j+1} (with $td/2$ coefficients plus the leading 1) is ready.

All polynomial products needed for (2.11), including those in Section 2.8.1, have output degree at most $t \deg(F_j) = \deg(F_{j+1})$, which divides the final $\deg(F_m) = s_1$. The polynomial coefficients are saved in the (MZNZ) buffer of 2.10. The (MDFT) buffer allows convolution length $\ell_{\max}/2$, which is adequate when an RLP product has degree up to $2(\ell_{\max}/2) - 1 \geq s_1$. A smaller length might be better for a particular product.

2.8.1 Scaling by a Power and Its Inverse

Let $F(X)$ be a monic RLP of even degree d, say $F(X) = c_0 + \sum_{i=1}^{d/2} c_i(X^i + X^{-i})$, where each $c_i \in \mathbb{Z}/N\mathbb{Z}$ and $c_{d/2} = 1$. Given $Q \in \mathbb{Z}/N\mathbb{Z}$, where $Q = \gamma + \gamma^{-1}$ for some unknown γ, we want the d coefficients (excluding the leading 1) of $F(\gamma X)\, F(\gamma^{-1}X) \bmod N$ in place of the $d/2$ such coefficients of F. We are allowed a few scalar temporaries and any storage internal to the polynomial multiplier.

Denote $Y = X + X^{-1}$. Rewrite, while pretending to know γ,

$$\begin{aligned} F(\gamma X) &= c_0 + \sum_{i=1}^{d/2} c_i(\gamma^i X^i + \gamma^{-i} X^{-i}) \\ &= c_0 + \sum_{i=1}^{d/2} \frac{c_i}{2} \Big((\gamma^i + \gamma^{-i})(X^i + X^{-i}) + (\gamma^i - \gamma^{-i})(X^i - X^{-i}) \Big) \\ &= c_0 + \sum_{i=1}^{d/2} \frac{c_i}{2} \Big(V_i(Q)V_i(Y) + (\gamma - \gamma^{-1})U_i(Q)(X - X^{-1})U_i(Y) \Big). \end{aligned}$$

Replace γ by γ^{-1} and multiply to get

$$\begin{aligned} F(\gamma X)\, F(\gamma^{-1}X) &= G^2 - (\gamma - \gamma^{-1})^2 (X - X^{-1})^2\, H^2 \\ &= G^2 - (Q^2 - 4)(X - X^{-1})^2\, H^2, \end{aligned} \tag{2.13}$$

where

$$\begin{aligned} G &= c_0 + \sum_{i=1}^{d/2} c_i \frac{V_i(Q)}{2} V_i(Y) \\ H &= \sum_{i=1}^{d/2} c_i \frac{U_i(Q)}{2} U_i(Y). \end{aligned}$$

This G is a (not necessarily monic) RLP of degree at most d in the standard basis $\{1\} \cup \{V_j(Y) : 1 \le j \le d/2\}$, with coefficients in $\mathbb{Z}/N\mathbb{Z}$. This H is another RLP, of degree at most $d-2$, but using the basis $\{U_i(Y) : 1 \le i \le d/2\}$. Starting with the coefficient of $U_{d/2}(Y)$, we can repeatedly use $U_{j+1}(Y) = V_j(Y)U_1(Y) + U_{j-1}(Y) = V_j(Y) + U_{j-1}(Y)$ for $j > 0$, along with $U_1(Y) = 1$ and $U_0(Y) = 0$, to convert H to standard basis. This conversion costs $O(d)$ additions in $\mathbb{Z}/N\mathbb{Z}$.

Use the identities $V_{i+1}(Q) = QV_i(Q) - V_{i-1}(Q)$ and $U_{i+1}(Q) = QU_i(Q) - U_{i-1}(Q)$ from (2.3) and (2.2) to evaluate $V_i(Q)/2$ and $U_i(Q)/2$ for consecutive i when computing the $d/2+1$ coefficients of G and the $d/2$ coefficients of H. If a weighted NTT-based algorithm such as Algorithm 4 is used for multiplying RLPs and a memory model as in Section 2.10, the algorithm can write the NTT images of the standard-basis coefficients of G and H to different parts of (MDFT) and recover the coefficients of G^2 and H^2 via the CRT and combine them directly into the coefficients of $F(\gamma x)F(\gamma^{-1}x)$ to avoid allocating temporary storage for G and H. Algorithm 5 shows a simplified implementation with temporary storage.

2.9 Multipoint Polynomial Evaluation

We have constructed $f = F_m$ in (2.5). The monic RLP $f(X)$ has degree s_1, say $f(X) = f_0 + \sum_{j=1}^{s_1/2} f_j \cdot (X^j + X^{-j}) = \sum_{j=-s_1/2}^{s_1/2} f_j X^j$ where $f_j = f_{-j} \in \mathbb{Z}/N\mathbb{Z}$.

Assuming the P–1 method (otherwise see Section 2.9.1), compute $r = b_1^P \in \mathbb{Z}/N\mathbb{Z}$. Set $\ell = \ell_{\max}$ and $M = \ell - 1 - s_1/2$.

Equation (2.6) needs $\gcd(f(X),\, N)$ where $X = b_1^{2k_2+(2m+1)P}$, for several consecutive m, say $m_1 \le m < m_2$. By setting $x_0 = b_1^{2k_2+(2m_1+1)P}$, the arguments to f become $x_0 b_1^{2mP} = x_0 r^{2m}$ for $0 \le m < m_2 - m_1$. The points of evaluation form a geometric progression with ratio r^2. We can evaluate these for $0 \le m < \ell - 1 - s_1$ with one convolution of length ℓ and $O(\ell)$ setup cost [1, exercise 8.27].

To be precise, set $h_j = r^{-j^2} f_j$ for $-s_1/2 \le j \le s_1/2$. Then $h_j = h_{-j}$. Set $h(X) = \sum_{j=-s_1/2}^{s_1/2} h_j X^j$, an RLP. The construction of h does not reference x_0 — we reuse h as x_0 varies.

Let $g_i = x_0^{M-i}\, r^{(M-i)^2}$ for $0 \le i \le \ell - 1$ and $g(X) = \sum_{i=0}^{\ell-1} g_i X^i$.

All nonzero coefficients in $g(X)h(X)$ have exponents from $0 - s_1/2$ to $(\ell-1) + s_1/2$. Suppose $0 \le m \le \ell - 1 - s_1$. Then $M - m - \ell = -1 - s_1/2 - m < -s_1/2$ whereas $M - m + \ell = (\ell - 1 + s_1/2) + (\ell - s_1 - m) > \ell - 1 + s_1/2$. The coefficient of X^{M-m} in $g(X)h(X)$, reduced modulo $X^\ell - 1$, is

$$\sum_{\substack{0 \le i \le \ell-1 \\ -s_1/2 \le j \le s_1/2 \\ i+j \equiv M-m \pmod{\ell}}} g_i h_j = \sum_{\substack{0 \le i \le \ell-1 \\ -s_1/2 \le j \le s_1/2 \\ i+j = M-m}} g_i h_j = \sum_{j=-s_1/2}^{s_1/2} g_{M-m-j} h_j$$

$$= \sum_{j=-s_1/2}^{s_1/2} x_0^{m+j} r^{(m+j)^2} r^{-j^2} f_j = \sum_{j=-s_1/2}^{s_1/2} x_0^m r^{m^2} \left(x_0 r^{2m}\right)^j f_j = x_0^m r^{m^2} f(x_0 r^{2m}).$$

Since we want only $\gcd(f(x_0\, r^{2m}),\, N)$, the $x_0^m\, r^{m^2}$ factors are harmless.

We can compute successive $g_{\ell-i}$ with two ring multiplications each as the ratios $g_{\ell-1-i}/g_{\ell-i} = x_0\, r^{2i-s_1-1}$ form a geometric progression.

```
Input: RLP $F(x) = f_0 + \sum_{i=1}^{d/2} f_i(x^i + x^{-i})$, $f_i \in \mathbb{Z}/N\mathbb{Z}$
    $Q \in \mathbb{Z}/N\mathbb{Z}$
Output: Coefficients of $F(\gamma x)F(\gamma^{-1}x)$ in standard basis $\{1\} \cup \{x^i + x^{-i} : 1 \leq i \leq d\}$,
        where $\gamma + \gamma^{-1} = Q$, overwriting $f_{0\ldots d}$
Data: Storage for $v, u, g_i, h_i \in \mathbb{Z}/N\mathbb{Z}$, for $0 \leq i < d$
$v := 1$;                                                    /* $V_0(Q)/2 = 1$ */
$u := Q/2$;                                                  /* $V_1(Q)/2 = Q/2$ */
$g_0 = f_0$;
for $i := 1$ to $d/2$ do
    $g_i = f_i u$;
    $(u, v) := (uQ - v, u)$;                    /* $u := V_{i+1}(Q)/2, v := V_i(Q)/2$ */
/* Now $G = g_0 + \sum_{i=1}^{d/2} g_i V_i\left(x + x^{-1}\right)$ */
MUL_RLP ($g_{(0,\ldots,d)}$, $g_{(0,\ldots,d/2)}$, $d/2$, $g_{(0,\ldots,d/2)}$, $d/2$);        /* Use Algorithm 4 */
/* Now $G^2 = g_0 + \sum_{i=1}^{d} g_i V_i\left(x + x^{-1}\right)$ */
$v := 0$;                                                    /* $v := U_0(Q)/2 = 0$ */
$u := 1/2$;                                                  /* $u := U_1(Q)/2 = 1/2$ */
for $i := 1$ to $d/2$ do /* store $h_i$ shifted by 1 to simplify change to $V_i$ basis */
    $h_{i-1} := f_i u$;
    $(u, v) := (uQ - v, u)$;                    /* $u := U_{i+1}(Q)/2, v := U_i(Q)/2$ */
/* Now $H = \sum_{i=1}^{d/2} h_{i-1} U_i\left(x + x^{-1}\right)$ */
for $i := d/2$ downto 3 do                      /* convert $h_i$ from $U_i$ to $V_i$ basis */
    $h_{i-3} := h_{i-3} + h_{i-1}$;
/* Now $H = h_0 + \sum_{i=1}^{d/2-1} h_i V_i\left(x + x^{-1}\right)$, i.e., in standard basis */
MUL_RLP ($h_{(0,\ldots,d-2)}$, $h_{(0,\ldots,d/2-1)}$, $d/2-1$, $h_{(0,\ldots,d/2-1)}$, $d/2-1$); /* Use Algorithm 4 */
/* Now $H^2 = h_0 + \sum_{i=1}^{d-2} h_i V_i\left(x + x^{-1}\right)$ */
for $i := 0$ to $d - 2$ do
    $h_i := h_i\left(Q^2 - 4\right)$;
/* Now $\left(Q^2 - 4\right) H^2 = h_0 + \sum_{i=1}^{d-2} h_i V_i\left(x + x^{-1}\right)$ */
/* Compute $G^2 + \left(x - x^{-1}\right)^2 \left(Q^2 - 4\right) H^2$ */
if $d = 2$ then
    $g_0 := g_0 + 2h_0$;
    $g_2 := g_2 - h_0$;
else
    $g_0 := g_0 + 2(h_0 - h_2)$;
    $g_1 := g_1 + h_1$;
    if $d > 4$ then
        $g_1 := g_1 - h_3$;
    for $i := 2$ to $d - 4$ do
        $g_i := g_i - h_{i-2} + 2h_i - h_{i+2}$;
    for $i := d - 3$ to $d - 2$ do
        $g_i := g_i - h_{i-2} + 2h_i$;
    $g_{d-1} := g_{d-1} - h_{d-3}$;
    $g_d := g_d - h_{d-2}$;
for $i := 0$ to $d$ do
    $f_i := g_i$;                                            /* Store result in $f$ */
```

Algorithm 5: Algorithm for scaling a reciprocal Laurent polynomial by a power and its inverse.

2.9.1 Adaptation for P+1 Algorithm

If we replace b_1 with α_1, then r becomes α_1^P, which satisfies $r + r^{-1} = V_P(P_1)$. The above algebra evaluates f at powers of α_1. However α_1, r, h_j, x_0, and g_i lie in an extension ring.

Arithmetic in the extension ring can use a basis $\{1, \sqrt{\Delta}\}$ where $\Delta = P_1^2 - 4$. The element α_1 maps to $(P_1 + \sqrt{\Delta})/2$. A product $(c_0 + c_1\sqrt{\Delta})(d_0 + d_1\sqrt{\Delta})$ where c_0, c_1, d_0, $d_1 \in \mathbb{Z}/N\mathbb{Z}$ can be done using four base-ring multiplications: $c_0 d_0$, $c_1 d_1$, $(c_0 + c_1)(d_0 + d_1)$, $c_1 d_1 \Delta$, plus five base-ring additions.

We define linear transformations E_1, E_2 on $(\mathbb{Z}/N\mathbb{Z})[\sqrt{\Delta}]$ so that $E_1(c_0 + c_1\sqrt{\Delta}) = c_0$ and $E_2(c_0 + c_1\sqrt{\Delta}) = c_1$ for all c_0, $c_1 \in \mathbb{Z}/N\mathbb{Z}$. Extend E_1 and E_2 to polynomials by applying them to each coefficient.

Some multiplication involves powers of α_1 and r. These have norm 1, which may allow simplifications. For example,

$$(c_0 + c_1\sqrt{\Delta})^2 = 2c_0^2 - 1 + 2c_0c_1\sqrt{\Delta}$$

needs only two multiplications and three additions if $c_0^2 - c_1^2\Delta = 1$.

To compute r^{n^2} for successive n, we use recurrences. We observe

$$\begin{aligned} r^{n^2} &= r^{(n-1)^2+2} \cdot V_{2n-3}(r + r^{-1}) - r^{(n-2)^2+2}, \\ r^{n^2+2} &= r^{(n-1)^2+2} \cdot V_{2n-1}(r + r^{-1}) - r^{(n-2)^2} \ . \end{aligned}$$

After initializing the variables $\mathtt{r1}[i] := r^{i^2}$, $\mathtt{r2}[i] := r^{i^2+2}$, $\mathtt{v}[i] := V_{2i+1}(r + r^{-1})$ for two consecutive i, we can compute $\mathtt{r1}[i] = r^{i^2}$ for larger i in sequence by

$$\begin{aligned} \mathtt{r1}[i] &:= \mathtt{r2}[i-1] \cdot \mathtt{v}[i-2] - \mathtt{r2}[i-2], \\ \mathtt{r2}[i] &:= \mathtt{r2}[i-1] \cdot \mathtt{v}[i-1] - \mathtt{r1}[i-2], \\ \mathtt{v}[i] &:= \mathtt{v}[i-1] \cdot V_2(r + 1/r) - \mathtt{v}[i-2] \ . \end{aligned} \tag{2.14}$$

Since we won't use $\mathtt{v}[i-2]$ and $\mathtt{r2}[i-2]$ again, we can overwrite them with $\mathtt{v}[i]$ and $\mathtt{r2}[i]$. For the computation of r^{-n^2} where r has norm 1, we can use r^{-1} as input, by taking the conjugate.

All $\mathtt{v}[i]$ are in the base ring but $\mathtt{r1}[i]$ and $\mathtt{r2}[i]$ are in the extension ring. Each application of (2.14) takes five base-ring multiplications (compared to two multiplications per r^{n^2} in the P–1 algorithm).

We can compute successive $g_i = x_0^{M-i} r^{(M-i)^2}$ similarly. One solution to (2.14) is $\mathtt{r1}[i] = g_i$, $\mathtt{r2}[i] = r^2 g_i$, $\mathtt{v}[i] = x_0 r^{2M-2i-1} + x_0^{-1} r^{1+2i-2M}$. Again each $\mathtt{v}[i]$ is in the base ring, so (2.14) needs only five base-ring multiplications.

If we try to follow this approach for the multipoint evaluation, we need twice as much space for an element of $(\mathbb{Z}/N\mathbb{Z})[\sqrt{\Delta}]$ as one of $\mathbb{Z}/N\mathbb{Z}$. We also need a convolution routine for the extension ring.

If p divides the coefficient of X^{M-m} in $g(X)h(X)$, then p divides both coordinates thereof. The coefficients of $g(X)h(X)$ occasionally lie in the base ring, making $E_2(g(X)h(X))$ a poor choice for the gcd with N. Instead we compute

$$E_1(g(X)h(X)) = E_1(g(X))E_1(h(X)) + \Delta E_2(g(X))E_2(h(X)) \ . \tag{2.15}$$

The RLPs $E_1(h(X))$ and $\Delta E_2(h(X))$ can be computed once and for each of the $\ell_{\max}/2 + 1$ distinct coefficients of its length $\ell_{\max}$ DFT saved in (MHDFT). To compute $\Delta E_2(h(X))$, multiply $E_2(\mathtt{r1}[i])$ and $E_2(\mathtt{r2}[i])$ by Δ after initializing for two consecutive i. Then apply (2.14).

Later, as each g_i is computed we insert the NTT image of $E_2(g_i)$ into (MDFT) while saving $E_1(g_i)$ in (MZNZ) for later use. After forming $E_1(g(X))E_1(h(X))$, retrieve and save coefficients of X^{M-m} for $0 \le m \le \ell - 1 - s_1$. Store these in (MZNZ) while moving the entire saved $E_1(g_i)$ into the (now available) (MDFT) buffer. Form the $E_2(g(X))E_2(\Delta h(X))$ product and the sum in (2.15).

2.10 Memory Allocation Model

We aim to fit our major data into the following:

(MZNZ) An array with $s_1/2$ elements of $\mathbb{Z}/N\mathbb{Z}$, for convolution inputs and outputs. This is used during polynomial construction.

This is not needed during P–1 evaluation. During P+1 evaluation, it grows to $\ell_{\max}$ elements of $\mathbb{Z}/N\mathbb{Z}$ (if we compute both coordinate of each g_i together, saving one of them), or $\ell_{\max} - s_1$ elements (if we compute the coordinates individually).

(MDFT) An NTT array holding $\ell_{\max}$ values modulo p_j per prime p_j, for use during DWTs.

Section 2.8.1 does two overlapping squarings, whereas Section 2.8 multiplies two arbitrary RLPs. Each product degree is at most $\deg(f) = s_1$. Algorithm 4 needs $\ell \ge s_1/2$ and might use convolution length $\ell = \ell_{\max}/2$, assuming $\ell_{\max}$ is even. Two arrays of this length fit in MDFT.

After f has been constructed, MDFT is used for NTT transforms with length up to $\ell_{\max}$.

(MHDFT) Section 2.9 scales the coefficients of f by powers of r to build h. Then it builds and stores a length-ℓ DFT of h, where $\ell = \ell_{\max}$. This transform output normally needs ℓ elements per p_j for P–1 and 2ℓ elements per p_j for P+1. The symmetry of h lets us cut these needs almost in half, to $\ell/2 + 1$ elements for P–1 and $\ell + 2$ elements for P+1.

During the construction of F_{j+1} from F_j, if we need to multiply pairs of monic RLPs occupying adjacent locations within (MZNZ) (without the leading 1's), we use (MDFT) and algorithm 4. The outputs overwrite the inputs within (MZNZ).

During polynomial evaluation for P–1, we need only (MHDFT) and (MDFT). Send the NTT image of each g_i coefficient to (MDFT) as g_i is computed. When (MDFT) fills (with $\ell_{\max}$ entries), do a length-$\ell_{\max}$ forward DFT on (MDFT), pointwise multiply by the saved DFT output from h in (MHDFT), and do an inverse DFT in (MDFT). Retrieve each needed polynomial coefficient, compute their product, and take a gcd with N.

2.10.1 Potentially Large B_2

In 2008/2009, a typical PC memory is 4 gigabytes. The median size of composite cofactors N in the Cunningham project **http://homes.cerias.purdue.edu/~ssw/cun/index.html** is about 230 decimal digits, which fits in twelve 64-bit words (called *quadwords*). Table 2.1 estimates the memory requirements during Stage 2, when factoring a 230-digit number, for both polynomial construction and polynomial evaluation phases, assuming convolutions use the NTT approach in Section 2.7.1. The product of our NTT prime moduli must be at least $\ell_{\max}(N-1)^2$. If $N^2\ell_{\max}$ is below $0.99 \cdot (2^{63})^{25} \approx 10^{474}$, then it will suffice to have 25 NTT primes, each 63 or 64 bits.

The P–1 polynomial construction phase uses an estimated $40.5\ell_{\max}$ quadwords, vs. $37.5\ell_{\max}$ quadwords during polynomial evaluation. We can reduce the overall maximum to $37.5\ell_{\max}$ by

Table 2.1: Estimated memory usage (quadwords) while factoring 230-digit number.

Array name	Construct f. Both P±1	Build h.	Evaluate f.
(MZNZ)	$12(s_1/2)$	$12(s_1/2)$	0 (P–1) $12\ell_{\max}$ (P+1)
(MDFT)	$25\ell_{\max}$	$25\ell_{\max}$	$25\ell_{\max}$
(MHDFT)	0	$25(\ell_{\max}/2+1)$ (P–1) $25(\ell_{\max}+2)$ (P+1)	$25(\ell_{\max}/2+1)$ (P–1) $25(\ell_{\max}+2)$ (P+1)
Totals, if $s_1 = \ell_{\max}/2$	$28\ell_{\max}+O(1)$	$40.5\ell_{\max}+O(1)$ (P–1) $53\ell_{\max}+O(1)$ (P+1)	$37.5\ell_{\max}+O(1)$ (P–1) $62\ell_{\max}+O(1)$ (P+1)

taking the (full) DFT transform of h in (MDFT), and releasing the (MZNZ) storage before allocating (MHDFT).

Four gigabytes is 537 million quadwords. A possible value is $\ell_{\max} = 2^{23}$, which needs 315 million quadwords. When transform length $3 \cdot 2^k$ is supported, we could use $\ell_{\max} = 3 \cdot 2^{22}$, which needs 472 million quadwords.

We might use $P = 3 \cdot 5 \cdot 7 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23 = 111546435$, for which $\phi(P) = 36495360 = 2^{13} \cdot 3^4 \cdot 5 \cdot 11$. We choose $s_2 \mid \phi(P)$ so that s_2 is close to $\phi(P)/(\ell_{\max}/2) \approx 8.7$, i.e., $s_2 = 9$ and $s_1 = 4055040$, giving $s_1/\ell_{\max} \approx 0.48$.

We can do 9 convolutions, one for each $k_2 \in S_2$. We will be able to find $p \mid N$ if $b_1^q \equiv 1 \pmod{p}$ where q satisfies (2.7) with $m < \ell_{\max} - s_1 = 4333568$. As described in Section 2.6, the effective value of B_2 will be about $9.66 \cdot 10^{14}$.

2.11 Opportunities for Parallelization

Modern PC's are multi-core, typically with 2–4 CPUs (cores) and a shared memory. When running on such systems, it is desirable to utilize multiple cores.

While building $h(X)$ and $g(X)$ in Section 2.9, each core can process a contiguous block of subscripts. Use the explicit formulas to compute r^{-j^2} or g_i for the first two elements of a block, and the recurrences elsewhere.

If convolutions use NTT's and the number of processors divides the number of primes, then allocate the primes evenly across the processors. The (MDFT) and (MHDFT) buffers in Section 2.10 can have separate subbuffers for each prime. On NUMA architectures, the memory for each subbuffer should be allocated locally to the processor that will process it. Accesses to remote memory occur only when converting the h_j and g_i to residues modulo small primes, and when reconstructing the coefficients of $g(x)h(x)$ with the CRT.

2.12 Our Implementation

Our implementation is based on GMP-ECM, an implementation of P–1, P+1, and the Elliptic Curve Method for integer factorization. It uses the GMP library [49] for arbitrary precision arithmetic. The code for Stage 1 of P–1 and P+1 is unchanged; the code for the new Stage 2 has been written from scratch and has replaced the previous implementation [103] which used product trees of cost $O\big(n(\log n)^2\big)$ modular multiplications for building polynomials of degree n

and a variant of Montgomery's POLYEVAL [67] algorithm for multipoint evaluation which has cost $O\big(n(\log n)^2\big)$ modular multiplications and $O(n \log n)$ memory. The practical limit for B_2 was between 10^{14} and 10^{15}.

GMP-ECM includes modular arithmetic routines, using, e.g., Montgomery's REDC [64], or fast reduction modulo a number of the form $2^n \pm 1$. It also includes routines for polynomial arithmetic, in particular convolution products. One algorithm available for this purpose is a small prime NTT/CRT, using the "Explicit CRT" [12] variant which speeds reduction modulo N after the CRT step but requires 2 or 3 additional small primes. Its current implementation allows only for power-of-two transform lengths. Another is Kronecker-Schönhage's segmentation method [103], which is faster than the NTT if the modulus is large and the convolution length is comparatively small, and works for any convolution length. Its main disadvantage is significantly higher memory use, reducing the possible convolution length.

On a 2.4 GHz Opteron with 8 GB memory, P–1 Stage 2 on a 230-digit composite cofactor of $12^{254} + 1$ with $B_2 = 1.2 \cdot 10^{15}$, using the NTT with 27 primes for the convolution, can use $P = 64579515$, $\ell_{\max} = 2^{24}$, $s_1 = 7434240$, $s_2 = 3$ and takes 1738 seconds while P+1 Stage 2 takes 3356 seconds. Using multi-threading to use both CPUs on the same machine, P–1 Stage 2 with the same parameters takes 1753 seconds CPU and 941 seconds elapsed time while P+1 takes 3390 seconds CPU and 2323 seconds elapsed time. For comparison, the previous implementation of P–1 Stage 2 in GMP-ECM [103] needs to use a polynomial $F(X)$ of degree 1013760 and 80 blocks for $B_2 = 10^{15}$ and takes 34080 seconds on one CPU of the same machine.

On a 2.6 GHz Opteron with 8 cores and 32 GB of memory, a multi-threaded P–1 Stage 2 on the same input number with the same parameters takes 1661 seconds CPU and 269 seconds elapsed time, while P+1 takes 3409 seconds CPU and 642 seconds elapsed time. With $B_2 = 1.34 \cdot 10^{16}$, $P = 198843645$, $\ell_{\max} = 2^{26}$, $s_1 = 33177600$, $s_2 = 2$, P–1 Stage 2 takes 5483 seconds CPU and 922 elapsed time while P+1 takes 10089 seconds CPU and 2192 seconds elapsed time.

2.13 Some Results

We ran at least one of P±1 on over 1500 composite cofactors, including:

(a) Richard Brent's tables with $b^n \pm 1$ factorizations for $13 \le b \le 99$;

(b) Fibonacci and Lucas numbers F_n and L_n with $n < 2000$, or $n < 10000$ and cofactor size $< 10^{300}$;

(c) Cunningham cofactors of $12^n \pm 1$ with $n < 300$;

(d) Cunningham cofactors of 300 digits and larger.

The B_1 and B_2 values varied, with $B_1 = 10^{11}$ and $B_2 = 10^{16}$ being typical. Table 2.2 has new large prime factors p and the largest factors of the corresponding $p \pm 1$.

The 52-digit factor of $47^{146} + 1$ and the 60-digit factor of L_{2366} each set a new record for the P+1 factoring algorithm upon their discovery. The previous record was a 48-digit factor of L_{1849}, found by the author in March 2003.

The 53-digit factor of $24^{142} + 1$ has $q = 12750725834505143$, a 17-digit prime. To our knowledge, this is the largest prime in the group order associated with any factor found by the P–1, P+1 or Elliptic Curve methods of factorization.

The largest q reported in Table 2 of [74] is $q = 6496749983$ (10 digits), for a 19-digit factor p of $2^{895} + 1$. That table includes a 34-digit factor of the Fibonacci number F_{575}, which was the P–1 record in 1989.

Input Method	Factor p found Largest factors of $p \pm 1$	Size
$73^{109} - 1$	76227040047863715568322367158695720006439518152299	c191
P–1	$12491 \cdot 37987 \cdot 156059 \cdot 2244509 \cdot 462832247372839$	p50
$68^{118} + 1$	7506686348037740621097710183200476580505073749325089*	c151
P–1	$22807 \cdot 480587 \cdot 14334767 \cdot 89294369 \cdot 4649376803 \cdot 5380282339$	p52
$24^{142} + 1$	20489047427450579051989683686453370154126820104624537	c183
P–1	$4959947 \cdot 7216081 \cdot 16915319 \cdot 17286223 \cdot 12750725834505143$	p53
$47^{146} + 1$	7986478866035822988220162978874631335274957495008401	c235
P+1	$20540953 \cdot 56417663 \cdot 1231471331 \cdot 1632221953 \cdot 843497917739$	p52
L_{2366}	725516237739635905037132916171116034279215026146021770250523	c290
P+1	$932677 \cdot 62754121 \cdot 19882583417 \cdot 751245344783 \cdot 483576618980159$	p60

* = Found during Stage 1

Table 2.2: Large P±1 factors found

Table 2.3: Timing for $24^{142} + 1$ factorization

Operation	Minutes (per CPU)	Parameters
Compute f	22	$P = 198843645$
Compute h	2	$\ell_{\max} = 2^{26}$
Compute DCT–I(h)	8	$s_1 = 33177600$
Compute all g_i	6 (twice)	$s_2 = 1$
Compute $g \times h$	17 (twice)	$m_1 = 246$
Test for non-trivial gcd	2 (twice)	
Total	$32 + 2 \cdot 25 = 82$	

The largest P–1 factor reported in [103, pp. 538–539] is a 58-digit factor of $2^{2098} + 1$ with $q = 9909876848747$ (13 digits). Site **http://www.loria.fr/~zimmerma/records/Pminus1.html** has other records, including a 66-digit factor of $960^{119} - 1$ found by P–1 for which $q = 2110402817$ (only ten digits).

The p53 of $24^{142} + 1$ in Table 2.2 used $B_1 = 10^{11}$ at Montgomery's site. Stage 1 took 44 hours using a 2200 MHz AMD Athlon processor in 32-bit mode.

Stage 2 ran on an 8-core, 32 GB Grid5000 cluster at the author's site. Table 2.3 shows where the time went. The overall Stage 2 time is $8 \cdot 82 = 656$ minutes, about 25% of the Stage 1 CPU time.

Chapter 3

The Number Field Sieve

3.1 Introduction

The Number Field Sieve (NFS) is the best currently known general factorization algorithm. As opposed to special factorization algorithms such as those described in Chapter 2 or ECM [62] whose run time depend strongly on the size (or other properties) of the prime factor p of N we hope to find and much less on the size of N, the run time of general factorization algorithms does not depend on particular properties of the prime factors of an input number, but only on the size of N. This makes the NFS the method of choice for factoring “hard” integers, i.e., integers that contain no prime factors that are small or have other properties that would make them easy to find by special factoring algorithms.

Factoring the modulus of the public key is one possible attack on the RSA encryption system [85] and, for keys that are not weak keys and where the secret key cannot be obtained directly, it is the most efficient known attack on RSA. A variant of NFS [48] can be used to solve the discrete logarithm problem in $\mathbb{F}_p^*$ and so is one possible attack on the Diffie-Hellman (DH) key exchange algorithm [36]. Therefore great effort has been made over the last twenty years to improve NFS in order to estimate the minimum modulus size for RSA and DH keys that are out of range for NFS with available computing resources. This size changed considerably over the years, see Section 3.3 for an overview of NFS integer factoring records.

NFS is a successor of the Quadratic Sieve, which was the best general factoring algorithm before the advent of NFS and has conjectured asymptotic running time in $L_N[1/2, 1]$ for factoring N, where the L-function is defined as

$$L_x[t, c] = e^{(c+o(1)) \log(x)^t \log\log(x)^{1-t}}.$$

The Number Field Sieve achieves conjectured complexity

$$L_N[1/3, c],$$

where the constant c is $(32/9)^{1/3} \approx 1.526$ or $(64/9)^{1/3} \approx 1.923$, depending on the variant of NFS, see Section 3.2.1.

This chapter gives a brief overview of the Number Field Sieve: how it relates to its predecessor, the Quadratic Sieve, and a short description of the different steps performed in an NFS factorization, to provide context for the following Chapters which focus on the problem of co-factorization in the sieving step of NFS.

3.1.1 The Quadratic Sieve

The Quadratic sieve was developed by Carl Pomerance [84], based on the unpublished (but described in [84]) Linear Sieve by Schröppel. Silverman [92] implements a Multiple Polynomial Quadratic Sieve variant based on ideas of Montgomery (which are also mentioned in [84]), and Contini [27] reduces the cost of polynomial changes with the Self-Initialising Quadratic Sieve. We give a brief description of the QS to introduce a few concepts common to QS and NFS, and don't consider the mentioned improvements to QS here to keep the description simple.

The basic idea of the Quadratic Sieve is to collect relations $y = x^2 - N$ where x ranges over an interval of integers close to $\sqrt{N}$, and y will consequently take integer values bounded by $N^{1/2+\epsilon}$. Values of y which are "smooth," i.e., that have no prime factor greater than some smoothness bound $\mathfrak{B}$, are completely factored and stored together with the associated x-value. Once enough such relations are found, a subset of these (x, y)-pairs such that the product of y-values in this subset forms an integer square can be determined by linear algebra: an integer is a square if all its prime factors appear in even power, and the exponent vector of the canonical factorization of a product is the sum of the exponent vectors of the multiplier and the multiplicand. Thus we can look for kernel vectors of a matrix over $\mathbb{F}_2$ to find a product of y-values whose exponent vector has all components $0 \pmod 2$, i.e., which is an integer square. If we have at least as many relations as primes appear among the factorizations of the y-values, the linear system can be solved. The product of the so selected y-values and the product of the associated x^2-values thus form congruent squares,

$$X^2 \equiv Y^2 \pmod N \tag{3.1}$$

where X and Y can be obtained from the stored x and factored y-values. If $X \not\equiv \pm Y \pmod N$, then $\gcd(X - Y, N)$ finds a proper factor of N. For composite N that are not pure powers, (3.1) has at least 4 solutions for any $X \perp N$; if the factoring algorithm produces one of these solutions at random, then the probability of having a non-trivial factorization is at least $1/2$.

The great advantage of QS over earlier general factoring algorithms such as the Continued Fraction Method [59] is that values of x where y is divisible by a prime p form arithmetic progressions of common difference p, i.e., if $f(x)$ is a polynomial with integer coefficients and we choose an x such that $p \mid f(x)$, then $p \mid f(x+kp)$ for all integers k. This greatly simplifies the task of factoring the y-values in the QS: given $y = f(x)$ for n consecutive values of x, instead of trying all candidate prime divisors $2 \le p < \mathfrak{B}$ for each y-value individually with cost in $O(n\pi(\mathfrak{B}))$, we determine the roots $f(x_{1|2,p}) \equiv 0 \pmod p$ for each $2 \le p < \mathfrak{B}$ where $\left(\frac{\mathrm{disc}(f)}{p}\right) \ne -1$ with cost $O(\pi(\mathfrak{B}))$ and can divide out this prime p from all $y = f(x)$-values with $x \equiv x_{1|2,p} \pmod p$ ("sieve," similar as in the Sieve of Eratosthenes) for a cost approximately $O(n/p)$, which gives a total cost for sieving all primes p of only $O(n \log\log(\mathfrak{B}) + \pi(\mathfrak{B}))$.

3.1.2 NFS: a First Experiment

The basic idea of NFS, still in infancy, was described by Pollard [82] who demonstrated how to factor the seventh Fermat number $N = F_7 = 2^{128} + 1$ by use of cubic integers. Underlying the idea is the fact that a small multiple of N can be expressed easily by a monic polynomial of degree 3 as $2N = 2^{129} + 2 = f(x) = x^3 + 2$ for $x = 2^{43}$, i.e., 2^{43} is a root of $f(x) \bmod N$. A complex root α of $f(x)$ defines an algebraic number field $\mathbb{Q}[\alpha]$ which contains the ring $\mathbb{Z}[\alpha]$ of elements $a + b\alpha + c\alpha^2$, $a, b, c \in \mathbb{Z}$. A natural homomorphism $\phi : \mathbb{Q}[\alpha] \to \mathbb{Z}/N\mathbb{Z}$ exists by $\alpha \mapsto 2^{43} \pmod N$.

Pollard now looks for a non-empty set $S \subset \mathbb{Z}^2$ such that

$$\gamma^2 = \prod_{(a,b)\in S} (a - b\alpha) \tag{3.2}$$

is the square of some element $\gamma \in \mathbb{Z}[\alpha]$, and

$$g^2 = \prod_{(a,b)\in S} (a - b2^{43}) \tag{3.3}$$

is the square of an integer g. With these, he computes γ and uses the homomorphism ϕ to obtain $\phi(\gamma) \in \mathbb{Z}/N\mathbb{Z}$ such that $\phi(\gamma)^2 \equiv g^2 \pmod{N}$ but hopefully $\phi(\gamma) \not\equiv \pm g \pmod{N}$, since then $\gcd(\phi(\gamma) - g, N)$ produces a non-trivial factor of N.

The search for a suitable set S works similarly as in the Quadratic Sieve: he collects relations, (a, b)-pairs with $a, b \in \mathbb{Z}$ and $a \perp b$, from a small search region $|a| \le 4800$, $1 \le b \le 2000$, where the norm of $a - b\alpha$ (denoted $N(a - b\alpha)$, a rational integer) and of $a + b2^{43}$ both factor into primes not exceeding $\mathfrak{B} = 3571$, plus at most one prime less than 10000 in each $a + b2^{43}$ value. This is facilitated by using the fact that both $N(a - b\alpha) = a^3 + 4b^3$ and $a - bM$ are homogeneous polynomials in a and b. For each fixed value of b, the two polynomials can be sieved over a range of a-values. In this example $\mathbb{Z}[\alpha]$ is a unique factorization domain, is equal to the ring of integers of $\mathbb{Q}[\alpha]$, and has unit group of rank 1 where the principal unit $1+\alpha$ is easy to find. This way each $a - b\alpha$ of smooth norm can be readily factored over a small set of prime elements of $\mathbb{Z}[\alpha]$, a sign and a power of the principal unit. Given sufficiently many relations with completely factored $a - b\alpha$ (in $\mathbb{Z}[\alpha]$) and $a - b2^{43}$ (in $\mathbb{Z}$), he constructs a set S satisfying (3.2) and (3.3), again by use of linear algebra to ensure that in each of the two products all primes and units occur in even exponent. Since the explicit factorization of each $a - b\alpha$ and $a - b2^{43}$ into powers of primes and units is known, the square root can be taken by dividing exponents by 2 and computing the product.

Both QS and NFS look for values of polynomials that are “smooth,” i.e., that contain no prime factors greater than some smoothness limit $\mathfrak{B}$. In case of (our simplified) QS, we choose integers x close to $\sqrt{N}$ and look for smooth $y = x^2 - N$ where the y-values are roughly as large as $\sqrt{N}$; for the NFS example, we look for pairs (a, b) where two polynomials $F(a, b) = a^3 + 4b^3$ and $G(a, b) = a + b2^{43}$ are simultaneously smooth. The reason why NFS is asymptotically faster than QS, even though for each relation it requires two values both being smooth instead of only one, is that the values are *smaller*. In Pollard’s example, the values of $G(a, b)$ are of size roughly $N^{1/3}$ and the values of $F(a, b)$ are smaller still. The probability of an integer n being smooth to a given bound decreases rapidly with the size of n, and even though we have two values of size roughly $N^{1/3}$, for large enough N, assuming independent smoothness probability, they are more likely both smooth than a single value around $\sqrt{N}$.

At the time of Pollard’s experiment, it was not at all clear whether the idea could be extended to numbers that are not of such a simple form as $2^{128} + 1$, or where the relevant ring $\mathbb{Z}[\alpha]$ in the number field is not as “well-behaved,” and if it could, whether this algorithm would be faster than the Quadratic Sieve for input sizes of interest. The answer to both turned out to be an enthusiastic “yes,” and the NFS currently stands unchallenged for factoring hard integers of more than approximately 100 decimal digits.

3.2 Overview of NFS

In this section we briefly summarize the Number Field Sieve. It requires two distinct polynomials

$$f_1(x) = \sum_{i=0}^{d_1} f_{1,i} x^i \qquad \text{and} \qquad f_2(x) = \sum_{i=0}^{d_2} f_{2,i} x^i \tag{3.4}$$

of degree d_1 and d_2, respectively, with $f_{1,i},\ f_{2,i} \in \mathbb{Z}$, each polynomial irreducible over $\mathbb{Q}$, of content 1 and with a known common root M modulo N, the number to be factored:

$$f_1(M) \equiv f_2(M) \equiv 0 \pmod{N}. \tag{3.5}$$

The homogeneous forms of these polynomials are

$$F_1(a,b) = f_1\left(\frac{a}{b}\right) b^{d_1} \qquad \text{and} \qquad F_2(a,b) = f_2\left(\frac{a}{b}\right) b^{d_2}. \tag{3.6}$$

Let α_1 be a complex root of $f_1(x)$, then $\mathbb{Q}[\alpha_1]$ defines a number field which contains the ring $\mathbb{Z}[\alpha_1]$, however this ring is not integral if $f_1(x)$ is non-monic and even if it is, generally is neither the full ring of integers of $\mathbb{Q}[\alpha_1]$, nor has unique factorization. Since M is a root of $f_1(x) \pmod{N}$, a natural homomorphism $\phi_1 : \mathbb{Q}[\alpha_1] \to \mathbb{Z}/N\mathbb{Z}$ exists by $\alpha_1 \mapsto M \pmod{N}$. Similarly for the second polynomial.

The goal of NFS is to construct $\gamma_1 \in \mathbb{Z}[\alpha_1]$ and $\gamma_2 \in \mathbb{Z}[\alpha_2]$ with $\phi_1(\gamma_1^2) \equiv \phi_2(\gamma_2^2) \pmod{N}$, since then $X = \phi_1(\gamma_1)$ and $Y = \phi_2(\gamma_2)$ satisfy $X^2 \equiv Y^2 \pmod{N}$ and so, if $X \not\equiv \pm Y \pmod{N}$ holds, $\gcd(X - Y, N)$ reveals a proper factor of N. We achieve this by constructing

$$\gamma_1^2 = \prod_{(a,b) \in S} (a - b\alpha_1) \qquad \text{and} \tag{3.7}$$

$$\gamma_2^2 = \prod_{(a,b) \in S} (a - b\alpha_2) \tag{3.8}$$

with a suitably chosen set S such that (3.7) and (3.8) are a square in $\mathbb{Z}[\alpha_1]$ and $\mathbb{Z}[\alpha_2]$, respectively. Since $\phi_1(a - b\alpha_1) \equiv a - bM \equiv \phi_2(a - b\alpha_2) \pmod{N}$, the images of (3.7) and (3.8) are congruent modulo N as required.

In a number field $K = \mathbb{Q}[x]/f(x)\mathbb{Q}[x]$ of degree d with $\alpha = \bar{x}$ the norm of an element $\omega(x) = \sum_{0 \le i < d} c_i x^i$ is defined as $N(\omega) = \prod_{1 \le j \le d} \omega(\alpha_j)$ where the α_j are the d complex roots of $f(x)$. For $\omega \in \mathbb{Z}[\alpha]$ the norm is a rational integer if $f(x)$ is monic, (otherwise the norm times the leading coefficient of $f(x)$ is an integer, for simplicity we assume the monic case) and for elements $a - b\alpha$ we have simply $N(a - b\alpha) = b^d f(a/b) = F(a,b)$, where $F(a,b)$ is the homogeneous form of $f(x)$. The norm is multiplicative, i.e., $N(\omega\theta) = N(\omega)N(\theta)$ for any $\omega,\ \theta \in K$, implying that $N(\omega^2)$ is an integer square for any $\omega \in \mathbb{Z}[\alpha]$.

To construct S, we look for relations (a,b), $a \perp b$, where $F_1(a,b)$ is $\mathfrak{B}_1$-smooth and $F_2(a,b)$ is $\mathfrak{B}_2$-smooth. By considering the norms, we see that $\prod_{(a,b) \in S} F_1(a,b)$ must be a square in $\mathbb{Z}$ for (3.7) to be a square in $\mathbb{Z}[\alpha_1]$ (likewise for the second polynomial in the following), but this condition is generally not sufficient, as distinct primes in $\mathbb{Z}[\alpha_1]$ may have equal norm. Therefore, instead of considering only the factorization of the norm $F(a,b)$ into rational primes, we consider the factorization of the ideal generated by $a - b\alpha_1$ in $\mathbb{Z}[\alpha_1]$ into prime ideals. Each prime ideal that occurs as a divisor of $(a - b\alpha_1)$ is uniquely identified by (p, r) where p is a prime factor of $F_1(a,b)$ and is the norm of the prime ideal, and $r = a \cdot b^{-1} \pmod{p}$ is the corresponding root of $f_1(x) \pmod{p}$. That is, we do not consider only the prime factors of $N(a - b\alpha_1)$, but further distinguish them by which of the up to d_1 possible roots of $f(x) \pmod p$ they correspond to. The set S is then chosen such that all prime ideals occur in even exponent in both (3.7) and (3.8). This is still not quite sufficient for ensuring that these products are squares in their respective ring, as the unit group and class group parts might not be squares, but this problem is elegantly circumvented by use of quadratic characters, described in Section 3.2.4.

Very frequently the second polynomial is chosen to be linear in which case $\mathbb{Q}[\alpha_2]$ is simply $\mathbb{Q}$ and factorization of $a - b\alpha$ into prime ideals is equivalent to factorization of $F_2(a,b)$ into rational primes; then the condition that the product $\prod_{(a,b)\in S} F_2(a,b)$ is an integer square is sufficient. In this case everything relating to $\mathbb{Z}[\alpha_2]$ throughout the NFS algorithm is called the "rational side" and anything relating to $\mathbb{Z}[\alpha_1]$ is called the "algebraic side."

In the sieving step we try to find sufficiently many relations (a, b) within a sieving region $|a| \leq A, 0 < b \leq B$, see Section 3.2.2. The polynomials $f_1(x)$ and $f_2(x)$ are chosen such that the values of $F_1(a, b)$ and $F_2(a, b)$ are likely smooth for a, b in the sieve region; an overview of methods for polynomial selection is given in Section 3.2.1. The relations obtained in the sieving step are processed to remove duplicate relations and to reduce the size of the resulting matrix, see Section 3.2.3. In the linear algebra step we determine a subset S of the relations found during sieving such that (3.7) and (3.8) hold. This involves solving a very large and very sparse homogeneous linear system over $\mathbb{F}_2$; two suitable algorithms are mentioned in Section 3.2.4. The square root step, described in Section 3.2.5, determines γ_1 and γ_2 from γ_1^2 and γ_2^2, respectively, and computes $\gcd(\phi_1(\gamma_1) - \phi_2(\gamma_2), N)$, hoping to find a proper factor of N.

3.2.1 Polynomial Selection

To reduce the cost of the sieving, we try to choose $f_1(x)$ and $f_2(x)$ so as to maximise the expected number of relations found in the sieve region, or conversely to allow the smallest sieve region to produce the required number of relations that lets us construct congruent squares. As mentioned, the probability of the polynomial values being smooth decreases rapidly with their size, so one criteria is that we would like to choose polynomials with small coefficients. A trivial method is to pick a degree $d_1 \approx (3\log(n)/\log(\log(n)))^{1/3}$ and to take $M = \lfloor N^{1/(d_1+1)} \rfloor$. Now we can write N in base-M to obtain the coefficients of $f_1(x)$, and choose $f_2(x) = -x + M$.

Somewhat surprisingly, this trivial idea is asymptotically the best possible (see [20, §3]) in the sense that any improvements due to better polynomial selection are absorbed in the $o(1)$ term of the $L_x[t, c]$ notation. In practice, elaborate methods for finding good polynomials are used which offer a significant speedup over the naïve method.

Early GNFS implementations such as in [11] used basically the base-M method, but included a brute-force search for a good value of M that leads to small polynomial coefficients.

Murphy [75] presents a way of modelling the expected number of relations found by sieving two given polynomials over a sieve region with given smoothness bounds, and shows how to improve the base-M method for selecting polynomials that enjoy not only small average value over the sieve region, but also have favourable root properties. The root properties model the average contribution of small prime factors to the size of polynomial values. For polynomials which have many roots modulo small primes, this contribution is greater, and these polynomial values are more likely smooth than when few small prime divisors are present.

Kleinjung [54] extends Murphy's work by allowing a common root M of the two polynomials that is not an integer close to N^{d_1+1}, but a rational number $M = k/l$. This leads to a linear polynomial $g(x) = lx - k$ and greatly extends the search space of suitable M values which allows picking one that leads to particularly small polynomial values. He further improves techniques to generate polynomials with good root properties.

For integers N of no special form, a suitable polynomial $f(x)$ is found by the above methods and the second polynomial $g(x)$ is chosen to be linear; then the coefficients of both polynomials are bounded by $O\big(N^{1/d_1}\big)$. With this bound, the constant c in (3.1) is conjectured to be $(64/9)^{1/3}$.

For integers of a simple algebraic form such as $F_7 = 2^{128} + 1$, a polynomial with very small coefficients can easily be found manually. For this type of numbers, we can take the size of the coefficients of $f(x)$ to be bounded by a constant, which reduces the constant c to conjecturally $(32/9)^{1/3}$.

3.2.2 Sieving

The task of the sieving step is to identify many relations, (a, b)-pairs with $a \perp b$ such that $F_1(a, b)$ and $F_2(a, b)$ are both smooth. The smoothness criterion determines the sieving parameters, so we choose smoothness bounds $\mathfrak{B}_1$ and $\mathfrak{B}_2$, a typical order of magnitude being 10^7 for a factorization of 150-digit numbers, and consider $F_1(a, b)$ smooth if no prime exceeding $\mathfrak{B}_1$ divides it (similarly for $F_2(a, b)$). In practice, a large prime variant is used as it greatly increases the number of relations found at little extra cost. We add large prime bounds $\mathfrak{L}_1$ and $\mathfrak{L}_2$, usually about 100 times the respective factor base bound, and consider $F_1(a, b)$ smooth if all its prime factor do not exceed $\mathfrak{B}_1$ except for at most k_1 prime factors up to $\mathfrak{L}_1$, similarly for $F_2(a, b)$.

To find (a, b)-pairs where $F_1(a, b)$ and $F_2(a, b)$ are smooth, a sieving method is used, using the fact that $F_1(a, b)$-values (and likewise for $F_2(a, b)$ in the following) that are divisible by a prime p form a regular pattern in $\mathbb{Z}^2$. Let r be a root of $f_1(x) \pmod p$, then the (a, b)-pairs where $p \mid F_1(a, b)$ are exactly those where $a \equiv br \pmod p$. (The homogeneous form $F_1(a, b)$ may have roots with $b \equiv 0 \pmod p$, namely for p that divide the leading coefficient of $f_1(x)$; such roots correspond to roots at infinity of $f_1(x) \pmod p$ and are not considered here.)

The sieving process starts by building a factor base: a list of primes $p \le \mathfrak{B}_1$ and for each the roots of $f_1(x) \pmod p$, likewise for $f_2(x)$. For the rational side (assuming $f_2(x)$ is linear), this process is simple enough to do it at the start of the siever program, for the algebraic side the factor base is commonly computed once and stored in a file.

The sieving is performed over a sieve region $-A \le a < A$, $0 < b \le B$ which is chosen large enough that one may expect to find sufficiently large set of relations so that the linear algebra phase can find a subset S that satisfies (3.7) and (3.8). In principle, sieving can end when the number of relations (forming the variables of the linear system over $\mathbb{F}_2$, each relation can be included in S or not) exceeds the number of prime ideals that occur among the relations (forming the equations of the linear system, the sum of exponents of each prime ideal must be even in the solution), since then the resulting matrix has at least one non-zero kernel vector. In practice one wants a healthy amount of excess (the difference of the number of relations and of the prime ideals among them), as this allows reducing the size of the matrix and several kernel vectors may need to be tried to find a non-trivial factorization. A ratio of 10% more relations than ideals is a good rule-of-thumb.

To speed up the sieving process, it is not performed on the values of $F_1(a, b)$ and $F_2(a, b)$ themselves. Instead, for each (a, b) in the sieve region, a rounded base-l logarithm $\lfloor \log_l(F_1(a, b)) \rceil$ is stored in an array, and each prime in the factor base that divides $F_1(a, b)$ ("hits at (a,b)") subtracts $\lfloor \log_l(p) \rceil$ from the corresponding array location. This replaces an integer division by a simpler integer subtraction per hit; the logarithm base l is commonly chosen such that the rounded logarithms fit into one byte to conserve memory. The pairs (a, b) where $F_1(a, b)$ is smooth will have a small value remaining in their array entry; entries where the remaining sieve value is below a threshold are remembered and the sieving process is repeated for $F_2(a, b)$. The repeated subtraction of rounded logarithms accumulates some rounding error which needs to be taken into account when choosing the sieve report threshold. Those (a, b) where both sieving passes left values below the respective threshold are good candidates for being proper relations and are examined more closely: the corresponding $F_{1,2}(a, b)$-values are factored exactly over the integers to test whether they satisfy the smoothness criterion. Here, the large prime variants come into play: if one allows large primes, a larger sieve report threshold is chosen accordingly, and the exact factorization of $F_{1,2}(a, b)$ needs to be able to handle composite cofactors after dividing out the factor base primes. The cofactors can consist of between $16 - 30$ decimal digits (and even more for very large factorizations) and prime factors of typically $8 - 10$ decimal digits are permitted as large primes and need to be found efficiently. Suitable methods are described in Chapter 4.

The sieve region $[-A, A] \times [1, B] \bigcap \mathbb{Z}^2$ is usually far too large to be sieved all at once. Instead is it partitioned into smaller pieces which are sieved independently. Two major variants of this sieving process exist: line sieving and lattice sieving.

Line sieving is the simpler one: for each value of b in the sieve region, the line $-A \le a < A$ is treated individually. If it is still too large, the line can be partitioned further. Within a line, for each factor base prime p with root r, the smallest $a_0 \ge -A$ with $a_0 \equiv br \pmod{p}$ is computed, then each $a = a_0 + kp \le A$, $k \in \mathbb{N}$, is hit by the sieving.

Lattice sieving was suggested by Pollard [83] and, while more complicated, performs significantly better and has superseded line sieving in larger factorization efforts. The idea is to consider the lattice in $\mathbb{Z}^2$ where one of the two polynomials, usually $F_1(a, b)$ (although for some SNFS factorizations $F_2(a, b)$ is chosen instead), is known to be divisible by a "special-q" value. If ρ is a root of $f_1(x) \pmod{q}$, then $q \mid F_1(a, b)$ if $a \equiv b\rho \pmod{q}$, and $\begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} q & \rho \\ 0 & 1 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix}$, $i, j \in \mathbb{Z}$, is the lattice of points (a, b) where $ab^{-1} = \rho \pmod{q}$, implying $q \mid F_1(a, b)$. Examining only such a, b where we know a prime factor q of $F_1(a, b)$ significantly increases the chance that $F(a, b)/q$ will be smooth, thus increasing the yield of the sieving. This allows choosing a smaller factor base and sieve region and still obtaining the required number of relations, thus reducing computation time and memory use.

The sieving procedure becomes more complicated, however. The sieve region in the i, j-plane is chosen relatively small, typically (depending on the size of the input number) $-I \le i < I$, $0 \le j < J$ with $I = 2^k$, $J = I/2$, $11 \le k \le 16$. Since each line in the sieve region in the i, j-plane is rather short, line-sieving in this plane is inefficient, since each factor base prime would need to be applied to each line individually, resulting in complexity $O(J\mathfrak{B} + IJ \log\log(\mathfrak{B}))$ per special-q. Since $\mathfrak{B} \gg I$, the $J\mathfrak{B}$ term would dominate (corresponding to finding the first location in the line where the factor base prime hits, yet in any given line, most factor base primes don't hit at all). Instead, for each factor base prime p and each associated root of the polynomial being sieved, the lattice where p hits *in the i, j-plane* is computed which allows enumerating the locations that are hit very efficiently, and the complexity drops to $O(\mathfrak{B} + IJ \log\log(\mathfrak{B}))$. The implied constant for the $\mathfrak{B}$ term is greater than that in the $J\mathfrak{B}$ term seen before, due to the need to transform the roots of factor base primes to the i, j-plane and to compute a reduced lattice basis for each, but this increase is far smaller than the factor J (typically several thousand) that appears if individual j-values were line-sieved.

Franke and Kleinjung [40] give the details of a very efficient lattice sieving algorithm.

3.2.3 Filtering

Before a matrix is built, the relations are processed in the filtering step of NFS, to allow building a matrix that is easier to solve than one containing all relations found in the sieving would be.

The processing is performed in several successive stages.

Duplicates. First of all, duplicate relations are deleted, as those might lead to trivial solutions of the matrix which would produce only trivial factorizations. Duplicate relations occur in large number when lattice sieving is used, as one (a, b)-pair may have a polynomial value $F_1(a, b)$ (assuming the special-q values are chosen for the $f_1(x)$ polynomial) that contains two or more prime factors in the range of primes that are used for special-q values. Even with line sieving, which in theory does not produce duplicates, partial output from interrupted and re-started sieving jobs or accidental sieving of overlapping ranges of b-values often leads to duplicates in practice. In the duplicate removal step, the number of relations decreases while the number of prime ideals occurring among the relations stays constant, so that the excess decreases by 1 for each deleted relation. This effect makes it tricky to predict the number of useful relations found by lattice sieving via sampling the sieving over an estimated range of special-q values: in the small sampling data, the ratio of duplicates to unique relations will be very small, but in the complete data set, it is often as large as 30%.

Singletons. In the next filtering step, relations containing singleton ideals are deleted. Since the goal is finding a subset of relations in whose product every prime ideal occurs to an even power, relations containing a prime ideal in an odd power (usually with exponent 1) that occurs in no other relations cannot be part of any solution, and can be omitted from the matrix. When such relations are deleted, prime ideals that occurred among the deleted and exactly one not deleted relation now become new singletons ("ripple-effect"), and the singleton removal can be repeated until no relations with singleton ideals remain or so few are left that they add negligible size to the matrix. Each deleted relation contains at least one prime ideal that occurs nowhere else among the relations, so that when the number of remaining relations decreases by 1 the number of remaining prime ideals also decreases by at least 1, thus the excess does not decrease in this step. In fact, some relations contain two (very rarely more) singleton ideals, and deleting these actually increases the excess slightly.

Since the prime ideals occurring among the relations must be identified to find singleton ideals, they can be counted as well, and the exact amount of excess can be determined. If there remains any positive excess after the singleton removal step, then a matrix could in principle be built and solved. In practice, one wants a good deal of excess in order to reduce the size of the matrix, and to be able to satisfy some additional constraints on the product of relations from a set of kernel vectors of the matrix.

Connected components. Given a relation set with excess after singleton removal, the data set and hence the matrix size can be shrunk very efficiently by removing connected components (often called "cliques" in this context, although the connected components in question aren't necessarily complete subgraphs). For this, each relation of the data set is considered a node of a graph, and two nodes are connected if and only if they have a prime ideal that occurs in odd power in these two and in no other relation. Deleting any relation in a connected component causes an ideal that formed the vertex to this node to become singleton, and by applying singleton removal, the entire connected component is removed eventually. For each deleted connected component, the excess drops by at most 1 (for the first relation that is deleted, as singleton removal does not reduce excess) and removing large connected components is a very effective way of reducing excess in a way that minimizes the number of ideals among the remaining relations. Removing connected components may cause ideals among the deleted relations to occur in odd power among exactly two of the remaining relations, causing new vertices to appear in the graph. Thus the removal of connected components should be done in several passes, until the excess is reduced to a small positive number, e.g., around 100.

Merging. The remaining relations could be turned into a matrix, using the concatenated exponent vectors modulo 2 of the prime ideals of $\mathbb{Z}[\alpha_1]$ and of $\mathbb{Z}[\alpha_2]$ as equations, and one variable in $\mathbb{F}_2$ per relation (whether to include it in the solution or not) as variables. The resulting linear system is large and very sparse, and with algorithms typically used for solving such sparse large matrices over $\mathbb{F}_2$ such as Block-Lanczos and Block-Wiedemann, the run time depends on the product wM^2, where w is the weight (the number of nonzero entries) and M is the dimension of the matrix. Therefore we may be able to save time in the matrix phase if we can make the matrix smaller, but somewhat more heavy. This is the task of the merging phase.

First observe that if a prime ideal occurs to odd exponent in exactly two relations, then those two relations must either be both included to form a square, or both not included. Hence those two relations can be merged into one by taking their product, i.e., adding their exponent vectors; the shared prime ideal occurs to even exponent in this product and does not occur in any other relation, thus it needs not be considered in the matrix. The resulting vector will have at most two non-zero entries fewer than the two original relations had, so the matrix dimension decreases by 1 and the weight by at least 2, making “2-way merges” always worthwhile.

If a prime ideal occurs in exactly 3 relations, we can form two distinct products from them and use them in place of the three original relations. Again one prime ideal disappears from the matrix, but now we have two relations that each may be about twice as heavy as the original ones, so the total weight might increase. This can be mitigated by choosing the two distinct products whose exponent vectors have smallest weight, but problem of weight increase during merging becomes apparent, and becomes more pressing for higher-way merges.

Cavallar [21] describes an implementation of the filtering step of NFS, including all steps from duplicate removal to merges of relations with prime ideals of frequency up to 19, and examines the effect of merging on the run-time of the Block-Lanczos algorithm for solving the resulting matrix.

3.2.4 Linear Algebra

The goal of the linear algebra step of NFS is, given a set of relations produced by the sieving and filtering steps, to find a subset S such that (3.7) and (3.8) are satisfied. In the filtering we made sure that we can combine the remaining relations into a product where all prime ideals dividing the ideal generated by γ_1 and γ_2, respectively, occur in even power, and this is a necessary condition for being a square in $\mathbb{Z}[\alpha_1]$ and $\mathbb{Z}[\alpha_2]$, respectively, but it is generally not sufficient.

In spite of this, we first look at the problem of finding solutions which ensure that all prime ideals occur in even power; given a small number of such solutions, the remaining conditions can then be satisfied relatively easily.

From the factorization of each ideal $(a - b\alpha_1)$ in $\mathbb{Z}[\alpha_1]$ and $(a - b\alpha_2)$ in $\mathbb{Z}[\alpha_2]$ into prime ideals, we take the exponent modulo 2 of each prime ideal to form column vectors over $\mathbb{F}_2$. This forms a matrix that contains a column for each (merged) relation and a row for each prime ideal that occurs among the relations. This produces a large and very sparse matrix, since each relation contains only a small number of prime ideals. For factorizations of general numbers of around 150 digits, the matrix size is of the order of a few million rows and columns, see Table 3.1. The number of entries per column is typically between 50 and 150.

To solve such large, sparse linear systems, the venerable Gaussian elimination algorithm is ill-suited. As rows are added during the elimination, the sparseness of the matrix is lost, and a dense representation of the matrix would have to be used which is not feasible for matrices of dimension well above 10^6. Suitable algorithms are iterative methods such as Block-Wiedemann [31] and Block-Lanczos [69] that both find a set of kernel vectors by performing only (possibly transposed) matrix-vector products, leaving the matrix unaltered.

Choosing a set S in this way with resulting products Γ_i of relations $a - b\alpha_i$ for each $i = 1, 2$ is generally not sufficient to ensure that Γ_i is the square of an element of $\gamma_i \in \mathbb{Z}[\alpha_i]$. The possible obstructions are given in [20, §6].

Fortunately, this problem can be solved easily with an elegant trick. For each relation (a, b) we determine the quadratic character $\chi_p(a - b\alpha_1)$ in $\mathbb{F}_p[x]/f(x)$ for a prime p that does not divide any $F(a, b)$ among our relations and likewise for the second polynomial $g(x)$ (unless it is linear, in which case attention must be given only to the unit -1 of $\mathbb{Z}$ if the values of $G(a, b)$ can be negative.) In any set S such that the product (3.7) is a square, $1 = \prod_{(a,b)\in S} \chi_p(a - b\alpha_1)$. Thus we can use quadratic characters as a "probabilistic squareness test" of sorts; by doing sufficiently many and choosing S such that they all indicate a square product, we can be quite certain to obtain a proper square in $\mathbb{Z}[\alpha]$. Thus we can use $\log_{-1}(\chi_p(a - b\alpha))$, which is 0 if the character indicates a square and is 1 otherwise, and use it as an additional equation in the linear system. Solving the homogeneous system ensures that the sum of logarithms of each character is 0, i.e., their product is 1, indicating that the solution is a square. Assuming that a random non-square element ω of $\mathbb{Z}[\alpha]$ has $\chi_p(\omega) = 1$ with probability 1/2 and that the probabilities are independent for χ_p with different p, each character added to the linear system reduces the number of non-square solutions in the kernel of the matrix by 1/2. By adding a few more characters than the rank of the unit group plus $h_{\mathbb{Z}[\alpha]}$, the class number of $\mathbb{Z}[\alpha]$, we can be reasonably sure that the kernel of the matrix contains mostly vectors that indicate sets S that satisfy (3.7) and (3.8). Computing the class number is itself a non-trivial problem, and in practice one usually chooses a constant number of characters that "ought to do it," the actual number varying between implementations, but usually being between 32 and 64, rarely more.

3.2.5 Square Root

The linear algebra step produced a set $S \subset \mathbb{Z}^2$ that satisfies (3.7) and (3.8). We now need to take the square roots of $\gamma_{1,2}^2$ in their respective number fields so that we can finally take $\gcd(\phi(\gamma_1) - \phi(\gamma_2), N)$, hoping to find a non-trivial factor of N.

In Pollard's first experiment, computing the square root was easy, since $\mathbb{Z}[\sqrt{-2}]$ enjoys unique factorization so that he could factor each $a - b\sqrt{-2}$ into prime elements. Given their factorization, he obtains the factorization of γ_1^2 and computes γ_1 simply by halving the exponent of each prime element. With the second polynomial linear, taking a square root on the rational side is simply taking a square root of a rational integer.

A simple approach that was considered impractical in the early days of NFS due to insufficient memory in the available computing hardware has recently been revived. The idea is to use isomorphism of $\mathbb{Z}[\alpha_1]$ with $\mathbb{Z}[x]/f(x)\mathbb{Z}[x]$ and to compute

$$\Gamma(x) = \left(\prod_{(a,b)\in S} (a - bx) \right) \bmod f(x),$$

where the coefficients of the resulting polynomial will be very large, and fast algorithms for integer multiplication such as the one described in Chapter 1 must be used; with a naïve $O\big(n^2\big)$

multiplication algorithm, this simple square root step has time complexity similar to the sieving and linear algebra steps, but with fast multiplication techniques its time complexity is asymptotically negligible and practically satisfactory. Given $\Gamma(x)$, we would like to obtain $\gamma(x) = \sqrt{\Gamma(x)}$ in $\mathbb{Z}[x]/f(x)\mathbb{Z}[x]$, which we can do by computing $\sqrt{\Gamma(x)}$ in $\mathbb{F}_p/f(x)\mathbb{F}_p$ for a small prime p such that $f(x)$ is irreducible over $\mathbb{F}_p$, and using Hensel lifting of the result to a power p^k such that p^k is greater than the coefficients of $\gamma(x)$.

Couveignes [32] presents an algorithm that performs the square root modulo different smaller prime powers $p_i^{k_i}$ and constructs the solution via the Chinese Remainder Theorem. It can operate with less memory than the simple method, but works only for polynomials $f(x)$ of odd degree and has fallen out of use.

Montgomery [68] proposes an algorithm based on lattice reduction for computing square (and higher) roots of products such as (3.7) and (3.8) in algebraic number fields; Nguyen [76] implements it. It is fast and uses little memory, but is far more complex to implement than the two methods previously mentioned.

3.3 NFS Factoring Records

The maximum size of numbers that can be factored with NFS increased considerably over the 20 years since its inception, both due to algorithmic improvements and computers with faster CPUs and larger main memory becoming available. Table 3.1 lists factorization records for the General Number Field Sieve and Table 3.2 lists records for the Special Number Field Sieve. For SNFS, the difficulty is listed, which is defined as the base-10 logarithm of the resultant of the two polynomials used. The number $2,1642\text{M}$ refers to the algebraic factor $2^{821} + 2^{411} + 1$ of $2^{1642} + 1$. Where available we give the number of unique relations obtained, size of matrix and total computation time for the factorization. The latter is often hard to state precisely due to a large number of different computer systems used within one factorization. Where a formal publication of the result exists, it is cited. In the other cases, the factorization is usually announced by a message to a number theory mailing list; such messages of record factorizations are collected by Contini [26].

Number	Digits	Year	Nr. rel.	Matrix size	Approx. time	By
RSA-130	130	1996	56.515.672	3.516.502	1000 MIPS years	Lenstra et al.
RSA-140	140	1999	56.605.498	4.704.451	2000 MIPS years	te Riele et al. [23]
RSA-155	155	1999	85.534.738	6.711.336	8000 MIPS years	te Riele et al. [24]
$2^{953}+1$	158	2002	254.033.792	5.792.705	4 GHz-years	Franke et al.
RSA-160	160	2003	289.145.711	5.037.191		Franke et al.
RSA-576	174	2003				Franke et al.
$11^{281}+1$	176	2005	455.989.949	8.526.749	32 GHz years	Aoki et al.
RSA-200	200	2005	2.260.000.000	64.000.000	165 GHz years	Franke et al.
RSA-768	232	2009	47.762.243.404	192.796.550	3700 GHz years	Kleinjung et al. [55]

Table 3.1: Records for the General Number Field Sieve

Number	Difficulty	Year	Nr. rel.	Matrix size	Approx. time	By
$10^{211}-1$	211	1999	56.394.064	4.895.741	11 CPU years	The Cabal
$2^{773}+1$	233.0	2000	85.786.223	6.758.509	57 CPU years	The Cabal
$2^{809}-1$	243.8	2003	343.952.357			Franke et al.
$2, 1642\text{M}$	247.4	2004	438.270.192	7.429.778	22 GHz years	Aoki et al.
$6^{353}-1$	275.5	2006	2.208.187.490	19.591.108	61 GHz years	Aoki et al.
$2^{1039}-1$	312.8	2007	13.822.743.049	66.718.354	400 GHz years	Aoki et al. [2]

Table 3.2: Records for the Special Number Field Sieve

Chapter 4

Factoring small integers with P–1, P+1, and ECM in the Number Field Sieve

4.1 Introduction

The sieving step of the Number Field Sieve [60] identifies integer pairs (a, b) with $a \perp b$ such that the values of two homogeneous polynomials $F_i(a, b)$, $i \in \{1, 2\}$, are both smooth, where the sieving parameters are chosen according to the smoothness criterion. Typically the two polynomials each have a "factor base bound" $\mathfrak{B}_i$, a "large prime bound" $\mathfrak{L}_i$, and a permissible maximum number of large primes k_i associated with them, so that $F_i(a, b)$ is considered smooth if it contains only prime factors up to $\mathfrak{B}_i$ except for up to k_i prime factors greater than $\mathfrak{B}_i$, but none exceeding $\mathfrak{L}_i$. For example, for the factorization of the RSA-155 challenge number [24] (a hard integer of 512-bit) the values $\mathfrak{B} = 2^{24}$, $\mathfrak{L} = 10^9$ and $k = 2$ were used for both polynomials for most of the sieving. Kleinjung [53] gives an estimate for the cost of factoring a 1024-bit RSA key based on the parameters $\mathfrak{B}_1 = 1.1 \cdot 10^9$, $\mathfrak{B}_2 = 3 \cdot 10^8$, and $\mathfrak{L}_1 = \mathfrak{L}_2 = 2^{42}$ with $k_1 = 5$ and $k_2 = 4$.

The contribution of the factor base primes to each polynomial value $F_i(a, b)$ for a set of (a, b) pairs is approximated with a sieving procedure, which estimates roughly what the size of the polynomial values will be after factor base primes have been divided out. If these estimates for a particular (a, b) pair are small enough that both $F_i(a, b)$ values might be smooth, the polynomial values are computed, the factor base primes are divided out, and the two cofactors c_i are tested to see if they satisfy the smoothness criterion.

If only one large prime is permitted, no factoring needs to be carried out at all for the large primes: if $c_i > \mathfrak{L}_i$ for either i, this (a, b) pair is discarded. Since generally $\mathfrak{L}_i < \mathfrak{B}_i^2$ and all prime factors below $\mathfrak{B}_i$ have been removed, a cofactor $c_i \leq \mathfrak{L}_i$ is necessarily prime and need not be factored.

If up to two large primes are permitted, and the cofactor c_i is composite and therefore greater than the large prime bound but below $\mathfrak{L}_i^2$ (or a suitably chosen threshold somewhat less then $\mathfrak{L}_i^2$), it is factored. Since the prime factors in c_i are bounded below by $\mathfrak{B}_i$, and $\mathfrak{L}_i$ is typically less than $\mathfrak{B}_i^{1.5}$, the factors can be expected not to be very much smaller than the square root of the composite number. This way the advantage of special purpose factoring algorithms when small divisors (compared to the composite size) are present does not come into great effect, and general purpose factoring algorithms like SQUFOF or MPQS perform well. In previous implementations of QS and NFS, various algorithms for factoring composites of two prime factors have been used, including SQUFOF and Pollard-Rho in [38, Chapter 3.6], and P–1, SQUFOF, and Pollard-Rho in [22, §3].

If more than two large primes are allowed, the advantage of special purpose factoring algorithms pays off. Given a composite cofactor $c_i > \mathfrak{L}_i^2$, we know that it can be smooth only if it has at least three prime factors, of which at least one must be less than $c_i^{1/3}$. If it has no such small factor, the cofactor is not smooth, and its factorization is not actually required, as this (a, b) pair will be discarded. Hence an early-abort strategy can be employed that uses special-purpose factoring algorithms until either a factor is found and the new cofactor can be tested for smoothness, or after a number of factoring attempts have failed, the cofactor may be assumed to be not smooth with high probability so that this (a, b) pair can be discarded.

Suitable candidates for factoring algorithms for this purpose are the P–1 method, the P+1 method, and the Elliptic Curve Method (ECM). All have in common that a prime factor p is found if the order of some group defined over $\mathbb{F}_p$ is itself smooth. A beneficial property is that for ECM, and to a lesser extent for P+1, parameters can be chosen so that the group order has known small factors, making it more likely smooth. This is particularly effective if the prime factor to be found, and hence the group order, is small, see Chapter 5.

Although the P–1 and P+1 methods by themselves have a relatively poor asymptotic algebraic complexity in $O(\sqrt{p})$ (assuming an asymptotically fast stage 2 as described in Chapter 2), they find surprisingly many primes in far less time, making them useful as a first quick try to eliminate easy cases before ECM begins. In fact, P–1 and P+1 may be viewed as being equivalent to less expensive ECM attempts (but also less effective, due to fewer known factors in the group order).

Another well-known special-purpose factoring algorithm is Pollard's "Rho" method [81] which looks for a collision modulo p in an iterated pseudo-random function modulo N, where p is a prime factor of N we hope to find. When choosing no less than $\sqrt{2\log(2)n} + 0.28$ integers uniformly at random from $[1, n]$, the probability of choosing at least one integer more than once is at least 0.5, well known as the Birthday Paradox which states that in a group of only 23 people, two share a birthday with more than 50% probability. For the Rho method, the expected number of iterations to find a prime factor p is in $O\big(\sqrt{p}\big)$, and in the case of Pollard's original algorithm, the average number of iterations for primes p around 2^{30} is close to $2^{15} \approx \sqrt{p}$, where each iteration takes three modular squarings and a modular multiplication, for an average of ≈ 130000 modular multiplications when counting squarings as multiplications. Brent [14] gives an improved iteration which reduces the number of multiplications by about 25% on average. We will see that a combination of P–1, P+1, and ECM does better on average.

Furthermore, trying the Pollard-Rho method with only a low number of iterations before moving on to other factoring algorithms has a negligible probability of success — among the 4798396 primes in $[2^{30}, 2^{30}+10^8]$, only 3483 are found with at most 1000 iterations of the original Pollard-Rho algorithm with pseudo-random map $x \mapsto x^2+1$ and starting value $x_0 = 2$. For P–1, there are 1087179 primes p in the same range where the largest prime factor of $p-1$ does not exceed 1000, and exponentiating by the product of all primes and prime powers up to B requires only $B/\log(2) + O\Big(\sqrt{B}\Big) \approx 1.44B$ squarings, compared to 4 multiplications per iteration for the original Pollard-Rho algorithm. By using a stage 2 for P–1, its advantage increases further. Figure 4.1 shows the distribution of the largest prime factor of $p-1$ and the required number of Pollard-Rho iterations for finding p, respectively, for primes p in $[2^{30}, 2^{30}+10^8]$. The distribution of the largest prime factor of $p+1$ is identical to that of $p-1$, up to statistical noise. We conclude that unlike P–1 and P+1, the Pollard-Rho method is not suitable for removing "easy pickings."

This chapter describes an implementation of trial division for composites of a few machine words, as well as the P–1, P+1, and Elliptic Curve Method of factorization for small composites of one or two machine words, aimed at factoring cofactors as occur during the sieving phase of

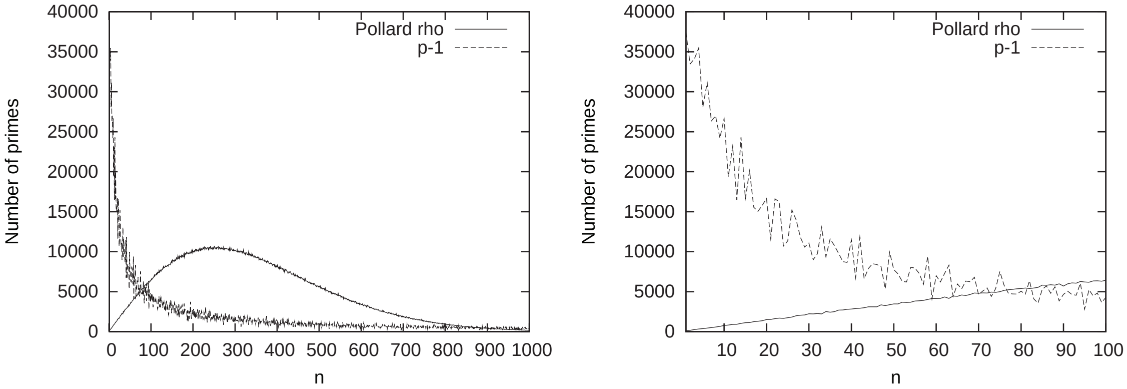

Figure 4.1: Number of primes p in $\left[2^{30}, 2^{30} + 10^8\right]$ where the largest prime factor of $p-1$, respectively the number of Pollard-Rho iterations to find p, is in $[100n, 100n+99]$, $n \in \mathbb{N}$. The left graph shows $0 \le n \le 1000$, the right graph shows a zoom on $0 \le n \le 100$.

the Number Field Sieve. It is part of the CADO [45] implementation of the NFS.

4.2 Trial Division

Before factoring of the non-sieved cofactor of the polynomial values into large primes can commence, the cofactor needs to be determined by dividing out all the factor base primes. For medium size factor base primes, say larger than a few hundred or a few thousand, a sieving technique (”re-sieving”) can be used again that stores the primes when re-sieving hits a location previously marked as ”likely smooth.” For large factor base primes, say larger than a few ten thousand, the number of hits in the sieve area is small enough that the primes can be stored during the initial sieving process itself. For the smallest primes, however, re-sieving is inefficient, and a trial division technique should be used. This section examines a fast trial division routine, based on ideas by Montgomery and Granlund [50] [70], that precomputes several values per candidate prime divisor to speed up the process.

4.2.1 Trial Division Algorithm

Given many composite integers N_i, $0 \le i < n$, we want to determine which primes from some set $P = \{p_j, 0 \le j < k\}$ of small odd primes divide each N_i. We assume $n \gg k$. Each N_i is a multi-word integer of up to $\ell + 1$ words, $N_i = \sum_{j=0}^{\ell} n_{i,j}\beta^j$, where β is the machine word base (e.g., $\beta = 2^{32}$ or $\beta = 2^{64}$) and ℓ is on the order of ”a few,” say $\ell \le 4$. For each prime $p \in P$, we precompute $w_j = \beta^j \bmod p$ for $1 \le j \le \ell$, $p_{\mathrm{inv}} = p^{-1} \pmod{\beta}$ and $p_{\mathrm{lim}} = \left\lfloor \frac{\beta-1}{p} \right\rfloor$.

Consider a particular integer $N = \sum_{j=0}^{\ell} n_j\beta^j$, and a particular prime $p \in P$. The algorithm first does a semi-reduction modulo p to obtain a single-word integer congruent to $N \pmod{p}$, then tests this single-word integer for divisibility by p.

To do so, we compute $r = n_0 + \sum_{j=1}^{\ell} n_j w_j \le (\beta - 1)(\ell(p-1)+1)$. To simplify the next steps, we require $p < \sqrt{\frac{\beta}{\ell}}$. Even for $\beta = 2^{32}$, $\ell = 4$, this gives $p < 32768$ which is easily sufficient for trial division in NFS.

With this bound on p, we have $r < (\beta-1)(\sqrt{\beta\ell}-\ell+1)$. We then decompose r into $r = r_1\beta+r_0$, where $0 \le r_0 < \beta$. This implies $r_1 < \sqrt{\beta\ell}$, and $r_1w_1 \le r_1(p-1) < \sqrt{\beta\ell}\left(\sqrt{\frac{\beta}{\ell}}-1\right) = \beta - \sqrt{\beta\ell}$.

The algorithm then does another reduction step by $s = r_1w_1 + r_0$. We would like $s = s_1\beta + s_0 < 2\beta - p$, so that a final reduction step $t = s_0 + s_1w_1 < \beta$ produces a one-word result. Since $r_1(p-1) < \beta-\sqrt{\beta\ell}$, $s < 2\beta-\sqrt{\beta\ell}-1 < 2\beta-p$. Since s_1 is either 0 or 1, the multiplication and addition in $s_0 + s_1w_1$ is really just a conditional addition.

Now we have a one-word integer t which is divisible by p if and only if N is. To determine whether $p \mid t$, we use the idea from [50, §9] to compute $u = tp^{-1} \bmod \beta$, using the precomputed $p_{\text{inv}} = p^{-1} \pmod{\beta}$. If $p \mid t$, t/p is an integer $< \beta$ and so the modular arithmetic mod β must produce the correct $u = t/p$. There are $\left\lfloor \frac{\beta-1}{p} + 1 \right\rfloor$ multiples of p (including 0) less than β, under division by p these map to the integers $\left[0, \ldots, \left\lfloor \frac{\beta-1}{p} \right\rfloor\right]$. Since p is coprime to β, multiplication by $p^{-1} \pmod{\beta}$ is a bijective map, so all non-multiples of p must map to the remaining integers $\left[\left\lfloor \frac{\beta-1}{p} \right\rfloor + 1, \beta - 1\right]$. Hence the test for divisibility can be done by a one-word multiplication by the precomputed constant p_{inv}, and one comparison to the precomputed constant $p_{\text{lim}} = \left\lfloor \frac{\beta-1}{p} \right\rfloor$.

4.2.2 Implementation

The algorithm is quite simple to implement on an x86 CPU, which offers the two-word product of two one-word arguments by a single MUL instruction. It might run as shown in Algorithm 6, where x_1, x_0 are registers that temporarily hold two-word products. A pair of registers holding a two-word value $r_1\beta + r_0$ is written as $r_1 : r_0$. The values $r_{0,1}$, $s_{0,1}$, and t_0 can all use the same registers, written $r_{0,1}$ here. The loop over j should be unrolled.

```
**Input**: Length $\ell$
    $N = \sum_{i=0}^{\ell} n_i\beta^i$, $0 \le n_i < \beta$
    Odd prime $p < \sqrt{\frac{\beta}{\ell}}$
    $w_j = \beta^j \bmod p$ for $1 \le j \le \ell$
    $p_{\text{inv}} = p^{-1} \bmod \beta$
    $p_{\text{lim}} = \left\lfloor \frac{\beta-1}{p} \right\rfloor$
**Output**: 1 if $p \mid N$, 0 otherwise
$r_0 := n_0$;
$r_1 := 0$;
**for** $j := 1$ **to** $\ell$ **do**
    $x_1 : x_0 := n_j \cdot w_j$;
    $r_1 : r_0 := r_1 : r_0 + x_1 : x_0$;
$x_0 := r_1 \cdot w_1$;
$r_0 := (r_0 + x_0) \bmod \beta$;
**if** *last addition set carry flag* **then**
    $r_0 := (r_0 + w_1) \bmod \beta$;
$r_0 := (r_0 \cdot p_{\text{inv}}) \bmod \beta$;
**if** $r_0 \le p_{lim}$ **then**
    **return** 1;
**else**
    **return** 0;
```

Algorithm 6: Pseudo-code for trial division of numbers of up to $\ell + 1$ words.

This code uses ℓ multiplications of two words to a two-word product. These multiplications are independent of one another, so they can overlap on a CPU with pipelined multiplier. On an Athlon64, Opteron, and Phenom CPUs, a multiplication can start every 2 clock cycles, the low word of the product is available after 4 clock cycles, the high word after 5 clock cycles. Thus in case of $\ell = 4$, the latency for the first 4 products and building their sum should be 12 cycles. The two remaining multiplies, the additions and conditional moves should be possible in about 11 cycles, giving a theoretical total count of about 23 clock cycles for trial dividing a 5 word integer by a small prime. Data movement from cache may introduce additional latency.

4.2.3 Use in NFS

Given a sieve region of size s with every d-th entry a sieve report, trial dividing by the prime p for all sieve reports has cost $O(s/d)$, while resieving has cost $O(rs/p)$, where r is the number of roots modulo p the sieved polynomial has. Hence whether trial division or resieving is preferable will depend on $\frac{p}{dr}$, where those p with $\frac{p}{dr} < c$ for some threshold c should use trial division.

As primes are divided out of N, the number of words in N may decrease, making the following trial division faster. It might be worthwhile to try to reduce the size of N as quickly as possible. The probability that a prime p divides N may be estimated as r/p, the size decrease as $\log(p)$, so the probability that trial division by p will decrease the number of words in N may be estimated as being proportional to $r\log(p)/p$. For trial division, the candidate divisors p can be sorted so that this estimate is decreasing. This probability estimate does not take into account the fact that N, being a sieve report, is likely smooth, and under this condition the probability that p divides N increases by Bayes' theorem, more so for larger p than for small ones.

4.2.4 Testing Several Primes at Once

Algorithm 6 reduces the input number to a one-word integer which is congruent to $N \pmod{p}$, then tests divisibility by p of that one-word integer. It is possible to do the reduction step for composite candidate divisors q, then test divisibility of the resulting one-word integer for all $p \mid q$. This way, for integers consisting of several words, the expensive reduction needs to be done only once for each q, the relatively cheap divisibility test for each p. This is attractive if the bound $q < \sqrt{\beta/\ell}$ is not too small. With $w = 2^{64}$, $\ell = 4$, we can use $q < 2147483648$, which allows for several small primes in q. For integers N with a larger number of words, it may be worthwhile to introduce an additional reduction step (for example, using Montgomery's REDC for a right-to-left reduction) to relax the bound on q to, e.g., $q < w/\ell$, so that the number of primes in q can be doubled at the cost of only two additional multiplies. In NFS, if the primes found by re-sieving have been divided out already before trial division begins, the N_i may not be large enough to make this approach worthwhile.

4.2.5 Performance of Trial Division

To measure the performance of the trial division code, we divide 10^7 consecutive integers of $1, \ldots, 5$ words by the first $n = 256$, 512, 1024, and 2048 odd primes on a 2 GHz AMD Phenom CPU, see Figure 4.1. The higher timings per trial division for $n = 256$ are due to the additional cost of dividing out found divisors, which has a greater relative contribution for smaller primes which divide more frequently. The timing for $\ell = 4, n = 2048$ is close to the predicted 23 clock cycles. The sudden increase for $n = 2048$ in the case of N with one word is due to caching: with 7 stored values (p, p_{inv}, p_{lim}, $w_{1,\ldots,4}$) of 8 bytes each, $n = 2048$ has a table of precomputed values of size 112KB, which exceeds the level-1 data cache size of 64KB of the Phenom. For large

	Number of words in N				
n	1	2	3	4	5
256	6.8 (2.6)	15.3 (6.0)	20.8 (8.1)	27.5 (10.7)	32.4 (12.6)
512	11.3 (2.2)	28.2 (5.5)	38.8 (7.6)	52.0 (10.2)	61.32 (12.0)
1024	21.3 (2.1)	54.9 (5.4)	75.9 (7.4)	102.0 (10.0)	120.7 (11.8)
2048	85.4 (4.1)	108.4 (5.3)	149.8 (7.3)	200.8 (9.8)	237.8 (11.6)

Table 4.1: Time in seconds for trial division of 10^7 consecutive integers by the first n odd primes on a 2 GHz AMD Phenom CPU. Time per trial division in nanoseconds in parentheses.

sets of candidate primes, the sequential passes through the precomputed data cause frequent misses in the level-1 cache, and the trial divisions for N of only one word are fast enough that transfer rate from the level-2 cache limits the execution. This could be avoided by computing fewer w_i constants (i.e., choosing a smaller ℓ) if the N are known to be small, or storing the w_i in separate arrays rather than interleaved, so that the w_i for larger i do not occupy cache while the N processed are small. Since the value of p is not actually needed during the trial division, it is possible to avoid storing it and recomputing it, e.g., from p_{inv} when it needs to be reported as a divisor.

4.3 Modular Arithmetic

The modular arithmetic operations are relatively inexpensive when moduli and residues of only a few machine words are considered, and should be implemented in a way that lets the compiler perform in-lining of simple arithmetic functions to avoid unnecessary function call overhead and data movement between registers, memory and stack due to the calling conventions of the language and architecture. Many simple arithmetic operations can be implemented easily and efficiently using assembly language, but are cumbersome to write in pure C code, especially if multi-word products or carry propagation are involved. The GNU C compiler offers a very flexible method of injecting assembly code into C programs, with an interface that tells the compiler all constraints on input and output data of the assembly block so that it can perform optimization on the code surrounding the assembly statements. By defining some commonly used arithmetic operations in assembly, much of the modular arithmetic can be written in C, letting the compiler handle register allocation and data movement. The resulting code is usually not optimal, but quite usable. For the most time-critical operations, writing hand-optimized assembly code offers an additional speed improvement.

For the present work, modular arithmetic for moduli of 1 machine word and of 2 machine words with the two most significant bits zero is implemented.

4.3.1 Assembly Support

To give an example of an elementary function that is implemented with the help of some assembly code, we examine modular addition with a modulus of 1 machine word. This is among the most simple operations possible, but useful as an example.

Let a "reduced residue" with respect to a positive modulus m mean an integer representative $0 \le r < m$ of the residue class $r \pmod m$. Modular addition of two reduced residues can be defined as

$$(a+b) \bmod m = \begin{cases} a+b-m & \text{if } a+b-m \geq 0 \\ a+b & \text{otherwise.} \end{cases}$$

If any modulus $m < \beta$ is permitted, where β is the machine word base, then the problem that $a+b$ might overflow the machine word arises. One could test for this case, then test if $a+b \geq m$, and subtract m if either is true, but this necessitates two tests. With a slight rearrangement, we can do with one:

```
1 $r := a + b$;
2 $s := a - m$;
3 $t := s + b$;
4 if last addition set carry flag then
5     r := t;
```

All arithmetic in this code is assumed modulo the word base β, i.e., the integers in r, s, and t are reduced residues modulo β. In line 2, since a is reduced modulo m, the subtraction $a - m$ necessarily produces a borrow, so that $s = a - m + \beta$. In line 3, if $s + b < \beta$, then this addition does not produce a carry, and $t = a+b-m+\beta < \beta$, i.e., $a+b-m < 0$. If $s+b \geq \beta$, the addition does produce a carry, and $0 \leq t = s+b-\beta = a+b-m$. Hence t is the proper result if and only if a carry occurs in line 3, to make up for the borrow of line 2. Lines 1 and 2 are independent and can be executed in parallel, leading to a dependent chain of length 3. We require $a < m$ for correctness, if $b \geq m$, the result still satisfies $r \equiv a + b \pmod m$ and $r < b$, but not necessarily $r < m$.

The implementation in C with a GCC x86 assembly block shown below. The value of s, shown separately for clarity above, is stored in `t` here.

```
r = a + b;
t = a - m;

__asm__ (
  "add %2, %1\n\t"    /* t := t + b */
  "cmovc %1, %0\n\t"  /* if (carry) r := t */
  : "+r" (r), "+&r" (t)
  : "g" (b)
  : "cc"
);
```

The computation of the initial `t` and `r` are done in C, to give the compiler some scheduling freedom. Since C does not provide direct access to the carry flag, the addition $t := t + b$ and the following conditional assignment are done in assembly. The constraints on the data passed to the assembly block state that the values of `r` and `t` must reside in registers (`"r"`) since the target of the conditional move instruction `cmovc` must be a register, and at least one of source or target of the addition instruction `add` must be a register. We allow the variable `b` to be passed in a register, in memory or as an immediate operand (`"g"`, "general" constraint, for x86_64 the correct constraint is `"rme"` since immediate constants are only 32 bit wide), which is the source operand to the `add` instruction. The `"+"` modifier tells that the values in `r` and `t` will be modified, and the `"&"` modifier tells that `t` may be modified before the end of the assembly block and thus no other input variable should be passed in the register assigned to `t`, even if their values are known to be identical. Finally, `"cc"` tells the compiler that the values of the flags register may change. These constraints provide the information the compiler needs to be able to use the assembly block correctly, while leaving enough flexibility that it can optimize register allocation and data movement, compared to, e.g., compilers that require all parameters to assembly blocks in a fixed set of registers.

An alternative solution is to compute $r := b-(m-a)$ and adding m if the outer subtraction produced a borrow. However, this requires a conditional addition rather than a conditional move.

Similar to the modular addition, various functions such as modular subtraction and multiplication for one and two-word moduli, two-word addition, subtraction, multiplication and binary shift, and division with a two-word dividend (used, for example, for preparing a residue for use with REDC modular reduction with a two-word modulus, see 4.3.2) are written as functions with assembly support. As optimization effort progresses, more time-critical functions currently written in C with assembly macros will be replaced by dedicated assembly code.

4.3.2 Modular Reduction with REDC

Montgomery presented in [64] a method for fast modular reduction. Given an integer $0 \le a < \beta m$, for odd modulus m of one machine word and machine word base β (here assumed a power of 2), and a precomputed constant $m_{\text{inv}} = -m^{-1} \bmod \beta$, it computes an integer $0 \le r < m$ which satisfies $r\beta \equiv a \pmod{m}$. It does so by computing the minimal non-negative tm such that $a + tm \equiv 0 \pmod{\beta}$, to make use of the fact that division by β is very inexpensive. Since $t < \beta$, $(a + tm)/\beta < 2m$, and at most one final subtraction of m ensures $r < m$. He calls the algorithm that carries out this reduction "REDC," shown in Algorithm 7.

```
Input: $m$, the modulus
       $\beta$, the word base
       $a < \beta m$, integer to reduce
       $m_{\text{inv}} < \beta$ such that $m m_{\text{inv}} \equiv -1 \pmod{\beta}$
Output: $r < m$ with $r\beta \equiv a \pmod{m}$
$t := a \cdot m_{\text{inv}} \bmod \beta$;
$r := (a + t \cdot m)/\beta$;
if $r \ge m$ then
    $r := r - m$;
```

Algorithm 7: Algorithm REDC for modular reduction with one-word modulus. All variables take non-negative integer values.

The reduced residue output by this algorithm is not in the same residue class mod m as the input, but the residue class gets multiplied by $\beta^{-1} \pmod{m}$ in the process. To prevent accumulating powers of $\beta^{-1} \pmod{m}$ and having unequal powers of β when, e.g., adding or comparing residues, any residue modulo m is converted to Montgomery representation first, by multiplying it by β and reducing (without REDC) modulo m, i.e., the Montgomery representation of a residue $a \pmod{m}$ is $a\beta \pmod{m}$. This way, if two residues in Montgomery representation $a\beta \pmod{m}$ and $b\beta \pmod{m}$ are multiplied and reduced via REDC, then $\text{REDC}(a\beta b\beta) \equiv ab\beta \pmod{m}$ is the product in Montgomery representation. This ensures the exponent of β in the residues always stays 1, and so allows addition, subtraction, and equality tests of residues in Montgomery representation. Since $\beta \perp m$, we also have $a\beta \equiv 0 \pmod{m}$ if and only if $a \equiv 0 \pmod{m}$, and $\gcd(a\beta, m) = \gcd(a, m)$. Since $\beta = 2^{32}$ or 2^{64} is an integer square, the Jacobi symbol satisfies $\left(\frac{a\beta}{m}\right) = \left(\frac{a}{m}\right)$.

For moduli m of more than one machine word, say $m < \beta^k$, a product of two reduced residues may exceed β, but is below $m\beta^k$. The reduction can be carried out in two ways: one essentially performs the one-word REDC reduction k times, performing $O\big(k^2\big)$ one-word multiplies, the other replaces arithmetic modulo β in REDC by arithmetic modulo β^k, performing $O(1)$ k-word multiplications. In either case, a full reduction with (repeated one-word or a single multi-word) REDC divides the residue class of the output by β^k, and the conversion to Montgomery representation must multiply by β^k accordingly. The former method has lower overhead and is preferable for small moduli, the latter can use asymptotically fast multiplication algorithms if the modulus is large. As in our application the moduli are quite small, no more than two machine words, we use the former method.

Before modular arithmetic with REDC for a particular m can begin, the constant m_{inv} needs to be computed. If β is a power of 2, Hensel lifting makes this computation very fast. To speed it up further, we try to guess an approximation to m_{inv} so that a few least significant bits are correct, thus saving a few Newton iterations. The square of any odd integer is congruent to 1 (mod 8), so $m_{\text{inv}} \equiv m \pmod 8$. The fourth bit of m_{inv} is equal to the binary exclusive-or of the second, third, and fourth bit of m, but on many microprocessors an alternative suggestion from Montgomery [72] is slightly faster: $(3m)$ XOR 2 gives the low 5 bits of m_{inv} correctly. Each Newton iteration $x \mapsto 2x - x^2 m$ doubles the number of correct bits, so that with either approximation, 3 iterations for $\beta = 2^{32}$ or 4 for $\beta = 2^{64}$ suffice.

Converting residues out of Montgomery representation can be performed quickly with REDC, but converting them to Montgomery representation requires another modular reduction algorithm. If such conversions are to be done frequently, it pays to precompute $\ell = \beta^2 \bmod m$, so that $\text{REDC}(a\ell) = a\beta \bmod m$ allows using REDC for the purpose.

In some cases, the final conditional subtraction of m in REDC can be omitted. If $a < m$, then $a + tm < m\beta$ since $t < \beta$, so $r = (a + tm)/\beta < m$ which can be used when converting residues out of Montgomery form, or when division by a power of 2 modulo m is desired.

4.3.3 Modular Inverse

To compute a modular inverse $r \equiv a^{-1} \pmod m$ for a given reduced residue a and odd modulus m with $a \perp m$, we use a binary extended Euclidean algorithm. Modular inverses are used at the beginning of stage 2 for the P–1 algorithm, and for initialisation of stage 1 of ECM (except for a select few curves which have simple enough parameters that they can be initialised using only division by small constants). Our code for a modular inverse takes about 0.5μs for one-word moduli, which in case of P–1 with small B_1 and B_2 parameters accounts for several percent of the total run-time, showing that some optimization effort is warranted for this function.

The extended Euclidean algorithm solves

$$ar + ms = \gcd(a, m)$$

for given a, m by initialising $e_0 = 0, f_0 = 1, g_0 = m$ and $e_1 = 1, f_1 = 0, g_1 = a$, and computing sequences e_i, f_i and g_i that maintain

$$ae_i + mf_i = g_i \tag{4.1}$$

where $\gcd(a, m) \mid g_i$ and the g_i are strictly decreasing until $g_i = 0$. The original Euclidean algorithm uses $g_i = g_{i-2} \bmod g_{i-1}$, that is, in each step we write $g_i = g_{i-2} - g_{i-1}\lfloor\frac{g_{i-2}}{g_{i-1}}\rfloor$ and likewise $e_i = e_{i-2} - e_{i-1}\lfloor\frac{g_{i-2}}{g_{i-1}}\rfloor$ and $f_i = f_{i-2} - f_{i-1}\lfloor\frac{g_{i-2}}{g_{i-1}}\rfloor$, so that equation (4.1) holds for each i. If n is the smallest i such that $g_i = 0$, then $g_{n-1} = \gcd(a, m)$, $s = f_{n-1}$, and $r = e_{n-1}$. Since we only want the value of $r = e_{n-1}$, we don't need to compute the f_i values. We can write

$u = e_{i-1}, v = e_i, x = g_{i-1}, y = g_i$ and for $i = 1$ initialise $u = 0, v = 1, x = m$, and $y = a$. Then each iteration $i \mapsto i + 1$ is computed by

$$(u, v, x, y) := (v, u - \lfloor x/y \rfloor v, y, x - \lfloor x/y \rfloor y).$$

At the first iteration where $y = 0$, we have $r = u$ and $x = 1$ if a and m were indeed coprime.

A problem with this algorithm is the costly computation of $\lfloor x/y \rfloor$ as integer division is usually slow. The binary extended Euclidean algorithm avoids this problem by using only subtraction and division by powers of 2. Our implementation is inspired by code written by Robert Harley for the ECCp-97 challenge and is shown in Algorithm 8. The updates maintain $ua \equiv -x2^t \pmod{m}$ and $va \equiv y2^t \pmod{m}$ so that when $y = 1$, we have $r = v2^{-t} = a^{-1} \pmod{m}$.

```
Input: Odd modulus $m$
    Reduced residue $a \pmod{m}$
Output: Reduced residue $r \pmod{m}$ with $ar \equiv 1 \pmod{m}$, or failure if $\gcd(a, m) > 1$
if $a = 0$ then
    return failure;
$t := \mathrm{Val}_2(a)$;                                   /* $2^t \,||\, a$ */
$u := 0; v := 1; x := m; y := a/2^t$;
while $x \neq y$ do
    $\ell := \mathrm{Val}_2(x - y)$;                        /* $2^\ell \,||\, x - y$ */
    if $x < y$ then
        $(u, v, x, y, t) := (u2^\ell, u + v, x, (y - x)/2^\ell, t + \ell)$;
    else
        $(u, v, x, y, t) := (u + v, v2^\ell, (x - y)/2^\ell, y, t + \ell)$;
if $y \neq 1$ then
    return failure;
$r := v2^{-t} \bmod m$;
```

Algorithm 8: Binary extended GCD algorithm.

In each step we subtract the smaller of x, y from the larger, so they are decreasing and non-negative. Neither can become zero as that implies $x = y$ in the previous iteration, which terminates the loop. Since both are odd at the beginning of each iteration, their difference is even, so one value decreases by at least a factor of 2, and the number of iterations is at most $\log_2(am)$. In each iteration, $uy + vx = m$, and since x and y are positive, $u, v \leq m$ so that no overflow occurs with fixed-precision arithmetic.

To perform the modular division $r = v/2^{t_i}$, we can use REDC. While $t \geq \log_2(\beta)$, we replace $v := \mathrm{REDC}(v)$ and $t := t - \log_2(\beta)$. Then, if $t > 0$, we perform a variable-width REDC to divide by 2^t rather than by β by computing $r = \left(v + \left((vm_{\mathrm{inv}}) \bmod 2^t\right) m\right)/2^t$ with $mm_{\mathrm{inv}} \equiv -1 \pmod{\beta}$. Since $v < m$, we don't need a final subtraction in these REDC.

If the residue a whose inverse we want is given in Montgomery representation $a\beta^k \bmod m$ with k-word modulus m, we can use REDC $2k$ times to compute $a\beta^{-k} \bmod m$, then compute the modular inverse to obtain the inverse of a in Montgomery representation: $a^{-1}\beta^k \equiv \left(a\beta^{-k}\right)^{-1} \pmod{m}$. This can be simplified by using the fact that the binary extended GCD computes $v = a^{-1}2^t$. If we know beforehand that $t \geq \log_2 \beta$, we can skip divisions by β via REDC both before and after the binary extended GCD. Let the function $t(x, y)$ give the value of t at the end of Algorithm 8 for coprime inputs x, y. It satisfies

$$t(x,y) = \begin{cases} 0 & \text{if } x = y \text{ (implies } x = y = 1), \\ t(x/2, y) + 1 & \text{if } x \neq y,\ 2 \mid x, \\ t(x-y, y) & \text{if } x > y,\ 2 \nmid x, \\ t(y, x) & \text{if } x < y,\ 2 \nmid x. \end{cases}$$

Assuming y odd, case 3 is always followed by case 2, and we can substitute case 3 by $t(x, y) = t((x-y)/2, y) + 1$. We compare the decrease of the sum $x + y$ and the increase of t. In case 2, $(x+y) \mapsto x/2+y > (x+y)/2$, and t increases by 1. In the substituted case 3, $(x+y) \mapsto (x+y)/2$, and t increases by 1. We see that whenever $x+y$ decreases, t increases, and whenever t increases by 1, $x + y$ drops by at most half, until $x + y = 2$. Hence $t(x, y) \geq \log_2(x + y) - 1$, and therefore $t(x, y) \geq \log_2(y)$, since $x > 0$.

Thus in case of k-word moduli $\beta^{k-1} < m < \beta^k$, we have $t(x, m) \geq (k-1)\log_2(\beta)$ for any positive x, so using $a\beta^{-1} \pmod m$ as input to the binary extended GCD is sufficient to ensure that at the end we get $a^{-1}\beta \equiv v2^{-t} \pmod m$, or $a^{-1}\beta^k \equiv v2^{-t+(k-1)\log_2(\beta)} \pmod m$ and the desired result $a^{-1}\beta^k$ can be obtained from $v2^{-t}$ with a division by $2^{t-(k-1)\log_2(\beta)}$ via REDC.

4.3.4 Modular Division by Small Integers

Initialisation of P+1 and ECM involves division of residues by small integers such as 3, 5, 7, 11, 13 or 37. These can be carried out quickly by use of dedicated functions. To compute $r \equiv ad^{-1} \pmod m$ for a reduced residue a with $d \perp m$, we first compute $t = a + km$, with k such that $t \equiv 0 \pmod d$, i.e., $k = a\left(-m^{-1}\right) \bmod d$, where $-m^{-1} \bmod d$ is determined by look-up in a precomputed table for the $d - 1$ possible values of $m \bmod d$.

For one-word moduli, the resulting integer t can be divided by d via multiplication by the precomputed constant $d_{\text{inv}} \equiv d^{-1} \pmod \beta$. Since $t/d < m < \beta$ is an integer, the result $r = td_{\text{inv}} \bmod \beta$ produces the correct reduced residue r. This implies that computing t modulo β is sufficient.

For two-word moduli, we can choose an algorithm depending on whether m and d are large enough that t may overflow two machine words or not. In either case, we may write $t = t_1\beta + t_0$ with $0 \leq t_0 < \beta$, $0 \leq t_1 < d\beta$ and $r = r_1\beta + r_0$ with $0 \leq r_0, r_1 < \beta$, and can compute $r_0 = t_0 d_{\text{inv}} \bmod \beta$.

If t does not overflow, we may write $t = t'' + t'd\beta, 0 \leq t'' < d\beta$, where $d \mid t''$. Then $r = t/d = t'\beta + t''/d$ with $t''/d < \beta$, so we can compute $r_1 = \lfloor t_1/d \rfloor$. The truncating division by the invariant d can be implemented by the methods of [50]. An advantage of this approach is that the computation of the low word r_0 from t_0 is independent of the computation of the high word r_1 from t_1.

If t may overflow two machine words, we can compute r_0 as before, and use that $t - dr_0$ is divisible by $d\beta$, so we may write $r_1\beta + r_0 \equiv t/d \pmod{\beta^2}$ as $r_1 \equiv (t - dr_0)/\beta \cdot d_{\text{inv}} \pmod \beta$.

4.4 P–1 Algorithm

The P–1 algorithm is described in Section 2.2. We recapitulate some elementary facts here. The first stage of P–1 computes

$$x_1 = x_0^e \bmod N$$

for some starting value $x_0 \not\equiv 0, \pm 1 \pmod N$ and a highly composite integer exponent e. By Fermat's little theorem, if $p - 1 \mid e$ for any $p \mid N$, then $x_1 \equiv 1 \pmod p$ and $p \mid \gcd(x_1 - 1, N)$.

This condition is sufficient but not necessary: it is enough (and necessary) that $\mathrm{ord}_p(x_0) \mid e$, where $\mathrm{ord}_p(x_0)$ is the order of x_0 in $\mathbb{F}_p^*$. To maximise the probability that $\mathrm{ord}_p(x_0) \mid e$ for a given size of e, we could choose e to contain all primes and prime powers that divide $\mathrm{ord}_p(x_0)$ with probability better than some bound $1/B_1$. One typically assumes that a prime power q^k divides $\mathrm{ord}_p(x_0)$ with probability q^{-k}, so that e is taken as the product of all primes and prime powers not exceeding B_1, or $e = \mathrm{lcm}(1, 2, 3, 4, \ldots, B_1)$. The choice of e is described in more detail in Chapter 5.

The value of e is precomputed and passed to the P–1 stage 1 routine, which basically consists only of a modular exponentiation, a subtraction and a gcd. The base x_0 for the exponentiation is chosen as 2; in a left-to-right binary powering ladder, this requires only squarings and doublings, where the latter can be performed quickly with an addition instead of a multiplication by x_0.

To reduce the probability that all prime factors of N (i.e., N itself) are found simultaneously and reported as a divisor at the end of stage 1, only the odd part of e is processed at first, and then the factors of 2 in e one at a time by successive squarings. After each one we check if the new residue is 1 (mod N), indicating that all prime factors of N have been found now, and if so, revert to the previous value to use it for the gcd. Unless the same power of 2 divides $\mathrm{ord}_p(x_0)$ exactly for all primes $p \mid N$, then this will discover a proper factor. This backtracking scheme is simple but satisfactorily effective: among 10^6 composite numbers that occurred during an sieving experiment of the RSA155 number, each composite being of up to 86 bits and with prime factors larger than 2^{24}, only 48 had the input number reported as the factor in P–1 stage 1 with $B_1 = 500$. Without the backtracking scheme (i.e., processing the full exponentiation by e, then taking a GCD), 879 input numbers are reported as factors instead.

The second stage of P–1 can use exactly the same implementation as the second stage of P+1, by passing $X_1 = x_1 + x_1^{-1}$ to the stage 2 algorithm. The stage 2 algorithm for ECM is very similar as well, and they are described together in Section 4.7.

4.4.1 P–1 Stage 1 Performance

Table 4.2 compares the performance of the P–1 stage 1 implementation for different B_1 values and modulus sizes on AMD Phenom and Intel Core 2 CPUs.

	Core 2		Phenom	
B_1	1 word	2 words −2 bits	1 word	2 words −2 bits
100	3.15	6.24	2.49	4.59
200	5.38	12.2	4.12	8.26
300	7.28	17.2	5.51	11.3
400	9.23	22.2	6.92	14.5
500	11.4	27.8	8.49	18.0
600	13.2	32.7	9.83	21.0
700	15.4	38.2	11.4	24.4
800	17.2	43.1	12.7	27.5
900	19.4	48.5	14.2	30.9
1000	21.4	53.8	15.7	34.1

Table 4.2: Time in microseconds for P–1 stage 1 with different B_1 values on 2.146 GHz Intel Core 2 and 2 GHz AMD Phenom CPUs.

4.5 P+1 Algorithm

The P+1 algorithm is described in detail in Section 2.2. We recapitulate the basic algorithm here.

The first stage of P+1 computes $x_1 = V_e(x_0) \bmod N$, where $x_0 \in \mathbb{Z}/N\mathbb{Z}$ is a parameter, $V_n(x)$ is a degree-n Chebyshev polynomial defined by $V_n(x+x^{-1}) = x^n + x^{-n}$, and e is a highly composite integer chosen as for the P–1 method. These Chebyshev polynomials satisfy $V_0(x) = 2$, $V_1(x) = x$, $V_{-n}(x) = V_n(x)$, $V_{mn}(x) = V_m(V_n(x))$, and $V_{m+n}(x) = V_m(x)V_n(x) - V_{m-n}(x)$.

We test for a factor by taking $\gcd(x_1 - 2, N)$. If there is a prime p such that $p \mid N$ and $p - \left(\frac{\Delta}{p}\right) \mid e$, where $\Delta = x_0^2 - 4$ and $\left(\frac{\Delta}{p}\right)$ is the Legendre symbol, then $p \mid \gcd(x_1 - 2, N)$.

Since V_{n-m} is required for computing V_{n+m}, these polynomials cannot be evaluated with a simple binary addition chain as in the case of the exponentiation in stage 1 of P–1. Instead, an addition chain needs to be used that contains $n - m$ whenever the sum $n + m$ is formed from n and m. These chains are described in Section 4.5.1.

The required addition chain for the stage 1 multiplier e is precomputed and stored as compressed byte code, see Section 4.5.2.

As for P–1, a backtracking scheme is used to avoid finding all factors of N and thus reporting the input number as the factor found. Since factors of 2 in e can easily be handled by $V_{2n}(x) = V_2\left(V_n(x)\right) = V_n(x)^2 - 2$, they need not be stored in the precomputed addition chain, and can be processed one at a time. Similarly as in stage 1 of P–1, we remember the previous residue, process one factor of 2 of e, and if the result is $2 \pmod N$, meaning that all factors of N have been found, we revert to the previous residue to take the GCD with N. Using the same 10^6 composite inputs as for P–1, P+1 with $B_1 = 500$ reports 117 input numbers as factors with backtracking, and 1527 without.

If stage 1 of P+1 is unsuccessful, we can try to find a factor yet by running stage 2, using as input the output x_1 of stage 1. Our stage 2 is identical for P–1 and P+1, and very similar for ECM, and is described in Section 4.7.

4.5.1 Lucas Chains

Montgomery shows in [66] how to generate addition chains $a_0, a_1, \ldots, a_\ell$ with $a_0 = 1$ and length ℓ such that for any $0 < i \le \ell$, there exist $0 \le s, t < i$ such that $a_i = a_s + a_t$ and $a_s - a_t$ is either zero, or is also present in the chain. He calls such chains "Lucas chains." For example, the addition chain $1, 2, 4, 5$ is not a Lucas chain since the last term 5 can be generated only from $4 + 1$, but $4 - 1 = 3$ is not in the chain. The addition chain $1, 2, 3, 5$, however, is a Lucas chain. For any positive integer n, $L(n)$ denotes the length of an optimal (i.e., shortest possible) Lucas chain that ends in n.

A simple but generally non-optimal way of generating such chains uses the reduction $(n, n-1) \mapsto (\lceil n/2\rceil, \lceil n/2\rceil - 1)$. We can compute $V_n(x)$ and $V_{n-1}(x)$ from $V_{\lceil n/2\rceil}(x)$, $V_{\lceil n/2\rceil-1}(x)$, $V_1(x) = x$, and $V_0(x) = 2$. In the case of n even, we use $V_n(x) = V_{\lceil n/2\rceil}(x)^2 - V_0(x)$, and $V_{n-1}(x) = V_{\lceil n/2\rceil}(x)V_{\lceil n/2\rceil-1}(x) - V_1(x)$ and in the case of n odd, we use $V_n = V_{\lceil n/2\rceil}(x)V_{\lceil n/2\rceil-1}(x) - V_1(x)$ and $V_{n-1}(x) = V_{\lceil n/2\rceil-1}(x)^2 - V_0(x)$. The resulting chain allows processing the multiplier left-to-right one bit at a time, and thus is called binary chain by Montgomery. Each bit in the multiplier adds two terms to the addition chain, except that when processing the final bit, only one of the two values needs to be computed, and if the two most significant bits (MSB) are 10_b, the above rule would compute $V_2(x)$ twice of which one should be skipped. Any trailing zero bits can be handled by $V_{2n}(x) = V_n(x)^2 - V_0(x)$ at the cost of 1 multiplication each. The length $L_b(n2^k)$ for the binary Lucas chain for a number $n2^k$ with n odd is therefore $2\lfloor \log_2(n)\rfloor - 1 + k$ if the two

MSB are 10_b, or $2\lfloor\log_2(n)\rfloor + k$ if $n = 1$ or the two MSB are 11_b. Examples are in Table 4.3. It lists the binary chain, the length $L_b(n)$ of the binary chain, an optimal chain, and the length $L(n)$ of an optimal chain, for odd n up to 15.

n	Binary chain	$L_b(n)$	Optimal chain	L(n)
$3 = 11_b$	$1, 2, 3$	2	$1, 2, 3$	2
$5 = 101_b$	$1, 2, 3, 5$	3	$1, 2, 3, 5$	3
$7 = 111_b$	$1, 2, 3, 4, 7$	4	$1, 2, 3, 4, 7$	4
$9 = 1001_b$	$1, 2, 3, 4, 5, 9$	5	$1, 2, 3, 6, 9$	4
$11 = 1011_b$	$1, 2, 3, 5, 6, 11$	5	$1, 2, 3, 5, 6, 11$	5
$13 = 1101_b$	$1, 2, 3, 4, 6, 7, 13$	6	$1, 2, 3, 5, 8, 13$	5
$15 = 1111_b$	$1, 2, 3, 4, 7, 8, 15$	6	$1, 2, 3, 6, 9, 15$	5

Table 4.3: Binary and optimal Lucas chains for small odd values n

The binary chain is very easy to implement, but produces non-optimal Lucas chains except for very small multipliers. The smallest positive integer where the binary method does not produce an optimal chain is 9, and the smallest such prime is 13. Montgomery shows that if n is a prime but not a Fibonacci prime, an optimal Lucas chain for n has length $L(n) \geq r$ with r minimal such that $n \leq F_{r+2} - F_{r-3}$, where F_k is the k-th Fibonacci number. Since $F_k = (\phi^k - \phi^{-k})/\sqrt{5}$ where $\phi = (1 + \sqrt{5})/2$ is the Golden Ratio, this suggests that if this bound is tight, for large n an optimal chain for n should be about 28% shorter than the binary chain.

In a Lucas chain $a_0, a_1, \ldots, a_\ell$ of length ℓ, a doubling step $a_{k+1} = 2a_k$ causes all a_i with $k \leq i \leq \ell$ to be multiples of a_k, and all these terms a_i are formed using sums and differences only of terms $a_j, k < j \leq \ell$, see [66]. Such a doubling step corresponds to a concatenation of Lucas chains. For composite $n = n_1 \cdot n_2$, a Lucas chain can be made by concatenating the chains of its factors. E.g., for $n = 15$, we could multiply every entry in the chain $1, 2, 3, 5$ by 3 and append it to the chain $1, 2, 3$ (omitting the repeated entry 3) to form the Lucas chain $1, 2, 3, 6, 9, 15$. Since any Lucas chain starts with $1, 2$, every concatenation introduces one doubling step, and every doubling step leads to a chain that is the concatenation of two Lucas chains. Chains that are not the concatenation of other chains (i.e., that contain no doubling step other than $1, 2$) are called simple chains. For prime n, only simple chains exist. In the case of binary Lucas chains, the concatenated chain is never longer than the chain for the composite value and usually shorter, so that forming a concatenated Lucas chain from chains of the prime factors of n (if known) is always advisable. The same is not true for optimal chains, as shown below.

Optimal chains can be found by exhaustive search for a chosen maximal length $l_{\max}$ and maximal end-value $n_{\max}$. For odd $n \geq 3$, a Lucas chain for n always starts with $1, 2, 3$ since a doubling step $2, 4$ would produce only even values in the remainder of the chain. In the exhaustive search, the Lucas chain $a_0, \ldots, a_k$ can be extended recursively if $k < \ell_{\max}$ and $a_k < n_{\max}$ by adding an element $a_{k+1} > a_k$ such that the resulting sequence is still a Lucas chain, i.e., satisfying that there are $0 \leq i, j \leq k$ such that either $a_{k+1} = 2a_i$, or $a_{k+1} = a_i + a_j$ and $a_i - a_j$ is present in the chain. For each chain so created, we check in a table of best known lengths whether the length $k + 1$ is smaller than the previously known shortest length for reaching a_{k+1}, and if so, update it to $k + 1$ and store the current chain as the best known for reaching a_{k+1}. By trying all possible chain expansions, we are certain to find an optimal chain for every $n \leq n_{\max}$. This recursive search is very time consuming due to a large number of combinations to try. To reach a worthwhile search depth, the possible chain extensions can be restricted. The last step of an optimal chain is always $a_l = a_{\ell-1} + a_{\ell-2}$ as otherwise one or both of $a_{\ell-1}, a_{\ell-2}$ are obsolete, so

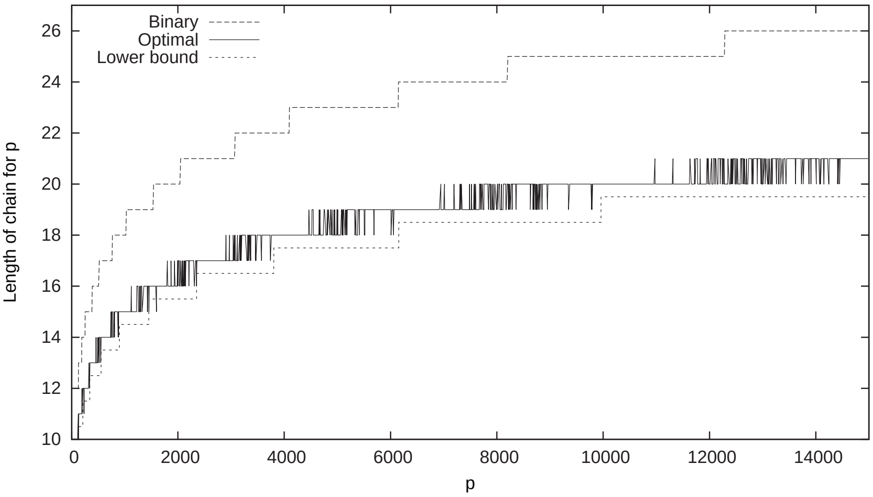

Figure 4.2: Length of binary and optimal Lucas chains for odd primes p in $[100, 15000]$, and a lower bound on the length for primes that are not Fibonacci primes. The graph for the bound is set 0.5 lower to make it visible. The Fibonacci prime 1597 is seen to undershoot this lower bound.

the table of best known lengths needs to be checked and updated only after such an addition step, and the final recursion level of the search needs to consider only this addition step. Any doubling step $a_{k+1} = 2a_k$ causes the chain to become the equivalent of a concatenated chain, so during the recursive chain expansion, doubling steps need not be considered. Instead the recursive search produces only the optimal lengths of simple chains. Then for all possible pairs $3 \leq m \leq n \leq \sqrt{n_{\max}}$, the length of the chain for mn is updated with the sum of the lengths of chains for m and n, if the latter is shorter. This is repeated until no more improvements occur. After the first pass, the optimal lengths of chains for all n where n has at most two prime factors are known. After the second pass, for all n that contain at most three primes, etc., until after at most $O(\log(n_{\max}))$ passes optimal lengths for all values are known. Using this search method, the minimal lengths of Lucas chains for primes $100 < n < 10000$ have been determined, shown in Figure 4.2. It compares the length of the binary Lucas chain, the optimal Lucas chain and the lower bound on the length of Lucas chains for primes that aren't Fibonacci primes. This lower bound is quite tight, in the examined data $L(n)$ does not exceed it by more than 1. The Fibonacci prime 1597 can be seen to undershoot this lower bound (as do the smaller Fibonacci primes, but they are difficult to see in the graph).

The exhaustive search method is extremely slow and useless for producing addition chains for P+1 or ECM if large B_1 values are desired. Montgomery [66] suggests the algorithm "PRAC," which produces Lucas chains based on GCD chains, noting that a subtractive GCD algorithm for n, r with $n > r$ and $n \perp r$ always produces a valid Lucas chain for n. However, the resulting Lucas chain has length equal to the sum of the partial quotients in the continued fraction expansion of $n/(n-r)$, and if a large partial quotient appears, the resulting Lucas chain is unreasonably long. He fixes this problem by introducing additional rules for reduction in the GCD chain (rather than just replacing the larger of the two partial remainders by their absolute difference as in a purely subtractive GCD chain) to avoid situations where the quotient of the partial remainders deviates too far from the Golden Ratio, yet satisfying the conditions for a Lucas chain. The great advantage is that PRAC usually produces very good chains and does so rapidly. This way it is feasible to try a few different suitable r for a given n, and for n in the range of interest for P+1 and ECM, one usually discovers an optimal chain this way.

It remains the problem of choosing a suitable $r \perp n$ to start the GCD chain, hoping to find a (near) optimal chain. Montgomery suggests trying $r = n - \lfloor n/c \rceil$ for several irrational c such that the continued fraction expansion of c has small partial quotients. This way, the partial fraction expansion of $n/(n-r)$ starts with small partial quotients as well. Good choices are the golden Ratio $c_0 = \phi$, whose partial quotients all are 1, or numbers with partial quotients all 1 except for one or two 2 among the first 10 partial quotients. The resulting large number of multipliers is not a problem if the Lucas chains are precomputed, but in cases where they are computed on-the-fly during stage 1 of P+1 or ECM, a smaller set of multipliers should be used, say, only those with at most one 2 among the first ten partial quotients.

Even with a large set of c_i values to try, PRAC in the form given by Montgomery cannot always obtain an optimal chain. The smallest example is $n = 751$ which has two Lucas chains of optimal length $L(751) = 14$:
$1, 2, 3, 5, 7, 12, 19, 24, 43, 67, 110, 177, 287, 464, 751$ and
$1, 2, 3, 5, 8, 13, 21, 34, 55, 68, 123, 191, 314, 437, 751$.
Both chains involve an addition step that references a difference that occurred 5 steps before the new term: for the former sequence in the step $a_8 = 43 = a_7 + a_6 = 24 + 19$, with difference $a_7 - a_6 = 5 = a_3$, and for the latter sequence in the step $a_{10} = 123 = a_9 + a_8 = 68 + 55$, with difference $a_9 - a_8 = 13 = a_5$. The original PRAC algorithm does not have any rule that allows utilizing a difference that occurred more than 4 steps before the new term and so cannot find either of these two chains. Another, similar case is $n = 1087$. For primes below 10000, I found 40 cases where PRAC did not find an optimal chain. For the purpose of generating Lucas chains for P+1 and ECM, these missed opportunities at optimal chains are of no great consequence. When using P+1 and ECM as a factoring subroutine in NFS, the B_1 value is often less than 751 so that such cases do not occur at all, and if a greater B_1 should be used, they occur so rarely that adding more rules to PRAC so that optimal chains are found for all primes below B_1 would increase the code complexity of our P+1 or ECM stage 1, which implements each PRAC rule (see Section 4.5.2), for little gain. For our implementation, this was not deemed worthwhile. For the purpose of finding optimal Lucas chains rapidly, it would be interesting to augment PRAC with a suitable rule for the required addition step $a_k = a_{k-1} + a_{k-2}$ with $a_{k-1} - a_{k-2} = a_{k-5}$, and testing which primes remain such that the modified PRAC cannot find optimal chains.

For composite $n = pq$, we trivially have $L(n) \leq L(p) + L(q)$, since we can concatenate the chain for p and the chain for q. In some cases, a shorter chain for the composite n exists than for the concatenated chains of its factors. The smallest example is $1219 = 23 \cdot 53$ which has

$$1, 2, 3, 4, 7, 11, 18, 29, 47, 76, 123, 170, 293, 463, 756, 1219$$

as an optimal chain of length 15, while an optimal chain for 23 is $1, 2, 3, 4, 5, 9, 14, 23$ of length 7, and for 53 is $1, 2, 3, 5, 6, 7, 13, 20, 33, 53$ of length 9.

Similarly, composite numbers n exist where PRAC with a certain set of multipliers finds a chain for n that is shorter than the concatenated simple chains for the divisors of n. A problem is that the starting pair n, r for the GCD sequence must be coprime, possibly making several c multipliers ineligible for an n with small prime factors. Starting with a large enough set of multipliers, usually enough of them produce coprime n and r that an optimal chain can be found, if one exists of a form suitable for PRAC. The example $n = 1219$ above is found, e.g., with $r = 882$, using the multiplier $3 - \Phi$ with continued fraction expansion $//1, 2, 1, 1, 1, 1, \ldots//$.

4.5.2 Byte Code and Compression

In implementations of P+1 or ECM such as in GMP-ECM [103] that typically operates on numbers of hundreds to ten-thousands of digits, or in the ECM implementation of Prime95 [102] that operates on number of up to several million digits, the cost of generating good Lucas chains on-the-fly during stage 1 is mostly negligible, except for P+1 on relatively small numbers of only a few hundred digits. However, in an implementation of ECM and especially P+1 designed for numbers of only a few machine words, the on-the-fly generation of Lucas chains would take an unacceptable part of the total run-time. Since in our application of using P+1 and ECM as a factoring sub-routine in NFS, identical stage 1 parameters are used many times over again, it is possible to precompute optimized Lucas chains and process the stored chain during P+1 or ECM stage 1.

This raises the question how the chain should be stored. Since the PRAC algorithm repeatedly applies one of nine rules to produce a Lucas chain for a given input, an obvious method is to store the sequence of PRAC rules to apply. The precomputation outputs a sequence of bytes where each byte stores the index of the PRAC rule to use, or one of two extra indices for the initial doubling resp. the final addition that is common to all (near-)optimal Lucas chains. This way, a byte code is generated that can be processed by an interpreter to carry out the stage 1 computations for P+1 or ECM. For each prime to include in stage 1, the corresponding byte code is simply appended to the byte code, which results in a (long) concatenated Lucas chain for the product of all stage 1 primes. If primes are to be included whose product is known to have a better simple Lucas chain than the concatenation of the chains for the individual primes, then their product should be passed to the byte-code generating function.

The byte code generated by PRAC is highly repetitive. For example, byte codes for the PRAC chains for the primes 101, 103, 107, and 109 are

$$\begin{array}{rl} 101: & 10,3,3,0,3,3,0,5,3,3,3,3,11 \\ 103: & 10,3,0,3,3,0,3,3,0,4,3,11 \\ 107: & 10,3,0,3,3,0,3,0,4,3,3,3,11 \\ 109: & 10,3,0,3,0,1,1,3,11 \end{array}$$

It is beneficial to reduce redundancy in the byte code to speed up stage 1. The byte code interpreter that executes stage 1 must fetch a code byte, then call the program code that carries out the arithmetic operations that implement the PRAC rule indicated by the code; thus there is a cost associated with each code byte. If the interpreter uses a computed jump to the code implementing each PRAC rule, there is also a branch misprediction each time a code byte is different from the previous one, as current microprocessors typically predict computed jumps as going to the same address as they did the previous time. Some PRAC rules frequently occur together, such as rule 3 followed by rule 0, so that merging them may lead to simplifications in the arithmetic. In particular, rules 11 (end of a simple chain) and 10 (start of a new simple chain) always appear together, except at the very start and at the very end of the byte code.

These issues are addressed by byte code compression. A simple static dictionary coder greedily translates frequently observed patterns into new codes. The byte code interpreter implements merged rules accordingly. For example, the byte code sequence "3, 0" (for an addition followed by a swap of variable contents) occurs very frequently and may be translated to a new code, say 12, and the interpreter performs a merged addition and swap. The codes "11, 10" always occur as a pair and can be substituted except at the very start and the very end of the bytecode, but these two occurrences can be hard-coded into the interpreter, so they do not need to be considered individually at all.

Since we often can choose among several different Lucas chains of equal length for a given stage 1 prime by using different multipliers in PRAC, we can pick one that leads to the simplest compressed code by compressing each candidate chain, and choosing the one that has the smallest number of code bytes and code byte changes.

For comparison, without any compression or effort to reduce the number of code bytes or code changes when choosing PRAC multipliers, the byte code for a stage 1 with $B_1 = 500$ consists of 1487 code bytes and 1357 code changes, whereas even with the simple substitution rules described above and careful choice of PRAC multipliers to minimize the number of code bytes and code changes, only 554 code bytes with 435 code changes remain.

4.5.3 P+1 Stage 1 Performance

Table 4.4 compares the performance of the P+1 stage 1 implementation for different B_1 values and modulus sizes on AMD Phenom and Intel Core 2 CPUs.

	Core 2		Phenom	
B_1	1 word	2 words −2 bits	1 word	2 words −2 bits
100	4.04	8.44	3.45	6.30
200	7.50	17.3	6.32	12.3
300	10.3	24.6	8.69	17.2
400	13.4	32.5	11.2	22.3
500	16.6	40.7	14.0	27.9
600	19.5	48.0	16.4	32.8
700	22.8	56.6	19.1	38.5
800	25.7	64.0	21.5	43.5
900	28.9	72.4	24.2	48.9
1000	32.0	80.4	26.7	54.2

Table 4.4: Time in microseconds for P+1 stage 1 with different B_1 values on 2.146 GHz Intel Core 2 and 2 GHz AMD Phenom CPUs, using precomputed Lucas chains stored as compressed byte code

For comparison, without using byte code compression or choosing the PRAC multipliers to minimize byte code length and number of code changes, on Core 2, P+1 stage 1 with 1 word and $B_1 = 500$ takes $20.4\mu s$ and so is about 22% slower, and with 2 words takes $50.4\mu s$ and so is about 24% slower.

4.6 ECM Algorithm

The Elliptic Curve Method of factorization was introduced by H. W. Lenstra in 1987 [62]. Whereas P–1 works in the group $\mathbb{F}_p^*$ of order $p-1$ and P+1 in a subgroup of $\mathbb{F}_{p^2}^*$ of order $p-1$ or $p+1$, ECM works in the Mordell-Weil group of points on an elliptic curve defined over $\mathbb{F}_p$. By Hasse's theorem, the number of points and therefore the order of the Mordell-Weil group of an elliptic curve over $\mathbb{F}_p$ is in $[p+1-2\sqrt{p}, p+1+2\sqrt{p}]$. The number of points on a particular curve depends on both the curve parameters and the field. ECM finds a prime factor p of N if the curve over $\mathbb{F}_p$ has smooth order; the advantage of ECM over previous algorithms such as P–1 and P+1 (which always work in a group of order $p-1$ or $p+1$) is that many different curves can be tried, until one with sufficiently smooth order is encountered.

Any elliptic curve E over a field K of characteristic neither 2 nor 3 can be defined by the Weierstraß equation

$$y^2 = x^3 + ax + b. \tag{4.2}$$

This equation defines an elliptic curve if and only if the discriminant $4a^3 + 27b^2$ does not vanish. The set of points on E consists of the solutions $(x, y) \in K^2$ of (4.2), plus the point at infinity $\mathcal{O}$.

The group addition law of two points on the curve is defined geometrically by putting a straight line through the two points (or, if the points are identical, the tangent of the curve in that point), taking the line's new intersection point with the curve and mirroring it at the x-axis. Since the curve is symmetric around the x-axis, the mirrored point is on the curve, and is the result. If the straight line is vertical, no new intersection point exists; in this case the point at infinity is taken as the result. The point at infinity is the identity element of the group, adding it to any point results in the same point. The inverse of a point is the point mirrored at the x-axis. This addition law on the points of an elliptic curve defines an Abelian group, see for example [91].

The Weierstraß form of elliptic curves can be used for implementing ECM, but requires a costly modular inverse in the computation of point additions. Montgomery [65] proposes an alternative form of elliptic curve equation in projective coordinates so that its addition law avoids modular inverses, while still keeping the number of required multiplications low. His curves are of form

$$BY^2Z = X(X^2 + AXZ + Z^2), \tag{4.3}$$

with points $(X : Y : Z) \in K^3$ satisfying (4.3), where X, Y, Z are not all zero. Two points are identical if $(X_2 : Y_2 : Z_2) = (kX_1 : kY_1 : kZ_1)$ for some $k \in K$, $k \neq 0$. The point at infinity is $\mathcal{O} = (0 : 1 : 0)$.

Not all elliptic curves over finite fields can be brought into form (4.3), but we may restrict our ECM implementation to use only these curves. Montgomery describes an addition law for curves of this form. Given two distinct points P_1 and P_2, we can compute the X and Z-coordinates of $P_1 + P_2$ from the X and Z-coordinates of P_1, P_2 and $P_1 - P_2$. Similarly, we can compute the X and Z-coordinates of $2P$ from the X and Z-coordinates of P and the curve parameters. Surprisingly, the Y-coordinate is not needed in these computations, and can be ignored entirely when using curves in Montgomery form for ECM, and points are commonly written as only $(X :: Z)$ with Y-coordinate omitted. The details of the addition law are found in [65, 10.3.1] or [67, 2.3].

This addition law requires that in order to form the sum of two points, their difference is known or zero. This is reminiscent of the P+1 method where we need $V_{m-n}(x)$ to compute $V_{m+n}(x)$ from $V_m(x)$ and $V_n(x)$, and the same Lucas chains used to compute $V_k(x)$ for integer k in P+1 can be used to compute the multiple kP of a point P on a curve in Montgomery form in ECM.

4.6.1 ECM Stage 1

In stage 1 of ECM, we choose a suitable curve E of form (4.3) defined over $\mathbb{Z}/N\mathbb{Z}$, where N is the integer we wish to factor. Naturally N is composite, so $\mathbb{Z}/N\mathbb{Z}$ is a ring but not a field, but this has little consequence for the arithmetic of the curve as the only operation that could fail is inversion of a ring element, and an unsuccessful inversion of a non-zero element in $\mathbb{Z}/N\mathbb{Z}$ reveals a proper factor of N which is the exact purpose of ECM. We often consider the curve E_p for a prime $p \mid N$, which is the curve E reduced modulo p, i.e., E over the field $\mathbb{F}_p$.

We then choose a point P_0 on E and compute $P_1 = e \cdot P_0$ for a highly composite integer e, usually taken to be divisible by all primes and prime powers up to a suitably chosen value B_1, i.e., $e = \text{lcm}(1, 2, 3, 4, \ldots, B_1)$. We hope that for some prime factor p of N, the order of P_0 on E_p is B_1-smooth (and thus divides e), since then the point P_1 on E_p will be the point at infinity $(0 :: 0)$ so that P_1 has Z-coordinate $0 \pmod p$ and $p \mid \gcd(P_Z, N)$.

To find a point P_0 on E over $\mathbb{Z}/N\mathbb{Z}$, we choose a point of E over $\mathbb{Q}$ and map it to $\mathbb{Z}/N\mathbb{Z}$. The point over $\mathbb{Q}$ must not be a torsion point, or P_0 will have identical order on E_p for all $p \mid N$ (unless p divides the order of P_0 or the discriminant of E, which is unlikely except for small primes) so that P_1 is the point at infinity either for all E_p or for none, producing only the trivial factorizations N or 1.

By careful selection of the curve E we can ensure that number of points of E_p is a multiple of 12 or 16, significantly increasing the probability that the order of P_0 is smooth. The choice of E is described in Section 4.6.2.

The computation of $P_1 = e \cdot P_0$ on E is carried out by use of precomputed Lucas chains, similarly as in the P+1 algorithm. The selection of near-optimal Lucas chains for ECM is described in Section 4.6.3.

If stage 1 of ECM is unsuccessful, we try stage 2 where we hope to find a prime p such that the order of P_0 on E_p factors into primes and prime powers up to B_1, except for one bigger (but not too much bigger) prime q. Our stage 2 is very similar for P–1, P+1, and ECM and is described in Section 4.7.

4.6.2 Choice of Curve

In a letter to Richard Brent, Hiromi Suyama [95] showed that curves of form (4.3) over $\mathbb{F}_p$ always have group order divisible by 4, and also showed a parametrization that ensures that the group order is divisible by 12, which Brent describes in [16]. This parametrization generates an infinite family of curves over $\mathbb{Q}$ which can be used to generate a large number of distinct curves modulo N. For a given integer parameter $\sigma \neq 0, 1, 3, 5$, let

$$\begin{aligned} &u = \sigma^2 - 5, v = 4\sigma, \\ &X_0 = u^3, Z_0 = v^3 \text{ and } A = \tfrac{(v-u)^3(3u+v)}{4u^3v} - 2. \end{aligned} \tag{4.4}$$

Then the point $(X_0 :: Z_0)$ is on the curve (4.3) with parameter A. The same parametrization is used by GMP-ECM [103, 1] and Prime95 [102].

Montgomery showed in his thesis [67] how to choose curves of form (4.3) such that the curve over $\mathbb{Q}$ has a torsion subgroup of order 12 or 16, leading to group order divisible by 12 or 16, respectively, when the curve is mapped to $\mathbb{F}_p$ for almost all p.

For curves with rational torsion group of order 12 he uses

$$\begin{aligned} &t^2 = \tfrac{u^2-12}{4u}, a = \tfrac{t^2-1}{t^2+3} \\ &X_0 = 3a^2 + 1, Z_0 = 4a \text{ and } A = \tfrac{-3a^4-6a^2+1}{4a^3}, \end{aligned} \tag{4.5}$$

where $u^3 - 12u$ is a rational square. The solutions of $v^2 = u^3 - 12u$ form an elliptic curve of rank 1 and 2-torsion over $\mathbb{Q}$, with generator $(-2, 4)$ and 2-torsion point $(0, 0)$. However, adding the torsion point or not seems to produce isomorphic curves for ECM, so we ignore it. Hence for a given integer parameter $k > 1$ we can compute suitable values of u and v by computing $k \cdot (-2, 4)$ on $v^2 = u^3 - 12u$. We can then let $t = v/(2u)$. This produces an infinite family of curves over $\mathbb{Q}$.

Curves with torsion 16 and positive rank over $\mathbb{Q}$ are more difficult to generate, see [67, 6.2] for details. We currently implement only one such curve with $X_0 = 8$, $Z_0 = 15$, and $A = 54721/14400$.

These parametrizations ensure that the group order is divisible by 12 or 16, respectively, but the resulting group order of the curve over $\mathbb{F}_p$ does not behave like an integer chosen uniformly at random from the integers that are multiples of 12 or 16, respectively, in the Hasse interval around p. In particular, the average valuation of 2 in the group order for curves with rational torsion 12 is $11/3$, slightly higher than $10/3$ for curves in Brent-Suyama parametrization (which have rational torsion 6), making them somewhat more likely to find factors. The divisibility properties will be examined in more detail in Chapter 5.

Very small σ-values for the Brent-Suyama parametrization lead to curves with simple rationals for the point coordinate and curve parameter, and very small k-values for Montgomery's parametrization for curves with rational torsion 12 lead to simple rationals for a, see Table 4.5. These rationals can be mapped to $\mathbb{Z}/N\mathbb{Z}$ easily, as the denominators are highly composite integers so that the required divisions modulo N can be done by the methods of Section 4.3.4 and a few multiplications.

When factoring cofactors after the sieving step of NFS into large primes, only very few curves are required on average since the primes to be found are relatively small, and with an early-abort strategy, only the first few curves work on larger composites where arithmetic is more expensive. In spite of the small number of curves with such simple rationals as curve parameters, it is useful to implement them as special cases.

σ	X_0	Z_0	A
2	-1	512	$-3645/32$
4	1331	4096	$6125/85184$

k	a	X_0	Z_0	A
2	$-3/13$	$196/169$	$-12/13$	$-4798/351$
3	$28/37$	$3721/1369$	$112/37$	$-6409583/3248896$

Table 4.5: Some elliptic curves chosen by the Brent-Suyama parametrization with group order divisible by 12, and by Montgomery's parametrization with rational torsion group of order 12.

4.6.3 Lucas Chains for ECM

In principle, Lucas chains for ECM can be chosen exactly as for P+1. However, a subtle difference exists: in P+1, the cost of a doubling $V_{2n}(x) = V_n(x)^2 - 2$ is identical to that of an addition $V_{m+n}(x) = V_m(x)V_n(x) - V_{m-n}$ if V_{m-n} is known and a squaring is taken to have the same cost as a multiplication. This way, the cost of a Lucas chain depends only on its length.

In ECM, the cost of a point doubling usually differs from the cost of an addition of distinct points. In the addition rules given by Montgomery, a doubling takes 5 modular multiplications of which 2 are squarings, whereas an addition of distinct points takes 6 modular multiplications of which again 2 are squarings.

These different costs can be taken into account when choosing Lucas chains. For example, to multiply a point by 7, we can choose between the chains $1, 2, 3, 5, 7$ or $1, 2, 3, 4, 7$ of equal length. In the former, all additions except for the initial doubling $1, 2$ are additions of distinct values. In the latter, we can produce 4 by doubling 2, so that this Lucas chain would save 1 modular multiplication in the elliptic curve arithmetic.

When generating Lucas chains with PRAC using several multipliers, we can choose the best chain not according to its length but by the cost of the arithmetic performed in each PRAC rule that is used to build the chain.

The speedup in practice is relatively small: with two-word modulus, ECM stage 1 with $B_1 = 500$ is about 1% faster when counting the cost of a doubling as 5/6 of the count of an addition when choosing Lucas chains. Still, this improvement is so simple to implement that it may be considered worthwhile.

As for P+1, the precomputed addition chains are stored as byte code that describes a sequence of PRAC rules to apply. Code compression may be used to reduce the overhead in the byte code interpreter, but since the elliptic curve arithmetic is more expensive than in the case of P+1, the relative speedup gained by compression is much smaller.

4.6.4 ECM Stage 1 Performance

Table 4.6 compares the performance of the ECM stage 1 implementation for different B_1 values and modulus sizes on AMD Phenom and Intel Core 2 CPUs.

	Core 2		Phenom	
B_1	1 word	2 words −2 bits	1 word	2 words −2 bits
100	11.8	35.6	9.33	24.4
200	24.5	77.9	19.4	52.6
300	35.3	113	27.8	76.0
400	46.7	151	36.6	101
500	58.7	190	46.2	127
600	69.6	226	54.6	151
700	82.3	266	64.5	178
800	93.6	302	72.4	202
900	105	342	82.5	229
1000	117	381	92.0	255

Table 4.6: Time in microseconds for ECM stage 1 with different B_1 values on 2.146 GHz Core 2 and 2 GHz AMD Phenom CPUs

4.7 Stage 2 for P–1, P+1, and ECM

Stage 1 of P–1, P+1, and ECM all compute an element g_0^e of some (multiplicatively written) group G for a highly composite integer e, typically chosen as $e = \mathrm{lcm}(1, 2, 3, 4, \ldots, B_1)$ for some integer B_1. If the order of g_0 is B_1-smooth, then $g_1 = g_0^e$ is the identity in G. Since G is defined over $\mathbb{F}_p$ where p divides N, the number to factor, we can construct from the identity in G a residue $r \pmod{N}$ such that $r \equiv 0 \pmod{p}$ but hopefully not $r \equiv 0 \pmod{N}$, and then $\gcd(r, N)$ usually reveals p. If the order of g_0 is not B_1-smooth, stage 1 fails to find p. However, we may be able to find it yet if the order of g_0 consists of a B_1-smooth part times a not-too-large prime q.

Stage 2 of P–1, P+1, and ECM tries to find the value of q efficiently on the assumption that q is prime and not very large, although larger than B_1, by looking for a match $g_1^m = g_1^n$ which occurs when $q \mid m - n$. We will describe the stage 2 for the P+1 algorithm; P–1 can use the same algorithm by adjusting its stage 1 output, and the stage 2 for ECM is structurally very similar. Differences between the P+1 and ECM stage 2 are noted.

Our stage 2 is modeled after the enhanced standard continuation described by Montgomery [65]. For given search limits B_1 and B_2 and input X_1 it chooses a value d with $6 \mid d$ and computes two lists

$$\begin{aligned} f_i &= V_{id}(X_1) \bmod N \text{ for } \lfloor B_1/d \rceil \leq i \leq \lfloor B_2/d \rceil \text{ and} && (4.6)\\ g_j &= V_j(X_1) \bmod N \text{ for } 1 \leq j < d/2 \text{ and } j \perp d, && (4.7)\end{aligned}$$

so that all primes q in $]B_1, B_2]$ can be written as $q = id - j$ or $q = id + j$.

Let $X_1 \equiv \alpha_1 + 1/\alpha_1 \pmod N$, where α_1 may be in a quadratic extension of $\mathbb{Z}/N\mathbb{Z}$, and assume

$$\alpha_1^q \equiv 1 \pmod p \tag{4.8}$$

for some unknown prime p, $p \mid N$ and a prime q, $B_1 < q \leq B_2$. Let $q = id - j$ or $q = id + j$. Then, using $V_{-n}(X) = V_n(X)$, we have

$$\begin{aligned} V_{id}(X_1) \equiv V_{q\pm j}(X_1) &\equiv \alpha_1^{q\pm j} + 1/\alpha_1^{q\pm j}\\ &\equiv \alpha_1^{\pm j} + 1/\alpha_1^{\pm j} \equiv V_{\pm j}(X_1) \equiv V_j(X_1) \pmod p\end{aligned}$$

and so

$$V_{id}(X_1) - V_j(X_1) \equiv 0 \pmod p. \tag{4.9}$$

After the lists f_i, g_j are computed, we can collect the product

$$A = \prod_{\substack{id\pm j=q\\ B_1<q\leq B_2}} (f_i - g_j) \bmod N. \tag{4.10}$$

If there is a prime q in $]B_1, B_2]$ such that (4.8) holds, the product (4.10) will include i, j such that (4.9) holds, and thus $p \mid \gcd(A, N)$.

Stage 1 of P–1 computes $x_1 = x_0^e \pmod N$ and we can set $X_1 = x_1 + 1/x_1$ to make the P–1 stage 1 output compatible with our stage 2 at the cost of one modular inverse. Stage 1 of P+1 computes $x_1 = V_e(x_0) = V_e(\alpha_0 + 1/\alpha_0) = \alpha_0^e + 1/\alpha_0^e$ and we may simply set $X_1 = x_1$.

For P–1 stage 2, we could also use $f_i = x_1^{id} \bmod N$ and $g_j = x_1^j \bmod N$, for $1 \leq j < d$ and $j \perp d$, instead of (4.6). An advantage of using (4.6) is that $V_{-n}(X) = V_n(X)$, so that $g_j = V_j(X) \bmod N$ needs to be computed only for $1 \leq j < d/2$, and one (i, j)-pair can sometimes include two primes at once. The same could be achieved by using $f_i = x_1^{(id)^2}$ and $g_j = x_1^{j^2}$, but computing these values for successive i or j via $(x^{(n+1)^2}, x^{2(n+1)+1}) = (x^{n^2} \cdot x^{2n+1}, x^{2n+1} \cdot x^2)$ costs two multiplications, whereas $V_{n+1}(x) = V_n(x)V_1(x) - V_{n-1}(x)$ costs only one. However, a modular inverse is required to convert the P–1 stage 1 output into the required form. Which approach is better thus depends on the choice of stage 2 parameters, i.e., on how many values need to be precomputed for the f_i and g_j lists. Assuming a small B_1, when using $B2 \approx 5000$ and $d = 210$, we need about 24 values for f_i and another 24 for g_j. The cost of a modular inverse is roughly 50 times the cost of a modular multiplication in our implementation, so the two approaches are about equally fast. Using the same stage 2 for P–1 and P+1 has the advantage of requiring only one implementation for both methods.

For ECM, we again would like two lists f_i and g_j such that $f_i \equiv g_j \pmod p$ if $id \cdot P_1 = j \cdot P_1$ on E_p, where P_1 is the point that was output by ECM stage 1. We can use $f_i = (id \cdot P_1)_X$, the X-coordinate of $id \cdot P_1$, and $g_j = (jP_1)_X$. A point and its inverse have the same X-coordinate on curves in Weierstraß and Montgomery form, so again we have $f_i - g_j \equiv 0 \pmod p$ if $q \mid id \pm j$.

With points in projective coordinates, the points need to be canonicalized first to ensure that identical points have identical X-coordinates, which is described in Section 4.7.2.

How to choose the parameter d and the set (i, j)-pairs needed during stage 2 for given B_1 and B_2 values is described in Section 4.7.1. Section 4.7.2 shows how to compute the lists f_i and g_j efficiently, and Section 4.7.3 describes how to accumulate the product (4.10).

4.7.1 Generating Plans

The choice of d, the sets of i and j values to use for generating f_i and g_j, respectively, and the set of (i, j)-pairs for which to accumulate the product of $f_i - g_j$ depend on the B_1 and B_2 parameters for stage 2, but are independent of N, the number to factor. These choices are precomputed for given B_1 and B_2 and are stored as a "stage 2 plan." The stage 2 implementation then carries out the operations described by the plan, using arithmetic modulo N.

The plan provides the values d, i_0, i_1, a set S and a set T, chosen so that all primes q in $]B_1, B_2]$ appear as $q = id \pm j$ for some $(i, j) \in T$ with $i_0 \leq i \leq i_1$ and $j \in S$.

We try to choose parameters that minimize the number of group operations required for building the lists f_i and g_j and minimize the number of (i, j)-pairs required to cover all primes in the $]B_1, B_2]$ interval. This means that we would like to maximise i_0, minimize i_1, and cover two primes in $]B_1, B_2]$ with a single (i, j)-pair wherever possible.

We choose d highly composite and $S = \{1 \leq j < d/2, j \perp d\}$, so that all integers coprime to d, in particular all primes not dividing d, can be written as $id \pm j$ for some integer i and $j \in S$. We assume B_1 is large enough that no prime greater than B_1 divides d. Choosing values of $i_0 = \lfloor B_1/d \rceil$ and $i_1 = \lfloor B_2/d \rceil$ is sufficient, but may be improved as shown below.

Computing the lists f_i and g_j requires at least one group operation per list entry, which is expensive especially in the case of ECM. The list f_i has $i_1 - i_0 + 1$ entries where $i_1 - i_0 \approx (B_2 - B_1)/d$, and g_j has $\phi(d)/2$ entries, so we choose d highly composite to achieve small $\phi(d)$ and try to minimize $i_1 - i_0 + 1 + \phi(d)/2$ by ensuring that $i_1 - i_0 + 1$ and $\phi(d)/2$ are about equally large. In our application of finding primes up to, say, 2^{32} as limited by the large prime bound used in the NFS sieving step, the value of B_2 will usually be of the order of a few thousand, and a choice $d = 210$ works well in this case. With $B_2 = 5000$, $i_1 = 24$ and $|S| = 24$, so the two lists of f_i and g_j are about equally large, assuming small i_0. For smaller B_2, a smaller d is preferable, for example $d = 90$ for $B_1 = 100$, $B_2 = 1000$.

We have chosen i_1 as an upper bound based on B_2, but we may reduce i_1 yet if $[i_1 d - d/2, i_1 d + d/2]$ does not include any primes up to B_2, and so obtain the final value of i_1.

Having chosen d, S, and i_1, we can choose T. We say a prime $q \in]B_1, B_2]$ is covered by an (i, j)-pair if $q \mid id \pm j$; assuming that only the largest prime factor of any $id \pm j$ value lies in $]B_1, B_2]$, each pair may cover up to two primes. For each prime $q \in]B_1, B_2]$ we mark the corresponding entry $a[q]$ in an array to signify a prime that yet needs to be covered.

Let r be the smallest prime not dividing d. Then $q \mid id \pm j$ and $q \neq id \pm j$ implies $q = (id \pm j)/s$ with $s \geq r$ since $id \pm j \perp d$, thus $q \leq (id \pm j)/r$. Hence composite values of $id \pm j$ with $i \leq i_1$ can cover only primes up to $\lfloor (i_1 d + d/2)/r \rfloor$, and each prime $q > \lfloor (i_1 d + d/2)/r \rfloor$ can be covered only by $q = id \pm j$.

In a first pass, we examine each prime q, $\lfloor (i_1 d + d/2)/r \rfloor < q \leq B_2$, highest to lowest and the (i, j)-pair covering this prime. This pair is the only way to cover q and must eventually be included in T. If this (i, j)-pair also covers a smaller prime q' as a composite value, then $a[q']$ is un-marked.

In a second pass, we look for additional (i, j)-pairs that cover two primes, both via composite values. We examine each (i, j)-pair with $i_0 \leq i \leq i_1$ highest to lowest, and $j \in S$. If there are two primes q' and q'' marked in the array that are covered by the (i, j)-pair under examination, then $a[q']$ and $a[q'']$ are un-marked, and $a[id - j]$ is marked instead.

In the third pass, we cover the remaining primes $q \leq \lfloor (i_1 d + d/2)/r \rfloor$ using (i, j)-pairs with large i, if possible, hoping that we may increase the final i_0 value. As in the second pass, we examine each (i, j)-pair in order of decreasing i and, if there is a prime q' with $a[q']$ marked, $q' \mid id \pm j$ but $q' \neq id \pm j$, we un-mark $a[q']$ and mark $a[id - j]$ instead. This way, all primes in $]B_1, B_2]$ are covered, and each with an (i, j)-pair with the largest possible $i \leq i_1$.

We now choose the final i_0 value by looking for the smallest (not necessarily prime) q such that a $a[q]$ is marked, and setting $i_0 = \lfloor q/d \rceil$. The set T is determined by including each (i, j)-pair where an array element $a[id - j]$ or $a[id + j]$ is marked. The pairs in T are stored in order of increasing i so that the f_i can be computed sequentially for P–1 and P+1.

4.7.2 Initialisation

In the initialisation phase of stage 2 for P–1 and P+1 (and similarly for ECM), we compute the values $g_j = V_j(X_1)$ with $1 \leq j < d/2$, $j \perp d$ and set up the computation of $f_i = V_{id}(X_1)$ for $i_0 \leq i \leq i_1$. To do so, we need Lucas chains that generate all required values of id and j. We try to find a short Lucas chain that produces all required values to save group operations which are costly especially for ECM.

Lucas chains for values in an arithmetic progression are particularly simple, since the difference of successive terms is constant. We merely need to start the chain with terms that generate the common difference and the first two terms of the arithmetic progression.

The values of j with $j \perp d$ and $6 \mid d$ can be computed in two arithmetic progressions $1 + 6m$ and $5 + 6m$, via the Lucas chain $1, 2, 3, 5, 6, 7, 11, 13, 17, 19, 23, \ldots$ For $d = 210$, the required 24 values of j can therefore be generated with a Lucas chain of length 37.

To add the values of id with $i_0 \leq i \leq i_1$, we need to add d, $i_0 d$, and $(i_0 + 1)d$ to the chain. If $2 \mid\mid d$, we have $d/2 - 2 \perp d$ and $d/2 + 2 \perp d$ and we can add d to the Lucas chain by including $4 = 2 + 2$ and $d = d/2 + 2 + d/2 - 2$. If $4 \mid d$, we have $d/2 - 1 \perp d$ and $d/2 + 1 \perp d$ and we can add d simply via $d = d/2 + 1 + d/2 + 1$ as 2 is already in the chain. Since i_0 is usually small, we can compute both $i_0 d$ and $(i_0 + 1)d$ from d with one binary chain.

Using this Lucas chain, we can compute and store all the $g_j = V_j(X_1)$ residues as well as $V_d(X_1)$, $f_{i_0 d}(X_1)$, and $f_{(i_0+1)d}(X_1)$.

In the case of P–1 and P+1, since the (i, j)-pairs are stored in order of increasing i, all the f_i values need not be computed in advance, but can be computed successively as the (i, j)-pairs are processed.

Initialisation for ECM

For ECM stage 2 we use curves in Montgomery form with projective coordinates, just as in stage 1, to avoid costly modular inverses. The initialisation uses the same Lucas chain as in 4.7.2 for the required values of id and j, so that $id \cdot P_1$ and $j \cdot P_1$ can be computed efficiently. However, two points $(X_1 :: Z_1)$ and $(X_2 :: Z_2)$ in projective coordinates being identical does not imply $X_1 = X_2$, but $X_1/Z_1 = X_2/Z_2$, where Z_1 and Z_2 are generally not equal, so the X-coordinates of these points cannot be used directly to build the lists f_i and g_j.

There are several ways to overcome this obstruction. Several authors (for example [33, 7.4.2] or [78]) propose storing both X and Z coordinate in the f_i and g_j lists, and then accumulating the product $A = \prod_{(i,j) \in T} ((f_i)_X (g_j)_Z - (g_j)_X (f_i)_Z)$. An advantage of this approach is that the f_i can be computed sequentially while the product is being accumulated and the number of g_j

to precompute and store can be controlled by choice of d, which allows ECM stage 2 to run under extremely tight memory conditions such as in an FPGA implementation. The obvious disadvantage is that each (i, j)-pair now uses 3 modular multiplications instead of 1 as in (4.10).

Another approach and much preferable in an implementation where sufficient memory is available is canonicalizing the precomputed points so that all points have the same Z-coordinate. To produce the desired lists f_i and g_j, we therefore compute all the required points $Q_i = id \cdot P_1$ and $R_j = j \cdot P_1$. If all Z-coordinates of Q_i and R_j are made identical, $Q_i = R_j$ on E_p implies $(Q_i)_X \equiv (R_j)_X \pmod{p}$, as desired, and we may set $f_i = (Q_i)_X$ and $g_j = (R_j)_X$.

We suggest two methods for this. One is to set all Z-coordinates to 1 (mod N) via $(X :: Z) = (XZ^{-1} :: 1)$. To do so, we need the inverse modulo N of each Z-coordinate of our precomputed points. A trick due to Montgomery, described for example in [25, 10.3.4], replaces n modular inverses of residues r_k modulo N, $1 \leq k \leq n$, by 1 modular inverse and $3n - 3$ modular multiplications. This way we can canonicalize a batch of n points with $4n - 3$ modular multiplications and 1 modular inverse. Not all points needed for the f_i and g_j lists need to be processed in a single batch; if memory is insufficient, the points needed for f_i can be processed in several batches while product (4.10) is being accumulated.

A faster method was suggested by P. Zimmermann. Given $n \geq 2$ points $P_1, \ldots, P_n$, $P_i = (X_i :: Z_i)$, we set all Z-coordinates to $\prod_{1 \leq i \leq n} Z_i$ by multiplying each X_k by $T_k = \prod_{1 \leq i \leq n, i \neq k} Z_i$. This can be done efficiently by computing two lists $s_k = \prod_{1 \leq i \leq k} Z_i$ and $t_k = \prod_{k < i \leq n} Z_i$ for $1 \leq k < n$, each at the cost of $n-2$ modular multiplications. Now we can set $T_1 = t_1, T_n = s_{n-1}$, and $T_i = s_{i-1} t_i$ for $1 < i < n$, also at the cost of $n-2$ multiplications. Multiplying X_i by T_i costs another n modular multiplications for a total of only $4n - 6$ modular multiplications, without any modular inversion. Algorithm 9 implements this idea. Since the common Z-coordinate of the canonicalized points is the product of all points idP_1 and jP_1, the complete set of points needed for the f_i and g_j lists must be processed in a single batch.

Interestingly, if the curve parameters are chosen such that the curve initialisation can be done with modular division by small constants rather than with a modular inverse, then ECM implemented this way does not use any modular inverses at all, without sacrificing the optimal cost of 1 modular multiplication per (i, j)-pair in stage 2.

```
Input: $n \geq 2$, an integer
       $N$, a positive integer
       $Z_1, \ldots, Z_n$, residues modulo $N$
Data: $s$, a residue modulo $N$
Output: $T_1, \ldots, T_n$, residues modulo $N$ with $T_i \equiv \prod_{1 \leq i \leq n, i \neq k} Z_i \pmod{N}$
$T_{n-1} := Z_n$;
for $k := n - 1$ downto 2 do
    $T_{k-1} := T_k \cdot Z_k \bmod N$;
$s := Z_1$;
$T_2 := T_2 \cdot s \bmod N$;
for $k := 3$ to $n$ do
    $s := s \cdot Z_{k-1} \bmod N$;
    $T_k := T_k \cdot s \bmod N$;
```

Algorithm 9: Batch cross multiplication algorithm.

4.7.3 Executing Plans

The stage 2 plan stores the (i, j)-pairs which cover all primes in $]B1, B2]$. The f_i and g_j lists are computed as described in 4.7.2. Stage 2 then reads the stored (i, j)-pairs, and multiplies $f_i - g_j$ to an accumulator:

$$A = \prod_{(i,j)\in T} (f_i - g_j) \bmod N. \tag{4.11}$$

Since the pairs are stored in order of increasing i, the full list f_i need not be precomputed for P–1 and P+1, but each f_i can be computed sequentially by $V_{(i+1)d}(X_1) = V_{id}(X_1)V_d(X_1) - V_{(i-1)d}(X_1)$. At the end of stage 2, we take $r = \gcd(A, N)$, hoping that $1 < r < N$ and so that r is a proper factor of N.

Backtracking

We would like to avoid finding all prime factors of the input number N simultaneously, i.e., finding N as a trivial factor. As in stage 1 of P–1 and P+1, a backtracking mechanism is used to recover from such a situation.

Since $r = \gcd(A, N)$ and A is a reduced residue modulo N, we find $r = N$ as a factor if and only if $A = 0$. We set up a "backup" residue $A' = 1$ at the start of evaluation of (4.11). At periodic intervals during the evaluation of (4.11), for example each time that i is increased, we test if $A = 0$, which is easy since the residue does not need to be converted out of Montgomery representation if REDC (see Section 4.3.2) is used for the arithmetic. If $A = 0$, we take $r = \gcd(A', N)$ and end stage 2. Otherwise, we set $A' = A$. This way, a proper factor of N can be discovered so long as all prime factors of N are not found between two tests for $A = 0$.

4.7.4 P+1 and ECM stage 2 Performance

Tables 4.7 and 4.8 compares the performance of the P+1 and the ECM stage 2 implementation for different B_2 values and modulus sizes on AMD Phenom and Intel Core 2 CPUs. In each case, the timing run used $B_1 = 10$ and $d = 210$, and the time for a run with $B_1 = 10$ and without any stage 2 was subtracted.

4.7.5 Overall Performance of P–1, P+1 and ECM

Tables 4.9 and 4.10 shows the expected time to find primes close to $2^{25}, 2^{26}, \ldots, 2^{32}$ for moduli of 1 word and of 2 words, and the B_1 and B_2 parameters chosen to minimize the expected time. The empirically determined probability estimate is based on the elliptic curve with rational 12 torsion and parameter $k = 2$ in Section 4.6.2. That the B_1 and B_2 parameters are not monotonously increasing with factor size is due to the fact that the expected time to find a prime factor as a function of B_1 and B_2 is very flat around the minimum, so that even small perturbations of the timings noticeably affect the parameters chosen as optimal.

4.8 Comparison to Hardware Implementations of ECM

Several hardware implementations of ECM for use as a cofactorization device in NFS have been described recently, based on the proposed design "SHARK" by Franke et al. [41]. SHARK is a hardware implementation of GNFS for factoring 1024-bit integers which uses ECM for cofactorization after sieving. The idea of implementing GNFS in hardware is inspired by the

	Core 2		Phenom	
B_2	1 word	2 words −2 bits	1 word	2 words −2 bits
1000	3.06	6.72	2.91	6.24
2000	4.09	9.86	3.64	8.08
3000	5.07	12.7	4.37	10.1
4000	6.00	15.5	5.01	11.8
5000	6.95	18.2	5.77	13.8
6000	7.80	20.8	6.40	15.4
7000	8.83	23.7	7.09	17.3
8000	9.69	26.3	7.73	19.0
9000	10.7	29.0	8.39	20.7
10000	11.5	31.4	9.01	22.5
20000	20.3	57.0	15.3	39.3
30000	28.9	81.8	21.3	55.0
40000	37.4	106	27.2	70.8
50000	45.7	130	33.1	86.2
60000	54.1	154	38.8	102

Table 4.7: Time in microseconds for P+1 stage 2 with different B_2 values on 2.146 GHz Intel Core 2 and 2 GHz AMD Phenom CPUs

observation of Bernstein [7] that dedicated hardware could achieve a significantly lower cost in terms of Area-Time product than a software implementation that uses sieving on a regular PC. He proposes, among other algorithms, to use ECM for the smoothness test.

Pelzl et al. [78] present a scalable implementation of ECM stage 1 and stage 2 for input numbers of up to 200 bits, based on Xilinx Virtex2000E-6 FPGAs with an external microcontroller. Their design has one modular multiplication unit per ECM unit, and each ECM unit performs both stage 1 and stage 2. They propose using the bounds $B_1 = 910$ and $B_2 = 57000$ for finding primes of up to about 40 bits. They use curves in Montgomery form (4.3) and a binary Lucas chain for stage 1 that uses 13740 modular multiplications (including squarings), and estimate that an optimized Lucas chain could do it in $\approx$ 12000 modular multiplications. They use an enhanced standard stage 2 that uses 3 modular multiplications per (i, j)-pair, see 4.7.2. With a value $d = 210$, they estimate 303 point additions and 14 point doublings in the initialisation of stage 2, and 13038 modular multiplications for collecting the product (4.10) with 4346 different (i, j)-pairs for a total of 14926 modular multiplications in stage 2. However, to minimize the AT product, they propose using $d = 30$ with a total of 24926 modular multiplications in stage 2.

In our implementation, stage 1 with $B_1 = 910$ and PRAC-generated chains (using cost 6 for point addition, 5 for doubling, 0.5 for each byte code and 0.5 for each byte code change as parameters for rating candidate chains) uses 11403 modular multiplications, 83% of the figure for the binary Lucas chain. (Using chains for composite values where the resulting chain is shorter than the concatenated chains for the factors is not currently used and could probably reduce this figure by a few more percent.) Our stage 2 with $B_2 = 57000$ and $d = 210$ uses 290 point additions, 13 point doublings, 1078 modular multiplications for point canonicalization and 4101 pairs which cost 1 modular multiplication each, for a total of 6945 modular multiplications. The cost of computing and canonicalizing the points idP_1 has a relatively large share in this figure, suggesting that a value for d such that $B_2/(d\phi(d))$ is closer to 1 might reduce the total multiplication count. In a hardware implementation, the extra memory requirements may make larger d values inefficient in terms of the AT product, but this is not an issue in a software

	Core 2		Phenom	
B_2	1 word	2 words −2 bits	1 word	2 words −2 bits
1000	5.86	17.2	7.10	17.5
2000	7.46	21.5	7.87	19.7
3000	8.83	25.4	8.79	22.0
4000	10.1	29.7	9.55	24.1
5000	11.5	33.7	10.5	26.5
6000	12.7	37.6	11.2	28.2
7000	14.0	41.4	12.1	30.8
8000	15.4	45.8	12.9	32.7
9000	16.7	49.6	13.7	34.6
10000	17.9	53.4	14.5	36.9
20000	30.5	91.3	22.3	56.6
30000	42.8	128	29.7	75.0
40000	54.9	164	37.2	94.3
50000	66.7	200	44.5	113
60000	78.3	235	51.8	131

Table 4.8: Time in microseconds for ECM stage 2 with different B_2 values on 2.146 GHz Intel Core 2 and 2 GHz AMD Phenom CPUs

n	B_1	B_2	Prob.	1 word	2 words −2 bits
25	300	5000	0.249	46	103
26	310	6000	0.220	55	125
27	320	6000	0.186	67	151
28	400	6000	0.167	81	182
29	430	7000	0.149	100	224
30	530	11000	0.158	119	275
31	530	10000	0.128	144	330
32	540	10000	0.105	177	410

Table 4.9: Expected time in microseconds and probability to find prime factors close to 2^n of composites with 1 or 2 words with P–1 on 2 GHz AMD Phenom CPUs. The B_1 and B_2 parameters are chosen empirically to minimize the time/probability ratio.

implementation on a normal PC. In our implementation, $d = 630$ provides the minimum total number of 5937 modular multiplications in stage 2, only 40% of the number reported by Pelzl et al. for $d = 210$, and only 24% of their number for $d = 30$.

These figures suggest that a software implementation of ECM on a normal PC enjoys an advantage over an implementation in embedded hardware by having sufficient memory available that choice of algorithms and of parameters are not constrained by memory, which significantly reduces the number of modular multiplications in stage 2. This problem might be reduced by separating the implementation of stage 1 and stage 2 in hardware, so that each stage 1 units needs only very little memory and forwards its output to a stage 2 unit which has enough memory to compute stage 2 with a small multiplication count, while the stage 1 unit processes the next input number.

Gaj et al. [44] improve on the design by Pelzl et al. mainly by use of a more efficient implementation of modular multiplication, by avoiding limitations due to the on-chip block RAM

n	B_1	B_2	Prob.	1 word	2 words −2 bits
25	130	7000	0.359	67	176
26	130	7000	0.297	81	213
27	150	11000	0.290	101	264
28	160	13000	0.256	124	324
29	180	12000	0.220	151	395
30	200	12000	0.188	190	496
31	260	14000	0.182	231	604
32	250	15000	0.147	283	744

Table 4.10: Expected time in microseconds and probability per curve to find prime factors close to 2^n of composites with 1 or 2 words with ECM on 2 GHz AMD Phenom CPUs. The B_1 and B_2 parameters are chosen empirically to minimize the expected time.

which allows them to fit more ECM units per FPGA, and removing the need for an external microcontroller. The algorithm of ECM stage 1 and stage 2 is essentially the same as that of Pelzl et al. They report an optimal performance/cost ratio of 311 ECM runs per second per $100 for an input number of up to 198 bits with $B_1 = 910, B_2 = 57000, d = 210$, using an inexpensive Spartan 3E XC3S1600E-5 FPGA for their implementation. They also compare their implementation to an ECM implementation in software, GMP-ECM [39], running on a Pentium 4, and conclude that their design on a low-cost Spartan 3 FPGA offers about 10 times better performance/cost ratio than GMP-ECM on a Pentium 4. However, GMP-ECM is a poor candidate for assessing the performance of ECM in software for very small numbers with low B_1 and B_2 values. GMP-ECM is optimized for searching large prime factors (as large as reasonably possible with ECM) of numbers of at least a hundred digits size by use of asymptotically fast algorithms in particular in stage 2, see [103]. For very small input, the function call and loop overhead in modular arithmetic and the cost of generating Lucas chains on-the-fly in stage 1 dominates the execution time; likewise in stage 2, the initialisation of the polynomial multi-point evaluation and again function call and loop overhead will dominate, while the B_2 value is far too small to let the asymptotically fast stage 2 (with time in $\tilde{O}(\sqrt{B_2})$) make up for the overhead.

De Meulenaer et al. [34] further improve the performance/cost-ratio by using a high-performance Xilinx Virtex4SX FPGA with embedded multipliers instead of implementing the modular multiplication with general-purpose logic. They implement only stage 1 of ECM and only for input of up to 135 bits. One ECM unit utilizes all multipliers of the selected FPGA, so one ECM unit fits per device. By scaling the throughput of the design of Gaj et al. to 135-bit input, they conclude that their design offers a 15.6 times better performance/cost ratio. In particular, assuming a cost of $116 per device, they state a throughput of 13793 ECM stage 1 with $B_1 = 910$ per second per $100.

We compare the cost of finding 40-bit factors using our software implementation of ECM with that given by de Meulenaer et al. Our implementation is currently limited to moduli of size 2 words with the two most significant bits zero, or 126 bits on a 64-bit system, whereas the implementation of de Meulenaer et al. allows 135-bit moduli. Extending our implementation to numbers of 3 words is in progress, but not functional at this time. We expect that ECM with 3-word moduli will take about twice as long as for 2-word moduli. For the comparison we use timings for 126-bit moduli (2 words) and estimates for 135-bit moduli (3 words).

The timings for our code are obtained using an AMD Phenom X4 9350e with four cores at 2.0 GHz. The AMD 64-bit CPUs all can perform a full 64 × 64-bit product every 2 clock cycles,

Device	XC4VSX25-10	Phenom 9350e	Phenom II X4 955
Clock rate	0.22 GHz	2.0 GHz	3.2 GHz
Cores per device	1	4	4
126-bit modulus (2 words in software)			
Time per stage 1	$62.5\mu s$	$232.1\mu s$	$\approx 145\mu s$
Time per stage 2	$59.2\mu s$	$121.5\mu s$	$\approx 76\mu s$
Time per trial	$121.7\mu s$	$353.6\mu s$	$\approx 221\mu s$
#Trials/sec/device	8217	11312	18100
Cost per device	$300		$215
#Trials/sec/$100	2739		8418
135-bit modulus (3 words in software)			
Time per stage 1	$62.5\mu s$	$\approx 464\mu s$	$\approx 290\mu s$
Time per stage 2	$59.2\mu s$	$\approx 243\mu s$	$\approx 152\mu s$
Time per trial	121.7	$\approx 707\mu s$	$\approx 442\mu s$
#Trials/sec/device	8217	≈ 5658	≈ 9052
Cost per device	$300		$215
#Trials/sec/$100	2739		4210

Table 4.11: Comparison of ECM with $B_1 = 910$, $B_2 = 57000$ for 126-bit and 135-bit input on a Virtex4SX25-10 FPGA and on AMD 64-bit microprocessors

making them an excellent platform for multi-precision modular arithmetic. The fastest AMD CPU currently available is a four-core 3.2 GHz Phenom II X4 955 at a cost of around $215 (regular retail price, according to `www.newegg.com` on July 28th 2009) and we scale the timings linearly to that clock rate. Since the code uses almost no resources outside the CPU core, linear scaling is reasonable. The number of clock cycles used is assumed identical between the Phenom and Phenom II. Similarly, running the code on n cores of a CPU is assumed to increase total throughput n-fold.

Table 4.11 compares the performance of the implementation in hardware of de Meulenaer et al. and of our software implementation, using the parameters $B_1 = 910$, $B_2 = 57000$. The software implementation uses $d = 630$ for stage 2. De Meulenaer et al. do not implement stage 2, but predict its performance as capable of $16,900$ stage 2 per second per device. We use this estimate in the comparison. They also give the cost of one Xilinx XC4VSX25-10 FPGA as $116 when buying 2500 devices. The current quote at `www.nuhorizons.com` and `www.avnet.com` for this device is about $300, however. We base the price comparison on the latter figure. Only the cost of the FPGA or the CPU, respectively, are considered. The results show that a software implementation of ECM can compete in terms of cost per ECM trial with the published designs for ECM in hardware. An advantage of the software implementation is flexibility: it can run on virtually any 64-bit PC, and so utilize otherwise idle computing resources. If new systems are purchased, they involve only standard parts that can be readily used for a wide range of computational tasks. Given a comparable performance/cost ratio, an implementation in software running on standard hardware is the more practical.

Our current implementation is sufficient for one set of parameters proposed by the SHARK [41] design for factoring 1024-bit integers by GNFS which involves the factorization of approximately $1.7 \cdot 10^{14}$ integers of up to 125 bits produced by the sieving step. The time for both stage 1 and stage 2 with $B_1 = 910, B_2 = 57000$ is $353.6\mu s$ on a 2 GHz Phenom, and about $221\mu s$

on a 3.2 GHz Phenom II. Using the latter, $1.7 \cdot 10^{14}$ ECM trials can be performed in approximately 300 CPU-years. But how many curves need to be run per input number? Pelzl et al. [78] state that 20 curves at $B_1 = 910$, $B_2 = 57000$ find a 40-bit factor with $> 80\%$ probability, and doing 20 trials per input number gives a total time of about 6000 CPU years. However, the vast majority of input numbers will not be 2^{40}-smooth, and fewer than 20 curves suffice to establish non-smoothness with high probability, making this estimate somewhat pessimistic. Assuming a cost of about $350 for a bare-bone but functional system with one AMD Phenom II X4 955 CPU, this translates to a pessimistic estimate of about $2.1M for hardware capable of performing the required ECM factorizations within a year.

Bernstein et al. [10] recently demonstrated a highly efficient implementation of ECM on graphics cards that support general-purpose programming. An NVidia GTX 295 card is reported as performing stage 1 of 4928 ECM curves with $B_1 = 8192$ and a 210-bit modulus per second, and is estimated to perform 5895 curves with the same parameters with a 196-bit modulus. Assuming a purchase price of $500 per card, this translates to 11.8 curves per Dollar and second. Our implementation of ECM takes 2087μs on a Phenom 9350e per curve (only stage 1) with the same parameters and a 126-bit modulus; with 192-bit modulus on a Phenom II X4 955 we estimate the time as approximately 2609μs, or 1533 curves per second per device, which results in approximately 7.1 curves per Dollar and second. ECM on graphics cards is therefore a serious contender for performing cofactorization in the NFS. The graphics card implementation does not include a stage 2, however, and implementing one may be difficult due to severely restricted low-latency memory in graphics processing units.

Chapter 5

Parameter selection for P–1, P+1, and ECM

5.1 Introduction

In Chapter 4 we described an efficient implementation of the P-1, P+1, and ECM factoring algorithms tailored for rapidly processing many small input numbers. However, nothing was said about how to choose the various parameters to these algorithms, in particular the B_1 and B_2 values and, for ECM, the parameters of the curve, so that the algorithms can be used efficiently within the Number Field Sieve.

The sieving phase of the Number Field Sieve looks for (a, b)-pairs with $a \perp b$ such that the values of two homogeneous polynomials $F_i(a, b)$, $i \in 1, 2$, are both smooth as described in Section 4.1. The sieving step identifies which of the primes up to the factor base limit $\mathfrak{B}_i$ divide each $F_i(a, b)$ and produces the cofactors c_i of $F_i(a, b)$ after the respective factor base primes have been divided out. The task of the cofactorization step in the sieving phase is to identify those cofactors that are smooth according to some smoothness criterion; typically a cofactor c_i is considered smooth if it does not exceed the cofactor bound C_i and has no prime factor exceeding the large prime bound L_i. The typical order of magnitude for B_i is around $10^7 \ldots 10^8$, and the L_i are typically between $100B_i$ and $1000B_i$.

To determine whether the cofactor pair (c_1, c_2) satisfies this smoothness criterion, we try to factor it. For this we attempt a sequence of factoring methods, where by "method" we mean a factoring algorithm with a particular set of parameters, such as: ECM with the elliptic curve generated by the Brent-Suyama parametrization with parameter $\sigma = 6$, with $B_1 = 200$ and $B_2 = 5000$.

In order to choose good parameters, we need to be able to compute the probability that a particular parameter choice finds a factor of the input number. That is, we need to compute the probability that a method finds a factor of a particular size and the expected number of factors in the input number of that size so that by summing over possible factor sizes we get the expected value for the number of factors the method will find.

We start by examining which parameters of the factoring methods affect the probability of finding a factor in Section 5.2, then show how this probability can be computed in Section 5.3. Finally we give an accurate estimate of the expected number of prime factors in a cofactor produced by NFS in Section 5.4.

5.2 Parametrization

In this section we take a closer look at the parameters available for the P–1, P+1, and ECM algorithms, how they affect run-time and the divisibility properties of the associated group orders. Each method finds a prime factor p of N if the starting element x_0 in a (here additively written) group G_p defined over $\mathbb{F}_p$ has smooth order; more precisely, if a multiple ex_0 of the starting element is the identity element of the group, or if qex_0 for a not-too-large prime q is the identity. The group operation of G_p requires arithmetic modulo p, a yet unknown prime factor of N, but we can instead do all arithmetic in $\mathbb{Z}/N\mathbb{Z}$ which contains $\mathbb{F}_p$. From the identity element of the group, a residue $r \pmod N$ is constructed with $r \equiv 0 \pmod p$ so that $p \mid \gcd(r, N)$, but hopefully $r \not\equiv 0 \pmod N$, since then the gcd reveals a proper factor of N.

Stage 1 of these algorithms computes ex_0 in their respective group, where e is typically chosen as $e = \operatorname{lcm}(1, 2, 3, 4, \ldots, B_1)$ for an integer parameter B_1, the "stage 1 bound," so that e includes as divisors all primes and prime powers up to B_1. This way, ex_0 is the identity if the order of x_0 is B_1-powersmooth, i.e., has no prime or prime power greater than B_1 as a divisor.

If stage 1 is unsuccessful, stage 2 takes ex_0 as input and efficiently tests if any qex_0 for many candidate primes q is the identity element; the set of primes to test is typically chosen as all primes greater than B_1, but not exceeding an integer parameter B_2, the "stage 2 bound."

The property that determines success or failure of these methods is the smoothness of the order of the starting element x_0 in G_p, and unlike P–1, the P+1 and Elliptic Curve methods have parameters that affect the order of the group G_p, and hence the order of x_0. By careful choice of parameters, the probability that the order of G_p (and hence of x_0) is smooth can be increased significantly. We therefore examine how the choice of parameters affects divisibility properties of the group order $|G_p|$ for a random prime p. By the probability of a random prime p having a property, we mean the ratio of primes $p < n$ that have that property in the limit of $n \to \infty$, assuming this limit exists.

The effect of these modified divisibility properties on the probability of the order of x_0 being smooth is examined in Section 5.3.

5.2.1 Parametrization for P–1

The P–1 method, described in Section 2.2 and Section 4.4, always works in the multiplicative group $\mathbb{F}_p^*$ of order $p - 1$, independently of the choice of the starting element x_0. Therefore we choose x_0 primarily to simplify the arithmetic in stage 1 of P–1, which is basically just a modular exponentiation. With $x_0 = 2$, the modular exponentiation can be carried out with only squarings and doublings in a binary left-to-right exponentiation ladder. A minor effect of the choice $x_0 = 2$ is that 2 is a quadratic residue if $p \equiv \pm 1 \pmod 8$, so in this case we know that $\operatorname{ord}_p(x_0) \mid (p-1)/2$, and (assuming $4 \mid e$) that $p - 1 \mid 2e$ is sufficient for finding any prime p.

The probability that $\operatorname{Val}_q(p-1) = k$ for a random prime p with q prime is $1 - 1/(q-1)$ for $k = 0$ and $1/q^k$ for $k > 0$; the expected value for $\operatorname{Val}_q(p-1)$ is $q/(q-1)^2$.

5.2.2 Parametrization for P+1

The P+1 method, described in Section 2.3 and Section 4.5, works in a subgroup of the group of units of $\mathbb{F}_p[X]/(X^2 - x_0X + 1)$ with $x_0 \not\equiv 0 \pmod p$; the order of the group is $p - \left(\frac{\Delta}{p}\right)$ where $\Delta = x_0^2 - 4$, hence it can be either $p - 1$ or $p + 1$, depending on p and the choice of x_0. This allows choosing x_0 so that the group order is more likely to be smooth.

For example, with $\Delta = -1 \cdot k^2$ for some rational k, the group order is always divisible by 4, increasing the average exponent of 2 in the group order from 2 to 3. Other prime factors in the group order appear with average exponent as in the P–1 method. A suitable value is $x_0 = 6/5$, giving $\Delta = -64/25 = -1 \cdot (8/5)^2$.

With $\Delta = -3 \cdot k^2$, the group order is always divisible by 6, which increases the average exponent of 3 in the group order from 3/4 to 3/2, with other primes factors of the group order behaving as in P–1. A suitable value is $x_0 = 2/7$, giving $\Delta = -192/49 = -3 \cdot (8/7)^2$. This choice is suggested in [65, §6].

5.2.3 Parametrization for ECM

The Elliptic Curve Method, described in Section 4.6, works in the group of points of an elliptic curve defined over $\mathbb{F}_p$. The order of the group depends on both p and the curve parameters and is an integer in the Hasse interval $\left[p + 1 - 2\sqrt{p}, p + 1 + 2\sqrt{p}\right]$.

Careful selection of the curve parameters allows forcing the group order to be divisible by 12 or by 16 and increasing the average exponent of small primes even beyond what this known factor of the group order guarantees, which greatly increases the probability of the group order being smooth, especially if the prime to be found and hence the order is small.

The ECM algorithm needs a non-singular curve over $\mathbb{F}_p$ and a starting point P_0 known to be on the curve. Since the prime factor p of N we wish to find is not known in advance, one chooses a non-singular curve over $\mathbb{Q}$ with a rational non-torsion point on it, and maps both to $\mathbb{Z}/N\mathbb{Z}$, hoping that the curve remains non-singular over $\mathbb{F}_p$, which it does for all primes p that don't divide the curve's discriminant.

The torsion points on an elliptic curve over $\mathbb{Q}$ remain distinct when mapping to $\mathbb{F}_p$ for almost all p, and the map to $\mathbb{F}_p$ retains the group structure of the torsion group, so that a curve with rational n-torsion over $\mathbb{Q}$ guarantees a subgroup of order n of the curve over $\mathbb{F}_p$, hence a group order divisible by n. By a theorem of Mazur [25, 7.1.11], the torsion group of an elliptic curve over $\mathbb{Q}$ is either cyclic with $1 \le n \le 10$ or $n = 12$, or is isomorphic to $\mathbb{Z}/2\mathbb{Z} \times \mathbb{Z}/(2m)\mathbb{Z}$ with $1 \le m \le 4$, giving $n \in \{4, 8, 12, 16\}$. Thus the two largest possible orders of the torsion group are 12 and 16.

For an effective ECM implementation we therefore seek a parametrization of elliptic curves that produces a large, preferably infinite, family of non-singular curves with a known point over $\mathbb{Q}$, that have a large torsion group over $\mathbb{Q}$, and whose curve parameters and coordinates of the starting point can be computed effectively in $\mathbb{Z}/N\mathbb{Z}$ where N is composite with unknown prime factors, which in particular rules out taking any square roots.

The first choice to make is the form of the curve equation. Any elliptic curve over a field with characteristic neither 2 nor 3 can be written in the short Weierstraß form

$$y^2 = x^3 + ax + b,$$

however the addition law for points on this curve over $\mathbb{Z}/N\mathbb{Z}$ involves a costly modular inverse.

Montgomery [65] suggests projective curves of the form

$$BY^2Z = X(X^2 + AXZ + Z^2) \tag{5.1}$$

which allow for an addition law without modular inverses, but require that to add two points, their difference is known, leading to more complicated addition chains for multiplying a point by an integer, described in 4.5.1 and 4.6.3.

	2	3	5	7	11	13	17	19
$12\mathbb{Z}$	3	1.5	0.25	0.167	0.1	0.0833	0.0625	0.0556
$\sigma = 2$	3.323	1.687	0.301	0.191	0.109	0.0898	0.0662	0.0585
$\sigma = 11$	3.666	1.687	0.302	0.191	0.109	0.0898	0.0662	0.0584
torsion 12	3.667	1.687	0.302	0.191	0.109	0.0898	0.0662	0.0585
$16\mathbb{Z}$	5	0.5	0.25	0.167	0.1	0.0833	0.0625	0.0556
torsion 16	5.333	0.680	0.302	0.191	0.109	0.0898	0.0661	0.0584

Table 5.1: Average exponent of primes up to 19 in integers that are a multiple of 12 or 16, and experimentally determined average exponent in the order of elliptic curves with Brent-Suyama parametrization and parameter $\sigma = 2$ and 11, and in curves with Montgomery's parametrization for rational torsion 12 and 16.

Edwards [37] suggests a new form of elliptic curve which Bernstein et al. [9] use for an efficient implementation of ECM. Edwards' curve form was not used for the present work, but in the light of the results of Bernstein et al. should be considered in future implementations of ECM.

The Brent-Suyama Parametrization

The Brent-Suyama parametrization of elliptic curves in Montgomery form, described in Section 4.6.2, is the most popular in existing implementations of ECM and is used, for example, in GMP-ECM and in Prime95. It produces curves with 6 rational torsion points, plus complex points of order 12 of which at least one maps to $\mathbb{F}_p$ for almost all p, leading to a group order of the curve over $\mathbb{F}_p$ that is divisible by 12. However, the order does not behave like an integer that is a multiple of 12 chosen uniformly at random from the Hasse interval; the average exponent of small primes is greater than ensuring divisibility by 12 suggests. Table 5.1 compares the average exponent of small primes in integers that are divisible by 12 and in the group orders of elliptic curves over $\mathbb{F}_p$, $10^3 < p < 10^9$, using the Brent-Suyama parametrization with $\sigma = 2$ (all other integer σ-values examined, except for $\sigma = 11$, produced same average exponents up to statistical noise) as well as the choice $\sigma = 11$, which surprisingly leads to a higher average exponent of 2 in the order.

The unexpected increase of the average exponent of 2 for $\sigma = 11$ has been examined by Barbulescu [19] who found two sub-families of curves produced by the Brent-Suyama parametrization that show this behavior. For each sub-family the σ-values live on an elliptic curve of rank 1 over $\mathbb{Q}$, thus producing an infinite family of curves for ECM.

The Montgomery Parametrizations

Montgomery [67, Chapter 6] gives two parametrizations for curves of form (4.3), one for curves with rational 12-torsion and one with rational 16-torsion, see Section 4.6.2.

Table 5.1 shows the average exponent of small primes in curves with 12 ($k = 2$) or 16 ($A = 54721/14400$) rational torsion points; curves with other parameters in the respective family produce similar figures.

5.2.4 Choice of Stage 1 Multiplier

The usual choice of the stage 1 multiplier is $E = \mathrm{lcm}(1, 2, \ldots, B_1)$, i.e., the product of all primes and prime powers up to B_1. This is based on the rationale that we want to include each prime and prime power that has a probability of dividing the group order of at least $1/B_1$. However, this choice does not take into account the different cost of including different primes q in the multiplier, which grows like $\log(q)$.

Consider the probability $\Pr[q^k \parallel n]$ where n is the value that we hope will divide E (say, the order of the starting element in the P–1, P+1 or ECM algorithm). In order to have $n \mid E$, every prime q must divide E in at least the same exponent as in n. Examining each prime q independently, we can choose to increase the exponent k of q so long as the probability that this increase causes q to occur in high enough power, divided by the cost of increasing k by 1, is better than some constant c, i.e., choose k maximal such that $\Pr[q^k \parallel n]/\log(q) \geq c$. This modified choice favours the inclusion of smaller primes, in higher powers than in $\mathrm{lcm}(1, 2, \ldots, B_1)$.

For a comparison, we assume that n behaves like a "random" integer, meaning that $\Pr[q^k \parallel n] = (1-1/q)/q^k$. With $E = \mathrm{lcm}(1, \ldots, 227)$, there are 871534 integers in $[900000000, 1100000000]$ with $n \mid E$, whereas using the improved choice with $c = 1/1200$ gives $E'/E = 27132/50621 = 2^2 \cdot 3 \cdot 7 \cdot 17 \cdot 19/(223 \cdot 227)$ and finds 913670 in the same interval, a 4.8% increase, even though E' is slightly smaller than E. The advantage quickly diminishes with larger B_1, however. With $E = \mathrm{lcm}(1, \ldots, 990)$, 7554965 integers are found, whereas the choice $c = 1/6670$ leads to $E'/E = 182222040/932539661 = 2^3 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdot 13 \cdot 37 \cdot 41/(971 \cdot 977 \cdot 983)$ and finds 7623943 integers, approximately a 1% increase. Here, the ratio $E'/E \approx 1/5$ is higher than before, but this is not the reason for the smaller gain. Even with $6E'$, the number of integers found is 7635492, still only an about 1% difference.

The group order $|G_p|$ in P–1, P+1, and ECM does not behave like a random integer, but for small primes q the probability that q^k divides exactly the group order $|G_p|$ (or better, the order of the starting element x_0 in G_p) for a random prime p is easy enough to determine empirically, which allows choosing the multiplier E so that the particular divisibility properties of the group order are taken into account.

5.3 Estimating Success Probability of P–1, P+1, and ECM

5.3.1 Smooth Numbers and the Dickman Function

To estimate the probability of finding a prime factor with the P–1, P+1, or ECM algorithms, we need to estimate the probability that the order of the starting element is smooth. Even though the order of the starting element does not behave quite like a random integer, we start by recapitulating well-known methods of estimating smoothness probabilities for random integers, and show how to modify the estimate to take into account the known divisibility properties of the order of the starting element.

I have investigated the question of how to compute smoothness probabilities using Dickman's ρ-function, and how to modify the computation to give the probability of smoothness of an integer close to N in the context of my Diploma thesis [58]. The relevant parts of the text are included here for completeness.

Let $S(x, y)$ be the set of y-smooth natural numbers not exceeding x, $\Psi(x, y)$ the cardinality of the set $S(x, y)$, $P(1/u, x)$ the number of $x^{1/u}$-smooth integers not exceeding x for real $u > 0$, and $\rho(u)$ the limit of $P(1/u, x)/x$ for $x \to \infty$ (Dickman's function, [35]).

$$\begin{aligned}
S(x,y) &= \{n \in \mathbb{N}, n \leq x : p \in \mathbb{P}, p \mid x \Rightarrow p \leq y\} \\
\Psi(x,y) &= |S(x,y)| \\
P(1/u,x) &= \Psi(x, x^{1/u}) \\
\rho(u) &= \lim_{x\to\infty} \frac{P(1/u,x)}{x}.
\end{aligned}$$

That the limit for $\rho(u)$ exists is proved, for example, by Knuth and Trapp-Pardo [57] where they show that $\Psi(x,y)$ and $\rho(u)$ satisfy

$$\Psi(x, x^{1/u}) = \rho(u)x + O\left(\frac{x}{\log(x)}\right). \tag{5.2}$$

They also give an improvement that lowers the error term,

$$\Psi(x, x^{1/u}) = \rho(u)x + \sigma(u)\frac{x}{\log(x)} + O\left(\frac{x}{\log(x)^2}\right) \tag{5.3}$$

with $\sigma(u) = (1-\gamma)\rho(u-1)$.

Evaluating $\rho(u)$

The Dickman $\rho(u)$ function can be evaluated by using the definition

$$\rho(u) = \begin{cases} 0 & u \leq 0 \\ 1 & 0 < u \leq 1 \\ 1 - \int_1^u \frac{\rho(t-1)}{t}\mathrm{d}t & u > 1. \end{cases} \tag{5.4}$$

In principle $\rho(u)$ can be computed from (5.4) for all u via numerical integration, however when integrating over this delay differential equation, rounding errors accumulate quickly, making it difficult to obtain accurate results for larger u.

This problem can be alleviated somewhat by using a closed form of $\rho(u)$ for small u. For example, for $1 < u \leq 2$ we have

$$\begin{aligned}
\rho(u) &= 1 - \int_1^u \frac{\rho(t-1)}{t}\mathrm{d}t \\
&= 1 - \int_1^u \frac{1}{t}\mathrm{d}t \\
&= 1 - \log(u)
\end{aligned}$$

and for $2 < u \leq 3$

$$\begin{aligned}
\rho(u) &= 1 - \int_1^u \frac{\rho(t-1)}{t}\mathrm{d}t \\
&= 1 - \log(2) - \int_2^u \frac{1-\log(t-1)}{t}\mathrm{d}t \\
&= 1 - \log(2) - \Big(\log(t) + \log(t)\big(\log(1-t) - \log(t-1)\big) + \mathrm{Li}_2(t)\Big)\Big|_2^u
\end{aligned}$$

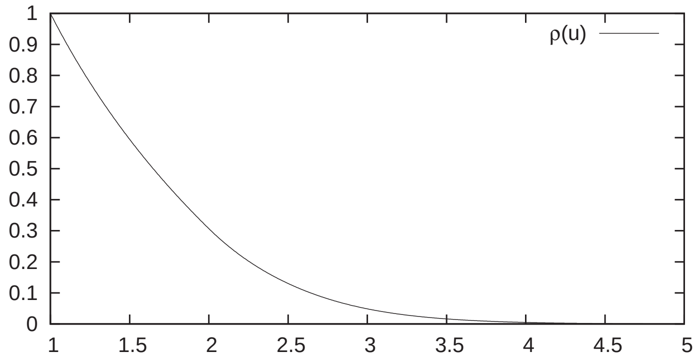

Figure 5.1: The Dickman function $\rho(u)$ for $1 \leq u \leq 5$.

where $\mathrm{Li}_2(x)$ is the dilogarithm of x, which is defined by the sum

$$\mathrm{Li}_2(x) = \sum_{k=1}^{\infty} \frac{x^k}{k^2}.$$

Note that some authors and software define the dilogarithm of x as what corresponds to $\mathrm{Li}_2(1-x)$ in our notation. To avoid negative arguments to logarithms, we use the functional identity

$$\mathrm{Li}_2(x) = \frac{1}{6}\pi^2 - \log(x)\log(1-x) - \mathrm{Li}_2(1-x)$$

and $\mathrm{Li}_2(-1) = -\pi^2/12$ which produces for $2 < u \leq 3$

$$\rho(u) = 1 + \frac{\pi^2}{12} - \log(u)\bigl(1 - \log(u-1)\bigr) + \mathrm{Li}_2(1-u).$$

In this way, values of $\rho(u)$ for $0 \leq u \leq 3$ can easily be computed with high accuracy. Values up to approximately $u = 7$ can be computed via numerical integration; for larger values the accumulated rounding errors start having significant effect on the computed $\rho(u)$-values. For our application, only relatively small u-values are considered, so the numerical integration approach is sufficient. A graph of $\rho(u)$ for $1 \leq u \leq 5$ is shown in 5.1.

If $\rho(u)$ is needed with high accuracy for possibly large u-values, Marsaglia and Zama [63] show how to compute $\rho(u)$ (and other functions defined by delay-differential equations) accurately to up to hundreds of digits by expressing $\rho(u)$ piecewise as power series.

Smooth Numbers Close to x

Dickman's function estimates the ratio of $x^{1/u}$-smooth numbers between 1 and x. However, we would like to compute the probability that a number of a certain magnitude close to some x is smooth instead. Fortunately it is easy to convert to this model.

The ratio of y-smooth integers between x and $x + dx$ is

$$\frac{\Psi(x + dx, y) - \Psi(x, y)}{dx}.$$

Using the approximation (5.2) with $u = \log(x)/\log(y)$ and ignoring the $O(x/\log(x))$ term, we get

$$\begin{aligned}\frac{\rho\left(\frac{\log(x+dx)}{\log(y)}\right)(x+dx) - \rho(u)x}{dx} &= \\ \rho\left(u + \frac{\log(1+d)}{\log(y)}\right) + \frac{\rho\left(u + \frac{\log(1+d)}{\log(y)}\right) - \rho(u)}{d}\end{aligned}$$

and with $d \to 0$, this becomes

$$\begin{aligned}\rho(u) + \frac{1}{\log(y)}\rho'(u) &= \\ \rho(u) + \frac{u}{\log(x)}\rho'(u).\end{aligned}$$

Now substituting $\rho'(u) = -\rho(u-1)/u$ yields

$$\rho(u) - \frac{1}{\log(x)}\rho(u-1). \tag{5.5}$$

Considering the $O(x/\log(x))$ term again, we find

$$\frac{\Psi(x+dx,y) - \Psi(x,y)}{dx} = \rho(u) + O\left(\frac{1}{\log(x)}\right)$$

for $x \to \infty$, no better than using Dickman's function immediately. Starting instead with the better approximation (5.3) leads to

$$\rho(u) - \gamma\frac{\rho(u-1)}{\log(x)} - (1-\gamma)\frac{\rho(u-2)}{u\log(x)^2}$$

and

$$\frac{\Psi(x+dx,y) - \Psi(x,y)}{dx} = \rho(u) - \gamma\frac{\rho(u-1)}{\log(x)} + O\left(\frac{1}{\log(x)^2}\right)$$

for $x \to \infty$. Thus we define

$$\hat{\rho}(u,x) = \rho(u) - \gamma\frac{\rho(u-1)}{\log(x)} \tag{5.6}$$

and have

$$\frac{\Psi(x+dx,y) - \Psi(x,y)}{dx} = x\hat{\rho}(u,x) + O\left(\frac{1}{\log(x)^2}\right). \tag{5.7}$$

This approximation shows that using (5.3), which includes the $\sigma(u)$ correction term but estimates the ratio of smooth values between 1 and x, actually produces a worse result for the ratio of smooth numbers close to x than the plain Dickman $\rho(u)$ alone, whereas (5.5), which estimates the ratio to smooth values close to x but without the $\sigma(u)$ term, does not result in any improvement of the estimate over the $\rho(u)$-function, leaving the magnitude of the error almost the same but changing the sign.

Table 5.2 compares the estimates produced by the Dickman function $\rho(3)$ and by $\hat{\rho}(3,x)$ with experimental counts. The relative error of the plain $\rho(3)$ function is between 12 and 16 percent in the investigated range, while the relative error for $\hat{\rho}(3,x)$ for larger x is more than an order of magnitude smaller and about 0.5 percent for $x \geq 10^{14}$. For larger values of u, the relative error for $\hat{\rho}(u,x)$ is significantly higher, due to the $\rho(u-2)$ factor in the $O()$ term. The ratio $\rho(u-2)/\rho(u)$ increases quickly with u, approximately like u^2 [57, 6.3]. For example, the relative error for $\hat{\rho}(5,10^{15})$ is about 12 percent.

x	Count	$\rho(3)\cdot 10^8$		$\hat{\rho}(3,x)\cdot 10^8$	
		Est.	Rel. err. in %	Est.	Rel. err. in %
10^9	4192377	4860839	15.94	4006146	−4.44
10^{10}	4239866	4860839	14.64	4091615	−3.50
10^{11}	4238110	4860839	14.69	4161545	−1.81
10^{12}	4280377	4860839	13.56	4219819	−1.41
10^{13}	4304040	4860839	12.94	4269128	−0.81
10^{14}	4336571	4860839	12.09	4311393	−0.58

Table 5.2: A comparison of experimental count $\Psi(x^{1/3}, x+5\cdot 10^7) - \Psi(x^{1/3}, x-5\cdot 10^7)$ and estimated number of $x^{1/3}$-smooth numbers in $\left]x-5\cdot 10^7, x+5\cdot 10^7\right]$

5.3.2 Effect of Divisibility Properties on Smoothness Probability

The P–1, P+1, and ECM algorithms all find a prime factor p if a group defined over $\mathbb{F}_p$ has smooth order, and we have seen that the group order does not behave like a random integer with respect to divisibility by small primes. We would like to quantify the effect of these divisibility probabilities of small primes on the probability that the order is smooth so that the effectiveness of different parameter choices can be compared accurately.

The effect of the higher frequency of small factors on smoothness probability can be estimated with a technique developed by Schröppel and Knuth for the analysis of the Continued Fraction method of factorisation [56, 4.5.4]. The idea is to compare the average exponent $f(q,S)$ of small primes q in values chosen uniformly at random from the set S of numbers being tested for smoothness with the average exponent found when choosing values from the integers, $f(q,\mathbb{N}) = 1/(q-1)$.

When choosing an $s \in S$ and dividing out the primes $q < k$, we can expect the remaining cofactor r to have logarithmic size

$$\log(r) = \log(s) - \sum_{q\in\mathbb{P}, q<k} f(q,S)\log(q). \tag{5.8}$$

Comparing the size of this cofactor with that for the case $S = \mathbb{N}$, we find that the expected value for $\log(r)$ is smaller by

$$\delta = \sum_{q\in\mathbb{P}, q<k} \left(f(q,S) - \frac{1}{q-1}\right)\log(q). \tag{5.9}$$

Knuth and Trapp-Pardo then argue that, since the log size of the cofactor is smaller by δ, the number s is as likely smooth as a random integer smaller by a factor e^δ in value.

For the P–1 algorithm,

$$\begin{aligned}\delta_{\text{P–1}} &= \sum_{q\in\mathbb{P}, q<k} \left(\frac{q}{(q-1)^2} - \frac{1}{q-1}\right)\log(q)\\ &= \sum_{q\in\mathbb{P}, q<k} \frac{1}{(q-1)^2}\log(q),\end{aligned}$$

which is approximately 1.22697 for $k \to \infty$.

For the P+1 algorithm with $x_0 = 6/5$, the group order is always divisible by 4. The average exponent of primes in the group order is as in P–1, except for the exponent of 2 which is 3 on average instead of 2, giving $\delta_{\text{P+1},4} = \delta_{\text{P–1}} + \log(2) \approx 1.92012$. With $x_0 = 2/7$, the group order is divisible by 6, increasing the average exponent of 3 from 3/4 to 3/2, so that $\delta_{\text{P+1,6}} = \delta_{\text{P–1}} + 3/4 \log(3) \approx 2.05093$.

For ECM it is more difficult to give a theoretical estimate for the average exponent of small primes in the group order, although Montgomery [67, 6.3] proposes conjectures for the exponent of 2 and 3 in the group order in curves with rational torsion 12 or 16. The approximate δ value for ECM has been determined experimentally by computing the average exponent as in Table 5.1, but extended to primes up to 100:

	ECM $\sigma = 2$	ECM $\sigma = 11$	ECM torsion 12	ECM torsion 16
δ	3.134	3.372	3.373	3.420
e^{δ}	22.97	29.14	29.17	30.57

5.3.3 Success Probability for the Two-Stage Algorithm

The P–1, P+1, and ECM factoring algorithms work in two stages, where stage 1 finds a factor p if the order $|G_p|$ of the respective group over p is B_1-smooth, and stage 2 finds p if the order contains one prime factor q with $B_1 < q \le B_2$, and the cofactor $|G_p|/q$ is B_1-smooth. Assuming that a stage 2 prime q divides the order with probability $1/q$, the probability of a factoring method finding a factor can therefore be computed as

$$P(N_{\text{eff}}, B_1, B_2) = \hat{\rho}\left(\frac{\log(N_{\text{eff}})}{\log(B_1)}\right) N_{\text{eff}} + \sum_{\substack{B1<q\le B2 \\ q\in\mathbb{P}}} \hat{\rho}\left(\frac{\log(N_{\text{eff}}/q)}{\log(B_1)}\right) N_{\text{eff}}/q, \tag{5.10}$$

where $N_{\text{eff}} = Ne^{-\delta}$ is the approximate size of p, adjusted by the δ parameter for the respective factoring algorithm as described in Section 5.3.2. For sufficiently large B_1 and B_2, the sum can be replaced by an integral as shown for example in [15], but in our case where these parameters are quite small, the sum produces significantly more accurate results and is still acceptably fast to evaluate.

5.3.4 Experimental Results

We compare the ratio of prime factors being found by P–1, P+1, and the Elliptic Curve Method with the estimate produced by Equation (5.10). For $n = 25, 26$ we test primes in $\left[2^n - 10^6, 2^n + 10^6\right]$ and for each $27 \le n \le 32$ we test primes in $\left[2^n - 10^7, 2^n + 10^7\right]$ with the P–1 method, P+1 with $x_0 = 6/5$ and $x_0 = 2/7$, and ECM with the Brent-Suyama parametrization with $\sigma = 2$ as well as the parametrization for curves with rational 12 and 16 torsion, and record the ratio of primes found by the respective method. The Brent-Suyama parametrization with $\sigma = 11$ behaves identically to that of curves with rational 12-torsion. The B_1 and B_2 parameters for P–1 and P+1 are chosen as in Table 4.9 and all ECM parametrizations use the B_1 and B_2 values from Table 4.10.

The largest relative error of about 3% occurs for P–1 with $n = 25$, but most errors are below 1%. It is somewhat surprising how accurate the results are, considering that estimating the density of y-smooth numbers around x by $\hat{\rho}(\log(x)/\log(y))$ includes a rather large error term for small values of x. From a purely pragmatic point of view, however, we may be content with using (5.10) as an easily computable and, for the range of parameters we are interested in, very accurate estimate of the probability of success for our factoring algorithms.

	P–1		P+1				ECM					
			$x_0 = 6/5$		$x_0 = 7/2$		$\sigma = 2$		12 torsion		16 torsion	
n	Est.	Emp.	Est.	Emp.	Est.	Emp.	Est.	Emp.	Est.	Emp.	Est.	Emp.
25	0.242	0.249	0.287	0.289	0.297	0.299	0.336	0.337	0.358	0.359	0.363	0.359
26	0.217	0.220	0.259	0.258	0.267	0.265	0.275	0.276	0.295	0.297	0.299	0.298
27	0.186	0.186	0.220	0.221	0.226	0.227	0.271	0.271	0.291	0.290	0.291	0.291
28	0.166	0.167	0.198	0.197	0.204	0.203	0.238	0.237	0.256	0.256	0.255	0.254
29	0.148	0.149	0.177	0.177	0.183	0.183	0.203	0.207	0.218	0.220	0.217	0.216
30	0.157	0.158	0.186	0.186	0.190	0.191	0.177	0.178	0.189	0.188	0.189	0.190
31	0.126	0.128	0.150	0.150	0.154	0.155	0.170	0.172	0.182	0.182	0.183	0.182
32	0.104	0.105	0.124	0.126	0.128	0.129	0.136	0.137	0.147	0.147	0.145	0.144

Table 5.3: Comparison of estimated probability of finding a prime factor close to 2^n with the P–1, P+1, and ECM algorithm with empirical results. The choice of parameters is described in the text

5.4 Distribution of Divisors

In Section 5.3 we estimate the probability that a factoring method finds a factor of a certain size if it exists. To estimate the expected number of factors the method finds for an input number N, we also need the expected number of such factors in N, taking into account the available information on the number: its size, that it is composite, and that it has no prime factors up to the factor base bound used for sieving. That is, we would like to compute the expected number of prime factors p with $p \in [z_1, z_2]$ of a composite N, chosen uniformly at random from the integers in $[N_1, N_2]$ that have no prime factors up to y. To do so, we need to estimate the number of integers up to a bound that have no small prime factors.

Let

$$\begin{aligned} T(x, y) &= \{n \in \mathbb{N}, 1 \le n \le x : p \in \mathbb{P}, p \mid x \Rightarrow p > y\}, \quad x \ge y \ge 2, && (5.11)\\ \Phi(x, y) &= |T(x, y)| && (5.12) \end{aligned}$$

be the set of positive integers up to x having no small prime factors up to y (which always includes 1 in $T(x, y)$) and the number of such integers, respectively.

Much like Dickman's function $\rho(u)$, with $u = \log(x)/\log(y)$ throughout, can be used to estimate the number $\Psi(x, y)$ of positive integers up to x that have no large prime factor above y, the Buchstab function $\omega(u)$ can be used to estimate $\Phi(x, y)$. Tenenbaum [96, III.6.2] shows that

$$\Phi(x, y) = (x\omega(u) - y)/\log(y) + O\big(x/\log(y)^2\big) \text{ for } x^{1/3} \le y \le x. \tag{5.13}$$

This estimate is reasonably accurate for large u and y, but not for small $u > 2$. Since we frequently need to treat composites that have only two relatively large prime factors, we need an estimate that is more accurate for $2 < u < 3$, which is described in 5.4.2.

5.4.1 Evaluating $\omega(u)$

The Buchstab $\omega(u)$ function is defined by

$$\omega(u) = \begin{cases} 1/u & 1 \le u \le 2 \\ (1 + \int_1^{u-1} \omega(t)\,\mathrm{d}t)/u & u > 2. \end{cases} \tag{5.14}$$

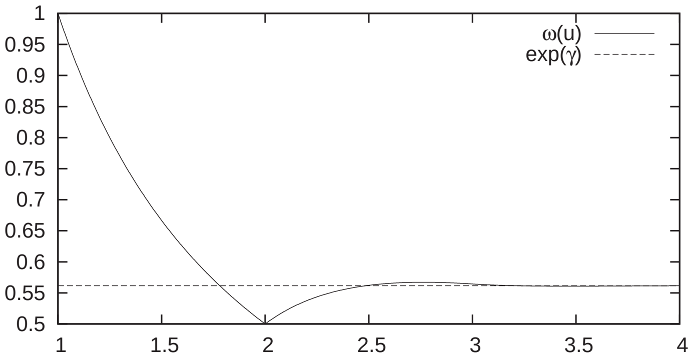

Figure 5.2: The Buchstab function $\omega(u)$ for $1 \le u \le 4$, and the limiting value $e^{\gamma} \approx 0.56146$.

Figure 5.2 shows the graph for $1 \le u \le 4$.

Like the Dickman function, the Buchstab function can be expressed in logarithms and the dilogarithm for small u values:

$$\omega(u) = \begin{cases} 1/u & 1 \le u \le 2 \\ (1 + \log(u-1))/u & 2 < u \le 3 \\ \left(\mathrm{Li}_2(2-u) + (1+\log(u-2))\log(u-1) + \pi^2/12 + 1\right)/u & 3 < u \le 4. \end{cases} \tag{5.15}$$

However, unlike the Dickman function $\rho(u)$ which monotonously tends to 0 for $u \to \infty$, the Buchstab function $\omega(u)$ oscillatingly tends to $e^{-\gamma}$ and has almost reached this limit for $u = 4$ already, with relative error only $-2.2 \cdot 10^{-6}$. This greatly simplifies the evaluation for $\omega(u)$ for our application, as using $\omega(u) = e^{-\gamma}$ for $u > 4$ is sufficiently accurate for our purposes, and for smaller u values the closed forms of (5.15) can be used.

The methods of Marsaglia and Zaman [63] can be readily applied to the Buchstab $\omega(u)$ function if highly accurate results for large u are required.

5.4.2 Estimating $\Phi(x, y)$

Equation (5.13) is not sufficiently accurate for some small values of $u = \log(x)/\log(y)$. For example, $\Phi(10^9, 15000) = 54298095$, but (5.13) estimates 55212172. This estimate has a relative error of only 1.7%, but we need the number of *composites* with no small prime factors, and after subtracting $\pi(10^9) - \pi(15000) = 50845780$, the number of primes between 15000 and 10^9 from both, the correct value is 3452315 while the estimate is 4366392 with 26% relative error.

Fortunately a better estimate is given in [96, III, 6.4]:

$$\Phi(x, y) \approx x \frac{e^{\gamma} \log(y)}{\zeta(1, y)} \int_0^{u-1} \omega(u-v) y^{-v} \mathrm{d}v. \tag{5.16}$$

Using $x = y^u$, (5.16) can be rewritten as

$$\Phi(x, y) \approx \frac{e^{\gamma} \log(y)}{\zeta(1, y)} \int_1^{u} \omega(v) y^{v} \mathrm{d}v. \tag{5.17}$$

.

y	u	$\Phi(10^9, y)$	Estimate by (5.13)	Rel. error in percent	Estimate by (5.16)	Rel. error in percent
1000	3	81515102	81702575	0.23	81540712	0.031
2000	2.73	73931443	74604335	0.91	73964201	0.044
3000	2.59	69618529	70581865	1.4	69649564	0.045
4000	2.50	66671714	67774139	1.7	66688977	0.026
5000	2.43	64375942	65618739	1.9	64392876	0.026
10000	2.25	57680218	59021636	2.3	57679743	−0.00082
15000	2.16	54298095	55212172	1.7	54278816	−0.036
20000	2.09	52354286	52522945	0.32	52330198	−0.046

Table 5.4: The number $\Phi(10^9, y)$ of positive integers up to 10^9 without prime divisors up to y and comparison to estimates based on Buchstab's function

By Mertens' 3rd Theorem, $1/\zeta(1, y) = \prod_{p\in\mathbb{P},p\leq y}(1-1/p) \approx 1/(e^\gamma \log(y))$ and for our purpose where y is the factor base bound and hence in the millions, the estimate by Mertens' theorem is accurate enough (for $y = 10^6$, the relative error is 0.03%) and we can simplify (5.17) to

$$\Phi(x, y) \approx \int_1^{\log(x)/\log(y)} \omega(v)y^v \mathrm{d}v. \tag{5.18}$$

This estimate gives $\Phi(10^9, 15000) \approx 54319245$ with relative error only 0.039%, and after subtracting $\pi(10^9) - \pi(15000)$, the relative error is still only 0.6%. Table 5.4 shows the values of $\Phi(10^9, y)$ for other y-values and compares them to the estimate using (5.17) (since these y-values are rather small, the simplified form (5.18) introduces a noticeable error).

5.4.3 Divisors in Numbers with no Small Prime Factors

We would like to estimate the total number

$$D(x, y, z_1, z_2) = \sum_{n\in T(x,y)} \sum_{\substack{p\in\mathbb{P} \\ z_1<p\leq z_2}} \mathrm{Val}_p(n), \quad x \geq z_2 \geq z_1 \geq y \geq 2 \tag{5.19}$$

of prime divisors $p \in \,]z_1, z_2]$ with multiplicity among positive integers up to x that have no prime factors up to y. A hint towards the solution is Buchstab's identity [96, §6.2, (14)]

$$\Phi(x, y) = 1 + \sum_{\substack{p\in\mathbb{P} \\ y<p\leq x}} \sum_{\nu\geq1} \Phi(x/p^\nu, p)$$

which partitions $T(x, y)$ into $\{1\}$ and sets of integers divisible by a prime $p > y$ and by no other prime $q \leq p$, for $y < p \leq x$. For our problem, however, we would like to count integers which contain several prime factors (possibly as powers) from $]z_1, z_2]$ with multiplicity. This leads to

$$D(x, y, z_1, z_2) = \sum_{\substack{p\in\mathbb{P} \\ z_1<p\leq z_2}} \sum_{\nu\geq1} \Phi(x/p^\nu, y)$$

which for each prime $z_1 < p \leq z_2$ counts the positive integers $kp^\nu \leq x$ such that k has no prime factor up to y, i.e., $k \in T(x/p^\nu, y)$.

z_1	$D(10^{18}+10^{11},10^7,z_1,z_1+10^7)$	Estimate	Rel. error
	$-D(10^{18},10^7,z_1,z_1+10^7)$	using (5.20)	in percent
$1 \cdot 10^7$	168541774	168537797	−0.002
$2 \cdot 10^7$	97523916	97542736	0.02
$3 \cdot 10^7$	68793931	68800701	0.01
$4 \cdot 10^7$	53152172	53155498	0.006
$5 \cdot 10^7$	43292469	43305316	0.03
$6 \cdot 10^7$	36540046	36531701	−0.02
$7 \cdot 10^7$	31578964	31587779	0.03
$8 \cdot 10^7$	27811212	27820533	0.03
$9 \cdot 10^7$	24867513	24854715	−0.05
$10 \cdot 10^7$	22475389	22459308	−0.07
$15 \cdot 10^7$	15144159	15150750	0.04
$20 \cdot 10^7$	11429968	11427841	−0.02
$25 \cdot 10^7$	9168249	9172594	0.05
$30 \cdot 10^7$	7665357	7660398	−0.06
$35 \cdot 10^7$	6573271	6576127	0.04
$40 \cdot 10^7$	5758937	5760733	0.03
$45 \cdot 10^7$	5123766	5125277	0.03
$50 \cdot 10^7$	4609652	4616138	0.1

Table 5.5: The number $D(10^{18}+10^{10},10^7,z_1,z_1+10^7) - D(10^{18},10^7,z_1,z_1+10^7)$ of prime factors $z_1 < p \le z_1 + 10^7$ with multiplicity among the integers in $\left[10^{18}, 10^{18}+10^{10}\right]$ without prime divisors up to 10^7, and comparison to estimates using (5.20).

If z_1 is reasonably large, the contribution of prime powers ($\nu > 1$) is quite small and can be omitted for an approximate result. By replacing the resulting sum by an integral over (5.18), we obtain

$$\begin{aligned} D(x,y,z_1,z_2) &\approx \int_{z_1}^{z_2} \Phi(x/t,y)/\log(t)\mathrm{d}t \\ &\approx \int_{z_1}^{z_2} \frac{1}{\log(t)} \left(\int_1^{u-\log(t)/\log(y)} \omega(v) y^v \mathrm{d}v \right) \mathrm{d}t, \end{aligned} \tag{5.20}$$

where $u = \log(x)/\log(y)$.

By $D(x_2,y,z_1,z_2) - D(x_1,y,z_1,z_2)$ we can estimate the number of prime factors p with $z_1 < p \le z_2$ among numbers $N \in [x_1,x_2]$ that have no prime factors up to y. Table 5.5 compares this estimate with experimental counts for parameters of approximately the magnitude as might occur in the Number Field Sieve: we consider composites in $[10^{18}, 10^{18}+10^{10}]$ that have no prime factors up to 10^7, and estimate the number of prime factors in $\left[i \cdot 10^7, (i+1) \cdot 10^7\right]$ for some i up to 50. The estimates in this table are remarkably accurate, with relative error mostly below 0.1%.

Conclusion

Here we briefly summarize the results of the individual chapters of the thesis.

Schönhage-Strassen's algorithm is among the fastest integer multiplication algorithms that use only integer arithmetic. Significant speedups can be gained by improving cache-locality, using $\sqrt{2}$ as a root of unity in the transform to double the possible transform length, mixing Mersenne and Fermat transforms at the top-most recursion level, and fine-grained parameter selection depending on input number size. These improvements combined resulted in a factor 2 speedup over the implementation of Schönhage-Strassen's algorithm in GMP version 4.1.4, on which our implementation is based.

The P–1 and P+1 factoring methods allow a particularly fast implementation for stage 2 of the algorithms. It is based on polynomial multi-point evaluation. The polynomial to be evaluated can be built from its roots much more quickly than with a general product tree by exploiting patterns in the roots. With suitably chosen roots it is a reciprocal Laurent polynomial, and such polynomials can be stored using half the space, and can be multiplied with a weighted Fourier transform in about half the time, as general polynomials of the same degree. The multi-point evaluation of the polynomial is particularly efficient for the P-1 algorithm, requiring essentially only one cyclic convolution product. It can be adapted to the P+1 algorithm, but needs to work in a quadratic extension ring, increasing memory use and computation time. The new code is distributed as part of GMP-ECM version 6.2 and later.

The sieving step is in practice the most computationally expensive step of the Number Field Sieve. The cofactorization during the sieving can be performed efficiently with the P–1, P+1, and ECM algorithms. We have designed a software implementation optimized for low-overhead, high-throughput factorization of small integers that is competitive in terms of cost/performance-ratio with proposed FPGA implementations of ECM for NFS.

The following ideas may be worth considering for further research.

The excellent performance of convolutions with a complex floating-point FFT for integer multiplication is intriguing, but requiring guaranteed correct rounding would greatly increase the necessary transform length and thus run-time. A promising idea is to use the Schönhage-Strassen algorithm only at the top-most recursive level, and compute the point-wise nega-cyclic convolutions with a complex floating-point FFT. The relatively short length of those FFTs should make an implementation with correct rounding feasible without too great a sacrifice in speed.

Fürer's algorithm for integer multiplication has slightly better asymptotic complexity of $O\big(n \log(n) 2^{\log^*(n)}\big)$ than Schönhage-Strassen's with complexity $O(n \log(n) \log(\log(n)))$. Which one is faster in practice? No well-optimized implementation of Fürer's algorithm seems to exist at the time of writing, and creating one might be worthwhile.

Schönhage and Strassen conjecture in the paper introducing their algorithm that the complexity for an optimal integer multiplication algorithm should be $O(n \log(n))$, but no such algorithm is known so far, although Fürer's algorithm shortened the gap. Discovery of such an algorithm would be an exiting result for computational complexity theory, and if fast in practice, of great value in large-integer arithmetic.

Is a stage 2 for ECM possible that is similarly fast as stage 2 for P–1 and P+1? Currently it is not clear how it might work, as for P–1 and P+1 the rapid construction of a polynomial from its roots and the multi-point evaluation use the fact that $\mathbb{Z}/N\mathbb{Z}$ (or a quadratic extension for P+1) has ring structure which is not present in the group of points on an elliptic curve. But maybe another novel approach might offer a significant speedup for the ECM stage 2.

Factoring small integers with P-1, P+1, and ECM with a software implementation on a general-purpose CPU was found to be very competitive with implementations of ECM in FPGAs, but alternative computing hardware such as graphics cards that support general purpose programming offer vast amounts of processing power at consumer prices which may give them the advantage. A complete implementation of ECM with stage 1 and stage 2 on graphics cards would be interesting; if it should turn out to be too difficult due to restricted memory, doing stage 1 on the graphics card and stage 2 on the CPU might be feasible.

Parameters can be chosen accurately for cofactorization with P–1, P+1, and ECM to maximise the ratio of probability of success versus time for one individual method and one composite number. However, pairs of composites that both must be smooth to form a relation need to be factored in the sieving step of NFS, and generally a succession of factoring methods needs to be tried to obtain the factorization of each. How can we choose a sequence of factoring attempts on the two numbers so that a conclusion, either by factoring both or finding at least one (probably) not smooth, is reached as quickly as possible? This choice should take into account the effect of unsuccessful factoring attempts on the expected number of factors of a given size in the composite number.

Bibliography

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

[2] Kazumaro Aoki, Jens Franke, Thorsten Kleinjung, Arjen K. Lenstra, and Dag Arne Osvik. A Kilobit Special Number Field Sieve Factorization. In Kaoru Kurosawa, editor, *Advances in Cryptology – ASIACRYPT 2007*, number 4833 in Lecture Notes in Computer Science, pages 1–12. Springer-Verlag, 2008.

[3] Jörg Arndt. *Matters Computational*. November 2009. Draft, available at `http://www.jjj.de/fxt/`.

[4] David H. Bailey. FFTs in external or hierarchical memory. *J. Supercomputing*, 4:23–35, 1990.

[5] Paul Barrett. Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In *Proceedings on Advances in cryptology—CRYPTO '86*, pages 311–323, London, UK, 1987. Springer-Verlag.

[6] Günter Baszenski and Manfred Tasche. Fast Polynomial Multiplication and Convolutions Related to the Discrete Cosine Transform. *Linear Algebra and its Applications*, 252:1–25, 1997.

[7] Daniel J. Bernstein. Circuits for integer factorization: a proposal. Manuscript, 2001. `http://cr.yp.to/nfscircuit.html`.

[8] Daniel J. Bernstein. Multidigit multiplication for mathematicians. `http://cr.yp.to/papers.html#m3`, 2001.

[9] Daniel J. Bernstein, Peter Birkner, Tanja Lange, and Christiane Peters. ECM using Edwards curves. `http://eecm.cr.yp.to/index.html`, 2009.

[10] Daniel J. Bernstein, Hsueh-Chung Chen, Ming-Shing Chen, Chen-Mou Cheng, Chun-Hung Hsiao, Tanja Lange, Zong-Cing Lin, and Bo-Yin Yang. The Billion-Mulmod-Per-Second PC. In *Special-purpose Hardware for Attacking Cryptographic Systems (SHARCS'09)*, 2009.

[11] Daniel J. Bernstein and Arjen K. Lenstra. A general number field sieve implementation. In Lenstra and Lenstra [60], pages 103–126.

[12] Daniel J. Bernstein and Jonathan P. Sorenson. Modular exponentiation via the explicit Chinese remainder theorem. *Mathematics of Computation*, 76:443–454, 2007.

[13] Marco Bodrato and Alberto Zanoni. Integer and Polynomial Multiplication: Towards Optimal Toom-Cook Matrices. In C. W. Brown, editor, *ISSAC '07: Proceedings of the 2007 International Symposium on Symbolic and Algebric Computation*, pages 17–24, New York, 2007. ACM.

[14] Richard P. Brent. An improved Monte Carlo factorization algorithm. *BIT Numerical Mathematics*, 20(2):176–184, 1980.

[15] Richard P. Brent. Some Integer Factorization Algorithms using Elliptic Curves. *Australian Computer Science Communications*, 8:149–163, 1986.

[16] Richard P. Brent, Richard E. Crandall, Karl Dilcher, and Christopher van Halewyn. Three new factors of Fermat numbers. *Mathematics of Computation*, 69(231):1297–1304, 2000.

[17] Richard P. Brent and John M. Pollard. Factorization of the eighth Fermat number. *Mathematics of Computation*, 36:627–630, 1981. `http://wwwmaths.anu.edu.au/~brent/pub/pub061.html`.

[18] E. Brockmeyer, C. Ghez, J. D'Eer, F. Catthoor, and H. De Man. Parametrizable behavioral IP module for a data-localized low-power FFT. In *Proc. IEEE Workshop on Signal Processing Systems (SIPS)*, pages 635–644, Taipeh, Taiwan, 1999. IEEE Press.

[19] Răzvan Bărbulescu. Familles de courbes elliptiques adaptées à la factorisation des entiers. Technical report, LORIA, 2009. `http://hal.inria.fr/inria-00419218/fr/`.

[20] Joe P. Buhler, Hendrik W. Lenstra, Jr., and Carl Pomerance. Factoring integers with the number field sieve. In Lenstra and Lenstra [60], pages 50–94.

[21] Stefania Cavallar. Strategies in Filtering in the Number Field Sieve. In *Algorithmic Number Theory, 4th International Symposium*, volume 1838 of *Lecture Notes in Computer Science*, pages 209–231. Springer-Verlag, 2000.

[22] Stefania Cavallar. Three-Large-Primes Variant of the Number Field Sieve. Technical Report MAS-R0219, Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands, August 2002.

[23] Stefania Cavallar, B. Dodson, Arjen K. Lenstra, Paul C. Leyland, Walter M. Lioen, Peter L. Montgomery, Brian Murphy, Herman J. J. Te Riele, and Paul Zimmermann. Factorization of RSA-140 Using the Number Field Sieve. In Kwok Yan Lam, Eiji Okamoto, and Chaoping Xing, editors, *Advances in Cryptology, Asiacrypt '99*, volume 1716 of *Lecture Notes in Computer Science*, pages 195–207. Springer-Verlag, 1999.

[24] Stefania Cavallar, Bruce Dodson, Arjen K. Lenstra, Walter Lioen, Peter L. Montgomery, Brian Murphy, Herman te Riele, Karen Aardal, Jeff Gilchrist, Gérard Guillerm, Paul Leyland, Jöel Marchand, François Morain, Alec Muffett, Chris and Craig Putnam, and Paul Zimmermann. Factorization of a 512-Bit RSA Modulus. In G. Goos, J. Hartmanis, and J. van Leeuwen, editors, *Advances in Cryptology – EUROCRYPT 2000*, volume 1807 of *Lecture Notes in Computer Science*, pages 1–18. Springer-Verlag, 2000.

[25] Henri Cohen. *A Course in Computational Algebraic Number Theory*. Springer-Verlag, 1st edition, 1995.

[26] Scott Contini. Factor world. `http://www.crypto-world.com/FactorWorld.html`.

[27] Scott P. Contini. Factoring Integers with the Self-Initializing Quadratic Sieve. Master's thesis, University of Georgia, 1997.

[28] Stephen A. Cook. *On the Minimum Computation Time of Functions*. PhD thesis, Harvard University, 1966.

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

[30] James W. Cooley and John W. Tukey. On the Origin and Publication of the FFT Paper. *This Week's Citation Classics*, (51–52):8–9, 1993.

[31] Don Coppersmith. Solving homogeneous linear equations over $gf(2)$ via block wiedemann algorithm. *Mathematics of Computation*, 62(205):333–350, 1994.

[32] Jean-Marc Couveignes. Computing a square root for the number field sieve. In Lenstra and Lenstra [60], pages 95–102.

[33] Richard Crandall and Carl Pomerance. *Prime Numbers: A Computational Perspective*. Springer-Verlag, 2nd edition, 2005.

[34] Giacomo de Meulenaer, François Gosset, Guerric Meurice de Dormale, and Jean-Jacques Quisquater. Elliptic Curve Factorization Method : Towards Better Exploitation of Reconfigurable Hardware. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM07)*, pages 197–207. IEEE Computer Society Press, 2007.

[35] Karl Dickman. On the Frequency of Numbers Containing Prime Factors of a Certain Relative Magnitude. *Arkiv för Matematik, Astronomi och Fysik*, 22A:1–14, 1930.

[36] Whitfield Diffie and Martin E. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.

[37] Harold M. Edwards. A normal form for elliptic curves. *Bulletin of the American Mathematical Society*, 44:393–422, 2007.

[38] Reina-Marije Elkenbracht-Huizing. *Factoring integers with the Number Field Sieve*. PhD thesis, Rijksuniversiteit te Leiden, 1997.

[39] Paul Zimmermann et al. The ECMNET Project. `http://www.loria.fr/~zimmerma/records/ecmnet.html`.

[40] Jens Franke and Thorsten Kleinjung. Continued fractions and lattice sieving. `http://www.math.uni-bonn.de/people/thor/confrac.ps`.

[41] Jens Franke, Thorsten Kleinjung, Christof Paar, Jan Pelzl, Christine Priplata, and Colin Stahlke. SHARK: A Realizable Special Hardware Sieving Device for Factoring 1024-Bit Integers. In *Cryptographic Hardware and Embedded Systems — CHES 2005*, volume 3659 of *Lecture Notes in Computer Science*, pages 119–130, 2005.

[42] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. special issue on "Program Generation, Optimization, and Platform Adaptation".

[43] Martin Fürer. Faster integer multiplication. In *STOC '07: Proceedings of the thirty-ninth annual ACM symposium on Theory of computing*, pages 57–66, New York, NY, USA, 2007. ACM.

[44] Kris Gaj, Soonhak Kwon, Patrick Baier, Paul Kohlbrenner, Hoang Le, Mohammed Khaleeluddin, and Ramakrishna Bachimanchi. Implementing the Elliptic Curve Method of Factoring in Reconfigueable Hardware. In Louis Goubin and Mitsuru Matsui, editors, *Cryptographic Hardware and Embedded Systems - CHES 2006*, volume 4249 of *Lecture Notes in Computer Science*, pages 119–133. Springer-Verlag, 2006.

[45] Pierrick Gaudry, Jérémie Detrey, Guillaume Hanrot, Alexander Kruppa, François Morain, Emmanuel Thomé, and Paul Zimmermann. Crible algébrique: distribution, optimisation (CADO-NFS). `http://cado.gforge.inria.fr/index.en.html`.

[46] Pierrick Gaudry, Alexander Kruppa, and Paul Zimmermann. A GMP-based implementation of Schönhage-strassen's large integer multiplication algorithm. In *ISSAC '07: Proceedings of the 2007 International Symposium on Symbolic and Algebric Computation*, pages 167–174, Waterloo, Canada, July–August 2007. ACM.

[47] W. Morven Gentleman and Gordon Sande. Fast Fourier transforms–for fun and profit. In *AFIPS Conference Proceedings*, volume 29, pages 563–578, Washington, 1966. Spartan Books.

[48] Daniel M. Gordon. Discrete Logarithms in $GF(p)$ using the Number Field Sieve. *SIAM Journal on Discrete Mathematics*, 6(1):124–138, 1993.

[49] Torbjörn Granlund. GNU MP: The GNU Multiple Precision Arithmetic Library, 2009. `http://gmplib.org/`.

[50] Torbjörn Granlund and Peter L. Montgomery. Division by invariant integers using multiplication. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 61–72, New York, NY, USA, 1994. ACM.

[51] Anatolii Karatsuba and Yuri P. Ofman. Multiplication of multidigit numbers on automata. *Doklady Akademii Nauk SSSR*, 145(2):293–294, 1962.

[52] Anatolii Karatsuba and Yuri P. Ofman. Multiplication of multidigit numbers on automata. *Soviet Physics Doklady*, 7(7):595–596, 1963.

[53] Thorsten Kleinjung. Cofactorisation strategies for the number field sieve and an estimate for the sieving step for factoring 1024 bit integers. In *Special-purpose Hardware for Attacking Cryptographic Systems (SHARCS'06)*, 2006.

[54] Thorsten Kleinjung. On Polynomial Selection for the General Number Field Sieve. *Mathematics of Computation*, 75(256):2037–2047, 2006.

[55] Thorsten Kleinjung, Kazumaro Aoki, Arjen K. Lenstra, Emmanuel Thomé, Joppe W. Bos, Pierrick Gaudry, Alexander Kruppa, Peter L. Montgomery, Dag Arne Osvik, Herman te Riele, Andrey Timofeev, and Paul Zimmermann. Factorization of a 768-bit RSA modulus. `http://eprint.iacr.org/2010/006`.

[56] Donald E. Knuth. *The Art of Computer Programming, volume 2: Seminumerical Algorithms.* Addison-Wesley, third edition, 1997.

[57] Donald E. Knuth and Luis Trabb Pardo. Analysis of a Simple Factorization Algorithm. *Theoretical Computer Science*, 3(3):321–348, 1976.

[58] Alexander Kruppa. Optimising the Enhanced Standard Continuation of the P–1 Factoring Algorithm. Diploma thesis, Technische Universität München, München, Germany, May 2005.

[59] D. H. Lehmer and R. E. Powers. On Factoring Large Numbers. *Bulletin of the American Mathematical Society*, 37:770–776, 1931.

[60] Arjen K. Lenstra and Hendrik W. Lenstra, Jr., editors. *The development of the number field sieve*, volume 1554 of *Lecture Notes in Mathematics*. Springer-Verlag, Berlin, 1993.

[61] Arjen K. Lenstra, Hendrik W. Lenstra, Jr., Mark S. Ma nasse, and John M. Pollard. The Factorization of the Ninth Fermat Number. *Mathematics of Computation*, 61(203):319–349, 1993.

[62] Hendrik W. Lenstra, Jr. Factoring Integers with Elliptic Curves. *Annals of Mathematics*, 126(3):649–673, 1987.

[63] George Marsaglia, Arif Zaman, and John C. W. Marsaglia. Numerical Solution of Some Classical Differential-Difference Equations. *Mathematics of Computation*, 53(187):191–201, July 1989.

[64] Peter L. Montgomery. Modular Multiplication Without Trial Division. *Mathematics of Computation*, 44(170):519–521, 1985.

[65] Peter L. Montgomery. Speeding the Pollard and Elliptic Curve Methods of Factorization. *Mathematics of Computation*, 48:243–264, 1987.

[66] Peter L. Montgomery. Evaluating Recurrences of Form $X_{m+n} = f(X_m, X_n, X_{m-n})$ Via Lucas Chains. Unpublished Manuscript, 1992. `ftp://ftp.cwi.nl/pub/pmontgom/Lucas.ps.gz`.

[67] Peter L. Montgomery. *An FFT Extension to the Elliptic Curve Method of Factorization*. PhD thesis, UCLA, 1992. `ftp://ftp.cwi.nl/pub/pmontgom/ucladissertation.psl.gz`.

[68] Peter L. Montgomery. Square Roots of Products of Algebraic Numbers. In Walter Gautschi, editor, *Mathematics of Computation 1943–1993: A Half-Century of Computational Mathematics*, volume 48 of *Proceedings of Symposia in Applied Mathematics*, 1994.

[69] Peter L. Montgomery. A Block Lanczos Algorithm for Finding Dependencies over GF(2). In G. Goos, J. Hartmanis, and J. van Leeuwen, editors, *Advances in Cryptology – EUROCRYPT '95*, volume 921 of *Lecture Notes in Computer Science*, pages 106–120. Springer-Verlag, 1995.

[70] Peter L. Montgomery. Personal Communication, 2001.

[71] Peter L. Montgomery. Five, Six, and Seven-Term Karatsuba-Like Formulae. *IEEE Transactions on Computers*, 54(3):362–369, 2005.

[72] Peter L. Montgomery. Personal Communication, 2008.

[73] Peter L. Montgomery and Alexander Kruppa. Improved Stage 2 to $p \pm 1$ Factoring Algorithms. In Alfred J. van der Poorten and Andreas Stein, editors, *Proceedings of the 8th Algorithmic Number Theory Symposium (ANTS VIII)*, volume 5011 of *Lecture Notes in Computer Science*, pages 180–195. Springer-Verlag, 2008.

[74] Peter L. Montgomery and Robert D. Silverman. An FFT Extension to the $p-1$ Factoring Algorithm. *Mathematics of Computation*, 54:839–854, 1990.

[75] Brian A. Murphy. *Polynomial Selection for the Number Field Sieve Integer Factorisation Algorithm.* PhD thesis, The Australian University, 1999.

[76] Phong Nguyen. A Montgomery-like Square Root for the Number Field Sieve. In Joe Buhler, editor, *Algorithmic Number Theory, Third International Symposium*, volume 1423 of *Lecture Notes in Computer Science*, 1998.

[77] Henri J. Nussbaumer. *Fast Fourier Transform and Convolution Algorithms.* Springer-Verlag, 2nd edition, 1982.

[78] Jan Pelzl, Martin Šimka, Thorsten Kleinjung, Jens Franke, Christine Priplata, Colin Stahlke, Miloš Drutarovský, Viktor Fischer, and Christof Paar. Area-Time Efficient Hardware Architecture for Factoring Integers with the Elliptic Curve Method. *IEE Proceedings Information Security*, 152(1):67–78, 2005.

[79] Colin Percival. Rapid multiplication modulo the sum and difference of highly composite numbers. *Mathematics of Computation*, 72(241):387–395, 2003.

[80] John M. Pollard. Theorems on factorisation and primality testing. *Proceedings of the Cambridge Philosophical Society*, 5:521–528, 1974.

[81] John M. Pollard. A Monte Carlo method for factorization. *BIT Numerical Mathematics*, 15(3):331–334, 1975.

[82] John M. Pollard. Factoring with cubic integers. In Lenstra and Lenstra [60], pages 4–10.

[83] John M. Pollard. The lattice sieve. In Lenstra and Lenstra [60], pages 43–49.

[84] Carl Pomerance. The Quadratic Sieve Factoring Algorithm. In Thomas Beth, Norbert Cot, and Ingemar Ingemarsson, editors, *Advances in Cryptology: Proceedings of EUROCRYPT 84*, volume 209, pages 169–182. Springer-Verlag, 1985.

[85] Ronald L. Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public key encryption. *Communications of the ACM*, 21(2):120–126, 1978.

[86] Manuel Rubio, P. Gómez, and Karim Drouiche. A new superfast bit reversal algorithm. *International Journal of Adaptive Control and Signal Processing*, 16:703–707, 2002.

[87] Arnold Schönhage. Asymptotically fast algorithms for the numerical multiplication and division of polynomials with complex coeficients. In *EUROCAM '82: Proceedings of the European Computer Algebra Conference on Computer Algebra*, pages 3–15, London, UK, 1982. Springer-Verlag.

[88] Arnold Schönhage. Sqrt2 in SSA. Personal communications, January 2008.

[89] Arnold Schönhage, Andreas F. W. Grotefeld, and Ekkehart Vetter. *Fast Algorithms – A Multitape Turing Machine Implementation.* BI Wissenschaftsverlag, 1994.

[90] Arnold Schönhage and Volker Strassen. Schnelle Multiplikation großer Zahlen. *Computing*, 7(3–4):281–292, 1971.

[91] Joseph J. Silverman and John Tate. *Rational Points on Elliptic Curves.* Springer-Verlag, 1995.

[92] Robert D. Silverman. The Multiple Polynomial Quadratic Sieve. *Mathematics of Computation*, 48(177):329–339, 1987.

[93] Allan Steel. Magma V2.12-1 is up to 2.3 times faster than GMP 4.1.4 for large integer multiplication, 2005. `http://magma.maths.usyd.edu.au/users/allan/intmult.html`.

[94] Allan Steel. Reduce everything to multiplication. Presented at Computing by the Numbers: Algorithms, Precision, and Complexity, Matheon Workshop 2006 in honor of the 60th birthday of Richard Brent, July 2006.

[95] Hiromi Suyama. Informal preliminary report (8). Letter to Richard P. Brent, October 1985.

[96] Gérald Tenenbaum. *Introduction to analytic and probabilistic number theory.* Cambridge University Press, first edition, 1995.

[97] Andrei L. Toom. The Complexity of a Scheme of Functional Elements Realizing the Multiplication of Integers. *Doklady Akademii Nauk SSSR*, 150(3):496–498, 1963.

[98] Andrei L. Toom. The Complexity of a Scheme of Functional Elements Realizing the Multiplication of Integers. *Soviet Mathematics Doklady*, 3:714–716, 1963.

[99] Joachim von zur Gathen and Jürgen Gerhard. *Modern Computer Algebra.* Cambridge University Press, 2nd edition, 2003.

[100] André Weimerskirch and Christof Paar. Generalizations of the Karatsuba Algorithm for Efficient Implementations. Technical report, Ruhr-Universität Bochum, 2003. `http://weimerskirch.org/papers/Weimerskirch_Karatsuba.pdf`.

[101] Hugh C. Williams. A $p+1$ Method of Factoring. *Mathematics of Computation*, 39:225–234, 1982.

[102] George Woltman and Scott Kurowski. The Great Internet Mersenne Prime Search. `http://www.gimps.org/`.

[103] Paul Zimmermann and Bruce Dodson. 20 Years of ECM. In Florian Hess, Sebastian Pauli, and Michael Pohst, editors, *Proceedings of the 7th Algorithmic Number Theory Symposium (ANTS VII)*, volume 4076 of *Lecture Notes in Computer Science*, pages 525–542. Springer-Verlag, 2006.

Abstract

This thesis explores improvements to well-known algorithms for integer multiplication and factorization.

The Schönhage-Strassen algorithm for integer multiplication, published in 1971, was the first to achieve complexity $O(n \log(n) \log(\log(n)))$ for multiplication of n-bit numbers and is still among the fastest in practice. It reduces integer multiplication to multiplication of polynomials over finite rings which allow the use of the Fast Fourier Transform for computing the convolution product. In joint work with Gaudry and Zimmermann, we describe an efficient implementation of the algorithm based on the GNU Multiple Precision arithmetic library, improving cache utilization, parameter selection and convolution length for the polynomial multiplication over previous implementations, resulting in nearly 2-fold speedup.

The P–1 and P+1 factoring algorithms find a prime factor p of a composite number quickly if $p - 1$, respectively $p + 1$, contains no large prime factors. They work in two stages: the first step computes a high power g_1 of an element g_0 of a finite group defined over $\mathbb{F}_p$, respectively $\mathbb{F}_{p^2}$, the second stage looks for a collision of powers of g_1 which can be performed efficiently via polynomial multi-point evaluation. In joint work with Peter Lawrence Montgomery, we present an improved stage 2 for these algorithms with faster construction of the required polynomial and very memory-efficient evaluation, increasing the practical search limit for the largest permissible prime in $p - 1$, resp. $p + 1$, approximately 100-fold over previous implementations.

The Number Field Sieve (NFS) is the fastest known factoring algorithm for "hard" integers where the factors have no properties that would make them easy to find. In particular, the modulus of the RSA encryption system is chosen to be a hard composite integer, and its factorization breaks the encryption. Great efforts are therefore made to improve NFS in order to assess the security of RSA accurately. We give a brief overview of the NFS and its history. In the sieving phase of NFS, a great many smaller integers must be factored. We present in detail an implementation of the P–1, P+1, and Elliptic Curve methods of factorization optimized for high-throughput factorization of small integers. Finally, we show how parameters for these algorithms can be chosen accurately, taking into account the distribution of prime factors in integers produced by NFS to obtain an accurate estimate of finding a prime factor with given parameters.

Keywords: Arithmetic, Integer Multiplication, Integer Factoring, Elliptic Curves, Number Field Sieve

Résumé

Cette thèse propose des améliorations aux problèmes de la multiplication et de la factorisation d'entier.

L'algorithme de Schönhage-Strassen pour la multiplication d'entier, publié en 1971, fut le premier à atteindre une complexité de $O(n \log(n) \log(\log(n)))$ pour multiplier deux entiers de n bits, et reste parmi les plus rapides en pratique. Il réduit la multiplication d'entier à celle de polynôme sur un anneau fini, en utilisant la transformée de Fourier rapide pour calculer le produit de convolution. Dans un travail commun avec Gaudry et Zimmermann, nous décrivons une implantation efficace de cet algorithme, basée sur la bibliothèque GNU MP; par rapport aux travaux antérieurs, nous améliorons l'utilisation de la mémoire cache, la sélection des paramètres et la longueur de convolution, ce qui donne un gain d'un facteur 2 environ.

Les algorithmes P–1 et P+1 trouvent un facteur p d'un entier composé rapidement si $p-1$, respectivement $p+1$, ne contient pas de grand facteur premier. Ces algorithmes comportent deux phases : la première phase calcule une grande puissance g_1 d'un élément g_0 d'un groupe fini défini sur $\mathbb{F}_p$, respectivement $\mathbb{F}_{p^2}$, la seconde phase cherche une collision entre puissances de g_1, qui est trouvée de manière efficace par évaluation-interpolation de polynômes. Dans un travail avec Peter Lawrence Montgomery, nous proposons une amélioration de la seconde phase de ces algorithmes, avec une construction plus rapide des polynômes requis, et une consommation mémoire optimale, ce qui permet d'augmenter la limite pratique pour le plus grand facteur premier de $p-1$, resp. $p+1$, d'un facteur 100 environ par rapport aux implantations antérieures.

Le crible algébrique (NFS) est le meilleur algorithme connu pour factoriser des entiers dont les facteurs n'ont aucune propriété permettant de les trouver rapidement. En particulier, le module du système RSA de chiffrement est choisi de telle sorte, et sa factorisation casse le système. De nombreux efforts ont ainsi été consentis pour améliorer NFS, de façon à établir précisément la sécurité de RSA. Nous donnons un bref aperçu de NFS et de son historique. Lors de la phase de crible de NFS, de nombreux petits entiers doivent être factorisés. Nous présentons en détail une implantation de P–1, P+1, et de la méthode ECM basée sur les courbes elliptiques, qui est optimisée pour de tels petits entiers. Finalement, nous montrons comment les paramètres de ces algorithmes peuvent être choisis finement, en tenant compte de la distribution des facteurs premiers dans les entiers produits par NFS, et de la probabilité de trouver des facteurs premiers d'une taille donnée.

Mots-clés: Arithmétique, multiplication des entiers, factorisation des entiers, courbes elliptiques, crible algébrique

(English abstract on inside back cover)

Département de formation doctorale en informatique **École doctorale IAEM Lorraine**
UFR STMIA

Améliorations de la multiplication et de la factorisation d'entier

THÈSE

présentée et soutenue publiquement le 28 janvier 2010

pour l'obtention du

Doctorat de l'université Henri Poincaré – Nancy 1
(spécialité informatique)

par

Alexander Kruppa

Composition du jury

Rapporteurs :	Karim Belabas (Professeur, Université Bordeaux 1) Antoine Joux (Professeur associé (P.A.S.T.), Université de Versailles)
Membres du jury :	Isabelle Debled-Rennesson (Maître de conférences, Université Henri Poincaré) Marc Joye (Cryptographer, Thomson R&D) Arjen Lenstra (Professeur, EPFL) Herman te Riele (Directeur de Recherche, CWI) Paul Zimmermann (Directeur de Recherche, INRIA)

Laboratoire Lorrain de Recherche en Informatique et ses Applications — UMR 7503

Mis en page avec la classe thloria.

Table des matières

Table des figures

Liste des tableaux

1 Introduction

La multiplication des entiers est utilisée dans pratiquement tous les algorithmes arithmétiques, et les problèmes en théorie algorithmique des nombres demandent en particulier des multiplications entre nombres entiers très grands. La factorisation d'entiers est un des problèmes fondamentaux dans la théorie des nombres et a gagné une importance pratique remarquable avec l'apparition du système de chiffrement à clé publique RSA dont la sécurité dépend de la difficulté du problème de la factorisation. Cette thèse présente des améliorations à l'algorithme de Schönhage-Strassen de multiplication d'entiers, aux méthodes P–1 et P+1 de factorisation, qui trouvent rapidement des facteurs premiers p lorsque $p-1$ ou $p+1$ n'ont pas de grand facteur premier, et au crible algébrique (NFS, Number Field Sieve en anglais) qui est l'algorithme le plus rapide asymptotiquement pour factoriser des entiers composés qui n'ont pas de facteur premier facile à trouver, comme ceux qu'on trouve dans les applications cryptographiques.

La multiplication d'entiers se trouve partout, et la multiplication de longs entiers arrive suffisamment fréquemment dans le calcul scientifique qu'il est surprenant que le premier algorithme dont la complexité binaire est strictement inférieure à $O(n^2)$, complexité demandée par la méthode naïve, n'a été découvert qu'en 1962. Cette année, Karatsuba et Ofman [13] ont montré comment réduire la multiplication de nombres de n bits à trois multiplications de nombres de $n/2$ bits, en atteignant une complexité asymptotique de $O(n^{1.585})$. Une année plus tard, Toom [27] a généralisé l'algorithme de Karatsuba et Ofman en exprimant d'abord la multiplication des entiers en termes de multiplication de polynômes, puis en utilisant l'évaluation des polynômes, la multiplication point à point, et l'interpolation pour calculer le polynôme produit. Ceci permet de réduire le calcul à $2k-1$ multiplications de n/k bits chacune, avec un coût asymptotique en $O(n^{\log_k(2k-1)})$ pour k fixé. En principe, ceci permet n'importe quel exposant $1+\epsilon$ dans la fonction de coût asymptotique, cependant un k grand n'est pas efficace pour des entrées de tailles observées en pratique car le coût de l'évaluation et de l'interpolation domine. En 1971, Schönhage et Strassen [25] ont essentiellement résolu le problème de la multiplication rapide d'entiers en utilisant la transformée de Fourier rapide (FFT, Fast Fourier Transform en anglais), introduite par Cooley et Tukey en 1965 [8], pour effectuer le produit de convolution sous-jacent. La complexité de leur méthode est $O(n\log(n)\log(\log(n)))$ pour multiplier des nombres de n bits. Plusieurs bibliotheques arithmétiques multiprécision offrent une fonction de multiplication rapide d'entiers. Parmi ces bibliotheques se trouve la bibliotheque *GNU Multiple Precision* (GMP) [12], développée principalement par Torbjörn Granlund. Elle est largement utilisée et connue pour être exceptionnellement rapide, grâce au choix soucieux des algorithmes et à une implantation très optimisée. Elle utilise l'algorithme de Schönhage-Strassen pour la multiplication d'entiers longs. Une application où la multiplication de grands entiers de GMP est utilisée de manière intensive est GMP-ECM [29], une implémentation des méthodes P–1, P+1, et de la méthode des courbes elliptiques (ECM, Elliptic Curve Method en anglais), développée principalement par Paul Zimmermann.

La factorisation des entiers est un problème ancien en théorie des nombres. Actuellement ce n'est plus un problème d'intérêt exclusivement académique. En fait, il est devenu une préoccupation d'une importance économique certaine avec la publication du très célèbre système de chiffrement à clé publique RSA (Rivest-Shamir-Adleman) [23], dont la sécurité répose sur la difficulté du problème de la factorisation des grands entiers. Le petit théorème de Fermat nous dit que

$$a^{p-1} \equiv 1 \pmod{p}$$

pour tout nombre premier p tel que $p \nmid a$. Euler a étendu ce théorème aux nombres composés, de sorte que

$$a^{\phi(N)} \equiv 1 \pmod{N}$$

lorsque $\gcd(a, N) = 1$, où $\phi(N)$ est la fonction indicatrice d'Euler, définie par

$$\phi(p_1^{\nu_1} \cdots p_k^{\nu_k}) = (p_1 - 1)p_1^{\nu_1 - 1} \cdots (p_k - 1)p_k^{\nu_k - 1}$$

où $p_1, \dots, p_k$ sont des nombres premiers distincts et $\nu_1, \dots, \nu_k$ des entiers positifs. Ainsi, étant donnée la factorisation de N en nombres premiers, $\phi(N)$ se calcule facilement. RSA utilise une clé publique pour chiffrer. Cette clé est constituée par un entier impair composé $N = pq$ – appelé *module* –, où p et q sont des nombres premiers d'à peu près la même taille, et d'un exposant public e. Pour déchiffrer, il utilise une clé privée, constituée du même module N, ainsi que d'un entier d tel que

$$de \equiv 1 \pmod{\phi(N)},$$

c'est-à-dire, $de = k\phi(N) + 1$ pour un certain entier k. RSA chiffre un message exprimé par un entier $1 < m < N$ par l'opération

$$c = m^e \bmod N,$$

et déchiffre c pour retrouver le message original par l'opération

$$c^d \bmod N = m^{de} \bmod N = m^{k\phi(N)+1} \bmod N = m.$$

La factorisation du module N permet de retrouver efficacement d. Factoriser N revient donc à casser le système de chiffrement. Les clés de RSA doivent donc être choisies de telle manière qu'il soit pratiquement impossible de factoriser le module pendant la durée de vie qu'on prétend lui donner, et ce avec des ressources de calcul importantes, en utilisant les meilleurs algorithmes connus.

Les algorithmes de factorisation peuvent se diviser en deux classes : algorithmes ad hoc, et algorithmes génériques. Les premiers utilisent des propriétés spécifiques des facteurs premiers du nombre reçu en entrée, typiquement leur taille, et le temps d'exécution de ces algorithmes dépend moins de la taille du nombre reçu en entrée (il en dépend à peu près de la même manière que la multiplication des entiers), que des propriétés de ses facteurs premiers.

Le temps d'exécution des algorithmes génériques dépend presque exclusivement de la taille du nombre reçu en entrée, et non pas des propriétés de ses facteurs premiers.

La taille des clés RSA est choisie de sorte à ce que les clés soient résistantes aux algorithmes ad hoc, et donc que leur sécurité dépende uniquement de la performance des algorithmes génériques.

Le meilleur algorithme générique connu pour factoriser un module RSA est le crible algébrique (NFS, Number Field Sieve en anglais), avec une complexité conjecturée en $L_N[1/3, (64/9)^{1/3}]$, où

$$L_x[\alpha, c] = e^{(c+o(1))\log(x)^\alpha \log(\log(x))^{1-\alpha}}.$$

Le coût de factorisation à l'aide du crible algébrique est le critère principal pour évaluer la sécurité associée aux tailles des clés RSA.

Malgré leur impuissance pour de casser un module RSA, les algorithmes ad hoc présentent encore un grand intérêt, d'abord pour factoriser des nombres qui ne sont pas des clés RSA et qui peuvent avoir des facteurs plus faciles à trouver, et puis en tant que sous-routine du crible algébrique.

Un concept central dans les algorithmes modernes de factorisation est la *friabilité* : un entier est dit B-friable lorsqu'aucun de ses facteurs premiers n'excède B, et B-superfriable si aucune des puissances de nombre premier le divisant n'excède B.

L'algorithme P–1 publié par Pollard en 1974 [21] a été le premier d'une classe d'algorithmes de factorisation ad hoc qui trouvent un facteur premier p de N rapidement si l'ordre d'un certain groupe fini défini sur $\mathbb{F}_p$ est friable. Dans le cas de l'algorithme P–1, le groupe est simplement le groupe des unités de $\mathbb{F}_p$, qui est d'ordre $p - 1$.

L'étape 1 de cet algorithme choisit un entier x_0 premier avec N et calcule $x_0^e \bmod N$, où e est un multiple de chaque nombre premier et puissance de premier n'excédant pas une borne B_1 choisie. Si $p - 1$ est B_1-superfriable et donc divise e, alors $x_0^e \equiv 1 \pmod{p}$, et $\text{pgcd}(x_0^e - 1, N)$ revèle p sauf lorsque ce pgcd est composé. Pollard propose en plus une étape 2 qui consiste à chercher une collision modulo p entre certaines puissances de x_0^e, ce qui lui permet de trouver des facteurs premiers p où $p - 1$ possède unique facteur premier entre B_1 et une deuxième borne B_2, mais dont le reste est B_1-superfriable. Il montre qu'un facteur premier p peut se trouver en temps $O\big(\sqrt{p}M(\log(N))\big)$ si la détection de la collision est faite par évaluation polynomiale multi-points, avec une routine de multiplication rapide basée sur la FFT.

Williams [28] étend l'idée P–1 de Pollard à la méthode P+1 qui trouve un nombre premier p rapidement si $p - 1$ ou $p + 1$ est friable. Parfois, les méthodes P–1 et P+1 trouvent des facteurs d'une taille surprenante si $p - 1$ ou $p + 1$ est suffisamment friable, mais ces méthodes sont inutiles si les deux contiennent un grand facteur premier. Par exemple, une proportion d'environ $1 - \log(2)$ des entiers plus petits que N est $N^{1/2}$-friable, donc à peu près la moitié des nombres premiers p de 40 chiffres ont un facteur premier qui excède 20 chiffres simultanément dans $p - 1$ et $p + 1$, ce qui fait que ces premiers ne sont pas traitables avec aucune de ces deux méthodes.

L'idée de Pollard d'une étape 2 asymptotiquement rapide, incluant FFT, pour l'algorithme P–1 a été implémentée pour la première fois par Montgomery et Silverman [20]. Les auteurs ont suggéré plusieurs idées pour accélérer encore plus leur algorithme, et pour l'adapter à la méthode P+1.

L'algorithme ad hoc de factorisation le plus rapide est connu sous le nom de *méthode des courbes elliptiques* (ECM, Elliptic Curve Method en anglais) et a été introduit par H.W. Lenstra Jr. [17]. Il peut être vu comme une généralisation de P–1 et de P+1 dans le sens qu'il travaille dans le groupe associé à une courbe elliptique définie sur $\mathbb{F}_p$ avec un ordre de groupe dans $[p - 2\sqrt{p} + 1, p + 2\sqrt{p} + 1]$, et qui dépend des paramètres de la courbe. Cette approche a comme avantage principal que l'on peut essayer différentes courbes jusqu'à ce que l'on trouve une courbe avec un ordre de groupe friable. Avec un choix optimal des paramètres, ECM a une complexité conjecturée de $L_p[1/2, 2]$, pour trouver un facteur premier p, ce qui dans le pire des cas $p \approx \sqrt{N}$, implique une complexité de $L_N[1/2, 1]$, ce qui en fait le seul algorithme ad hoc dont le temps d'exécution est sous-exponentiel. Brent [6] et Montgomery [18] ont présenté des extensions pour l'étape 2 d'ECM, et Montgomery [20] a développé une étape 2 qui utilise la FFT.

Malgré le fait qu'ECM a une complexité asymptotique très supérieure à celle des méthodes P–1 et P+1, et que ces deux méthodes fonctionnent, dans un certain sens, comme deux essais particuliers pour trouver un groupe d'ordre friable parmi l'infinité d'essais possibles permis par ECM, les méthodes anciennes ont des avantages qui les rendent compétitives. Un de ces avantages est leur vitesse pure. L'arithmétique pendant l'étape 1 est beaucoup plus simple pour P–1 et P+1 que pour ECM de sorte qu'avec des paramètres comparables, le temps processeur utilisé est moindre. Un autre avantage est que pour P–1 et P+1, une étape 2 beaucoup plus rapide, fondée sur la FFT, est possible, puisque $\mathbb{Z}/N\mathbb{Z}$ (ou une extension quadratique de celui-ci pour P+1) a une structure d'anneau, ce qui n'est pas le cas pour le groupe des points d'une courbe elliptique. La structure d'anneau permet d'utiliser un algorithme d'évaluation polynomial multi-points très efficace, en permettant que l'étape 2 fonctionne avec beaucoup moins de temps processeur et de mémoire que ce qui est possible pour l'extension FFT de ECM. Finalement, pour certaines entrées, on sait que $p-1$ possède un diviseur n (e.g., les nombres cyclotomiques $\phi_n(x)$ pour n, $x \in \mathbb{N}$, sauf si $p \mid n$), ce qui augmente la probabilité de friabilité de $(p-1)/n$. Ces avantages font conclure qu'il est très raisonnable de donner une chance à P–1 et à P+1 avant de passer à la plus puissante méthode ECM.

À présent, la méthode la plus rapide pour attaquer des modules RSA (ou d'autres nombres sans petits facteurs ou sans facteurs trouvables par d'autres moyens) est le crible algébrique (NFS, Number Field Sieve en anglais). Il a été proposé par Pollard en 1988 [22] et initialement demandait que l'entier à factoriser soit d'une forme algébrique simple, telle que $a^n \pm c$ avec a et c petits, mais dans les années suivantes il a été étendu pour factoriser des entiers quelconques [7]. On conjecture que le crible algébrique factorise un entier N en temps $L_N[1/3, c]$ avec $c = (32/9)^{1/3}$ pour des nombres de certaines formes simples, auquel cas l'algorithme est appelé crible algébrique spécial (SNFS, Special Number Field Sieve en anglais) ; ou avec $c = (64/9)^{1/3}$ pour des entiers quelconques, auquel cas l'algorithme est appelé crible algébrique général (GNFS, General Number Field Sieve en anglais). Un premier succès pour SNFS a été la factorisation du neuvième nombre de Fermat $F_9 = 2^{2^9} + 1$ en 1990 [16] et pour GNFS la factorisation d'un nombre de 130 chiffres décimaux, en l'occurrence le nombre RSA-130 du défi RSA (RSA challenge en anglais) en 1996. En janvier 2010, la factorisation de plus grande taille réalisée avec SNFS était celle de $2^{1039} - 1$, effectuée en 2007 [2], alors que la meilleure performance de GNFS était la factorisation d'un nombre RSA de 232 chiffres, en 2009 [14].

De même que d'autres algorithmes de factorisation génériques, le crible algébrique factorise un entier N en cherchant des congruences de carrés de la forme $x^2 \equiv y^2 \pmod{N}$ avec $x \not\equiv \pm y \pmod{N}$, à partir desquelles un facteur non-trivial de N est obtenu en calculant $\gcd(x + y, N)$. Les valeurs de x et y se trouvent en collectionnant un grand ensemble de "relations," qui sont essentiellement des paires d'entiers friables, dont on peut choisir un sous-ensemble dont le produit par composantes est de sorte que chaque premier apparaît avec un exposant pair, ce qui revient à dire que le produit en question est un nombre carré.

La partie la plus gourmande en temps pour le crible algébrique (dans ses deux versions, spécial et général) est la phase de collecte de relations qui consiste à examiner un nombre très grand de valeurs de polynômes pour trouver des valeurs friables en utilisant des techniques de crible et d'autres algorithmes de factorisation. Ici, un entier n est considéré friable s'il contient uniquement des facteurs premiers n'excédant pas une certaine borne B, sauf peut-être pour k entiers n'excédant pas une borne L, dite des grands premiers (large prime bound en anglais). La routine de crible renvoie les entiers n où le cofacteur qui reste après avoir divisé par les premiers plus petits que B est suffisamment petit, par exemple en dessous de L^k. Ces cofacteurs doivent être factorisés pour tester s'il y a un facteur premier excédant L.

La consommation mémoire pour le crible croît avec B, et pour des factorisations de grande échelle la mémoire disponible limite fréquemment B de sorte que L et parfois même k doivent être incrémentés pour permettre que suffisamment de valeurs soient considérées comme friables. De cette manière, un très grand nombre de cofacteurs apparaissent pendant le crible algébrique, et des algorithmes optimisés pour la factorisation de petits nombres doivent être utilisés pour éviter que la cofactorisation ne devienne le goulot d'étranglement du processus de crible. La méthode des courbes elliptiques ou ECM est fréquemment suggérée à ce propos [5] [10], et les méthodes P–1 et P+1 sont aussi des bon candidats.

Contributions

En collaboration avec Pierrick Gaudry et Paul Zimmermann, nous avons développé une implémentation améliorée de l'algorithme de multiplication d'entiers de Schönhage-Strassen, basée sur la version 4.1.4 du code GMP, écrit par Paul Zimmermann. La nouvelle implémentation améliore la localité des accès au cache mémoire pendant la phase FFT, augmente la longueur potentielle de la convolution pour une taille donnée, et fait un choix très soigneux des longueurs de convolution ainsi que d'autres paramètres qui dépendent de la taille des nombres donnés en entrée. L'implémentation est décrite dans la section 2. Ces améliorations ont abouti à une amélioration d'un facteur 2 par rapport au code GMP 4.1.4.

Par ailleurs, en collaboration avec Montgomery, nous avons implémenté une version améliorée de l'étape 2 des algorithmes P–1 et P+1 qui implémente les idées de [20], ainsi que d'autres améliorations. L'implémentation est basée sur GMP-ECM et on la décrit dans la section 3.

Une bibliotheque pour des factorisations rapides d'entiers d'au plus 38 chiffres décimaux, en utilisant les algorithmes P–1, P+1 et ECM, a été écrite pour être utilisée dans l'implémentation du crible algébrique développée dans le cadre du projet CADO (Crible algébrique : distribution, optimisation). Un aperçu du programme de factorisation de petits entiers se trouve dans la section 4, et son efficacité est comparée à des implémentations matérielles de ECM en tant que routine dédiée pour le crible algébrique.

2 Multiplication entière par l'algorithme de Schönhage-Strassen

2.1 Introduction

L'algorithme de multiplication entière de Schönhage et Strassen (SSA) [25] fut le premier à atteindre une complexité en $O(N \log(N) \log(\log(N)))$ pour la multiplication de nombres de N bits. Il exprime la multiplication de deux entiers a, b comme le produit de deux polynômes $A(x), B(x)$ tels que $a = A(2^M)$ et $b = B(2^M)$ pour un certain entier M, et utilise la transformée de Fourier rapide (FFT) pour en calculer le produit de convolution et ainsi obtenir $C(x) = A(x)B(x)$, duquel on retrouve le produit entier via $ab = C(2^M)$. Les auteurs décrivent deux variantes : l'une utilise une FFT complexe en virgule flottante et l'autre une FFT sur l'anneau fini $R_n = \mathbb{Z}/\left(2^n + 1\right)\mathbb{Z}$. Les deux algorithmes sont utilisés en pratique, mais seul le second est considéré dans la suite de ce document.

L'algorithme réduit la multiplication entière à ℓ multiplications dans R_n, pour ℓ et n bien choisis. Ces multiplications de taille réduite doivent être effectuées de manière efficace pour parvenir à la complexité annoncée, et SSA s'appelle lui-même récursivement dans ce but jusqu'à ce que les nombres à multiplier soient suffisamment petits pour que des méthodes plus simples puissent être appliquées, comme la multiplication quadratique classique. Pour permettre ces appels récursifs, SSA est conçu pour calculer un produit modulo $2^N + 1$ pour un certain N bien choisi. Pour calculer le produit de deux entiers a et b, on peut prendre $2^N + 1 > ab$ pour assurer un résultat correct.

Ainsi, SSA remplace le calcul de ab dans $\mathbb{N}$ par le calcul de $ab \bmod (2^N + 1)$ dans R_N, lui-même remplacé par un produit de polynômes $A(x)B(x)$ dans $\mathbb{Z}[x]/(x^\ell + 1)$, et enfin effectue ce produit de convolution dans $R_n[x]/(x^\ell + 1)$ tel que R_n se prête au calcul de convolution négacyclique par FFT pondérée de longueur ℓ (qui calcule le produit polynomial modulo $x^\ell + 1$) et enfin permette de relever les coefficients de $A(x)B(x) \bmod (x^\ell + 1)$. La figure 1 présente cette suite de transformations.

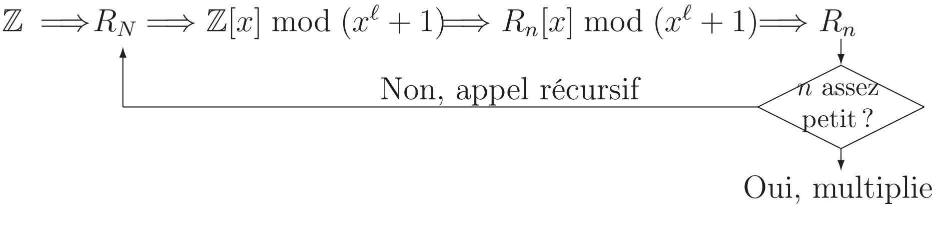
$\mathbb{Z} \Longrightarrow R_N \Longrightarrow \mathbb{Z}[x] \bmod (x^\ell + 1) \Longrightarrow R_n[x] \bmod (x^\ell + 1) \Longrightarrow R_n$
Non, appel récursif
n assez petit ?
Oui, multiplie

Figure 1 – Diagramme des applications dans l'algorithme de Schönhage-Strassen.

Pour a, b donnés et dont on cherche le produit modulo $2^N + 1$, avec $4 \mid N$, on écrit $N = \ell M$ où $\ell = 2^k$, $k \geq 2$, et on choisit un n tel que

$$\begin{array}{rcll} n & = & \ell m, & m \in \mathbb{N}, \\ n & \geq & 2M + k + 1. & \end{array} \tag{1}$$

Les deux conditions impliquent que $n > \sqrt{2N}$. Le choix des bonnes valeurs pour N, ℓ et n est crucial pour garantir les performances de SSA ; cette sélection est détaillée dans 2.2.

Puisque $2^n = (2^m)^\ell \equiv -1 \pmod{2^n{+}1}$, 2^m est une racine primitive (2ℓ)-ième de l'unité dans R_n, de telle sorte qu'il supporte la FFT pour calculer des convolutions circulaires de longueur une puissance de deux jusqu'à 2ℓ, ou la FFT pondérée pour des convolutions négacycliques de longueur au plus ℓ.

À partir des entiers à multiplier $0 \leq a, b < 2^N + 1$, on construit les polynômes

$$A(x) = \sum_{i=0}^{\ell-1} a_i x^i, \qquad 0 \leq a_i < 2^M \text{ for } 0 \leq i < \ell - 1 \qquad (2)$$
$$0 \leq a_{\ell-1} \leq 2^M,$$

c'est-à-dire, on découpe a en ℓ morceaux de M bits chacun, sauf le dernier morceau qui peut être égal à 2^M. En faisant de même pour b, on a $a = A(2^M)$ et $b = B(2^M)$. Pour obtenir le produit $c = ab \bmod (2^N + 1)$, on peut utiliser une convolution négacyclique pour calculer le polynôme produit $C(x) = A(x)B(x) \bmod (x^\ell + 1)$ de telle sorte que, lorsque les polynômes sont évalués en $x = 2^M$, le modulo $x^\ell + 1$ conserve les classes d'équivalence modulo $2^N + 1 = \left(2^M\right)^\ell + 1$ du produit sur les entiers.

Les coefficients du polynôme produit $C(x) = \sum_{i=0}^{\ell-1} c_i x^i$ peuvent être relevés de manière exacte depuis R_n si $c_i \bmod (2^n + 1)$ est unique pour toutes les valeurs possibles de chaque c_i. De par la structure négacyclique, chaque c_i peut être défini par

$$c_i = \sum_{0 \leq j \leq i} a_j b_{i-j} - \sum_{i < j < \ell} a_j b_{i-j+\ell}$$

de sorte que, en utilisant les bornes de (2), il suffit de choisir $2^n + 1 > 2 \cdot 2^{2M} \ell$, ou encore $n \geq 2M + k + 1$ avec $\ell = 2^k$. Ainsi les conditions sur n données dans (1) permettent d'effectuer le calcul de $C(x) = A(x)B(x) \bmod x^\ell + 1$ en une convolution négacyclique par une FFT pondérée sur l'anneau R_n.

Étant donnés les coefficients du polynôme produit $C(x)$, l'entier $c = C(2^M) \bmod (2^N + 1) = ab \bmod (2^N + 1)$ peut alors être calculé lors de l'étape de propagation de retenue finale.

SSA comprend ainsi cinq étapes consécutives, détaillées ci-dessous. Dans cet exemple, les coefficients c_i du polynôme produit viennent écraser les coefficients a_i dans le tableau $a[]$.

1. **Décomposition de a et b et pondération**
   Allouer de la mémoire pour le tableau $a[i]$, $0 \leq i < \ell$, avec au moins $n+1$ bits pour chaque élément
   Affecter le i-ème morceau de M bits de a à l'élément $a[i]$
   Appliquer la pondération en affectant $a[i] := w^i \cdot a[i] \bmod (2^n + 1)$, $w = 2^{n/\ell}$
   Faire de même pour le tableau $b[]$
2. **Transformée de a et b**
   Effectuer une FFT de longueur ℓ en place sur $a[]$, en travaillant modulo $2^n + 1$, et avec la racine de l'unité $\omega = 2^{2n/\ell}$
   Faire de même pour $b[]$
3. **Multiplication dans le domaine fréquentiel**
   Affecter $a[i] := a[i] \cdot b[i] \bmod (2^n + 1)$ pour $0 \leq i < \ell$

4. **Transformée inverse**
   Effectuer une FFT inverse de longueur ℓ en place sur $a[]$, en travaillant modulo $2^n + 1$, ainsi que la division par ℓ
5. **Pondération inverse, propagation de retenue**
   Supprimer la pondération en affectant $a[i] := w^{-i} \cdot a[i] \bmod (2^n + 1)$, $w = 2^{n/\ell}$
   Calculer la somme $c = \left(\sum_{i=0}^{\ell-1} a_i 2^{iM}\right) \bmod \left(2^N + 1\right)$

La plupart des idées concernant SSA présentées jusqu'ici sont tirées des articles de Schönhage et Strassen [25], et des articles suivants de Schönhage [24]. Nous décrivons dans la suite plusieurs améliorations pratiques permettant d'utiliser SSA pour multiplier rapidement de très grands entiers sur des ordinateurs actuels. L'implémentation est basée sur l'implémentation de SSA dans la *GNU Multiple Precision arithmetic library* (GMP) [12], version 4.2.1.

2.2 Améliorations

Amélioration de la longueur de la FFT

Puisque tous les facteurs premiers de $2^n + 1$ sont $p \equiv 1 \pmod 8$ si $4 \mid n$, 2 est un résidu quadratique dans R_n, et il se trouve que l'expression de $\sqrt{2}$ est suffisamment simple pour la rendre utile en tant que racine de l'unité d'ordre une puissance de deux. Plus précisément, $\left(\pm 2^{3n/4} \mp 2^{n/4}\right)^2 \equiv 2 \pmod{2^n + 1}$, que l'on peut vérifier aisément en développant le carré. Ainsi, étant donné $n = 2^k m$, $k \geq 2$, on peut utiliser $\sqrt{2} = 2^{3n/4} - 2^{n/4}$ comme racine de l'unité d'ordre 2^{k+2} pour doubler la longueur possible de la FFT. Dans le cas d'une convolution négacyclique, ceci permet d'effectuer une transformée de longueur 2^{k+1}, où $\sqrt{2}$ n'est utilisée que dans la pondération. Pour une convolution circulaire, $\sqrt{2}$ est utilisée normalement comme une racine de l'unité lors de la transformée, offrant une longueur de transformée de 2^{k+2}.

Amélioration de l'utilisation du cache

Lorsque l'on multiplie deux grands entiers avec SSA, le temps consacré à l'accès aux données lors du calcul des transformées de Fourier est non-négligeable ; d'autant plus que les opérations effectuées sur ces données sont peu coûteuses, rendant le coût relatif des accès-mémoires assez élevé.

Dans un calcul de FFT, l'opération principale est l'opération dite *butterfly* dans un anneau R_n qui, pour l'algorithme de Gentleman et Sande, calcule $a + b$ et $(a - b)\omega$, où a et b sont des coefficients dans R_n et ω est une racine de l'unité. Dans SSA, cette racine de l'unité est une puissance de 2 (possiblement la racine carrée de 2), de telle sorte que ce calcul ne requiert qu'additions, soustractions et décalages.

De sorte à améliorer la localité des données pour de grandes transformées, nous avons essayé des stratégies présentées dans la littérature ; parmi les plus efficaces, nous avons trouvé la transformée en base 2^k et l'algorithme de Bailey.

Le principe des transformations en grande base est d'utiliser une opération atomique qui regroupe plusieurs *butterflies*. Dans son livre [3], Arndt donne une description de plusieurs variantes selon ce principe. La FFT classique peut être vue comme une transformée en base 2. Le niveau supérieur est une transformée en base 4, où l'opération atomique a 4 entrées et 4 sorties (sans compter les racines de l'unité) et regroupe les 4 *butterflies* de deux étapes consécutives de la FFT. Puisque nous calculons 2 étapes en base 2 à chaque fois, la profondeur de la récursion est réduite d'un facteur jusqu'à 2. Dans l'opération atomique, chaque entrée est utilisée dans deux *butterflies*, de sorte que le nombre de défauts de cache est inférieur à 50%.

Bailey [4] a presenté un autre algorithme de FFT qui améliore le support du cache. Dans cet article, le lecteur pourra trouver des références permettant de remonter à l'idée originale, qui était en fait déjà mentionnée dans [11]. Pour simplifier, nous la désignons dans la suite sous le nom d'« algorithme de Bailey ».

Une manière de voir l'algorithme de Bailey est comme une transformée en base $\sqrt{\ell}$, où $\ell = 2^k$ est la longueur des entrées. Autrement dit, au lieu de regrouper 2 étapes comme en base 4, on regroupe $k/2$ étapes. De manière plus générale, notons $k = k_1 + k_2$, où k_1 et k_2 sont proches de $k/2$. L'algorithme de Bailey procède alors en deux étapes :

1. effectuer 2^{k_2} transformées de longueur 2^{k_1} ;
2. effectuer 2^{k_1} transformées de longueur 2^{k_2}.

Nous cherchons des nombres tels que $\sqrt{\ell}$ coefficients ne tiennent pas dans le cache L1, mais que pour tous les nombres que nous souhaitons multiplier, ils tiennent dans le cache L2. Ainsi, la structure du code suit la hiérarchie mémoire : l'algorithme de Bailey découpe les données en morceaux qui tiennent dans le cache L2, et la transformée en base 4 découpe encore les données en morceaux qui tiennent dans le cache L1.

Nous avons implémenté cette approche (avec un mécanisme de seuil pour n'employer l'algorithme de Bailey que pour des grands nombres), et en la combinant avec la transformée en base 4 nous avons obtenus nos meilleures performances. Nous avons aussi essayé la transformée en base 8, ainsi que des versions de l'algorithme de Bailey en plus grande dimension, en particulier en 3 étapes de taille $\sqrt[3]{\ell}$. Cependant, cela n'a pas apporté d'amélioration pour les tailles considérées.

Un autre moyen d'améliorer la localité des données est de combiner différentes étapes de l'algorithme pour travailler le plus possible sur un même jeu de données tant qu'il est dans le cache. Une amélioration simple suivant ce principe est de combiner la multiplication point-à-point et les transformées, en particulier lorsque l'algorithme de Bailey est utilisé. Dans la transformée du second nombre, après chaque transformée de taille 2^{k_2}, ces coefficients peuvent être multipliés par les coefficients (déjà transformés) du premier nombre, puis immédiatement utilisés pour la transformation inverse de taille 2^{k_2}, toujours en restant dans le cache L2. En poursuivant cette idée plus loin, on peut aussi combiner les étapes de décomposition et de recomposition avec les transformées, économisant ainsi encore un passage sur les données.

Multiplication sans réduction modulo $2^N + 1$

La raison pour laquelle SSA utilise une convolution négacyclique est que cela permet d'utiliser l'algorithme récursivement : les sous-produits modulo $2^n + 1$ peuvent à leur tour être calculés en utilisant le même algorithme. Un inconvénient de cette approche est qu'elle requiert une transformée pondérée, c'est-à-dire des opérations supplémentaires préalables à la transformée avant et après la transformée inverse, ainsi qu'une racine (2ℓ)-ième de l'unité pour les poids, ce qui divise par deux la longueur possible d'une transformée pour un n donné. La transformée négacyclique n'est nécessaire que pour pouvoir utiliser modulo $x^\ell + 1$ dans la multiplication polynomiale qui soit compatible avec le modulo $2^N + 1$ de la multiplication entière. Or, au plus haut niveau de récursion, on choisit N de sorte que le produit entier $c = ab$ ne soit pas affecté par une réduction modulo, et aucun modulo n'est en fait nécessaire pour les multiplications entière comme polynomiale. Ainsi, on peut utiliser une transformée de Fourier normale sans pondération pour calculer $ab \pmod{2^N - 1}$, qui nous donnera le bon résultat du moment que $ab < 2^N - 1$. Comme nous n'avons pas besoin d'une racine (2ℓ)-ième de l'unité pour la pondération, on peut doubler la taille de la transformée pour un même R_n. On peut alors calculer un produit $ab \pmod{2^N - 1}$ et un autre $ab \pmod{2^{aN} + 1}$ pour un entier positif a tel que $ab \geq 2^{aN} - 1$ mais $ab < (2^N - 1)(2^{aN} + 1)$, puis recontruire le résultat correct via le théorème des restes chinois.

Choix des paramètres et réglage automatique

SSA prend pour n un multiple de ℓ, de sorte que $\omega = 2^{2n/\ell}$ est une racine primitive ℓ-ème de l'unité, et $w = 2^{n/\ell}$ est utilisé pour la pondération (ou, si l'on utilise $\sqrt{2}$, $\omega = 2^{n/\ell}$ et $w = (\sqrt{2})^{n/\ell}$; pour raisons de simplicité, nous omettons ce cas dans la suite). Nous avons trouvé que des améliorations conséquentes pouvaient être obtenues en utilisant de meilleures méthodes de réglage, décrites ici, qui déterminent de bonnes valeurs de ℓ et n pour une taille de produit donnée.

Jusqu'à la version 4.2.1, GMP utilisait une méthode de choix de paramètres naïve pour la multiplication par FFT. Pour la transformée de Fermat modulo $2^N + 1$, une FFT de longueur 2^k était utilisée pour $t_k \leq N < t_{k+1}$, où t_k est la plus petite taille (en bits) pour laquelle la FFT-2^k est plus rapide que la FFT-2^{k-1}. Par exemple, sur un Opteron, le fichier par défaut `gmp-mparam.h` utilise $k = 4$ en dessous de 528 mots machine, puis $k = 5$ pour moins de 1184 mots machine, et ainsi de suite :

```
#define MUL_FFT_TABLE  { 528, 1184, 2880, 5376, 11264, 36864, 114688,
  327680, 1310720, 3145728, 12582912, ... }
```

Cette méthode n'est pas optimale, puisque les courbes entre les différentes FFT-2^k se croisent plusieurs fois, comme indiqué Figure 2.

Pour prendre ces multiples croisements en compte, la nouvelle méthode de sélection de paramètres détermine les intervalles $[m_1, m_2]$ (en nombre de mots) où la FFT de longueur 2^k est plus avantageuse :

```
#define MUL_FFT_TABLE2 {{1, 4 /*66*/}, {401, 5 /*96*/},
  {417, 4 /*98*/}, {433, 5 /*96*/}, {865, 6 /*96*/}, ...
```

Ces nombres signifient que de 401 mots jusqu'à 416 mots, la FFT de longueur 2^5 est plus intéressante. Puis de 417 à 432 mots on préfèrera une longueur de 2^4, et entre 433 et 864 mots c'est à nouveau 2^5 qu'il faudra choisir.

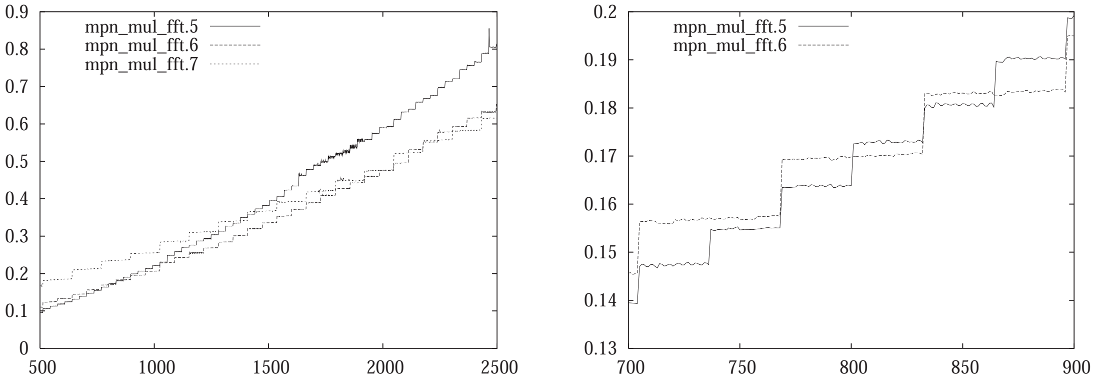

FIGURE 2 – Temps en millisecondes pour multiplier des nombres modulo $2^n + 1$ avec une FFT de longueur 2^k pour $k = 5, 6, 7$. L'agrandissement sur la droite (seulement pour $k = 5, 6$) illustre les croisements multiples de ces courbes.

Comme décrit en 2.2, nous utilisons des produits modulo $2^N - 1$ et modulo $2^{Na} + 1$ pour calculer des produits entiers (c'est-à-dire lors du plus haut niveau de récursion de SSA). Pour chaque taille, la meilleure valeur de a est là aussi déterminée par notre méthode de réglage des paramètres :

```
#define MUL_FFT_FULL_TABLE2 {{16, 1}, {4224, 2}, {4416, 6}, {4480, 2},
  {4608, 4}, {4640, 2}, ...
```

Par exemple, le couple `{4608, 4}` signifie que pour multiplier deux nombres de 4608 mots chacun — ou dont le produit a 2 × 4608 mots — jusqu'à 4639 mots chacun, le nouvel algorithme utilise une transformée modulo $2^N - 1$ et une transformée modulo $2^{4N} + 1$. La reconstruction par le théorème des restes chinois est facile puisque $2^{aN} + 1 \equiv 2 \bmod (2^N - 1)$.

2.3 Résultats

Le 1er juillet 2005, Allan Steel mit en ligne une page web [26] intitulée « *Magma V2.12-1 is up to 2.3 times faster than GMP 4.1.4 for large integer multiplication* », ce qui était une motivation pour améliorer l'implémentation de GMP. Nous comparons ici nos résultats avec les performances de Magma.

Comme montré Figure 3, le nouveau code est environ deux fois plus rapide que le code de GMP 4.1.4, et entre 30 et 40% plus rapide que GMP 4.2.1 qui implémentait déjà quelques-unes des idées présentées ici et était à peu près aussi rapide que Magma V2.13.6.

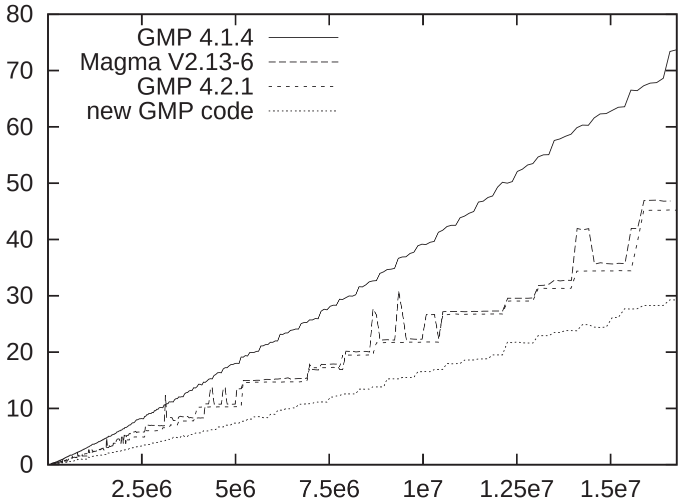

FIGURE 3 – Comparaison de GMP 4.1.4, GMP 4.2.1, Magma V2.13-6 et notre nouveau code pour la multiplication entière sur un Opteron à 2,4 GHz (nombre de mots de 64 bits en abscisse, temps de calcul en secondes en ordonnée).

3 Une version améliorée de l'étape 2 des algorithmes de factorisation P±1

3.1 L'algorithme P-1

En 1974 John Pollard [21, §4] introduit l'algorithme P–1 pour la factorisation d'entiers N impairs et composés. L'algorithme repose sur l'espoir de trouver un facteur premier p tel que $p-1$ soit friable. Un entier est dit *B-friable* lorsque tous ses facteurs premiers sont inférieurs à B. L'étape 1 de l'algorithme calcule $b_1 = b_0^e \bmod N$ où b_0 est un nombre (choisi aléatoirement) et où $e = \mathrm{lcm}(1, 2, 3, 4, \ldots, B_1)$ pour un certain paramètre B_1. Cette étape réussit lorsque $(p-1) \mid e$. Dans ce cas, d'après le petit théorème de Fermat, on a $b_1 \equiv 1 \pmod{p}$ et on peut alors généralement retrouver p à partir de $\gcd(b_1 - 1, N)$. Dans le cas contraire, on espère que $p - 1 = qn$ où $n \mid e$ et q n'est pas trop grand. Alors

$$b_1^q \equiv (b_0^e)^q = b_0^{eq} = (b_0^{nq})^{e/n} = \left(b_0^{p-1}\right)^{e/n} \equiv 1^{e/n} = 1 \pmod{p}, \tag{3}$$

de sorte que p divise $\gcd(b_1^q - 1,\, N)$.

L'étape 2 de l'algorithme P–1 essaie de trouver p, à condition que $B_1 < q \leq B_2$, où B_2 est un paramètre utilisé pour limiter la recherche. La fin de l'article de Pollard [21] suggère une version de type FFT pour la phase 2 de l'algorithme P–1. Montgomery et Silverman [20, p. 844] l'ont implantée, en utilisant un convolution cyclique pour évaluer un polynôme en des points en progression géométrique.

3.2 L'algorithme P+1

Hugh Williams [28] a introduit un algorithme de factorisation « P+1 » en 1982. Cet algorithme trouve facilement un facteur premier p de N lorsque $p+1$ est friable. Il s'inspire de l'algorithme P–1. L'étape 1 choisit un nombre $P_0 \in \mathbb{Z}/N\mathbb{Z}$ aléatoirement et calcule $P_1 = V_e(P_0) \pmod N$, où (à nouveau) $e = \mathrm{lcm}(1, 2, 3, 4, 5, \ldots, B_1)$ et où $V_n(x)$ est un polynôme de Chebyshev satisfaisant $V_n(x + 1/x) = x^n + 1/x^n$. Ces polynômes sont à coefficients entiers et peuvent être évalués efficacement en $O(\log(n))$ multiplications, par exemple en utilisant les méthodes de Montgomery [19]. Si $P_0^2 - 4$ est un non-résidu quadratique modulo p, $\gcd(P_1 - 2, N)$ donne le facteur p lorsque $p + 1$ divise e. Lorsqu'aucun facteur n'est trouvé par ce pgcd, on espère que $p + 1 = qn$ où n divise e et avec q pas trop grand. Dans ce cas, $\gcd(V_q(P_1) - 2, N)$ donne p, à condition que $P_0^2 - 4$ soit un non-residu.

3.3 Nouvelle approche pour l'étape 2

Nous nous intéressons ici uniquement à l'étape 2 de l'algorithme P–1 ; les modifications à apporter à l'algorithme P+1 sont décrites dans le manuscrit de thèse. Nous cherchons un nombre premier q avec $B_1 < q \le B_2$ et tel que

$$b_1^q \equiv 1 \pmod p \tag{4}$$

où p est un facteur premier du nombre N donné en entrée de l'algorithme. De la même façon que dans [20], nous évaluons un polynôme en des points en progressions géométriques. Nous construisons un polynôme

$$f(X) = X^{-|S_1|/2} \prod_{k_1 \in S_1} (X - b_1^{2k_1}) \bmod N \tag{5}$$

et nous évaluons

$$f\left(b_1^{2k_2+(2m+1)P}\right) \bmod N \tag{6}$$

pour toutes les valeurs $k_2 \in S_2$ et $m_1 \le m < m_2$, avec P, S_1, S_2, m_1 et m_2 choisis de telle sorte que tout nombre premier q, $B_1 < q \le B_2$ s'écrive $q = 2k_2 + (2m + 1)P - 2k_1$. Alors, si l'équation (4) est vérifiée, on a

$$f\left(b_1^{2k_2+(2m+1)P}\right) \equiv 0 \pmod p \tag{7}$$

car le facteur $X - b_1^{2k_1}$ de $f(X)$ vaut alors

$$\begin{aligned} b_1^{2k_2+(2m+1)P} - b_1^{2k_1} &\equiv \\ b_1^{2k_2+(2m+1)P-2k_1} - 1 &\equiv \\ b_1^q - 1 &\equiv 0 \pmod q. \end{aligned}$$

Alors, $\gcd(f(b_1^{2k_2+(2m+1)P}), N)$ fournit p.

Pour un P donné, en utilisant une décomposition de $(\mathbb{Z}/P\mathbb{Z})^*$ en somme de petites progressions arithmétiques, nous construisons les ensembles S_1 et S_2. Cette étape est décrite en section 3.4. À partir de ces progressions arithmétiques il est possible d'engendrer rapidement les coefficients de $f(X)$, comme expliqué en section 3.6. Les polynômes qui apparaissent dans ce calcul sont les polynômes de Laurent réciproques que l'on peut multiplier efficacement à l'aide d'un produit de convolution cyclique pondéré décrit en section 3.5. L'évaluation multipoints est présentée brièvement en section 3.7. Finalement, la section 3.9 donne quelques résultats nouveaux, en particulier un facteur P+1 record de 60 chiffres.

3.4 Sélection de S_1 et S_2

Nous notons avec l'opérateur « + » l'addition de deux ensembles : $S + T = \{a + b : a \in S, b \in T\}$. D'après le théorème des restes chinois,

$$(\mathbb{Z}/(mn)\mathbb{Z})^* = n(\mathbb{Z}/m\mathbb{Z})^* + m(\mathbb{Z}/n\mathbb{Z})^* \text{ si } \gcd(m, n) = 1. \tag{8}$$

Cette équation est indépendante des représentants choisis : si $S \equiv (\mathbb{Z}/m\mathbb{Z})^* \pmod m$ et $T \equiv (\mathbb{Z}/n\mathbb{Z})^* \pmod n$, alors $nS + mT \equiv (\mathbb{Z}/(mn)\mathbb{Z})^* \pmod{mn}$.

Nous choisissons S_1 et S_2 de telle sorte que $S_1 + S_2 \equiv (\mathbb{Z}/P\mathbb{Z})^* \pmod P$ ce qui assure que tous le nombres premiers avec P (donc en particulier tous les nombres premiers) soient couverts par l'intervalle de l'étape 2. Le nombre P est choisi très friable de façon à assurer que $|S_1 + S_2| = \phi(P)$ reste petit. Nous désirons exprimer S_1 comme la somme de nombreux petits ensembles afin de rendre la construction de $f(X)$ en section 3.6 la plus rapide possible. À cette fin, nous pouvons décomposer chacun des $(\mathbb{Z}/p\mathbb{Z})^*$ (avec $p \mid P$) en sommes d'ensembles plus petits.

Soit $R_n = \{2i - n - 1 : 1 \le i \le n\}$ un ensemble de n nombres en progression arithmétique de raison 2 et centrée en 0. Pour tout nombre premier impair p, R_{p-1} constitue une représentation possible de $(\mathbb{Z}/p\mathbb{Z})^*$. Dès lors que $p \neq 3$, la cardinal de R_{p-1} est composé et il est possible de le décomposer de la façon suivante en progressions arithmétiques dont les tailles sont des nombres premiers :

$$R_{mn} = R_m + mR_n. \tag{9}$$

Par exemple, si $P = 105$, on peut choisir

$$S_1 + S_2 = 35\{-1, 1\} + 42\{-1, 1\} + 21\{-1, 1\} + 45\{-1, 1\} + 15\{-2, 0, 2\}.$$

Toutes les progressions arithmétiques considérées sont symétriques par rapport à 0 et les racines du polynôme $f(X)$ dans l'équation (5) peuvent donc être regroupées en paires formées d'un nombre et de son inverse. Puisque $|S_1 + S_2| = \phi(P)$ est pair quand $P > 1$, il est toujours possible de choisir S_1 de telle sorte que $|S_1|$ soit pair. De cette façon, $f(X)$ est un polynôme de Laurent réciproque (cf. section suivante). Un tel polynôme peut être stocké et multiplié bien plus facilement qu'un polynôme quelconque. Le degré de $f(X)$ est $|S_1|$: ce paramètre est limité par la mémoire disponible. De plus $|S_2|$ évaluations multipoints distinctes doivent être effectuées.

3.5 Polynômes de Laurent réciproques et convolution pondérée

On définit les polynômes de Laurent réciproques (ou RLP de l'anglais *reciprocal Laurent polynomial*) comme étant les expansions en x de la forme

$$a_0 + \sum_{j=1}^{d} a_j \left(x^j + x^{-j}\right)$$

où les a_j sont des éléments d'un anneau. Un tel polynôme est dit *unitaire* lorsque $a_d = 1$. Si $a_d \neq 0$, on dit que le degré de ce polynôme est $2d$; le degré d'un polynôme de Laurent réciproque est donc toujours pair. Un RLP unitaire de degré $2d$ est représenté par d coefficients (le coefficient dominant $a_d = 1$ n'a pas besoin d'être stocké). On appelle *base standard* des RLP de degré au plus $2d$, la base $\{1\} \cup \{x^j + x^{-j} : 1 \leq j \leq d\}$.

Considérons un RLP $Q(x)$ de degré au plus $2d_q$: $Q(x) = q_0 + \sum_{j=1}^{d_q} q_j \left(x^j + x^{-j}\right)$. De la même façon, on suppose que $R(x)$ est un RLP de degré au plus $2d_r$. Pour former le produit S de P et Q, $S(x) = Q(x)R(x) = s_0 + \sum_{j=1}^{d_s} s_j \left(x^j + x^{-j}\right)$ (qui sera de degré au plus $2d_s = 2(d_q + d_r)$), on choisit une longueur $\ell > d_s$ et on effectue un produit de convolution cyclique pondéré $\tilde{S}(wx) = Q(wx)R(wx) \bmod \left(x^\ell - 1\right)$, où w est un poids bien choisi tel que $w^\ell \neq 0, \pm 1$. En toute généralité, un produit de degré d nécessite une convolution de longueur $\ell > d$, c'est-à-dire deux fois plus grand que les facteurs du produit.

Supposons que $\tilde{S}(x) = \sum_{j=0}^{\ell-1} \tilde{s} x^j$ et $\tilde{S}(wx) = S(wx) \bmod \left(x^\ell - 1\right)$. Alors,

$$\begin{aligned}
\tilde{S}(wx) &= s_0 + \sum_{j=1}^{d_s} \left(w^j s_j x^j + w^{-j} s_j x^{\ell-j}\right) \\
&= \sum_{j=0}^{d_s} w^j s_j x^j + \sum_{j=\ell-d_s}^{\ell-1} w^{j-\ell} s_{\ell-j} x^j \\
&= \sum_{j=0}^{\ell-d_s-1} w^j s_j x^j + \sum_{j=\ell-d_s}^{d_s} w^j \left(s_j + w^{-\ell} s_{\ell-j}\right) x^j + \sum_{j=d_s+1}^{\ell-1} w^{j-\ell} s_{\ell-j} x^j
\end{aligned}$$

et donc $\tilde{s}_j = s_j$ pour $0 \leq j < \ell - d_s$, et $\tilde{s}_j = s_j + w^{-\ell} s_{\ell-j}$ pour $\ell - d_s \leq j \leq d_s$. De là, on peut déduire s_j et $s_{\ell-j}$ en résolvant le système linéaire

$$\begin{pmatrix} 1 & w^{-\ell} \\ w^{-\ell} & 1 \end{pmatrix} \begin{pmatrix} s_j \\ s_{\ell-j} \end{pmatrix} = \begin{pmatrix} \tilde{s}_j \\ \tilde{s}_{\ell-j} \end{pmatrix}.$$

Lorsque cette matrice est inversible, c'est-à-dire lorsque $w^\ell \neq \pm 1$, il existe une unique solution :

$$\begin{aligned}
s_j &= \frac{w^{-\ell}\tilde{s}_j - \tilde{s}_{\ell-j}}{w^{-2\ell} - 1} \\
s_{\ell-j} &= \tilde{s}_{\ell-j} - w^{-\ell} s_j.
\end{aligned}$$

3.6 Calcul des coefficients de f

Le RLP unitaire $f(X)$ de l'équation (5), dont les racines sont les b_1^{2k} avec $k \in S_1$, peut être construit rapidement un utilisant la décomposition de S_1. Supposons que S_1 et S_2 aient été construits de la façon que nous avons décrite en section 3.4 : S_1 est de la forme $S_1 = T_1 + T_2 + \cdots + T_m$ où les T_j sont formés d'une progression arithmétique de taille un nombre premier et centrée en 0. Si $|S_1|$ est pair, au moins l'un des T_j est de cardinal égal à 2. Quitte à réorganiser les T_j, on peut supposer que c'est T_1 et que $|T_2| \geq |T_3| \geq \cdots \geq |T_m|$.

Notons $T_1 = \{-k_1, k_1\}$: on définit alors un RLP unitaire de degré 2

$$F_1(X) = X + X^{-1} - b_1^{2k_1} - b_1^{-2k_1} \pmod{N}.$$

Soit à présent $1 \leq j < m$. À partir des coefficients d'un RLP unitaire $F_j(X)$ de degré d_j et de racines $b_1^{2k_1}$ ($k_1 \in T_1 + \cdots + T_j$), on définit

$$F_{j+1}(X) = \prod_{k_2 \in T_{j+1}} F_j(b_1^{2k_2} X) \pmod{N}. \tag{10}$$

Ce polynôme peut être calculé efficacement en multipliant les coefficients de $F_j(X)$ par des puissances de $b_1^{2k_2}$ en temps $O(d_j)$ pour tous les $k_2 \in T_{j+1}$, puis en multipliant les polynômes ainsi obtenus les uns avec les autres avec la méthode décrite en section 3.5. Le temps total est $O(M(d_j{\cdot}|T_{j+1}|)|T_{j+1}|)$, où $M(n)$ désigne le temps nécessaire pour multiplier deux RLP de degré au plus n. Puisque $|S_1|$ contient beaucoup de petits facteurs premiers, en particulier une puissance élevée du nombre 2, le temps de calcul est dominé par les dernières multiplications : 2 polynômes de degré $|S_1|/2$ multipliés entre eux à l'étape m, 2 polynômes de degré $|S_1|/4$ multipliés entre eux à l'étape $m-1$, etc. Le coût total du calcul de $f(X)$ est finalement $O(M(|S_1|))$. À titre de comparaison, en toute généralité, le calcul d'un polynôme (donné par ses racines) en utilisant un arbre binaire a une complexité asymptotique de $O(M(n)\log(n))$. Le calcul récursif utilisant la factorisation de S_1 nous permet donc d'économiser un facteur $\log(|S_1|)$ sur le temps de calcul.

3.7 Évaluation polynomiale multipoints

Nous avons construits un RLP unitaire $f(X)$ de degré $|S_1|$: notons-le

$$f(X) = f_0 + \sum_{j=1}^{|S_1|/2} f_j \cdot (X^j + X^{-j}) = \sum_{j=-|S_1|/2}^{|S_1|/2} f_j X^j,$$

où $f_j = f_{-j} \in \mathbb{Z}/N\mathbb{Z}$.

L'équation (6) nécessite de calculer $f\left(b_1^{2k_2+(2m+1)P}\right)$ pour tous les $k_2 \in S_2$ et pour un certain nombre de valeurs consécutives m : $m_1 \leq m < m_2$.

En posant $r = b_1^P \in \mathbb{Z}/N\mathbb{Z}$ et $x_0 = b_1^{2k_2+(2m_1+1)P}$, les arguments de f s'écrivent $x_0 r^{2m}$ avec $0 \leq m < m_2 - m_1$. Ces points sont en progression géométrique de raison r^2. Pour une valeur de k_2 donnée, il est possible d'évaluer f en tous ces points à l'aide d'une unique convolution de longueur $\ell = m_2 - m_1 + |S_1| + 1$ et coût d'initialisation de $O(\ell)$ [1, exercise 8.27].

Plus précisément, on définit le RLP

$$h(X) = \sum_{j=-|S_1|/2}^{|S_1|/2} r^{-j^2} f_j X^j.$$

Cette définition ne fait pas référence à x_0, de sorte que h est calculé une fois pour toutes et sera utilisé pour les évaluations ultérieures de f.

On définit également

$$g(X) = \sum_{i=0}^{\ell-1} x_0^{M-i}\, r^{(M-i)^2} X^i.$$

Ce polynôme en revanche est recalculé pour chaque nouvelle évaluation.

Pour chaque évaluation multi-points, h et g sont multipliés avec une transformation cyclique de longueur ℓ, c'est-à-dire modulo $X^\ell - 1$. Le coefficient de $X^{\ell-1-|S_1|/2-m}$ (avec $0 \le m < m_2 - m_1$) dans $g(X)h(X) \bmod (X^\ell - 1)$ est

$$\sum_{j=-|S_1|/2}^{|S_1|/2} x_0^m r^{m^2} \left(x_0 r^{2m}\right)^j f_j = x_0^m r^{m^2} f(x_0 r^{2m}), \tag{11}$$

qui contient le terme $f(x_0 r^{2m})$ désiré. Comme x_0 et r sont premiers entre eux et que nous sommes intéressés uniquement par $\gcd(f(x_0\, r^{2m}),\, N)$, le facteur $x_0^m\, r^{m^2}$ n'a aucune importance. Après chaque évaluation multipoints, on forme le produit de ces coefficients et on en prend le pgcd avec N, dans l'espoir de trouver un facteur.

3.8 Notre implémentation

Notre implémentation repose sur GMP-ECM, qui est une implémentation des algorithmes P–1, P+1, et de la méthode ECM (*Elliptic Curve Method*) pour la factorisation d'entiers. GMP-ECM est développée principalement par Paul Zimmermann [29] et repose sur la bibliothèque d'arithmétique en précision arbitraire GMP [12].

Sur un Opteron cadencé à 2.4 GHz avec 8 Go de RAM, l'étape 2 de l'algorithme P–1 prend 1738 secondes pour traiter un nombre de 230 chiffres (un cofacteur de $2^{254} + 1$), avec $B_2 = 1.2 \cdot 10^{15}$. Si on utilise le *multi-threading* pour exploiter les deux processeurs présents sur cette machine, le même calcul prend 941 secondes (1753 secondes de temps processeur). À titre de comparaison, l'implémentation précédente présente dans GMP-ECM [29] met 34080 secondes pour réaliser ce calcul sur la même machine en utilisant un seul processeur.

3.9 Quelques résultats

Nous avons fait tourné les algorithmes P–1 ou P+1 sur environ 1500 entiers composés. Les paramètres B_1 and B_2 variaient d'une fois sur l'autre ; des valeurs typiques sont par exemple $B_1 = 10^{11}$ et $B_2 = 10^{16}$. Le tableau 1 indique des facteurs nouveaux p sur quelques exemples. À chaque fois, le plus grand facteur de $p \pm 1$ est également indiqué.

Le facteur de $47^{146} + 1$ (52 chiffres) et le facteur du nombre de Lucas L_{2366} (60 chiffres) ont successivement constitué deux records pour l'algorithme de factorisation P+1. Le record précédent était un facteur de 48 chiffres de L_{1849} et et avait été découvert par nous-même en mars 2003.

Entrée Méthode	Facteur p découvert Plus grands facteurs de $p \pm 1$	Taille
$73^{109} - 1$ P–1	7622704004786371556832236715869572000643951815229 9 $12491 \cdot 37987 \cdot 156059 \cdot 2244509 \cdot 462832247372839$	c191 p50
$68^{118} + 1$ P–1	7506686348037740621097710183200476580505073749325089* $22807 \cdot 480587 \cdot 14334767 \cdot 89294369 \cdot 4649376803 \cdot 5380282339$	c151 p52
$24^{142} + 1$ P–1	20489047427450579051989683686453370154126820104624537 $4959947 \cdot 7216081 \cdot 16915319 \cdot 17286223 \cdot 12750725834505143$	c183 p53
$47^{146} + 1$ P+1	7986478866035822988220162978874631335274957495008401 $20540953 \cdot 56417663 \cdot 1231471331 \cdot 1632221953 \cdot 843497917739$	c235 p52
L_{2366} P+1	725516237739635905037132916171116034279215026146021770250523 $932677 \cdot 62754121 \cdot 19882583417 \cdot 751245344783 \cdot 483576618980159$	c290 p60

* = trouvé durant l'étape 1 de l'algorithme

Table 1 – Grands facteurs $P \pm 1$ découverts

4 Factorisation des petits entiers par P–1, P+1 et ECM dans le crible algébrique

Comme indiqué en introduction, la phase de crible de NFS [15] cherche un grand nombre d'entiers ne contenant que des facteurs premiers plus petits qu'une borne B, sauf certains — au plus k — pouvant aller jusqu'à une borne L, où B, k, et L sont des paramètres. L'ordre de grandeur est typiquement de 10^7 à 10^8 pour B, de l'ordre de 10^9 pour L, et de 2 à 3 pour k. Des factorisations records peuvent utiliser des valeurs plus grandes. Les facteurs premiers plus petits que B sont identifiés lors du crible, laissant des cofacteurs de 15 à 30 chiffres, qu'il faut factoriser pour déterminer s'ils contiennent des facteurs premiers plus grands que L.

Parmi les algorithmes possibles pour cela, on trouve les méthodes P–1, P+1, et la méthode des courbes elliptiques (ECM). Toutes ces méthodes ont en commun le fait qu'un facteur premier p est trouvé lorsque l'ordre d'un groupe défini sur $\mathbb{F}_p$ est lui-même friable.

Contrairement à la phase 2 rapide de P–1 et P+1 décrite en section 3, les facteurs premiers attendus ici sont trop petits pour des algorithmes asymptotiquement rapides. Les algorithmes utilisés sont plutôt optimisés pour un faible surcoût et un haut débit, par exemple en précalculant des chaînes d'addition pour la phase 1, ou un plan d'exécution optimisé pour la phase 2 de ces algorithmes. Ces algorithmes sont développés pour des nombres à factoriser d'un mot-machine de 64 bits (jusque 19 chiffres décimaux environ) et pour deux mots-machine moins deux bits (jusque 38 chiffres décimaux environ).

4.1 Algorithme P–1

La phase 1 de l'algorithme P–1 est essentiellement identique à la description faite en section 3.1. La valeur de e est précalculée pour différentes valeurs de B_1 et donnée en argument à la routine effectuant la phase 1, qui consiste simplement en une exponentiation modulaire, une soustraction et un pgcd. La base d'exponentiation choisie est $b_0 = 2$; dans l'algorithme d'exponentiation gauche-droite, cela ne requiert que des carrés et des multiplications par 2, qui pour ces dernières peuvent être calculées via une addition plutôt qu'une multiplication par x_0.

La phase 2 de P–1 peut utiliser exactement le même code que celle de P+1, et est décrite en section 4.4.

4.2 Algorithme P+1

L'algorithme P+1 est essentiellement comme décrit dans la section 3.1. Les polynômes de Chebyshev peuvent être évalués efficacement par la méthode de Montgomery [19]. Cela requiert de trouver des « chaînes de Lucas » quasi-optimales, qui sont un type spécial de chaînes d'addition. Montgomery montre comment trouver rapidement de telles chaînes, et lors de la factorisation de grands nombres il est possible de les produire à la volée. Pour des petits nombres, ce procédé est trop coûteux, et les chaînes sont précalculées pour diverses valeurs de B_1.

Si la phase 1 de P+1 échoue, on essaie de trouver un facteur avec la phase 2, voir la section 4.4.

4.3 Algorithme ECM

La méthode de factorisation via les courbes elliptiques (ECM) a été introduite par H. W. Lenstra, Jr., en 1987 [17]. ECM travaille dans le groupe de Mordell-Weil des points d'une courbe elliptique sur $\mathbb{F}_p$. Par le théorème de Hasse, l'ordre de ce groupe est compris dans l'intervalle $[p+1-2\sqrt{p}, p+1+2\sqrt{p}]$. ECM choisit une courbe E avec des coefficients dans $\mathbb{Z}/N\mathbb{Z}$ et calcule $P_1 = e \cdot P_0$ pour un point initial P_0 sur la courbe et $e = \text{lcm}(1, 2, 3, 4, \ldots, B_1)$. ECM trouve un facteur premier p de N si la courbe sur $\mathbb{F}_p$ a un ordre divisant e, car alors P_1 est le point à l'infini de la courbe sur $\mathbb{F}_p$, avec coordonnée Z égale à $0 \pmod p$, et $\gcd((P_1)_Z, N)$ révèle p.

L'implantation de la phase 1 d'ECM suit essentiellement la description de Montgomery [18]. Montgomery utilise des courbes elliptiques en coordonnées projectives pour éviter de coûteux inverses modulaires dans la formule d'addition. Cependant, ses courbes nécessitent des chaînes d'addition spéciales pour multiplier un point par un scalaire. Ces chaînes sont identiques à celles nécessaires pour P+1, et sont précalculées et stockées comme pour P+1.

Notre phase 2 d'ECM est similaire à celle de P–1 et P+1, et est décrite en section 4.4.

4.4 Phase 2 pour P–1, P+1 et ECM

La phase 2 de P–1, P+1 et ECM essaie de trouver un facteur p de N si l'ordre du groupe correspondant sur $\mathbb{F}_p$ est B_1-friable, excepté un facteur premier plus grand q. On cherche un tel q plus petit qu'une borne B_2 en cherchant une collision $b_1^m = b_1^n$ (dans le cas de P–1) qui survient quand $q \mid m - n$. Nous décrivons ici la phase 2 pour l'algorithme

P–1 ; P+1 peut utiliser le même algorithme en posant $X_1 = P_1$, et la phase 2 d'ECM a une structure très similaire.

Notre phase 2 suit la "enhanced standard continuation" décrite par Montgomery [18]. Pour des bornes données B_1 et B_2, et une entrée b_1, on définit $X_1 = b_1 + b_1^{-1}$, on choisit une valeur d divisible par 6, et on calcule deux listes

$$\begin{aligned} f_i &= V_{id}(X_1) \bmod N \text{ pour } \lfloor B_1/d \rceil \le i \le \lfloor B_2/d \rceil \text{ et} \\ g_j &= V_j(X_1) \bmod N \text{ pour } 1 \le j < d/2 \text{ et } j \perp d, \end{aligned}$$

de telle sorte que tout nombre premier q dans $]B_1, B_2]$ s'écrit sous la forme $q = id - j$ ou $q = id + j$.

Alors, si $b_1^q \equiv 1 \pmod p$,

$$\begin{aligned} V_{id}(X_1) &\equiv V_{q \pm j}(X_1) \equiv b_1^{q \pm j} + 1/b_1^{q \pm j} \\ &\equiv b_1^{\pm j} + 1/b_1^{\pm j} \equiv b_1^j + 1/b_1^j \equiv V_j(X_1) \pmod p \end{aligned}$$

et donc

$$V_{id}(X_1) - V_j(X_1) \equiv 0 \pmod p. \tag{12}$$

Une fois les listes f_i, g_j calculées, on calcule le produit

$$A = \prod_{\substack{id \pm j = q \\ B_1 < q \le B_2}} (f_i - g_j) \bmod N. \tag{13}$$

S'il existe $p \mid N$ et un nombre premier q dans $]B_1, B_2]$ tel que $b_1^q \equiv 1 \pmod p$, le produit (13) comporte i, j tels que (12) est vérifiée, et donc $p \mid \gcd(A, N)$.

Pour ECM, on peut utiliser $f_i = (id \cdot P_1)_X$, la coordonnée X de $id \cdot P_1$, et $g_j = (j \cdot P_1)_X$, voir [18]. Cependant, comme on utilise des coordonnées projectives, deux points identiques ou opposés n'ont pas forcément la même coordonnée X, mais ont un même rapport X/Z. Pour garantir que $id \cdot P = j \cdot P$ implique $(id \cdot P)_X = (j \cdot P)_X$, il faut normaliser les coordonnées Z des points calculés.

Certaines paires (i, j) incluent plus d'un nombre premier, en particulier quand à la fois $id - j$ et $id + j$ sont divisibles par un nombre premier q dans $]B_1, B_2]$. Notre implantation choisit les paires (i, j) soigneusement, de telle sorte que tous les nombres premiers dans $]B_1, B_2]$ sont inclus avec un nombre quasi-minimal de paires.

Les tables 2 et 3 indiquent le temps — déterminé de façon empirique — pour trouver des nombres premiers proches de $2^{25}, 2^{26}, \ldots, 2^{32}$ dans des nombres composés de 1 et 2 mots, et avec des paramètres B_1 et B_2 choisis pour minimiser le temps moyen. Le fait que B_1 et B_2 ne croissent pas avec la taille des facteurs est dû au fait que le temps moyen pour trouver un facteur premier en fonction de B_1 et B_2 est une fonction très plate autour de son minimum, de telle sorte que de petites perturbations des temps mesurés influent notablement sur les paramètres mesurés comme optimaux.

4.5 Comparaison avec des implantations matérielles d'ECM

Plusieurs implantations matérielles d'ECM pour la co-factorisation dans NFS ont été décrites dans la littérature. Un article récent de de Meulenaer et al. [9] utilise une carte

n	B_1	B_2	proba.	1 mot	2 mots -2 bits
25	300	5000	0.249	46	103
26	310	6000	0.220	55	125
27	320	6000	0.186	67	151
28	400	6000	0.167	81	182
29	430	7000	0.149	100	224
30	530	11000	0.158	119	275
31	530	10000	0.128	144	330
32	540	10000	0.105	177	410

Table 2 – Temps moyen en microsecondes et probabilité de trouver des facteurs premiers proches de 2^n dans des nombres composés de 1 ou 2 mots-machine avec P–1 sur un AMD Phenom à 2 GHz. Les paramètres B_1 et B_2 ont été choisis de manière empirique pour minimiser le rapport temps/probabilité.

n	B_1	B_2	proba.	1 mot	2 mots -2 bits
25	130	7000	0.359	67	176
26	130	7000	0.297	81	213
27	150	11000	0.290	101	264
28	160	13000	0.256	124	324
29	180	12000	0.220	151	395
30	200	12000	0.188	190	496
31	260	14000	0.182	231	604
32	250	15000	0.147	283	744

Table 3 – Temps moyen en microsecondes et probabilité par courbe de trouver des facteurs premiers proches de 2^n dans des nombres composés de 1 ou 2 mots-machine avec ECM sur un AMD Phenom à 2 GHz. Les paramètres B_1 et B_2 ont été choisis de manière empirique pour minimiser le temps moyen.

haute-performance Xilinx Virtex4SX FPGA avec multiplieurs embarqués. Ces auteurs ont implanté uniquement la phase 1 d'ECM, et pour des entrées jusque 135 bits. Ils indiquent un débit de 13793 phases 1 d'ECM par seconde avec $B_1 = 910$ pour \$100, avec un prix unitaire de \$116. Cependant, le prix indiqué sur `www.nuhorizons.com` et `www.avnet.com` pour cette carte est plutôt de \$300. Nous basons notre comparaison sur ce dernier chiffre.

La table 4 compare le rapport performance/coût de notre implantation logicielle d'ECM avec celle de de Meulenaer et al. Nous estimons qu'ECM avec des nombres de 3 mots-machine (135 bits) prendrait environ deux fois plus de temps que pour 2 mots-machine. Pour cette comparaison nous utilisons les temps mesurés pour 126 bits (2 mots-machine) et des estimations pour 135 bits (3 mots-machine). Les temps pour notre implantation logicielle ont été obtenus sur un AMD Phenom X4 9350e avec 4 cœurs à 2.0 GHz, et ont été mis à l'échelle d'un Phenom II X4 955 avec 4 cœurs à 3.2 GHz. De Meulenaer et al. n'ont pas implanté la phase 2, mais ont prédit sa performance à raison de $16,900$ phases 2 par seconde et par carte. Nous utilisons cette estimation pour notre comparaison. Nos résultats montrent qu'une implantation logicielle d'ECM est compétitive en termes

Carte	XC4VSX25-10	Phenom 9350e	Phenom II X4 955
Fréquence	0.22 GHz	2.0 GHz	3.2 GHz
Nombre de cœurs	1	4	4
nombre de 126 bits (2 mots en implantation logicielle)			
Temps de la phase 1	$62.5\mu s$	$232.1\mu s$	$\approx 145\mu s$
Temps de la phase 2	$59.2\mu s$	$121.5\mu s$	$\approx 76\mu s$
Temps par essai	$121.7\mu s$	$353.6\mu s$	$\approx 221\mu s$
#essais/sec/carte	8217	11312	18100
Coût par carte	\$300		\$215
#essais/sec/\$100	2739		8418
nombre de 135 bits (3 mots en implantation logicielle)			
Temps de la phase 1	$62.5\mu s$	$\approx 464\mu s$	$\approx 290\mu s$
Temps de la phase 2	$59.2\mu s$	$\approx 243\mu s$	$\approx 152\mu s$
Temps par essai	121.7	$\approx 707\mu s$	$\approx 442\mu s$
#essais/sec/carte	8217	≈ 5658	≈ 9052
Coût par carte	\$300		\$215
#essais/sec/\$100	2739		4210

Table 4 – Comparaison d'ECM avec $B_1 = 910$, $B_2 = 57000$ pour des nombres de 126 et 135 bits, sur un FPGA Virtex4SX25-10 et sur des processeurs AMD 64 bits.

de coût par courbe par rapport aux circuits matériels pour ECM.

Bibliographie

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

[2] Kazumaro Aoki, Jens Franke, Thorsten Kleinjung, Arjen K. Lenstra, and Dag Arne Osvik. A Kilobit Special Number Field Sieve Factorization. In Kaoru Kurosawa, editor, *Advances in Cryptology – ASIACRYPT 2007*, number 4833 in Lecture Notes in Computer Science, pages 1–12. Springer-Verlag, 2008.

[3] Jörg Arndt. *Matters Computational*. November 2009. Draft, available at `http://www.jjj.de/fxt/`.

[4] David H. Bailey. FFTs in external or hierarchical memory. *J. Supercomputing*, 4 :23–35, 1990.

[5] Daniel J. Bernstein. Circuits for integer factorization : a proposal. Manuscript, 2001. `http://cr.yp.to/nfscircuit.html`.

[6] Richard P. Brent. Some Integer Factorization Algorithms using Elliptic Curves. *Australian Computer Science Communications*, 8 :149–163, 1986.

[7] Joe P. Buhler, Hendrik W. Lenstra, Jr., and Carl Pomerance. Factoring integers with the number field sieve. In Lenstra and Lenstra [15], pages 50–94.

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

[9] Giacomo de Meulenaer, François Gosset, Guerric Meurice de Dormale, and Jean-Jacques Quisquater. Elliptic Curve Factorization Method : Towards Better Exploitation of Reconfigurable Hardware. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM07)*, pages 197–207. IEEE Computer Society Press, 2007.

[10] Jens Franke, Thorsten Kleinjung, Christof Paar, Jan Pelzl, Christine Priplata, and Colin Stahlke. SHARK : A Realizable Special Hardware Sieving Device for Factoring 1024-Bit Integers. In *Cryptographic Hardware and Embedded Systems — CHES 2005*, volume 3659 of *Lecture Notes in Computer Science*, pages 119–130, 2005.

[11] W. Morven Gentleman and Gordon Sande. Fast Fourier transforms–for fun and profit. In *AFIPS Conference Proceedings*, volume 29, pages 563–578, Washington, 1966. Spartan Books.

[12] Torbjörn Granlund. GNU MP : The GNU Multiple Precision Arithmetic Library, 2009. `http://gmplib.org/`.

[13] Anatolii Karatsuba and Yuri P. Ofman. Multiplication of multidigit numbers on automata. *Doklady Akademii Nauk SSSR*, 145(2) :293–294, 1962.

[14] Thorsten Kleinjung, Kazumaro Aoki, Arjen K. Lenstra, Emmanuel Thomé, Joppe W. Bos, Pierrick Gaudry, Alexander Kruppa, Peter L. Montgomery, Dag Arne Osvik, Herman te Riele, Andrey Timofeev, and Paul Zimmermann. Factorization of a 768-bit RSA modulus. `http://eprint.iacr.org/2010/006`.

[15] Arjen K. Lenstra and Hendrik W. Lenstra, Jr., editors. *The development of the number field sieve*, volume 1554 of *Lecture Notes in Mathematics*. Springer-Verlag, Berlin, 1993.

[16] Arjen K. Lenstra, Hendrik W. Lenstra, Jr., Mark S. Ma nasse, and John M. Pollard. The Factorization of the Ninth Fermat Number. *Mathematics of Computation*, 61(203) :319–349, 1993.

[17] Hendrik W. Lenstra, Jr. Factoring Integers with Elliptic Curves. *Annals of Mathematics*, 126(3) :649–673, 1987.

[18] Peter L. Montgomery. Speeding the Pollard and Elliptic Curve Methods of Factorization. *Mathematics of Computation*, 48 :243–264, 1987.

[19] Peter L. Montgomery. Evaluating Recurrences of Form $X_{m+n} = f(X_m, X_n, X_{m-n})$ Via Lucas Chains. Unpublished Manuscript, 1992. `ftp://ftp.cwi.nl/pub/pmontgom/Lucas.ps.gz`.

[20] Peter L. Montgomery and Robert D. Silverman. An FFT Extension to the $p-1$ Factoring Algorithm. *Mathematics of Computation*, 54 :839–854, 1990.

[21] John M. Pollard. Theorems on factorisation and primality testing. *Proceedings of the Cambridge Philosophical Society*, 5 :521–528, 1974.

[22] John M. Pollard. Factoring with cubic integers. In Lenstra and Lenstra [15], pages 4–10.

[23] Ronald L. Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public key encryption. *Communications of the ACM*, 21(2) :120–126, 1978.

[24] Arnold Schönhage. Asymptotically fast algorithms for the numerical multiplication and division of polynomials with complex coeficients. In *EUROCAM '82 : Proceedings of the European Computer Algebra Conference on Computer Algebra*, pages 3–15, London, UK, 1982. Springer-Verlag.

[25] Arnold Schönhage and Volker Strassen. Schnelle Multiplikation großer Zahlen. *Computing*, 7(3–4) :281–292, 1971.

[26] Allan Steel. Magma V2.12-1 is up to 2.3 times faster than GMP 4.1.4 for large integer multiplication, 2005. `http://magma.maths.usyd.edu.au/users/allan/intmult.html`.

[27] Andrei L. Toom. The Complexity of a Scheme of Functional Elements Realizing the Multiplication of Integers. *Doklady Akademii Nauk SSSR*, 150(3) :496–498, 1963.

[28] Hugh C. Williams. A $p+1$ Method of Factoring. *Mathematics of Computation*, 39 :225–234, 1982.

[29] Paul Zimmermann and Bruce Dodson. 20 Years of ECM. In Florian Hess, Sebastian Pauli, and Michael Pohst, editors, *Proceedings of the 7th Algorithmic Number Theory Symposium (ANTS VII)*, volume 4076 of *Lecture Notes in Computer Science*, pages 525–542. Springer-Verlag, 2006.

Résumé

Cette thèse propose des améliorations aux problèmes de la multiplication et de la factorisation d'entier.

L'algorithme de Schönhage-Strassen pour la multiplication d'entier, publié en 1971, fut le premier à atteindre une complexité de $O(n \log(n) \log(\log(n)))$ pour multiplier deux entiers de n bits, et reste parmi les plus rapides en pratique. Il réduit la multiplication d'entier à celle de polynôme sur un anneau fini, en utilisant la transformée de Fourier rapide pour calculer le produit de convolution. Dans un travail commun avec Gaudry et Zimmermann, nous décrivons une implantation efficace de cet algorithme, basée sur la bibliothèque GNU MP ; par rapport aux travaux antérieurs, nous améliorons l'utilisation de la mémoire cache, la sélection des paramètres et la longueur de convolution, ce qui donne un gain d'un facteur 2 environ.

Les algorithmes P–1 et P+1 trouvent un facteur p d'un entier composé rapidement si $p-1$, respectivement $p+1$, ne contient pas de grand facteur premier. Ces algorithmes comportent deux phases : la première phase calcule une grande puissance g_1 d'un élément g_0 d'un groupe fini défini sur $\mathbb{F}_p$, respectivement $\mathbb{F}_{p^2}$, la seconde phase cherche une collision entre puissances de g_1, qui est trouvée de manière efficace par évaluation-interpolation de polynômes. Dans un travail avec Peter Lawrence Montgomery, nous proposons une amélioration de la seconde phase de ces algorithmes, avec une construction plus rapide des polynômes requis, et une consommation mémoire optimale, ce qui permet d'augmenter la limite pratique pour le plus grand facteur premier de $p-1$, resp. $p+1$, d'un facteur 100 environ par rapport aux implantations antérieures.

Le crible algébrique (NFS) est le meilleur algorithme connu pour factoriser des entiers dont les facteurs n'ont aucune propriété permettant de les trouver rapidement. En particulier, le module du système RSA de chiffrement est choisi de telle sorte, et sa factorisation casse le système. De nombreux efforts ont ainsi été consentis pour améliorer NFS, de façon à établir précisément la sécurité de RSA. Nous donnons un bref aperçu de NFS et de son historique. Lors de la phase de crible de NFS, de nombreux petits entiers doivent être factorisés. Nous présentons en détail une implantation de P–1, P+1, et de la méthode ECM basée sur les courbes elliptiques, qui est optimisée pour de tels petits entiers. Finalement, nous montrons comment les paramètres de ces algorithmes peuvent être choisis finement, en tenant compte de la distribution des facteurs premiers dans les entiers produits par NFS, et de la probabilité de trouver des facteurs premiers d'une taille donnée.

Mots-clés: Arithmétique, multiplication des entiers, factorisation des entiers, courbes elliptiques, crible algébrique