UNIVERSITY OF CALIFORNIA

Los Angeles

An FFT Extension of the Elliptic Curve Method of Factorization

A dissertation submitted in partial satisfaction of the

requirements for the degree Doctor of Philosophy

in Mathematics

by

Peter Lawrence Montgomery

1992

The dissertation of Peter Lawrence Montgomery is approved.

Bryan C. Ellickson

Miloš D. Ercegovac

Basil Gordon

Murray M. Schacher

David G. Cantor, Committee Chair

University of California, Los Angeles

1992

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

This work was sponsored in part by U.S. Army fellowship DAAL03–89–G–0063 (1989–1992). Thanks to Unisys and UCLA for letting me use their facilities during this work. Thanks to Professors David Cantor, Basil Gordon, and Samuel S. Wagstaff, Jr. for reviewing early drafts hereof. Thanks to Professor Alfred W. Hales for being an alternate on my committee, even though his services were not needed.

VITA

September 25, 1947	Born, San Francisco, California
December, 1967	Ranked among top five participants in William Lowell Putnam Mathematical Competition.
June, 1969	A.B. with honors, Mathematics, University of California, Berkeley
September, 1971	M.A., Mathematics, University of California, Berkeley
1972–1982	Scientific and system programming at System Development Corporation (SDC), Huntsville, Alabama
1982–1989	Transferred to SDC (later renamed Unisys) in Santa Monica, California. Did simulations and formal verification.
1987–1992	Graduate student in Ph.D. program, Department of Mathematics, University of California, Los Angeles Teaching Associate, Summer, 1991 Coached UCLA Putnam team 1989–1991
1992	Joined Department of Mathematics, Oregon State University, Corvallis, Oregon 97331

PUBLICATIONS

John Brillhart, Peter L. Montgomery & Robert D. Silverman (January 1988). Tables of Fibonacci and Lucas factorizations. *Mathematics of Computation* 50 (181):251–260 & S1–S15.

P. Erdös, R. L. Graham, P. Montgomery, B. L. Rothschild, J. Spencer & E. G. Straus (May 1973). Euclidean Ramsey Theorems, I. *Journal of Combinatorial Theory, Series A* 14 (3):341–363.

— (1975). *Euclidean Ramsey Theorems, II.* In Colloquia Mathematica Societatis János Bolyai, 10, Infinite and Finite Sets, vol. I, Edited by A. Hajnal, R. Rado, Vera T. Sós, North-Holland, Amsterdam-London, pp. 529–557.

— (1975). *Euclidean Ramsey Theorems, III.* In Colloquia Mathematica Societatis János Bolyai, 10, Infinite and Finite Sets, vol. I, Edited by A. Hajnal, R. Rado, Vera T. Sós, North-Holland, Amsterdam-London, pp. 559–583.

Ronald Evans & Peter Montgomery (May 1990). Problem 6631. *American Mathematical Monthly,* 97 (5):432–433.

Peter L. Montgomery (December 1977). Problem E2686. *American Mathematical Monthly,* 84 (10):820.

— (December 1978). Evaluation of boolean expressions on one's complement machines. *SIGPLAN Notices,* 13 (12):60–72.

— (September 1979). Letter to the Editor. *SIGPLAN Notices,* 14 (9):7.

— (November 1984). Proposal 1202. *Mathematics Magazine,* 57 (5):298.

— (April 1985). Modular multiplication without trial division. *Mathematics of Computation,* 44 (170):519–521.

— (January 1987). Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of Computation,* 48 (177):243–264.

— (August 1989). Design of an FFT continuation to the ECM method of factorization. Abstract 850–11–25. *AMS Abstracts,* 10 (4):278.

— (to appear July 1993). New solutions of $a^{p-1} \equiv 1 \pmod{p^2}$. *Mathematics of Computation.*

Peter L. Montgomery & Robert D. Silverman (April 1990). An FFT extension to the $P - 1$ factoring algorithm. *Mathematics of Computation,* 54 (190):839–854.

ABSTRACT OF THE DISSERTATION

An FFT Extension of the Elliptic Curve Method of Factorization

by

Peter Lawrence Montgomery
Doctor of Philosophy in Mathematics
University of California, Los Angeles, 1992
Professor David G. Cantor, Chair

Factorization of arbitrary integers is believed to be a hard problem. The Elliptic Curve Method (ECM), discovered by Hendrik Lenstra, Jr. in 1985, is the best known method for finding 20– to 30– digit factors of a large integer N. The ECM algorithm has two main steps. It computes a large multiple of an element P of an elliptic curve group modulo N, obtaining another element Q. Step 1 succeeds if we strike the identity element of the group modulo p for some prime divisor p of N. If Step 1 is unsuccessful, then Step 2 compares multiples of Q, looking for a duplicate modulo some $p|N$. This thesis describes how to apply convolutions modulo N and fast polynomial arithmetic algorithms in the search. This effectively increases the range of ECM by a factor of 100 with about twice the combined Step 1/Step 2 execution time previously required (but more memory). The revised algorithm was tested by trying it on the RSA Factoring Challenge list.

We discuss many architectural considerations relating to the implementation, such as the identifying which portions of the computation can be vectorized or parallelized. We also discuss the algorithms for computer arithmetic.

We give a detailed analysis of intermediate results of the fast polynomial GCD algorithm.

We give a family of elliptic curves with torsion group of order 16 and positive rank over $\mathbb{Q}$, and compare the smoothness of their orders to the smoothness of curves with torsion group of order 12.

CHAPTER 1

Introduction

An integer $N > 1$ is called *composite* if is a product of two smaller positive integers; otherwise it is called *prime*. The Fundamental Theorem of Arithmetic [10, p. 5] states that any positive integer has a unique factorization into primes (except for order). But its proof gives no clue about how to find these prime factors. This problem has interested mathematicians for centuries. In his *Disquitiones Arithmeticae*, for example, Gauss [17, p. 398] wrote

> "The problem of distinguishing prime numbers from composites, and of resolving composite numbers into their prime factors, is one of the most useful in all of arithmetic. The dignity of science seems to demand that every aid to the solution of such an elegant and celebrated problem be zealously cultivated."

As Gauss observes, there are two problems here: checking whether a number is prime and factoring a number into primes if it is not itself prime. Fermat's Theorem states that if p is prime and a is an integer not divisible by p, then $a^{p-1} \equiv 1 \pmod{p}$. An integer $p > 1$ which fails this test for some a cannot be prime and must therefore be composite. M. O. Rabin used a variation of this observation [17, p. 379] to give a probabilistic primality test whose running time is polynomial in $\log p$. Rabin's algorithm is termed *probabilistic* because it may err, wrongly proclaiming that a number is prime when that number is really composite (but never the reverse error); the failure probability can be made arbitrarily small through repeated execution of the algorithm with different random a. There has been considerable recent progress in algorithms which rigorously prove that a number p which passes Rabin's test is truly prime. Although none of the latter algorithms run in polynomial time, the state of the art is only slightly worse. For example Morain [33, p. 65] used elliptic curves to prove that

$$\lceil 10^{1137}\gamma \rceil = 2 \cdot 47 \cdot 4231 \cdot 7789 \cdot p_{1128},$$

where γ is Euler's constant and p_{1128} is an 1128–digit prime. In April, 1992 Morain proved that the 1505–digit partition number $p(1840296)$ is prime, using four machine years of time on a network of SUN 3/60's.

The second problem, that of splitting an arbitrary composite integer N into its prime factors, is believed to be much harder. The problem was once primarily of academic interest, but has gained attention since the 1978 publication [35] of a public-key cryptosystem whose strength depends on the difficulty of factorization. Three major factorization algorithms were introduced between 1980 and 1990: Quadratic Sieve, Elliptic Curve Method, Number Field Sieve.

The Quadratic Sieve algorithm [34] [39] and the Number Field Sieve algorithm [21][13] take time dependent on the size of the integer N being factored. Each algorithm finds many congruences $x_i \equiv y_i \pmod{N}$, where x_i and y_i are either squares or products of elements from a *factor base*. After enough such congruences have been found, selected congruences are multiplied together to get another congruence $X \equiv Y \pmod{N}$ in which all elements of the factor base occur to even exponents both in X and in Y; this selection is done using linear algebra modulo 2. Then $\sqrt{X}$ and $\sqrt{Y}$ are integers, which can be computed modulo N. Any prime divisor of N must divide either $\gcd(\sqrt{X}+\sqrt{Y},\ N)$ or $\gcd(\sqrt{X}-\sqrt{Y},\ N)$; if either GCD is nontrivial, then one has a factor of N. The two methods differ in how the congruences $x_i \equiv y_i$ are found. The Quadratic Sieve algorithm has been used for N as large as 116 decimal digits, using a worldwide network of computers [23]. That network also factored the 155–digit Fermat number $2^{512}+1$ using Number Field Sieve [22].

The time for Elliptic Curve Method (ECM) depends primarily on the size of the prime factor p of N. The ECM algorithm requires considerable arithmetic modulo N; if p is a prime divisor of N, then ECM is really doing arithmetic modulo p, since there is a natural ring homomorphism from $\mathbb{Z}/N\mathbb{Z}$ to $\mathbb{Z}/p\mathbb{Z} = \mathrm{GF}(p)$. Over the field $\mathrm{GF}(p)$, the algorithm operates in an abelian group whose order is $p + O(p^{1/2})$. If this order is sufficiently smooth (i.e. if the group order has no large prime divisors), then ECM usually finds p. When the algorithm is unsuccessful, it can be repeated using a (presumably) different group. The group order seems more likely to be smooth when the prime p (and hence the order) is small, in which case fewer trials are needed to find p (but see Section 6.3). The time for ECM also grows with the time to do arithmetic modulo N, but this growth is less severe.

The Quadratic Sieve algorithm and improved computer hardware were major ingredients in causing the smallest composite cofactor in Appendix C of [11] (a book listing known factors of $b^n \pm 1$ for selected b and n) to leap from 51 digits in its 1983 edition to 80 digits in its 1988 edition. Between those years, the ECM algorithm found a thousand previously unknown prime factors for these tables. Even when ECM found only some factors of a number, the cofactor was sometimes sufficiently small for Quadratic Sieve to complete the work quickly.

As of April, 1992, seven calendar years and hundreds of machine years after ECM's introduction, three 38–digit factors, one 40–digit factor, and one 42–digit factor had been found; these appear in Table 1.0.1. Here F_n denotes the n–th

Fibonacci number. Lenstra found his 40–digit factor of an 89–digit cofactor of the partition number $p(11279)$, using ECM on a MasPar with 2^{14} processors and allocating 11 processors per curve. Rusin found the 42–digit factor of the Cunningham number $10^{134}+10^{67}+1$ on a SPARC using the program in [29] with limits $B_1 = 2 \cdot 10^6$ and $B_2 = 10^8$.

Factor	Of	Discoverer(s)
648 38817 74757 80953 23592 82791 43858 46481	$11^{118}+1$	Arjen K. Lenstra, Mark S. Manasse
733 30281 86548 76640 68079 92440 06620 01093	F_{467}	Robert D. Silverman
792 94907 48252 58311 72666 22855 30467 08561	F_{667}	Peter L. Montgomery
12320 79689 56766 26861 48201 86399 55442 47703	$p(11279)$	Arjen K. Lenstra
18 49764 79633 09293 11033 13037 83550 43553 63361	$10^{201}-1$	Dave Rusin

Table 1.0.1: Largest ECM factors found

For a history of factorization and primality testing algorithms, see [11] and [10].

1.1 Summary of main results

This work implements the algorithm summarized in [30], extending the work of [9]. With optimal parameters, the method described in this thesis beats its predecessor [29] when searching for prime factors over 25 digits. The major factors found during this work appear in Tables 9.2.1 and 9.3.1. The largest factor found was a 33–digit factor of the partition number $p(13421)$. A 31–digit factor and a 32–digit factor were found during Step 1, but no other factor over 29 digits was found during six months of runs (primarily on a DEC 5000).

1.2 Acronyms and notations

Table 1.2.1 lists acronyms used in this report; Tables 1.2.2, 1.2.3, and 1.2.4 list some notations used. Non-standard acronyms and notations are defined where first used.

The notation $a \equiv b \pmod{c}$ means that $a-b$ is divisible by c. The notation $b \bmod c$ (without parentheses) denotes the nonnegative remainder upon dividing the integer b by the positive integer c; it satisfies

$$b \bmod c = b - c\lfloor b/c \rfloor \qquad \text{and} \qquad 0 \le b \bmod c \le c-1.$$

For polynomials, $B(X) \bmod C(X)$ denotes the remainder upon dividing $B(X)$ by $C(X) \neq 0$; it satisfies

$$B(X) \bmod C(X) = B(X) - C(X) \left\lfloor \frac{B(X)}{C(X)} \right\rfloor,$$

$$\deg\Big(B(X) \bmod C(X)\Big) < \deg\Big(C(X)\Big).$$

AIX	Operating system on IBM RS/6000
AMS	**A**merican **M**athematical **S**ociety
DEC	**D**igital **E**quipment **C**orporation (computer manufacturer)
DP	**D**esired **p**roperty (Chapter 5)
ECM	**E**lliptic **C**urve **M**ethod
ECM/FFT	**E**lliptic **C**urve **M**ethod with **F**ast **F**ourier **T**ransform extension
FFT	**F**ast **F**ourier **T**ransform
GCD	**G**reatest **c**ommon **d**ivisor
HALFGCD	Polynomial **half-GCD** algorithm (Figure 3.8.1)
HG	**H**ALF**G**CD input or output property (Section 3.8)
IBM	**I**nternational **B**usiness **M**achines (computer manufacturer)
LIFO	**L**ast **i**n, **f**irst **o**ut
MIMD	**M**ultiple **i**nstruction, **m**ultiple **d**ata (parallel architecture)
MIPS	Computer manufacturer
MODMULN	**Mod**ular **mul**tiplication algorithm (Figure 8.1.1)
N.A.	**N**ot **a**pplicable
RECIP	Polynomial **recip**rocal (Section 3.5)
POLYEVAL	**Poly**nomial **eval**uation algorithm (Section 3.7)
POLYGCD	**Poly**nomial **GCD** algorithm (Figure 3.8.2)
RSA	**R**ivest, **S**hamir, and **A**dleman (public-key cryptosystem [35])
SIMD	**S**ingle **i**nstruction, **m**ultiple **d**ata (parallel architecture)
UCLA	**U**niversity of **C**alifornia, **L**os **A**ngeles

Table 1.2.1: Acronyms

A	Coefficient of X^2Z^2 in homogeneous form (2.3.3) of elliptic curve
B	Coefficient of Y^2Z^2 in homogeneous form (2.3.3) of elliptic curve
B_1	Upper bound for prime powers during Step 1
$\mathbb{C}$	Field of complex numbers
$\mathrm{Cost}(\dots)$	Cost per elliptic curve: see (7.3.1)
d_1, d_2	Degree of $F(X)$ (resp. $G(X)$) in Section 4.3
$d(n)$	Number of divisors of n, including 1 and n
$E_{(p)}$	Reduction of elliptic curve E modulo a prime p
$\mathrm{EOR}(x, y)$	Bitwise exclusive OR (two or more arguments)
$F(X)$	Polynomial modulo which computations are done in Section 4.3
F_n	nth Fibonacci number: $F_0 = 0$, $F_1 = 1$, $F_{n+2} = F_{n+1} + F_n$ if $n \geq 0$
$G(X)$	Latest polynomial remainder in Section 4.3
$g_{k,\alpha}(X)$	Dickson polynomial (Section 5.1)
$\gcd(m, n)$	Greatest common divisor of m and n
$\mathrm{GF}(p^k)$	Field of p elements (p a prime, $k > 0$, usually $k = 1$)
$\mathrm{GF}(p^k)^*$	Nonzero elements of $\mathrm{GF}(p^k)$
$H(X)$	Polynomial multiplied by $G(X)$ modulo $F(X)$ in Section 4.3
I_n	$n \times n$ identity matrix
$L(x)$	$\exp(\sqrt{\ln x \ \ln\ln x}\,)$ (defined for $x > 1$)
$M(n)$	Time to multiply two polynomials each of degree at most $n-1$ in a (fixed) ring R (Chapter 3)
m_i	Multiples of Q used to compute x-coordinates for F in Section 4.3

Table 1.2.2: Notations, part I

N	The integer being factored, or modulo which arithmetic is done
n_j	Multiples of Q used to compute x-coordinates for H in Section 4.3
$0_{m,n}$	$m \times n$ zero matrix
O	Point at infinity on elliptic curve (group identity element)
$O(f(n))$	Any function $g(n)$ such that $g(n)/f(n)$ is bounded as $n \to +\infty$
$o(f(n))$	Any function $g(n)$ such that $\lim_{n\to\infty} g(n)/f(n) = 0$
$P(X)$	Polynomial used to select multiples of Q in Chapter 5
$\Pr_{\mathrm{succ}}(\dots)$	Success probability per curve: see (7.2.2)
$p(n)$	Number of unordered partitions of an integer n
$\mathbb{Q}$	Field of rational numbers
Q	Point on elliptic curve group as output from Step 1
$(m \cdot Q)_x$	x-coordinate of $m \cdot Q$, in elliptic curve group
$Q_{(p)}$	Reduction of Q modulo a prime p
q	Order of Q modulo some prime p
$q_{\max}$	Upper bound for q in Chapter 4
R	Commutative ring (within Chapter 3)
$\mathrm{Res}(F, G)$	Resultant of two polynomials; see (3.10.2)
$T_{m,n,k}(F)$	$m \times n$ Toeplitz matrix (3.10.1) from coefficients of polynomial F
t_{add}	Time for addition, subtraction, or negation in ring R
t_{inv}	Time for finding a multiplicative inverse (when it exists) in ring R
t_{mul}	Time for multiplication in ring R
$\mathbb{Z}$	Ring of integers
$\mathbb{Z}/N\mathbb{Z}$	Ring of integers modulo N

Table 1.2.3: Notations, part II

$\sqrt[8]{16}$	An 8–th root of 16 modulo q
$\mu(n)$	Möbius function: $\begin{cases} 0 \text{ if } n \text{ has repeated factors,} \\ (-1)^d \text{ if } n \text{ has } d \text{ distinct prime factors.} \end{cases}$
$\phi(n)$	Euler's totient function
$\Phi_n(X, Y)$	n–th cyclotomic polynomial in two variables: $\prod_{d\mid n}(X^d - Y^d)^{\mu(n/d)}$
$\rho(\alpha)$	Dickman function: see Section 7.1
$\zeta(s)$	Zeta function: $\sum_{n=1}^{\infty} n^{-s}$
$\lfloor x \rfloor$	Largest integer n such that $n \le x$ (x real)
$\lceil x \rceil$	Least integer n such that $n \ge x$ (x real)
$\left\lfloor \dfrac{F(X)}{G(X)} \right\rfloor$	Polynomial quotient (remainder discarded)
	(F, G polynomials in X)
$x \ll y$,	
$y \gg x$	Both mean x is much less than y
$x \sim y$,	
$x \approx y$	Both mean x is approximately y; $\sim$ is stronger than $\approx$
$x\mid y$	x divides y (x, y integers)
$x\nmid y$	x does not divide y (x, y integers)

Table 1.2.4: Notations, part III

CHAPTER 2

Elliptic Curve Method and its History

An *elliptic curve* E over a field K is defined by a cubic equation

$$Y^2 + a_1XY + a_0Y = b_3X^3 + b_2X^2 + b_1X + b_0 \tag{2.0.1}$$

with coefficients in K, where $b_3 \neq 0$ and where the discriminant does not vanish (i.e. the right side of (2.0.1) must not have repeated roots after completing the square in Y on the left). We assume henceforth that $\mathrm{char}(K) \neq 2, 3$. Then the linear change of variable

$$y = b_3\left(Y + a_1X/2 + a_0/2\right), \qquad x = b_3X + b_2/3 + a_1^2/12,$$

converts (2.0.1) to the *Weierstrass form*

$$y^2 = x^3 + ax + b, \tag{2.0.2}$$

where $a, b \in K$ and $4a^3+27b^2 \neq 0$. The curve consists of all points (x, y) satisfying (2.0.1), together with a *point at infinity*, denoted by O.

The points on an elliptic curve form an abelian group with identity element O if we define the group law suitably [38, pp. 55ff.]. Suppose E is given by (2.0.2). The negative of O is $-O = O$; the negative of any other point $P_1 = (x_1, y_1)$ on E is its reflection with respect to the x-axis: $-P_1 = (x_1, -y_1)$. The sum $P_1 + P_2$ of two points $P_1, P_2 \in E$ is defined by [38, pp. 58–59]:

(i) If $P_1 = O$, then $P_1 + P_2 = P_2$.

(ii) If $P_2 = O$, then $P_1 + P_2 = P_1$.

(iii) If $P_1 = -P_2$, then $P_1 + P_2 = O$.

(iv) If none of (i), (ii), (iii) holds, then $P_1 + P_2 = (x_3, y_3)$ where

$$\lambda = \begin{cases} \dfrac{y_2 - y_1}{x_2 - x_1}, & \text{if } x_1 \neq x_2, \\ \dfrac{3x_1^2 + a}{2y_1}, & \text{if } x_1 = x_2, \end{cases} \tag{2.0.3}$$

$$x_3 = \lambda^2 - x_1 - x_2,$$

$$y_3 = -(\lambda(x_3 - x_1) + y_1).$$

The parameter λ in (2.0.3) is chosen so that the line $y = \lambda(x - x_1) + y_1$ passes through P_1 and P_2 if $P_1 \neq P_2$, and is tangent to E at P_1 if $P_1 = P_2$. This line intersects the curve at a third point $(x_3, -y_3) = -P_3$ (which may equal P_1 or P_2). The group law is defined so that if a straight line passes through three points of E, then those three points sum to O. Figure 2.0.1 illustrates the group law, by computing $2P + Q$ in two ways for $P = (0, 1)$ and $Q = (-1, 2)$ on the curve $y^2 = x^3 - 4x + 1$.

When $K = \mathrm{GF}(p)$ is the field of p elements, the group E has finite order. Hasse [38, p. 131] proved that the order of this group satisfies

$$\Big| p + 1 - |E| \Big| \leq 2\sqrt{p}.$$

Hence the group order is numerically very close to p. The actual group order varies with the choice of curve.

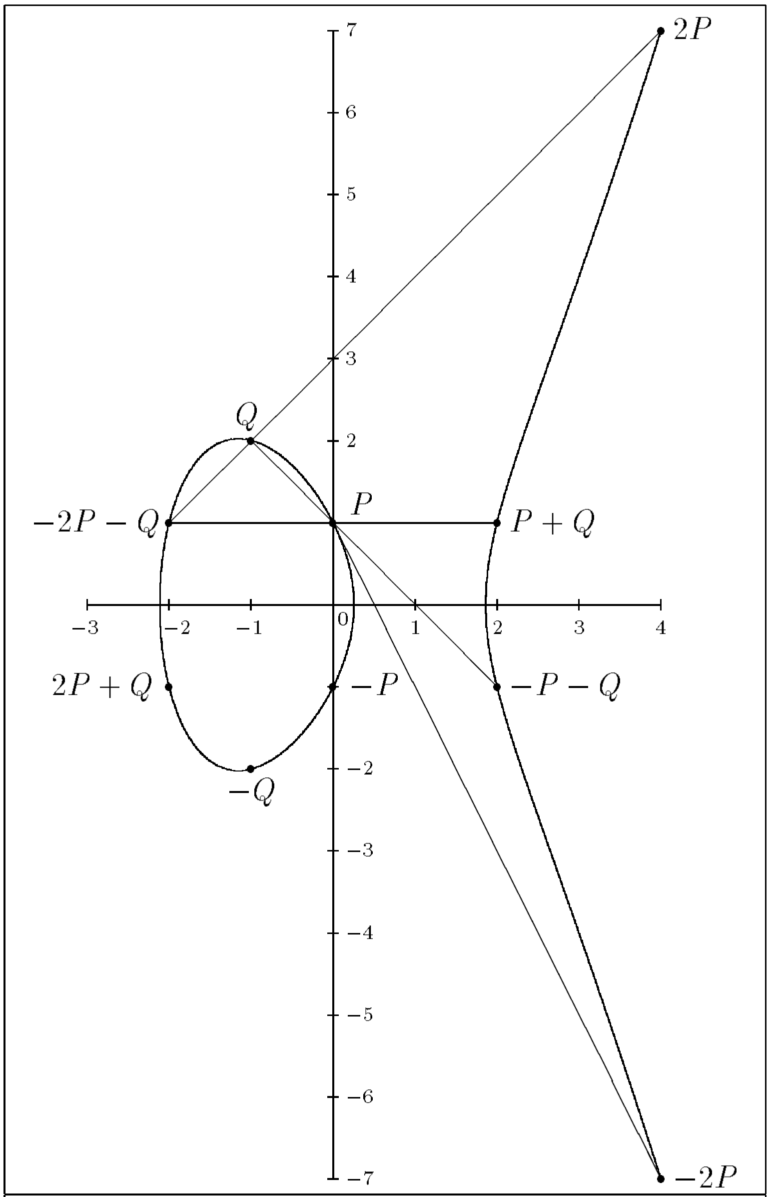
2.1 Lenstra's original algorithm

Lenstra announced the ECM algorithm in February, 1985 and published it in 1987 [24]. It is modeled after Pollard's $P-1$ algorithm [10, pp. 67ff.]. With proper choice of parameters, the expected amount of arithmetic modulo N required to find a prime factor p of N by ECM is $L(p)^{2+o(1)}$ [24, p. 651] where

$$L(p) = \exp\left(\sqrt{\ln p \ \ln \ln p}\right).$$

When $p \approx N^{1/2}$, this complexity matches that of the Quadratic Sieve algorithm mentioned in Chapter 1, but with higher constants. For $p = O(N^{1/2-\epsilon})$ where $\epsilon > 0$, the ECM algorithm is asymptotically faster than Quadratic Sieve as $N \to \infty$.

Lenstra [24, pp. 663ff.] generalizes the definition of elliptic curve to work over a ring $\mathbb{Z}/N\mathbb{Z}$ with $\gcd(N, 6) = 1$ as well as over a field. He defines a pseudo-addition



$$
\begin{array}{rcccccl}
2P & = & (0,1) & + & (0,1) & = & (4,7) \\
P+Q & = & (0,1) & + & (-1,2) & = & (2,1) \\
2P+Q & = & (4,7) & + & (-1,2) & = & (-2,-1) \\
P+(P+Q) & = & (0,1) & + & (2,1) & = & (-2,-1)
\end{array}
$$

Figure 2.0.1: Group law on $y^2 = x^3 - 4x + 1$

on these curves by emulating the algebraic rules for adding two points over a field, except when (2.0.3) might encounter a nonzero, non-invertible denominator. Any such denominator must be a zero divisor modulo N, which is fortuitous rather than disastrous since the greatest common divisor algorithm will reveal a non-trivial factor of N. From this pseudo-addition Lenstra defines a multiplication which is algebraically equivalent to ordinary group multiplication unless a zero divisor of N is encountered.

Lenstra selects a random curve E over $\mathbb{Z}/N\mathbb{Z}$ with known initial point P. Compute $Q = R \cdot P$, where R is a positive integer divisible by all prime powers below some bound B_1. This computation can fail only if some denominator is a zero divisor, in which case a factor is found. For each prime factor p of N, let $E_{(p)}$ denote the reduction of E modulo p (see [16, Chapter 5] for definition). If any order $|E_{(p)}|$ divides R, then the reduction $Q_{(p)}$ of Q in $E_{(p)}$ is the identity, by Lagrange's Theorem about subgroup orders. When the reduction of Q is the identity in $E_{(p)}$ for some but not all primes p dividing N, then Q's denominator is divisible by some but not all prime factors of N, triggering the aforementioned "failure".

2.2 Step 2 — Brent and Montgomery improvements

Lenstra's algorithm multiplies an initial point P on a curve E by an integer R, obtaining $Q = R \cdot P$ (a process called "Step 1"). Step 1 fails if no zero divisor is encountered. It succeeds if the group order $|E_{(p)}|$ divides R for some prime $p|N$.

Suppose that $|E_{(p)}| \nmid R$ but $|E_{(p)}| \mid Rq$ for some prime q, with q not too large and $p|N$. Then

$$q \cdot Q = q \cdot (R \cdot P) = (qR) \cdot P = O$$

in $E_{(p)}$. So the reduction $Q_{(p)}$ has order dividing q, and must either be the identity or have order q.

Both Brent [9] and Montgomery [29] observed early that Lenstra's algorithm can be modified to take advantage of such occurrences. Each suggests computing several multiples $n_i \cdot Q$ for selected n_i. If some $n_i \equiv n_j \pmod{q}$, then the points $n_i \cdot Q$ and $n_j \cdot Q$ agree upon reduction modulo p; unless $n_i = n_j$, these are unlikely to match modulo other primes dividing N, so p can be found by comparing $n_i \cdot Q$ and $n_j \cdot Q$. The match can be done by testing the difference of their x-coordinates; then the match also succeeds if $q|(n_i + n_j)$. This entire process is nicknamed "Step 2".

Brent and Montgomery differ on how to select the n_i. Montgomery suggests comparing multiples of Q along an arithmetic progression to a few fixed multiples of Q. Brent suggests the use of semirandom multiples. This question is discussed in detail in Chapter 5.

2.3 Weierstrass and Montgomery parameterizations

When $P_1 \neq P_2$, (2.0.3) uses one inversion and two multiplications (one a squaring); it uses another squaring if $P_1 = P_2$. The inversion may be avoided by using homogeneous coordinates, in which each coordinate is represented as a quotient of two elements of K, and using rational arithmetic in (2.0.3). Straightforward attempts (whether using a common denominator or separate denominators for x and y) use about 10 multiplications in place of the inversion in (2.0.3).

Montgomery [29, pp. 260–261] suggested an alternate parameterization which allows the use of homogeneous coordinates (i.e. no inversions) while keeping the number of multiplications small. To motivate it, consider the affine curve

$$y^2 = b_3x^3 + b_2x^2 + b_1x + b_0 \qquad (b_3 \neq 0). \tag{2.3.1}$$

If $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ are two points on (2.3.1) with $x_1 \neq x_2$, with sum $P_+ = (x_+, y_+)$ and difference $P_- = (x_-, y_-)$, then

$$\begin{aligned}
x_+ &= \frac{\left(\dfrac{y_1 - y_2}{x_1 - x_2}\right)^2 - b_2}{b_3} - x_1 - x_2 \\
&= \frac{b_3x_1x_2(x_1 + x_2) + 2b_2x_1x_2 + b_1(x_1 + x_2) + 2b_0 - 2y_1y_2}{b_3(x_1 - x_2)^2}, \\
x_- &= \frac{\left(\dfrac{y_1 + y_2}{x_1 - x_2}\right)^2 - b_2}{b_3} - x_1 - x_2 \\
&= \frac{b_3x_1x_2(x_1 + x_2) + 2b_2x_1x_2 + b_1(x_1 + x_2) + 2b_0 + 2y_1y_2}{b_3(x_1 - x_2)^2}.
\end{aligned}$$

A straightforward calculation gives

$$x_+x_- = \frac{(b_3x_1x_2 - b_1)^2 - 4b_0(b_3x_1 + b_3x_2 + b_2)}{b_3^2(x_1 - x_2)^2}.$$

If we require $b_1 = b_3$ and $b_0 = 0$, then this simplifies to

$$x_+x_- = \frac{(x_1x_2 - 1)^2}{(x_1 - x_2)^2}. \tag{2.3.2}$$

This allows one to compute the x-coordinate x_+ of $P_1 + P_2$ from those of P_1, P_2, and $P_1 - P_2$ with a few multiplications, inversions and squarings.

By itself, (2.3.2) costs more that (2.0.3), but (2.3.2) imposes little overhead when switching to homogeneous coordinates. Upon setting $b_0 = 0$, $b_1 = b_3 = 1/B$, $b_2 = A/B$ in (2.3.1) and putting $y = Y/Z$, $x = X/Z$, we obtain

$$BY^2Z = X(X^2 + AXZ + Z^2). \tag{2.3.3}$$

This is an elliptic curve if and only if $B \neq 0$ and $A \neq \pm 2$.

Equation (2.3.3) is said to be in *homogeneous* (or *projective*) form, because it is invariant upon replacing (X, Y, Z) by (kX, kY, kZ) for any $k \neq 0$. Its coordinates are often written $(X : Y : Z)$ rather than (X, Y, Z). The group identity element is $O = (0 : 1 : 0)$. The negative of $P = (X : Y : Z)$ is $-P = (X : -Y : Z)$.

If $P_1 = (X_1 : Y_1 : Z_1)$ and $P_2 = (X_2 : Y_2 : Z_2)$ are two points on (2.3.3) with distinct X/Z ratios (hence not equal or negatives of each other) and with difference $P_1 - P_2 = (X_- : Y_- : Z_-)$, then their sum $P_1 + P_2 = (X_+ : Y_+ : Z_+)$ satisfies

$$\begin{aligned} \frac{X_+}{Z_+} &= \frac{Z_-(X_1X_2 - Z_1Z_2)^2}{X_-(X_1Z_2 - Z_1X_2)^2} \\ &= \frac{Z_-\Big((X_1 - Z_1)(X_2 + Z_2) + (X_1 + Z_1)(X_2 - Z_2)\Big)^2}{X_-\Big((X_1 - Z_1)(X_2 + Z_2) - (X_1 + Z_1)(X_2 - Z_2)\Big)^2}, \end{aligned} \tag{2.3.4}$$

by (2.3.2). Montgomery also gives a doubling rule: if $P_1 = (X_1 : Y_1 : Z_1)$, then $2P_1 = (X_2 : Y_2 : Z_2)$ is given by

$$\begin{aligned} \frac{X_2}{Z_2} &= \frac{(X_1^2 - Z_1^2)^2}{4X_1Z_1(X_1^2 + AX_1Z_1 + Z_1^2)} \\ &= \frac{(X_1 + Z_1)^2(X_1 - Z_1)^2}{(4X_1Z_1)\Big((X_1 - Z_1)^2 + ((A+2)/4)(4X_1Z_1)\Big)}, \end{aligned} \tag{2.3.5}$$

$$\text{where} \quad 4X_1Z_1 = (X_1 + Z_1)^2 - (X_1 - Z_1)^2.$$

He proposes computing only the $(X : Z)$ ratios. By (2.3.4) and (2.3.5), one can add two points on (2.3.3) with six multiplications if their difference is known, and one can double a point with five multiplications if $(A+2)/4$ is known. He computes large multiples of a point using the methods in [31].

Not every elliptic curve of the form (2.0.1) can be linearly transformed to form (2.3.3) unless the base field K is algebraically closed or has other nice properties. But ECM lets its user choose the curve, and there is no prohibition against selecting one of form (2.3.3). The curve selection strategy is discussed in detail in Chapter 6.

CHAPTER 3

Fast Polynomial Arithmetic

Our algorithms will require fast polynomial arithmetic over the ring $\mathbb{Z}/N\mathbb{Z}$, including multiplication, remaindering, and greatest common divisor. This chapter summarizes some of the fast algorithms found in the literature, emphasizing how they apply to this ring. It includes a detailed analysis of intermediate results of the fast GCD algorithm.

All rings in this chapter are assumed commutative. Polynomials are univariate unless otherwise indicated.

3.1 Minimal time for polynomial multiplication

Let $M(n)$ be the time required to multiply two polynomials $F(X)$ and $G(X)$ of degrees at most $n-1$ over a given ring R, i.e. to compute the coefficients of the product given the coefficients of the inputs. We assume that a ring addition or subtraction can be computed in time t_{add}, a ring multiplication in time t_{mul}, and a multiplicative inversion (when it exists) in time t_{inv}. These times are assumed constant for any given ring R, but may depend on R.

Since any multiplication algorithm must read all $2n$ input coefficients, we have the trivial lower bound $M(n) \geq O(n)$. The straightforward algorithm uses n^2 multiplications and $(n-1)^2$ additions in the ring, giving the upper bound

$$M(n) \leq n^2 t_{\mathrm{mul}} + (n-1)^2 t_{\mathrm{add}}.$$

Karatsuba [17, pp. 278–279] [2, pp. 62–64] observed that we can do better by divide and conquer. Suppose that $n = 2m$ is even. Write

$$F(X) = F_0(X) + F_1(X)X^m \quad \text{and} \quad G(X) = G_0(X) + G_1(X)X^m,$$

where $\deg(F_0)$, $\deg(F_1)$, $\deg(G_0)$, $\deg(G_1) < m$. Then

(3.1.1)

$$\begin{aligned} FG &= (F_0 + F_1X^m)(G_0 + G_1X^m) \\ &= F_0G_0 + (F_0G_1 + F_1G_0)X^m + F_1G_1X^{2m} \\ &= F_0G_0 + \Big((F_0 + F_1)(G_0 + G_1) - F_0G_0 - F_1G_1\Big)X^m + F_1G_1X^{2m}. \end{aligned}$$

The polynomials F_0, G_0, F_1, G_1, $F_0 + F_1$, and $G_0 + G_1$ all have degrees less than $m = n/2$. This technique gives

$$M(n) \le 3M(m) + (2m + 2(2m-1))t_{\mathrm{add}} = 3M(n/2) + (3n-2)t_{\mathrm{add}}.$$

Together with $M(1) = t_{\mathrm{mul}}$, the solution of this recurrence is

$$M(2^n) \le 3^n t_{\mathrm{mul}} + (5 \cdot 3^n - 6 \cdot 2^n + 1)t_{\mathrm{add}} \qquad (n \ge 0),$$

so

$$M(n) = O\left(n^{\log_2 3}(t_{\mathrm{mul}} + 5t_{\mathrm{add}})\right) = O\left(n^{1.585}(t_{\mathrm{mul}} + 5t_{\mathrm{add}})\right). \tag{3.1.2}$$

If the original bound n on the degrees is not a power of 2, then we can increase the bound and still achieve (3.1.2). This method works over any ring.

We will assume in subsequent analyses that the $M(n)$ function satisfies

$$aM(n) \le M(an) \le a^2 M(n) \tag{3.1.3}$$

for $a \ge 1$ [2, p. 280].

3.2 Circular convolutions

We start by defining a circular convolution of two vectors (called *cyclic convolution* in [17, p. 491] and *positive wrapped convolution* in [2, p. 256]).

Definition 3.2.1 *Let* $\mathbf{f} = [f_0,\ f_1,\ f_2,\ \cdots,\ f_{n-1}]^T$ *and* $\mathbf{g} = [g_0,\ g_1,\ g_2,\ \cdots,\ g_{n-1}]^T$ *be two n-vectors over a ring R. Their* circular convolution *(of* length n*)* $\mathbf{f} \otimes \mathbf{g}$ *is defined to be the n-vector* $\mathbf{h} = [h_0,\ h_1,\ h_2,\ \cdots,\ h_{n-1}]$ *where*

$$h_k = \sum_{i+j \equiv k \pmod{n}} f_i g_j \quad (0 \le k \le n-1).$$

Circular convolutions of length n are really polynomial multiplications modulo $X^n - 1$. If $\mathbf{f}$, $\mathbf{g}$, and $\mathbf{h}$ are as in Definition 3.2.1, and if

$$F(X) = \sum_{i=0}^{n-1} f_i X^i, \qquad G(X) = \sum_{i=0}^{n-1} g_i X^i, \qquad H(X) = \sum_{i=0}^{n-1} h_i X^i,$$

then $\mathbf{h} = \mathbf{f} \otimes \mathbf{g}$ is equivalent to $H(X) \equiv F(X)G(X) \pmod{X^n - 1}$. For example, if $\mathbf{h} = \mathbf{f} \otimes \mathbf{g}$, then

$$\begin{aligned} H(X) = \sum_{k=0}^{n-1} h_k X^k &= \sum_{k=0}^{n-1} \sum_{i+j \equiv k \pmod{n}} f_i g_j X^k \\ &\equiv \sum_{k=0}^{n-1} \sum_{i+j \equiv k \pmod{n}} f_i g_j X^{i+j} = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} f_i g_j X^{i+j} \\ &= F(X)G(X) \pmod{X^n - 1}. \end{aligned}$$

If we can perform fast circular convolutions, then we have an algorithm for fast polynomial multiplication. Given two polynomials F and G, choose $n > \deg(F) + \deg(G)$. Use a circular convolution of length n (after padding with leading zeros) to form $F(X)G(X) \pmod{X^n - 1}$; this must equal $F(X)G(X)$ since $\deg(FG) < n$. We can also use $n = \deg(F) + \deg(G)$ if we compute the leading (or constant) coefficient of the product directly.

3.3 FFT for polynomial multiplication

The Fast Fourier Transform (FFT) algorithm [2, pp. 252ff.] is the basis of the fastest known polynomial multiplication algorithms. If the base ring R satisfies certain conditions, then the FFT algorithm executes a circular convolution of length n using $O(n \log n)$ operations (additions and multiplications) in the base ring. This is asymptotically better than the $O(n^{1.585})$ bound for $M(n)$ from Section 3.1.

Specifically, when the length n is a power of 2 and $n > 1$, the FFT algorithm requires (i) that 2 have a multiplicative inverse in R, and (ii) that there exist a (known or easily computable) element ω such that $\omega^{n/2} = -1$. When $R = \mathbb{C}$ is the field of complex numbers, these are satisfied for $\omega = e^{2\pi i/n}$. The conditions are also satisfied when $R = \mathrm{GF}(p)$ is the field of p elements if $p \equiv 1 \pmod{n}$ and ω is the $((p-1)/n)$–th power of a primitive element of the field.

Given two polynomials $F(X) = \sum_{i=0}^{n-1} f_i X^i$ and $G(X) = \sum_{i=0}^{n-1} g_i X^i$ of degree at most $n-1$, the FFT algorithm constructs the n-vectors

$$\hat{\mathbf{f}} = [F(1),\ F(\omega),\ F(\omega^2),\ \cdots,\ F(\omega^{n-1})]^T,$$

$$\hat{\mathbf{g}} = [G(1),\ G(\omega),\ G(\omega^2),\ \cdots,\ G(\omega^{n-1})]^T$$

from the coefficient vectors $\mathbf{f} = [f_0,\ f_1,\ \cdots,\ f_{n-1}]^T$ and $\mathbf{g} = [g_0,\ g_1,\ \cdots,\ g_{n-1}]^T$. These vectors $\hat{\mathbf{f}}$ and $\hat{\mathbf{g}}$ are the *discrete Fourier transforms* (DFT) of $\mathbf{f}$ and $\mathbf{g}$. A straightforward computation [2, p. 255] shows that the pointwise product

$$[F(1)G(1),\ F(\omega)G(\omega),\ F(\omega^2)G(\omega^2),\ \cdots,\ F(\omega^{n-1})G(\omega^{n-1})]^T \tag{3.3.1}$$

of $\hat{\mathbf{f}}$ and $\hat{\mathbf{g}}$ is the DFT of the circular convolution $\mathbf{h} = \mathbf{f} \otimes \mathbf{g}$. From $\hat{\mathbf{h}}$, an inverse discrete Fourier transform (i.e. an interpolation) produces $\mathbf{h}$ and hence the coefficients of $H(X) = F(X)G(X) \pmod{X^n - 1}$. The beauty of the FFT is that the evaluation and interpolation stages take only

$$(n \log_2 n - 2n + 2)t_{\mathrm{mul}} + (2n \log_2 n)t_{\mathrm{add}},$$

plus an extra n multiplications by n^{-1} during the interpolation. (This assumes that all roots of unity are known, and does not count trivial multiplies by ± 1.) Adding the time for (3.3.1) yields a bound of

$$(3n\log_2 n - 4n + 6)t_{\mathrm{mul}} + (6n\log_2 n)t_{\mathrm{add}}$$

for multiplication modulo $X^n - 1$. Replacing n by $2n$ gives

$$M(n) \le 2n\log_2(2n)(3t_{\mathrm{mul}} + 6t_{\mathrm{add}})$$

if the ring satisfies the requirements and n is a power of 2. See [2, pp. 257ff.] for details.

3.4 Circular convolutions over $\mathbb{Z}/N\mathbb{Z}$

We would like to apply the methods of Section 3.3 to the ring $R = \mathbb{Z}/N\mathbb{Z}$, where N is the integer we are attempting to factor. The FFT has two requirements: (i) 2 must have a multiplicative inverse in R, and (ii) there must be a known element ω with $\omega^{n/2} = -1$. Requirement (i) is satisfied, since our N will be odd. Requirement (ii) will usually not be satisfied.

Montgomery and Silverman [32, pp. 842–843] perform circular convolutions modulo N by executing several such convolutions modulo small primes and using the Chinese Remainder Theorem to get a result modulo N. The small primes are selected so that the requirements in Section 3.3 are satisfied.

Let n be a power of 2. Suppose that we want a polynomial product $F(X)G(X)$ $\pmod{N,\ X^n - 1}$ where $\deg(F) < n$ and $\deg(G) < n$, and where all coefficients of F and G are in the interval $[0,\ N-1]$. Select distinct primes $\{p_i\}_{i=1}^{K}$ such that each $p_i \equiv 1 \pmod{n}$ and

$$P = \prod_{i=1}^{K} p_i > nN^2/(1-\epsilon). \tag{3.4.1}$$

Here $\epsilon > 0$ depends on the precision of floating point arithmetic. Perform a single-precision circular convolution modulo each p_i, using the methods of Section 3.3, after reducing the coefficients of F and G modulo p_i. The next step is to construct

$$F(X)G(X) \equiv \sum_{j=0}^{n-1} h_j X^j \pmod{X^n - 1}$$

over $\mathbb{Z}$ from the known products

$$F(X)G(X) \equiv \sum_{j=0}^{n-1} h_{ij} X^j \pmod{p_i,\ X^n - 1}.$$

Since $h_j \equiv h_{ij} \pmod{p_i}$, the Chinese Remainder Theorem shows that

$$h_j \equiv \sum_{i=1}^{K} \frac{P}{p_i} y_{ij} \pmod{P},$$

where

$$y_{ij} \equiv (P/p_i)^{-1} h_{ij} \pmod{p_i}. \tag{3.4.2}$$

The bounds $0 \le h_j \le n(N-1)^2 < P$ lead to the explicit formula [32, equation (4.4)]

$$h_j = \sum_{i=1}^{K} \frac{P}{p_i} y_{ij} - P \left\lfloor \sum_{i=1}^{K} \frac{y_{ij}}{p_i} + \frac{\epsilon}{2} \right\rfloor . \tag{3.4.3}$$

We can get h_j mod N directly from (3.4.3). A more computationally convenient formula for h_j mod N is

$$h_j \equiv \sum_{i=1}^{K} \left(\frac{P}{p_i} \bmod N \right) y_{ij} + \Big((-P) \bmod N \Big) \left\lfloor \sum_{i=1}^{K} \frac{y_{ij}}{p_i} + \frac{\epsilon}{2} \right\rfloor \pmod{N}. \tag{3.4.4}$$

The coefficients $((P/p_i)^{-1} \bmod p_i)$ in (3.4.2) can be pre-computed and stored in tables. So can $(P/p_i \bmod N)$ and the 0–th through $(K+1)$–st multiples of $(-P) \bmod N$ in (3.4.4). With these tables, computation of an h_j from $\{h_{ij}\}_{i=1}^{K}$ requires at most

(i) K modular multiplications to get the y_{ij}, with $0 \le y_{ij} \le p_i - 1$;

(ii) $O(K)$ floating point operations to evaluate $\left\lfloor \sum_{i=1}^{K} \frac{y_{ij}}{p_i} + \frac{\epsilon}{2} \right\rfloor$;

(iii) K multiplications of y_{ij} by residues in $[0,\ N-1]$;

(iv) K additions of numbers in $[0,\ N-1]$;

(v) One reduction modulo N of a integer in $\Big[0,\ \Big(N-1\Big)\Big(K + \sum_{i=1}^{K}(p_i - 1)\Big)\Big]$.

The cost is $O(K)$ for (i) and (ii), $O(K \log N)$ for (iii) and (iv), and $O(\log N \log K)$ for (v). Since there are n different coefficients h_j, the cost of the Chinese Remainder Theorem is $O(nK \log N)$. Reducing the $2n$ input coefficients of $F(X)$ and $G(X)$ modulo the K primes p_i also takes time $O(nK \log N)$. A circular convolution of length n modulo p_i takes $O(n \log n)$ operations in $\mathrm{GF}(p_i)$, each of assumed cost $O(1)$. Hence the total time for a circular convolution modulo N is

$$O(nK \log N) + O(Kn \log n) = O(nK \log(nN)).$$

Since $K = O(\log(nN^2))$ by (3.4.1), this time bound is

$$O(n\log(nN^2)\log(nN)) = O(n(\log n + \log N)^2). \tag{3.4.5}$$

David Cantor mentioned another algorithm for polynomial products modulo N, due to David Robbins. Suppose we are given $F(X) = \sum_{i=0}^{n-1} f_i X^i$ and $G(X) = \sum_{i=0}^{n-1} g_i X^i$, where $0 \le f_i,\, g_i < N$. Select a radix R and write

$$f_i = \sum_{j=0}^{\ell-1} f_{ij} R^j, \qquad g_i = \sum_{j=0}^{\ell-1} g_{ij} R^j.$$

The coefficients of $F(X)G(X)$ and hence those of $F(X)G(X) \bmod N$ can be found from those of the polynomial product

$$\left(\sum_{i=0}^{n-1}\sum_{j=0}^{\ell-1} f_{ij} X^i Y^j\right)\left(\sum_{i=0}^{n-1}\sum_{j=0}^{\ell-1} g_{ij} X^i Y^j\right) \tag{3.4.6}$$

upon substituting $Y = R$. This polynomial product in (3.4.6) can be found using methods for univariate polynomial multiplication after setting $Y = X^{2\ell-1}$. This converts the original problem from a convolution of length $2n$ with coefficients at most $N-1$ to a convolution of length about $d = 2n(2\log_R N - 1)$ with coefficients at most $R-1$. The coefficients of the latter product are bounded by $(d/2)(R-1)^2$.

The latter convolution can be carried out by modular arithmetic; it can also be done by floating point arithmetic if R is sufficiently small. This allows the use of commercial FFT routines, which are often optimized for a particular architecture. For example, when 128–bit hardware floating point arithmetic (with mantissa circa 110 bits) becomes available, then $R \approx 2^{30}$ should be feasible even if $n \approx 2^{20}$ (degree 10^6) and $\ell \approx 2^{10}$ (9000–digit integers).

The estimated time for Robbins's algorithm is

$$O\Big(d\log d\ \log\big((d/2)(R-1)^2\big)\Big)$$
$$= O\Big(n\log N(\log n + \log\log N)^2\Big)$$

if R is kept fixed and we approximate the time for floating point arithmetic linearly in the number of bits of precision required. This is as good as (3.4.5) (except possibly for the constant factor) as $n \to \infty$ with N fixed, but better as $N \to \infty$ with n fixed. Either algorithm might be better in practice.

3.5 Polynomial reciprocals and division

Algorithms for fast polynomial division use Newton's method to compute a reciprocal, followed by a multiplication to compute the desired quotient. The remainder, if desired, can be found after another polynomial multiplication and subtraction.

If $F(X)$ is a polynomial of degree $n-1$ whose leading coefficient is invertible, then its *reciprocal* [2, p. 287] is defined to be the polynomial quotient

$$\mathrm{RECIP}(F) = \left\lfloor \frac{X^{2n-2}}{F(X)} \right\rfloor .$$

Observe that $F(X)/X^{n-1}$ is a finite Laurent series with invertible constant term, and $\mathrm{RECIP}(F)/X^{n-1}$ has precisely the terms through $X^{-(n-1)}$ in the Laurent series for $\left(F(X)/X^{n-1}\right)^{-1}$. The first term of the latter reciprocal can be found directly; we can then apply Newton's method at any time to double the accuracy. See [2, p. 287]. When n is a power of 2 and F has degree n, the algorithm in Figure 3.5.1 gives all $n+1$ coefficients of $\mathrm{RECIP}(F)$.

procedure $\mathrm{RECIP}\left(\sum_{j=0}^{n} f_j X^j\right)$
Cmt. Assume that n is a power of 2 and f_n is invertible.
$R_1(X) := 1/f_n$
$e_1 := -f_{n-1}/f_n$
for $k := 2,\ 4,\ 8,\ \ldots,\ n$ **do**
    let $\sum_{j=0}^{2k-3} h_j X^j := R_{k/2}(X)^2 \sum_{j=0}^{k-1} f_{n-j} X^{k-1-j}$
    $R_k(X) := 2R_{k/2}(X)X^{k/2} - \sum_{j=0}^{k-1} h_{j+k-2} X^j$
    $e_k := e_{k/2}^2 - h_{k-3} f_n - f_{n-k}/f_n$ (Use $h_{k-3} = 0$ if $k = 2$.)
end for
return $XR_n(X) + e_n/f_n$
end RECIP

Figure 3.5.1: Algorithm for polynomial reciprocals

It is straightforward to check that $R_k(X)$ has degree $k-1$. It is more tedious to verify the inductive assertion

$$\deg\Big((XR_k(X) + e_k/f_n)F(X) - X^{n+k}\Big) \le n - 1$$

for $k = 1, 2, 4, \cdots, n$. One proof defines $R'_k(X) = XR_k(X) + e_k/f_n$; the algorithm in Figure 3.5.1 is equivalent to

$$R'_1(X) = \frac{X}{f_n} - \frac{f_{n-1}}{f_n^2};$$

$$R'_k(X) = 2X^{k/2} R'_{k/2}(X) - \left\lfloor \frac{R_{k/2}(X)^2 \left\lfloor F(X)/X^{n-k} \right\rfloor}{X^k} \right\rfloor \qquad (k > 1).$$

When $\deg(F(X))$ is not a power of 2, it can be scaled up to the next power of 2, by multiplying by a power of X and adjusting the reciprocal accordingly. The computations of e_k in Figure 3.5.1 can then be suppressed, since the constant coefficient of the output is not used in this case.

To compute a quotient $\lfloor G(X)/F(X) \rfloor$ where $\deg(G(X)) \leq 2n-2$ and where $F(X)$ has degree $n-1$ with invertible leading coefficient, one can use the identity

$$\left\lfloor \frac{G(X)}{F(X)} \right\rfloor = \left\lfloor \frac{\left\lfloor G(X)/X^{n-1} \right\rfloor \mathrm{RECIP}(F(X))}{X^{n-1}} \right\rfloor. \tag{3.5.1}$$

To prove (3.5.1), let $H(X)$ be its right side. We must show that $\deg(G(X) - F(X)H(X)) \leq n-2$. Each summand on the right side of

$$\begin{aligned} X^{n-1}\Big(G(X) - F(X)H(X)\Big) = \quad & X^{n-1}\left(G(X) - X^{n-1}\left\lfloor \frac{G(X)}{X^{n-1}} \right\rfloor\right) \\ & - \left\lfloor \frac{G(X)}{X^{n-1}} \right\rfloor \Big(F(X)\,\mathrm{RECIP}(F(X)) - X^{2n-2}\Big) \\ & - F(X)\left(H(X)X^{n-1} - \left\lfloor \frac{G(X)}{X^{n-1}} \right\rfloor \mathrm{RECIP}(F(X))\right) \end{aligned}$$

is a product of polynomials of degrees at most $n-1$ and $n-2$, so its left side has degree at most $2n-3$, as desired.

Assuming (3.1.3), the times for polynomial reciprocal and division are bounded by $t_{\mathrm{inv}} + O\Big(M(n) + (t_{\mathrm{mul}} + t_{\mathrm{add}}) \log n\Big)$ [2, p. 288].

3.6 Constructing a monic polynomial from its roots

Let R be a ring and n be a power of 2. Given $a_i \in R$ for $0 \leq i < n$, we can compute the coefficients of

$$F(X) = \prod_{i=0}^{n-1} (X - a_i) \tag{3.6.1}$$

in time

$$(3.6.2) \qquad (M(n/2) + nt_{\text{add}}) \log_2 n + nt_{\text{add}},$$

by repeatedly invoking the procedure for multiplication.

The idea is to multiply two factors of equal degree at each stage. Define

$$F_{i,d}(X) = \prod_{j=0}^{d-1} (X - a_{i+j})$$

whenever $d|n$ and i is a multiple of d with $0 \le i \le n - d$. We are given the a_i and can compute the $F_{i,1}(X) = X - a_i$ with n negations. Construct

$$(3.6.3) \qquad F_{i,2d}(X) = F_{i,d}(X)\, F_{i+d,d}(X) \qquad (i = 0,\, 2d,\, 4d,\, \cdots,\, n - 2d)$$

from $\{F_{i,d}(X)\}$ for $i = 0, d, 2d, \cdots, n - d$ by pairwise multiplying $n/2d$ pairs of monic polynomials, each of degree d. Repeat this procedure $\log_2 n$ times to get $F_{0,n}(X) = F(X)$.

We can multiply two monic polynomials of degree d in time $M(d) + 2dt_{\text{add}}$ (by dropping the leading coefficients before doing the multiply). Doing this $n/2d$ times (as while replacing d by $2d$) costs $(n/2d)M(d) + nt_{\text{add}} \le M(n/2) + nt_{\text{add}}$ by (3.1.3). Repeating this $\log_2 d$ times (for $d = 1, 2, 4, \cdots, n/2$) gives the bound (3.6.2).

This algorithm needs temporary storage for at most $O(n)$ ring elements, since we can store the n coefficients of $\{F_{i,2d}(X)\}$ (excluding the leading 1's) atop the n coefficients of $\{F_{i,d}(X)\}$.

The bound $O((M(n) + nt_{\text{add}}) \log n)$ applies even if n is not a power of 2, since we can append some zeros to $\{a_i\}$ beforehand and divide by a power of X at the end.

3.7 Evaluating a polynomial at many points

Let $G(X)$ be a polynomial of degree at most $n - 1$ over a ring R, where n is a power of 2. Given $a_i \in R$ for $0 \le i < n$, we claim that we can evaluate all $G(a_i)$ in total time

$$(3.7.1) \qquad (7M(n/2) + 6nt_{\text{add}}) \log_2 n + nt_{\text{add}} + O(M(n))$$

using fast polynomial techniques [2, pp. 292–294]. We call the resulting algorithm POLYEVAL.

First we form the $\{F_{i,d}(X)\}$ in (3.6.3), obtaining $F(X) = \prod_{i=0}^{n-1}(X - a_i)$. Invert $F(X)$ to get $\text{RECIP}(F(X)) = \text{RECIP}(F_{0,n}(X))$ as in Section 3.5. We are given $G(X) \bmod F_{0,n}(X)$ since we assume $\deg(G(X)) < n$.

POLYEVAL proceeds recursively, in reverse order to that used during the construction of $F(X)$. Given $G(X) \bmod F_{i,2d}(X)$ and $\mathrm{RECIP}(F_{i,2d}(X))$, compute

$$\mathrm{RECIP}(F_{i,d}(X)) = \left\lfloor \frac{F_{i+d,d}(X) \left\lfloor \mathrm{RECIP}(F_{i,2d}(X))/X^d \right\rfloor}{X^d} \right\rfloor \tag{3.7.2}$$

and $\mathrm{RECIP}(F_{i+d,d}(X))$ similarly, as justified by (3.6.3). Next compute

$$G(X) \bmod F_{i,d}(X) = \big(G(X) \bmod F_{i,2d}(X)\big) \bmod F_{i,d}(X), \tag{3.7.3}$$

$$G(X) \bmod F_{i+d,d}(X) = \big(G(X) \bmod F_{i,2d}(X)\big) \bmod F_{i+d,d}(X),$$

using the reciprocals from (3.7.2) and the methods of (3.5.1).

Each step of this backwards recursion uses six products of polynomials which either are monic of degree d or have degree at most $d-1$, for a combined time of $6M(d)$. An extra $2d$ additions are used per reciprocal in (3.7.2) and per remainder in (3.7.3) to multiply by the X^d terms. There are also d subtractions required per remainder. Therefore the time for (3.7.2) and (3.7.3) is bounded by $6M(d) + 10dt_{\text{add}}$. These operations are repeated $n/2d$ times when replacing $2d$ by d, so the net time is bounded by $(3n/d)M(d) + 5nt_{\text{add}} \le 6M(n/2) + 5nt_{\text{add}}$. Summing this over all $\log_2 n$ levels of the recursion, and adding the time for constructing $F(X)$ and its reciprocal, we get the bound (3.7.1).

If polynomial multiplication is done modulo $X^d - 1$ or $X^{2d} - 1$ using an FFT algorithm, then there are many repeated operations in this calculation, due to multiplying two polynomials by the same polynomial. While using (3.6.3) to replace d by $2d$ and later using (3.7.2) and (3.7.3) to replace $2d$ by d, it suffices to compute six forward transforms of length $2d$, for the polynomials

$$\begin{array}{ll} F_{i,d}(X), & F_{i+d,d}(X), \\[1ex] \left\lfloor \dfrac{\mathrm{RECIP}(F_{i,2d}(X))}{X^d} \right\rfloor, & \left\lfloor \dfrac{G(X) \bmod F_{i,2d}(X)}{X^d} \right\rfloor, \\[2ex] \mathrm{RECIP}(F_{i,d}(X)), & \mathrm{RECIP}(F_{i+d,d}(X)), \end{array}$$

and four forward transforms of length d, for the polynomials

$$\begin{array}{ll} F_{i,d}(X), & F_{i+d,d}(X), \\[1ex] \left\lfloor \dfrac{G(X) \bmod F_{i,2d}(X)}{F_{i,d}(X)} \right\rfloor, & \left\lfloor \dfrac{G(X) \bmod F_{i,2d}(X)}{F_{i+d,d}(X)} \right\rfloor. \end{array}$$

(Since the remainders in (3.7.3) have degree at most $d-1$, it suffices to compute them modulo $X^d - 1$.) The transforms of $F_{i,d}(X)$ and $F_{i+d,d}(X)$ of length d can be obtained from the corresponding transforms of length $2d$, by extracting every second element (since the transform evaluates the polynomial at the d–th roots of unity, which are a subset of the $2d$–th roots of unity). We also need five pointwise products and reverse transforms (i.e. interpolations) of length $2d$, to get

$$
\begin{array}{ll}
F_{i,2d}(X), & \\[1em]
\mathrm{RECIP}(F_{i,d}(X)), & \mathrm{RECIP}(F_{i+d,d}(X)), \\[0.5em]
\left\lfloor \dfrac{G(X) \bmod F_{i,2d}(X)}{F_{i,d}(X)} \right\rfloor, & \left\lfloor \dfrac{G(X) \bmod F_{i,2d}(X)}{F_{i+d,d}(X)} \right\rfloor,
\end{array}
$$

and two of these of length d for computing the final remainders (3.7.3). If we approximate the cost of a forward transform or of a pointwise product and interpolation of length d as as $M(d/2)/3$, then the leading coefficient of our estimated time is

$$(6+5)M(2d/2d)/3 + (2+2)M(d/2)/3 \leq 13M(d)/3,$$

allowing us to replace the $7M(n/2)\log_2 n$ in (3.7.1) by $(13/3)M(n/2)\log_2 n$.

The temporary storage requirements of this algorithm are $O(n \log n)$ ring elements for storing all coefficients of $\{F_{i,d}(X)\}$. That is, intermediate storage requirements are $O(\log n)$ times the input size. Since the $F_{i,d}(X)$ polynomials are reused only once, these polynomials (or their forward transforms if using an FFT) can be saved in external storage rather than main memory, if the storage system supports LIFO access.

If the a_i have a special pattern, then we may be able to evaluate all $G(a_i)$ in less time than (3.7.1). If the a_i form a geometric progression, then all $G(a_i)$ can be found using one convolution of length $2n-1$ and $O(n)$ extra multiplications [2, exercise 8.27] [32, p. 844]. If instead the a_i form an arithmetic progression, then we can do it with $\deg(G)$ additions per $G(a_i)$ after suitable initialization (cf. Section 5.9).

3.8 Polynomial GCDs over a field

The standard Euclidean algorithm for polynomial greatest common divisors (GCDs) [17, pp. 405ff.] takes $O(n^2)$ operations when applied to two polynomials of degree at most n. Moenck [27] found an asymptotically faster algorithm using fast multiplication and division algorithms.

To simplify the presentation, this section assumes that the base ring R is a field. Section 3.11 presents the changes required when $R = \mathbb{Z}/N\mathbb{Z}$.

Definition 3.8.1 *Let M be a matrix of polynomials over a field. Then the* degree *of M, written* $\deg(M)$*, is the maximum of the degrees of the entries of M.*

Definition 3.8.2 *A 2×2 matrix M of polynomials over a field is* lopsided of degree n *if $n = 0$ and $M = \mathrm{I}_2$, or if $n > 0$ and*

$$M = \begin{pmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{pmatrix},$$

where

$$\deg(m_{11}) \leq n-2, \qquad \deg(m_{12}) \leq n-1, \qquad \deg(m_{21}) \leq n-1, \qquad \deg(m_{22}) = n.$$

For purposes of this definition, the zero polynomial has degree $-\infty$.

Lemma 3.8.3 *If M_1 is lopsided of degree n_1 and M_2 is lopsided of degree n_2, then $M_1 M_2$ is lopsided of degree $n_1 + n_2$.*

Proof. Straightforward. Since all computations are over a field, the product of two polynomials of degrees n_1 and n_2 has degree $n_1 + n_2$. ■

Figure 3.8.1 describes recursive procedure HALFGCD, which is an optimized version of procedure HGCD in [2, p. 364]. HALFGCD has two polynomial inputs U and V, and an integer input d_{red} telling how much to reduce one of the degrees before exiting; it has two polynomial outputs U_{out} and V_{out} and a 2×2 matrix output M_{out}. HALFGCD has two input requirements:

(HG1) U and V are polynomials in X over a field R, with $\deg(U) > \deg(V)$;

(HG2) d_{red} is a positive integer, with $d_{\mathrm{red}} \leq \lceil \deg(U)/2 \rceil$;

Theorem 3.8.4 (upcoming) shows that HALFGCD's outputs satisfy:

(HG3) M_{out} is a lopsided 2×2 matrix of degree $\deg(U) - \deg(U_{\mathrm{out}})$ and determinant ± 1;

(HG4) $\begin{pmatrix} U_{\mathrm{out}} \\ V_{\mathrm{out}} \end{pmatrix} = M_{\mathrm{out}} \begin{pmatrix} U \\ V \end{pmatrix}$;

procedure HALFGCD($U,\ V,\ d_{\text{red}},\ M_{\text{out}},\ U_{\text{out}},\ V_{\text{out}}$)
Cmt. Given U, V, d_{red} satisfying (HG1) and (HG2),
Cmt. finds M_{out}, U_{out}, V_{out} satisfying (HG3), (HG4), (HG5).
if $\deg(V) \leq \deg(U) - d_{\text{red}}$ **then** **Cmt**. Degree already small enough.

$$M_{\text{out}} := \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} U_{\text{out}} \\ V_{\text{out}} \end{pmatrix} := \begin{pmatrix} U \\ V \end{pmatrix}$$

else **Cmt**. Make two recursive calls to HALFGCD.
Cmt. Choose n with $0 \leq n \leq \deg(U) - 2d_{\text{red}} + 2$ (as large as convenient).
Cmt. Choose d' with $1 \leq d' \leq d_{\text{red}} - 1$ (as close to $\lceil d_{\text{red}}/2 \rceil$ as convenient).

$$\text{HALFGCD}\left(\left\lfloor \frac{U}{X^n} \right\rfloor,\ \left\lfloor \frac{V}{X^n} \right\rfloor,\ d',\ M',\ U',\ V' \right)$$

Cmt. $\deg(V') \leq \deg(U) - n - d' < \deg(U') = \deg(U) - n - \deg(M')$.
$d'' := \deg(V') - \deg(U) + n + d_{\text{red}}$
if $d'' \leq 0$ **then** **Cmt**. One recursive call is enough.

$$M_{\text{out}} := M' \quad \text{and} \quad \begin{pmatrix} U_{\text{out}} \\ V_{\text{out}} \end{pmatrix} := X^n \begin{pmatrix} U' \\ V' \end{pmatrix} + M_{\text{out}} \begin{pmatrix} U \bmod X^n \\ V \bmod X^n \end{pmatrix}$$

else

$$Q := \left\lfloor \frac{U'}{V'} \right\rfloor \quad \text{and} \quad W' := U' - QV' \qquad \textbf{Cmt}.\ W' = U' \bmod V'.$$

HALFGCD($V',\ W',\ d'',\ M'',\ V'',\ W''$)

$$M_{\text{out}} := M'' \begin{pmatrix} 0 & 1 \\ 1 & -Q \end{pmatrix} M'$$

$$\begin{pmatrix} U_{\text{out}} \\ V_{\text{out}} \end{pmatrix} := X^n \begin{pmatrix} V'' \\ W'' \end{pmatrix} + M_{\text{out}} \begin{pmatrix} U \bmod X^n \\ V \bmod X^n \end{pmatrix} \qquad \left(= M_{\text{out}} \begin{pmatrix} U \\ V \end{pmatrix} \right)$$

Cmt. $\deg(V_{\text{out}}) \leq \deg(U) - d_{\text{red}} < \deg(U_{\text{out}}) = \deg(V') + n - \deg(M')$.
end if
end if
end HALFGCD

Figure 3.8.1: Algorithm HALFGCD

(HG5) $\deg(V_{\text{out}}) \le \deg(U) - d_{\text{red}} \le \deg(U_{\text{out}}) - 1$.

A corollary of (HG3) and (HG5) is $\deg(M_{\text{out}}) \le d_{\text{red}} - 1$; this is useful when deciding how much storage to allocate for the matrix. A consequence of (HG3) and (HG4) is $\gcd(U,\ V) = \gcd(U_{\text{out}},\ V_{\text{out}})$.

Algorithm POLYGCD of Figure 3.8.2 computes an arbitrary polynomial GCD by repeatedly invoking HALFGCD. Each iteration of POLYGCD's main loop calls HALFGCD with a pair $\{U',\ V'\}$ such that $\gcd(U',\ V') = \gcd(U,\ V)$, where U and V are POLYGCD's original inputs. The output of HALFGCD is another pair $\{U'',\ V''\}$ with $\gcd(U'',\ V'') = \gcd(U',\ V')$ and where

$$\deg(V'') \le \deg(U') - \lceil \deg(U')/2 \rceil = \lfloor \deg(U')/2 \rfloor.$$

Unless $V'' = 0$, both the new $U' = V''$ and the new $V' = U' \bmod V''$ have degrees at most half the degree of the old U', so $\max(\deg(U'),\ \deg(V'))$ drops quickly.

procedure POLYGCD($U,\ V$)
Cmt. Assume that $\deg(U) > \deg(V)$.
$U' := U; \quad V' := V$
while $V' \ne 0$ **do**
  Cmt. By induction, $\deg(U') > \deg(V')$.
  HALFGCD($U',\ V',\ \lceil \deg(U')/2 \rceil,\ M_{\text{out}},\ U'',\ V''$)
  if $V'' = 0$ **then**
    $U' := U''; \quad V' := V''$
  else
    $U' := V''; \quad V' := U'' \bmod V''$
  end if
end while
return $U'(X)$ (normalized to a monic polynomial)

Figure 3.8.2: Algorithm for polynomial GCDs

Theorem 3.8.4 *Procedure* HALFGCD *always terminates after being invoked with inputs satisfying* (HG1) *and* (HG2). *Its outputs* M_{out}, U_{out}, *and* V_{out} *satisfy* (HG3), (HG4), (HG5).

Proof (long). We proceed by induction on d_{red}.

The simplest case is when $\deg(V) \leq \deg(U) - d_{\mathrm{red}}$. Then $M_{\mathrm{out}} = \mathrm{I}_2$ is the identity matrix, while $U_{\mathrm{out}} = U$ and $V_{\mathrm{out}} = V$. Since $\deg(M_{\mathrm{out}}) = 0$ and $\det(M_{\mathrm{out}}) = +1$, conditions (HG3) to (HG5) are trivially satisfied.

Otherwise the algorithm selects two integers n and d' such that

$$0 \leq n \leq \deg(U) - 2d_{\mathrm{red}} + 2 \quad \text{and} \quad 1 \leq d' \leq d_{\mathrm{red}} - 1. \tag{3.8.5}$$

The range for n is nonempty since $d_{\mathrm{red}} \leq (\deg(U)+1)/2$ by (HG2). The range for d' is nonempty since failure of the **if** means that $d_{\mathrm{red}} > \deg(U) - \deg(V)$, which is positive by (HG1). A corollary of (3.8.5) is

$$n + 2d' \leq \deg(U). \tag{3.8.6}$$

Next the procedure calls itself with U replaced by $\lfloor U/X^n \rfloor$, with V replaced by $\lfloor V/X^n \rfloor$, and with d_{red} replaced by d'. Requirement (HG1) is satisfied since

$$\deg\left(\left\lfloor \frac{U}{X^n} \right\rfloor\right) = \deg(U) - n > \deg(V) - n = \deg\left(\left\lfloor \frac{V}{X^n} \right\rfloor\right)$$

by (HG1). Requirement (HG2) is also satisfied, since

$$\left\lceil \frac{\deg\left(\lfloor U/X^n \rfloor\right)}{2} \right\rceil = \left\lceil \frac{\deg(U) - n}{2} \right\rceil \geq \left\lceil \frac{2d'}{2} \right\rceil = d'.$$

by (3.8.6). Since $d' < d_{\mathrm{red}}$ by (3.8.5), we may assume by induction that the outputs from the first recursive call satisfy

$$\deg(M') = \deg(U) - n - \deg(U'), \tag{3.8.7}$$

$$\det(M') = \pm 1, \tag{3.8.8}$$

$$\begin{pmatrix} U' \\ V' \end{pmatrix} = M' \begin{pmatrix} \lfloor U/X^n \rfloor \\ \lfloor V/X^n \rfloor \end{pmatrix},$$

$$\deg(V') \leq \deg(U) - n - d' \leq \deg(U') - 1, \tag{3.8.9}$$

with M' lopsided.

The algorithm defines d'' by

$$d'' = \deg(V') - \deg(U) + n + d_{\mathrm{red}}. \tag{3.8.10}$$

This satisfies

$$d'' \le d_{\text{red}} - d' \le d_{\text{red}} - 1 \tag{3.8.11}$$

by (3.8.9) and (3.8.5). If the second **if** in HALFGCD succeeds (i.e. if $d'' \le 0$), then the procedure exits with $M_{\text{out}} = M'$, which has determinant ± 1 by (3.8.8). It returns

$$\begin{aligned}
\begin{pmatrix} U_{\text{out}} \\ V_{\text{out}} \end{pmatrix} &= X^n \begin{pmatrix} U' \\ V' \end{pmatrix} + M' \begin{pmatrix} U \bmod X^n \\ V \bmod X^n \end{pmatrix} \\
&= X^n M' \begin{pmatrix} \lfloor U/X^n \rfloor \\ \lfloor V/X^n \rfloor \end{pmatrix} + M' \begin{pmatrix} U \bmod X^n \\ V \bmod X^n \end{pmatrix} \\
&= M' \begin{pmatrix} X^n \lfloor U/X^n \rfloor + (U \bmod X^n) \\ X^n \lfloor V/X^n \rfloor + (V \bmod X^n) \end{pmatrix} = M_{\text{out}} \begin{pmatrix} U \\ V \end{pmatrix};
\end{aligned} \tag{3.8.12}$$

this proves (HG4).

We claim that

$$\deg(U_{\text{out}}) = n + \deg(U'). \tag{3.8.13}$$

The term $X^n U'$ of U_{out} in the first line of (3.8.12) has degree exactly $n + \deg(U')$ while the contribution from $M' \begin{pmatrix} U \bmod X^n \\ V \bmod X^n \end{pmatrix}$ has degree at most

$$\begin{aligned}
\deg(M') + n - 1 = \deg(U) - \deg(U') - 1 &\le n + d' - 2 \\
\le \deg(U) - d' - 2 &\le n + \deg(U') - 3
\end{aligned}$$

by (3.8.7), (3.8.9) (twice), and (3.8.6). This proves (3.8.13) and also (HG3), since

$$\deg(M_{\text{out}}) = \deg(M') = \deg(U) - n - \deg(U') = \deg(U) - \deg(U_{\text{out}})$$

by (3.8.7).

The second inequality in (HG5) follows from

$$\deg(U_{\text{out}}) - 1 = n + \deg(U') - 1 \geq \deg(U) - d' > \deg(U) - d_{\text{red}}$$

by (3.8.13), (3.8.9), and (3.8.5). The first inequality in (HG5) follows from (3.8.12) since

$$\begin{aligned}
\deg(V_{\text{out}}) &\leq \max(n + \deg(V'),\ \deg(M') + n - 1) \\
&= \max(d'' + \deg(U) - d_{\text{red}},\ \deg(U) - \deg(U') - 1) \\
&\leq \max(\deg(U) - d_{\text{red}},\ n + d' - 2) \\
&\leq \max(\deg(U) - d_{\text{red}},\ \deg(U) - d_{\text{red}} - 1) = \deg(U) - d_{\text{red}}
\end{aligned}$$

by (3.8.10), (3.8.7), (3.8.9), and (3.8.5) (recall that $d'' \leq 0$).

The final subcase occurs when $d'' > 0$. The procedure computes the quotient $Q = \lfloor U'/V' \rfloor$ and remainder $W' = U' - QV'$, so that

$$\begin{aligned}
&\deg(Q) = \deg(U') - \deg(V'), \\
&\deg(W') < \deg(V').
\end{aligned} \tag{3.8.14}$$

Then it calls itself recursively with U, V, d_{red} replaced by V', W', d'', respectively. From (3.8.10), (3.8.11), and (3.8.5) it follows that

$$\begin{aligned}
2d'' &\leq (\deg(V') - \deg(U) + n + d_{\text{red}}) + (d_{\text{red}} - 1) \\
&= \deg(V') + 1 + (n - \deg(U) + 2d_{\text{red}} - 2) \\
&\leq \deg(V') + 1.
\end{aligned}$$

Hence $d'' \leq \lceil \deg(V')/2 \rceil$, fulfilling requirement (HG2) for the recursive call. Since $d'' < d_{\text{red}}$ by (3.8.11), we may assume by induction that the outputs M'', V'', and

W'' satisfy

$$\deg(M'') = \deg(V') - \deg(V''), \tag{3.8.15}$$

$$\det(M'') = \pm 1, \tag{3.8.16}$$

$$\begin{pmatrix} V'' \\ W'' \end{pmatrix} = M'' \begin{pmatrix} V' \\ W' \end{pmatrix},$$

$$\deg(W'') \le \deg(V') - d'' \le \deg(V'') - 1,$$

with M'' lopsided. We substitute the definition of d'' from (3.8.10) to deduce

$$\deg(W'') \le \deg(U) - n - d_{\text{red}} \le \deg(V'') - 1. \tag{3.8.17}$$

The procedure exits with

$$M_{\text{out}} = M'' \begin{pmatrix} 0 & 1 \\ 1 & -Q \end{pmatrix} M'.$$

Its determinant is $-\det(M'')\det(M') = \pm 1$ by (3.8.16) and (3.8.8). Lemma 3.8.3, together with (3.8.15), (3.8.14), and (3.8.7), shows that M_{out} is lopsided of degree

$$\begin{aligned} \deg(M_{\text{out}}) &= \deg(M'') + \deg(Q) + \deg(M') \\ &= (\deg(V') - \deg(V'')) + (\deg(U') - \deg(V')) \\ &\quad + (\deg(U) - n - \deg(U')) \\ &= \deg(U) - n - \deg(V''). \end{aligned} \tag{3.8.18}$$

The procedure also exits with

(3.8.19)

$$\begin{aligned}
\begin{pmatrix} U_{\text{out}} \\ V_{\text{out}} \end{pmatrix} &= X^n \begin{pmatrix} V'' \\ W'' \end{pmatrix} + M_{\text{out}} \begin{pmatrix} U \bmod X^n \\ V \bmod X^n \end{pmatrix} \\
&= X^n M'' \begin{pmatrix} V' \\ W' \end{pmatrix} + M_{\text{out}} \begin{pmatrix} U \bmod X^n \\ V \bmod X^n \end{pmatrix} \\
&= X^n M'' \begin{pmatrix} 0 & 1 \\ 1 & -Q \end{pmatrix} \begin{pmatrix} U' \\ V' \end{pmatrix} + M_{\text{out}} \begin{pmatrix} U \bmod X^n \\ V \bmod X^n \end{pmatrix} \\
&= X^n M'' \begin{pmatrix} 0 & 1 \\ 1 & -Q \end{pmatrix} M' \begin{pmatrix} \lfloor U/X^n \rfloor \\ \lfloor V/X^n \rfloor \end{pmatrix} + M_{\text{out}} \begin{pmatrix} U \bmod X^n \\ V \bmod X^n \end{pmatrix} \\
&= M_{\text{out}} \begin{pmatrix} X^n \lfloor U/X^n \rfloor + (U \bmod X^n) \\ X^n \lfloor V/X^n \rfloor + (V \bmod X^n) \end{pmatrix} = M_{\text{out}} \begin{pmatrix} U \\ V \end{pmatrix},
\end{aligned}$$

fulfilling (HG4).

We claim that

$$\deg(U_{\text{out}}) = n + \deg(V''). \tag{3.8.20}$$

The contributions to U_{out} in (3.8.19) comes from $X^n V''$ and $M_{\text{out}} \begin{pmatrix} U \bmod X^n \\ V \bmod X^n \end{pmatrix}$; the latter has degree at most

$$\begin{aligned}
\deg(M_{\text{out}}) + n - 1 = \deg(U) - \deg(V'') - 1 &\leq n + d_{\text{red}} - 2 \\
&\leq \deg(U) - d_{\text{red}} \leq \deg(V'') + n - 1
\end{aligned} \tag{3.8.21}$$

by (3.8.18), (3.8.17) (twice), and (3.8.5).

The formula for $\deg(M_{\text{out}})$ in (HG3) follows from (3.8.18) and (3.8.20). The second inequality of (HG5) follows from (3.8.20) and (3.8.17). The first inequality of (HG5) follows from

$$\deg(V_{\text{out}}) \le \max(n + \deg(W''),\ \deg(M_{\text{out}}) + n - 1)$$

$$\le \max(\deg(U) - d_{\text{red}},\ \deg(U) - d_{\text{red}}) = \deg(U) - d_{\text{red}},$$

by (3.8.19), (3.8.17), and (3.8.21). ■

3.9 Complexity analysis of fast GCD algorithm

Aho et al [2, p. 308] show that HALFGCD's time is $O\big(M(\deg(U)) \log \deg(U)\big)$ for $\deg(V) < \deg(U)$, if n and d' are selected optimally within the algorithm. Algorithm POLYGCD also has this time bound. In order to provide a more precise bound with explicit constants, we make the following simplifying assumptions about HALFGCD's behavior:

(a) d_{red} is a power of 2;

(b) $\deg(U) \equiv -2 \pmod{d_{\text{red}}}$ and $\deg(U) = \deg(V) - 1$;

(c) When $d_{\text{red}} > 1$, the algorithm chooses $n = \deg(U) - 2d_{\text{red}} + 2$ and $d' = d_{\text{red}}/2$;

(d) The outputs at each level of recursion satisfy

$$\deg(M_{\text{out}}) = d_{\text{red}} - 1,$$

$$\deg(U_{\text{out}}) = \deg(U) - \deg(M_{\text{out}}),$$

$$\deg(V_{\text{out}}) = \deg(U_{\text{out}}) - 1.$$

The division of U' by V' yields a quotient Q of degree 1 and a remainder W' of degree $\deg(V') - 1 = \deg(U') - 2$.

Assumptions (a) and (c) can be programmed. Assumptions (b) and (d) say that the leading coefficients of certain inputs and intermediate results never vanish; these are often valid when applying the algorithm with random inputs if the ring is large [17, p. 415], but are not guaranteed (and indeed are not satisfied when the eventual polynomial GCD has positive degree).

Given these assumptions, we claim that the time for HALFGCD is bounded by

(3.9.1)

$$(\deg(U) - 2d_{\text{red}} + 2)(8M(d_{\text{red}})/d_{\text{red}} + 10t_{\text{add}}) + (d_{\text{red}} - 1)t_{\text{inv}}$$

$$+ (d_{\text{red}}(6\log_2 d_{\text{red}} - 9) + 9)t_{\text{mul}} + (d_{\text{red}}(25\log_2 d_{\text{red}} - 30) + 15)t_{\text{add}}$$

$$+ M(d_{\text{red}})(16\log_2 d_{\text{red}} - 12).$$

for $d_{\text{red}} \geq 4$. There is no cost for $d_{\text{red}} = 1$.

Our assumptions mean that HALFGCD always takes both **else** branches in Figure 3.8.1 unless $d_{\text{red}} = 1$. Hence it calls itself twice recursively. The first recursive call has $d' = d_{\text{red}}/2$ with inputs of degrees $2d_{\text{red}} - 2 = 4d' - 2$ and $2d_{\text{red}} - 3 = 4d' - 3$. Its output matrix M' is lopsided of degree $d' - 1$, while U' and V' have degrees $3d' - 1$ and $3d' - 2$, respectively. The algorithm next computes $d'' = d' > 0$. The quotient Q is linear, and W' has degree $3d' - 3$. The second recursive call returns a lopsided matrix M'' of degree $d' - 1$ and outputs V'' and W'' of degrees $2d' - 1$ and $2d' - 2$ respectively. The algorithm then constructs the outputs M_{out}, U_{out}, and V_{out}, in this simplified scenario.

Excluding the recursive calls, the work in HALFGCD consists of (1) computing Q and W'; (2) computing M_{out}; and (3) computing U_{out} and V_{out}. Using the classical algorithm for polynomial division (since Q is assumed linear), the cost of (1) is

(3.9.2)
$$t_{\text{inv}} + 2t_{\text{mul}} + 2(\deg(V') + 1)(t_{\text{mul}} + t_{\text{add}})$$

$$= t_{\text{inv}} + 2t_{\text{mul}} + 2(3d' - 1)(t_{\text{mul}} + t_{\text{add}}).$$

When computing M_{out}, both M'' and M' are lopsided of (positive) degree $d' - 1$, while Q is linear. Using classical methods to multiply $\left(\begin{smallmatrix} 0 & 1 \\ 1 & -Q \end{smallmatrix}\right)M'$ takes time

$$(2d' - 1)(t_{\text{mul}} + t_{\text{add}}).$$

It remains to multiply two lopsided matrices of polynomials of degrees $d' - 1$ and d'; the degrees of their entries are

$$\begin{pmatrix} d' - 3 & d' - 2 \\ d' - 2 & d' - 1 \end{pmatrix}, \qquad \begin{pmatrix} d' - 2 & d' - 1 \\ d' - 1 & d' \end{pmatrix}.$$

We can compute M_{out} using eight multiplications of polynomials of degree at most $d'-1$, with $4(2d'-1)$ additions to combine these products, followed by $4d'-8$ multiplications and additions to multiply by the leading coefficient of the polynomial of degree d'. This shows that the time (2) for computing M_{out} is bounded by

(3.9.3)

$$(2d'-1)(t_{\text{mul}}+t_{\text{add}}) + 8M(d') + 4(2d'-1)t_{\text{add}} + (4d'-8)(t_{\text{mul}}+t_{\text{add}})$$

$$\leq 4M(2d') + (6d'-9)t_{\text{mul}} + (14d'-13)t_{\text{add}}.$$

To construct U_{out} and V_{out} in (3), we multiply the lopsided matrix M_{out} of degree d_{red} by a vector with two components of degree at most $n-1$. Assumptions (b) and (c) ensure that $n \equiv 0 \pmod{d_{\text{red}}}$, so we can split the computation into n/d_{red} matrix-vector products where all degrees are bounded by $d_{\text{red}}-1$. Each such product can be computed in time $8M(d_{\text{red}}) + 4(2d_{\text{red}}-1)t_{\text{add}}$. The upper half of each vector product must be added to another vector, giving a total time bounded by

$$\text{(3.9.4)} \qquad \frac{n}{d_{\text{red}}}\Big(8M(d_{\text{red}}) + 4(2d_{\text{red}}-1)t_{\text{add}} + 2(d_{\text{red}}-2)t_{\text{add}}\Big)$$

$$\leq 8nM(d_{\text{red}})/d_{\text{red}} + 10nt_{\text{add}}.$$

Adding (3.9.2), (3.9.3), and (3.9.4) while using $d' = d_{\text{red}}/2$ and $n = \deg(U) - 2d_{\text{red}} + 2$ gives a cost (excluding recursive calls) of

$$\text{(3.9.5)} \qquad \Big(t_{\text{inv}} + 2t_{\text{mul}} + 2(3d'-1)(t_{\text{mul}}+t_{\text{add}})\Big)$$

$$+ \Big(4M(2d') + (6d'-9)t_{\text{mul}} + (14d'-13)t_{\text{add}}\Big)$$

$$+ \Big((8n/d_{\text{red}})M(d_{\text{red}}) + 10nt_{\text{add}}\Big)$$

$$= n(8M(d_{\text{red}})/d_{\text{red}} + 10t_{\text{add}}) + 4M(d_{\text{red}})$$

$$+ t_{\text{inv}} + (12d'-9)t_{\text{mul}} + (20d'-15)t_{\text{add}}$$

$$= (\deg(U) - 2d_{\text{red}} + 2)(8M(d_{\text{red}})/d_{\text{red}} + 10t_{\text{add}})$$

$$+ 4M(d_{\text{red}}) + t_{\text{inv}} + (6d_{\text{red}}-9)t_{\text{mul}} + (10d_{\text{red}}-15)t_{\text{add}}$$

Both recursive calls replace d_{red} by $d' = d_{\text{red}}/2$; one replaces $\deg(U)$ by $4d' - 2$ and one by$\deg(U)$. Assuming (by induction) that the time bound (3.9.1) applies to these calls, their combined time is

$$
\begin{aligned}
(3.9.6) \quad & (d' + 2d')(8M(d')/d' + 10t_{\text{add}}) \\
& \quad + 2\Big((d' - 1)t_{\text{inv}} + (d'(6\log_2 d' - 9) + 9)t_{\text{mul}} \\
& \quad + (d'(25\log_2 d' - 30) + 15)t_{\text{add}} + M(d')(16\log_2 d' - 12)\Big) \\
& = (2d' - 2)t_{\text{inv}} + (2d'(6\log_2 d' - 9) + 18)t_{\text{mul}} \\
& \quad + (2d'(25\log_2 d' - 30) + 30)t_{\text{add}} + 16(2M(d'))\log_2 d' + 30d't_{\text{add}} \\
& \leq (d_{\text{red}} - 2)t_{\text{inv}} + (d_{\text{red}}(6\log_2 d_{\text{red}} - 15) + 18)t_{\text{mul}} \\
& \quad + (d_{\text{red}}(25\log_2 d_{\text{red}} - 40) + 30)t_{\text{add}} + M(d_{\text{red}})(16\log_2 d_{\text{red}} - 16).
\end{aligned}
$$

When $d_{\text{red}} = 2$, the recursive cost in (3.9.6) simplifies to zero, which is correct since the recursive calls for $d' = 1$ and and $d'' = 1$ are free. Adding (3.9.5) and (3.9.6) gives a cost of

$$
\begin{aligned}
& \Big((\deg(U) - 2d_{\text{red}} + 2)(8M(d_{\text{red}})/d_{\text{red}} + 10t_{\text{add}}) \\
& \quad + 4M(d_{\text{red}}) + t_{\text{inv}} + (6d_{\text{red}} - 9)t_{\text{mul}} + (10d_{\text{red}} - 15)t_{\text{add}}\Big) \\
& + [(d_{\text{red}} - 2)t_{\text{inv}} + (d_{\text{red}}(6\log_2 d_{\text{red}} - 15) + 18)t_{\text{mul}} \\
& \quad + (d_{\text{red}}(25\log_2 d_{\text{red}} - 40) + 30)t_{\text{add}} + (16\log_2 d_{\text{red}} - 16)M(d_{\text{red}})] \\
& = (\deg(U) - 2d_{\text{red}} + 2)(8M(d_{\text{red}})/d_{\text{red}} + 10t_{\text{add}}) + (d_{\text{red}} - 1)t_{\text{inv}} \\
& \quad + (d_{\text{red}}(6\log_2 d_{\text{red}} - 9) + 9)t_{\text{mul}} + (d_{\text{red}}(25\log_2 d_{\text{red}} - 30) + 15)t_{\text{add}} \\
& \quad + M(d_{\text{red}})(16\log_2 d_{\text{red}} - 12),
\end{aligned}
$$

in agreement with (3.9.1). This completes the induction.

To estimate the time of Algorithm POLYGCD (Figure 3.8.2) we use the leading terms

$$(3.9.7)\qquad (\deg(U) - 2d_{\text{red}} + 2)(8M(d_{\text{red}})/d_{\text{red}} + 10t_{\text{add}}) + (d_{\text{red}} - 1)t_{\text{inv}}$$

$$+ d_{\text{red}}(6t_{\text{mul}} + 25t_{\text{add}}) \log_2 d_{\text{red}} + 16M(d_{\text{red}}) \log_2 d_{\text{red}},$$

from our estimate (3.9.1). The calls from POLYGCD to HALFGCD always have $\deg(U) \approx 2d_{\text{red}} - 2$, allowing us to neglect the first term of (3.9.7) (the congruence assumption (b) need not hold for the call from POLYGCD to HALFGCD, but (b) does hold for the recursive calls from HALFGCD to itself). If the original degrees passed to POLYGCD are at most d, then HALFGCD is called successively with $d_{\text{red}} \approx d$, $d/2$, $d/4$, $\cdots$, 1. The polynomial divisions within POLYGCD have negligible cost unless the quotients have large degree. Its total estimated time is

$$(3.9.8)\qquad (d(6t_{\text{mul}} + 25t_{\text{add}}) + 32M(d/2)) \log_2 d.$$

This is about 32 times as long as the estimated time (3.6.2) for constructing a polynomial of degree d from its roots. Tables 9.1.1 and 9.1.2 suggest that the actual time ratio is closer to 10.

The storage costs of HALFGCD and POLYGCD are proportional to the size of the input data, if d' is chosen so that the degrees of the polynomials are approximately halved at successive levels of the recursion. The temporary storage requirements within HALFGCD are $O(\deg(U))$ ring elements. Summing over all levels of recursion, the combined temporary storage requirement is

$$(3.9.9)\qquad O(\deg(U)) + O(\deg(U)/2) + O(\deg(U)/4) + \cdots = O(\deg(U)).$$

The implied constant is rather large, since we need storage for the matrices of polynomials and for the transforms used during the convolutions.

3.10 Connection with polynomial resultants

If $F(X) = \sum_{j=0}^{d} f_j X^j$ is a polynomial of degree d, and m, n, k are integers, let $T_{m,n,k}(F)$ denote the $m \times n$ Toeplitz-like matrix $\{t_{ij}\}$ in which $t_{ij} = f_{i-j+n-m+k}$, where f_i is interpreted as 0 if $i < 0$ or $i > d$. Each row has some polynomial coefficients, which shift to the right by one column as we go down one row. The entry in its lower right corner is the coefficient of X^k. Pictorially,

(3.10.1)

$$T_{m,n,k}(F) = \begin{pmatrix} f_{k+n-m} & f_{k+n-m-1} & f_{k+n-m-2} & \cdots & f_{k-m+2} & f_{k-m+1} \\ f_{k+n-m+1} & f_{k+n-m} & f_{k+n-m-1} & \cdots & f_{k-m+3} & f_{k-m+2} \\ \vdots & \vdots & & \ddots & & \vdots \\ f_{k+n-2} & f_{k+n-3} & f_{k+n-4} & \cdots & f_k & f_{k-1} \\ f_{k+n-1} & f_{k+n-2} & f_{k+n-3} & \cdots & f_{k+1} & f_k \end{pmatrix}.$$

If $F(X)$ and $G(X)$ are polynomials in X of degrees m and n respectively, then their *resultant* is the $(m+n) \times (m+n)$ determinant

$$\operatorname{Res}(F,\, G) = \begin{vmatrix} T_{n,m+n,0}(F) \\ T_{m,m+n,0}(G) \end{vmatrix}. \tag{3.10.2}$$

The resultant of F and G vanishes if and only if F and G share a common polynomial factor [20, p. 210].

Output condition (HG4) can be expressed in terms of polynomial resultants. Suppose Algorithm HALFGCD of Figure 3.8.1 is called with two polynomials U and V, where $d = \deg(U) > \deg(V)$. Suppose the outputs are U_{out}, V_{out}, and $M_{\text{out}} = \begin{pmatrix} m_{11}(X) & m_{12}(X) \\ m_{21}(X) & m_{22}(X) \end{pmatrix}$. If $\det(M_{\text{out}}) = m$, then

$$\deg(m_{11}),\ \deg(m_{12}),\ \deg(m_{21}),\ \deg(m_{22}) \le m, \quad \deg(V_{\text{out}}) < \deg(U_{\text{out}}) = d - m.$$

by (HG3) and (HG5). We also know that

$$m = \deg(M_{\text{out}}) < d_{\text{red}} \le \lceil d/2 \rceil \le (d+1)/2$$

and hence $d \ge 2m - 1$. The matrix-vector equation (HG4) implies the following equation, where I_m denotes the $m \times m$ identity matrix and $0_{m,n}$ denotes the $m \times n$ zero matrix:

(3.10.3)

$$\left(\begin{array}{cc|cc} \mathrm{I}_m & 0_{m,d-m} & \multicolumn{2}{c}{0_{m,d}} \\ \multicolumn{2}{c|}{T_{d-m,d,0}(m_{11})} & \multicolumn{2}{c}{T_{d-m,d,0}(m_{12})} \\ \hline \multicolumn{2}{c|}{0_{m,d}} & \mathrm{I}_m & 0_{m,d-m} \\ \multicolumn{2}{c|}{T_{d-m,d,0}(m_{21})} & \multicolumn{2}{c}{T_{d-m,d,0}(m_{22})} \end{array}\right) \left(\begin{array}{c} T_{d,2d,0}(U) \\ \hline T_{d,2d,0}(V) \end{array}\right) = \left(\begin{array}{cc} T_{m,d+m,0}(U) & 0_{m,d-m} \\ \multicolumn{2}{c}{T_{d-m,2d,0}(U_{\mathrm{out}})} \\ \hline T_{m,d+m,0}(V) & 0_{m,d-m} \\ \multicolumn{2}{c}{T_{d-m,2d,0}(V_{\mathrm{out}})} \end{array}\right).$$

All three matrices in (3.10.3) are $2d \times 2d$. The first determinant

$$\Big(m_{11}(0)m_{22}(0) - m_{12}(0)m_{21}(0)\Big)^{d-m} = \Big(\det(M_{\mathrm{out}})(0)\Big)^{d-m} = \pm 1.$$

(After removing the rows and columns with the I_m blocks, the remaining matrix has four $(d-m) \times (d-m)$ lower triangular blocks whose diagonals are $m_{11}(0)$, $m_{12}(0)$, $m_{21}(0)$, and $m_{22}(0)$). The second determinant is $\mathrm{Res}(U, V)$ times the $(\deg(U) - \deg(V))$–th power of the leading coefficient of U. Hence the third determinant in (3.10.3) vanishes if and only if $\mathrm{Res}(U, V) = 0$. Because $\deg(U_{\mathrm{out}}) = d - m$ and $\deg(V_{\mathrm{out}}) < d - m$, the $T_{d-m,2d,0}(U_{\mathrm{out}})$ and $T_{d-m,2d,0}(V_{\mathrm{out}})$ blocks begin with $2m$ columns of zeros. The third determinant in (3.10.3) can be therefore decomposed into the product of the $2m \times 2m$ and $(2d - 2m) \times (2d - 2m)$ determinants

$$\det \begin{vmatrix} T_{m,2m,d-m}(U) \\ T_{m,2m,d-m}(V) \end{vmatrix}, \qquad \det \begin{vmatrix} T_{d-m,2d-2m,0}(U_{\mathrm{out}}) \\ T_{d-m,2d-2m,0}(V_{\mathrm{out}}) \end{vmatrix}. \tag{3.10.4}$$

The right determinant in (3.10.4) is $\mathrm{Res}(U_{\mathrm{out}}, V_{\mathrm{out}})$ times the $(\deg(U_{\mathrm{out}}) - \deg(V_{\mathrm{out}}))$–th power of the leading coefficient of U_{out}. From (HG3) and (HG4), we know that the common factors of U_{out} and V_{out} are precisely the common factors of U and V. Therefore it is plausible that the first determinant in (3.10.4) does not vanish; such is indeed the case. This is clear when $m = 0$, since the empty matrix has determinant 1. For $m > 0$, the first determinant vanishes only if its rows are linearly dependent, which is equivalent to the existence of polynomials $F(X)$ and $G(X)$ of degree at most $m - 1$ and not both zero with

$$\deg(F(X)U(X) + G(X)V(X)) \le d - m - 1. \tag{3.10.5}$$

But $U_{\text{out}}(X) = m_{11}(X)U(X) + m_{12}(X)V(X)$ has degree exactly $d - m$, while $\deg(m_{11}) \le m - 2$ and $\deg(m_{12}) \le m - 1$ (since M_{out} is lopsided). Comparing degrees on both sides of the identity

$$\begin{aligned} & U(X)\Big(m_{12}(X)F(X) - m_{11}(X)G(X)\Big) \\ = {} & m_{12}(X)\Big(F(X)U(X) + G(X)V(X)\Big) - G(X)\Big(m_{11}(X)U(X) + m_{12}(X)V(X)\Big) \end{aligned}$$

gives the contradiction $d = \deg(U) \le (m - 1) + (d - m)$ unless both sides of the identity vanish, in which case $m_{12}(X)F(X) = m_{11}(X)G(X)$. Since $m_{11}(X)$ and $m_{12}(X)$ have no common factor by (HG3), there must exist $Q(X)$ such that $F(X) = Q(X)\,m_{11}(X)$ and $G(X) = Q(X)\,m_{12}(X)$. Plugging these into (3.10.5) shows that $\deg(Q(X)\,U_{\text{out}}(X)) \le d - m - 1$, a contradiction.

Knuth [17, pp. 410ff.] discusses the intermediate polynomials which occur in the Euclidean GCD algorithm, in terms of resultants.

Schwartz [37, pp. 705–707] describes how to compute the resultant of two polynomials using Moenck's fast GCD algorithm.

3.11 Polynomial GCDs over $\mathbb{Z}/N\mathbb{Z}$

The concept of a polynomial GCD over $\mathbb{Z}/N\mathbb{Z}$ is not well-defined when N is composite, this will be turned to our advantage in Chapter 4. For example, suppose we try to compute

$$\gcd(X^2 + 9X + 8,\ 2X + 9) \pmod{35}$$

using the Euclidean algorithm. We start by dividing $X^2 + 9X + 8$ by $2X + 9$, getting a quotient of $18X + 11$ and a remainder of 14 modulo 35. Next we try to divide $2X + 9$ by 14, but 14 is not invertible. Instead we discover the factor $\gcd(35, 14) = 7$ of 35. The explanation is that

$$\gcd(X^2 + 9X + 8,\ 2X + 9) = 1 \pmod 5$$

$$\text{but}\quad \gcd(X^2 + 9X + 8,\ 2X + 9) = X + 1 \pmod 7.$$

The GCD has degree 0 modulo 5 but degree 1 modulo 7. No monic polynomial can meet both requirements.

Nonetheless, we try to follow Algorithm POLYGCD in Figure 3.8.2 using arithmetic in $\mathbb{Z}/N\mathbb{Z}$ instead of arithmetic over a field. We assume that the original polynomials U and V satisfy $\deg(U) > \deg(V)$ and that the leading coefficient of U is invertible modulo N (indeed, U will be monic in Section 4.3). The latter condition ensures that $\deg(U) > \deg(V)$ (which is requirement (HG1) preceding Theorem 3.8.4) remains true even after the coefficients of U and V are reduced modulo p for some $p|N$.

Theorem 3.11.1 *Suppose Algorithm* POLYGCD *is applied to two polynomials U and V over $\mathbb{Z}/N\mathbb{Z}$, where* $\deg(U) > \deg(V)$ *and the leading coefficient of U is invertible modulo N. Then either* (i) *the algorithm finds a monic GCD of U and V over $\mathbb{Z}/N\mathbb{Z}$, or* (ii) *some polynomial division has a non-zero, non-invertible leading coefficient. When* (ii) *occurs and a factor p is found, either* (i) *or* (ii) *remains true for the ring $\mathbb{Z}/(N/p)\mathbb{Z}$.*

Proof. Consider the situation just before case (ii) first occurs, if ever.

We claim that every previous call to HALFGCD had an invertible leading coefficient for U. Every such coefficient was one of:

(a) The leading coefficient of the original U passed to POLYGCD in Figure 3.8.2;

(b) The leading coefficient of a polynomial passed to HALFGCD for U at a higher level (in the first recursive call to HALFGCD in Figure 3.8.1);

(c) The leading coefficient of a polynomial used for a division (in the second recursive call to HALFGCD in Figure 3.8.1, and in calls from POLYGCD to HALFGCD after the first iteration of the **while** loop in Figure 3.8.2).

In each case the assertion follows by induction.

Letting $M_{\text{out}} = \binom{m_{11}\ m_{12}}{m_{21}\ m_{22}}$, we now use (HG4) to obtain

$$\begin{pmatrix} U \\ V \end{pmatrix} = M_{\text{out}}^{-1} \begin{pmatrix} U_{\text{out}} \\ V_{\text{out}} \end{pmatrix} = \frac{1}{\det(M_{\text{out}})} \begin{pmatrix} m_{22} & -m_{12} \\ -m_{21} & m_{11} \end{pmatrix} \begin{pmatrix} U_{\text{out}} \\ V_{\text{out}} \end{pmatrix}.$$

Hence $U = \pm(m_{22}U_{\text{out}} - m_{12}V_{\text{out}})$. Since M_{out} is lopsided and $\deg(V_{\text{out}}) < \deg(U_{\text{out}})$ by (HG5), the leading coefficient of U is (up to sign) the product of the leading coefficients of m_{22} and of U_{out}. Consequently both of the latter are invertible modulo N.

When (ii) first occurs, during computation of $\lfloor U'/V' \rfloor$ in HALFGCD or of $U'' \bmod V''$ in POLYGCD, we find a (proper but not necessarily prime) factor p of N. All previous input and output assertions for HALFGCD and POLYGCD remain true over the ring $\mathbb{Z}/(N/p)\mathbb{Z}$ (the degrees of V and V_{out} may drop, but those of U, U_{out}, and M_{out} remain unchanged, and M_{out} remains lopsided). The only way these assertions might fail is if $1 \equiv 0 \pmod{N/p}$; in such case the leading coefficient of V' or V'' was zero modulo N, not merely a zero divisor of $\mathbb{Z}/N\mathbb{Z}$.

When we finish POLYGCD, and are working modulo some divisor N' of N, there will exist a matrix M of determinant ± 1 over $\mathbb{Z}/N'\mathbb{Z}$ such that $\binom{U'}{V'} = M\binom{U}{V}$. This, together with the exit condition $V' = 0$, implies that U' generates the same ideal as U and V in the polynomial ring $(\mathbb{Z}/N'\mathbb{Z})[X]$. Since the leading coefficient of U' is invertible modulo N', this generator can be normalized to a monic polynomial. ■

Remark 3.11.2 CAUTION. *Although the degree of U does not change under reduction modulo N/p, the degree of V may drop if N is not squarefree. For example, this occurs while attempting to divide $X^2 + 1$ by $3X + 5$ modulo 9.*

3.12 Opportunities for optimization and parallelization

The classical FFT algorithm over the complex numbers has many opportunities for parallelization [3, Chapter 9]. The algorithm in Section 3.4 shares many of these opportunities, working modulo the p_i instead of over $\mathbb{C}$. The convolution is done separately for each p_i; these computations can be arranged in vector or SIMD fashion. If one chooses to vectorize over the p_i, then one can ensure unit strides everywhere by storing residues modulo different primes contiguously.

This convolution algorithm reduces each input coefficient modulo all the primes p_i. These reductions can proceed concurrently.

After the individual convolutions modulo p_i are complete, the Chinese Remainder Theorem is used to construct the outputs h_j in (3.4.4). Computations for different j are independent and can be parallelized.

Algorithm HALFGCD multiplies two matrices, or a matrix and a vector. When using FFT-like multiplication, one can do forward transforms on all elements of each matrix, perform the matrix multiplication directly on the Fourier transforms of the input data, and take one inverse Fourier transform per element of the output matrix (cf. Section 3.7). When multiplying two 2×2 matrices, this trick cuts the work almost in half. When using the algorithm in Section 3.4 for convolutions over $\mathbb{Z}/N\mathbb{Z}$, the lower bound for P in (3.4.1) should be adjusted to reflect the largest possible coefficient which might appear in the matrix product before reduction modulo N.

Some cost estimates in this chapter are pessimistic. For example, when multiplying two polynomials both of degree d with d a power of 2, we multiplied by the leading coefficients separately. If our convolution algorithm produces a product modulo $X^{2d} - 1$, then the leading (or constant) coefficient of the product can be calculated directly, and the proper multiple of $X^{2d} - 1$ added on. Similarly, a polynomial remainder known to have degree at most $d-1$ is uniquely determined if we compute it modulo $X^d - 1$.

Both POLYEVAL and the algorithm for constructing a polynomial from its roots proceed by divide and conquer. Each divides the problem into two pieces, which may be attacked independently. For example, we may compute all $F_{i,2d}$ for fixed d in (3.6.3) concurrently. The same applies to (3.7.2) and (3.7.3). For small d, we can compute $F_{i,2d}$ for several different i at once, if the original degree is sufficiently large. On the other hand, if d is large, then there may be only a few values of i, but parallelism can be employed within the convolutions themselves.

The polynomial GCD algorithm also uses divide and conquer, but its second recursive call cannot begin until the first is complete, severely restricting its potential parallelism. There is some parallelism during the larger convolutions, but little during the highly nested stages of the recursion. Whenever Algorithm HALFGCD computes a quotient Q, it inverts the leading coefficient of the denominator; no good parallelizable algorithm for modular inversion is known. This distinction is reflected in Table 9.1.2, which shows speedups approaching 4.3 using five processors to construct a polynomial from its roots, but at most 3.7 for the polynomial GCD problem.

CHAPTER 4

Application to ECM

Step 1 of ECM multiplies an initial point on an elliptic curve by a large integer, obtaining another point Q on the curve. Step 2 assumes that the reduction $Q_{(p)}$ of Q over GF(p) has small positive order q for some $p|N$. It constructs several multiples of Q and looks at their x-coordinates, hoping for a match modulo p. The birthday paradox, described by Brent [9, pp. 10–12], predicts that if we randomly pick $O(\sqrt{q})$ integers m_i, then some $m_i \equiv \pm m_j \pmod{q}$ for $i \neq j$. This leads to a match among the x-coordinates of $m_i \cdot Q_{(p)}$ and $m_j \cdot Q_{(p)}$. This chapter describes how to use fast polynomial techniques when testing for a match modulo p.

4.1 Checking two lists for matches modulo p, where $p|N$

Suppose that $p|N$ and we are given two lists $\{a_i\}_{i=0}^{d_1-1}$ and $\{b_j\}_{j=0}^{d_2-1}$ of values modulo N, where $0 \leq a_i < N$ and $0 \leq b_j < N$. How can we check whether there exist i and j such that

$$(4.1.1) \qquad a_i \equiv b_j \pmod{p} \qquad (0 \leq i \leq d_1 - 1, \quad 0 \leq j \leq d_2 - 1)?$$

A related question asks for duplicates within either list; for example do there exist solutions of

$$(4.1.2) \qquad a_i \equiv a_j \pmod{p} \qquad \text{where} \qquad 0 \leq i < j \leq d_1 - 1\,?$$

If p were known, then we could reduce all a_i and b_j modulo p, sort both lists, and look for duplicates. This takes time $O((d_1 + d_2) \log N \log p)$ for the reductions modulo p using classical division algorithms, and $O((d_1 \log d_1 + d_2 \log d_2) \log p)$ for the sorting since the entries have size $O(\log p)$. If d_1, $d_2 < N$, then these processing times are only $O(\log p)$ times the input size $O((d_1 + d_2) \log N)$. Another approach, also for known p, uses a hash table [2, pp. 111ff.].

Lubiw and Rácz [26] obtain a lower bound of $\Omega(n \log n)$ operations when checking whether n elements are distinct.

For the intended application, p is unknown. The real question is whether there exists a factor p of N such that some $a_i \equiv b_j \pmod{p}$ (or $a_i \equiv a_j$); if so we want to find p. Sorting seems inapplicable since no comparison function is known.

Hashing seems inapplicable since we do not know how to hash in such a way that two numbers congruent modulo p will hash to the same value.

We can form the polynomial

$$F(X) = \prod_{i=0}^{d_1-1} (X - a_i) \bmod N$$

in time $(M(d_1/2) + d_1 t_{\text{add}}) \log_2 d_1 + d_1 t_{\text{add}}$, as shown in Section 3.6. Problem (4.1.2) asks whether F has multiple roots modulo p for some $p|N$. Any such multiple root of F modulo p must be a root of its formal derivative F' modulo p. We therefore attempt the computation

$$\gcd(F(X),\ G(X)) \pmod N. \tag{4.1.3}$$

where $G(X) = F'(X)$. Problem (4.1.1) asks whether $F(b_j) \equiv 0 \pmod p$ for some j and some $p|N$. Here we attempt (4.1.3) with $G(X) = \prod_{j=0}^{d_2-1}(X - b_j) \bmod N$.

By Theorem 3.11.1, Algorithm POLYGCD in Figure 3.8.2 produces a monic polynomial $g(X)$ which divides both $F(X)$ and $G(X)$ or finds a factor p of N (not necessarily prime). If we are lucky, it may find several factors of N. When a monic GCD is found, either it has degree zero (implying there were no solutions of (4.1.2) for any $p|N$) or it has positive degree. The latter case seems unlikely when N is not a prime power, unless either $a_i = a_j$ for some $i \neq j$ (resp. $a_i = b_j$ for some i and j) or N has only small prime factors, as it means that (4.1.2) holds for all $p|N$ (though not necessarily with the same i and j).

4.2 Use of fast polynomial evaluation

Brent [9, p. 13] suggests another way to do the match. After constructing $F(X)$ and $G(X)$ as above, check whether $G(a_i) \equiv 0 \pmod p$ for some $p|N$ and some i with $0 \le i \le d_1 - 1$. That is, check whether

$$\gcd\left(N,\ \prod_{i=0}^{n-1} G(a_i)\right) > 1.$$

Algorithm POLYEVAL of Section 3.7 requires that $\deg(G(X)) < \deg(F(X))$; this can be ensured by taking a remainder initially. Since POLYEVAL computes RECIP($F(X)$), the additional overhead for the polynomial division is small.

We saw earlier that (3.7.1) estimates time $7M(n/2)\log_2 n$ for POLYEVAL, whereas (3.9.8) estimates time $32M(n/2)\log_2 n$ for POLYGCD. These suggest that POLYEVAL is faster than POLYGCD, while also being more parallelizable (Section 3.12).

POLYEVAL seems simpler to program that POLYGCD, because its recursion is readily replaced by iteration. POLYEVAL is also easy to test, since one can independently evaluate a few $G(a_i)$ the long way to check results. While coding POLYGCD, one must worry about finding a factor p of N midway through the algorithm, with all future calculations done modulo N/p. POLYEVAL does not require any modular inversions and does not attempt a GCD with N until all $G(a_i)$ have been computed, avoiding this problem (but there is still the prospect of encountering a zero divisor while computing the a_i themselves, such as while computing a slope in (2.0.3)).

However, POLYGCD retains one tangible advantage over POLYEVAL: less storage is required for large d_1. We observed near the end of Section 3.9 that algorithms HALFGCD and POLYGCD use $O(n)$ ring elements for intermediate storage when taking the GCD of two polynomials of degree at most n. POLYEVAL's storage requirements are $O(n \log n)$ ring elements when evaluating a polynomial of degree at most $n-1$ at n points, as observed in Section 3.7. This consideration may diminish in importance as memories grow and parallelism becomes more important, esp. since the storage (3.9.9) required by POLYGCD and HALFGCD has a large proportionality constant.

Table 7.4.1 suggests values of d_1 and d_2 when searching for factors of various sizes. The suggested d_1 is below 4096 until one searches for factors of 35 digits, so the extra storage may be affordable. But the data in Table 7.4.1 assume that POLYGCD is used; switching to POLYEVAL affects the cost equation (7.3.1) and the optimal parameters.

4.3 Construction of polynomials

Assume that we are using ECM to factor an integer N, as in Section 2.2. Let Q be the output of Step 1 of ECM, after selecting an initial point P_0 on a curve and computing a large multiple of P_0. If $p|N$, then the reduction $Q_{(p)}$ of Q in $E_{(p)}$ has finite order, say q. We hope that q is not too large, say $q < 10^9$.

Select two disjoint sets $\{m_i\}_{i=0}^{d_1-1}$ and $\{n_j\}_{j=0}^{d_2-1}$ of large positive integers; we will assume later that d_1 is a power of 2 and $d_1|d_2$. The selection process is discussed in detail in Chapter 5; that process need not concern us here. Form the two sets $\{a_i\}_{i=0}^{d_1-1}$ and $\{b_j\}_{j=0}^{d_2-1}$, where

$$a_i = (m_i \cdot Q)_x \qquad (0 \le i \le d_1 - 1), \tag{4.3.1}$$

$$b_j = (n_j \cdot Q)_x \qquad (0 \le j \le d_2 - 1)$$

are x-coordinates of selected multiples of Q. If we are sufficiently lucky in this selection, then

$$m_i \equiv \pm n_j \pmod{q} \tag{4.3.2}$$

for some i and j; the corresponding a_i and b_j in (4.3.1) satisfy (4.1.1).

Next compute the polynomials

$$F(X) = \prod_{i=0}^{d_1-1} (X - a_i) \pmod{N}, \quad G(X) = \prod_{j=0}^{d_2-1} (X - b_j) \pmod{N}. \tag{4.3.3}$$

These have a common root modulo p if (4.1.1) holds for some i and j. Either POLYGCD or POLYEVAL can be used to check for a match.

If $d_1 \approx d_2$, then $F(X)$ and $G(X)$ in (4.3.3) can be constructed directly, as in Section 3.6; then POLYGCD or POLYEVAL can be applied to F and to G mod F (reducing G modulo F ensures that the constraint $\deg(U) > \deg(V)$ in Figure 3.8.2 is satisfied).

This test has approximately

$$\deg(F)\deg(G) = d_1 d_2$$

opportunities for a match; if the sets $\{m_i\}$ and $\{n_j\}$ are chosen carefully, there will be a match in (4.3.2) for most $q < 2d_1d_2$. If we want to find p whenever $q < q_{\max}$ for some pre-selected $q_{\max}$, then we need

$$d_1 d_2 \geq C q_{\max}, \tag{4.3.4}$$

for some positive constant C.

If (4.3.4) were the only constraint on d_1 and d_2, then we would choose $d_1 \approx d_2$ so as to minimize $d_1 + d_2$ and hence the total time for constructing the $\{a_i\}$ and $\{b_j\}$ in (4.3.1). Memory requirements impose another constraint. If $q_{\max} = 10^9$ and $C = 1$, for example, then (4.3.4) suggests taking d_1, $d_2 \approx 2^{15} = 32768$. For $N \approx 10^{200}$, each residue modulo N occupies about 90 bytes, so a polynomial of degree 32768 occupies 2.9 megabytes. The input to POLYGCD then requires 5.8 megabytes, and the storage of the output matrix M_{out} is also approximately this large if HALFGCD is called with $d_{\text{red}} = \lceil \deg(U')/2 \rceil$ as in Figure 3.8.2. More space is used by the convolutions, such as when constructing M_{out} and $\begin{pmatrix} U_{\text{out}} \\ V_{\text{out}} \end{pmatrix}$ at the end of Figure 3.8.1. This work was run on systems shared by other users, and it is unfriendly to hog the memory even if it is available. So one should keep d_1 small, say $d_1 \leq 2^{13}$.

One remedy selects $d_2 > d_1$, with $d_1 | d_2$. Compute $F(X)$ as in (4.3.3), but modify the construction of $G(X)$ as in Figure 4.3.1 below. This keeps $\gcd(F(X),\ G(X))$

$$
\boxed{\begin{array}{l}
G(X) := 1 \\
\textbf{for } j \textbf{ from } 1 \textbf{ to } d_2/d_1 \textbf{ do} \\
\quad H_j(X) := \displaystyle\prod_{i=(j-1)d_1}^{jd_1-1} (X - b_i) \pmod{N} \\
\quad G(X) := G(X)H_j(X) \pmod{F(X),\ N} \\
\textbf{end for}
\end{array}}
$$

Figure 4.3.1: Computing $G(X)$ mod $F(X)$ in pieces of degree d_1

unchanged, but leaves $\deg(G(X))$ bounded by $d_1 - 1$ rather than by d_2. If $d_1 \ll d_2$, then this can save memory, though at a cost of more computation in (4.3.1).

Each iteration of the inner loop in Figure 4.3.1 divides a polynomial of degree at most $2d_1 - 1$ by $F(X)$ of degree d_1. The discrete forward transform (defined in Section 3.3) of the reciprocal RECIP(F) needs to be computed only once, as does the discrete forward transform of F itself.

On the first iteration of the inner loop in Figure 4.3.1, the computation of $G(X)$ reduces to $G(X) := H_1(X) - F(X) \pmod{N}$ since H_1 and F are monic of degree d_1. A variation initializes

$$G(X) := F'(X)$$

instead of $G(X) := 1$; this variation finds a match if $a_{i_1} \equiv a_{i_2} \pmod{p}$ for some $i_1 \neq i_2$, and hence if two of the m_i (or their negatives) agree modulo q.

Since the FFT algorithms in Sections 3.3 and 3.4 are designed for length a power of 2, it is convenient to choose d_1 as a power of 2. POLYEVAL and the polynomial construction algorithm in Section 3.6 also work well with this choice.

CHAPTER 5

Selection and Generation of Multiples of Q

In Section 4.3 we chose integers d_1 and d_2, with d_1 a power of 2 and $d_1|d_2$ and $d_2 \gg d_1$. Let Q be the output of Step 1 of ECM, and suppose that the reduction $Q_{(p)}$ has prime order q for some prime $p|N$. We hope that q is not too large, say $q < 10^9$. The algorithm uses x-coordinates $(m_i \cdot Q)_x$ for $0 \le i < d_1$ and $(n_j \cdot Q)_x$ for $0 \le j < d_2$ as polynomial roots, usually finding p if $q|(m_i \pm n_j)$ for some i, j with

$$0 \le i < d_1 \quad \text{and} \quad 0 \le j < d_2 \tag{5.0.1}$$

or (in the variation to Figure 4.3.1) if $q|(m_{i_1} \pm m_{i_2})$ for some i_1, i_2 with

$$0 \le i_1,\ i_2 < d_1 \quad \text{and} \quad i_1 \ne i_2. \tag{5.0.2}$$

The algorithm does not specify how to select the sequences $\{m_i\}$ and $\{n_j\}$. The next sections describe the strategy used and some motivation behind it.

Some desired properties (DP's) of these sequences are:

(DP1) Most small and moderate primes q should divide some $m_{i_1} \pm m_{i_2}$ satisfying (5.0.2) or some $m_i \pm n_j$ satisfying (5.0.1). It is acceptable if q instead divides some m_i or n_j, since the computation of that $m_i \cdot Q$ or $n_j \cdot Q$ reveals the factor p while inverting a denominator.

If this property holds for all primes $q < d_1 d_2$ say, then we can honestly claim to have $B_2 \ge d_1 d_2$ in the notation of [29].

(DP2) The sums and differences $m_{i_1} \pm m_{i_2}$ and $m_i \pm n_j$ should have many prime divisors, not just the ones ensured by (DP1). However, no such sum or difference should be identically zero. No m_i or n_j should be identically zero. The n_j should be distinct.

(DP3) The average time for computing an x-coordinate $(m_i \cdot Q)_x$ or $(n_j \cdot Q)_x$ should not exceed 50–100 multiplications modulo N. This figure was selected by equating the time for computation of the roots of H in (4.3.1) to the combined time for building H and updating $G \leftarrow GH \pmod F$ in Figure 4.3.1; the figure is implementation-dependent.

(DP4) The above computations of the $(m_i \cdot Q)_x$ and $(n_j \cdot Q)_x$ should be amenable to parallel computation on a machine with 4 to 32 parallel processors and shared memory, such as the Alliant FX/80 in UCLA's Department of Mathematics (whose cluster had six processors during the early parts of this study but five at the end).

(DP5) If there can be many duplicate n_j's modulo q, we try to ensure a match $m_{i_1} \equiv \pm m_{i_2} \pmod q$ satisfying (5.0.2) or $m_i \equiv \pm n_j \pmod q$ satisfying (5.0.1). Ideally, either all n_j are distinct modulo q, so that we have the full $d_1 d_2$ opportunities for a congruence $m_i \equiv \pm n_j$, or we have a guaranteed match.

(DP6) Minimize the number of instances where two pairs $m_{i_1} \pm n_{j_1}$ and $m_{i_2} \pm n_{j_2}$ share large factors (or one divides the other) due to algebraic identities.

Section 5.1 tries to achieve these objectives while letting each m_i and n_j be a k–th power or a value attained by a Dickson polynomial. Some advantages and disadvantages of each choice are presented. Section 5.9 describes how to evaluate successive $m_i \cdot Q$ quickly when m_i is a polynomial function of i. Section 5.10 describes the choices made in the implementation.

5.1 Use of k–th powers or Dickson polynomials

We attempt to satisfy (DP2) by letting $m_i = P(M_i)$ and $n_j = P(N_j)$ for selected integers $\{M_i\}$ and $\{N_j\}$, where P is a polynomial such that $P(X) \pm P(Y)$ together have many polynomial divisors. One such polynomial is $P(X) = X^k$ where $2k$ is highly composite, since $(P(X)-P(Y))(P(X)+P(Y)) = X^{2k} - Y^{2k}$ has $d(2k)$ irreducible polynomial factors [20, p. 315]. Here $d(2k)$ denotes the number of divisors of $2k$ (including 1 and $2k$).

Perhaps surprisingly, there are other monic polynomials P of degree k for which $P(X) \pm P(Y)$ have a total of $d(2k)$ irreducible polynomial factors.

Definition 5.1.1 *Let k be a positive integer. For fixed α, define the Dickson polynomial $g_{k,\alpha}$* [25, pp. 355ff.] *by the formal identity*

$$g_{k,\alpha}\left(X + \frac{\alpha}{X}\right) = X^k + \frac{\alpha^k}{X^k}.$$

It is easy to verify that $g_{k,\alpha}$ is a monic polynomial of degree k, because

$$g_{1,\alpha}(X) = X,$$

$$g_{2,\alpha}(X) = X^2 - 2\alpha,$$

$$g_{k+2,\alpha}(X) = Xg_{k+1,\alpha}(X) - \alpha g_{k,\alpha}(X) \qquad (k \geq 1).$$

If $\alpha = 0$ and $k > 0$, then $g_{k,\alpha}(X) = X^k$. When $\alpha \neq 0$, the Dickson polynomials are related to the Chebyshev polynomials [1, chapter 22], since

$$g_{k,\alpha}(2\alpha^{1/2}\cos\theta) = 2\alpha^{k/2}\cos k\theta.$$

5.2 Polynomial divisors of $g_{k,\alpha}(X) \pm g_{k,\alpha}(Y)$

When $k > 0$, we claim that $g_{k,\alpha}(X) \pm g_{k,\alpha}(Y)$ have a total of $d(2k)$ irreducible factors over $\mathbb{Z}$, just as when $\alpha = 0$. We illustrate this below for $k = 6$:

$$g_{6,\alpha}(X) = X^6 - 6\alpha X^4 + 9\alpha^2X^2 - 2\alpha^3,$$

$$g_{6,\alpha}(X) - g_{6,\alpha}(Y) = (X - Y)(X + Y)$$

$$(X^2 + XY + Y^2 - 3\alpha)(X^2 - XY + Y^2 - 3\alpha),$$

$$g_{6,\alpha}(X) + g_{6,\alpha}(Y) = (X^2 + Y^2 - 4\alpha)(X^4 - X^2Y^2 + Y^4 - 2\alpha(X^2 + Y^2) + \alpha^2).$$

Theorem 5.2.1 *Let k be a positive integer and fix $\alpha \in \mathbb{Z}$. Then the polynomial $g_{k,\alpha}(X) - g_{k,\alpha}(Y)$ has exactly $d(k)$ irreducible factors over $\mathbb{Z}$.*

Proof. The case $\alpha = 0$ is covered by the theory of cyclotomic polynomials. When $\alpha \neq 0$, we show that there are at least $d(k)$ irreducible factors and at most $d(k)$ such factors. The upper bound is clear (and applies to any polynomial of degree k, not just $g_{k,\alpha}$), since the highest total degree terms of $g_{k,\alpha}(X) - g_{k,\alpha}(Y)$ are $X^k - Y^k$, which has only $d(k)$ irreducible factors over $\mathbb{Z}$.

For existence, recall that

$$X^k - Y^k = \prod_{d|k} \Phi_d(X, Y)$$

where the cyclotomic polynomial Φ_d is homogeneous of degree $\phi(d)$ and satisfies $\Phi_d(X, Y) = \pm\Phi_d(Y, X)$ (the minus sign is needed only if $d = 1$). Hence

$$\begin{aligned} g_{k,\alpha}(U + \alpha/U) - g_{k,\alpha}(V + \alpha/V) &= U^k + \frac{\alpha^k}{U^k} - V^k - \frac{\alpha^k}{V^k} \\ &= (U^k - V^k)\,(1 - \alpha^k/U^kV^k) \\ &= \prod_{d|k} \Phi_d(U, V)\,\Phi_d(1, \alpha/UV). \end{aligned}$$

We claim that each of the $d(k)$ factors on the right is a non-constant polynomial function of $U + \alpha/U$ and $V + \alpha/V$ over $\mathbb{Q}$. This will show that $g_{k,\alpha}(X) - g_{k,\alpha}(Y)$ has at least $d(k)$ irreducible factors.

To prove this claim, fix $d|k$ and define

$$g(U, V) = \Phi_d(U, V)\,\Phi_d(1, \alpha/UV).$$

By homogeneity and the symmetry of $\Phi_d(X, Y)$ in X and Y,

$$\begin{aligned} g(\alpha/U, V) &= \Phi_d(\alpha/U, V)\,\Phi_d(1, U/V) \\ &= \Big(V^{\phi(d)}\,\Phi_d(\alpha/UV, 1)\Big)\,\Big(V^{-\phi(d)}\,\Phi_d(V, U)\Big) \\ &= \Big(\pm\Phi_d(1, \alpha/UV)\Big)\,\Big(\pm\Phi_d(U, V)\Big) \\ &= g(U, V). \end{aligned}$$

Therefore g is symmetric in U and α/U. Similarly

$$\begin{aligned} g(U, \alpha/V) &= \Phi_d(U, \alpha/V)\,\Phi_d(1, V/U) \\ &= \Big(U^{\phi(d)}\,\Phi_d(1, \alpha/UV)\Big)\,\Big(U^{-\phi(d)}\,\Phi_d(U, V)\Big) \\ &= g(U, V), \end{aligned}$$

so g is symmetric in V and α/V. Since g is (by definition) a polynomial function of U, V, $1/U$, and $1/V$ over $\mathbb{Z}$, g must be a polynomial in the symmetric polynomials $U + \alpha/U$, $U \cdot (\alpha/U) = \alpha$, $V + \alpha/V$, and $V \cdot (\alpha/V) = \alpha$ over $\mathbb{Q}$. Moreover, g cannot be constant because $\Phi_d(U, V)$ is homogeneous in U and V but $\Phi_d(1, \alpha/UV)$ is not homogeneous for $\alpha \neq 0$. ■

Corollary 5.2.2 *If k is a positive integer and $\alpha \in \mathbb{Z}$, then $g_{k,\alpha}(X) + g_{k,\alpha}(Y)$ has exactly $d(2k) - d(k)$ irreducible factors over $\mathbb{Z}$.*

Proof. From

$$
\begin{aligned}
g_{2k,\alpha}(X + \alpha/X) &= X^{2k} + \alpha^{2k}/X^{2k} \\
&= (X^k + \alpha^k/X^k)^2 - 2\alpha^k \\
&= \Big(g_{k,\alpha}(X + \alpha/X)\Big)^2 - 2\alpha^k,
\end{aligned}
$$

it follows that $g_{2k,\alpha} = g_{k,\alpha}^2 - 2\alpha^k$. By Theorem 5.2.1,

$$
g_{k,\alpha}(X) + g_{k,\alpha}(Y) = \frac{g_{2k,\alpha}(X) - g_{2k,\alpha}(Y)}{g_{k,\alpha}(X) - g_{k,\alpha}(Y)}
$$

has exactly $d(2k) - d(k)$ irreducible factors. ■

5.3 Prime divisors of $g_{k,\alpha}(X) \pm g_{k,\alpha}(Y)$

According to [17, solution to exercise 4.6.2.38], if $u(X)$ is an irreducible polynomial over $\mathbb{Z}$, then the average number of linear factors of $u(X)$ modulo q tends to 1 as the prime $q \to \infty$. If $u(X, Y)$ is an irreducible factor of $P(X) \pm P(Y)$ over $\mathbb{Z}$, then $u(X, Y)$ remains irreducible over $\mathbb{Z}$ (as a polynomial in X) after almost all substitutions of an integer for Y, by Hilbert's Irreducibility Theorem [19, Chapter 9]. So for almost all fixed $Y \in \mathbb{Z}$, the average number of linear factors of $u(X, Y)$ modulo q tends to 1 as $q \to \infty$. If $P(X) = g_{k,\alpha}(X)$ where α is a fixed integer (allowing the case where $\alpha = 0$ and $P(X) = X^k$), then, for almost all $Y \in \mathbb{Z}$, the average total number of linear factors of $P(X) \pm P(Y)$ modulo q tends to $d(2k)$ as $q \to \infty$, by Theorem 5.2.1 and Corollary 5.2.2. When X and Y are independently and randomly selected integers modulo a large q, then a crude estimate for the probability that $P(X) \equiv \pm P(Y) \pmod{q}$ is $d(2k)/q$.

For fixed q, the actual probability depends on the residue class of q modulo $2k$. Theorem 5.3.2 asserts that this probability is $\gcd(2k, q-1)/q + O(q^{-2})$ when $P(X) = X^k$, and is $\big(\gcd(2k, q-1) + \gcd(2k, q+1)\big)/2q + O(q^{-2})$ when $P(X) = g_{k,\alpha}(X)$ with $\alpha \not\equiv 0 \pmod{q}$. Lemma 5.3.1 says that the average value of $\gcd(2k, q-1)$ (or of $\gcd(2k, q+1) = \gcd(2k, -q-1)$) as q ranges over the $\phi(2k)$ residue classes relatively prime to $2k$ is $d(2k)$, confirming that the average probability is $d(2k)/q + O(q^{-2})$ if the residue class of q modulo $2k$ is randomly chosen.

Lemma 5.3.1 *If $k > 0$, then*

$$\sum_{\substack{0 \le i < k \\ \gcd(i,\, k)=1}} \gcd(i-1,\, k) = \phi(k) d(k).$$

Proof. Let $k = p_1^{e_1} p_2^{e_2} \ldots p_n^{e_n}$ be the prime factorization of k, with each $e_j \ge 1$.

Each i with $0 \le i < k$ is uniquely identified by its remainders $r_j \bmod p_j^{e_j}$ for $j = 1, 2, \ldots, n$. We can compute $\gcd(i,\, k) = \prod_{j=1}^{n} \gcd(r_j,\, p_j^{e_j})$ for any such i, and we can compute $\gcd(i-1,\, k)$ similarly. The desired sum is

$$\sum_{\substack{0 \le i < k \\ \gcd(i,\, k)=1}} \gcd(i-1,\, k) = \sum_{\substack{r_1, r_2, \ldots, r_n \\ 0 \le r_j < p_j^{e_j} \ \forall j, \\ \gcd(r_j,\ p_j^{e_j})=1 \ \forall j}} \prod_{j=1}^{n} \gcd(r_j - 1,\, p_j^{e_j}) = \prod_{j=1}^{n} \sum_{\substack{0 \le r_j < p_j^{e_j} \\ \gcd(r_j,\ p_j)=1}} \gcd(r_j - 1,\, p_j^{e_j}).$$

This inner gcd is 1 for the $p_j^{e_j} - 2p_j^{e_j - 1}$ values of r_j incongruent to 0 or 1 modulo p_j, is p_j for $p_j^{e_j-1} - p_j^{e_j-2}$ values of r_j, etc. So the inner sum is

$$\sum_{\substack{0 \le r_j < p_j^{e_j} \\ \gcd(r_j, p_j)=1}} \gcd(r_j - 1,\, p_j^{e_j})$$

$$= 1 \cdot (p_j^{e_j} - 2p_j^{e_j-1}) + p_j \cdot (p_j^{e_j-1} - p_j^{e_j-2}) + p_j^2 \cdot (p_j^{e_j-2} - p_j^{c_j-3}) + \cdots + p_j^{e_j} \cdot 1$$

$$= p_j^{e_j} - 2p_j^{e_j-1} + (e_j - 1)(p_j^{e_j} - p_j^{e_j-1}) + p_j^{e_j}$$

$$= (p_j^{e_j} - p_j^{e_j-1})(2 + e_j - 1) = p_j^{e_j-1}(p_j - 1)(e_j + 1).$$

The claim follows from $\phi(k) = \prod_{j=1}^{n} p_j^{e_j}(p_j - 1)$ and $d(k) = \prod_{j=1}^{n}(e_j + 1)$. ■

Theorem 5.3.2 *Let q be a prime not dividing k and $\alpha \in \mathbb{Z}$. As X and Y range over the interval $[0,\, q-1]$, the number of cases where $g_{k,\alpha}(X) \equiv g_{k,\alpha}(Y) \pmod q$ is*

$$\begin{cases} 1 + (q-1)\gcd(k,\, q-1), & \text{if } q|\alpha, \\[2ex] q\dfrac{\gcd(k,\, q-1) + \gcd(k,\, q+1)}{2} + O(1), & \text{if } q\nmid\alpha. \end{cases}$$

The $O(1)$ term may depend on k but not q.

Lemma 5.3.3 *Let G be a finite cyclic group. If $g_0 \in G$ and $k > 0$, then the equation $x^k = g_0$ has either exactly $\gcd(k,\, |G|)$ solutions with $x \in G$, or no such solution.*

PROOF. We may assume that G is the additive group of integers modulo $|G|$, since the assertion is invariant under group isomorphism. Then $x^k = g_0$ translates into $kx \equiv g_0 \pmod{|G|}$. This congruence has exactly $\gcd(k, |G|)$ solutions when $\gcd(k, |G|)$ divides g_0, and no solution otherwise. ■

PROOF OF THEOREM 5.3.2. We may regard α, X, and Y as elements of $\mathrm{GF}(q)$.

If $\alpha = 0$, then $g_{k,\alpha}(X) = g_{k,\alpha}(Y)$ simplifies to $X^k = Y^k$. For $Y = 0$, the only solution is $X = 0$. For each $Y \neq 0$, the equation has a known solution $X = Y$ and must have exactly $\gcd(k, q-1)$ solutions $X \in \mathrm{GF}(q)^*$ by Lemma 5.3.3.

Suppose instead that $\alpha \neq 0$. After possibly adjusting the $O(1)$, we may assume that q is odd. Given $z \in \mathrm{GF}(q)$, we count the solutions of $g_{k,\alpha}(X) = z$ with $X \in \mathrm{GF}(q)$. We claim that this equation has:

- At most k solutions when $z^2 - 4\alpha^k = 0$.
- Either no solution or $\gcd(k, q-1)$ solutions when $z^2 - 4\alpha^k$ is a quadratic residue. For any such solution, $X^2 - 4$ is a quadratic residue. (Throughout this proof, interpret "quadratic residue" as "quadratic residue modulo q" and likewise for quadratic non-residue.)
- Either no solution or $\gcd(k, q+1)$ solutions when $z^2 - 4\alpha^k$ is a quadratic non-residue. For any such solution, $X^2 - 4$ is a quadratic non-residue.

Suppose that we have proved this claim. There are at most $2k$ values of X for which $\left(g_{k,\alpha}(X)\right)^2 - 4\alpha^k = 0$. For any such X, the polynomial equation $g_{k,\alpha}(Y) = g_{k,\alpha}(X)$ has at most k solutions Y, so there are at most $2k^2$ total solutions (X, Y) of this type. When we exclude these $2k$ values of X, there remain $q/2 + O(1)$ values of X for which $X^2 - 4\alpha$ is a quadratic residue and another $q/2 + O(1)$ values of X for which $X^2 - 4$ is a quadratic non-residue. For each X in the first category, there are $\gcd(k, q-1)$ values of Y satisfying $g_{k,\alpha}(Y) = g_{k,\alpha}(X)$ according to the claim. For the second category, this count is $\gcd(k, q+1)$. Hence there are

$$(q/2 + O(1))\gcd(k, q-1) + (q/2 + O(1))\gcd(k, q+1) + O(1)$$

total solutions of $g_{k,\alpha}(X) = g_{k,\alpha}(Y)$ when $\alpha \neq 0$, as asserted by Theorem 5.3.2.

It remains to prove the claim. The case $z^2 - 4\alpha^k = 0$ is easy, since the polynomial equation $g_{k,\alpha}(X) = z$ of degree k in X can have at most k roots for X.

We therefore concentrate on the case where $z^2 - 4\alpha^k \neq 0$. Suppose that $z = g_{k,\alpha}(X)$ with $X \in \mathrm{GF}(q)$. Write $X = U + \alpha/U$ with $U \in \mathrm{GF}(q^2)$. Then $z = U^k + \alpha^k/U^k$, and the assumption $z^2 \neq 4\alpha^k$ becomes $U^k \neq \alpha^k/U^k$. In particular, $U \neq \alpha/U$.

Under these conditions, we claim that the following are equivalent:

(i) $X^2 - 4\alpha$ is a quadratic residue;

(ii) $U \in \mathrm{GF}(q)$;

(iii) $U - \alpha/U \in \mathrm{GF}(q)$;

(iv) $U^k - \alpha^k/U^k \in \mathrm{GF}(q)$;

(v) $z^2 - 4\alpha^k$ is a quadratic residue.

The equivalence of (i) and (iii) is evident since $X^2 - 4\alpha = (U - \alpha/U)^2$ and the latter is nonzero. Similarly for (iv) and (v). The equivalence of (ii) and (iii) is also immediate, since $X = U + \alpha/U \in \mathrm{GF}(q)$ and q is assumed odd. For (iii) and (iv), we observe that the quotient $(U^k - \alpha^k/U^k)/(U - \alpha/U)$ is in $\mathrm{GF}(q)$. This can be shown directly using symmetric functions, since the quotient is symmetric in U and α/U. Or we can proceed by induction on k, using the identity

$$\frac{U^k - \alpha^k/U^k}{U - \alpha/U} = (U + \alpha/U)\frac{U^{k-1} - \alpha^{k-1}/U^{k-1}}{U - \alpha/U} - \alpha\frac{U^{k-2} - \alpha^{k-2}/U^{k-2}}{U - \alpha/U}.$$

If $z^2 - 4\alpha^k$ is a quadratic residue, then $U \in \mathrm{GF}(q)$ must satisfy

$$U^k = \frac{z \pm \sqrt{z^2 - 4\alpha^k}}{2}. \tag{5.3.4}$$

For each choice of $\pm$, Lemma 5.3.3 gives us either no solution $U \in \mathrm{GF}(q)^*$ or $\gcd(k,\, q-1)$ such solutions. If there are any solutions for one selection of $\pm$, then there are also solutions for the other choice, since (5.3.4) is equivalent to

$$(\alpha/U)^k = \frac{z \mp \sqrt{z^2 - 4\alpha^k}}{2}.$$

This leads to $2\gcd(k,\, q-1)$ solutions for U and to $\gcd(k,\, q-1)$ solutions for X when $z^2 - 4\alpha^k$ is a quadratic residue, as claimed.

If instead $z^2 - 4\alpha^k$ is a quadratic non-residue, then $U \in \mathrm{GF}(q^2) \setminus \mathrm{GF}(q)$ (set difference). Fix $u_0 \in \mathrm{GF}(q^2)$ such that $u_0^{q+1} = \alpha$; this is possible since $\alpha^{q-1} = 1$ and the polynomial $u^{q^2-1} - 1$ splits completely over $\mathrm{GF}(q^2)$. Since the conjugate of U is U^q as well as α/U, we require $U^q = \alpha/Q$ so $U^{q+1} = \alpha = u_0^{q+1}$. Hence U/u_0 is a $(q+1)$–st root of unity in $\mathrm{GF}(q^2)$. The equation $z = U^k + \alpha^k/U^k$ implies that

$$\left(\frac{U}{u_0}\right)^k = \frac{z \pm \sqrt{z^2 - 4\alpha^k}}{2u_0^k}.$$

For each choice of sign, this has either $\gcd(k,\, q+1)$ solutions U/u_0 in the group of $(q+1)$–st roots of unity or no such solution, by Lemma 5.3.3. As above, there are either zero or $2\gcd(k,\, q+1)$ total solutions, leading to $\gcd(k,\, q+1)$ different values of $X = U + \alpha/U$ satisfying $g_{k,\alpha}(X) = z$. The equation $(U/u_0)^{q+1} = 1$ implies that $U^q = \alpha/U$, so any such $X = U + \alpha/U = U + U^q \in \mathrm{GF}(q)$. ■

If we select T independent random pairs $\{(X_t,\, Y_t)\}_{t=1}^{T}$, and p_q is the probability that $P(X) \equiv \pm P(Y) \pmod q$ for randomly chosen X and Y, then our estimated probability of a match $P(X_t) \equiv \pm P(Y_t)$ for some t becomes $1 - (1 - p_q)^T$. When $p_q = o(1)$, we can approximate this probability by $1 - e^{-Tp_q}$. By Theorem 5.3.2, this p_q depends on the residue class of q modulo $2k$. When the residue class of q is unknown, we should average this success probability over all possible residue classes. The result is

(5.3.5)

$$\mathrm{Pr}_{\mathrm{match}} = \begin{cases} 1 - \dfrac{1}{\phi(2k)} \displaystyle\sum_{q_0} \exp\left(-\dfrac{T}{q}\gcd(2k,\, q_0 - 1)\right), & \text{if } \alpha = 0; \\ 1 - \dfrac{1}{\phi(2k)} \displaystyle\sum_{q_0} \exp\left(-\dfrac{T}{q}\,\dfrac{\gcd(2k,\, q_0 - 1) + \gcd(2k,\, q_0 + 1)}{2}\right), & \text{if } \alpha \neq 0, \end{cases}$$

where q_0 runs over the residue classes modulo $2k$ which are relatively prime to $2k$.

If we expand (5.3.5) as a Laurent series in q, then the leading term is $d(2k)T/q$ for any choice of α, by Lemma 5.3.1. The averaged success probability (5.3.5) is never larger for $\alpha = 0$ than for fixed nonzero α, because for any pair of residues q_0 and $-q_0$,

$$\begin{aligned} &\exp\left(-\frac{T}{q}\gcd(2k,\, q_0 - 1)\right) + \exp\left(-\frac{T}{q}\gcd(2k,\, -q_0 - 1)\right) \\ &\geq 2\exp\left(-\frac{T}{q}\,\frac{\gcd(2k,\, q_0 - 1) + \gcd(2k,\, -q_0 - 1)}{2}\right) \\ &= \exp\left(-\frac{T}{q}\,\frac{\gcd(2k,\, q_0 - 1) + \gcd(2k,\, q_0 + 1)}{2}\right) \\ &\quad + \exp\left(-\frac{T}{q}\,\frac{\gcd(2k,\, -q_0 - 1) + \gcd(2k,\, -q_0 + 1)}{2}\right), \end{aligned}$$

by the arithmetic-geometric mean inequality.

The difference in these success probabilities can become significant for large k. Table 5.3.1 gives the number of trials T needed for the probability (5.3.5) to reach 0.05, 0.20, 0.50, 0.80, and 0.95, for selected k. The values of k selected are those for which $d(2k)$ exceeds all earlier values.

k	$d(2k)$	Polynomial	5%	20%	50%	80%	95%
1	2	$X = g_{1,1}(X)$	$.0256q$	$.1116q$	$.3466q$	$.8047q$	$1.4979q$
2	3	X^2	$.0171q$	$.0753q$	$.2406q$	$.5917q$	$1.1951q$
		$g_{2,1}(X)$	$.0171q$	$.0743q$	$.2310q$	$.5365q$	$.9986q$
3	4	X^3	$.0129q$	$.0574q$	$.1911q$	$.5175q$	$1.1562q$
		$g_{3,1}(X)$	$.0128q$	$.0558q$	$.1733q$	$.4024q$	$.7489q$
6	6	X^6	$.0086q$	$.0389q$	$.1338q$	$.3797q$	$.8940q$
		$g_{6,1}(X)$	$.0086q$	$.0373q$	$.1167q$	$.2744q$	$.5210q$
12	8	X^{12}	$.0065q$	$.0303q$	$.1118q$	$.3424q$	$.8630q$
		$g_{12,1}(X)$	$.0064q$	$.0283q$	$.0909q$	$.2235q$	$.4449q$
18	9	X^{18}	$.0059q$	$.0281q$	$.1123q$	$.3640q$	$.8906q$
		$g_{18,1}(X)$	$.0057q$	$.0256q$	$.0851q$	$.2198q$	$.4522q$
24	10	X^{24}	$.0053q$	$.0258q$	$.1039q$	$.3365q$	$.8624q$
		$g_{24,1}(X)$	$.0052q$	$.0233q$	$.0787q$	$.2062q$	$.4272q$
30	12	X^{30}	$.0044q$	$.0218q$	$.0902q$	$.2983q$	$.7756q$
		$g_{30,1}(X)$	$.0043q$	$.0196q$	$.0677q$	$.1857q$	$.4047q$
60	16	X^{60}	$.0034q$	$.0173q$	$.0751q$	$.2663q$	$.7405q$
		$g_{60,1}(X)$	$.0033q$	$.0151q$	$.0537q$	$.1507q$	$.3406q$
90	18	X^{90}	$.0030q$	$.0163q$	$.0749q$	$.2805q$	$.7707q$
		$g_{90,1}(X)$	$.0029q$	$.0138q$	$.0511q$	$.1480q$	$.3441q$
120	20	X^{120}	$.0028q$	$.0150q$	$.0694q$	$.2596q$	$.7394q$
		$g_{120,1}(X)$	$.0026q$	$.0126q$	$.0470q$	$.1378q$	$.3241q$

Table 5.3.1: Trials needed to reach confidence level

5.4 Comparative computational costs

If minimizing the estimated number of trials were the only concern, then we would prefer $m_i = g_{k,\alpha}(M_i)$ and $n_j = g_{k,\alpha}(N_j)$ with $\alpha \neq 0$. However, other desired properties given early in this chapter impose additional requirements, such as ease of computation.

Suppose that we select $\{M_i\}$ and $\{N_j\}$ randomly, subject to $0 \leq M_i, N_j < B$ where the bound B is pre-selected. Since $g_{k,\alpha}$ is monic of degree k, the m_i (and n_j) are bounded approximately by B^k. Any individual $m_i \cdot Q$ or $n_j \cdot Q$ can be computed with $O(\log B^k)$ group operations, using the same algorithm [31] as used during Step 1, for a total cost of $O((d_1 + d_2)k \log B)$ group operations. But there is considerable redundancy in this computation if we repeat it for every m_i and n_j. We can do better by viewing the m_i, $n_j \in [0, B^k]$ as integers in a radix R. Approximately $R \log_R B^k$ group operations suffice to build a table of $(rR^j) \cdot Q$ subject to $1 \le r < R$ and $rR^j \le B^k$. Each $m_i \cdot Q$ (or $n_j \cdot Q$) can then be computed with another $\log_R m_i < \log_R B^k$ group operations (Weierstrass coordinates), for a total cost of

$$(R + d_1 + d_2) \log_R B^k$$

group operations. This cost is minimized when $R \ln R = R + d_1 + d_2$. Using $R \approx (d_1 + d_2)/\ln(d_1 + d_2)$ reduces the total cost to about

$$\frac{d_1 + d_2}{\ln(d_1 + d_2)} \ln B^k$$

Weierstrass group operations, and the average cost per $m_i \cdot Q$ or $n_j \cdot Q$ to $\dfrac{\ln B^k}{\ln(d_1 + d_2)}$ such operations, for arbitrarily chosen m_i, $n_j \in [0, B^k]$.

Since we require $B \gg d_1 + d_2$ to ensure distinctness of the $\{m_i\}$ and $\{n_j\}$, the average cost of this algorithm exceeds k group operations. Section 5.9 shows how to compute these $(m_i \cdot Q)$ with an overhead of k Weierstrass group operations apiece (plus initialization costs) if the $\{M_i\}$ form an arithmetic progression, regardless of α; likewise for the $\{N_j\}$. If we restrict $\alpha = 0$ so that $P(X) = X^k$, and if $4|k$, then we can reduce the overhead to $k/2$ homogeneous group operations per $(m_i \cdot Q)_x$ (or $(n_j \cdot Q)_x$) by using a different sequence $\{M_i\}$, as seen below.

Lemma 5.4.1 *If q is an odd prime, then 16 is an 8–th power modulo q.*

Proof. If 2 is a quadratic residue modulo q, then $16 = \left(\sqrt{2}\right)^8$.

If -2 is a quadratic residue modulo q, then $16 = \left(\sqrt{-2}\right)^8$.

If -1 is a quadratic residue modulo q, then $16 = \left(1 + \sqrt{-1}\right)^8$.

Since at least one of these is a quadratic residue, the proof is complete. ∎

Remark 5.4.2 *We subsequently refer to this root as $\sqrt[8]{16}$, and treat it as an integer in the analysis. Algorithms use only powers of $\left(\sqrt[8]{16}\right)^4 = \pm 4$. Since $(4Q)_x = (-4Q)_x$, the sign ambiguity will not matter.*

Corollary 5.4.3 *Let k be a positive integer divisible by 4 and let q be an odd prime. Then at least one of $\pm 2^{k/2}$ is a k–th power modulo q.*

Proof. If $\alpha \equiv \sqrt[8]{16} \pmod{q}$, then $\alpha^{2k} = (\alpha^8)^{k/4} \equiv 16^{k/4} = 2^k \pmod{q}$. Consequently $\alpha^k \equiv \pm 2^{k/2} \pmod{q}$, and one of these is a k–th power modulo q. ∎

If our k is divisible by 4, then we can multiply any previously created k–th power or negative thereof by $2^{k/2}$ to create another such power. From a point P, we can construct $(2^{k/2} \cdot P)_x$ using $k/2$ doubling operations in the group. If we are allowed 100 multiplications modulo N per point, then we require $5(k/2) \leq 100$, or $k \leq 40$, since each application of the homogeneous doubling rule (2.3.5) uses five such multiplications.

This trick should be employed on only one of the sequences $\{m_i\}$ and $\{n_j\}$. If it is used on both sequences, then there are many cases where $m_{i_2} = 2^{k/2} m_{i_1}$ and $n_{j_2} = 2^{k/2} n_{j_1}$, violating (DP6). Since we are assuming that $d_2 \gg d_1$, it is probably cheaper to employ this trick during generation of the $\{n_j\}$ than during the $\{m_i\}$.

5.5 Using powers of $2^{k/2}$ and 3^k

A straightforward scheme based on this idea lets

$$m_i = 3^{ik} \qquad (0 \leq i < d_1);$$

$$n_j = 3^{(d_1/2)k} \cdot 2^{(j+1)k/2} \quad (0 \leq j < d_2)$$

where $4|k$. One can compute the x-coordinates $(m_i \cdot Q)_x$ using $(d_1 - 1)k$ successive cubings, and hence $2(d_1 - 1)k$ group operations (which can be done using homogeneous coordinates). Another $d_2 k/2$ doubling steps suffice to compute the $(n_j \cdot Q)_x$, starting from the known $(3^{(d_1/2)k} \cdot Q)_x$. If $d_2 \gg d_1$, then the average cost per x-coordinate is slightly above $k/2$ group operations.

Property (DP5) is achieved if Q has odd order, since a match $n_i \equiv \pm n_j$ with $0 \leq i < j < d_2$ implies that

$$3^{(d_1/2)k} \cdot 2^{(i+1)k/2} \equiv 3^{(d_1/2)k} \cdot 2^{(j+1)k/2},$$

$$3^{(d_1/2)k} \equiv 3^{(d_1/2)k} \cdot 2^{(j-i)k/2},$$

$$m_{d_1/2} \equiv n_{j-i-1},$$

ensuring that a match exists in (5.0.1).

This scheme also satisfies (DP1), since any prime q with

$$2 < q - 1 < d_1 d_2 \gcd(q-1, 2k)/2$$

divides some $m_{i_1} \pm m_{i_2}$ or some $m_i \pm n_j$. To prove this, consider $\{3^{2ik} \cdot 2^{jk}\}$ for $0 \leq i < d_1/2$ and $0 \leq j < d_2$. All of these are nonzero $(2k)$–th powers modulo q

since $4|k$. By the Pigeon-hole principle, there must be a duplicate amongst these $d_1 d_2/2$ values. This leads to a congruence $3^{2ik} \equiv 2^{jk}$ where $|i| < d_1/2$ and $0 \le j < d_2$, with i, j not both zero. If $j = 0$, then $m_{|i|} \equiv \pm m_0$. If instead $j \ne 0$, then $n_{j-1} \equiv \pm m_{d_1/2+i}$.

This scheme fails to be parallel as in (DP4), but such is not important when implementing ECM on a sequential architecture. This scheme also has many cases where one $3^{ik} \pm 2^{jk/2}$ divides multiple $m_{i_1} \pm n_{j_1}$ (some with $\gcd(i_1 - d_1/2,\ j_1+1) > 1$), in possible violation of (DP6).

5.6 Achieving parallelism

The last scheme failed to achieve parallelism, in part because each point was multiplied by $2^{k/2}$ to get the next point. When using k–th powers (i.e. $\alpha = 0$), it would be desirable to instead multiply several points by $2^{k/2}$ concurrently.

Suppose that our parallelism requirements dictate multiplying ρ different points by $2^{k/2}$ at once (ρ might be the number of processors available). If we have selected the n_j for $0 \le j < \rho$ by some alternate means, then we can let $n_j = 2^{k/2} \cdot n_{j-\rho}$ for $\rho \le j < d_2$. Assuming that $\rho | d_1 | d_2$, our algorithm can resemble:

- Choose k divisible by 4, with $d(2k)$ as large as feasible. Construct $(m_i \cdot Q)_x$ for $0 \le i < d_1$, with each $m_i = M_i^k$ for some $\{M_i\}$. Use these x-coordinates for roots of $F(X)$ in Section 4.3. Initialize $G(X)$.
- Construct $(n_j \cdot Q)_x$ for $0 \le j < \rho$, with each $n_j = N_j^k$ for some $\{N_j\}$.
- Perform the next two steps for $\ell = 1, \cdots, d_2/\rho$, in this order.
  - If $\ell > 1$, then set $n_j = 2^{k/2} \cdot n_{j-\rho}$ for $(\ell - 1)\rho \le j < \ell\rho$. Each corresponding $(n_j \cdot Q)_x$ can be computed from $(n_{j-\rho} \cdot Q)_x$ using $k/2$ doubling steps; values for different j can be computed in parallel.
  - If $d_1 | \ell\rho$ (i.e. when d_1 new values of $(n_j \cdot Q)_x$ have been found), then form another H-polynomial and reduce $G(X)\,H(X)$ modulo $F(X)$, as in Figure 4.3.1.

For the rest of this section, we assume that $\rho = d_1$ and that $n_j = 2^{k/2} \cdot m_j$ for $0 \le j < d_1$. Then property (DP5) is automatically satisfied if our point Q has odd order (which can be ensured by doing enough doublings during Step 1). More precisely, suppose that some $n_i \equiv \pm n_j$ where $0 \le i < j < d_2$. If $i < d_1$ and $j < d_1$, then $2^{k/2} \cdot m_i \equiv \pm 2^{k/2} \cdot m_j \pmod{q}$, implying that $m_i \equiv \pm m_j$. If $i < d_1$ but $j \ge d_1$, then $m_i \equiv \pm n_{j-d_1}$. If $i \ge d_1$ and $j \ge d_1$, then $n_{i-d_1} \equiv \pm n_{j-d_1}$ and we proceed by induction on j.

It remains to select the $\{m_i\}$. We should satisfy the cost requirement (DP3), although we may be able to afford more overhead per m_i than per n_j since we will not be needing as many.

There should be few if any multiplicative relations amongst the $\{m_i\}$. If, for example, $m_{i_1}/m_{i_2} = m_{i_3}/m_{i_4}$ then $2^{\ell(k/2)}m_{i_1} - m_{i_2}$ shares many factors with $2^{\ell(k/2)}m_{i_3} - m_{i_4}$ for each ℓ, violating (DP6). This precludes, for example, defining $M_i = 3^i$ and $m_i = M_i^k = 3^{ik}$ for $0 \le i < d_1$, even though such values can be successively computed with $2k$ (non-parallel) group operations apiece. [This duplication was not a problem in Section 5.5, where all powers of $2^{k/2}$ were multiplied by the same m_{i_1}, namely by $m_{d_1/2}$.]

If we use random values for $\{M_i\}$, then the only apparent violations of (DP6) occur for those constructed from the same M_i, since for example $2^{\ell(k/2)} - 1$ divides all $n_{i+(\ell-1)d_1} - m_i$. That is, the only apparent redundant $m_i \pm n_j$ occur when $i \equiv j \pmod{d_1}$. If $d_1 \ge 128$, then this redundancy occurs for under 1% of the $m_i \pm n_j$, an allowance which appears negligible.

But large numbers of random k–th powers may not be easy to compute efficiently, as desired in (DP3), despite the method presented early in Section 5.4. One computationally feasible proposal uses arithmetic progressions for $\{M_i\}$. If a and b are fixed, then the values of $m_i = (ai+b)^k$ are successive values of a polynomial of degree k, and successive $m_i \cdot Q$ can be computed with k group operations apiece, after suitable initialization; see Section 5.9. That algorithm allows parallelism (on up to k processors), but uses Weierstrass rather than homogeneous coordinates and hence requires modular inversions. Its initialization cost depends primarily on the magnitude of the largest $(ai+b)^k$ for $0 \le i \le k$.

However the use of arithmetic progressions for these M_i may lead to decreased overall effectiveness. Property (DP1) requires that most or all small primes q divide some $m_{i_1} \pm m_{i_2}$ subject to (5.0.2) or some $m_i \pm n_j$ subject to (5.0.1). The primes least likely to have this property are those where $\gcd(q-1, 2k) = 2$, since any such q divides an $M_i^k \pm M_j^k$ only if it divides $M_i \pm M_j$. Suppose that we have used arithmetic progressions for $\{M_i\}$, say $M_i = ai + b$. Then $M_i + M_j = a(i+j) + 2b$ and $M_i - M_j = a(i-j)$. There are only $2d_1 - 3$ distinct sums $a(i+j)+2b$ for $0 \le i, j < d_1$ and $i \ne j$, compared to the desired $\binom{d_1}{2} = d_1(d_1-1)/2$ such sums. There are also very few distinct differences $a(i-j)$. For small odd ℓ the problem persists, since

$$n_{i+\ell \cdot d_1} \pm n_j = 2^{(\ell+1)k/2}(ai+b)^k \pm (aj+b)^k$$

is divisible by

$$2^{(\ell+1)/2}(ai+b) \pm (aj+b) = a(2^{(\ell+1)/2}i \pm j) + b(2^{(\ell+1)/2} \pm 1),$$

and there are many duplicate sums and differences on the right. There are few duplications of this type when ℓ is even and the numerical value of $\left(\sqrt[8]{16}\right)^{\ell+1}$ is (presumably) large modulo most primes q, but many of the potential opportunities for a match modulo q are being wasted when ℓ is odd, violating (DP6) and possibly (DP1).

One work-around uses multiple arithmetic progressions for $\{M_i\}$. If we use d_1 such progressions, each of length 1, then the $\{M_i\}$ are essentially random. If we use eight or sixteen such progressions, then the above troubles occur only for $M_i \pm M_j$ where M_i and M_j come from the same progression, and hence for one eighth or one sixteenth of such pairs, which may be an acceptable tolerance.

5.7 Separate arithmetic progressions

If we abandon our convention that $n_j = m_j \cdot 2^{k/2}$ for $0 \le j < d_1$, but retain $n_j = n_{j-\rho} 2^{k/2}$ if $\rho \le j < d_2$, then we must select the early n_j as well as the m_i. Unless we restrict $\{m_i\}$, property (DP5) need no longer hold.

One approach uses one arithmetic progression for the $\{M_i\}$ and another for $\{N_j\}_{j=0}^{\rho-1}$. Subsequent values of N_j are $\sqrt[8]{16}$ times earlier values. That is,

$$M_i = a_1 i + b_1 \quad (0 \le i < d_1)$$

and

$$N_j = \begin{cases} a_2 j + b_2, & \text{if } 0 \le j < \rho, \\ (a_2(j-\rho) + b_2)\sqrt[8]{16}, & \text{if } \rho \le j < 2\rho, \\ 2(a_2(j-2\rho) + b_2), & \text{if } 2\rho \le j < 3\rho, \\ \ldots. \end{cases}$$

We can achieve (DP1) by choosing the progressions so that most small primes have the form $M_i \pm N_j$. For example, let $M_i = 6i+1$ for $0 \le i < d_1$ and $N_j = 6d_1(j+1)$ for $0 \le j < \rho$, with $N_j = N_{j-\rho}\sqrt[8]{16}$ for $\rho \le j < d_2$. This particular example has many duplicate n_j since for example

$$n_{2\rho} = n_0 \cdot \left(2^{k/2}\right)^2 = N_0^k \cdot 2^k = (2N_0)^k = (12d_1)^k = N_1^k = n_1.$$

Such overlap is rectified by letting $N_j = 6d_1(2j+1)$ take on only odd multiples of $6d_1$ for $0 \le j < \rho$; even multiples appear in later rows of Figure 5.7.1 (i.e. H_j for larger j) due to the multiplications by $\sqrt[8]{16}$. We should also shift the values in the first row, say to $M_i = 6(i+d_1)+1$, to increase the number of distinct ratios M_i/N_j with $0 \le i < d_1$ and $0 \le j < d_2$. For example, if $d_1 = \rho = 4$, then M_0 to M_3 are 25, 31, 37, and 43, whereas the early N_j appear in Figure 5.7.1.

$$\begin{array}{lcccc}
H_1 \text{ roots} & N_0 = 24 & N_1 = 72 & N_2 = 120 & N_3 = 168 \\
& \downarrow & \downarrow & \downarrow & \downarrow \\
H_2 \text{ roots} & N_4 = 24\sqrt[8]{16} & N_5 = 72\sqrt[8]{16} & N_6 = 120\sqrt[8]{16} & N_7 = 168\sqrt[8]{16} \\
& \downarrow & \downarrow & \downarrow & \downarrow \\
H_3 \text{ roots} & N_8 = 48 & N_9 = 144 & N_{10} = 240 & N_{11} = 336 \\
& \downarrow & \downarrow & \downarrow & \downarrow \\
H_4 \text{ roots} & N_{12} = 48\sqrt[8]{16} & N_{13} = 144\sqrt[8]{16} & N_{14} = 240\sqrt[8]{16} & N_{15} = 336\sqrt[8]{16} \\
& \downarrow & \downarrow & \downarrow & \downarrow \\
H_5 \text{ roots} & N_{16} = 96 & N_{17} = 288 & N_{18} = 480 & N_{19} = 672
\end{array}$$

Figure 5.7.1: Dependencies using two arithmetic progressions and doubling

5.8 Use of arithmetic progressions and Dickson polynomials

The trick of multiplying a point by a k–th power (or by $2^{k/2}$) to get another point when $\alpha = 0$ used the identity $(XY)^k = X^k Y^k$ so that

$$g_{k,0}(XY) \cdot Q = (XY)^k \cdot Q = Y^k \cdot \Big(X^k \cdot Q\Big) = Y^k \cdot \Big(g_{k,0}(X) \cdot Q\Big).$$

No such identity relates $g_{k,\alpha}(XY)$ to $g_{k,\alpha}(X)$ when $\alpha \neq 0$.

The data in Table 5.3.1 suggest that Dickson polynomials $g_{k,\alpha}(X)$ with $\alpha \neq 0$ perform better than X^k when selecting the m_i and n_j, if our sole objective is maximize the probability that a random prime q divides some $m_{i_1} \pm m_{i_2}$ satisfying (5.0.2) or some $m_i \pm n_j$ satisfying (5.0.1). Of the methods of generation considered in Section 5.4, with $m_i = g_{k,\alpha}(M_i)$ for some $\{M_i\}$ (resp. $n_j = g_{k,\alpha}(N_j)$ for some $\{N_j\}$), the use of random M_i requires over k group operations per m_i, while the use of arithmetic progressions for $\{M_i\}$ reduces this cost to just k such operations, after suitable initialization. Since theory suggests that all $\alpha \neq 0$ perform comparably, it is simplest to use $\alpha = 1$ (or $\alpha = -1$, to force nonnegative coefficients). We can let, for example,

$$M_i = 6i + 1 \qquad (0 \le i < d_1);$$

$$N_j = 6d_1(j + 1) \quad (0 \le j < d_2).$$

5.9 Evaluation of $\{(m_i \cdot Q)_x\}$ where m_i is a polynomial function

Let $P(X)$ be a polynomial in X of degree k. Some of the above schemes require evaluating $(P(i) \cdot Q)_x$ for several successive integers i. This can be done using a straightforward modification to the technique for evaluating a polynomial along an arithmetic progression [17, p. 469], at a cost of $O(k)$ Weierstrass group operations per evaluation, after suitable initialization. The computations can be arranged so that these $O(k)$ operations can be done in parallel.

For example, consider $P(i) \cdot Q$ for successive i where $P(X) = X^4$ and $k = 4$. Tabulate $P(0)$ to $P(k)$ and take finite differences as in the left of Figure 5.9.1.

0		1		16		81		256		625
	1		15		65		175		369	
		14		50		110		194		
			36		60		84			
				24		24				

Figure 5.9.1: Finite differences of polynomial function $P(X) = X^4$

Because $\deg(P) = 4$, the fourth row of finite differences is constant. The vector [24, 60, 110, 175, 256] on the first full upward diagonal of Figure 5.9.1 can be computed from the top row using $k(k+1)/2$ integer subtractions, and the results used to evaluate $[24Q,\ 60Q,\ 110Q,\ 175Q,\ 256Q]$ as in Section 5.4. Each subsequent upward diagonal, such as $[24Q,\ 84Q,\ 194Q,\ 369Q,\ 625Q]$, can be computed from the previous such diagonal with k group operations, by following the arrows in Figure 5.9.2. Once an entire diagonal vector is known, a value of $P(i) \cdot Q$ can be extracted from its last component.

$$\begin{array}{ccccccccc} 24Q & & 60Q & & 110Q & & 175Q & & 256Q \\ \downarrow & & \downarrow & & \downarrow & & \downarrow & & \downarrow \\ 24Q & \rightarrow & 84Q & \rightarrow & 194Q & \rightarrow & 369Q & \rightarrow & 625Q \end{array}$$

Figure 5.9.2: Dependencies when updating an upward diagonal

A problem with this scheme is that the components of the vector cannot be updated in parallel, because each component is dependent on the previous component, as evidenced by the horizontal arrows in Figure 5.9.2.

Fortunately, there is an easy remedy. If we use downward diagonals rather than upward diagonals in Figure 5.9.1, then we can proceed very similarly but all components can be updated in parallel. Figure 5.9.3 illustrates how the downward diagonal $[1Q,\ 15Q,\ 50Q,\ 60Q,\ 24Q]$ can be calculated from the previous downward diagonal $[0Q,\ 1Q,\ 14Q,\ 36Q,\ 24Q]$ using four parallel group operations.

$$\begin{array}{ccccccccc} 0Q & & 1Q & & 14Q & & 36Q & & 24Q \\ \downarrow & \swarrow & \downarrow & \swarrow & \downarrow & \swarrow & \downarrow & \swarrow & \downarrow \\ 1Q & & 15Q & & 50Q & & 60Q & & 24Q \end{array}$$

Figure 5.9.3: Dependencies when updating a downward diagonal

Whether we use upward or downward diagonals, this computation requires Weierstrass coordinates (2.0.2) rather than homogeneous coordinates (2.3.3), because $(Q_1 - Q_2)_x$ is usually not known when we need to compute $Q_1 + Q_2$ from two known multiples Q_1 and Q_2 of Q. Hence each group operation needs a modular division when computing the slope in (2.0.3). Downward diagonals are preferable to upward diagonals even on a sequential architecture, because the modular inversions in these divisions are independent. All but one inversion can be exchanged for three modular multiplications by repeatedly using the identities [29, p. 260]

$$1/x = y(1/xy) \qquad \text{and} \qquad 1/y = x(1/xy).$$

For an arbitrary polynomial $P(X)$ of degree k, suppose that we want to enumerate $\{P(i)\}$ for $i = 1, 2, \ldots$. Define

$$P_0(X) = P(X), \qquad P_j(X) = P_{j-1}(X+1) - P_{j-1}(X) \quad (1 \le j \le k).$$

Each P_j is a polynomial of degree $k - j$; in particular, P_k is constant. We evaluate $P(0)$ to $P(k)$ numerically, and use those to evaluate $P_j(i)$ for $0 \le j \le k$ and $0 \le i \le k-j$. Hence we can determine the $(k+1)$-vector $[P_0(0),\ P_1(0),\ \cdots,\ P_k(0)]$.

Each $P_j(0) \cdot Q$ can be computed as described for random multiples of Q early in Section 5.4. This gives us the vector $v(0)$ where

$$v(i) = [P_0(i) \cdot Q,\ P_1(i) \cdot Q,\ \cdots,\ P_k(i) \cdot Q].$$

Next, for $i = 1, 2, \ldots$, we can compute $v(i)$ from $v(i-1)$ using k parallel group operations. The first component of $v(i)$ is the desired $P_0(i) \cdot Q = P(i) \cdot Q$.

5.10 Implementation choices

It was decided to implement two schemes. Of the values of k appearing in Table 5.3.1, cost requirement (DP3) restricts us to $k \le 24$ when using $P(X) = X^k$ where $4|k$, and to $k \le 12$ when using $P(X) = g_{k,\alpha}(X)$ where $\alpha \ne 0$.

For $P(X) = X^k$ with $k = 24$, a variation of the scheme in Section 5.5 was used. The main problem was a lack of parallelism when using only powers of 2 and 3. If there are under ρ processors and $d_1 \gg \rho$, then we can let

$$\begin{aligned} m_{\rho i+j} &= 3^{ik} \cdot 5^{jk} \qquad && (0 \le \rho i + j < d_1 \text{ and } 0 \le j < \rho); \\ n_{\rho i+j} &= 2^{(i+1)k/2} \cdot 7^{jk} \qquad && (0 \le \rho i + j < d_2 \text{ and } 0 \le j < \rho). \end{aligned} \tag{5.10.1}$$

The values of $(5^{jk} \cdot Q)_x$ and $(7^{jk} \cdot Q)_x$ for $0 \le j < \rho$ can be computed sequentially, using $(\rho - 1)(3k + 4k) = 168(\rho - 1)$ (non-parallel) applications of (2.3.4) or (2.3.5). (If this is an unacceptable amount of sequential calculation, then one can use additional primes besides 5 and 7.) Once these have been built, k applications of (2.3.5) and k of (2.3.4) suffice to get each new $(m_{\rho i+j} \cdot Q)_x$ (cost: $5k + 6k = 264$ modular multiplications apiece). Only $k/2$ applications of (2.3.5) (cost $5k/2 = 60$ modular multiplications apiece) are needed per new $(n_{\rho i+j} \cdot Q)_x$. For $d_2 \gg d_1$, the average overall cost drops below the 100 multiplications allowed by (DP3). Figure 5.10.1 illustrates the dependency picture when building the $(m_{\rho i+j} \cdot Q)_x$. The ρ computations along any row (except the first) can proceed in parallel, with this process repeated until d_1 values are available.

This construction appears to fail property (DP1), by not ensuring that small primes divide some $m_{i_1} \pm m_{i_2}$ or $m_i \pm n_j$ (though they divide with high probability

$$
\begin{array}{ccccccc}
m_0 = 1 & \rightarrow & m_1 = 5^k & \rightarrow & m_2 = 5^{2k} & \cdots & m_{\rho-1} = 5^{(\rho-1)k} \\
\downarrow & & \downarrow & & \downarrow & & \downarrow \\
m_\rho = 3^k & & m_{\rho+1} = 3^k \cdot 5^k & & m_{\rho+2} = 3^k \cdot 5^{2k} & & m_{2\rho-1} = 3^k \cdot 5^{(\rho-1)k} \\
\downarrow & & \downarrow & & \downarrow & & \downarrow \\
m_{2\rho} = 3^{2k} & & m_{2\rho+1} = 3^{2k} \cdot 5^k & & m_{2\rho+2} = 3^{2k} \cdot 5^{2k} & & m_{3\rho-1} = 3^{2k} \cdot 5^{(\rho-1)k} \\
\vdots & & \vdots & & \vdots & & \vdots
\end{array}
$$

Figure 5.10.1: Dependencies when using geometric progression

by (5.3.5)). The only apparent algebraic divisibility relation amongst the $m_i \pm n_j$ where i and j satisfy (5.0.1) occurs when the exponents in $3^{i_1 k} \cdot 5^{j_1 k} \pm 2^{(i_2+1)k/2} \cdot 7^{j_2 k}$ satisfy $\gcd(i_1, j_1, i_2{+}1, j_2) > 1$; for four random integers, this event has probability $1 - 1/\zeta(4) = 1 - 90/\pi^4 < 0.08$, which appears acceptably small for (DP6). The scheme almost satisfies (DP5); if

$$2^{(i_1+1)k/2} \cdot 7^{j_1 k} \equiv \pm 2^{(i_2+1)k/2} \cdot 7^{j_2 k} \pmod{q},$$

then there is a congruence $2^{ik/2} \equiv 7^{jk} \pmod{q}$ with $0 \le j < \rho$ and $|i| < d_2/\rho$ and i, j not both zero. If $i < 0$, then $n_{\rho(-i-1)+j} \equiv \pm 1 = \pm m_0$. If we redefine $m_{d_1-1} = 7^{(\rho-1)k}$ instead of the value in (5.10.1), then we have a similar result with $i > 0$ since $m_{\rho(i-1)+(\rho-1-j)} \equiv \pm 7^{(\rho-1)k} = \pm m_{d_1-1}$ Hence we have a match in (5.0.1) except possibly when $i = 0$, meaning $7^{jk} \equiv 1 \pmod{q}$ for some j with $0 < j < \rho$. We can check these cases separately.

The other scheme implemented used $P(X) = g_{k,\alpha}(X)$ with $\alpha = 1$ and $k = 12$. The sequences were chosen by $m_i = g_{12,1}(M_i)$ and $n_j = g_{12,1}(N_j)$ where

$$M_i = 6i + 1 \qquad (0 \le i < d_1);$$

$$N_j = 6d_1(j+1) \quad (0 \le j < d_2).$$

We claim that this satisfies (DP1), by ensuring that all small primes q divide some $M_i \pm N_j$ and hence some $m_i \pm n_j$. If $\gcd(q, 6) = 1$ and $6d_1 < q < 6d_1 d_2$, then we can write $q = 6d_1(j+1) \pm (6i+1)$ where $0 \le i < d_1$ and $0 \le j < d_2$. For $q < 6d_1$ and $q \equiv 1 \pmod{6}$ we can represent $5q$ instead of q.

Selecting $N_j = 6d_1(j + d_2/4)$ rather than $N_j = 6d_1(j + 1)$ would enable one to extend the upper bound to $7.5d_1d_2 - 6d_1$, albeit at the cost of computing higher multiples of Q when initializing the algorithm in Section 5.9.

This scheme seems to be remarkably free of algebraic identities where one $P(X) \pm P(Y)$ divides another, and so seems to satisfy (DP6).

However, this scheme may fail (DP5), by having duplicate $\pm n_j \pmod q$ without ever satisfying (5.0.2) or (5.0.1).

Richard Crandall [14] has also used $P(X) = X^k$, but without the multiplies by $2^{k/2}$ allowed by Corollary 5.4.3. His preliminary estimates suggest $k = 60$ or $k = 72$ when $B_1 \approx 10^6$ and $k = 120$ or $k = 240$ when $B_1 \approx 10^7$. Crandall did not use an FFT, so his asymptotic cost per test of $m_i - n_j$ is much higher. The data in Chapter 7 suggest that $k = 12$ is sufficiently high when using $P(X) = X^k$.

CHAPTER 6

Selection of Curve

So far we have not specified which elliptic curve to use, except that it should have the form

$$By^2 = x^3 + Ax^2 + x \tag{6.0.1}$$

for some A and B with $\gcd((A^2-4)B, N) = 1$ ((6.0.1) is the affine equivalent of (2.3.3)). We also require a known initial point. Suyama proved that any curve of the form (6.0.1) has order divisible by 4 when reduced modulo any prime p, since at least one of $B(A+2)$, $B(A-2)$, A^2-4 is a quadratic residue modulo p [29, p. 262].

We show how to select curves of form (6.0.1) whose torsion group over $\mathbb{Q}$ has order 12 or 16, with a known rational non-torsion point (hence positive rank over $\mathbb{Q}$). Then we present numerical data comparing the actual exponents of 2 and 3 which divide the orders of these curves when they are reduced modulo a prime.

6.1 Torsion subgroup of order 12 and positive rank over $\mathbb{Q}$

Montgomery [29, pp. 262–263] showed how to select a curve with known initial point and with torsion subgroup $\mathbb{Z}/12\mathbb{Z}$ over $\mathbb{Q}$. If

$$A = \frac{-3a^4 - 6a^2 + 1}{4a^3}, \qquad B = \frac{(a^2-1)^2}{4a^3}, \qquad \text{where} \qquad a = \frac{t^2-1}{t^2+3}, \tag{6.1.1}$$

then (6.0.1) has the following rational torsion points:

$$\begin{aligned} &(6.1.2)\quad O & 6P &= \left(0,\ 0\right), \\ &P = \left(\frac{(1+t)^2}{(1-t)^2},\ t\frac{(1+t)^2}{(1-t)^2}\right), & 7P &= \left(\frac{(1-t)^2}{(1+t)^2},\ -t\frac{(1-t)^2}{(1+t)^2}\right), \end{aligned}$$

$$2P = \left(\frac{t^2+3}{t^2-1},\ \frac{t^2+3}{t^2-1}\right) = \left(\frac{1}{a},\ \frac{1}{a}\right), \qquad 8P = \left(\frac{t^2-1}{t^2+3},\ -\frac{t^2-1}{t^2+3}\right) = \left(a,\ -a\right),$$

$$3P = \left(1,\ 2\frac{t}{t^2+1}\right), \qquad 9P = \left(1,\ -2\frac{t}{t^2+1}\right),$$

$$4P = \left(\frac{t^2-1}{t^2+3},\ \frac{t^2-1}{t^2+3}\right) = \left(a,\ a\right), \qquad 10P = \left(\frac{t^2+3}{t^2-1},\ -\frac{t^2+3}{t^2-1}\right) = \left(\frac{1}{a},\ -\frac{1}{a}\right),$$

$$5P = \left(\frac{(1-t)^2}{(1+t)^2},\ t\frac{(1-t)^2}{(1+t)^2}\right), \qquad 11P = \left(\frac{(1+t)^2}{(1-t)^2},\ -t\frac{(1+t)^2}{(1-t)^2}\right).$$

The point

$$\left(x_0,\ y_0\right) = \left(\frac{3a^2+1}{4a},\ \frac{\sqrt{3a^2+1}}{4a}\right)$$

is on (6.0.1) if $3a^2+1 = 4(t^4+3)/(t^2+3)^2$ is a rational square. We can achieve this by letting $t^2 = (u^2-12)/4u$, where $u^3 - 12u$ is a rational square.

6.2 Torsion subgroup of order 16 and positive rank over $\mathbb{Q}$

Bremner [8] discovered that the curve

$$y^2 = x(x+4096)(x+50625)$$

has a torsion subgroup of order 16 and positive rank over $\mathbb{Q}$. Replacing Bremner's x by $120^2 x$ and his y by $120^3 y$ gives the curve

$$y^2 = x\left(x + \frac{64}{225}\right)\left(x + \frac{225}{64}\right) = x^3 + \frac{54721}{14400}x^2 + x, \tag{6.2.1}$$

which has the form (6.0.1). More generally, if $a^2 + b^2 = c^2$, then the curve

$$y^2 = x\left(x + \frac{a^2}{b^2}\right)\left(x + \frac{b^2}{a^2}\right) = x^3 + \frac{a^4+b^4}{a^2b^2}x^2 + x \tag{6.2.2}$$

has the following rational torsion points, where P has order 8 and Q has order 2:

$$
\begin{aligned}
&(6.2.3)\ O && Q=\left(-\frac{b^2}{a^2},\ 0\right),\\
P&=\left(\frac{a+b+c}{a+b-c},\ 2\frac{(a+b)c}{(a+b-c)^2}\right), & P+Q&=\left(\frac{a-b-c}{a-b+c},\ 2\frac{(a-b)c}{(a-b+c)^2}\right),\\
2P&=\left(1,\ \frac{c^2}{ab}\right), & 2P+Q&=\left(-1,\ \frac{a^2-b^2}{ab}\right),\\
3P&=\left(\frac{a+b-c}{a+b+c},\ 2\frac{(a+b)c}{(a+b+c)^2}\right), & 3P+Q&=\left(\frac{a-b+c}{a-b-c},\ 2\frac{(a-b)c}{(a-b-c)^2}\right),\\
4P&=\left(0,\ 0\right), & 4P+Q&=\left(-\frac{a^2}{b^2},\ 0\right),\\
5P&=\left(\frac{a+b-c}{a+b+c},\ -2\frac{(a+b)c}{(a+b+c)^2}\right), & 5P+Q&=\left(\frac{a-b+c}{a-b-c},\ -2\frac{(a-b)c}{(a-b-c)^2}\right),\\
6P&=\left(1,\ -\frac{c^2}{ab}\right), & 6P+Q&=\left(-1,\ -\frac{a^2-b^2}{ab}\right),\\
7P&=\left(\frac{a+b+c}{a+b-c},\ -2\frac{(a+b)c}{(a+b-c)^2}\right), & 7P+Q&=\left(\frac{a-b-c}{a-b+c},\ -2\frac{(a-b)c}{(a-b+c)^2}\right).
\end{aligned}
$$

These points are distinct for all but finitely many ratios $a:b:c$, and give a torsion subgroup of order 16. Mazur showed that this is the largest possible torsion subgroup for an elliptic curve over $\mathbb{Q}$ [38, p. 223].

Theorem 6.2.4 *Any elliptic curve with torsion group $\mathbb{Z}/8\mathbb{Z}\times\mathbb{Z}/2\mathbb{Z}$ over $\mathbb{Q}$ is equivalent to one of form (6.2.2) with $ab\neq 0$.*

Proof. Kubert [18, p. 217] gives the parameterization

$$Y^2+(1-c)XY-bY=X^3-bX^2, \tag{6.2.5}$$

where

$$b=(2d-1)(d-1),\quad c=\frac{(2d-1)(d-1)}{d},\qquad d=\frac{2\alpha(4\alpha+1)}{8\alpha^2-1}.$$

Here $\alpha\in\mathbb{Q}$ and

$$d(d-1)(2d-1)(8d^2-8d+1)\neq 0. \tag{6.2.6}$$

Completing the square in (6.2.5) gives

$$\begin{aligned}
&\left(Y + \frac{(1-c)X - b}{2}\right)^2 \\
&= X^3 + \frac{(c-1)^2 - 4b}{4}X^2 + \frac{b(c-1)}{2}X + \frac{b^2}{4} \\
&= (X + d - d^2)\left(X^2 - \frac{(2d-1)(4d^2-6d+1)}{4d^2}X - \frac{(d-1)(2d-1)^2}{4d}\right).
\end{aligned}$$

The linear change of variables

$$Y = (d-1)^3 y - (1-c)X/2 + b/2,$$

$$X = (d-1)^2 x + d^2 - d,$$

converts this to the form (6.0.1) with $B = 1$ and

$$A = \frac{8d^4 - 16d^3 + 16d^2 - 8d + 1}{4d^2(d-1)^2} = \left[\frac{(2d-1)^2}{2d(d-1)}\right]^2 - 2 = \left[\frac{(8\alpha^2 + 4\alpha + 1)^2}{(8\alpha^2 + 4\alpha)(4\alpha + 1)}\right]^2 - 2.$$

This has the form (6.2.2) with

$$a : b : c = 8\alpha^2 + 4\alpha : 4\alpha + 1 : 8\alpha^2 + 4\alpha + 1.$$

Restriction (6.2.6) becomes

$$\frac{2\alpha(2\alpha+1)(4\alpha+1)(8\alpha^2+4\alpha+1)(8\alpha^2+8\alpha+1)^2}{(8\alpha^2-1)^5} \neq 0;$$

this simplifies to $\alpha(2\alpha+1)(4\alpha+1) \neq 0$ and hence $ab \neq 0$ since α is rational. ■

Bremner's curve (6.2.1), with $a : b : c = 8 : 15 : 17$, has rational non-torsion points and hence positive rank. Sample x-coordinates are

$$x_0 = -\frac{3}{10}, \quad \frac{8}{15}, \quad \frac{1}{18}, \quad \frac{8}{25},$$

and their reciprocals. A search found rational non-torsion points on (6.2.2) for some other Pythagorean ratios $a : b : c$. These solutions were analyzed for patterns.

The solutions $x_0 = 8/25$ when $a : b : c = 8 : 15 : 17$, $x_0 = 4/45$ when $a : b : c = 9 : 40 : 41$, and $x_0 = 200/289$ when $a : b : c = 39 : 80 : 89$ all have the form

$$x_0 = (4c - 4a)/(5c - 4a) \quad \text{or} \quad x_0 = (4c - 4b)/(5c - 4b).$$

This point is a torsion point when $a : b : c = 9 : 40 : 41$ but not in the other two cases. Substituting $x = (4c - 4a)/(5c - 4a)$ and $b^2 = c^2 - a^2$ in (6.2.2) gives

$$y^2 = 4\frac{c^2(2c-a)^2(5c+a)(c-a)}{a^2(c+a)(5c-4a)^3}. \tag{6.2.7}$$

Since $(c + a)(c - a) = b^2$ is assumed to be a perfect square, the computed y^2 is a perfect square whenever $(5c + a)(5c - 4a)$ is a perfect square.

We can get infinitely many solutions to (6.2.7) by setting $a = 1 - t^2$, $b = 2t$, and $c = 1 + t^2$, where t is a rational number to be determined. We require that $(t^2 + 1/9)(t^2 + 3/2)$ be a rational square. Letting $u = t^2$, we want u and $(u + 1/9)(u + 3/2)$ to be squares. By selecting an arbitrary point $(u, v) = (u_0, v_0)$ on the elliptic curve

$$v^2 = u(u + 1/9)(u + 3/2) \tag{6.2.8}$$

and doubling it, we accomplish our objective, since the u-coordinate of the doubled point is always a perfect square. Specifically, we can set

$$t = \frac{u_0^2 - 1/6}{2v_0}.$$

Our "arbitrary point" can be a random multiple of the known point $(u, v) = (1,\ 5/3)$ or $(-1,\ 2/3)$ on (6.2.8).

This and other ways to get an initial point for (6.2.2) are listed in Table 6.2.1. Each method requires some homogeneous quadratic polynomial in a, b, and c to be a perfect square, while also requiring that $a^2 + b^2 = c^2$. Upon parameterizing $a = 1 - t^2$, $b = 2t$, and $c = 1 + t^2$, each entry leads to a fourth-degree polynomial in t which must be a perfect square; its solutions (if any) lie on an elliptic curve. Table entries were found in an *ad hoc* manner, so I make no claim of completeness. The last entry is due to Atkin and Morain [5], who also show how to construct curves of positive rank with other torsion groups over $\mathbb{Q}$. Elkies [15, p. 832] describes how to add points on an elliptic curve $Y^2 = \text{quartic}(t)$ with known rational points; this can be used to find more ratios for the last column of Table 6.2.1 (sometimes using a trivial rational point where $ab = 0$).

6.3 Numerical comparison of torsion subgroups of orders 12 and 16

If a curve E has a torsion subgroup of order 12 over $\mathbb{Q}$ and p is a prime which does not divide the denominator of any coefficient of E or of its torsion points, then its reduction $E_{(p)}$ modulo p has order divisible by 12 unless $E_{(p)}$ is singular modulo p or two points in the torsion subgroup agree modulo p, and hence for all

x_0	Required square	Example ratios $a : b : c$
$\frac{a}{b}$	$a^2 - ab + b^2$	$8 : 15 : 17, \quad 1768 : -2415 : 2993$
$\frac{b}{b+c}$	$a^2 + bc$	$4 : -3 : 5, \quad 15 : 8 : 17, \quad 136 : 273 : 305$
$\frac{4(c-a)}{5c-4a}$	$(5c+a)(5c-4a)$	$15 : 8 : 17, \quad 40 : 9 : 41, \quad 39 : 80 : 89$
$-\frac{4a}{3a+b}$	$(3a+b)(3a+4b)$	$7 : -24 : 25, \quad 20 : 21 : 29, \quad 209 : -120 : 241$
$-\frac{a+c}{b+c}$	$ab + ac + bc$	$80 : -39 : 89, \quad 199088 : 258825 : 326537$
$\frac{b-c}{b+c}$	$(c+2b)(c-2b)$	None found
$\frac{bc-ac-2ab}{(b+c-a)^2}$	$bc - ac - 2ab$	$111 : 680 : 689$

Table 6.2.1: Some ways to ensure that curve's group order is divisible by 16

but finitely many primes p. The ECM algorithm succeeds if $|E_{(p)}|/12$ is sufficiently smooth; this quotient is approximately $p/12$. Likewise, if E has a torsion subgroup of order 16 over $\mathbb{Q}$, then the ECM algorithm succeeds if $|E_{(p)}|/16$ is sufficiently smooth; this quotient is approximately $p/16$. Intuitively, since $p/16 < p/12$, the former seems more likely to be smooth, so curves with torsion subgroup of order 16 are "better".

A numerical experiment was conducted to check this hypothesis. It used five curves of form (6.1.1), with $u = 4$, 54, 49/4, 2166/625, 14884/1089. It also used five curves of form (6.2.2) as described in the paragraph near (6.2.8), with

$$\sqrt{u_0} \in \left\{ \frac{1}{4}, \frac{5}{8}, \frac{437}{342}, \frac{173}{14960}, \frac{4096849}{2658604} \right\}.$$

The orders of all ten curves, plus one other curve with $A = 101$ and $B = 103$, were computed modulo each of 8356 primes from 10000 to 100000; those primes for which a denominator vanished or for which one of the curves was singular (i.e. $A \equiv \pm 2$) were excluded.

When the torsion subgroup had order 12 over $\mathbb{Q}$, the prime 2 divided the group order $|E_{(p)}|$ an average of 3.68 times and the prime 3 divided the order an average of 1.68 times, effectively subtracting an average of

$$3.68 \ln 2 + 1.68 \ln 3 \approx 4.40$$

from the natural logarithm of the order. When the torsion subgroup had order 16 over Q, the prime 2 divided the order an average of 5.32 times and the prime 3 divided the order an average of 0.68 times, effectively subtracting an average of

$$5.32 \ln 2 + 0.68 \ln 3 \approx 4.43$$

from the natural logarithm of the order. Both averages are considerably larger than the $\ln(12 \cdot 2 \cdot 3^{1/2}) \approx 3.73$ (or $\ln(16 \cdot 2 \cdot 3^{1/2}) \approx 4.02$) which one would expect given a random multiple of 12 (resp. 16), but the difference between the two expectations seems slight. As expected, the curve with $A = 101$ and $B = 103$ (torsion group of order 4) performed much worse than the others, with 2 appearing to the 3.65 power and 3 to the 0.68 power on average.

In all cases the prime 5 divided the order average of 0.30 times while 7 divided the order an average of 0.19 times.

The data was subsequently analyzed for patterns depending on the residue class of p. Primes $p \equiv 1 \pmod 6$ fared better when the torsion subgroup had order 12, but primes $p \equiv 5 \pmod 6$ fared better when the torsion subgroup had order 16, as seen in Table 6.3.1. The statistics appeared not to depend on the residue class of p modulo 18 once the residue class of p modulo 6 is fixed.

$p \pmod 6$	1	1	5	5
Torsion subgroup order	12	16	12	16
Curves tried	20860	20860	20920	20920
Average exponent of 2	3.68	5.32	3.69	5.32
Average exponent of 3	1.87	0.61	1.50	0.75
Average $\ln \lvert E_{(p)} \rvert$ reduction	4.60	4.35	4.21	4.51
Power of 3:				
3^0	0	13041	0	10446
3^1	9264	4654	13968	7025
3^2	7306	2046	4605	2282
3^3	2809	746	1582	802
3^4	978	244	493	259
3^5	350	88	190	75
3^6 or more	153	41	82	31

Table 6.3.1: Power of 3 dividing group order

Tables 6.3.2 and 6.3.3 have data about the exponent of 2 dividing $|E_{(p)}|$ and $p - 1$. These data suggest Conjectures 6.3.1 and 6.3.2.

$p \pmod{16}$	1	9	5, 13	3, 7, 11, 15
Torsion subgroup order	12	12	12	12
Curves tried	5210	5210	10420	20940
Average exponent of 2	3.80	3.82	3.92	3.51
Average exponent of 3	1.68	1.67	1.69	1.68
Average $\ln \lvert E_{(p)} \rvert$ reduction	4.49	4.48	4.57	4.28
Power of 2:				
2^2	1269	1291	2540	5209
2^3	1334	1284	2582	7840
2^4	1293	1308	1949	3941
2^5	564	584	1655	1955
2^6	366	330	821	954
2^7	191	202	444	530
2^8	81	105	215	263
2^9	60	51	102	123
2^{10}	22	27	51	59
2^{11} or more	30	28	61	66

Table 6.3.2: Power of 2 dividing group order when torsion subgroup has order 12

Conjecture 6.3.1 *Let E be an elliptic curve with torsion subgroup $\mathbb{Z}/12\mathbb{Z}$ over $\mathbb{Q}$. For any prime p, let $|E_{(p)}|$ denote the order of its reduction $E_{(p)}$ modulo p. Then*

(a) *As p ranges through the primes congruent to 5 modulo 6, the largest power of 3 dividing $|E_{(p)}|$ is 3^α with probability $2 \cdot 3^{-\alpha}$ for each $\alpha \geq 1$.*

(b) *As p ranges through the primes congruent to 3 modulo 4, the largest power of 2 dividing $|E_{(p)}|$ is 2^α with probability $1/4$ if $\alpha = 2$ and probability $3 \cdot 2^{-\alpha}$ if $\alpha \geq 3$.*

(c) *As p ranges through the primes congruent to 5 modulo 8, the largest power of 2 dividing $|E_{(p)}|$ is 2^α with probability $1/4$ if $\alpha = 2$ or $\alpha = 3$, probability $3/16$ if $\alpha = 4$, and probability $5 \cdot 2^{-\alpha}$ if $\alpha \geq 5$.*

Conjecture 6.3.2 *Let E be an elliptic curve with torsion subgroup $\mathbb{Z}/8\mathbb{Z} \times \mathbb{Z}/2\mathbb{Z}$ over $\mathbb{Q}$. For any prime p, let $|E_{(p)}|$ denote the order of its reduction $E_{(p)}$ modulo p. Then*

$p \pmod{16}$	1	9	5, 13	3, 7, 11, 15
Torsion subgroup order	16	16	16	16
Curves tried	5210	5210	10420	20940
Average exponent of 2	5.77	5.87	5.49	4.98
Average exponent of 3	0.67	0.66	0.67	0.68
Average $\ln \lvert E_{(p)} \rvert$ reduction	4.74	4.80	4.55	4.20
Power of 2:				
2^4	1290	1329	2672	10583
2^5	1349	1274	3849	5235
2^6	1307	998	1916	2552
2^7	562	791	1010	1308
2^8	329	402	478	640
2^9	173	222	247	307
2^{10}	98	99	120	154
2^{11}	56	47	76	90
2^{12} or more	46	48	52	71

Table 6.3.3: Power of 2 dividing group order when torsion subgroup has order 16

(a) *As p ranges through the primes congruent to* 5 *modulo* 6*, the largest power of* 3 *dividing* $|E_{(p)}|$ *is* 3^α *with probability* $1/2$ *if* $\alpha = 0$ *and probability* $3^{-\alpha}$ *if* $\alpha \geq 1$.

(b) *As p ranges through the primes congruent to* 3 *modulo* 4*, the largest power of* 2 *dividing* $|E_{(p)}|$ *is* 2^α *with probability* $2^{3-\alpha}$ *for each* $\alpha \geq 4$.

(c) *As p ranges through the primes congruent to* 5 *modulo* 8*, the largest power of* 2 *dividing* $|E_{(p)}|$ *is* 2^α *with probability* $1/4$ *if* $\alpha = 4$ *and probability* $3 \cdot 2^{2-\alpha}$ *if* $\alpha \geq 4$.

According to Conjectures 6.3.1 and 6.3.2, the average exponent of 3 dividing the order $|E_{(p)}|$ is $3/2$ (resp. $3/4$) when the torsion group has order 12 (resp. 16) over $\mathbb{Q}$ and $p \equiv 5 \pmod{6}$. The average exponent of 2 dividing $|E_{(p)}|$ is $7/2$ (resp. 5) if $p \equiv 3 \pmod{4}$, and $63/16$ (resp. $11/2$) if $p \equiv 5 \pmod{8}$.

The conjectures make no prediction for the exponent of 3 when $p \equiv 1 \pmod{6}$, or the exponent of 2 when $p \equiv 1 \pmod{8}$. The fractions 13041/20860, 9264/20860, and 4654/20860 in Table 6.3.1 are approximately 5/8, 4/9, and 2/9 respectively, but the other entries seem hard to guess.

The evidence for these conjectures is weak, even if it is correct for curves generated using (6.1.1) or using (6.2.2) and (6.2.8), because another method of selecting the curves may give different statistics. The program was rerun, using 100 random curves with torsion group $\mathbb{Z}/12\mathbb{Z}$ (not necessarily with positive rank). and another 100 curves with torsion group $\mathbb{Z}/8\mathbb{Z} \times \mathbb{Z}/2\mathbb{Z}$, for 984 primes in [10000, 20000]. (98400 curves with each torsion group). The results resembled those in Tables 6.3.1, 6.3.2, and 6.3.3.

If p is a prime and $E_{(p)}$ has a subgroup isomorphic to $\mathbb{Z}/n\mathbb{Z} \times \mathbb{Z}/n\mathbb{Z}$ for some integer n, then $p \equiv 1 \pmod{n}$ [7, p. 954]. For example, if $p \equiv 2 \pmod{3}$, then $E_{(p)}$ cannot have a subgroup isomorphic to $\mathbb{Z}/3\mathbb{Z} \times \mathbb{Z}/3\mathbb{Z}$, so its Sylow 3-subgroup must be cyclic. Hence a power 3^k cannot divide $E_{(p)}$ unless $E_{(p)}$ has a point of order 3^k. When instead $p \equiv 1 \pmod{3}$, the Sylow 3-subgroup of $E_{(p)}$ need not be cyclic. This rationalizes why the exponent of 3 dividing the order in Table 6.3.1 varies with $p \pmod{3}$, and why the exponent of 2 in Tables 6.3.2 and 6.3.3 varies with the exponent of 2 dividing $p-1$, but does not predict the actual distributions.

CHAPTER 7

Selection of Search Limits B_1, d_1, d_2

The ECM algorithm usually finds the prime $p|N$ if the order $|E_{(p)}|$ has all but possibly one prime divisor below a bound B_1, and its remaining prime divisor q (if any) divides $m_i \pm n_j$ for some i and j satisfying (5.0.1). We want to estimate the expected cost required to find p, in terms of the size of p and the search parameters B_1, d_1, d_2. Here d_1, d_2 are as in Chapter 5. To achieve this, we estimate $\mathrm{Pr}_{\mathrm{succ}}(B_1, d_1, d_2)$, the probability of finding p using parameters B_1, d_1, and d_2 with a random curve. We also estimate $\mathrm{Cost}(B_1, d_1, d_2)$, the cost of running one curve with these parameters. Then we attempt to minimize the expected total cost

$$(7.0.1) \qquad \frac{\mathrm{Cost}(B_1, d_1, d_2)}{\mathrm{Pr}_{\mathrm{succ}}(B_1, d_1, d_2)}.$$

The numerator estimate depends on the magnitude of N (arithmetic is cheaper if N is smaller). The denominator estimate depends on the prime p (larger probabilities for smaller p). The minimization process treats B_1, d_1, and d_2 as continuous real variables, while fixing N and p. This process ignores the requirements that d_1 be a power of 2 and that $d_1|d_2$. Fortunately, the expected cost is very flat near its minimum (cf. the last two columns of Table 7.4.1), so imposing these restrictions later does not significantly affect the estimated total cost.

The analysis assumes use of curves with torsion group of order 16 over $\mathbb{Q}$ (Section 6.2). It also assumes that Step 2 uses $P(X) = X^{24}$ as in Section 5.10.

7.1 Dickman's function

Dickman's function $\rho(\alpha)$ [9, pp. 3–4] [17, p. 367] estimates the probability that a large integer x has all its prime factors below $x^{1/\alpha}$. It satisfies the functional equation

$$(7.1.1) \qquad \rho(\alpha) = 1 \qquad (0 \le \alpha \le 1),$$

$$\alpha\rho'(\alpha) = -\rho(\alpha - 1) \qquad (\alpha > 1)$$

(the ranges for α in [9] are incorrect). An asymptotic formula [9] is

$$\ln \rho(\alpha) = -\alpha(\ln \alpha + \ln\ln \alpha - 1) + o(\alpha) \qquad (\alpha \to \infty).$$

7.2 Estimated success probability per curve

The curve has been chosen to have order divisible by 16; its order is approximately p. In terms of Dickman's function (7.1.1), the estimated success probability during Step 1 is the probability that an integer near $p/16$ has all its prime factors below B_1, namely

$$\rho\left(\frac{\ln(p/16)}{\ln B_1}\right). \tag{7.2.1}$$

(This estimate is slightly pessimistic, since the data in Section 6.3 suggest that the average powers of 2 and 3 dividing $|E_{(p)}|/16$ are larger than those dividing random integers.)

Step 2 succeeds if there exists a prime $q > B_1$ such that

(i) q divides the group order;

(ii) all prime factors of $|E_{(p)}|/16q$ are below B_1;

(iii) two of the multiples of Q constructed during Step 2 satisfy (5.0.1).

Item (i) has estimated probability $1/q$ for $q < p/16$ and probability 0 otherwise. The estimated probability of (ii) is given by Dickman's function. By (5.3.5), the estimated probability of (iii) is

$$\begin{aligned}
\Pr_{\text{match}}(d_1,\, d_2,\, q) &= 1 - \frac{1}{\phi(48)} \sum_{\substack{q_0 \pmod{48} \\ \gcd(q_0,\, 48)=1}} \exp\left(\frac{-d_1 d_2}{q} \gcd(48,\, q_0 - 1)\right) \\
&= 1 - \frac{1}{16}\Big(4\exp(-2d_1d_2/q) + 2\exp(-4d_1d_2/q) \\
&\qquad + 4\exp(-6d_1d_2/q) + \exp(-8d_1d_2/q) \\
&\qquad + 2\exp(-12d_1d_2/q) + \exp(-16d_1d_2/q) \\
&\qquad + \exp(-24d_1d_2/q) + \exp(-48d_1d_2/q)\Big)
\end{aligned}$$

Since at most one $q > B_1$ can satisfy both (i) and (ii), we can sum the product of these probabilities over all prime $q > B_1$, leading to an estimated Step 2 success probability of

$$\sum_{\substack{B_1<q<p/16\\ q\text{ prime}}} \frac{\mathrm{Pr}_{\mathrm{match}}(d_1,\, d_2,\, q)}{q}\, \rho\left(\frac{\ln(p/16q)}{\ln B_1}\right)$$
$$\sim \int_{B_1}^{p/16} \frac{\mathrm{Pr}_{\mathrm{match}}(d_1,\, d_2,\, q)}{q}\, \rho\left(\frac{\ln(p/16q)}{\ln B_1}\right) \frac{dq}{\ln q}.$$

Substituting $q = \exp(q')$ and adding (7.2.1) gives a total success probability of

(7.2.2)

$$\mathrm{Pr}_{\mathrm{succ}}(B_1,\, d_1,\, d_2) = \int_{\ln B_1}^{\ln(p/16)} \mathrm{Pr}_{\mathrm{match}}(d_1,\, d_2,\, \exp(q'))\, \rho\left(\frac{\ln(p/16) - q'}{\ln B_1}\right) \frac{dq'}{q'} + \rho\left(\frac{\ln(p/16)}{\ln B_1}\right).$$

7.3 Estimated time per curve

The rows of Table 7.3.1 summarize the major actions during the ECM algorithm. (Actions taking negligible time such as selection of the curve itself are omitted.) Each row has four entries:

(i) A brief description of the action. Actions appear in the order in which they are first executed.

(ii) The number of times the action is executed per curve.

(iii) Its asymptotic cost (per execution) for large d_1, d_2, and B_1 but fixed N. This column assumes that $M(d) = O(d \log d)$ in Section 3.1.

(iv) An estimated cost, using the first term of the asymptotic cost, with constants chosen to match actual run times (in milliseconds on a DEC 5000) for a 150–digit N. Specifically, that run attempted to factor the cofactor of $p(20021)$ listed in Table 9.2.1, with $B_1 = 3 \cdot 10^6$ and $d_1 = 8192$ and $d_2 = 81920$.

Action	Times executed	Asymptotic cost per execution	Fitted cost for 150–digit N (msec.)
Step 1	1	$O(B_1)$	$5.5\,B_1$
Roots of F	1	$O(d_1)$	$105\,d_1$
Construct $F(X)$	1	$O(d_1(\log d_1)^2)$	$0.16\,d_1(\log_2 d_1)^2$
Construct RECIP$(F(X))$	1	$O(d_1 \log d_1)$	$1.5\,d_1 \log_2 d_1$
Roots of H	d_2/d_1	$O(d_1)$	$25\,d_1$
Construct $H(X)$	d_2/d_1	$O(d_1(\log d_1)^2)$	$0.16\,d_1(\log_2 d_1)^2$
$G(X) \leftarrow G(X)H(X) \bmod F(X)$	$d_2/d_1 - 1$	$O(d_1 \log d_1)$	$1.1\,d_1 \log_2 d_1$
$\gcd(F(X),\, G(X))$	1	$O(d_1(\log d_1)^2)$	$1.5 d_1(\log_2 d_1)^2$

Table 7.3.1: Estimated time per curve (milliseconds)

Table 7.3.1 predicts a total time per curve of

(7.3.1)

$$\begin{aligned}
&\mathrm{Cost}(B_1,\, d_1,\, d_2) \\
&= 5.5B_1 + 105d_1 + 0.16d_1(\log_2 d_1)^2 + 1.5d_1 \log_2 d_1 \\
&\qquad + \frac{d_2}{d_1}\Big(25d_1 + 0.16d_1(\log_2 d_1)^2 + 1.1d_1 \log_2 d_1\Big) - 1.1d_1 \log_2 d_1 + 1.5d_1(\log_2 d_1)^2 \\
&= 5.5B_1 + 105d_1 + 25d_2 + (0.4d_1 + 1.1d_2)\log_2 d_1 + (1.66d_1 + 0.16d_2)(\log_2 d_1)^2.
\end{aligned}$$

The precise constants in this estimate depend on the implementation and the hardware available. Using more precise asymptotic costs also affects (7.3.1).

7.4 Estimated optimal parameters

We want to minimize (7.0.1). The constants in its numerator (cost estimate (7.3.1)) were derived assuming that N has 150 digits. If all costs of the computation grow in equal proportions N increases, then the location of the minimum does not depend on N. (However (3.4.5) suggests this proportionality assumption is incorrect; indeed the data in Table 9.1.1 show that operations modulo a 200–digit N take 2.4–2.8 times as long as those modulo a 100–digit N, whereas the multiplication modulo N during Step 1 presently uses an $O((\log N)^2)$ algorithm.) The denominator (success probability estimate (7.2.2)) depends heavily on the magnitude of p.

Table 7.4.1 gives estimates of the optimal parameters (i.e. those minimizing (7.0.1)) for various sizes of p. It includes estimated run times (in hours) for the environment of the last column Table 7.3.1. The minimization was done numerically. To approximate a prime of d decimal digits, we put $p = 10^{d-1/2}$ in (7.2.2). The integral in (7.2.2) was approximated by Simpson's rule. Dickman's function was approximated using interpolation in a table.

The third and fourth columns of Table 7.4.1 suggest that one should use $d_2 \approx 7d_1$. The expected cost in the sixth column increases by 50% for each each additional digit in p. With optimal parameters, approximately two-thirds of the run time is in Step 1 (this percentage is 57%, 65%, 68%, 72% when p has 20, 30, 40, 50 digits). The conditional probability that a success occurs during Step 1 rather than Step 2 while using optimal parameters drops from 11% to 5% as p increases from 20 to 50 digits.

In practice the size of p is usually unknown. The last column of Table 7.4.1 gives the estimated times to find p of various sizes using parameters optimized for a p of 31 digits. The estimated times using these parameters are at most twice the corresponding optimal times if p has 22–41 digits, and within 20% of the corresponding optimal times if p has 27–36 digits. If N has 150 digits, then each curve takes about 1.2 hours on a DEC 5000 using these parameters.

The program used to generate Table 7.4.1 was rerun, using $P(X) = X^{12}$ rather than $P(X) = X^{24}$. The estimated run times were 3%–10% smaller than the corresponding times in Table 7.4.1, suggesting that the exponent 24 is too large. The corresponding table for $P(X) = X^{12}$ has values of B_1 about 5%–10% smaller than those in Table 7.4.1; its values of d_1 are 2% larger while its values of d_2 are 15%–20% larger (suggesting $d_2 = 8d_1$ or $d_2 = 9d_1$). These differences are more significant for smaller p. Both $P(X) = X^{24}$ and $P(X) = X^{12}$ have smaller expected times than $P(X) = X^4$.

On a machine with 1024 parallel processors each as fast as a DEC 5000 (or a network of these), Table 7.4.1 predicts that one can find all factors up to 33 digits of a 150–digit N within an hour, up to 41 digits within a day, and 50 digits within a month, assuming perfect parallelism. Such systems may be widely available in ten years, allowing Rusin's 42–digit ECM record in Table 1.0.1 to be beaten many times. Brent [9, p. 18] predicts that factors up to 50 digits can be found this way.

Digits in p	B_1	d_1	d_2	Expected number of curves	Expected time (hrs.) for 150–digit N	Time with $B_1 = 500000$, $d_1 = 2048$, $d_2 = 16384$
20	18000	160	1000	51	2.4	7.3
21	25000	210	1300	61	4.0	9.7
22	34000	270	1700	73	6.5	13
23	47000	340	2200	87	11	18
24	65000	430	2900	100	17	26
25	88000	540	3700	120	27	37
26	120000	690	4700	150	43	53
27	160000	860	6100	170	68	77
28	220000	1100	7700	200	110	110
29	290000	1400	9800	240	170	170
30	390000	1700	12000	280	260	260
31	510000	2100	16000	330	390	390
32	670000	2600	19000	380	600	610
33	880000	3300	24000	450	920	950
34	1200000	4000	30000	520	1400	1500
35	1500000	5000	38000	610	2100	2400
36	1900000	6100	47000	710	3200	3800
37	2500000	7500	58000	830	4700	6100
38	3200000	9200	72000	960	7000	10000
39	4200000	11000	88000	1100	10000	16000
40	5300000	14000	110000	1300	15000	27000
41	6800000	17000	130000	1500	23000	45000
42	8700000	21000	160000	1700	33000	75000
43	11000000	25000	200000	2000	48000	130000
44	14000000	30000	250000	2300	71000	220000
45	18000000	37000	300000	2600	100000	370000
50	56000000	95000	790000	5300	630000	7800000

Table 7.4.1: Estimated optimal parameters

CHAPTER 8

Multiple-Precision and Modular Arithmetic

Timing runs revealed that this program's time was concentrated on four activities:

(i) Arithmetic modulo N, esp. multiplication. This is the major activity during Step 1; it is also used during Step 2 when calculating the x-coordinates for use in (4.3.1).

(ii) Arithmetic modulo (all) primes p_i during a convolution.

(iii) Finding remainders modulo all p_i given a value modulo N (Section 3.4).

(iv) Reconstructing remainder modulo N given remainders modulo several p_i (equation (3.4.4)).

The program has been designed to allow parallelism, but each of the above steps was treated to be an indivisible operation and assumed to be completed on a single processor. For example, the design allows several independent multiplications modulo N to proceed at once, but any such multiplication is completed by a single processor.

The only parallel architecture used during the study was an Alliant FX/80, a MIMD architecture which also supports vectorization [4]. If a convolution required K prime moduli, then the data was structured so K remainders modulo different primes were stored in adjacent locations, allowing vectorization with unit stride over the primes (this contrasts with Silverman's implementation [32], which tried to assign each prime modulus to a separate processor during the convolutions). A typical primitive operation used by the FFT is

$$(a_i,\ b_i) \leftarrow (a_i + \omega_i b_i,\ a_i - \omega_i b_i) \bmod p_i \qquad (1 \le i \le K), \tag{8.0.1}$$

which has one multiplication, one addition, and one subtraction modulo each p_i, all potentially vectorizable.

8.1 Arithmetic modulo N

Arithmetic modulo N used the algorithm in [28]. Suppose that N can be represented using ℓ digits in radix R, where R is a power of 2. Let $0 \le A,\ B < N$, with

$$(8.1.1) \qquad A = \sum_{i=0}^{\ell-1} a_i R^i, \qquad B = \sum_{i=0}^{\ell-1} b_i R^i, \qquad N = \sum_{i=0}^{\ell-1} n_i R^i,$$

and $0 \le a_i,\ b_i,\ n_i < R$ for all i. The algorithm requires a constant N' such that

$$N' \cdot N \equiv -1 \pmod{R};$$

such exists since N is odd.

Procedure MODMULN in Figure 8.1.1 is based on the classical (not high-speed) multiplication techniques (i.e. its time is $O(\ell^2)$). It returns $AB/R^\ell \bmod N$ rather than $AB \bmod N$. To compensate, all residues modulo N should be scaled by R^ℓ beforehand. Define

$$\overline{A} = AR^\ell \bmod N$$

for any integer A modulo N. Then

$$\overline{(A \pm B)} \equiv (A \pm B)R^\ell = AR^\ell \pm BR^\ell \equiv \overline{A} \pm \overline{B} \pmod{N},$$

$$\overline{(AB)} \equiv (AB)R^\ell \equiv (AR^\ell)(BR^\ell)R^{-\ell} \equiv \overline{A}\,\overline{B}R^{-\ell} \equiv \text{MODMULN}(\overline{A},\ \overline{B}) \pmod{N}.$$

If $A \bmod N$ is represented internally by $\overline{A}$, then using MODMULN on two internal representations gives the internal representation of their product modulo N. The addition and subtraction algorithms are unchanged. Algebraically, the mapping $A \to \overline{A}$ is an isomorphism from the ring $\mathbb{Z}/N\mathbb{Z}$ with conventional arithmetic to that set with ordinary addition but with multiplication defined by MODMULN. The additive identity remains $\overline{0} = 0$, but the multiplicative identity becomes $\overline{1}$.

Given two polynomials $F(X)$ and $G(X)$, the algorithms of Section 3.4 return $F(X)G(X) \bmod N$. Suppose we invoke it on

$$\overline{F}(X) = \sum_i \overline{f_i} X^i \qquad \text{and} \qquad \overline{G}(X) = \sum_i \overline{g_j} X^j.$$

Each output coefficient has the form

$$\sum_{i,j} \overline{f_i}\,\overline{g_j} \equiv \sum_{i,j} \overline{(f_i g_j)} R^\ell \pmod{N}$$

rather than the desired $\sum_{i,j} \overline{(f_i g_j)}$. That is, they are too large by a factor of $R^\ell \bmod N$. This correction is easily incorporated into the convolution algorithm, by scaling the pre-computed coefficients in (3.4.4).

procedure MODMULN(A, B)
$C := 0$
for j **from** 0 **to** $\ell - 1$ **do**
  Cmt. Suppose $C = \sum_{i=0}^{\ell-1} c_i R^i$.
  $dtemp := c_0 + a_j \cdot b_0$
  $q_j := (dtemp \cdot N') \bmod R$
  Cmt. Compute $C := ((C - c_0) + a_j \cdot (B - b_0) + q_j \cdot N + dtemp)/R$.
  $carry := (dtemp + q_j \cdot n_0)/R$
  for i **from** 1 **to** $\ell - 1$ **do**
    Cmt. $0 \le a_j,\ q_j,\ b_i,\ n_i \le R - 1$.
    Cmt. $0 \le carry \le 2(R-1)$.
    Cmt. $0 \le c_i \le R - 1$ if $i < \ell - 1$.
    Cmt. $0 \le dtemp \le 2R^2 - R - 1$ if $i < \ell - 1$.
    Cmt. $0 \le c_{\ell-1} \le R - 1$.
    Cmt. $0 \le dtemp \le 2R^2 - 1$ if $i = \ell - 1$.
    $dtemp := c_i + carry + a_j \cdot b_i + q_j \cdot n_i$
    $c_{i-1} := dtemp \bmod R$
    $carry := \lfloor dtemp/R \rfloor$
  end for
  $c_{\ell-1} := carry$
  Cmt. $C \cdot R^{j+1} = (a_j a_{j-1} \cdots a_0)_R \cdot B + (q_j q_{j-1} \cdots q_0)_R \cdot N$.
end for
if $C \ge N$ **then** $C := C - N$
return C
end MODMULN

Figure 8.1.1: Procedure for multiplication modulo N

8.2 Double-length multiplication

The portion of Algorithm MODMULN of Figure 8.1.1 preceding the final $C \geq N$ test requires the following primitive arithmetic operations:

(i) Addition of nonnegative integers with sum not exceeding $2R^2 - 1$.

(ii) Multiplication of two nonnegative integers below R, with product as large as $(R-1)^2$.

(iii) Integer quotient and remainder when dividing a nonnegative integer below $2R^2$ by R. On a binary machine this is equivalent to extracting the least significant $\log_2 R$ bits or the next $1 + \log_2 R$ bits of the input.

The length of the operands is $\ell = \lceil \log_R N \rceil$; to keep ℓ (and the number of loop iterations) small, the the radix R be as large as convenient while fitting into a machine word.

I coded MODMULN in four essentially different ways, some aimed for portability and some for high performance on certain architectures. The subalgorithm is chosen at compile time. Different subalgorithms impose different bounds on the radix R.

Direct. This subalgorithm requires the hardware and language to support integers as large as $2R^2 - 1$. On a 32–bit machine, we can restrict $R \leq 2^{15}$ while using ordinary integers, or $R \leq 2^{30}$ while using 64–bit results.

The Fortran 77, Fortran 90, and ANSI C language standards do not specify a long integer data type (though the **kind** parameter of Fortran 90 permits it). The GNU C compiler does support a **long long** data type, but UCLA's current version (1.39) does not inline the code operating on such data, making it needlessly slow (though Version 2 has improved support). This subalgorithm was used primarily in some assembly language versions of MODMULN.

Halfint. This requires that R be an even power of 2. To multiply two operands a and b where $0 \leq a,\, b < R$, one writes $a = a_0 + a_1\sqrt{R}$ and $b = b_0 + b_1\sqrt{R}$, where $0 \leq a_0,\, a_1,\, b_0,\, b_1 < \sqrt{R}$. Then $ab = a_0 b_0 + (a_1 b_0 + a_0 b_1)\sqrt{R} + a_1 b_1 R$; all products are less than R. We can use three multiplications rather than four by utilizing the Karatsuba identity (3.1.1).

If $\sqrt{R}$ is sufficiently small, then the multiplications of integers below $\sqrt{R}$ can be done by table look-up, using one of the identities

$$ab = \left\lfloor \frac{(a+b)^2}{4} \right\rfloor - \left\lfloor \frac{(a-b)^2}{4} \right\rfloor \tag{8.2.1}$$

$$ab = \frac{(a+b+1)(a+b)}{2} - \frac{a(a+1)}{2} - \frac{b(b+1)}{2}. \tag{8.2.2}$$

The use of a table of squares (8.2.1) requires only two table look-ups per multiplication versus three when using triangular numbers (8.2.2). Equation (8.2.1) uses a table indexed from from $1 - \sqrt{R}$ to $2\sqrt{R} - 2$ whereas (8.2.2) uses indices only from 0 to $2\sqrt{R} - 2$. A test showed (8.2.2) outperforming (8.2.1) (on a program where the most cache data would be from this table), but both table look-up methods did poorly.

Double. If the double precision floating point data type can represent integers as large as $2R^2 - 1$ exactly, and if all arithmetic is exact when the inputs and outputs are integers and outputs below this bound, then all arithmetic can be done using floating point and converted back afterwards. On a machine whose double precision mantissa has 53 bits, as in the IEEE standard, this allows $R = 2^{26}$. On a SUN 4, an optimized Fortran implementation of MODMULN using this subalgorithm performed almost as well as an assembly subprogram using the **direct** subalgorithm and **mulscc** instructions.

Twos-Comp. This subalgorithm assumes that all integer addition, subtraction, and multiplication is done modulo some 2^b with overflow ignored, where $2^b \geq R$ (often $b = 32$). That is, all integer arithmetic really operates in $\mathbb{Z}/2^b\mathbb{Z}$; this is the same as twos' complement arithmetic on b–bit integers. This subalgorithm also requires that the floating point data type hold items a few bits larger than required for R.

Suppose we want the upper and lower halves of $a_1a_2 + a_3a_4$, where $0 \leq a_i < R$. Compute $t_0 = (a_1a_2 + a_3a_4 \bmod 2^b) \bmod R$, using integer arithmetic with overflow ignored; its remainder modulo R is in the lower bits. The top half $(a_1a_2 + a_3a_4 - t_0)/R$ of this result is known to be an integer less than $2R$; it can be estimated using floating point arithmetic whose mantissa is sufficiently wide, with the result rounded to the nearest integer.

The **twos-comp** subalgorithm was originally developed for the Alliant, which can return a double-length (64–bit) integer product in scalar mode but not in vector mode [4]. It is also the best-performing non-assembly subalgorithm on the DEC 5000 and IBM RS/6000, whose integer and floating point calculations of $a_1a_2 + a_3a_4$ can proceed concurrently in separate functional units. The RS/6000 compilation used the unsupported **-qxflag=hsflt** of the **xlf** Fortran compiler, to suppress overflow checking when converting floating point to integer (both here and while preparing Table 9.1.1). Without this compilation option, **halfint** was the fastest subalgorithm on the RS/6000.

8.3 Arithmetic modulo small primes

The primes p_i selected in (3.4.1) should be large enough that only a few are needed, but small enough that arithmetic modulo the p_i can be done easily. In order that enough p_i exist satisfying the congruence condition for primitive roots, their minimum size is about 24 bits.

A compilation option allows elements of $\mathbb{Z}/p_i\mathbb{Z}$ to be represented as operands in the interval $[-p_i,\ p_i]$ (signed operands) or in $[0,\ p_i-1]$ (nonnegative operands). The typical operation (8.0.1) is the composition of two simpler operations: replacing

(8.3.1)

$$b_i \leftarrow \omega_i b_i \bmod p_i \qquad \text{followed by} \qquad (a_i,\ b_i) \leftarrow (a_i + b_i,\ a_i - b_i) \bmod p_i$$

given $\{a_i\}$, $\{b_i\}$, $\{p_i\}$. Since the p_i rarely change, we also permit auxiliary constants which depend only on the p_i and can be pre-computed. We want to implement these operations in a vectorizable fashion.

If a_i and b_i are nonnegative (i.e. in $[0,\ p_i - 1]$), then one can implement the second operation in (8.3.1) by computing $a_i - p_i + b_i$ and $a_i - b_i$, then adding p_i to either result if negative. If instead a_i and b_i are signed operands, then one can first adjust a_i and b_i to the interval $[0,\ p_i]$ if they are negative, followed by outputting $a_i - p_i + b_i$ and $a_i - b_i$. In each case the outputs follow the same conventions as the inputs, at a cost of five vectorizable adds/subtracts (two of them under conditional mask) plus two vector compares and some loads and stores. The selection of signed or unsigned data depends primarily on the subalgorithm for multiplication modulo p_i.

Suppose we want a product $\omega_i b_i \bmod p_i$. Analogous to the **direct** method in Section 8.2, one can compute the double-length product $\omega_i b_i$ and divide by p_i, if the machine and language support double-length multiplication and division. These operations are most simply done on nonnegative operands. Division can be avoided if one uses a method analogous to MODMULN, which is allowed to return $\omega_i b_i/2^{\ell'} \pmod{p_i}$ where ℓ' is a constant and $2^{\ell'} > p_i$. This variation finds $c_i \in [0,\ 2^{\ell'} - 1]$ such that $\omega_i b_i - c_i p_i$ is a multiple of $2^{\ell'}$, returning either $(\omega_i b_i - c_i p_i)/2^{\ell'}$ or $(\omega_i b_i - c_i p_i)/2^{\ell'} + p_i$.

Subalgorithm **halfint** can also be used here, by breaking each input into two pieces half as long, and using an even exponent ℓ'. It resembles MODMULN in Figure 8.1.1, with N replaced by p_i, ℓ replaced by 2, and R replaced by $2^{\ell'/2}$. This subalgorithm also prefers nonnegative operands.

Subalgorithm **double** find approximate quotients q_i such that $|q_i - \omega_i b_i/p_i| \le 1$, by computing $\omega_i b_i/p_i$ with round-off error at most 0.5 and rounding that result to the nearest integer. The outputs $\omega_i b_i - p_i q_i$ can then be computed exactly if the floating point mantissa is sufficiently wide (e.g. $p_i^2 < 2^{53}$ when using 53–bit mantissa). This subalgorithm prefers signed data. If the nearest integer function (Fortran NINT) is slow but truncation towards zero is fast, then approximate

$$\frac{\omega_i b_i}{p_i} \sim \omega_i b_i \frac{1 + 1/2p_i}{p_i}.$$

Estimate the right side using floating point arithmetic. Except when $\omega_i b_i = 0$, the approximation is too large in absolute value but its relative error is less than $1/p_i$, if floating precision is sufficiently large. Upon converting the right side to an integer while truncating towards zero, the estimated quotients q_i are either correct or one too large, and the computed remainders are in the interval $[-p_i, p_i]$, as desired.

Subalgorithm **twos-comp** also generalizes. As in **double**, estimate a quotient $q_i \sim \omega_i b_i / p_i$ via floating point arithmetic. Then compute the remainder $\omega_i b_i - p_i q_i$ using integer arithmetic modulo a power of 2 exceeding $2p_i$. We know that the computed remainder has absolute value at most p_i, so the computed result must be correct despite intermediate integer overflow. This subalgorithm prefers signed data.

Some of these subalgorithms find $\omega_i b_i / 2^{\ell'} \bmod p_i$ for some $\ell' > 0$ rather than $\omega_i b_i \bmod p_i$. One can compensate by scaling all inputs (including primitive roots) by $2^{\ell'} \bmod p_i$, much as used by inputs to MODMULN. Since we must divide by $2^{\ell'} \bmod p_i$ after the convolutions modulo p_i, the constant $(P/p_i)^{-1}$ in (3.4.2) should *not* be pre-scaled; that multiply removes any bias.

8.4 Finding remainders modulo p_i

The convolution algorithm in Section 3.4 reduces all input polynomial coefficients modulo p_i. It may also need to scale these residues by $2^{\ell'} \bmod p_i$ if the multiplication subalgorithm of Section 8.3 so requires. Let a sample input be $A = \sum_{j=0}^{\ell-1} a_j R^j$, as in Section 8.1.

The direct algorithm begins with the leading coefficient $a_{\ell-1}$, which it reduces modulo p_i. Then it reduces $a_{\ell-1}R + a_{\ell-2}$ modulo p_i, and so on.

To reduce the number of divisions, we can pre-compute all $2^{\ell'} R^j \bmod p_i$. Now the computation reduces to an inner product

$$2^{\ell'} A \equiv \sum_{j=0}^{\ell-1} a_j \left(2^{\ell'} R^j \bmod p_i\right) \pmod{p_i} \tag{8.4.1}$$

with each summand bounded by $(R-1)(p_i - 1)$. Analogues of the multiplication subalgorithms in Section 8.2 handle this inner product. For **direct** and **halfint** it is convenient to instead reduce $A \cdot 2^{\ell'} R$ modulo p_i, getting a remainder in $[0, Rp_i - 1]$ and dividing the latter by $R \bmod p_i$; this is accomplished by replacing $2^{\ell'} R^j$ by $2^{\ell'} R^{j+1}$ in (8.4.1) (and with an additional entry per p_i in the pre-computed table).

8.5 Reconstructing remainder modulo N

The outputs h_j of a convolution are determined by an inner product (3.4.4) in which one operand is a residue modulo p_i and the other is a residue modulo N; the inner product is calculated modulo N. It resembles

$$h \equiv \sum_{i=0}^{K} a_i B_i \pmod{N} \tag{8.5.1}$$

where a_i is a single-precision integer and $0 \leq B_i < N$; the integers B_i are pre-computed.

Writing $B_i R \equiv \sum_{j=0}^{\ell-1} b_{ij} R^j \pmod{N}$, (8.5.1) becomes

$$hR \equiv \sum_{i=0}^{K} a_i (RB_i) \equiv \sum_{i=0}^{K} a_i \sum_{j=0}^{\ell-1} b_{ij} R^j = \sum_{j=0}^{\ell-1} R^j \sum_{i=0}^{K} a_i b_{ij} \pmod{N}.$$

The inner sums are evaluated as triple-precision base-R integers, one per j. From the last digit of this calculated product, we can determine a multiple of N which when added to the product gives a multiple of R. That multiple is added while doing carry propagation, to get a result in radix R which is less than $\left((R-1) + \sum_{i=0}^{K} a_i\right)$ times N. Dividing this sum by R (by eliminating its last digit) gives a remainder which is at most a small multiple of N (at most $O(KN)$ if the p_i and R have the same magnitude); the final quotient and remainder h are determined using the binary search algorithm on a table of small multiples of N followed by a multiple-precision subtraction.

8.6 Vectorized carry propagation

Multiple-precision addition and subtraction are much cheaper than multiplication, but are nonetheless used heavily by ECM. They can be done by starting with the least significant digit and working up [17, p. 251]. For example, during modular addition one may need to compute $C = A + B - N$ where A, B, and N are given by (8.1.1) and C has a like form. Assuming that the radix R is sufficiently small that $4R$ fits in a word, one straightforward algorithm appears in Figure 8.6.1.

This algorithm does not vectorize well. Although the computations of $a_i + b_i - n_i$ can be vectorized, each carry potentially depends on all earlier carries.

Cmt. Compute $C = A + B - N$, plus carry out.
$carry := 0$
for i **from** 0 **to** $\ell - 1$ **do**
  $carry := carry + a_i + b_i - n_i$
  $c_i := carry \bmod R$
  $carry := \lfloor carry/R \rfloor$
end for

Figure 8.6.1: Straightforward multiple-precision addition and subtraction

Bailey [6, p. 286] vectorizes this operation in almost all cases by assuming that the carries propagate at most three places to the left. When this assumption fails, Bailey resorts to scalar operations.

Full vectorization is possible if one can perform integer arithmetic (shifts, adds, bitwise operations) on vector masks. Initialize $c_i = a_i + b_i - n_i$ for $0 \le i \le \ell - 1$. These satisfy $1 - R \le c_i \le 2R - 2$ for all i. Then apply the algorithm in Figure 8.6.2 to the $\{c_i\}$ vector.

Each iteration of the inner loop in Figure 8.6.2 does carry propagation across a vector of length $lnow + 1 \le 32$, ensuring that all but possibly the most significant element of the output vector are in range. The values of $\{v_i\}_{i=0}^{lnow+1}$ are assumed stored in a vector register. The first two **where**'s in the inner loop subtract R from any elements which previously were $R - 1$ or larger, while adding 1 to their left neighbors. This does not alter the numerical value of $\sum_{i=0}^{\ell-1} c_i R^i$, but avoids digits larger than $R - 1$. Negative digits, however, may still remain. All remaining carries are -1 or 0, whereas previously they might have been -1, 0, or $+1$.

The locations where a carry in is -1 are those where (i) the right neighbor is negative, or (ii) the right neighbor is zero and that neighbor has a carry in of -1. Define $m_i = 0$ if v_i is positive, $m_i = 1$ if $v_i = 0$, and $m_i = 2$ if $v_i < 0$. Form the binary sum $\sum_{i=0}^{lnow-1} m_i 2^i = msk_< + msk_\le$ using the integer add instruction on vector masks. There are binary carries from all columns where $m_i = 2$ (i.e. those where $v_i < 0$), and from those with $m_i = 1$ (i.e. $v_i = 0$) which receive a carry in. The algorithm sets bits in msk_2 corresponding to columns receiving a carry in; $msk_2/2$ has bits corresponding to carry outs. These are precisely the locations where the v_i need to be adjusted for carry ins and carry outs in radix R.

The algorithm in Figure 8.6.2 was implemented in assembly language, because the Alliant Fortran compiler does not support arithmetic operations on vector masks. Caveat: the bits in the Alliant's vector masks are numbered with bit 0 being most significant and bit 31 being least significant. In order to get the bits in

Cmt. Input digits of C assumed in $[1-R,\ 2R-2]$,
Cmt. except c_0 is allowed to be $-R$ or $2R-1$.
Cmt. On exit, all but possibly $c_{\ell-1}$ are in $[0,\ R-1]$.
$left := \ell$
while $left > 1$ **do**
  $lnow := \min(32,\ left) - 1$ /* 32 = Maximum vector length */
  Set $v_i := c_{\ell-left+i}$ for $0 \le i \le lnow$.
  $msk_1 :=$ mask where $v_i \ge R-1$ **and** $0 \le i < lnow$
  where msk_1 **do** $v_i := v_i - R$
  where $2 * msk_1$ **do** $v_i := v_i + 1$
  Cmt. Now all but possibly v_0 and v_{lnow} are in $[1-R,\ R-1]$;
  Cmt. v_0 may equal $-R$.
  $msk_< :=$ mask where $v_i < 0$ **and** $0 \le i < lnow$
  $msk_\le :=$ mask where $v_i \le 0$ **and** $0 \le i < lnow$
  $msk_2 := \mathrm{EOR}(msk_< + msk_\le,\ msk_<,\ msk_\le)$ /* EOR = exclusive OR */
  Cmt. $msk_2/2$ identifies which entries are too small.
  where msk_2 **do** $v_i := v_i - 1$
  where $msk_2/2$ **do** $v_i := v_i + R$
  Set $c_{\ell-left+i} := v_i$ for $0 \le i \le lnow$.
  $left := left - lnow$
end while

Figure 8.6.2: Vectorized carry propagation

the proper order for computing msk_2, the vector loads and stores used stride -1 rather than $+1$. That is, v_i really held $c_{\ell-left+lnow-i}$ rather than $c_{\ell-left+i}$.

A timing run on the Alliant FX/80 (using one processor) showed the algorithm in Figure 8.6.2 to be about 30% faster than a partially vectorized implementation of Figure 8.6.1 on long vectors but worse on short vectors; the crossover point is about $\ell = 10$. Using an **if** to bypass the **where**'s when a mask is identically zero gives an additional 10% improvement.

CHAPTER 9

Results

9.1 Timing

Table 9.1.1 gives timing information for various polynomial operations on an IBM RS/6000 under AIX 3.1. Computations were done modulo $N = \lceil 10^{49}\pi \rceil + 60$, $N = \lceil 10^{99}\pi \rceil + 70$, $N = \lceil 10^{149}\pi \rceil + 295$, $N = \lceil 10^{199}\pi \rceil + 64$, using random operands. These four values of N are probable primes of 50, 100, 150, and 200 digits. The convolution algorithm of Section 3.4 used respectively 12, 22, 33, and 44 primes just below 2^{31} for the convolutions modulo p_i. All times are given in hundredths of seconds, though digits beyond the two most significant are probably noise.

Table 9.1.2 compares the corresponding times on an Alliant FX/80 for the 100–digit N, once running on a single processor and once running with five MIMD (multiple-instruction, multiple-data) processors. The timing runs were made during a period of little other system activity. As on the RS/6000, convolutions modulo this N used 22 small primes near 2^{31}.

The Alliant times can be contrasted with those in [32, Table 1]. Silverman took $440 + 425 + 700 = 1565$ seconds to construct a polynomial of degree 15360 from its roots when $N \approx 10^{100}$, on an Alliant FX/8 using four processor. Our estimated time to construct such a polynomial of degree 16384 is $2 \cdot 457 + 95 = 1009$ seconds using one processor, and $2 \cdot 107 + 22 = 236$ seconds using five processors. According to the Alliant architecture manual [4, Appendices F and G], the FX/8 and FX/80 both have cycle times of 170 nanoseconds, but the FX/80 often requires fewer cycles per instruction. For example, converting a 32–bit integer vector in memory with stride 1 to a double precision vector in a register (**vmoveld** instruction), takes $7 + VL$ cycles on the FX/8 but $2 + \lceil VL/2 \rceil$ cycles on the FX/80, where VL is the vector length ($0 \le VL \le 32$). Some major difference in our implementations are:

(1) I vectorized over the primes p_i, while Silverman assigned each p_i to a separate processor.

(2) Silverman did not reduce the coefficients in (3.4.4) modulo N is advance, instead constructing a result approximately N^2 and reducing that modulo N.

Table 9.2.2 compares Step 1 and Step 2 times on a DEC 5000 using (eleven or twelve) values of N ranging from 105 to 152 digits. The IBM RS/6000 is about 10% faster than the DEC 5000 for this application.

Operation	n	Digits in N			
		50	100	150	200
Polynomial product, degrees n and $n-1$	128	0.14	0.32	0.55	0.86
	256	0.34	0.70	1.22	1.83
	512	0.75	1.55	2.64	3.95
	1024	1.62	3.38	5.67	8.46
	2048	3.45	7.27	12.21	17.92
	4096	7.50	15.62	26.01	38.05
	8192	15.97	33.22	54.92	80.27
Polynomial reciprocal, degree n	128	0.30	0.67	1.18	1.82
	256	0.70	1.48	2.58	3.89
	512	1.55	3.31	5.66	8.48
	1024	3.45	7.37	12.39	18.33
	2048	7.70	16.08	26.97	39.49
	4096	16.91	35.05	58.11	84.70
	8192	36.54	75.50	124.49	180.50
Construct polynomial of degree n from roots	128	0.26	0.60	1.06	1.68
	256	0.65	1.49	2.65	4.17
	512	1.62	3.68	6.46	10.12
	1024	3.92	8.86	15.55	24.09
	2048	9.47	21.03	37.21	57.20
	4096	22.34	49.54	85.81	131.36
	8192	52.14	114.49	197.26	301.01
Polynomial GCD, degrees n and $n-1$	128	2.15	4.93	8.68	13.73
	256	5.77	13.00	22.94	36.03
	512	14.70	33.10	58.04	90.79
	1024	36.69	81.70	142.77	222.94
	2048	89.45	198.08	344.09	531.16
	4096	214.91	472.11	816.46	1250.38
	8192	508.46	1109.90	1903.40	2903.61

Table 9.1.1: Times for polynomial operations on RS/6000 (seconds)

Operation	n	Times for 100–digit N (seconds)		
		1 processor	5 processors	Speedup
Polynomial product, degrees n and $n-1$	128	1.61	0.38	4.2
	256	2.91	0.67	4.4
	512	5.59	1.26	4.4
	1024	11.12	2.49	4.5
	2048	22.54	5.07	4.4
	4096	46.34	10.43	4.4
	8192	95.40	21.65	4.4
Polynomial reciprocal, degree n	128	3.47	0.87	4.0
	256	7.01	1.68	4.2
	512	13.73	3.21	4.3
	1024	26.99	6.17	4.4
	2048	54.03	12.26	4.4
	4096	109.43	24.63	4.4
	8192	223.62	50.41	4.4
Construct polynomial of degree n from roots	128	2.80	0.71	3.9
	256	7.19	1.77	4.1
	512	17.28	4.20	4.1
	1024	40.12	9.64	4.1
	2048	91.33	21.73	4.2
	4096	205.15	48.48	4.2
	8192	456.57	107.34	4.3
Polynomial GCD, degrees n and $n-1$	128	24.56	8.69	2.8
	256	66.01	21.45	3.1
	512	167.71	51.20	3.3
	1024	405.81	118.62	3.4
	2048	950.06	268.43	3.5
	4096	2173.50	598.15	3.6
	8192	4893.88	1318.86	3.7

Table 9.1.2: Comparative Alliant FX/80 times with one and five processors

9.2 Performance on RSA Factoring Challenge

RSA Data Security, Inc. announced the RSA Factoring Challenge [36] in March, 1991. The competition features two lists of numbers which contestants attempt to factor. Prizes are awarded quarterly for complete (though not for partial) factorizations.

One list, the RSA Challenge List, has potential keys for the RSA public key cryptosystem [35]. This list's smallest entry has 100 decimal digits, its next has 110 digits, then 120, ... , up to 500 digits. Presumably most or all entries in this list have two prime factors of comparable sizes. As of April, 1992, thirteen months after the contest began, the first two entries of this list had been factored (100 digits by Arjen Lenstra and Mark Manasse, 110 digits by Arjen Lenstra), each using the Quadratic Sieve algorithm. I did not attempt any entries on the RSA Challenge list.

The other list, the Partition Challenge List, has partition numbers $p(n)$ (the number of partitions of n into integer summands without regard to order). For example, $p(5) = 7$ since

$$5 = 1 + 4 = 2 + 3 = 1 + 1 + 3 = 1 + 2 + 2 = 1 + 1 + 1 + 2 = 1 + 1 + 1 + 1 + 1.$$

A table of $p(n)$ values can be computed using the generating function [1, p. 825]

$$\sum_{n=0}^{\infty} p(n) X^n = \prod_{n=1}^{\infty} \frac{1}{1 - X^n} = \frac{1}{\sum\limits_{n=-\infty}^{\infty} (-1)^n X^{(3n^2+n)/2}}.$$

To reduce the number of table entries, the competition uses only prime values of n. This table starts at $p(8681)$, the first such value beyond 10^{99}. Partition numbers appear to have many small prime factors, allowing them to be attacked profitably by ECM. About 30% of the original 1182 entries in the Partition Challenge List were factored completely by some contestant during the first week of the competition, and half had been finished by the end of its second quarter.

During Fall, 1991 (third quarter of the competition), the upper bound of the partition numbers being accepted by the competition was 153 digits. The numbers $p(n)$ for prime $n < 20000$ had appeared in the lists for previous quarters, but eighteen new entries appeared, as listed the first column of Table 9.2.1. I used a DEC 5000 at UCLA (luna) to find as many factors of these as I could, between October 1991 and January 1992. After one day, my old program [29] found the "Easily found factors" in Table 9.2.1. This completely factored six of the eighteen entries (those whose "Status" column begins with a "P", excluding $p(20107)$). Then I tried ECM with FFT on the twelve composite cofactors, whose sizes ranges from 105 to 152 decimal digits. Factors found by the new algorithm are listed separately; each is given a label which identifies its size and which is also used in Table 9.2.2. The "Status" column identifies the number of decimal digits in the cofactor, with "P" for probable prime and "C" for composite.

The FFT runs were done in three rounds, with increasing limits, as identified by the final three columns of Table 9.2.2. Each round had multiple runs, with the same input data but different random number seed. Each run used two curves per input number, both chosen to have a torsion group $\mathbb{Z}/8\mathbb{Z} \times \mathbb{Z}/2\mathbb{Z}$ over $\mathbb{Q}$ (see Chapter 6). After running both curves with with Step 1 limit B_1, Step 2 used $P(X) = X^{24}$ for one curve and the Dickson polynomial $P(X) = g_{12,1}(X)$ for the other curve (see Section 5.10); the choice of which polynomial to use first was decided pseudorandomly. Any one prime factor might be found during Step 1 or Step 2. but would be discovered only once per run even if both group orders were smooth. The search limits approximate those in Table 7.4.1 when searching for a 31–digit, 35–digit, or 38–digit prime factor.

Table 9.2.2 summarizes how often each factor was found in each way. After the first round of 13 runs, the two smallest factors found, of 16 and 17 digits, were removed from the input numbers (reducing the input from 12 composite numbers to 11), but the larger factors were retained to see how often they would be rediscovered.

Excluding the 16– and 17–digit factors, the data shows ten (re-)discoveries using $P(X) = g_{12,1}(X)$, nine using $P(X) = X^{24}$, and one in Step 1 (near 170000). The experimental data suggest that the two choices for $P(X)$ are approximately equally effective, and one should select the whichever is faster or easier to implement. In my implementation the overall Step 2 time was about 5% faster using $P(X) = X^{24}$. As noted in Section 5.10, that choice also parallelizes well since up to d_1 doublings can be done at once, while only k processors can cooperate to evaluate the next $g_{k,\alpha}(N_j) \cdot Q$ in Section 5.8.

Given a large random integer N, the estimated number of prime factors of N with n_1 to n_2 digits (inclusive) is

$$\int_{10^{n_1-1}}^{10^{n_2}} \frac{1}{p} \frac{dp}{\ln p} = \ln(\ln p)\Big|_{10^{n_1-1}}^{10^{n_2}} = \ln(n_2 \ln 10) - \ln((n_1 - 1)\ln 10) = \ln \frac{n_2}{n_1 - 1}.$$

Since there are 18 numbers in this study, the actual number of prime factors in this range should be approximately Poisson distributed with mean (and variance) $\lambda = 18 \ln(n_2/(n_1 - 1))$.

For the ranges 6–10, 11–15, 16–20, 21–25, 26–30 and 31–35 digits, Table 9.2.3 compares the total number of factors found to the expected count. The number found is within one standard deviation of the expected count through 25 digits. Subsequently the number found is more than one standard deviation too small, with only one factor found despite an expected six from 26–35 digits.

$p(n)$	Easily found factors	Factors found by ECM with FFT	Status
$p(20011)$	$2^4 \cdot 7^2$		C150
$p(20021)$	$2^2 \cdot 11 \cdot 12601 \cdot 19571$	p25a $= 11388\,95931\,68795\,51799\,29821$ p25b $= 95559\,95853\,69284\,14467\,61237$	C94
$p(20023)$	$2^5 \cdot 3 \cdot 29 \cdot 4\,05763$ $382\,85011$	p17 $= 12\,75127\,19358\,05687$	C120
$p(20029)$	$3^2 \cdot 5 \cdot 29$ $5810\,98232\,56591$		C136
$p(20047)$	$2 \cdot 3 \cdot 5 \cdot 19 \cdot 643$ $23833 \cdot 2960\,74447$ $7586\,18677$ $1\,17161\,63737$ $12\,16966\,72529$		C105
$p(20051)$	$193 \cdot 3954\,88947\,99047$		C137
$p(20063)$	$2^2 \cdot 71 \cdot 383 \cdot 421$ $1\,34293 \cdot 9\,37235\,68259$		C129
$p(20071)$	$2^2 \cdot 3^2 \cdot 5^3 \cdot 1093$		P146
$p(20089)$	$5 \cdot 13^3 \cdot 37 \cdot 443$	p23 $= 622\,80676\,23197\,85614\,44991$	C122
$p(20101)$	$11 \cdot 3840\,17087$ $9\,11683\,47724\,92833$		P128
$p(20107)$	$3\,84301 \cdot 94564\,80577$	p16 $= 6\,70729\,17733\,03397$	P122
$p(20113)$	$2 \cdot 3 \cdot 7 \cdot 53 \cdot 419$ $1451 \cdot 1435\,62241$		P136
$p(20117)$	$2^3 \cdot 139 \cdot 421$ $98\,07431$	p22 $= 12\,50438\,26612\,16276\,15503$ p26 $= 1\,12457\,67024\,22824\,41059\,90867$	C95
$p(20123)$	7^2		C152
$p(20129)$	$5 \cdot 109$ $16\,26673\,49399\,64211$		P134
$p(20143)$	$2 \cdot 11 \cdot 733$ $29\,76209 \cdot 74\,65313$ $34\,23460\,57843$		P124
$p(20147)$	$2 \cdot 11 \cdot 101 \cdot 25171$ $19\,25681 \cdot 695\,24729$		C132
$p(20149)$	$2^3 \cdot 3 \cdot 5^2 \cdot 7 \cdot 4099$		P146

Table 9.2.1: Some factors of 153–digit partition numbers

B_1		400,000	1,000,000	3,000,000
d_1		2048	4096	8192
d_2		12288	40960	81920
$8d_1d_2$		201,000,000	1,340,000,000	5,370,000,000
Number of runs made		13	12	4
Step 1 time (2 curves)		12 hr	28 hr	87 hr
Step 2 time ($P(X) = X^{24}$)		4 hr	11 hr	23 hr
Step 2 time (Dickson $P(X)$)		4 hr	11 hr	24 hr
Total CPU time/run		20 hr	50 hr	134 hr
Maximum memory		3 megabytes	7 megabytes	14 megabytes
p	p (mod 48)			
p16	5	2 (Step 1) 4 (Dickson) 1 (X^{24})	N.A.	N.A.
p17	23	4 (Step 1) 3 (Dickson) 1 (X^{24})	N.A.	N.A.
p22	31	2 (X^{24})	2 (Dickson) 1 (X^{24})	1 (Dickson) 1 (X^{24})
p23	47		3 (Dickson) 2 (X^{24})	
p25a	13	2 (Dickson) 1 (X^{24})		1 (X^{24})
p25b	37			1 (Step 1) 1 (Dickson) 1 (X^{24})
p26	19			1 (Dickson)

Table 9.2.2: How often factors were found by ECM with FFT

Digits n_1–n_2	Factors found	Expected count: $\lambda = 18 \ln \dfrac{n_2}{n_1 - 1}$	$\dfrac{\text{Found} - \lambda}{\sqrt{\lambda}}$
6–10	14	12.48	$+0.43$
11–15	6	7.30	-0.48
16–20	4	5.18	-0.52
21–25	4	4.02	-0.01
26–30	1	3.29	-1.26
31–35	0	2.77	-1.67

Table 9.2.3: Actual and expected numbers of prime factors, by size

These runs used about $13 \cdot 20 + 12 \cdot 50 + 4 \cdot 134 \approx 1400$ hours, according to Table 9.2.2. After removing known factors below 20 digits, eleven composite cofactors remained, with sizes averaging 135 digits. Approximately 120–130 hours were spent per cofactor. According to Table 7.4.1, this is enough time to find factors up to about 28 digits, if optimal parameters are used. The parameters for the middle column of Table 9.2.2 are close to the optimum, but those in the latter columns are much above the optimum values. Although nothing spectacular was found, the number of findings is only slightly below what might be expected in terms of the effort expended.

9.3 Additional findings

The program was also run on some other parts of the partition table and on the Fibonacci table [12]. Table 9.3.1 lists some additional findings.

N	B_1	Factor found
F_{731}	3,000,000	1 33449 64190 08897 77397 71473
F_{953}	400,000	5228 87989 13069 01009 56159 28073
F_{979}	3,000,000 [S1]	5 67731 16869 88237 05984 10032 04301
$p(9067)$	2,000,000 [S1]	27 53956 39520 89480 76372 83295 84083
$p(10141)$	2,000,000 [S1]	3 34818 44600 44472 51417
$p(13421)$	1,500,000	355 89593 04110 40585 17512 06891 80647
$p(13781)$	400,000	12 98157 14774 71385 16855 08801
$p(13921)$	400,000	5366 98856 20098 57941 60297
$p(15629)$	400,000	46326 47922 03425 53526 02553
$p(17729)$	2,000,000	1719 34650 11405 49531 27187
$p(17737)$	400,000	75 28266 97843 85245 24035 44267
$p(19259)$	400,000	189 34805 77670 39973 51251

[S1] – Identifies factors found during Step 1

Table 9.3.1: Additional factors found

Bibliography

[1] Milton Abramowitz and Irene A. Stegun, *Handbook of mathematical functions with formulas, graphs, and mathematical tables*, Dover Publications, Inc., New York, NY, 1965.

[2] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman, *The design and analysis of computer algorithms*, Addison-Wesley, Reading, MA, 1974.

[3] Selim G. Aki, *The design and analysis of parallel algorithms*, Prentice Hall, Englewood Cliffs, NJ, 1989.

[4] Alliant Computer Systems Corporation, Littleton, MA, *FX/series architecture manual*, April 1988, Part number 300–00001–C.

[5] A. O. L. Atkin and F. Morain, *Finding suitable curves for the elliptic curve method of factorization*, Draft of January 5, 1992; submitted to Mathematics of Computation.

[6] David H. Bailey, *The computation of π to 29,360,000 decimal digits using Borweins' quartically convergent algorithm*, Mathematics of Computation **50** (1988), no. 181, 283–296.

[7] I. Borosh, C. J. Moreno, and H. Porta, *Elliptic curves over finite fields, II*, Mathematics of Computation **29** (1975), no. 131, 951–964.

[8] Murray Bremner, *Elliptic curves with 16 torsion points and positive rank*, AMS Abstracts **10** (1989), no. 4, 306, Abstract 89T–11–157.

[9] Richard P. Brent, *Some integer factorization algorithms using elliptic curves*, Research Report CMA–R32–85, Centre for Mathematical Analysis, The Australian National University, GPO Box 4, Canberra, ACT 2601, Australia, September 1985.

[10] David M. Bressoud, *Factorization and primality testing*, Springer-Verlag, New York, NY, 1989, Undergraduate Texts in Mathematics.

[11] John Brillhart, D. H. Lehmer, J. L. Selfridge, Bryant Tuckerman, and S. S. Wagstaff, Jr., *Factorization of $b^n \pm 1$, $b = 2, 3, 5, 6, 7, 10, 11, 12$ up to high powers*, 2nd ed., Contemporary Mathematics, vol. 22, American Mathematical Society, Providence, RI, 1988.

[12] John Brillhart, Peter L. Montgomery, and Robert D. Silverman, *Tables of Fibonacci and Lucas factorizations*, Mathematics of Computation **50** (1988), no. 181, 251–260 & S1–S15.

[13] J. P. Buhler, H. W. Lenstra, Jr., and Carl Pomerance, *Factoring integers with the number field sieve*, Version 19920507, 1992.

[14] Richard Crandall, *Implementation of the n^k method*, Electronic mail message, January 1991.

[15] Noam D. Elkies, *On $A^4 + B^4 + C^4 = D^4$*, Mathematics of Computation **51** (1988), no. 184, 825–835.

[16] Dale Husemöller, *Elliptic curves*, Graduate Texts in Mathematics, vol. 111, Springer-Verlag, New York, 1987.

[17] Donald E. Knuth, *Seminumerical algorithms*, 2nd ed., The Art of Computer Programming, vol. 2, Addison-Wesley, Reading, MA, 1981.

[18] Daniel Sion Kubert, *Universal bounds on the torsion of elliptic curves*, Proc. London Math. Soc. **33** (1976), 193–237, Third series.

[19] Serge Lang, *Fundamentals of Diophantine geometry*, Springer-Verlag, New York, 1983.

[20] ———, *Algebra*, second ed., Addison-Wesley, Menlo Park, CA, 1984.

[21] A. K. Lenstra, H. W. Lenstra, Jr., M. S. Manasse, and J. M. Pollard, *The number field sieve*, Proceedings of the Twenty Second Annual ACM Symposium on Theory of Computing, Baltimore, May 14–16, 1990 (New York), ACM, 1990, pp. 564–572.

[22] ———, *The factorization of the ninth Fermat number*, February 1991.

[23] Arjen K. Lenstra and Mark S. Manasse, *Factoring by electronic mail*, Advances in Cryptology, EUROCRYPT '89 (J.-J. Quisquater and J. Vandewalle, eds.), Lecture Notes in Computer Science, vol. 434, Springer-Verlag, 1990, pp. 355–371.

[24] H. W. Lenstra, Jr., *Factoring integers with elliptic curves*, Annals of Mathematics **126** (1987), no. 3, 649–673, Second series.

[25] Rudolf Lidl and Harald Niederreiter, *Finite fields*, Encyclopedia of Mathematics and its Applications, vol. 20, Addison-Wesley, Reading, MA, 1983.

[26] Anna Lubiw and András Rácz, *A lower bound for the integer element distinctness problem*, Information and Computation **94** (1991), no. 1, 83–92.

[27] R. T. Moenck, *Fast computation of GCDs*, Proceedings Fifth Annual ACM Symposium on Theory of Computing, Austin, TX, 1973, pp. 142–151.

[28] Peter L. Montgomery, *Modular multiplication without trial division*, Mathematics of Computation **44** (1985), no. 170, 519–521.

[29] ———, *Speeding the Pollard and elliptic curve methods of factorization*, Mathematics of Computation **48** (1987), no. 177, 243–264.

[30] ———, *Design of an FFT continuation to the ECM method of factorization*, AMS Abstracts **10** (1989), no. 4, 278, Abstract 850–11–25.

[31] ———, *Evaluating recurrences of form $x_{m+n} = f(x_m, x_n, x_{m-n})$ via Lucas chains*, To be submitted to Fibonacci Quarterly, January 1992.

[32] Peter L. Montgomery and Robert D. Silverman, *An FFT extension to the $P-1$ factoring algorithm*, Mathematics of Computation **54** (1990), no. 190, 839–854.

[33] François Morain, *Courbes elliptiques et tests de primalité*, Ph.D. thesis, L'Université Claude Bernard, Lyon I, September 1990, Introduction in French, body in English.

[34] Carl Pomerance, *The quadratic sieve factoring algorithm*, Advances in Cryptology, Proceedings of EUROCRYPT 84 (New York) (T. Beth, N. Cot, and I. Ingemarsson, eds.), Lecture Notes in Computer Science, vol. 209, Springer-Verlag, 1985, pp. 169–182.

[35] R. L. Rivest, A. Shamir, and L. Adleman, *A method for obtaining digital signatures and public-key cryptosystems*, CACM **21** (1978), no. 2, 120–126.

[36] RSA Challenge Administrator, *Information about RSA Factoring Challenge*, March 1991, Send electronic mail to challenge-info@rsa.com.

[37] J. T. Schwartz, *Fast probabilistic algorithms for verification of polynomial identities*, JACM **27** (1980), no. 4, 701–717.

[38] Joseph H. Silverman, *The arithmetic of elliptic curves*, Graduate Texts in Mathematics, vol. 106, Springer-Verlag, New York, 1986.

[39] Robert D. Silverman, *The multiple polynomial quadratic sieve*, Mathematics of Computation **48** (1987), no. 177, 329–339.