UNIVERSITY OF CALGARY

Improved Exponentiation in the Ideal Class Group of Imaginary Quadratic Number Fields

With an Application to Integer Factoring

by

Maxwell Sayles

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA

June, 2013

Abstract

Paragraph 1

Paragraph 2

Paragragh 3

Acknowledgements

Paragraph 1

Paragraph 2

Paragraph 3

Table of Contents

List of Tables

List of Figures and Illustrations

List of Algorithms

Chapter 1

Motivation

1.1 Faster ideal exponentiation on imaginary quadratic fields

1.2 Exponentiation with fixed exponents

Remember to list all the contributions in the exponentiation section: the greedy pruned extensions, L best approximations, the ± 16-bit chains, and the search method for chains.

1.3 Faster class group computations

1.4 Applications in cryptography

1.5 Examples of fast ideal exponentiation in SuperSPAR

1.6 Contributions

Major contributions

1. A really fast implementation of the Extended Euclidean Algorithm for integers bound by 32, 64, and 128 bits.

2. Optimized implementations of ideal class arithmetic for discriminants bound by 64 and 128 bits.

3. An improvement in the wall clock running time to exponentiate in the ideal class group when the exponent is known in advance.

4. The fastest implementation that we tested for integer factoring in the range of 54bit integers to 62bit integers.

5. A library for optimized 32-bit, 64-bit, and 128-bit arithmetic is available at `https://www.github.com/maxwellsayles/liboptarith`

6. A library for arithmetic in the class group of imaginary quadratic number fields, specialized for 64-bit, 128-bit, and unbound discriminants is available at `https://www.github.com/maxwellsayles/libqform`

7. An integer factoring library (SuperSPAR) suitable for non-cryptographic sizes is available at `https://www.github.com/maxwellsayles/libsspar` (COMING SOON)

Minor contributions: Algorithm to compute a list of candidate search bounds, binary search on 2d values, some types of exponentiation, simplified left-to-right binary XGCD.

The software was developed using the GNU C compiler version 4.7.2 on Ubuntu 12.10. The hardware platform was a 2.7GHz Intel Core i7-2620M CPU with 8Gb of memory. The CPU has four cores, only one of which was used during timing experiments.

1.7 Overview of Thesis

Chapter 2

Ideal Arithmetic

A focus of this thesis is arithmetic and exponentiation in the ideal class group of imaginary quadratic number fields. We begin with with the relevant theory of quadratic number fields, then discuss quadratic orders and ideals of quadratic orders. Finally, we discuss arithmetic in the ideal class group. The theory presented here is available in detail in reference texts on algebraic number theory such as [16], [28], or [31].

2.1 Quadratic Number Fields

A quadratic number field $\mathbb{K}$ is an algebraic number field of degree 2 over the rational numbers $\mathbb{Q}$, and is defined as

$$\mathbb{K} = \mathbb{Q}(\alpha) = \{u + v\alpha : u, v \in \mathbb{Q}\}$$

for some quadratic irrational $\alpha \in \mathbb{C}$. Following Jacobson and Williams [34, p.77], let α be a root of a polynomial $f(x) = ax^2 + bx + c$ with integer coefficients and $f(x)$ is irreducible over the rationals. As such

$$\alpha = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

The *discriminant* of the polynomial $f(x)$ is $\Delta = b^2 - 4ac$, and our number field $\mathbb{K} = \mathbb{Q}(\alpha) = \mathbb{Q}(\sqrt{\Delta})$. Notice that $\Delta \equiv 0, 1 \pmod 4$ and $\sqrt{\Delta} \notin \mathbb{Q}$ since $f(x)$ is irreducible. Also, if $\Delta = f^2\Delta_0$ where Δ_0 is square free, then $\mathbb{Q}(\sqrt{\Delta}) = \mathbb{Q}(\sqrt{\Delta_0})$. When Δ or $\Delta/4$ is square free then Δ is a *fundamental discriminant.* In the case that Δ is positive, $\mathbb{K}$ is a subset of the real numbers, $\mathbb{R}$, and is a *real* quadratic number field. When Δ is negative, $\mathbb{K}$ is a subset of the complex numbers, $\mathbb{C}$, and is an *imaginary* quadratic number field. Throughout this thesis, the assumption is that Δ is negative.

2.2 Quadratic Integers

A polynomial with a leading coefficient of 1 is a *monic* polynomial, and the root α of a monic polynomial $f(x)$ with integer coefficients is an *algebraic integer.* The rational numbers are degree 1 algebraic numbers since they are roots of degree 1 polynomials $bx - a$, and the roots of monic degree 1 polynomials with integer coefficients are the integers, $\mathbb{Z}$, sometimes called the *rational integers.* When $f(x)$ is a monic quadratic polynomial with integer coefficients, the root α is a *quadratic integer.*

Theorem 2.2.1. [34, Theorem 4.10] A quadratic integer α is an algebraic integer of $\mathbb{Q}(\sqrt{\Delta_0})$ where α can be written as $\alpha = x + y\omega_0$ for $x, y \in \mathbb{Z}$ where

$$\omega_0 = \begin{cases} \sqrt{\Delta_0} & \text{when } \Delta_0 \not\equiv 1 \pmod 4 \\ \frac{1+\sqrt{\Delta_0}}{2} & \text{when } \Delta_0 \equiv 1 \pmod 4. \end{cases}$$

2.3 Maximal Order of Algebraic Integers

The *maximal order* of a field $\mathbb{K}$ is the set of all algebraic integers contained within $\mathbb{K}$ and is characterized using a $\mathbb{Z}$-module.

Definition 2.3.1. Let $X = \{\xi_1, \xi_2, \xi_3, ..., \xi_n\}$ be a subset of a number field $\mathbb{K}$. In this case, a $\mathbb{Z}$-*module*, $\mathcal{M}$, is an additive Abelian group such that

$$\begin{aligned} \mathcal{M} &= [\xi_1, \xi_2, ..., \xi_n] \\ &= \xi_1\mathbb{Z} + \xi_2\mathbb{Z} + \cdots + \xi_n\mathbb{Z} \\ &= \left\{ \sum_i^n x_i\xi_i : x_i \in \mathbb{Z}, \xi_i \in X \right\}. \end{aligned}$$

Definition 2.3.2. A *quadratic order* $\mathcal{O}$ of $\mathbb{Q}(\sqrt{\Delta})$ is a sub-ring of the quadratic integers of $\mathbb{Q}(\sqrt{\Delta})$ containing 1. Following Jacobson and Williams [34, p.81], we write $\mathcal{O}$ as

$$\left[1, \frac{\Delta + \sqrt{\Delta}}{2}\right] = [1, f\omega_0].$$

The maximal order $\mathcal{O}_\Delta = [1, \omega_0]$ of $\mathbb{Q}(\sqrt{\Delta})$ is the ring of all quadratic integers in $\mathbb{Q}(\sqrt{\Delta})$, and is maximal since any order $\mathcal{O} = [1, f\omega_0]$ is a sub-ring of $\mathcal{O}_\Delta$.

2.4 Ideals of $\mathcal{O}_\Delta$

Definition 2.4.1. An *ideal* $\mathfrak{a}$ is an additive subgroup of an order $\mathcal{O}$ with the property that for any $a \in \mathfrak{a}$ and $\xi \in \mathcal{O}$, it holds that ξa and $a\xi$ are both elements of the ideal $\mathfrak{a}$.

When $\mathcal{O}_\Delta$ is a maximal order, for $\alpha, \beta \in \mathcal{O}_\Delta$, the set $\mathfrak{a} = \{x\alpha + y\beta : x, y \in \mathcal{O}_\Delta\}$ is an ideal in the order $\mathcal{O}_\Delta$ and is denoted (α, β) [43, p.16]. Every ideal of a quadratic order $\mathcal{O}_\Delta$ can be represented by at most two generators [16, p.125–126, § 10], while some can be represented by a single generator. An ideal represented by a single generator is a *principal* ideal and is denoted $(\alpha) = \{x\alpha : x \in \mathcal{O}_\Delta\}$ [34, p.87].

For two ideals, $\mathfrak{a} = (\alpha_1, \beta_1)$ and $\mathfrak{b} = (\alpha_2, \beta_2)$ in $\mathcal{O}_\Delta$, their product is

$$\mathfrak{ab} = (\alpha_1\alpha_2, \alpha_1\beta_2, \beta_1\alpha_2, \beta_1\beta_2) = (\alpha_3, \beta_3) \tag{2.1}$$

for some $\alpha_3, \beta_3 \in \mathcal{O}_\Delta$ and is also an ideal in $\mathcal{O}_\Delta$. The principal ideal $(1) = \mathcal{O}_\Delta$ is the *identity* ideal since $\mathfrak{a} = \mathfrak{a}\mathcal{O}_\Delta = \mathcal{O}_\Delta\mathfrak{a}$. If there exists an ideal $\mathfrak{c}$ such that $\mathfrak{ac} = \mathfrak{b}$, then $\mathfrak{a}$ *divides* $\mathfrak{b}$ and we write $\mathfrak{a} \mid \mathfrak{b}$. If the ideal $\mathfrak{a}$ divides the identity ideal $\mathcal{O}_\Delta$, then $\mathfrak{a}$ has an *inverse*, which is denoted $\mathfrak{a}^{-1}$. Finally, an ideal $\mathfrak{p} \neq (1)$ is *prime* when $\mathfrak{p} \mid \mathfrak{ab}$ implies that either $\mathfrak{p} \mid \mathfrak{a}$ or $\mathfrak{p} \mid \mathfrak{b}$. As such, a prime ideal is divisible only by the identity ideal and itself.

Theorem 2.4.2. [32, p.13] When $\mathcal{O}_\Delta = [1, f\omega_0]$ is an order of a quadratic field, a non-zero ideal $\mathfrak{a}$ of $\mathcal{O}_\Delta$ can be uniquely written as a two dimensional $\mathbb{Z}$-module

$$\mathfrak{a} = s\left[a, \frac{b + \sqrt{\Delta}}{2}\right]$$

for $a, b \in \mathbb{Z}, s > 0, a > 0$, $\gcd(a, b, (b^2 - \Delta)/4a) = 1$, $b^2 \equiv \Delta \pmod{4a}$, and b is unique mod $2a$.

Definition 2.4.3. For an ideal $\mathfrak{a} = s[a, (b+\sqrt{\Delta})/2]$, when $s \in \mathbb{Z}$, $\mathfrak{a}$ is an *integral* ideal, and when $s \in \mathbb{Q}$, $\mathfrak{a}$ is a *fractional* ideal. Finally, when $s = 1$, $\mathfrak{a}$ is a *primitive* ideal.

For a prime ideal $\mathfrak{p} \in \mathcal{O}_\Delta$ it can be shown [32, p.19] that $\mathfrak{p} \cap \mathbb{Z} = p\mathbb{Z}$ for some prime integer $p \in \mathbb{Z}$. Let

$$\mathfrak{p} = s\left[a, \frac{b+\sqrt{\Delta}}{2}\right]$$

and it follows that either $s = p$ and $a = 1$, or $s = 1$ and $a = p$. In the first case $\mathfrak{p} = p\mathcal{O}_\Delta$, while in the second case, if there exists $b = \sqrt{\Delta} \pmod{4p}$, then $\mathfrak{p} = [p, (b+\sqrt{\Delta})/2]$. This follows since $\Delta = b^2 - 4ac$, $a = p$, and $b, c \in \mathbb{Z}$. As such $c = (b^2 - \Delta)/4p$ and $b^2 \equiv \Delta \pmod{4p}$.

The inverse of an ideal $\mathfrak{a} = s[a, (b+\sqrt{\Delta})/2]$ with $\gcd(a, b, (b^2-\Delta)/4a) = 1$ is given by [32, pp.14–15]

$$\mathfrak{a}^{-1} = \frac{s}{\mathcal{N}(\mathfrak{a})}\left[a, \frac{-b+\sqrt{\Delta}}{2}\right] \tag{2.2}$$

where $\mathcal{N}(\mathfrak{a}) = s^2 a$ is the norm of $\mathfrak{a}$ and is multiplicative. Notice that the resulting ideal $\mathfrak{a}^{-1}$ may be a fractional ideal. When $\mathcal{O}_\Delta$ is maximal, all ideals of $\mathcal{O}_\Delta$ have inverses, and the set of invertible ideals forms a group under ideal multiplication.

2.5 Ideal Class Group

Two ideals $\mathfrak{a}$ and $\mathfrak{b}$ are *equivalent* if there exists $\alpha, \beta \in \mathcal{O}_\Delta$ such that $\alpha\beta \neq 0$ and $(\alpha)\mathfrak{a} = (\beta)\mathfrak{b}$ [34, p.88]. We use $[\mathfrak{a}]$ to denote the *ideal class* of all ideals equivalent to the *representative* ideal $\mathfrak{a}$. The *ideal class group*, Cl_Δ, is the set of all equivalence classes of invertible ideals, with the group operation defined as the product of class representatives. By [16, p.136], the ideal class group is a finite Abelian group.

Our implementation uses the primitive ideal $\mathfrak{a} = [a, (b+\sqrt{\Delta})/2]$ as a representative for the ideal class $[\mathfrak{a}]$. Additionally, we maintain the value $c = (b^2 - \Delta)/4a$. The class group is represented by the discriminant Δ. Since an ideal class contains an infinitude of ideals, we work with reduced representatives. This also makes arithmetic faster, since the size of generators are typically smaller.

Subsection 2.5.1 defines the reduced form of a representative and gives an algorithm for finding the reduced form. For two reduced class representatives, Subsection 2.5.4 shows how to compute their product. Subsection 2.5.5 discusses how to perform multiplication such that the result is a reduced or almost reduced representative. This is then extended to the case of computing the square (Subsection 2.5.6) and cube (Subsection 2.5.7) of an ideal class.

2.5.1 Reduced Representatives

Definition 2.5.1. A primitive ideal $\mathfrak{a} = [a, (b + \sqrt{\Delta})/2]$ with $\Delta < 0$ is a *reduced* representative of the ideal class $[\mathfrak{a}]$ when $-a < b \leq a < c$ or when $0 \leq b \leq a = c$ for $c = (b^2 - \Delta)/4a$ [19, p.241]. Often, we refer to the ideal $\mathfrak{a}$ simply as being *reduced*.

Algorithm 2.1 Ideal Reduction ([33, p.90]).

Input: An ideal class representative $\mathfrak{a}_1 = [a_1, (b_1 + \sqrt{\Delta})/2]$ and $c_1 = ({b_1}^2 - \Delta)/4a_1$.
Output: A reduced representative $\mathfrak{a} = [a, (b + \sqrt{\Delta})/2]$.

```
$(a, b, c) \leftarrow (a_1, b_1, c_1)$
while $a > c$ or $b > a$ or $b \leq -a$ do
    if $a > c$ then
        swap $a$ with $c$ and let $b \leftarrow -b$
    if $b > a$ or $b \leq -a$ then
        $b \leftarrow b'$ such that $-a < b' \leq a$ and $b' \equiv b \pmod{2a}$
        $c \leftarrow (b^2 - \Delta)/4a$
if $a = c$ and $b < 0$ then
    $b \leftarrow -b$
return $[a, (b + \sqrt{\Delta})/2]$
```

In an imaginary quadratic field, every ideal class contains exactly one reduced ideal [43, p.20]. There are several algorithms to compute a reduced ideal, many of which are listed in [33]. Here we present the algorithm we use. We adapt the work presented in [33, p.90] and [34, p.99]. If $\mathfrak{a} = [a, (b + \sqrt{\Delta})/2]$ is a representative of the class $[\mathfrak{a}]$ then

$$\mathfrak{b} = \left[-\mathcal{N}((b + \sqrt{\Delta})/2)/a, -(b - \sqrt{\Delta})/2\right] \tag{2.3}$$

is equivalent, which can be verified by

$$\left(-(b - \sqrt{\Delta})/2\right) \mathfrak{a} = (a)\mathfrak{b}.$$

Simplifying Equation 2.3 gives

$$\mathfrak{b} = \left[\frac{b^2 - \Delta}{4a}, \frac{-b + \sqrt{\Delta}}{2}\right].$$

Since $c = (b^2 - \Delta)/4a$ we have

$$\mathfrak{b} = \left[c, \frac{-b + \sqrt{\Delta}}{2}\right]. \tag{2.4}$$

As such, the first step is to reduce a by setting $\mathfrak{a} = [c, (-b + \sqrt{\Delta})/2]$, if $a > c$. Then b is reduced since b is unique modulo $2a$. These steps are repeated while $\mathfrak{a}$ is not reduced. In the case that $a = c$, we use the absolute value of b, since by Equation 2.4 the ideals $[a, (b + \sqrt{\Delta})/2]$ and $[c, (-b + \sqrt{\Delta})/2]$ are equivalent. Algorithm 2.1 gives Pseudo-code.

2.5.2 Class Number

Using $|b| \leq a$ from the definition of a reduced representative 2.5.1, $-\Delta = 4ac - b^2 \geq 4ac - a^2$, and using $a \leq c$, it follows that $|b| \leq a \leq \sqrt{|\Delta|/3}$. Since a and b are bounded and c is determined by a, b, and Δ, it follows that the number of ideal classes in the class group Cl_Δ is finite [34, p.153].

Definition 2.5.2. The number of ideal classes in the class group Cl_Δ is the *class number* and is denoted h_Δ [34, p.153].

Cohen [19, p.247] states a bound on the number of elements in the class group Cl_Δ as

$$h_\Delta < \frac{1}{\pi}\sqrt{|\Delta|}\log|\Delta| \text{ when } \Delta < -4, \tag{2.5}$$

and C. Siegel shows that $h_\Delta = |\Delta|^{1/2+o(1)}$ as $\Delta \to -\infty$ [19, p.247].

2.5.3 Inverse of Ideal Class

Recall from Equation 2.2 that the inverse of an ideal $\mathfrak{a} = [a, (b + \sqrt{\Delta})/2]$ is the (possibly fractional) ideal

$$\mathfrak{a}^{-1} = \frac{s}{\mathcal{N}(\mathfrak{a})}\left[a, \frac{-b + \sqrt{\Delta}}{2}\right].$$

As such, the ideals $\mathfrak{a}^{-1}$ and $[a, (-b+\sqrt{\Delta})/2]$ are equivalent since

$$(\mathcal{N}(\mathfrak{a}))\,\mathfrak{a}^{-1} = (s)\left[a, \frac{-b+\sqrt{\Delta}}{2}\right]$$

and both $s, \mathcal{N}(\mathfrak{a}) \in \mathcal{O}_\Delta$. The inverse of an ideal class $[\mathfrak{a}]$ for the representative $\mathfrak{a} = [a, (b+\sqrt{\Delta})/2]$ is then given by the representative $[a, (-b+\sqrt{\Delta})/2]$ and by Definition 2.5.1 is reduced. As such, computing the inverse of an ideal class is virtually free.

2.5.4 Ideal Class Multiplication

The ideal class group operation is multiplication of ideal class representatives. Given two representative ideals $\mathfrak{a} = [a_1, (b_1+\sqrt{\Delta})/2]$ and $\mathfrak{b} = [a_2, (b_2+\sqrt{\Delta})/2]$ in reduced form, the (non-reduced) product $\mathfrak{ab}$ is computed using

$$\begin{aligned}
c_2 &= ({b_2}^2 - \Delta)/4a_2, \\
s &= \gcd(a_1, a_2, (b_1+b_2)/2) = Ya_1 + Va_2 + W(b_1+b_2)/2, && (2.6)\\
U &= (V(b_1-b_2)/2 - Wc_2) \bmod (a_1/s), && (2.7)\\
a &= (a_1a_2)/s^2, && (2.8)\\
b &= (b_2 + 2Ua_2/s) \bmod 2a, && (2.9)\\
\mathfrak{ab} &= s\left[a, \frac{b+\sqrt{\Delta}}{2}\right].
\end{aligned}$$

The remainder of this subsection is used to derive the above equations. We adapt much of the presentation given in [34, pp.117–118]. Equation 2.1 for the product of two ideals, $\mathfrak{a}$ and $\mathfrak{b}$, using module notation is

$$\mathfrak{ab} = \left[a_1a_2, \frac{a_1b_2 + a_1\sqrt{\Delta}}{2}, \frac{a_2b_1 + a_2\sqrt{\Delta}}{2}, \frac{b_1b_2 + (b_1+b_2)\sqrt{\Delta} + \Delta}{4}\right]. \qquad (2.10)$$

By the multiplicative property of the norm we have

$$\begin{aligned}
N(\mathfrak{ab}) = s^2a &= N(\mathfrak{a})N(\mathfrak{b}) = a_1a_2 \\
\Rightarrow\ a &= \frac{a_1a_2}{s^2},
\end{aligned}$$

which gives Equation 2.8. Now, by the second term of equation (2.10) we know that $(a_1b_2 + a_1\sqrt{\Delta})/2 \in \mathfrak{ab}$. It follows that there is some $x, y \in \mathbb{Z}$ such that

$$\frac{a_1b_2 + a_1\sqrt{\Delta}}{2} = xsa + ys\left(\frac{b+\sqrt{\Delta}}{2}\right).$$

Equating irrational parts gives

$$\frac{a_1\sqrt{\Delta}}{2} = \frac{ys\sqrt{\Delta}}{2}.$$

Hence, $s \mid a_1$. Similarly, by the third and fourth terms of equation (2.10) we have $(a_2b_1 + a_2\sqrt{\Delta})/2 \in \mathfrak{ab}$, which implies that $s \mid a_2$, and $(b_1b_2+(b_1+b_2)\sqrt{\Delta}+\Delta)/4 \in \mathfrak{ab}$, which implies that $s \mid (b_1 + b_2)/2$.

By the second generator, $s(b + \sqrt{\Delta})/2$, of $\mathfrak{ab}$ and the entire right hand side of equation (2.10) there exists $X, Y, V, W \in \mathbb{Z}$ such that

$$\frac{sb + s\sqrt{\Delta}}{2} = Xa_1a_2 + Y\frac{a_1b_2 + a_1\sqrt{\Delta}}{2} + V\frac{a_2b_1 + a_2\sqrt{\Delta}}{2} + W\frac{b_1b_2 + (b_1+b_2)\sqrt{\Delta} + \Delta}{4}.$$

Grouping rational and irrational parts gives

$$\frac{sb + s\sqrt{\Delta}}{2} = \left(Xa_1a_2 + Y\frac{a_1b_2}{2} + V\frac{a_2b_1}{2} + W\frac{b_1b_2+\Delta}{4}\right) + \left(Y\frac{a_1}{2} + V\frac{a_2}{2} + W\frac{b_1+b_2}{4}\right)\sqrt{\Delta}. \tag{2.11}$$

Again, by equating irrational parts we have

$$\begin{aligned}\frac{s\sqrt{\Delta}}{2} &= \left(Y\frac{a_1}{2} + V\frac{a_2}{2} + W\frac{b_1+b_2}{4}\right)\sqrt{\Delta} \\ s &= Ya_1 + Va_2 + W\frac{b_1+b_2}{2},\end{aligned} \tag{2.12}$$

which is the same as Equation 2.6. Since s divides each of a_1, a_2, and $(b_1 + b_2)/2$, take $s = \gcd(a_1, a_2, (b_1 + b_2)/2)$ to be the largest such common divisor.

It remains to compute $b \pmod{2a}$. Recall that $a = a_1a_2/s^2$. This time, by equating the rational parts of (2.11) we have

$$\begin{aligned}\frac{sb}{2} &= Xa_1a_2 + Y\frac{a_1b_2}{2} + V\frac{a_2b_1}{2} + W\frac{b_1b_2+\Delta}{4} \\ b &= 2X\frac{a_1a_2}{s} + Y\frac{a_1b_2}{s} + V\frac{a_2b_1}{s} + W\frac{b_1b_2+\Delta}{2s} \\ b &\equiv Y\frac{a_1b_2}{s} + V\frac{a_2b_1}{s} + W\frac{b_1b_2+\Delta}{2s} \pmod{2a}.\end{aligned} \tag{2.13}$$

This gives b. However, Equation (2.13) can be rewritten with fewer multiplies and divides. Equation (2.12) gives

$$\begin{aligned} s &= Ya_1 + Va_2 + W\frac{b_1+b_2}{2} \\ 1 &= Y\frac{a_1}{s} + V\frac{a_2}{s} + W\frac{b_1+b_2}{2s} \\ Y\frac{a_1}{s} &= 1 - V\frac{a_2}{s} - W\frac{b_1+b_2}{2s}. \end{aligned}$$

Then substituting into Equation (2.13) gives

$$\begin{aligned} b &\equiv b_2(1 - V\frac{a_2}{s} - W\frac{b_1+b_2}{2s}) + V\frac{a_2b_1}{s} + W\frac{b_1b_2+\Delta}{2s} && \pmod{2a} \\ &\equiv b_2 - V\frac{a_2b_2}{s} - W\frac{b_1b_2+{b_2}^2}{2s} + V\frac{a_2b_1}{s} + W\frac{b_1b_2+\Delta}{2s} && \pmod{2a} \\ &\equiv b_2 + V\frac{a_2(b_1-b_2)}{s} + W\frac{\Delta-{b_2}^2}{2s} && \pmod{2a} \\ &\equiv b_2 + V\frac{2a_2(b_1-b_2)}{2s} + W\frac{2a_2(\Delta-{b_2}^2)}{2a_2\cdot 2s} && \pmod{2a} \\ &\equiv b_2 + \frac{2a_2}{s}\left(V\frac{b_1-b_2}{2} + W\frac{\Delta-{b_2}^2}{4a_2}\right) && \pmod{2a}. \end{aligned}$$

Let $c_2 = ({b_2}^2 - \Delta)/4a_2$ and $U = (V(b_1-b_2)/2 - Wc_2) \bmod (a_1/s)$ and we have

$$b \equiv b_2 + \frac{2a_2}{s}U \pmod{2a},$$

which completes the derivation of Equation 2.9. Note that the product ideal $\mathfrak{ab}$ is not reduced and that the size of its coefficients can be as much as twice that of the ideal factors $\mathfrak{a}$ and $\mathfrak{b}$ [34, p.118].

2.5.5 Fast Ideal Multiplication (NUCOMP)

Shanks [48] gives an algorithm for multiplying two ideal class representatives such that their product is reduced or almost reduced. The algorithm is known as NUCOMP and stands for "New COMPosition". This algorithm is often faster in practice as the intermediate numbers are smaller and the final product requires at most two applications of the reduction operation to be converted to reduced form [34, pp.439–441]. The description of NUCOMP provided here is a high level description of the algorithm based on Jacobson and Williams [34, §5.4, pp. 119-123].

Equations 2.6, 2.8, and 2.9 from the previous subsection give a solution to the ideal product $\mathfrak{ab} = s[a, (b+\sqrt{\Delta})/2]$. In this case, the product ideal is not necessarily reduced. Computing the reduced ideal class representative corresponds to computing the simple continued fraction expansion of $(b/2)/a$ [34, p.119], but in this case, the coefficients a and b may be as large as Δ [34, p.118]. Instead of computing the simple continued fraction expansion of $(b/2)/a$, Jacobson and Williams [34, p.119] use sU/a_1 where U is given by Equation 2.7. To see why this is sufficient, recall from Subsection 2.5.2 that a_1 is approximately $\sqrt{|\Delta|}$ in size, so $s^2/2a_1 \approx 1/\sqrt{|\Delta|}$ and

$$\frac{b}{2a} = \frac{b_2 + 2Ua_2/s}{2a_1a_2/s^2} = \frac{s^2b_2 + s2Ua_2}{2a_1a_2} = \frac{s^2b_2}{2a_1a_2} + \frac{sU}{a_1} \approx \frac{sU}{a_1}.$$

Following [34, pp.120-121], we develop the simple continued fraction expansion of $sU/a_1 = \langle q_0, q_1, \ldots, q_i, \phi_{i+1}\rangle$ using the recurrences

$$q_i = \lfloor R_{i-2} \ / \ R_{i-1} \rfloor \tag{2.14}$$

$$R_i = R_{i-2} - q_i R_{i-1} \tag{2.15}$$

$$C_i = C_{i-2} - q_i C_{i-1} \tag{2.16}$$

until we have R_i and R_{i-1} such that

$$R_i < \sqrt{a_1/a_2}\ |\Delta/4|^{1/4} < R_{i-1}. \tag{2.17}$$

Initial values for the recurrence are given by

$$\begin{bmatrix} R_{-2} & C_{-2} \\ R_{-1} & C_{-1} \end{bmatrix} = \begin{bmatrix} sU & -1 \\ a_1 & 0 \end{bmatrix}.$$

We then compute

$$
\begin{aligned}
M_1 &= \frac{R_i a_2 + sC_i(b_1 - b_2)/2}{a_1}, \\
M_2 &= \frac{R_i(b_1 + b_2)/2 - sC_i c_2}{a_1}, \\
a &= (-1)^{i+1}(R_i M_1 - C_i M_2), \\
b &= \left(\frac{2(R_i a_2/s - C_{i-1} a)}{C_i} - b_2 \right) \bmod 2a
\end{aligned}
\tag{2.18}
$$

for the product $\mathfrak{ab} = [a, (b + \sqrt{\Delta})/2]$ where $\mathfrak{ab}$ is at most two steps from being reduced. Note that this procedure assumes that $\mathcal{N}(\mathfrak{a}) \geq \mathcal{N}(\mathfrak{b})$ and that if $a_1 < \sqrt{a_1/a_2}\ |\Delta/4|^{1/4}$ then R_{-1} and R_{-2} satisfy Equation 2.17 and we compute the product $\mathfrak{ab}$ as in the previous subsection without expanding the simple continued fraction sU/a_1. When $a_1 \geq \sqrt{a_1/a_2}\ |\Delta/4|^{1/4}$, at least one iteration of the recurrence 2.16 is performed and so $C_i \neq 0$ and there will not be a division by zero in Equation 2.18.

Our implementation of fast ideal multiplication includes many practical optimizations developed by Imbert, Jacobson, and Schmidt [29, Algorithm 6]. For example, by Equation 2.6, $s \mid a_1$ and $s \mid a_2$, and so after computing s, we use ${a_1}' = a_1/s$ and ${a_2}' = a_2/s$ throughout. Furthermore, they separate Equation 2.6 into the following computations. Let

$$
s' = \gcd(a_1, a_2) = Y' a_1 + V' a_2 \tag{2.19}
$$

$$
s = \gcd(s', (b_1 + b_2)/2) = V'' s' + W(b_1 + b_2)/2. \tag{2.20}
$$

Only when $s' \neq 1$ do we compute Equation 2.20, in which case $V = V'V''$. When $s' = 1$, we do not compute Equation 2.20 and instead set $s = 1$, $V = V'$, and $W = 0$ (which simplifies Equation 2.7). Also notice that Y from Equation 2.6 (and correspondingly Y' from Equation 2.19) is not used throughout the ideal product calculation. As such, we do not compute this coefficient when computing Equation 2.19.

Chapter 5 discusses practical optimizations for computing the extended GCD used in Equations 2.19 and 2.20, as well as the simple continued fraction expansion of sU/a_1, which is essentially the extended GCD computation [34, §3.2]. Chapter 6 gives a more thorough treatment of our implementation of ideal class arithmetic. Pseudo-code for the approach described here is given in Algorithm 2.2.

Algorithm 2.2 NUCOMP – Fast Ideal Multiplication ([34, pp.441-443]).

```
Input: Reduced representatives $\mathfrak{a} = [a_1, (b_1 + \sqrt{\Delta})/2]$, $\mathfrak{b} = [a_2, (b_2 + \sqrt{\Delta})/2]$
    with $c_1 = ({b_1}^2 - \Delta)/4a_1$, $c_2 = ({b_2}^2 - \Delta)/4a_2$, and discriminant $\Delta$.
Output: A reduced or almost reduced representative $\mathfrak{ab}$.
ensure $\mathcal{N}(\mathfrak{a}) < \mathcal{N}(\mathfrak{b})$ by swapping $\mathfrak{a}$ with $\mathfrak{b}$ if $a_1 < a_2$
compute $s', V' \in \mathbb{Z}$ such that $s' = \gcd(a_1, a_2) = Y'a_1 + V'a_2$ for $Y' \in \mathbb{Z}$
$s \leftarrow 1$
$U \leftarrow V'(b_1 - b_2)/2 \bmod a_1$
if $s' \neq 1$ then
    compute $s, V, W \in \mathbb{Z}$ such that $s = \gcd(s', (b_1 + b_2)/2) = Vs' + W(b_1 + b_2)/2$
    $(a_1, a_2) \leftarrow (a_1/s, a_2/s)$
    $U \leftarrow (VU - Wc_2) \bmod a_1$
if $a_1 < \sqrt{a_1/a_2}\ |\Delta/4|^{1/4}$ then
    $a \leftarrow a_1 a_2$
    $b \leftarrow (2a_2U + b_2) \bmod 2a$
    return $[a, (b + \sqrt{\Delta})/2]$
$\begin{bmatrix} R_{-2} & C_{-2} \\ R_{-1} & C_{-1} \end{bmatrix} \leftarrow \begin{bmatrix} U & -1 \\ a_1 & 0 \end{bmatrix}$
$i \leftarrow -1$
while $R_i > \sqrt{a_1/a_2}\ |\Delta/4|^{1/4}$ do
    $i \leftarrow i + 1$
    $q_i = \lfloor R_{i-2}\ /\ R_{i-1} \rfloor$
    $R_i = R_{i-2} - q_i R_{i-1}$
    $C_i = C_{i-2} - q_i C_{i-1}$
$M_1 \leftarrow (R_i a_2 + C_i(b_1 - b_2)/2)/a_1$
$M_2 \leftarrow (R_i(b_1 + b_2)/2 - sC_i c_2)/a_1$
$a \leftarrow (-1)^{i+1}(R_i M_1 - C_i M_2)$
$b \leftarrow ((2(R_i a_2 - C_{i-1}a)/C_i) - b_2) \bmod 2a$
return $[a, (b + \sqrt{\Delta})/2]$
```

2.5.6 Fast Ideal Squaring (NUDUPL)

When the two input ideals for multiplication are the same, as is the case when squaring, much of the arithmetic simplifies. In this case, $a_1 = a_2$, $b_1 = b_2$, and Equations 2.6 and 2.7

simplify to

$$s = \gcd(a_1, b_1) = Xa_1 + Yb_1$$

$$U = -Yc_1 \bmod (a_1/s).$$

One then computes the continued fraction expansion of sU/a_1, but using the bound

$$R_i < |\Delta/4|^{1/4} < R_{i-1}.$$

Computing the ideal class representative simplifies as well – we have

$$\begin{aligned}
M_1 &= R_i, \\
M_2 &= \frac{R_i b_1 - sC_i c_1}{a_1}, \\
a &= (-1)^{i+1}({R_i}^2 - C_i M_2), \\
b &= \left(\frac{2(R_i a_1/s - C_{i-1}a)}{C_i} - b_1\right) \bmod 2a.
\end{aligned}$$

Pseudo-code for our implementation is given in Algorithm 2.3.

2.5.7 Fast Ideal Cubing (NUCUBE)

When we consider binary-ternary representations of exponents, cubing is required. In general, if we want to compute $\mathfrak{a}^3$ for an ideal class representative $\mathfrak{a} = [a_1, (b_1 + \sqrt{\Delta})/2]$, we can take advantage of the simplification that happens when expanding the computation of $\mathfrak{a}^2\mathfrak{a}$. Here we provide a high level description of a technique for cubing based on similar ideas to NUCOMP and NUDUPL, namely that of computing the quotients of a continued fraction expansion. A detailed description and analysis of this technique can be found in [29].

Similar to ideal squaring, compute integers s' and Y' such that

$$s' = \gcd(a_1, b_1) = X'a_1 + Y'b_1.$$

Note that X' is unused. If $s' \neq 1$, compute

$$s = \gcd(s'a_1, {b_1}^2 - a_1c_1) = Xs'a_1 + Y({b_1}^2 - a_1c_1)$$

Algorithm 2.3 NUDUPL – Fast Ideal Squaring.

```
Input: Reduced representative $\mathfrak{a} = [a_1, (b_1 + \sqrt{\Delta})/2]$
       with $c_1 = ({b_1}^2 - \Delta)/4a_1$ and discriminant $\Delta$.
Output: A reduced or almost reduced representative $\mathfrak{a}^2$.
compute $s, Y \in \mathbb{Z}$ such that $s = \gcd(a_1, b_1) = Xa_1 + Yb_1$ for $X \in \mathbb{Z}$
$a_1 \leftarrow a_1/s$
$U \leftarrow -Yc_1 \bmod a_1$
if $a_1 < |\Delta/4|^{1/4}$ then
    $a \leftarrow {a_1}^2$
    $b \leftarrow (2Ua_1 + b_1) \bmod 2a$
    return $[a, (b + \sqrt{\Delta})/2]$
$\begin{bmatrix} R_{-2} & C_{-2} \\ R_{-1} & C_{-1} \end{bmatrix} \leftarrow \begin{bmatrix} U & -1 \\ a_1 & 0 \end{bmatrix}$
$i \leftarrow -1$
while $R_i > |\Delta/4|^{1/4}$ do
    $i \leftarrow i + 1$
    $q_i = \lfloor R_{i-2} \ / \ R_{i-1} \rfloor$
    $R_i = R_{i-2} - q_i R_{i-1}$
    $C_i = C_{i-2} - q_i C_{i-1}$
$M_2 \leftarrow (R_i b_1 - sC_i c_1)/a_1$
$a \leftarrow (-1)^{i+1}({R_i}^2 - C_i M_2)$
$b \leftarrow (2(R_i a_1 + C_{i-1} a)/C_i) \bmod 2a$
return $[a, (b + \sqrt{\Delta})/2]$
```

for $s, X, Y \in \mathbb{Z}$. If $s' = 1$ then let $s = 1$ too. Then compute U using

$$U = \begin{cases} Y'c_1(Y'(b_1 - Y'c_1a_1) - 2) \bmod {a_1}^2 & \text{if } s' = 1 \\ -c_1(XY'a_1 + Yb_1) \bmod {a_1}^2/s & \text{otherwise.} \end{cases}$$

Next, develop the simple continued fraction expansion of $sU/{a_1}^2$ until

$$R_i < \sqrt{a_1}|\Delta/4|^{1/4} < R_{i-1}.$$

Finally, compute the representative $\mathfrak{a}^3 = [a, (b + \sqrt{\Delta})/2]$ using the equations

$$M_1 = \frac{(R_i a_1 + C_i U a_1)}{{a_1}^2},$$

$$M_2 = \frac{R_i(b_1 + Ua_1) - sC_i c_1}{{a_1}^2},$$

$$a = (-1)^{i+1} R_i M_1 - C_i M_2,$$

$$b = \left(\frac{2(R_i a_1/s - C_{i-1}a)}{C_i} - b_1 \right) \bmod 2a.$$

By [29, p.15 Theorem 5.1], the ideal $[a, (b+\sqrt{\Delta})/2]$ is at most two reduction steps from being reduced. Pseudo-code for our implementation of fast ideal cubing is given in Algorithm 2.4.

Algorithm 2.4 NUCUBE – Fast Ideal Cubing ([29, p.26]).

```
Input: A reduced representative $\mathfrak{a} = [a_1, (b_1 + \sqrt{\Delta})/2]$.
Output: A reduced or almost reduced representative $\mathfrak{a}^3$.
compute $s', Y' \in \mathbb{Z}$ such that $s' = \gcd(a_1, b_1) = X'a_1 + Y'b_1$ and $X' \in \mathbb{Z}$
if $s' = 1$ then
    $s \leftarrow 1$
    $U \leftarrow Y'c_1(Y'(b_1 - Y'c_1a_1) - 2) \bmod {a_1}^2$
else
    compute $s, X, Y \in \mathbb{Z}$ such that $s = \gcd(s'a_1, {b_1}^2 - a_1c_1) = Xs'a_1 + Y({b_1}^2 - a_1c_1)$
    $U \leftarrow -c_1(XY'a_1 + Yb_1) \bmod {a_1}^2/s$
if ${a_1}^2/s < \sqrt{a_1}\ |\Delta/4|^{1/4}$ then
    $a \leftarrow {a_1}^3/s^2$
    $b \leftarrow (b_1 + 2Ua_1/s) \bmod 2a$
    return $[a, (b + \sqrt{\Delta})/2]$
$\begin{bmatrix} R_{-2} & C_{-2} \\ R_{-1} & C_{-1} \end{bmatrix} \leftarrow \begin{bmatrix} U & -1 \\ {a_1}^2/s & 0 \end{bmatrix}$
$i \leftarrow -1$
while $R_i > \sqrt{a_1}|\Delta/4|^{1/4}$ do
    $i \leftarrow i + 1$
    $q_i = \lfloor R_{i-2}\ /\ R_{i-1} \rfloor$
    $R_i = R_{i-2} - q_iR_{i-1}$
    $C_i = C_{i-2} - q_iC_{i-1}$
$M_1 \leftarrow (R_ia_1 + C_iUa_1)/{a_1}^2$
$M_2 \leftarrow (R_i(b_1 + Ua_1) - sC_ic_1)/{a_1}^2$
$a \leftarrow (-1)^{i+1}R_iM_1 - C_iM_2$
$b \leftarrow (2(R_ia_1/s - C_{i-1}a)/C_i - b_1) \bmod 2a$
return $[a, (b + \sqrt{\Delta})/2]$
```

The next chapter discusses some exponentiation techniques that use the ideal arithmetic presented in this chapter, namely fast multiplication, squaring, and cubing.

Chapter 3

Exponentiation

Exponentiation has many applications. Diffie-Hellman key exchange uses exponentiation so that two parties may jointly establish a shared secret key over an insecure channel. An application discussed in detail in this thesis is that of computing the order of an ideal class. The approach used in Chapter 4 is to exponentiate an ideal class to the product, E, of several small primes. The result is an ideal class whose order is likely to not be divisible by any of these primes. The order is then computed using a variant of Shanks' baby-step giant-step algorithm where only powers relatively prime to E are computed. Computing the order of several ideal classes is one method to factor an integer associated with the ideal class group. Faster exponentiation means the approach used there is faster.

In Sections 3.1 and 3.2 we discuss standard exponentiation techniques that rely on a base 2 representation of the exponent. In Section 3.3 we describe double-base number systems, which, as the name implies, are number systems that make use of two bases in the representation of a number. We are particularly interested in representations that use bases 2 and 3, since our implementation of ideal class group arithmetic provides multiplication, squaring, and cubing. In Section 3.4 we discuss some methods to compute double-base representations found in the literature. Throughout this chapter, the running time and space complexities are given in terms of the number of group operations and elements required. They do not take into consideration the binary costs associated with each.

3.1 Binary Exponentiation

The simplest method of exponentiation is binary exponentiation. Let g be an element in a group G and n a positive integer. To compute g^n, we first represent n in binary as

$$n = \sum_{i=0}^{\lfloor \log_2 n \rfloor} b_i 2^i$$

where $b_i \in \{0, 1\}$ such that b_i represents the i^{th} bit of n. We then represent g^n as

$$g^n = \prod_{i=0}^{\lfloor \log_2 n \rfloor} g^{b_i 2^i}.$$

and compute g^{2^i} by repeated squaring of g. The result, g^n, is the product of each g^{2^i} where $b_i = 1$. This description is known as right-to-left binary exponentiation because the result is computed by generating the terms of the product from right to left in the written binary representation of n. The left-to-right variant evaluates the bits of the exponent n from high order to lower order by repeatedly squaring an accumulator and multiplying this with the base element g when $b_i = 1$. The left-to-right variant has the advantage that one of the values in each multiplication, namely g, remains fixed throughout the evaluation. There also exist windowed variants where g^w is precomputed for each w in some window $0 \leq w < 2^k$ for some k (typically chosen to be cache efficient). The exponent n is then expressed in base 2^k. For further discussion of windowed techniques, see [15, Subsection 9.1.3. p.149].

Binary exponentiation algorithms require $\lfloor \log_2 n \rfloor$ squarings and $(\lfloor \log_2 \ n \rfloor + 1)/2$ multiplications on average, since a multiplication is only necessary when $b_i = 1$ and the probability of $b_i = 1$ is $1/2$.

3.2 Non-Adjacent Form Exponentiation

The Non-Adjacent Form (NAF) of an integer is a *signed* base two representation such that no two non-zero terms in the representation are adjacent. Each integer, n, has a unique representation in non-adjacent form. Formally, an integer n is represented by

$$n = \sum_{i=0}^{\lfloor \log_2 n \rfloor + 1} s_i 2^i$$

where $s_i \in \{0, 1, -1\}$ and $s_i \cdot s_{i+1} = 0$. For example, suppose $n = 23814216$. In binary we have

$$23814216 = 2^3 + 2^6 + 2^{13} + 2^{14} + 2^{16} + 2^{17} + 2^{19} + 2^{21} + 2^{22} + 2^{24} \tag{3.1}$$

and in non-adjacent form we have

$$23814216 = 2^3 + 2^6 - 2^{13} - 2^{15} - 2^{18} - 2^{20} - 2^{23} + 2^{25}. \tag{3.2}$$

Similar to the binary case, we compute

$$g^n = \prod_{i=0}^{\lfloor \log_2 n \rfloor + 1} g^{s_i 2^i}.$$

When computing in the ideal class group, the cost of inversion is negligible, but when inversion is expensive, we can instead compute

$$g^n = \left(\prod_{i:s_i=1} g^{2^i} \right) \cdot \left(\prod_{i:s_i=-1} g^{2^i} \right)^{-1}$$

which requires at most one inversion (but this is not necessary for our purposes).

To compute the non-adjacent form of an integer n, inspect n two bits at a time from least significant to most significant. Let $n = \sum b_i 2^i$ be the binary representation of n and let $j = 0$. If the bit pattern $\langle b_{j+1}, b_j \rangle = \mathtt{01}_2$ then let $s_j = 1$ and subtract 2^j from n. If $\langle b_{j+1}, b_j \rangle = \mathtt{11}_2$ then let $s_j = -1$ and add 2^j to n. When the bit pattern $\langle b_{j+1}, b_j \rangle$ is $\mathtt{00}_2$ or $\mathtt{10}_2$, let $s_j = 0$. Next, increment j and repeat while $n \neq 0$.

In our experiments, we use a variation of the above, originally due to Reitwiesner [44], that maintains a carry flag c (see Algorithm 3.1). Instead of adding 2^i to n, set $c = 1$, and instead of subtracting 2^i from n, set $c = 0$. When inspecting n two bits at a time, we consider the bit pattern $(m + c) \bmod 4$ where $m = 2b_{i+1} + b_i$. This technique is faster since addition and subtraction is performed with constant sized integers.

Algorithm 3.1 Computes g^n using right-to-left non-adjacent form (Reitwiesner [44]).

```
Input: $g \in G, n \in \mathbb{Z}_{\geq 0}$
$c \leftarrow 0$                                        {carry flag}
$T \leftarrow g$                                        {invariant: $T = g^{2^i}$}
$R \leftarrow 1_G$
$i \leftarrow 0$
while $n \geq 2^i$ do
    if $\lfloor n/2^i \rfloor + c \equiv 1 \pmod 4$ then
        $R \leftarrow R \cdot T$
        $c \leftarrow 0$
    else if $\lfloor n/2^i \rfloor + c \equiv 3 \pmod 4$ then
        $R \leftarrow R \cdot T^{-1}$
        $c \leftarrow 1$
    $T \leftarrow T^2$
    $i \leftarrow i + 1$
if $c = 1$ then
    $R \leftarrow R \cdot T$
return $R$
```

An advantage of non-adjacent form is that it requires at most $\lfloor \log_2 n \rfloor + 1$ squares and on average $(\lfloor \log_2 n \rfloor + 2)/3$ multiplications, as opposed to $(\lfloor \log_2 n \rfloor + 1)/2$ for binary exponentiation. To see this, recall that non-adjacent form requires that no two non-zero terms be adjacent. Consider any two adjacent terms. The possible outcomes are $(0, 0)$, $(s, 0)$, or $(0, s)$ where $s \in \{-1, 1\}$. This means that 2/3 of the time, 1/2 of the terms will be non-zero, and so the probability of any given term being non-zero is 1/3.

As with binary exponentiation, there exist windowed variants of non-adjacent form (see [15, Algorithm 9.20. p.153]). In this case, g^w is computed for each w in some window $-2^{k-1} < w \leq 2^{k-1}$, and the exponent n is repeatedly evaluated modulo 2^k using the smallest residue in absolute value, i.e. the residue w such that $-2^{k-1} < w \leq 2^{k-1}$.

3.3 Double-Base Number Systems

Binary representation and non-adjacent form use only a single base, namely base 2. Double-base number systems (DBNS), which were first discussed by Dimitrov and Cooklev [20, 21], use two bases. Given two coprime integers p and q and an integer n, we represent n as the sum and difference of the product of powers of p and q,

$$n = \sum_{i=1}^{k} s_i p^{a_i} q^{b_i} \tag{3.3}$$

where $s_i \in \{-1, 1\}$ and $a_i, b_i, k \in \mathbb{Z}_{\geq 0}$. This thesis pays particular attention to representations using bases $p = 2$ and $q = 3$ such that $n = \sum s_i 2^{a_i} 3^{b_i}$. Such representations are referred to as *2,3 representations*.

As an example of a 2,3 representation, consider the number $n = 23814216$ again. Given the bases $p = 2$ and $q = 3$, *one* possible representation of n is

$$23814216 = 2^3 3^3 - 2^4 3^5 + 2^5 3^6 + 2^7 3^7 + 2^9 3^8 + 2^{10} 3^9. \tag{3.4}$$

Another possible representation is

$$23814216 = 2^3 3^2 - 2^{13} 3^2 + 2^{15} 3^6. \tag{3.5}$$

There may be many possible 2,3 representations for a given number, and different representations will trade off between cubings, squarings, and the number of terms. Exponentiation using Equation 3.4 requires 10 squarings, 9 cubings, 1 inverse, and 5 multiplications, while Equation 3.5 requires 15 squarings, 6 cubings, 1 inverse, and 3 multiplications. Contrast this with the binary representation (3.1), which requires 24 squarings, 0 inverses, and 9 multiplications, and the non-adjacent form (3.2), which requires 25 squarings, 5 inverses, and 7 multiplications. The best representation will depend on the needs of the application and the cost of each operation. Later, we shall see some algorithms that take this into account (see Chapter 7), but many are designed to either find representations quickly, of a special form, or with few terms.

3.3.1 Chained 2,3 Representations

One way to classify algorithms that compute 2,3 representations is by the constraints placed on the partition of an integer n.

Definition 3.3.1. A *partition* of n is written $n = x_1 \pm x_2 \pm \cdots \pm x_k$ where the terms x_i are monotonically increasing by absolute value.

One such constraint is on the divisibility of subsequent terms.

Definition 3.3.2. A partition of an integer $n = x_1 \pm x_2 \pm \cdots \pm x_k$ is *chained* if every term x_i divides every term x_j for $i < j$ and $x_i \leq x_j$. A partition is said to be *strictly chained* if it is chained and x_i is *strictly* less than x_j for each $i < j$.

Binary and non-adjacent form are special types of strictly chained partitions, since for any two non-zero terms where $i < j$, we have $x_i = 2^i$, $x_j = 2^j$, $x_i \mid x_j$, and $x_i < x_j$. The 2,3 representation of $23814216 = 2^3 3^2 - 2^{13} 3^2 + 2^{15} 3^6$ is another example of a strictly chained partition, since $2^3 3^2 \mid 2^{13} 3^2 \mid 2^{15} 3^6$.

The benefit of restricting 2,3 representations to chained representations is the ease with which one can compute g^n when n is given as a chain. For example $g^{23814216} = g^{2^3 3^2 - 2^{13} 3^2 + 2^{15} 3^6}$ can be computed by first computing $x_0 = g$, $x_1 = g^{2^3 3^2}$, $x_2 = {x_1}^{2^{10}}$, $x_3 = {x_2}^{2^2 3^4}$, each term being computed by repeated squaring and cubing from the previous term. Finally $g^{23814216} = x_1 \cdot {x_2}^{-1} \cdot x_3$. When n is given as a chained 2,3 representation, Algorithm 3.2 will compute g^n using exactly a_k squares, b_k cubes, $k-1$ multiplications, and at most k inverses[1]. Since $x_i \mid x_{i+1}$, an implementation need only retain x_i in order to compute x_{i+1}, and so only requires storage of a constant number of group elements.

3.3.2 Unchained 2,3 Representations

There is evidence [29] that the shortest possible chained representations require a linear number of terms in relation to the size of the input. This is in contrast to unchained representations for which there are algorithms where the number of terms in the representation is

[1] Recall that when inversion is expensive, the product of terms with negative exponents can be computed separately from terms with positive exponents – the inverse is then computed only once for this product. Since computing inverses is negligible in the ideal class group, we instead compute the product of all terms directly.

Algorithm 3.2 Computes g^n given n as a chained 2,3 partition (Dimitrov et al [23]).

```
Input: $g \in G$, $n = \sum_{i=1}^{k} s_i 2^{a_i} 3^{b_i}$,
       $s_1, ..., s_k \in \{-1, 1\}$,
       $0 \leq a_1 \leq ... \leq a_k \in \mathbb{Z}$,
       $0 \leq b_1 \leq ... \leq b_k \in \mathbb{Z}$.
$i \leftarrow 1$
$a \leftarrow 0$                                   {current power of 2}
$b \leftarrow 0$                                   {current power of 3}
$T \leftarrow g$                                   {loop invariant: $T = g^{2^a 3^b}$}
$R \leftarrow 1_G$
while $i \leq k$ do
    while $a < a_i$ do
        $T \leftarrow T^2, a \leftarrow a + 1$
    while $b < b_i$ do
        $T \leftarrow T^3, b \leftarrow b + 1$
    $R \leftarrow R \cdot T^{s_i}$                 {multiply with $T$ or $T^{-1}$}
    $i \leftarrow i + 1$
return $R$
```

provably sublinear in the length of the input [22, 14], however, these algorithms still require $O(\log n)$ operations overall.

Given an unchained 2,3 representation for an integer $n = \sum s_i 2^{a_i} 3^{b_i}$, Méloni and Hasan [40, Section 3.2] give an algorithm that achieves the same bound on the number of operations as in algorithm 3.2, but with a bound of $O(\min\{\max a_i, \max b_i\})$ on the memory used. Suppose $\max b_i < \max a_i$ and that the terms are labelled and sorted such that $a_1 \geq ... \geq a_k$. The algorithm works by precomputing a table of $T_b = g^{3^b}$ for $0 \leq b \leq \max b_i$. Let $i = 1$ and begin with the first term $s_i 2^{a_i} 3^{b_i}$. Look up $T_{b_i} = g^{3^{b_i}}$ and, after applying the sign s_i, multiply ${T_{b_i}}^{s_i}$ with the running result. Let $a' = a_i - a_{i+1}$ when $i < k$ and $a' = a_k$ when $i = k$. Then square the running result a' times. The algorithm then removes the term $s_i 2^{a_i} 3^{b_i}$ from the list of terms, and continues in this way with the next largest a_i. The algorithm terminates when there are no more terms in the list. Algorithm 3.3 gives pseudo-code for this approach and requires the storage of $O(\max b_i)$ group elements. When $\max a_i < \max b_i$, we use a related algorithm that requires the terms to be sorted such that $b_1 \geq ... \geq b_k$; it precomputes $T_a = g^{2^a}$ for $0 \leq a \leq \max a_i$ and works similar to Algorithm 3.3 but with cubing and squaring appropriately swapped.

Algorithm 3.3 Compute g^n for a 2,3 representation of n ([40, Section 3.2]).

```
Input: $g \in G$, $n = \sum_{i=1}^{k} s_i 2^{a_i} 3^{b_i}$,
       $s_1, ..., s_k \in \{-1, 1\}$,
       $a_1 \geq ... \geq a_k \in \mathbb{Z}_{\geq 0}$,
       $b_1, ..., b_k \in \mathbb{Z}_{\geq 0}$.
$T_b \leftarrow g^{3^b}$ for $0 \leq b \leq \max\{b_1, ..., b_k\}$        {by repeated cubing}
$R \leftarrow 1_G$
$i \leftarrow 1$
while $i < k$ do
    $R \leftarrow R \cdot {T_{b_i}}^{s_i}$        {multiply with $T_{b_i}$ or ${T_{b_i}}^{-1}$}
    $a' \leftarrow a_i - a_{i+1}$
    $R \leftarrow R^{2^{a'}}$        {by squaring $a'$ number of times}
    $i \leftarrow i + 1$
$R \leftarrow R \cdot {T_{b_k}}^{s_k}$
$R \leftarrow R^{2^{a_k}}$        {by squaring $a_k$ number of times}
return $R$
```

For example, the 2,3 representation $23814216 = 2^{15}3^6 - 2^8 3^5 - 2^4 3^6 + 2^3 3^3$ is sorted by decreasing a_i. The algorithm first computes $T_b = g^{3^b}$ for $0 \leq b \leq 6$. Note that it is sufficient to store only the values of $g^{3^{b_i}}$ that actually occur in terms (in this case T_3, T_5, and T_6). Let R_j represent the partial exponentiation of the first j terms with the largest a_i such that $g^{2^{a_{j+1}}}$ is factored out. Let $R_0 = 1_G$ and then compute

$$\begin{aligned}
R_1 &= (T_6)^{2^7} && \Rightarrow g^{2^7 3^6}, \\
R_2 &= \left(R_1 {T_5}^{-1}\right)^{2^4} && \Rightarrow g^{-2^4 3^5 + 2^{11} 3^6}, \\
R_3 &= \left(R_2 {T_6}^{-1}\right)^{2^1} && \Rightarrow g^{-2^1 3^6 - 2^5 3^5 + 2^{12} 3^6}, \\
R_4 &= (R_3 T_3)^{2^3} && \Rightarrow g^{2^3 3^3 - 2^4 3^6 - 2^8 3^5 + 2^{15} 3^6},
\end{aligned}$$

using 15 squares, 6 cubes, 2 inverses, and 3 multiplications. The result of the computation is $R_4 = g^{23814216}$ and the steps (executed from left-to-right, top-to-bottom) are depicted by figure 3.1.

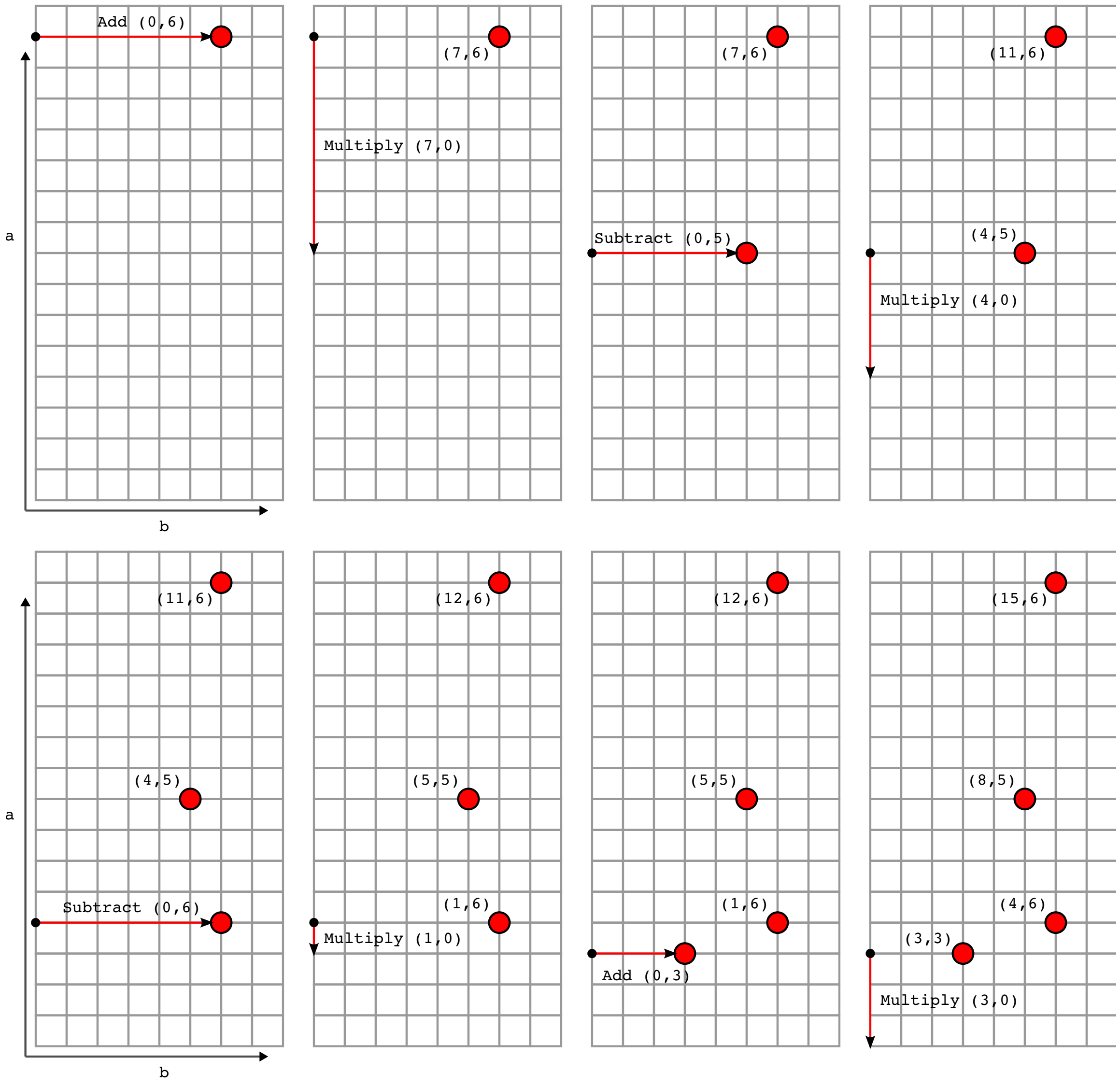

Figure 3.1: The construction of $2^{15}3^6 - 2^8 3^5 - 2^4 3^6 + 2^3 3^3$ using algorithm 3.3. Steps are executed from left-to-right, top-to-bottom.

3.4 Methods for Computing 2,3 Chains/Representations

The section discusses some of the methods from the literature for computing 2,3 representations. The first method generates strict chains from low order to high order (right-to-left), while the second method generates representations (both chained and unchained) from high order to low order (left-to-right). The third technique generates strict chains using a tree-based approach, while the final method computes additive only strict chains of shortest length in a manner similar to chains generated from low order to high order.

These methods trade off the time to compute a representation against the time to exponentiate using that representation. When the exponent is known in advance, one can precompute the chain or representation best suited to the application. Chapter 4 discusses two factoring algorithms that use precomputed representations of exponents to speed their computations. While none of the methods presented in this chapter take into account the relative cost of multiplying, squaring, or cubing ideals, Chapter 7 looks at some variations that attempt to minimize the cost of exponentiation given the average costs of group operations.

3.4.1 Right-to-Left Chains (from low-order to high-order)

The first method we present computes a strictly chained 2,3 partition that is generated from low order to high order and is from Ciet et al [13]. We begin by recalling the technique for binary exponentiation that computes from low order to high order. Given an element $g \in G$ and an integer n, the function

$$\operatorname{bin}(g,n) = \begin{cases} 1 & \text{if } n = 0 \\ \operatorname{bin}(g, n/2)^2 & \text{if } n \equiv 0 \pmod 2 \\ \operatorname{bin}(g, n-1) \cdot g & \text{if } n \equiv 1 \pmod 2 \end{cases}$$

will compute the binary exponentiation of g^n from low order to high order. This algorithm repeatedly removes factors of 2 from n. When n is not divisible by 2, it subtracts 1 such that the input to the recursive call will be divisible by 2. The recursion terminates with the base case of $n = 0$.

This concept is extended to a 2,3 number system by repeatedly removing factors of 2 from n, and then factors of 3 from n. At this point, either $n \equiv 1 \pmod 6$ or $n \equiv 5 \pmod 6$. When $n \equiv 1 \pmod 6$, we recurse on $n - 1$ and the input will be divisible by both 2 and 3. When $n \equiv 5 \pmod 6$, we recurse on $n + 1$. Again, the input to the recursive call will be divisible by both 2 and by 3. Using this idea, we perform a 2,3 exponentiation recursively as

$$\operatorname{rtl}(g,n) = \begin{cases} 1 & \text{if } n = 0 \\ \operatorname{rtl}(g,n/2)^2 & \text{if } n \equiv 0 \pmod 2 \\ \operatorname{rtl}(g,n/3)^3 & \text{if } n \equiv 0 \pmod 3 \\ \operatorname{rtl}(g,n-1)\cdot g & \text{if } n \equiv 1 \pmod 3 \\ \operatorname{rtl}(g,n+1)\cdot g^{-1} & \text{if } n \equiv 2 \pmod 3. \end{cases}$$

Algorithm 3.4 describes a non-recursive function with group operations that correspond to those generated by the function rtl. The idea is as follows: let $a = 0$, $b = 0$, and $i = 1$. While $n > 0$, repeatedly remove factors of 2 from n and increment a for each factor of 2 removed. Then repeatedly remove factors of 3 from n and increment b for each factor of 3 removed. At this point, either $n \equiv 1 \pmod 6$ or $n \equiv 5 \pmod 6$ and so continue on $n-1$ or $n+1$ respectively. When we continue on $n-1$, this corresponds to adding the current term, so we set $s_i = 1$, and when we continue on $n+1$, this corresponds to subtracting the current term, so we set $s_i = -1$. Let $a_i = a$ and $b_i = b$ and then increment i and then repeat this process while $n > 0$. We then use Algorithm 3.2 to compute the exponentiation given the strictly chained 2,3 partition. When we are not able to precompute the chain, it is relatively straightforward to interleave the computation of the partition with the computation of the exponentiation, since the terms $s_i 2^{a_i} 3^{b_i}$ are computed in increasing order for $i = 1..k$.

To see the correctness of the above procedure, consider a modification to the recursive function rtl such that it returns a partition of the input n as a list of terms $s_i 2^{a_i} 3^{b_i}$. When the result of the recursive call is squared, this corresponds to incrementing a_i in each term of the list. Similarly, when the result is cubed, this corresponds to incrementing b_i in each term of the list. When the result is multiplied with g, we prepend a term of $+1$ to the partition, and when the result is multiplied with g^{-1}, we prepend a term of -1 to the partition. On each iteration of the loop, either $n \equiv 0 \pmod 2$ or $n \equiv 0 \pmod 3$, so either a increases or b

Algorithm 3.4 2,3 strict chains from low order to high order (Ciet et al [13]).

```
Input: $n \in \mathbb{Z}_{\geq 0}$
$(a, b) \leftarrow (0, 0)$
$i \leftarrow 1$
while $n > 0$ do
    while $n \equiv 0 \pmod 2$ do
        $n \leftarrow n/2, a \leftarrow a + 1$
    while $n \equiv 0 \pmod 3$ do
        $n \leftarrow n/3, b \leftarrow b + 1$
    if $n \equiv 1 \pmod 3$ then
        $n \leftarrow n - 1, s \leftarrow 1$
    else if $n \equiv 2 \pmod 3$ then
        $n \leftarrow n + 1, s \leftarrow -1$
    $(s_i, a_i, b_i) \leftarrow (s, a, b)$
    $i \leftarrow i + 1$
$k \leftarrow i$
return $(a_1, b_1, s_1), ..., (a_k, b_k, s_k)$
```

increases. Since every term $|s_i 2^{a_i} 3^{b_i}|$ is strictly less than $|s_j 2^{a_j} 3^{b_j}|$ when $i < j$, the partition is strictly chained.

3.4.2 Left-to-Right Chains (from high-order to low-order)

The previous section gives a procedure for generating a strictly chained 2,3 partition for an integer n such that the terms are ordered from smallest absolute value to largest. Here we present a greedy approach, suggested by Berthé and Imbert [10], which generates the terms in order of the largest absolute value to the smallest. The idea is to find a term, $s2^a3^b$, that is closest to the remaining target integer n and then repeat on $n - s2^a3^b$. Let

$$\text{closest}(n) = s2^a3^b$$

such that $a, b \in \mathbb{Z}_{\geq 0}$ minimize $\big||n| - 2^a3^b\big|$ and $s = -1$ when $n < 0$ and $s = 1$ otherwise. A recursive function to compute a 2,3 representation greedily is

$$\text{greedy}(n) = \begin{cases} 0 & \text{if } n = 0 \\ \text{closest}(n) + \text{greedy}(n - \text{closest}(n)) & \text{otherwise.} \end{cases}$$

Note that the representation generated may not be a chained partition. To generate a chained partition we restrict the maximum powers of 2 and 3 generated by the function closest. We bound the function closest, such that it returns the triple

$$\text{closest}'(n, A, B) = (s2^a3^b, a, b)$$

where $0 \le a \le A$, $0 \le b \le B$, a and b minimize $\left||n| - 2^a3^b\right|$, and $s = -1$ when $n < 0$ and $s = 1$ when $n > 0$. Our recursive function is then

$$\text{greedy}'(n, A, B) = \begin{cases} 0 & \text{if } n = 0 \\ v + \text{greedy}'(n - v, a, b) & \text{where } (v, a, b) = \text{closest}'(n, A, B). \end{cases}$$

We present pseudo-code in Algorithm 3.5. Note that on successive invocations of greedy$'$, the absolute value of $v = |s2^a3^b|$ returned by closest$'$ is monotonically decreasing. Reversing the terms of the partition gives a chained 2,3 partition of n that we can use to perform exponentiation using Algorithm 3.2.

Algorithm 3.5 Greedy left to right representation (Berthé and Imbert [10]).

```
Input: $n \in \mathbb{Z}$, $A, B \in \{\mathbb{Z}_{\ge 0}, +\infty\}$        {$+\infty$ for unbounded $a$ or $b$}
$L \leftarrow$ empty list
while $n \ne 0$ do
    compute integers $a$ and $b$ that minimize $\left||n| - 2^a3^b\right|$
    such that $0 \le a \le A$ and $0 \le b \le B$
    $s \leftarrow -1$ when $n < 0$ and 1 otherwise
    push $(s, a, b)$ on to the front of $L$
    optionally set $(A, B) \leftarrow (a, b)$ when a chain is desired
    $n \leftarrow n - s2^a3^b$
return $L$
```

To compute the 2,3 term closest to n, a straightforward approach is to compute the set

$$V = \{2^a3^b, 2^{a+1}3^b : 0 \le b \le B \le \lceil \log_3 |n| \rceil, a = \lfloor \log_2 |n/(3^b)| \rfloor \text{ when } a \le A\}.$$

Then take the element $v \in V$ that is closest to $|n|$, and take $s \in \{-1, 1\}$ based on the sign of n. Since the set V contains $O(\log |n|)$ elements, computing the term closest to n by this method takes $\Omega(\log |n|)$ steps. When a and b are not constrained (i.e. $A \geq \lceil \log_2 |n| \rceil$ and $B \geq \lceil \log_3 |n| \rceil$), and we simply want to compute the 2,3 term closest to n, Berthé and Imbert [10] present a method that requires at most $O(\log \log |n|)$ steps.

They also found that applying a global bound A^* and B^* such that $0 \leq a \leq A^*$ and $0 \leq b \leq B^*$ often lead to representations with a lower density. In the unchained case, recursive calls to greedy′ use the global values of A^* and B^* rather than the values a and b generated by closest′. Finding the best greedy representation is then a matter of iterating over the global bounds A^* and B^* to compute 2,3 representations constrained appropriately. We discuss some of the results of this in Chapter 7.

3.4.3 Pruned Tree of ±1 Nodes

The next technique for finding strictly chained 2,3 partitions was suggested by Doche and Habsieger [24]. The idea is similar to the method for generating chains from right to left as described in Subsection 3.4.1 above, but this technique differs by generating multiple values that may be further reduced by powers of 2 and 3. The procedure is given in Algorithm 3.6. The idea is to maintain a tree, T, with at most L leaf nodes. At each iteration, each leaf node $v \in T$ generates two new leaves, $v - 1$ and $v + 1$, which are then reduced as much as possible by removing factors of 2 and 3. We then discard any duplicate nodes and all but the smallest L elements generated. The path from the root to the first leaf with a value of 1 represents a chained 2,3 partition of the number n.

Larger values of L sometimes produce chains with fewer terms, but take longer to compute. When the the input integer n is known in advance, this might not be a problem, however, large values of L can still be prohibitively expensive. Empirically, the authors found that $L = 4$ was a good compromise between the length of the chain generated and the time to compute the chain.

Algorithm 3.6 Chain from ±1 Pruned Tree (Doche and Habsieger [24]).

```
Input: $n \in \mathbb{Z}_{>0}$ and a bound $L \in \mathbb{Z}_{>0}$.
$T \leftarrow$ a binary tree on the node $n$
while no leaf is 1 do
    for all leaf nodes $v \in T$ do
        insert as a left child $(v - 1)$ with all factors of 2 and 3 removed
        insert as a right child $(v + 1)$ with all factors of 2 and 3 removed
    discard any duplicate leaves
    discard all but the smallest $L$ leaves
return the chained 2,3 partition generated by the path from the root to the first leaf
    node containing 1
```

3.4.4 Shortest Additive 2, 3 Chains

In the previous subsection, the search for a 2,3 chain iterates on ±1 the value of the L smallest candidates. When we further restrict a chain to contain only positive terms, the number of possible 2,3 chains is reduced. Imbert and Phillipe [30] consider searching for additive 2,3 strictly chained partitions that contain as few terms as possible. They give the following recursive function to compute the minimum number of terms in such a chain. Let $s(n)$ denote the smallest k such that n can be represented as $n = \sum_{i=1}^{k} 2^{a_i} 3^{b_i}$. We define $s(n)$ as

$$s(n) = \begin{cases} \min\{s(n/3), s(n/2)\} & \text{when } n \equiv 0 \pmod 6 \\ 1 + s(n-1) & \text{when } n \equiv 1 \pmod 6 \\ s(n/2) & \text{when } n \equiv 2 \pmod 6 \\ \min\{s(n/3), 1 + s((n-1)/2)\} & \text{when } n \equiv 3 \pmod 6 \\ \min\{s(n/2), 1 + s((n-1)/3)\} & \text{when } n \equiv 4 \pmod 6 \\ 1 + s((n-1)/2) & \text{when } n \equiv 5 \pmod 6 \end{cases}$$

where the base cases are handled by $s(n) = 1$ when $n \leq 2$.

The corresponding 2,3 chain is computed by memoizing a shortest chain for each solution to $s(n)$ encountered. When a recursive call uses $n/2$, the chain for n is the chain for $n/2$ with each term multiplied by 2. Similarly, if the recursion uses $n/3$, each term is multiplied by 3. When the recursion uses $n - 1$, we simply add the term 1 to the chain representing $n - 1$.

3.5 Summary

This chapter outlined some exponentiation techniques from the literature. We started with binary exponentiation based on a binary representation of the exponent. Next we described non-adjacent form using a signed base 2 encoding of the exponent. Since cubing is often faster than combined multiplication with squaring, we discussed 2,3 number systems where an integer can have many representations as the sum of the products of 2 and 3. Exponentiation based on 2,3 representations fall under two classes: chained and unchained. Chained representations can typically be computed while interleaved with the exponentiation operation. They also require storage of only a constant number of group elements in addition to the input arguments. Unchained representations often have fewer terms or operations in their representation. Exponentiation of a group element using an unchained representation of the exponent can be performed using a linear number of group elements in the size of the exponent.

Coming up, Chapter 7 discusses several variations of chained and unchained 2,3 representations, many of which take into account the average time to multiply, square, and cube elements from the ideal class group. The actual performance of these variations guide our implementation of a factoring algorithm called “SuperSPAR”. In the next chapter, we provide the background for SPAR and the SuperSPAR factoring algorithm.

Chapter 4

SuperSPAR

One contribution of this thesis is to improve the speed of arithmetic in the ideal class group of imaginary quadratic number fields with an application to integer factoring. Chapter 2 describes the ideal class group, and Chapter 3 gives methods for exponentiation in generic groups. This chapter makes a connection between the two and integer factorization. Section 4.1 discusses an algorithm due to Schnorr and Lenstra [45], called SPAR, that uses the ideal class group to factor an integer associated with the discriminant. Section 4.2 expands on SPAR by incorporating a primorial steps algorithm, due to Sutherland [50, §4.1], and discussed in Subsection 4.2.1. When the order of any element from a set is sufficient, Sutherland [50, §5.4] gives an algorithm with subexponential complexity, discussed in Subsection 4.2.2. Finally, Subsection 4.2.3 reconsiders the factoring algorithm SPAR in the context of primorial steps for order finding. We call this new algorithm "SuperSPAR".

4.1 SPAR

SPAR is an integer factoring algorithm that works by finding a reduced ambiguous class with a discriminant associated with the integer to be factored. The algorithm was published by Schnorr and Lenstra [45], but was independently discovered by Atkin and Rickert who named it SPAR after Shanks, Pollard, Atkin, and Rickert [32, p.182].

4.1.1 Ambiguous Classes and the Factorization of the Discriminant

The description of SPAR (see [45]) uses the group of equivalence classes of binary quadratic forms – a group isomorphic to the ideal class group of imaginary quadratic fields (see Frölich and Taylor [25]). Since this thesis discusses the ideal class group, SPAR is described in that setting here. This thesis uses reduced representatives for elements of the ideal class group, and the equivalence class $[\mathfrak{a}]$ for a reduced representative $\mathfrak{a}$ is denoted using the $\mathbb{Z}$-module $\mathfrak{a} = [a, (b + \sqrt{\Delta})/2]$. Our implementation also maintains a third term $c = (b^2 - \Delta)/4a$.

Definition 4.1.1. The *ambiguous classes* are the classes $[\mathfrak{a}]$ such that $[\mathfrak{a}]^2$ is the identity class [45, p.302]. Notice that the identity ideal class $[\mathcal{O}_\Delta] \in Cl_\Delta$ is an ambiguous class.

According to [45, p.303], every reduced representative of an ambiguous class with negative discriminant has either $b = 0$, $a = b$, or $a = c$. Since the discriminant is defined as $\Delta = b^2 - 4ac$, these reduced representatives correspond to a factorization of the discriminant. For a reduced ambiguous ideal class either

$$\begin{aligned} \Delta &= 4ac && \text{when } b = 0, \\ \Delta &= b(b - 4c) && \text{when } a = b, \text{ or} \\ \Delta &= (b - 2a)(b + 2a) && \text{when } a = c. \end{aligned}$$

Suppose we wish to find a factor of an odd integer N. Since $\Delta = b^2 - 4ac$ we must have $\Delta \equiv 0, 1 \pmod 4$. Therefore, for some square free integer k, let $\Delta = -kN$ when $-kN \equiv 1 \pmod 4$ and $\Delta = -4kN$ otherwise. Now to find a factor of N, we find a reduced ambiguous class representative with discriminant Δ. For a reduced ambiguous class representative, compute $d = \gcd(a, N)$ if $b = 0$ or $a = b$, and $d = \gcd(b - 2a, N)$ otherwise. If we are lucky, d is a proper factor of N. See Chapter 5 for a discussion on how to compute $\gcd(N, m)$.

4.1.2 SPAR Algorithm

The SPAR factoring algorithm uses two stages, an exponentiation stage and a search stage, in order to find an ambiguous class with a discriminant associated with the integer to factor. Recall from Subsection 2.5.2 that for a negative discriminant Δ, the ideal class group Cl_Δ has a finite number of elements. This means that for a random ideal class $[\mathfrak{a}]$, there exists an integer h such that $[\mathfrak{a}]^h = [\mathcal{O}_\Delta]$. We say that h is the *order* of the element $[\mathfrak{a}]$ and denote this by $h = \operatorname{ord}([\mathfrak{a}])$. When the order is even, then $[\mathfrak{b}] = [\mathfrak{a}]^{h/2}$ is an ambiguous ideal class, because $[\mathfrak{b}]^2 = [\mathcal{O}_\Delta]$. The exponentiation stage of SPAR chooses an exponent E such that, with a certain probability, E is a multiple of the odd part of the order of $[\mathfrak{a}]$, and by repeated squaring of $[\mathfrak{a}]^E$, the algorithm will find an ambiguous class. Failing this, the search stage performs a random walk on the ideal class generated by the first stage in an attempt to find a multiple of the order of $[\mathfrak{a}]$. Knowing a multiple of the order of $[\mathfrak{a}]$, the algorithm then attempts to find an ambiguous class and a factor of N.

Following Schnorr and Lenstra [45], the exponentiation stage takes the first t primes $p_1 = 2, p_2 = 3, ..., p_t \leq N^{1/2r}$ for $r = \sqrt{\ln N / \ln \ln N}$. Let $E = \prod_{i=2}^{t} {p_i}^{e_i}$ for $e_i = \max\{v : {p_i}^v \leq {p_t}^2\}$ and compute $[\mathfrak{b}] = [\mathfrak{a}]^E$. Notice that we exponentiate $[\mathfrak{a}]$ to the product of only *odd* prime powers. The reason for this is that if $\operatorname{ord}([\mathfrak{a}])$ divides $2^j E$ for some j, then we can compute $[\mathfrak{b}]^{\left(2^j\right)}$ for the smallest j such that $[\mathfrak{b}]^{\left(2^j\right)} = [\mathcal{O}_\Delta]$. It follows that $[\mathfrak{b}]^{\left(2^{j-1}\right)}$ is an ambiguous ideal class and we attempt to factor N. Since $\operatorname{ord}([\mathfrak{a}])$ may not divide $2^j E$, Schnorr and Lenstra [45, p.291] bound j to be no larger than $\ell = \left\lfloor \log_2 \sqrt{N} \right\rfloor$. If this stage fails to find $[\mathfrak{b}]^{\left(2^j\right)} = [\mathcal{O}_\Delta]$ for some j, the algorithm continues with the search stage.

The exponentiation stage computes $[\mathfrak{b}] = [\mathfrak{a}]^E$ and $[\mathfrak{c}] = [\mathfrak{b}]^{\left(2^\ell\right)}$. The search stage performs a random walk through the cyclic group generated by the ideal class $[\mathfrak{c}]$ in an attempt to find the order $h = \operatorname{ord}([\mathfrak{c}])$. Let $\langle [\mathfrak{c}] \rangle$ denote the cyclic group generated by $[\mathfrak{c}]$ and let $f : \langle [\mathfrak{c}] \rangle \to \langle [\mathfrak{c}] \rangle$ be a function from one element in the cyclic group to another. The function f should have the property that if x is known for some $[\mathfrak{c}]^x$, then y can be determined for $[\mathfrak{c}]^y = f([\mathfrak{c}]^x)$. Let $[\mathfrak{c}_1] = [\mathfrak{c}]$ and repeatedly compute

$$[\mathfrak{c}_{i+1}] = f([\mathfrak{c}_i])$$

until there is some $j < k$ such that $[\mathfrak{c}_j] = [\mathfrak{c}_k]$. By the function f, we compute u and v such that $[\mathfrak{c}_j] = [\mathfrak{c}]^u$ and $[\mathfrak{c}_k] = [\mathfrak{c}]^v$. Then $h = v - u$ is a multiple of the order of $[\mathfrak{c}]$, and we attempt to find an ambiguous class by computing $[\mathfrak{d}]^{\left(2^j\right)}$ for $[\mathfrak{d}] = [\mathfrak{b}]^h$ and the smallest such $j \leq \left\lfloor \log_2 \sqrt{N} \right\rfloor$.

4.1.3 SPAR Complexity

The original publication of SPAR by Schnorr and Lenstra [45] claimed that every composite integer N could be factored in $o\left(\exp\sqrt{\ln N \ln\ln N}\right)$ bit operations. This was the first factoring algorithm for which this runtime had been conjectured, and it was also the first for which this conjecture had to be withdrawn [39].

The first stage of the algorithm exponentiates a random ideal class $[\mathfrak{a}] \in Cl_\Delta$ to the product $\prod_{i=2}^{t} {p_i}^{e_i}$ of the first t primes where $e_i = \max\{v : {p_i}^v \leq {p_t}^2\}$. Using binary exponentiation, this takes $O(p_t)$ group operations since there are about $p_t / \log p_t$ prime powers in the product, each of which is at most $\lceil 2\log_2 p_t \rceil$ in size. According to [45, p.290], for a random composite integer $m \in [0, N]$, this stage will factor m with probability $\geq r^{-r}$. The search stage performs a random walk of at most $O(p_t)$ group operations and with probability $\geq (r-2)^{-(r-2)}$ will factor m [45, p.290]. Their claim is that if the exponentiation stage is run on each integer kN for $k \leq r^r$, then every composite integer N will be factored within $o\left(\exp\sqrt{\ln N \ln\ln N}\right)$ bit operations.

This claim was based on a false assumption – for a complete discussion, see Lenstra and Pomerance [39, § 11]. In short, the original assumption was that for fixed N and variable k, the class number (h_Δ for $\Delta = -kN$) was as likely to have no prime divisors larger than p_t as the class number associated with a random discriminant of approximately the same size. This assumption meant that one could take both k and p_t to be no larger than $N^{1/2r} = \exp\left(\frac{1}{2}\sqrt{\ln N \ln\ln N}\right)$, leading to an upper bound of $\exp\left(\sqrt{\ln N \ln\ln N}\right)$ for the expected running time. Lenstra and Pomerance show [39, §11] that this assumption is incorrect for a sufficiently dense sequence of integers, however, future work (Section 9.3) could use a median case analysis of SPAR, which might show a subexponential median case running time.

4.2 SuperSPAR

This section discusses SuperSPAR – an integer factoring algorithm and our motivation for improving the performance of exponentiation in the ideal class group. Similar to SPAR, SuperSPAR attempts to find an ambiguous class with a discriminant associated with N, the odd integer we wish to factor. Also similar is that SuperSPAR operates in two stages: an exponentiation stage and a search stage. The first stage of SuperSPAR is the same as in SPAR, although we empirically optimize the choice of the exponent in this stage (see Sections 8.3 and 8.6). The second stage of SuperSPAR differs from SPAR, however. In both algorithms, this stage attempts to find the order of an ideal class. Schnorr and Lenstra [45, p.294] suggest a Pollard-Brent recursion [12] based on Pollard's Rho method [42], but also remark [45, p.298] that Shanks' baby-step giant-step method [47] could be used to deterministically find the order. SuperSPAR instead uses a bounded primorial steps algorithm, due to Sutherland [50, §4.1], and discussed in Subsection 4.2.1 (later, Section 8.6 discusses the empirical optimisation of parameters for the search stage of SuperSPAR). In the case of SuperSPAR, any ideal class with a discriminant associated with N is a candidate for splitting N. As such, the algorithm does not need to find the order of a particular ideal class, but the order of any of several ideal classes will suffice. Subsection 4.2.1 describes a bounded primorial steps algorithm in this context. Finally, Subsection 4.2.3 gives a high level description of the operation of SuperSPAR.

4.2.1 Bounded Primorial Steps

The bounded primorial steps algorithm [50] is an order finding algorithm for generic groups with a worst case asymptotic complexity of $o(\sqrt{M})$ group operations where M is a bound on the order of the group element. This is asymptotically better than previously known order finding algorithms for generic groups, such as the Pollard-Brent method [12] and Shanks' baby-step giant-step method [47], both of which have complexity $O(\sqrt{M})$. The algorithm improves upon Shanks' baby-step giant-step method by using a group element whose order is known to not have any small divisors.

Shanks' baby-step giant-step method computes the order of an element α in a group G. Given a bound M on the order of the element α, let $s = \left\lceil \sqrt{M} \right\rceil$ and compute $\alpha^1, \alpha^2, ..., \alpha^s$ storing each $\alpha^i \mapsto i$ in a table[1] – these are the baby steps. If any $\alpha^i = 1_G$, then $\mathrm{ord}(\alpha) = i$ and we are finished. Otherwise compute $\alpha^{2s}, \alpha^{3s}, ...$ and for each α^{js}, if α^{js} is in the table then $\alpha^{js} = \alpha^i$ for some i and $js - i$ is the order of α. These are the giant steps. Notice that s is chosen such that after an equal number of baby steps and giant steps, the last giant step has an exponent $s^2 \geq M$. Following Sutherland [50, p.50], we point out that if computing the inverse of an element is cheaper than multiplying two elements, the number of multiplications is reduced by letting $s = \left\lceil \sqrt{M/2} \right\rceil$ and computing the giant steps $\alpha^{2s}, \alpha^{-2s}, \alpha^{4s}, \alpha^{-4s}, ...$ instead.

Sutherland observed [50, p.56] that if $h = \mathrm{ord}(\alpha)$ is odd, then computing only odd powers $\alpha^1, \alpha^3, ..., \alpha^{s-1}$ for the baby steps is sufficient. We still need to compute giant steps $\alpha^{2s}, \alpha^{3s}, ...$, for some s that is even since we want to find some $\alpha^{js} = \alpha^i$ where $js - i$ is odd. In this case, $s = \left\lceil \sqrt{2M} \right\rceil$ so that after roughly $\sqrt{M/2}$ baby steps and $\sqrt{M/2}$ giant steps, the last giant step has exponent $\geq M$.

The problem is that $\mathrm{ord}(\alpha)$ may not be odd. However, by repeated squaring of α it is easy to find an element whose order is guaranteed to be even [50, p.56]. Given a bound M on the group order, compute $\beta = \alpha^{2^\ell}$ where $\ell = \lfloor \log_2 M \rfloor$ and now run the modified algorithm on β to find $h' = \mathrm{ord}(\beta)$. The order of α can be found by computing $\zeta = \alpha^{h'}$ and then repeatedly squaring ζ until $\zeta^{2^k} = 1_G$ for some k. The order of α is then $2^k h'$.

This approach is extended to computing $\beta = \alpha^E$ where $E = 2^{\lfloor \log_2 M \rfloor} 3^{\lfloor \log_3 M \rfloor}$ and the order of β is coprime to both 2 and 3. In this case, compute baby steps with exponents coprime to 6 and giant steps that are a multiple of 6. More generally, this works for any

[1] The table maps group elements α^i to exponents i.

Algorithm 4.1 Primorial Steps (Sutherland [50, p.57]).

```
Input: $\alpha \in G$, a bound $M \geq \mathrm{ord}(\alpha)$, and a fast order algorithm $\mathcal{A}(\alpha, E)$.
maximize $w$ such that $P_w \leq \sqrt{M}$
maximize $m$ such that $m^2 P_w \phi(P_w) \leq M$
$s = mP_w$
$E = \prod_{i=1}^{n} p_i^{\lfloor \log_{p_i} M \rfloor}$
$\beta \leftarrow \alpha^E$
for $i$ from 1 to $s$ where $i$ is coprime to $P_w$ do
    compute $\beta^i$ and store $\beta^i \mapsto i$ in the table          {baby steps}
    if $\beta^i = 1_G$ then
        return $i \cdot \mathcal{A}(\alpha^i, E)$
for $j = 2s, 3s, ...$ do
    if $\beta^j$ is in the table then
        lookup $\beta^j \mapsto i$ from the table                          {giant steps}
        $h' = j - i$
        return $h' \cdot \mathcal{A}(\alpha^{h'}, E)$
```

primorial P_w such that

$$P_w = 2 \times 3 \times \cdots \times p_w = \prod_{i=1}^{w} p_i$$

where p_i is the i^{th} prime. Following Sutherland [50, p.57], select the largest $P_w \leq \sqrt{M}$ and then maximize m such that $m^2 P_w \phi(P_w) \leq M$, where $\phi(P_w)$ is the number of integers coprime to P_w given as

$$\phi(P_w) = (2-1) \times (3-1) \times \cdots \times (p_w - 1) = \prod_{i=1}^{w} (p_i - 1). \tag{4.1}$$

Let $e_i = \lfloor \log_{p_i} M \rfloor$ for $1 \leq i \leq w$ and $E = 2^{e_2} \times 3^{e_3} \times \cdots \times {p_w}^{e_w}$. Then compute $\beta = \alpha^E$. The bound on the largest baby step is $s = mP_w$, and we compute $h' = \mathrm{ord}(\beta)$ by taking at most $m\phi(P_w)$ baby steps coprime to P_w and at most $m\phi(P_w)$ giant steps of size mP_w. Pseudo-code for the bounded Primorial Steps technique is given in Algorithm 4.1. By [50, p.59 Proposition 4.2], the number of group operations in the worst case is bound by $O(\sqrt{M/\log\log M})$.

To compute $h = \mathrm{ord}(\alpha)$ given $h' = \mathrm{ord}(\beta)$, one uses a fast order finding algorithm. One fast order finding algorithm, $\mathcal{A}(\alpha^{h'}, E)$, uses the factorization of $E = \prod {p_i}^{e_i}$. Notice that $\alpha^{Eh'} = \beta^{h'} = 1_G$. The idea is to iterate on the factors of E, removing each factor p before computing $\zeta = \alpha^{E'h'}$ for the product $E' = E/p$. If $\zeta \neq 1_G$, then $\mathrm{ord}(\alpha)$ does not divide $E'h'$ and p must be a factor of the order of α. The algorithm then continues with the next prime factor of E'. Additional fast order finding algorithms are given in [50, Chapter 7].

4.2.2 Primorial Steps for a Set

Both SPAR and SuperSPAR use the order of an ideal class to find an ambiguous ideal class. As such, their success in splitting an integer is not limited to computing the order of a particular ideal class, but the order of any ideal class from a set of ideal classes may work. More generally, let $\{\alpha_i \in G_i\}$ be a set of elements from different groups such that the order of α_i is distributed uniformly at random on the interval $[1, M]$ where M is a bound on the largest order of the elements α_i. When the order of any element α_i from the set will suffice, Sutherland [50, §5.4] gives an algorithm with subexponential complexity.

To begin, an integer x is *y-smooth* if all of its prime factors are no larger than y. By [50, p.81], the probability that a random integer x is $x^{1/u}$ smooth is $u^{-u+o(1)}$. Assuming that there is some α_i such that $\mathrm{ord}(\alpha_i)$ is $M^{1/u}$ smooth, let $M' = M^{2/u}$ and then attempt to compute $\mathrm{ord}(\alpha_i)$ using M' as a bound for the bounded primorial steps algorithm. This will use $o(M^{1/u})$ group operations since the bounded primorial steps algorithm uses $o(\sqrt{M'})$ operations for a bound M'. If the algorithm fails to find the order of α_i, we try again for the next α_{i+1} in the set. Using this approach, according to [50, pp.81–82] the expected running time to find the order of some α_i is approximately

$$u^{u+o(1)} M^{1/u} = \exp\left(\frac{1}{u}\log M + u \log u + o(1)\right).$$

The cost is minimized for $u \approx \sqrt{2 \log M / \log\log M}$, which gives an expected running time of

$$\exp\left(\left(\sqrt{2} + o(1)\right) \sqrt{\log M \log\log M}\right).$$

Notice that the idea behind the SPAR factoring algorithm is to find an ambiguous ideal for one of the class groups with valid discriminant $\Delta = -kN$ for $1 \leq k \leq r^r$ where $r =$

$\sqrt{\ln N/\ln\ln N}$. In this case, the success of splitting a composite integer N is not limited to finding an ambiguous ideal within a single group, but instead to finding an ambiguous ideal from any of several groups. As such, the above approach is directly applied to that of the SuperSPAR factoring algorithm. Unfortunately, as shown by Lenstra and Pomerance [39, § 11], there exist integers N for which there is no multiplier k such that the class group Cl_Δ for $\Delta = -kN$ is sufficiently smooth. However, the next subsection assumes that such integers are sufficiently rare that SuperSPAR will perform well in practice.

4.2.3 SuperSPAR Algorithm

As mentioned previously, SuperSPAR works in two stages: an exponentiation stage and a search stage. First, let N be the odd composite integer that we wish to factor. Then, for some square free integer k, choose a discriminant $\Delta = -kN$ or $\Delta = -4kN$ such that $\Delta \equiv 0, 1 \pmod 4$. The exponentiation stage of SuperSPAR is the same as in SPAR, but we quickly recap this here.

In the exponentiation stage of SuperSPAR, we take the first t primes $p_1 = 2, p_2 = 3, ..., p_t \le N^{1/2r}$ for $r = \sqrt{\ln N/\ln\ln N}$, let $E = \prod_{i=2}^{t} {p_i}^{e_i}$ be the product of the odd prime powers where $e_i = \max\{v : {p_i}^v \le {p_t}^2\}$, and $[\mathfrak{b}] = [\mathfrak{a}]^E$ for a random ideal class $[\mathfrak{a}] \in Cl_\Delta$. If $[\mathfrak{b}] = [\mathcal{O}_\Delta]$, then $\operatorname{ord}([\mathfrak{a}])$ divides E and is odd, and we cannot find an ambiguous ideal class from $[\mathfrak{a}]$. As such, we try again with a different random ideal class in Cl_Δ. Assuming $[\mathfrak{b}] \ne [\mathcal{O}_\Delta]$, we then compute $[\mathfrak{b}]^{\left(2^j\right)}$, by repeated squaring, for $1 \le j \le \left\lfloor \log_2 \sqrt{|\Delta|} \right\rfloor$ or until $[\mathfrak{b}]^{\left(2^j\right)}$ is an ambiguous ideal. If there is some j such that $[\mathfrak{b}]^{\left(2^j\right)}$ is an ambiguous ideal, we attempt to split the integer N, and if this fails, we try again for another ideal class in Cl_Δ. If there is no j such that $[\mathfrak{b}]^{\left(2^j\right)}$ is an ambiguous ideal class, then we have computed $[\mathfrak{c}] = [\mathfrak{b}]^{\left(2^\ell\right)}$ for $\ell = \left\lfloor \log_2 \sqrt{|\Delta|} \right\rfloor$, and SuperSPAR moves on to the search stage.

The search stage of SuperSPAR differs from that of SPAR. Using SPAR, one attempts to find the order of $[\mathfrak{c}]$ by performing a random walk using at most $O(p_t)$ group operations. With SuperSPAR, on the other hand, one attempts to find the order of $[\mathfrak{c}]$ using a variation of the bounded primorial steps algorithm such that the exponent of the final giant step, $m^2\phi(P_w)P_w$, is as large as possible, while the total number of steps taken is still at most $O(p_t)$. Let η be the number of group operations performed during the exponentiation stage, and choose w to be as large as possible such that $\eta/2 \leq m\phi(P_w) < 3\eta/4$ for some integer m. Let $s = mP_w$ and $d = m\phi(P_w)$. Then take baby steps $[\mathfrak{c}]^i$ for $1 \leq i \leq s$ with i coprime to P_w. If any $[\mathfrak{c}]^i = [\mathcal{O}_\Delta]$, then we have computed the order $h' = i$ of $[\mathfrak{c}]$. Otherwise, we compute giant steps $[\mathfrak{c}]^j$ for $j = 2s, -2s, 4s, -4s, ..., 2sd, -2sd$. Notice that this sequence of giant steps is used since computing the inverse in the ideal class group is essentially free. If there exists a giant step $[\mathfrak{c}]^j$ such that $[\mathfrak{c}]^j = [\mathfrak{c}]^i$ for some corresponding baby step $[\mathfrak{c}]^i$, then we have computed the order $h' = j - i$ of $[\mathfrak{c}]$.

If we successfully compute $h' = \text{ord}([\mathfrak{c}])$ during the search stage, then assuming that $\text{ord}([\mathfrak{b}])$ is even, we compute an ambiguous ideal class by repeated squaring of $[\mathfrak{b}]^{h'}$, and then attempt to factor the integer associated with the discriminant. Since the order of $[\mathfrak{b}]^{h'}$ might not be even, we do not square more than $\ell = \left\lfloor \log_2 \sqrt{N} \right\rfloor$ times, as in the exponentiation stage. If the order is odd, we choose a different random ideal class and start over with the exponentiation stage. On the other hand, if we fail to compute the order of $[\mathfrak{c}]$ during the search stage, then we start from the beginning with a different square free multiplier k of the integer N.

The work by Lenstra and Pomerance [39, §11], precludes a subexponential running time complexity for both SPAR and SuperSPAR, however, their results dictate a worst case running time. Future work would include a median case complexity analysis of SuperSPAR (Section 9.3. In Chapter 8, we empirically search for parameters for SuperSPAR, such as the exponent E and the multiple of a primorial mP_w, in order to minimize the average running time to factor integers in the range where SuperSPAR is competitive with other factoring algorithms. Section **??** discusses several ways in which our implementation of SuperSPAR differs from the description given here in order to improve the performance of SuperSPAR in practice. Details of our implementation are given in Algorithm 8.4.

4.3 Summary

This chapter discusses SPAR, an integer factoring algorithm based on the ideal class group and our motivation for practical improvements to the performance of arithmetic and exponentiation in the class group. This chapter also discusses a primorial steps algorithm, due to Sutherland [50], which we apply to the search stage of SPAR. The resulting algorithm is referred to as "SuperSPAR". In the upcoming chapters, we detail our experiments and results to lead to practical improvements in ideal class arithmetic, exponentiation, and finally our implementation of SuperSPAR.

Chapter 5

Extended Greatest Common Divisor Experiments

One contribution of this thesis is an efficient implementation of arithmetic in the ideal class group of imaginary quadratic number fields. Much of the computational effort of ideal class multiplication (Algorithm 2.2) is in computing integral solutions to equations of the form

$$s = Ua + Vb$$

where a and b are fixed integers given as input, and s is the greatest common divisor of both a and b. Solutions to this equation are referred to as the *extended greatest common divisor* (or extended GCD for short). A first step to improving the performance of arithmetic in the ideal class group is to improve the performance of extended GCD computations. This chapter discusses several algorithms for computing solutions to the extended GCD.

Section 5.1 discuses the standard extended Euclidean Algorithm, which uses multiplication and division. Binary extended GCD computations, in contrast, emphasize bit shifting instead of multiplication and division. Section 5.2 discusses an extended GCD algorithm that works from low-order to high-order, referred to as *right-to-left* as this is the order in which bits are operated on in the binary representation. Subsection 5.2.1 discusses a windowed variant of this algorithm that operates on several bits of the input for each iteration. An extended GCD can also be computed from high-order to low-order, and is referred to as *left-to-right*. One such technique is discussed in Section 5.3. When the inputs are larger than a single machine word, Lehmer [38] observed that much of the computation of the extended GCD can still be performed using just the most significant machine word of the intermediates. Section 5.4 discusses Lehmer's standard extended GCD as well as some variations that we tried, namely precomputing intermediate solutions for 8-bit machine words and the use of a binary extended GCD for 64-bit machine words. Section 5.5 briefly describes each of these extended GCD computations in the context of the partial extended GCD, which is useful for computing the simple continued fraction expansion used by NUCOMP, NUDUPL, and NUCUBE from Subsections 2.5.5, 2.5.6, and 2.5.7 respective.

To improve the performance of these algorithms in practice, we specialized much of our implementation for the x64 architecture. The details of this specialization are discussed in Section 5.6. Many of our routines benefit when the input is bound by a single machine word, i.e. 64-bits, but we were also able to take advantage of integers that fit within two machine words by implementing a custom library for 128-bit arithmetic. When integers are larger than 128-bits, we use the GNU Multiple Precision (GMP) arithmetic library [3]. All the software in this chapter was developed using the GNU C compiler version 4.7.2 on Ubuntu 12.10. Assembly language was used for 128-bit arithmetic and processing features not available in the C language. The hardware platform was a 2.7GHz Intel Core i7-2620M CPU and 8Gb of memory. The CPU has four cores, only one of which was used during timing experiments. Section 5.7 shows the average time to compute the extended GCD for pairs of random positive integers (a, b) where a and b are the same number of bits in size. We use GMP [3], Pari [7], and MPIR [5] as reference implementations.

5.1 The Euclidean Algorithm

The Euclidean Algorithm (see [15, §9.3.2]) is an algorithm for computing the greatest common divisor of two integers. The input is the two positive integers a and b. At each iteration of the algorithm, we subtract the smaller of the two numbers from the larger, until one of them is 0. At this point, the non-zero number is the largest divisor of a and b. Since the smaller number may still be smaller after a single iteration, we use fewer steps by subtracting an integer multiple of the smaller number from the larger one.

The Euclidean Algorithm is extended by using a system of equations of the form

$$s = Ua + Vb \tag{5.1}$$

$$t = Xa + Yb. \tag{5.2}$$

Initially, let

$$\begin{bmatrix} s & U & V \\ t & X & Y \end{bmatrix} = \begin{bmatrix} a & 1 & 0 \\ b & 0 & 1 \end{bmatrix}$$

and Equations 5.1 and 5.2 hold. We maintain the invariant that $s \geq t$. When $s < t$, simply swap the rows in the matrix representation above. At each iteration, subtract $q = \lfloor s/t \rfloor$ times the second row from the first, and then swap rows to maintain the invariant. When $t = 0$, the first row of the matrix is a solution such that s is the largest positive divisor of a and b. The algorithm is given in Algorithm 5.1. Negative inputs a and b are handled by using $a' = |a|$ and $b' = |b|$ as inputs instead and modifying the output such that $U' = U \cdot \text{sign}(a)$ and $V' = V \cdot \text{sign}(b)$ where

$$\text{sign}(x) = \begin{cases} -1 & \text{when } x < 0 \\ 0 & \text{when } x = 0 \\ 1 & \text{when } x > 0. \end{cases}$$

Algorithm 5.1 Extended Euclidean Algorithm.

```
Input: $a, b \in \mathbb{Z}_{\geq 0}$
$\begin{bmatrix} s & U & V \\ t & X & Y \end{bmatrix} \leftarrow \begin{bmatrix} a & 1 & 0 \\ b & 0 & 1 \end{bmatrix}$
if $t > s$ then
    $\begin{bmatrix} s & U & V \\ t & X & Y \end{bmatrix} \leftarrow \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} s & U & V \\ t & X & Y \end{bmatrix}$    {Swap rows. Maintain $s \geq t$.}
while $t \neq 0$ do
    $q \leftarrow \lfloor s/t \rfloor$
    $\begin{bmatrix} s & U & V \\ t & X & Y \end{bmatrix} \leftarrow \begin{bmatrix} 0 & 1 \\ 1 & -q \end{bmatrix} \cdot \begin{bmatrix} s & U & V \\ t & X & Y \end{bmatrix}$    {Subtract $q$ times 2nd row and swap.}
return $(s, U, V)$    {Such that $s = Ua + Vb$.}
```

In practice, these operations are performed by manipulating each variable directly, rather than by using matrix arithmetic. Furthermore, we typically implement division with remainder, which solves $s = qt + r$ for $q, r \in \mathbb{Z}$ and $|r| < |t|$. Notice that $r = s - qt$ is the target value of t for each iteration of the Euclidean Algorithm.

5.2 Right-to-Left Binary Extended GCD

The extended Euclidean algorithm uses divide with remainder, which is often expensive. Binary extended GCD algorithms emphasize bit shifting over multiplication and division. Such algorithms may perform better than other extended GCD algorithms in practice (see Section 5.7 for results). Here we describe a binary extended GCD algorithm, originally published by Stein [49], that works from the least significant bit to the most significant bit. We refer to this as *right-to-left* since this is the direction in which we process the written binary representation.

To compute the greatest common divisor of two positive numbers, we repeatedly apply the following identities,

$$\gcd(a,b) = \begin{cases} 2 \cdot \gcd(a/2, b/2) & \text{when both } a \text{ and } b \text{ are even} \\ \gcd(a/2, b) & \text{when only } a \text{ is even} \\ \gcd(a, b/2) & \text{when only } b \text{ is even} \\ \gcd((a-b)/2, b) & \text{when } a \geq b \text{ and both are odd} \\ \gcd((b-a)/2, a) & \text{when } a < b \text{ and both are odd.} \end{cases}$$

In the case that only a is even, we divide a by 2 since 2 is not a common divisor of both. The same is true when only b is even. When both a and b are odd, their difference is even and so is further reduced by 2. Notice that each relation reduces at least one of the arguments and so the recursion terminates with either $\gcd(a, 0) = a$ or $\gcd(0, b) = b$.

When both a and b are even, $\gcd(a, b) = 2 \cdot \gcd(a/2, b/2)$. So the first step of the algorithm presented here is to remove all common powers of two from a and b. Let r be the number of times 2 is removed from both. Now either a or b or both are odd. If a is not odd, then swap a and b so that a is guaranteed to be odd. We will compute the extended GCD using this a and b rather than the input values. As such, the final solution to the original input is $s2^r = Ua2^r + Vb2^r$.

As before, begin with the matrix representation

$$\begin{bmatrix} s & U & V \\ t & X & Y \end{bmatrix} = \begin{bmatrix} a & 1 & 0 \\ b & 0 & 1 \end{bmatrix}.$$

An invariant of the algorithm is that s is odd at the beginning of each iteration. Since a was chosen to be odd, s is initially odd. First remove any powers of 2 from t. While t is even, we would like to apply the operation

$$(t, X, Y) \leftarrow \left(\frac{t}{2}, \frac{X}{2}, \frac{Y}{2}\right)$$

but this may result in rational values for X and Y if either are odd. We first point out that

$$\begin{aligned} t &= Xa + Yb \\ &= Xa + Yb + (ab - ab) \\ &= (X + b)a + (Y - a)b. \end{aligned}$$

As such, we can simultaneously add b to X and subtract a from Y when it suits us. To continue, we use the following theorem and proof, which we were unable to find in the literature.

Theorem 5.2.1. When t is even, either both X and Y are even, or Y is odd and both $X + b$ and $Y - a$ are even.

Proof. Assume t is even and that Y is odd. We have

$$\begin{aligned} & t \equiv Xa + Yb && \pmod 2 \\ \Rightarrow \quad & 0 \equiv X + b && \pmod 2 \quad \{\text{Since } t \text{ is even and } Y \text{ and } a \text{ are odd.}\} \\ \Rightarrow \quad & 0 \equiv X + b \equiv Y - a && \pmod 2 \quad \{\text{Since } Y - a \text{ is even.}\}. \end{aligned}$$

Now assume t is even and that X is odd. We have

$$\begin{aligned}
& t \equiv Xa + Yb && \pmod{2} && \\
\Rightarrow \quad & 0 \equiv 1 + Yb && \pmod{2} && \{\text{Since } t \text{ is even and } X \text{ and } a \text{ are odd.}\} \\
\Rightarrow \quad & 1 \equiv Yb && \pmod{2} && \{\text{Both } Y \text{ and } b \text{ are odd.}\} \\
\Rightarrow \quad & 0 \equiv X + b \equiv Y - a && \pmod{2}. &&
\end{aligned}$$

Therefore, if t is even, either both X and Y are even, or Y is odd and both $X + b$ and $Y - a$ are even. $\square$

By Theorem 5.2.1, we have a way to reduce t by 2 and maintain integer coefficients. While t is even,

$$(t, X, Y) \leftarrow \begin{cases} \left(\frac{t}{2}, \frac{X}{2}, \frac{Y}{2}\right) & \text{if } Y \text{ is even} \\ (t, X, Y) \leftarrow \left(\frac{t}{2}, \frac{X+b}{2}, \frac{Y-a}{2}\right) & \text{otherwise.} \end{cases}$$

At this point, both s and t are odd. If $s \geq t$ then let

$$\begin{bmatrix} s & U & V \\ t & X & Y \end{bmatrix} \leftarrow \begin{bmatrix} 0 & 1 \\ 1 & -1 \end{bmatrix} \cdot \begin{bmatrix} s & U & V \\ t & X & Y \end{bmatrix}$$

otherwise let

$$\begin{bmatrix} s & U & V \\ t & X & Y \end{bmatrix} \leftarrow \begin{bmatrix} 0 & 1 \\ -1 & 1 \end{bmatrix} \cdot \begin{bmatrix} s & U & V \\ t & X & Y \end{bmatrix}.$$

This ensures that s is odd, t is even, and that both s and t are positive. Repeat the steps of reducing t by powers of 2 and then subtracting one row from the other until $t = 0$. The complete algorithm is given in Algorithm 5.2. Note that in practice, integer division by 2 is performed using a bit shift right.

5.2.1 Windowed Right-to-Left Binary Extended GCD

Windowing is a common technique used to extend the base of an algorithm. We saw this earlier in our discussion of binary exponentiation in Section 3.1. The idea there was to

Algorithm 5.2 Right-to-left Binary Extended GCD (Stein [49]).

```
Input: $a, b, \in \mathbb{Z}_{>0}$.
let $r$ be the largest integer such that $2^r$ divides both $a$ and $b$
$a \leftarrow a/2^r, b \leftarrow b/2^r$
swap $a$ and $b$ if $a$ is not odd
$\begin{bmatrix} s & U & V \\ t & X & Y \end{bmatrix} \leftarrow \begin{bmatrix} a & 1 & 0 \\ b & 0 & 1 \end{bmatrix}$
while $t \neq 0$ do
    while $t$ is even do
        if $Y$ is odd then
            $(X, Y) \leftarrow (X + b, Y - a)$
        $(t, X, Y) \leftarrow \left(\frac{t}{2}, \frac{X}{2}, \frac{Y}{2}\right)$
    if $s \geq t$ then
        $\begin{bmatrix} s & U & V \\ t & X & Y \end{bmatrix} \leftarrow \begin{bmatrix} 0 & 1 \\ 1 & -1 \end{bmatrix} \cdot \begin{bmatrix} s & U & V \\ t & X & Y \end{bmatrix}$
    else
        $\begin{bmatrix} s & U & V \\ t & X & Y \end{bmatrix} \leftarrow \begin{bmatrix} 0 & 1 \\ -1 & 1 \end{bmatrix} \cdot \begin{bmatrix} s & U & V \\ t & X & Y \end{bmatrix}$
return $(s2^r, U, V)$ if $a$ and $b$ were not swapped and $(s2^r, V, U)$ otherwise
```

precompute g^w for each w in some window $0 \leq w < 2^k$ for some k and then to iterate over the exponent k bits at a time. We can apply this technique to the right-to-left extended binary GCD.

The algorithm repeatedly reduces either the equation $t = Xa + Yb$ or the equation $t = (X + b)a + (Y - a)b$ by 2. When Y is odd, we simultaneously add b to X and subtract a from Y in order to make both X and Y even. Suppose that t was a multiple of 4. We could simultaneously add b to X and subtract a from Y repeatedly until both X and Y were divisible by 4. Choose m such that $ma \equiv Y \pmod 4$, and then $t = (X + mb)a + (Y - ma)b$ is evenly divisible by 4 when t is divisible by 4.

This is easily extended for any 2^k where k is a positive integer. The algorithm first computes $x_j = mb$ and $y_j = ma$ for $0 \leq m < 2^k$ where $j = ma \bmod 2^k$. While t is divisible by 2^h for some $h \leq k$, we look up x_j and y_j for $j = Y \bmod 2^h$ and compute $(X + x_j)/2^h$ and $(Y - y_j)/2^h$. The complete Algorithm is given in listing 5.3.

Algorithm 5.3 Windowed Right-to-left Binary Extended GCD.

```
Input: $a, b, \in \mathbb{Z}_{>0}$ and let $k \in \mathbb{Z}_{>0}$ be the window size in bits.
let $r$ be the largest integer such that $2^r$ divides both $a$ and $b$
$a \leftarrow a/2^r, b \leftarrow b/2^r$
swap $a$ and $b$ if $a$ is not odd
for $m$ from $2^k - 1$ downto 0 do
    $j \leftarrow ma \bmod 2^k$
    $x_j \leftarrow mb$
    $y_j \leftarrow ma$
$\begin{bmatrix} s & U & V \\ t & X & Y \end{bmatrix} \leftarrow \begin{bmatrix} a & 1 & 0 \\ b & 0 & 1 \end{bmatrix}$
while $t \neq 0$ do
    while $t$ is even do
        let $h$ be the largest integer such that $h \leq k$ and $2^h$ divides $t$
        $j \leftarrow Y \bmod 2^h$
        $(t, X, Y) \leftarrow \left(\frac{t}{2^h}, \frac{X+x_j}{2^h}, \frac{Y-y_j}{2^h}\right)$        {Reduce by $2^h$}
    if $s \geq t$ then
        $\begin{bmatrix} s & U & V \\ t & X & Y \end{bmatrix} \leftarrow \begin{bmatrix} 0 & 1 \\ 1 & -1 \end{bmatrix} \cdot \begin{bmatrix} s & U & V \\ t & X & Y \end{bmatrix}$
    else
        $\begin{bmatrix} s & U & V \\ t & X & Y \end{bmatrix} \leftarrow \begin{bmatrix} 0 & 1 \\ -1 & 1 \end{bmatrix} \cdot \begin{bmatrix} s & U & V \\ t & X & Y \end{bmatrix}$
return $(s2^r, U, V)$ if $a$ and $b$ were not swapped and $(s2^r, V, U)$ otherwise
```

5.3 Left-to-Right Binary Extended GCD

Just as exponentiation can be performed from high-order to low-order, so too can an extended binary GCD computation. This is termed a left-to-right binary GCD, since it works from the left most bit to the right most bit of the written binary representation of the inputs.

Recall that at each iteration of the extended Euclidean Algorithm, we subtract $q = \lfloor s/t \rfloor$ times the equation $t = Xa + Yb$ from the equation $s = Ua + Vb$ and then swap (s, U, V) with (t, X, Y). Computing $q = \lfloor s/t \rfloor$ uses integer division, and then subtracting q times one equation from the other uses multiplication. Since it is not necessary to subtract exactly q times the equation, the idea is instead to use a value $q' = 2^k$ such that q' is *close* in some sense to q. Subtracting q' times the second equation from the first can then be done using a binary shift left by k bits.

Shallit and Sorenson [46] propose to select $q' = 2^k$ such that $q't \leq s < 2q't$. If $s - q't < 2q't - s$, compute

$$\begin{bmatrix} s & U & V \\ t & X & Y \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & -q' \end{bmatrix} \cdot \begin{bmatrix} s & U & V \\ t & X & Y \end{bmatrix},$$

otherwise, compute

$$\begin{bmatrix} s & U & V \\ t & X & Y \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -1 & 2q' \end{bmatrix} \cdot \begin{bmatrix} s & U & V \\ t & X & Y \end{bmatrix}.$$

Notice that after this step s has the previous value of t and that the binary representation of t is one digit shorter than the binary representation of the previous value of s, i.e. the the left most set bit of the previous value of s is now cleared. As with the extended Euclidean Algorithm, maintain the invariant that $s \geq t$; if after the above operation $s < t$, then swap the rows of the matrix to restore the invariant. The complete algorithm is given in Algorithm 5.4.

Algorithm 5.4 Shallit and Sorenson Left-to-Right Binary Extended GCD ([46]).

```
Input: $a, b \in \mathbb{Z}$
$\begin{bmatrix} s & U & V \\ t & X & Y \end{bmatrix} \leftarrow \begin{bmatrix} a & 1 & 0 \\ b & 0 & 1 \end{bmatrix}$
if $t > s$ then
    $\begin{bmatrix} s & U & V \\ t & X & Y \end{bmatrix} \leftarrow \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} s & U & V \\ t & X & Y \end{bmatrix}$        {Swap rows. Maintain $s \geq t$.}
while $t \neq 0$ do
    find $q = 2^k$ such that $qt \leq s < 2qt$
    if $s - qt < 2qt - s$ then
        $\begin{bmatrix} s & U & V \\ t & X & Y \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & -q \end{bmatrix} \cdot \begin{bmatrix} s & U & V \\ t & X & Y \end{bmatrix}$
    else
        $\begin{bmatrix} s & U & V \\ t & X & Y \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -1 & 2q \end{bmatrix} \cdot \begin{bmatrix} s & U & V \\ t & X & Y \end{bmatrix}$
    if $t > s$ then
        $\begin{bmatrix} s & U & V \\ t & X & Y \end{bmatrix} \leftarrow \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} s & U & V \\ t & X & Y \end{bmatrix}$        {Swap rows. Maintain $s \geq t$.}
return $(s, U, V)$
```

In practice, to find $q = 2^k$ we compute the number of bits in both s and t and then use the difference as a candidate for k. Let $k' = (\lfloor \log_2 s \rfloor + 1) - (\lfloor \log_2 t \rfloor + 1) = \lfloor \log_2 s \rfloor - \lfloor \log_2 t \rfloor$ be our candidate. If $t2^{k'} \leq s$ then $k = k'$, otherwise $k = k' - 1$. Notice that either $k = k'$ in which case $t2^k = t2^{k'}$, or $k = k' - 1$ and then $t2^{k+1} = t2^{k'}$. Either way $t2^{k'}$ can be reused for one half of the comparison of $s - qt < 2qt - s$. Also, if $k' = 0$ then $t \leq s$ (by our invariant) and so there is no possibility of using $t2^{-1}$, since we will use $t2^{k'}$ and $t2^{k'+1}$ in the comparison.

This approach requires us to first compare $t2^{k'}$ to s and then compare one of $t2^{k'-1}$ or $t2^{k'+1}$ to s in order to find which is closer. Because of this, we also experimented with a simplified version of the algorithm. We only compute $k = \lfloor \log_2 s \rfloor - \lfloor \log_2 t \rfloor$. Let $q = 2^k$ and compute

$$\begin{bmatrix} s & U & V \\ t & X & Y \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & -q \end{bmatrix} \cdot \begin{bmatrix} s & U & V \\ t & X & Y \end{bmatrix}.$$

If $qt > s$ then the resulting value for $s - qt$ is negative, and so we negate the first row of the product matrix to ensure that the new value for s is positive. The result is fewer comparisons for the inner loop of the GCD overall.

5.4 Lehmer's Extended GCD

In the extended Euclidean algorithm (Section 5.1), each iteration subtracts a multiple, $q = \lfloor s/t \rfloor$, of the smaller number from the larger number. Derrick Henry Lehmer [38] noticed that most of the quotients, q, were small and that those small quotients could be computed from the leading digits (or machine word) of the numerator and denominator.

The idea is similar to the extended Euclidean algorithm, only that there is an inner loop that performs an extended GCD computation using values that fit within a single machine word. As in the extended Euclidean algorithm, start by setting

$$\begin{bmatrix} s & U & V \\ t & X & Y \end{bmatrix} \leftarrow \begin{bmatrix} a & 1 & 0 \\ b & 0 & 1 \end{bmatrix}$$

for positive integers a and b. Assume that $s \geq t$ (if this is not the case, then swap the rows of the matrix). Compute $s' = \lfloor s/2^k \rfloor$ for some k such that s' fits within a single machine word and is as large as possible (if s already fits within a single machine word, then let $k = 0$), and compute $t' = \lfloor t/2^k \rfloor$ for the same value k. The idea is to then perform an extended GCD computation on the values s' and t' but only so long as the quotient ${q_i}' = \lfloor s'/t' \rfloor$, generated by the i^{th} step of a single precision extended Euclidean GCD computation, is equal to the quotient $q_i = \lfloor s/t \rfloor$ that would be generated by the i^{th} step of a corresponding full precision extended Euclidean GCD computation.

To determine when the single precision quotient ${q_i}'$ would differ from the full precision quotient q_i, we use a method originally proposed by Collins [17] and described by Jebelean [35, Theorem 1]. Let

$$\begin{bmatrix} s' & A & B \\ t' & C & D \end{bmatrix} \leftarrow \begin{bmatrix} s' & 1 & 0 \\ t' & 0 & 1 \end{bmatrix}$$

be the initial matrix consisting of single precision integers. We then perform a single precision extended GCD on the above matrix until

$$\begin{aligned} t' < -D \quad &\text{or} \quad s' - t' < C - A \quad \text{when } i \text{ is even,} \\ t' < -C \quad &\text{or} \quad s' - t' < D - B \quad \text{when } i \text{ is odd.} \end{aligned}$$

The resulting matrix

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix}$$

represents the concatenation of the operations performed during the single precision extended GCD. If $B \neq 0$, these operations are combined with the outer loop of the larger extended GCD by computing

$$\begin{bmatrix} s & U & V \\ t & X & Y \end{bmatrix} \leftarrow \begin{bmatrix} A & B \\ C & D \end{bmatrix} \cdot \begin{bmatrix} s & U & V \\ t & X & Y \end{bmatrix}.$$

We then continue with the outer loop of the computation until $t = 0$. In the event that $B = 0$, then s and t differ in length by more than a single machine word, and we use a step of the full precision extended Euclidean GCD computation to adjust their lengths.

Lehmer's original description [38] uses the extended Euclidean algorithm to compute each step of the single precision extended GCD, however, other single precision extended GCD computations will work. In Section 5.7, we give timing results for extended GCD computations using an implementation of Lehmer's extended GCD for both a single precision extended Euclidean GCD and a single precision left-to-right binary extended GCD. We also experimented with precomputing the result of the single precision extended GCD for 8-bit machine words. Pseudo-code for Lehmer's extended GCD is given in Algorithm 5.5.

Algorithm 5.5 Lehmer's extended GCD ([38]).

```
Input: $a, b, \in \mathbb{Z}$ and $a \geq b > 0$.
    Let $m$ be the number of bits in a machine word.
$\begin{bmatrix} s & U & V \\ t & X & Y \end{bmatrix} \leftarrow \begin{bmatrix} a & 1 & 0 \\ b & 0 & 1 \end{bmatrix}$
while $t \neq 0$ do
    $k \leftarrow \lfloor \log_2 s \rfloor + 1 - m$
    $s' \leftarrow \lfloor s/2^k \rfloor$                    {Shift right for most significant word.}
    $t' \leftarrow \lfloor t/2^k \rfloor$
    $\begin{bmatrix} A & B \\ C & D \end{bmatrix} \leftarrow \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$
    $i \leftarrow 0$
    while $t' \neq 0$ do
        perform one step of a single precision extended GCD on $\begin{bmatrix} s' & A & B \\ t' & C & D \end{bmatrix}$
        if $i$ is even then
            if ($t' < -D$ or $s' - t' < C - A$) then break
        else
            if ($t' < -C$ or $s' - t' < D - B$) then break
        $i \leftarrow i + 1$
    if $B = 0$ then
        $q \leftarrow \lfloor s/t \rfloor$                    {Full precision step.}
        $\begin{bmatrix} s & U & V \\ t & X & Y \end{bmatrix} \leftarrow \begin{bmatrix} 0 & 1 \\ 1 & -q \end{bmatrix} \cdot \begin{bmatrix} s & U & V \\ t & X & Y \end{bmatrix}$
    else
        $\begin{bmatrix} s & U & V \\ t & X & Y \end{bmatrix} \leftarrow \begin{bmatrix} A & B \\ C & D \end{bmatrix} \cdot \begin{bmatrix} s & U & V \\ t & X & Y \end{bmatrix}$                    {Combine step.}
```

5.5 Partial Extended GCD

Subsections 2.5.5, 2.5.6, and 2.5.7 describe algorithms to perform ideal class multiplication (NUCOMP), squaring (NUDUPL), and cubing (NUCUBE) respectively, where the result requires at most two reduction operations to be reduced. Each of these techniques involve computing a simple continued fraction, which is done using an extended GCD computation (see [34, §3.2]). Unlike the extended GCD algorithms previously described in this chapter, the extended GCD used to compute the simple continued fraction for ideal multiplication terminates once the next remainder computed is within a given bound. As such, this computation is referred to as the *partial extended GCD*.

The descriptions of NUCUBE, NUDUPL, and NUCUBE use the variables R_i and C_i. We relate these variables to the descriptions of the extended GCD in this chapter using

$$\begin{bmatrix} s & U & V \\ t & X & Y \end{bmatrix} = \begin{bmatrix} R_{i-1} & C_{i-1} & D_{i-1} \\ R_i & C_i & D_i \end{bmatrix}$$

for integers D_i. To expand the simple continued fraction a/b, let

$$\begin{bmatrix} R_{-2} & C_{-2} & D_{-2} \\ R_{-1} & C_{-1} & D_{-1} \end{bmatrix} = \begin{bmatrix} a & -1 & 0 \\ b & 0 & 1 \end{bmatrix}$$

and perform an extended GCD computation on the left hand side. As such, the coefficients of the partial extended GCD represent solutions to the equation

$$R_i = -C_i a + D_i b.$$

Since the coefficients D_i are never used, our implementation of the partial extended GCD does not compute them. This is similar to the case of computing the extended GCD in Equation 2.6, since the coefficient Y is never used.

By [19, §5.6.1], two ideals are equivalent if there exists an invertible integral linear change of variables from one ideal to the other. As such, each of the operations performed on the matrix

$$\begin{bmatrix} R_{i-1} & C_{i-1} \\ R_i & C_i \end{bmatrix}$$

during the partial extended GCD must be representable by an invertible 2 × 2 matrix with determinant ±1 (known as a *unimodular* matrix). This is because the resulting coefficients are applied to an ideal in order to reduce it, and so must have determinant ±1 for the ideal to remain in the same equivalence class. This is important since only extended GCD computations that are restricted to operations representable using invertible 2 × 2 matrices can be used for the partial extended GCD.

Of the extended GCD algorithms in this chapter, only the right-to-left binary extended GCD of Section 5.2 (and its windowed variants of Subsection 5.2.1) cannot be adapted to the partial extended GCD. The reason is due to the step where t is reduced by 2. The algorithm uses the computation

$$(t, X, Y) \leftarrow \begin{cases} \left(\frac{t}{2}, \frac{X}{2}, \frac{Y}{2}\right) & \text{if } Y \text{ is even} \\ (t, X, Y) \leftarrow \left(\frac{t}{2}, \frac{X+b}{2}, \frac{Y-a}{2}\right) & \text{otherwise.} \end{cases}$$

The step $(t, X, Y) \leftarrow (t/2, (X+b)/2, (Y-a)/2)$ cannot be expressed as a unimodular matrix.

The other extended GCD algorithms of this chapter can be adapted to the partial extended GCD by letting

$$\begin{bmatrix} R_{i-1} & C_{i-1} \\ R_i & C_i \end{bmatrix} = \begin{bmatrix} s & U \\ t & X \end{bmatrix}$$

and dropping the terms V and Y from the computation.

5.6 Specialized Implementations of the Extended GCD

To further improve performance, we specialized implementations of each of the GCD algorithms discussed in this section for the x64 architecture. All algorithms are implemented for the GNU C compiler version 4.7.2 and often benefit from hand optimized x64 assembler.

For each of the GCD algorithms presented here, we implemented 32-bit, 64-bit, and 128-bit versions.

Many of the techniques used here are described in the book “Hacker’s Delight” [52]. We use $\texttt{and}_2$ to denote bitwise ‘and’, $\texttt{or}_2$ for bitwise ‘or’, ‘$\oplus$’ for bitwise ‘exclusive or’, and ‘$\neg$’ for bitwise negation. Computing $x2^k$ corresponds to shifting x left by k bits, while computing the integer $\lfloor x/2^k \rfloor$ corresponds to shifting x right by k bits. To compute $x \bmod 2^k$ use $x \ \texttt{and}_2 \ (2^k - 1)$.

Assuming a two’s complement representation of machine words, the most significant bit of a machine word x is set if $x < 0$ and clear otherwise. Let m denote the number of bits in a machine word; then the result of an arithmetic shift right on x by $m - 1$ bits is either a word with m set bits when $x < 0$ or a word with m clear bits otherwise. Let $\texttt{sign\_mask}(x)$ denote this operation. Notice that a word with m set bits, corresponds to the integer -1 under a signed two’s complement representation.

Using these operations, the absolute value of a signed machine word x can be computed without using any conditional statements. Let $y = \texttt{sign\_mask}(x)$ and the absolute value of x is $(x \oplus y) - y$. To see this, suppose $x \geq 0$. Then y is 0 and so $(x \oplus y) - y = x$. When $x < 0$, y has all of its bits set (and is also -1). Therefore, $(x \oplus y) - y = \neg x + 1 = -x$.

Similarly, a word x can be conditionally negated depending on a bit-mask y. This is useful since the extended GCD algorithms described previously expect their input, a and b, to be positive integers. First compute the absolute value of a and b by computing a signed mask for each and then conditionally negate each. Since the extended GCD algorithm computes a solution to $s = Ua + Vb$, where a and b have been made positive, the solution for the original inputs is to conditionally negate U based on the signed mask of a, and V based on the signed mask of b.

Furthermore, many of the algorithms maintain the invariant that $s > t$ and that both are positive. Suppose the algorithm computes

$$\begin{bmatrix} s & U & V \\ t & X & Y \end{bmatrix} \leftarrow \begin{bmatrix} 0 & 1 \\ 1 & -q \end{bmatrix} \cdot \begin{bmatrix} s & U & V \\ t & X & Y \end{bmatrix}$$

when $s \geq t$ and

$$\begin{bmatrix} s & U & V \\ t & X & Y \end{bmatrix} \leftarrow \begin{bmatrix} 0 & 1 \\ -1 & q \end{bmatrix} \cdot \begin{bmatrix} s & U & V \\ t & X & Y \end{bmatrix}$$

otherwise. The conditional instruction is removed by first computing $m = \texttt{sign\_mask}(s - t)$ and then

$$\begin{bmatrix} s & U & V \\ t & X & Y \end{bmatrix} \leftarrow \begin{bmatrix} 0 & 1 \\ 1 & -q \end{bmatrix} \cdot \begin{bmatrix} s & U & V \\ t & X & Y \end{bmatrix}$$

and conditionally negating the triple (t, X, Y) based on the signed mask m.

To swap two machine words, x_0 and y_0, without using additional words, compute $x_1 = x_0 \oplus y_0$, $y_1 = x_1 \oplus y_0$, and then $x_2 = x_1 \oplus y_1$. Notice that x_2 expands to

$$x_2 = (x_0 \oplus y_0) \oplus ((x_0 \oplus y_0) \oplus y_0)$$

and this reduces to $x_2 = y_0$. Also, y_1 expands to $(x_0 \oplus y_0) \oplus y_0$, which is just x_0. In practice, each assignment to x_i and y_i overwrites the previous x_{i-1} and y_{i-1} and so is performed in place.

Two words, x and y, are conditionally swapped when $x < y$ by using two additional words. Let $d = x - y$. Notice that simply computing $x \leftarrow x - d$ and $y \leftarrow y + d$ swaps x and y. Instead, let $m = \texttt{sign\_mask}(d)$ and then $x \leftarrow x - (d \ \texttt{and}_2\ m)$ and $y \leftarrow y + (d \ \texttt{and}_2\ m)$ swaps x and y only when $x < y$. In the left-to-right binary GCD, we conditionally swap the triple (s, U, V) with (t, X, Y) when $s < t$. By fixing $m = \texttt{sign\_mask}(s - t)$, but letting d take on $s - t$, and then $U - X$, and finally $V - Y$, we conditionally swap the triples when $s < t$. The algorithm is given in Algorithm 5.6. On the x64 architecture we optimize the computation of $d \leftarrow s - t$ and $m \leftarrow \texttt{sign\_mask}(d)$ since the operation $s - t$ sets the carry flag when the result is negative. Using a subtract with borrow, we subtract m from itself. This sets m to 0 when the carry is clear, and -1 when the carry is set.

Algorithm 5.6 Conditionally swap (s, U, V) with (t, X, Y) when $s < t$.

```
$d \leftarrow s - t$
$m \leftarrow \texttt{sign\_mask}(d)$
$d \leftarrow d \ \texttt{and}_2\ m$
$s \leftarrow s - d$
$t \leftarrow t + d$
$d \leftarrow (U - X) \ \texttt{and}_2\ m$
$U \leftarrow U - d$
$X \leftarrow X + d$
$d \leftarrow (V - Y) \ \texttt{and}_2\ m$
$V \leftarrow V - d$
$Y \leftarrow Y + d$
```

Often our implementation uses the number of bits in a positive integer x. In the algorithm description we use $\lfloor \log_2 x \rfloor + 1$. On the x64 architecture the instruction `bsr` returns the index of the most significant set bit, while the instruction `bsf` return the index of the least significant set bit. This allows us to quickly compute either value. On other platforms, we use a logarithmic search to find the most (or least) significant set bit. To find the most significant bit, let $k = \lfloor m/2 \rfloor$ where m is the number of bits in a machine word and let i be the computed index (initially $i \leftarrow 0$). We first check if $x \geq 2^k$. If it is, then $i \leftarrow i + k$ and x is shifted right by k. We repeat on $k \leftarrow \lfloor k/2 \rfloor$ until $k = 0$. At which point, i is the index of the most significant set bit (see Algorithm 5.7). Notice that we can use the signed mask of $2^k - x$ instead of the conditional 'if' statement, and we can unroll the while loop for fixed values of m.

Algorithm 5.7 Return the index of the most significant set bit of x.

```
$i \leftarrow 0$
$k \leftarrow \lfloor m/2 \rfloor$                {$m$ is the number of bits in a machine word.}
while $k \neq 0$ do
    if $x \geq 2^k$ then
        $x \leftarrow \lfloor x/2^k \rfloor$
        $i \leftarrow i + k$
return $i$
```

Finally, Lehmer's extended GCD algorithm is especially useful when the inputs are several machine words. For 128-bit inputs, we implemented Lehmer's extended GCD using the extended Euclidean algorithm for 32 and 64-bit machine words, and our simplified left-to-right binary extended GCD for 64-bit machine words. For 32 and 64-bit inputs, we implemented Lehmer's extended GCD using 8-bit machine words. In this case, it is possible to precompute the resulting 2×2 matrix for all possible 8-bit values for the two inputs, i.e. all 65536 pairs of values. The coefficients of the resulting 2×2 matrix can be bound by 8-bits each, and so the resulting table requires 256Kb of memory. This simplifies the extended GCD computation since the inner loop of Lehmer's extended GCD becomes a lookup from this table.

5.7 Experimental Results

We specialized the implementation of each of the extended GCD algorithms for 32-bit, 64-bit, and 128-bit arithmetic. For each k from 1 to 127, we generated 1,000,000 pairs of pseudorandom positive integers of no more than k bits each. For the 32-bit implementations, we measured the time to compute the extended GCD of each pair for $1 \leq k \leq 31$, then for the 64-bit implementations, we measure the time for each pair for $1 \leq k \leq 63$, and finally for the 128-bit implementations, we measure the time for each pair for $1 \leq k \leq 127$. Only positive integers were considered since the prologue and epilogue code to handle negative inputs is the same in each implementation, namely the sign of the inputs is recorded, the inputs are made positive, the extended GCD is computed, and then the sign of the output coefficients is corrected. The source for these experiments was developed using the GNU C compiler version 4.7.2 on Ubuntu 12.10. The experiments were performed on a 2.7GHz Intel Core i7-2620M CPU with 8Gb of memory. The CPU has four cores, only one of which was used during testing.

To verify that this work has lead to practical performance improvements, we use Pari 2.5.3, MPIR 2.6.0, and GMP 5.1.1 as reference implementations of the extended GCD. In the case of both GMP and MPIR, we use instances of `mpz_t` modified to use at most two

64-bit words of locally allocated stack space for limbs. The only operations timed are calls to `mpz_xgcd`. Since MPIR is a fork of GMP, we simply change the `#include` directive, recompile, and then link against the appropriate library. For Pari, we link directly with the Pari library from C. For 64-bit integers, Pari `GEN` objects are created using `stoi`, while for 128-bit integers we use `mkintn(4, ...)`. All pairs of input integers are first converted to Pari `GEN` objects. As such, Pari timings only include calls to `bezout`, the symbol for Pari's implementation of the extended GCD. Garbage collection in Pari is performed by recording Pari's stack pointer (`avma`) and then restoring it at the end of each batch timing. For each of Pari, MPIR, and GMP, we statically link with the corresponding library.

We present our data in three sections, relating to our implementations specialized for 32-bit, 64-bit, and 128-bit arithmetic. In each case, we show that our 32-bit implementation is no slower than our 64-bit implementation, which is no slower than our 128-bit implementation. For input integers smaller than 32-bits, our implementation of the standard extended Euclidean algorithm performs best. For input integers larger than 31-bits but smaller than 64-bits, our implementation of a simplified left-to-right binary extended GCD is the fastest. Finally for input integers larger than 63-bits but smaller than 128-bits, our implementation of a simplified left-to-right binary extended GCD is fastest up to 118-bits and then our implementation of Lehmer's extended GCD using our simplified left-to-right binary extended GCD for 64-bit machine words becomes faster. For input integers larger than 127-bits, we defer to one of the reference implementations.

Ideal class arithmetic uses a variant of the extended GCD where only s and U are computed for the solution $s = Ua+Vb$ for input integers a and b. We refer to this as a *left-only* extended GCD. Ideal class arithmetic also uses the partial extended GCD, which is similar to the left-only extended GCD but the algorithm is terminated when the next quotient is less than a specified bound, and the initial value for U is -1. Future work (see Section 9.1) would test each implementation of the left-only and partial extended GCD specifically, however, we simply use the associated version of the left-only and partial extended GCD for whichever version of the full extended GCD performs best for inputs of a specific size. Since none of the reference implementations tested provide a partial extended GCD, we implemented one for `mpz_t` types using Lehmer's partial extended GCD and a 64-bit extended Euclidean GCD for the inner extended GCD.

5.7.1 32-bit Extended GCDs

Figures 5.1 through 5.12 show performance timings for each of our implementations of the extended GCD, as well as the reference implementations, for inputs less than 32-bits. For each bit size k from 1 through 31, a set of 1,000,000 pseudorandom positive integers of at most k bits was created. The times shown are the average time for a single extended GCD computation. In each case, our 32-bit implementations perform faster than our corresponding 64-bit and 128-bit implementations. Figure 5.1 shows that for inputs of this size, Pari is the fastest of the reference implementations tested. Figure 5.8 shows the performance of our 32-bit implementations of Stein's right-to-left binary extended GCD for window sizes from 1-bit to 5-bits. Of these implementations, the 1-bit window is fastest for inputs up to 15-bits, and then the 3-bit window is faster up to 29-bits. All the implementations tested here are known to be produce correct results for inputs up to 31-bits, with the exception of our 32-bit implementations of Stein's right-to-left binary extended GCD, which is only known to be correct up to 29-bits. Figure 5.12 shows the best performing implementations from among the reference implementations and each of our 32-bit implementations. In the case of our implementations of Stein's right-to-left binary extended GCD, this figure only includes the 1-bit and 3-bit windowed implementations, since these were the best performing of the windowed implementations. This figure shows that for inputs less than 32-bits, our 32-bit implementation of the extended Euclidean performs best, and takes roughly only 63% as long as the fastest reference implementation, for inputs of this size.

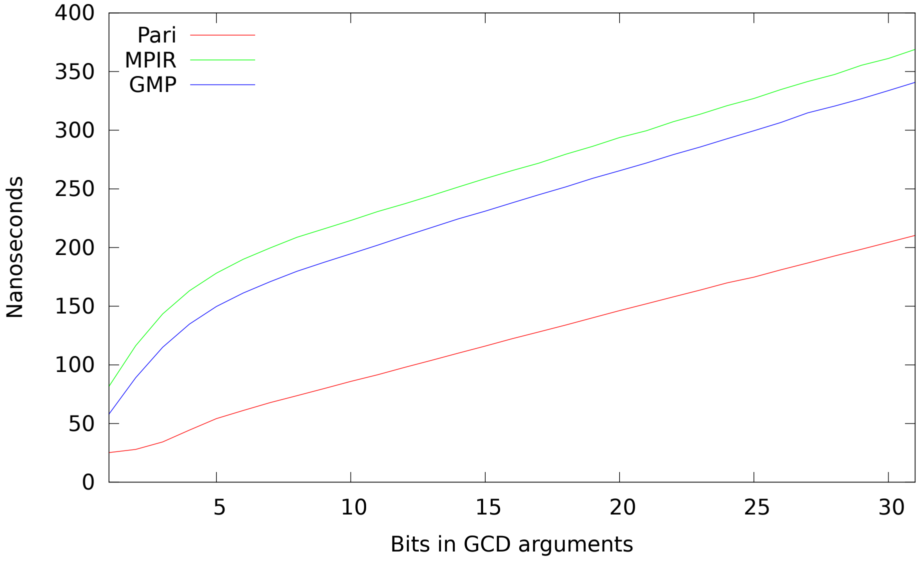

Figure 5.1: Reference Implementations for 32-bit Inputs.

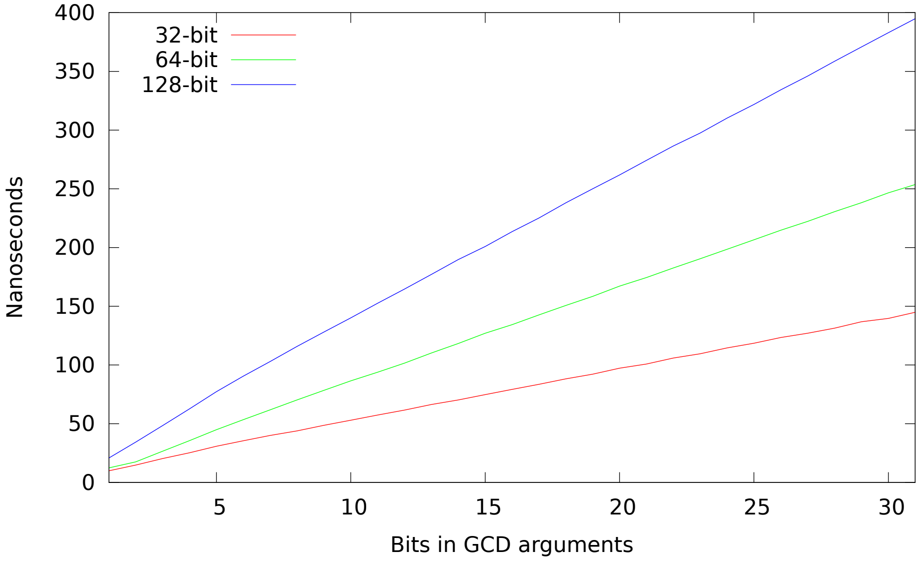

Figure 5.2: Extended Euclidean GCD for 32-bit Inputs.

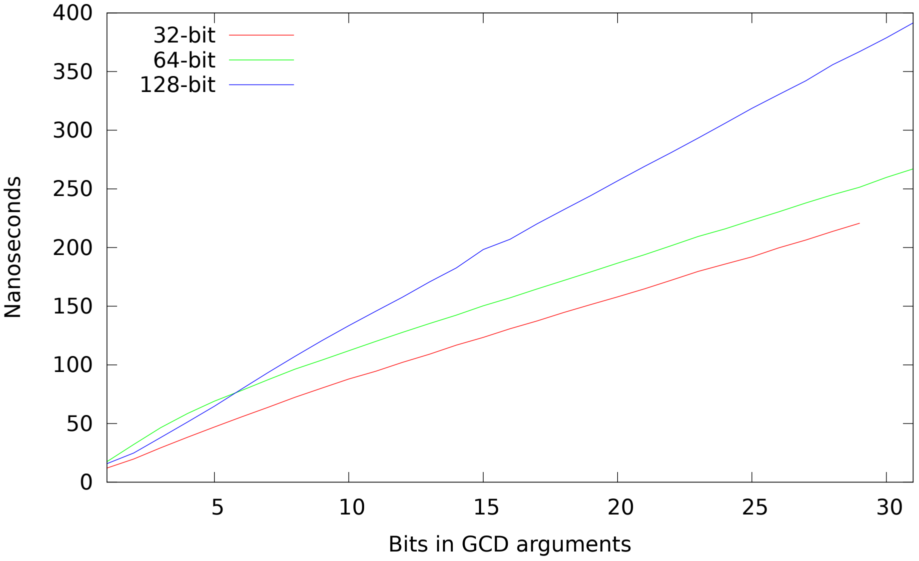

Figure 5.3: Stein's Extended GCD for 32-bit Inputs.

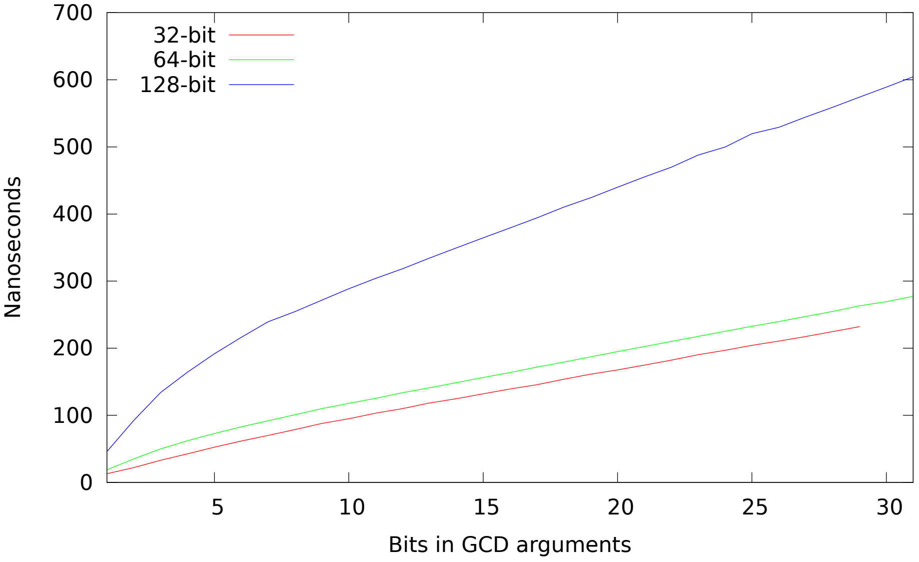

Figure 5.4: 2-bit Windowed Stein's Extended GCD for 32-bit Inputs.

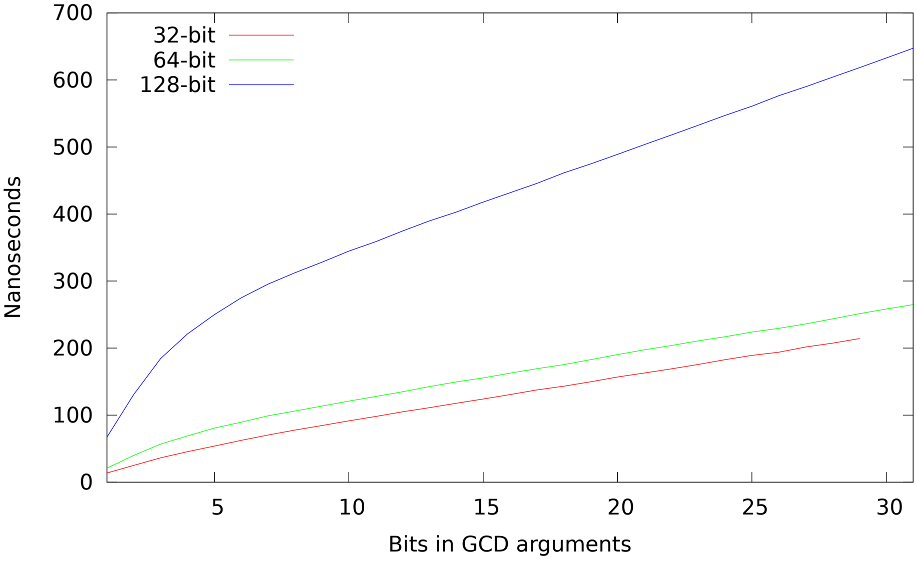

Figure 5.5: 3-bit Windowed Stein's Extended GCD for 32-bit Inputs.

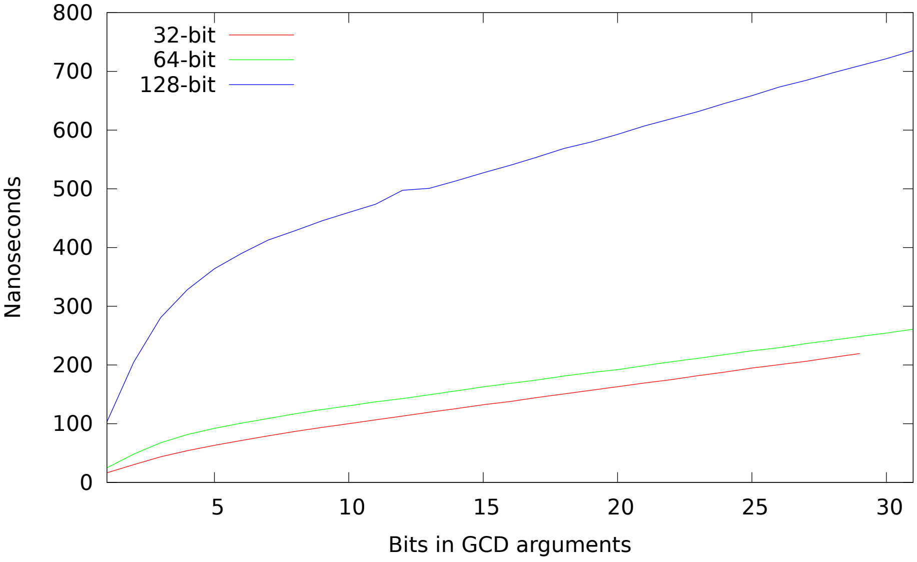

Figure 5.6: 4-bit Windowed Stein's Extended GCD for 32-bit Inputs.

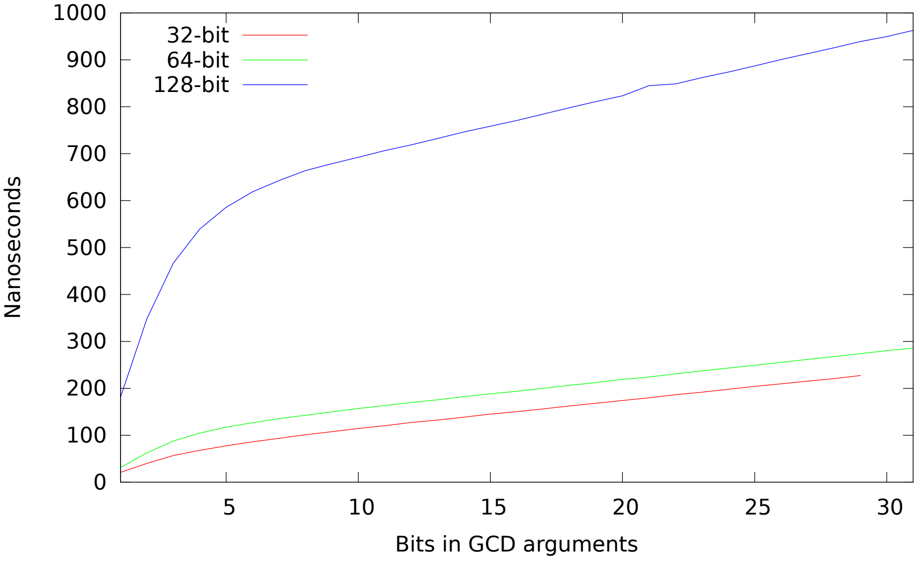

Figure 5.7: 5-bit Windowed Stein's Extended GCD for 32-bit Inputs.

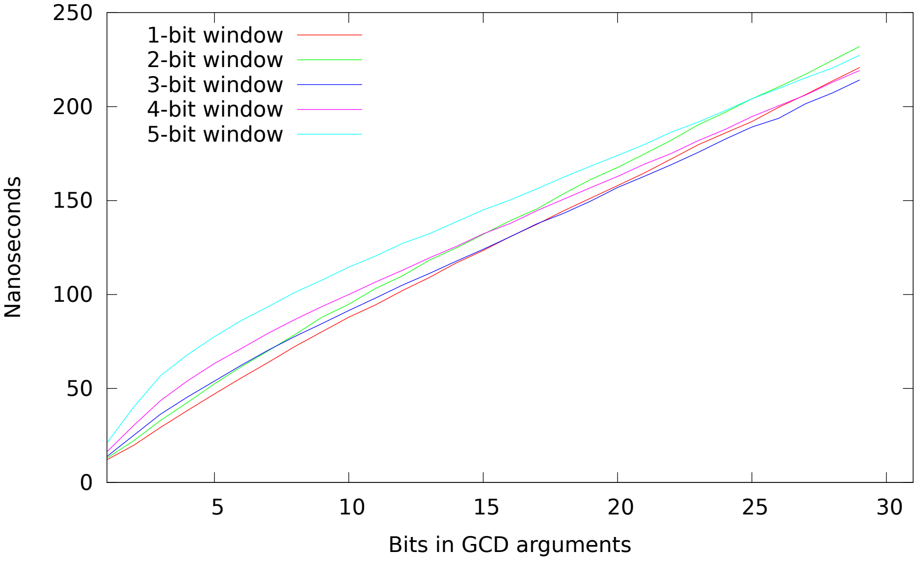

Figure 5.8: All 32-bit Implementations of Stein's Extended GCD.

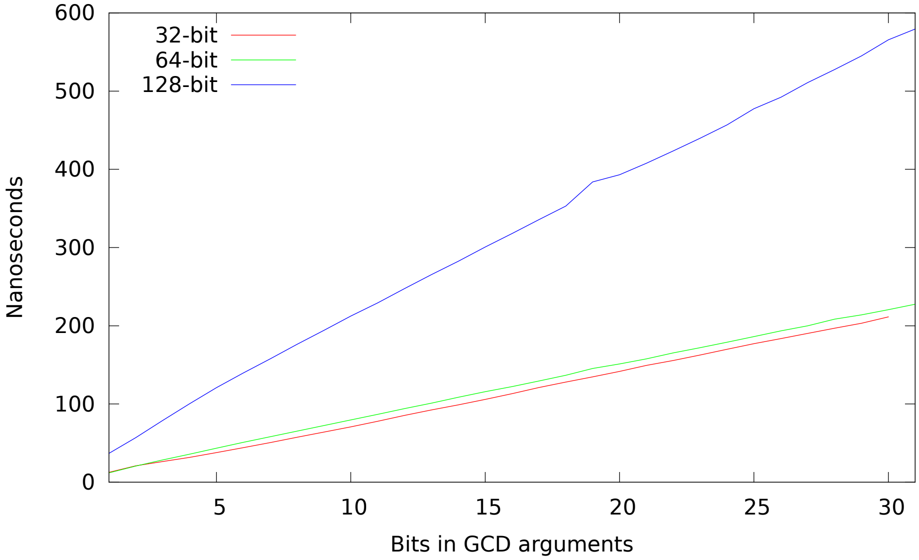

Figure 5.9: Shallit's Left-to-Right Binary Extended GCD for 32-bit Inputs.

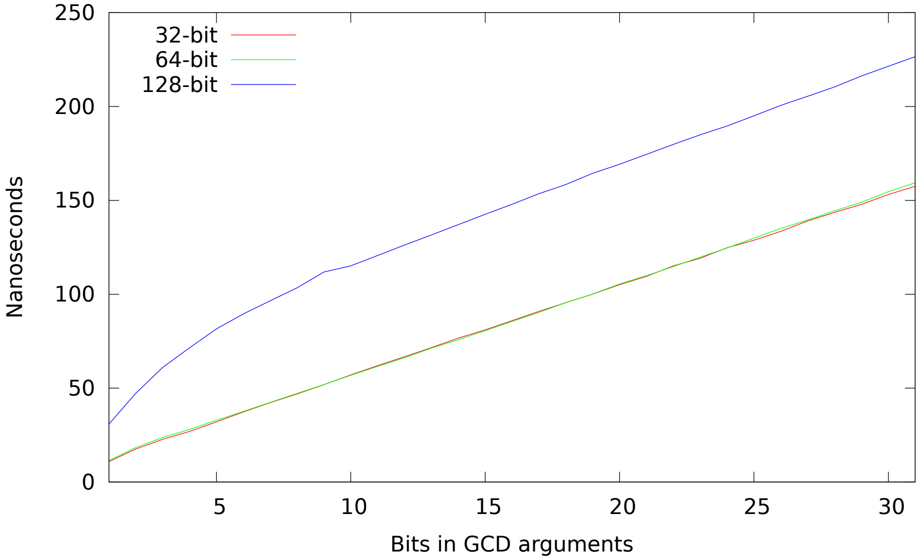

Figure 5.10: Simplified Left-to-Right Binary Extended GCD for 32-bit Inputs.

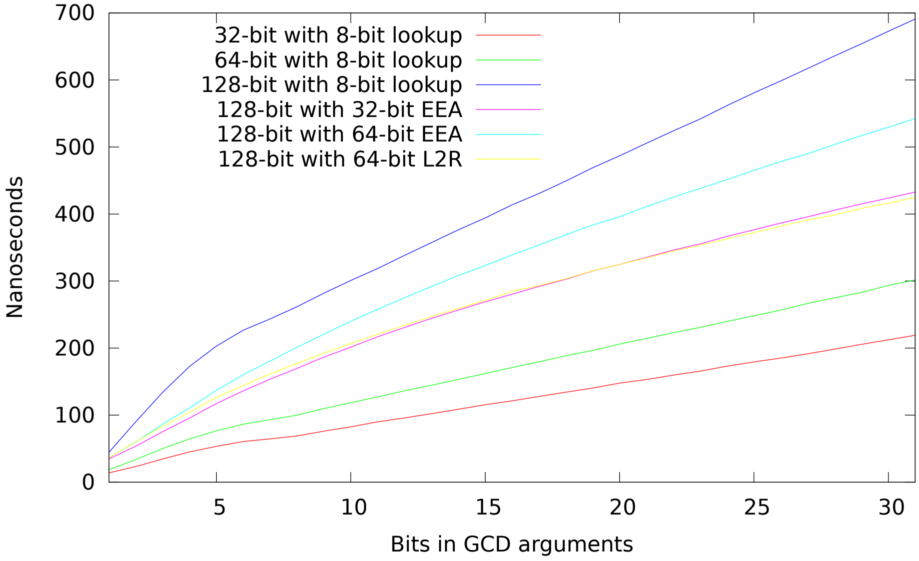

Figure 5.11: Lehmer's Extended GCD for 32-bit Inputs.

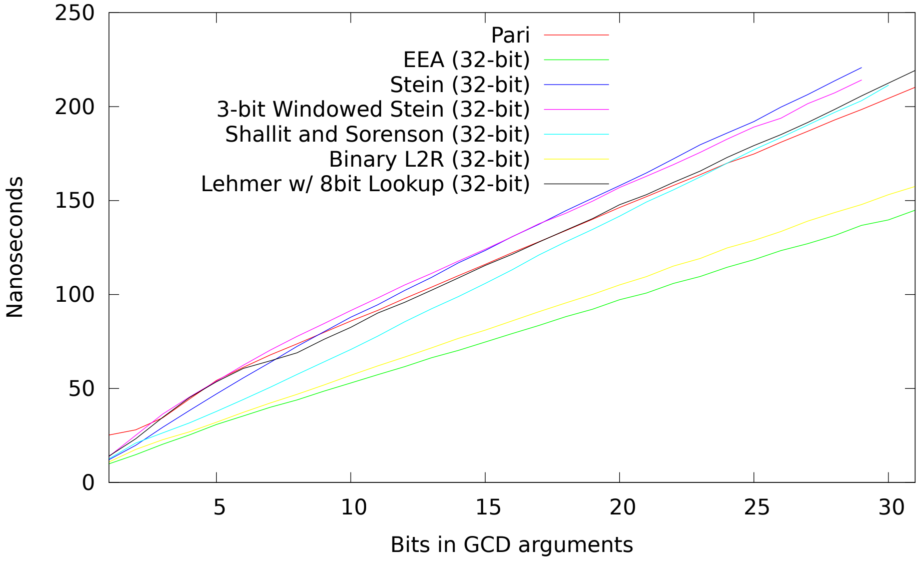

Figure 5.12: Best Extended GCDs for 32-bit Inputs.

5.7.2 64-bit Extended GCDs

Figures 5.13 through 5.24 show the average time to compute the extended GCD for a pair of pseudorandom positive integers of k bits for k from 1 through 63 using each of our 64-bit and 128-bit implementations of the extended GCD, as well as for the reference implementations. For each bit size, k, 1,000,000 pairs of pseudorandom positive integers were generated and used as inputs for each implementation of the extended GCD. In each case, our 64-bit implementation is faster than the corresponding 128-bit implementation. Figure 5.13 shows that for inputs of this size, Pari performs the fastest of the reference implementations. Figure 5.20 shows each of our 64-bit implementations of Stein's right-to-left binary extended GCD for window sizes from 1-bit to 5-bits. Of these implementations, the implementation using a 4-bit window is consistently faster than our other windowed implementations. Finally, Figure 5.24 shows each of our 64-bit implementations, as well as the fastest reference implementation, Pari. This figure only includes timings for our 4-bit windowed implementation of Stein's right-to-left binary extended GCD. This figure shows that for inputs larger than 31-bits but less than 64-bits that our simplified left-to-right binary extended GCD is the fastest and takes roughly 77% of the time of Pari. Each of the implementations tested in this subsection produce correct results up to 63-bits, with the exception of our implementations of Stein's right-to-left binary extended GCD, which only produce correct results for inputs up to 60-bits.

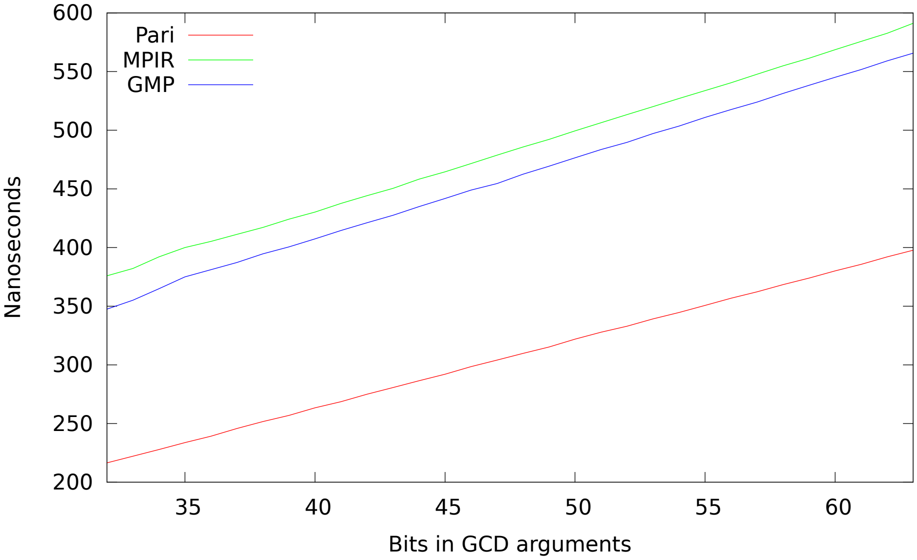

Figure 5.13: Reference Implementations for 64-bit Inputs.

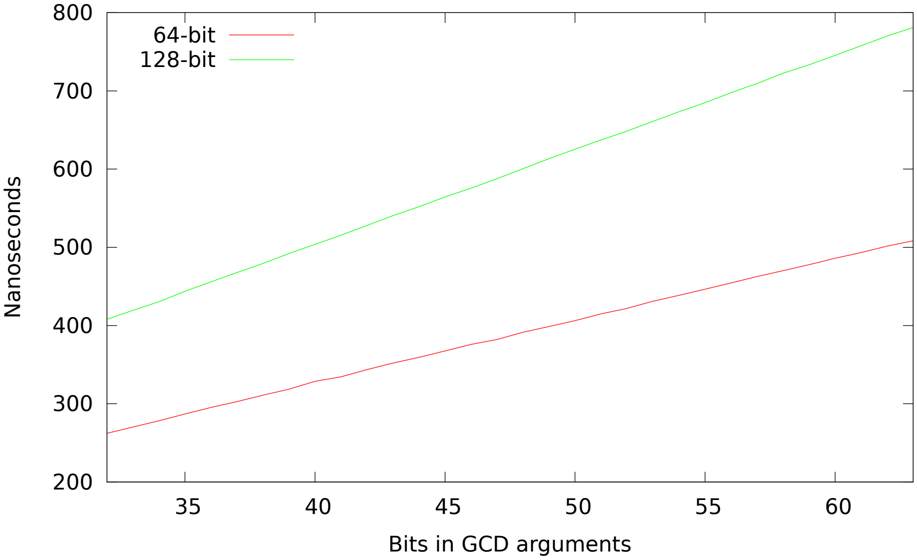

Figure 5.14: Extended Euclidean GCD for 64-bit Inputs.

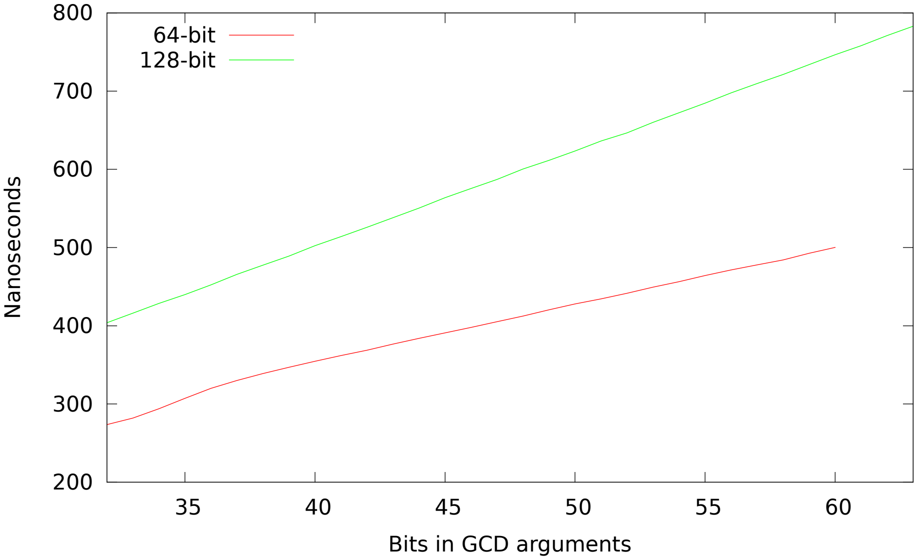

Figure 5.15: Stein's Extended GCD for 64-bit Inputs.

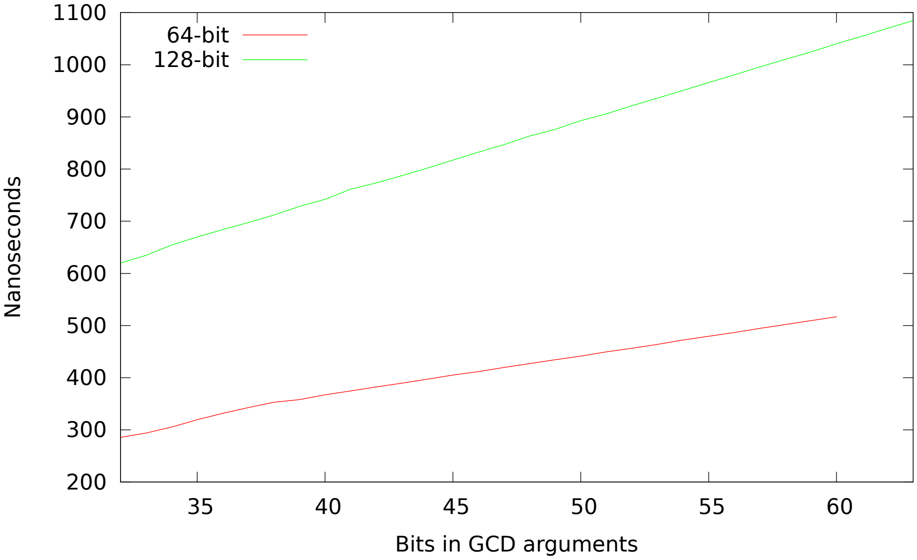

Figure 5.16: 2-bit Windowed Stein's Extended GCD for 64-bit Inputs.

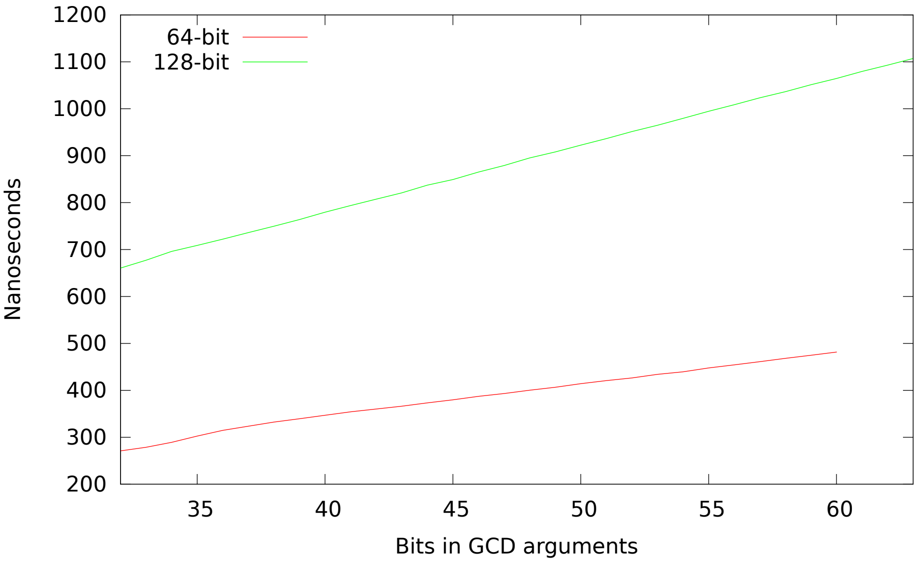

Figure 5.17: 3-bit Windowed Stein's Extended GCD for 64-bit Inputs.

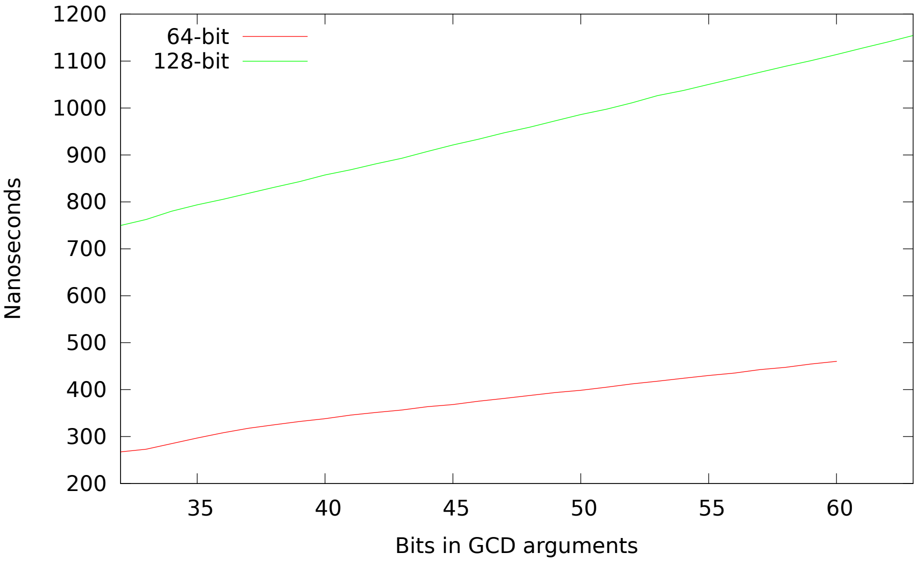

Figure 5.18: 4-bit Windowed Stein's Extended GCD for 64-bit Inputs.

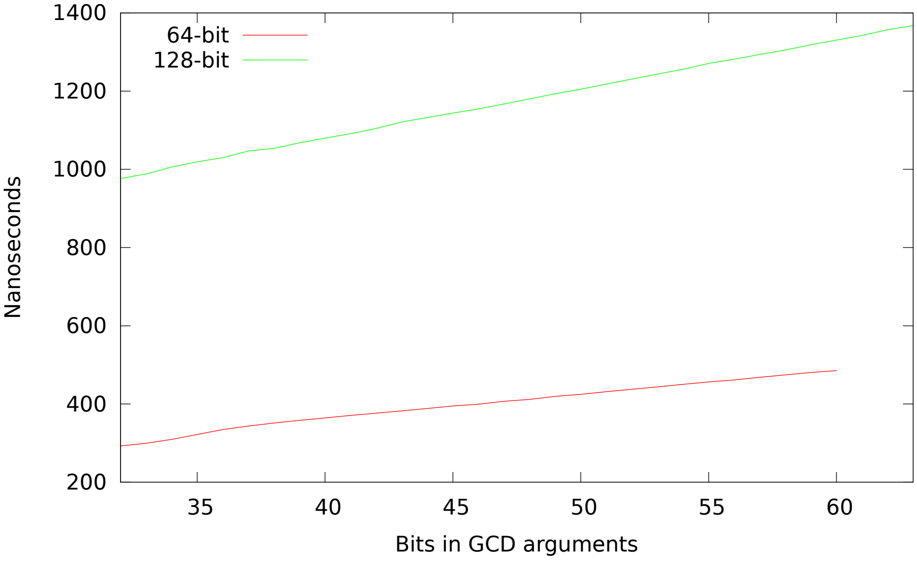

Figure 5.19: 5-bit Windowed Stein's Extended GCD for 64-bit Inputs.

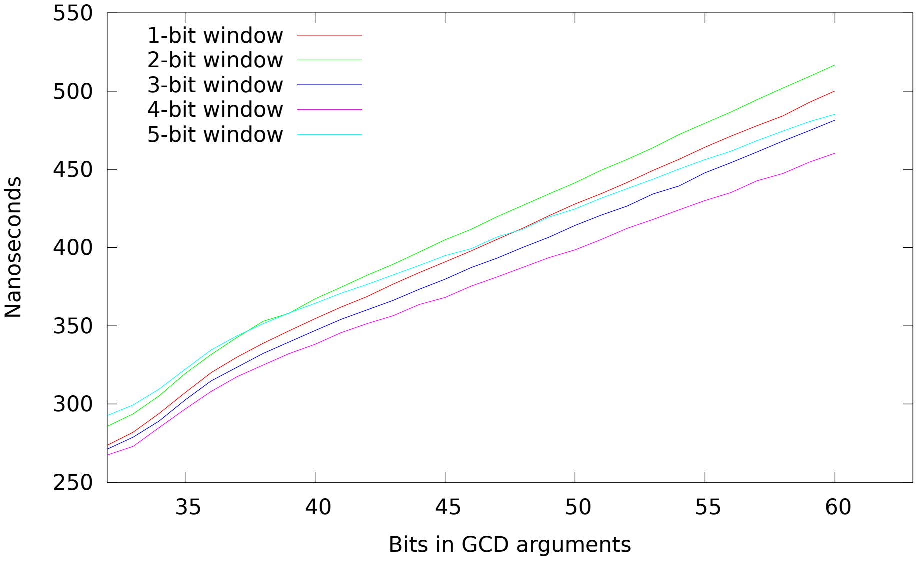

Figure 5.20: All 64-bit Implementations of Stein's Extended GCD.

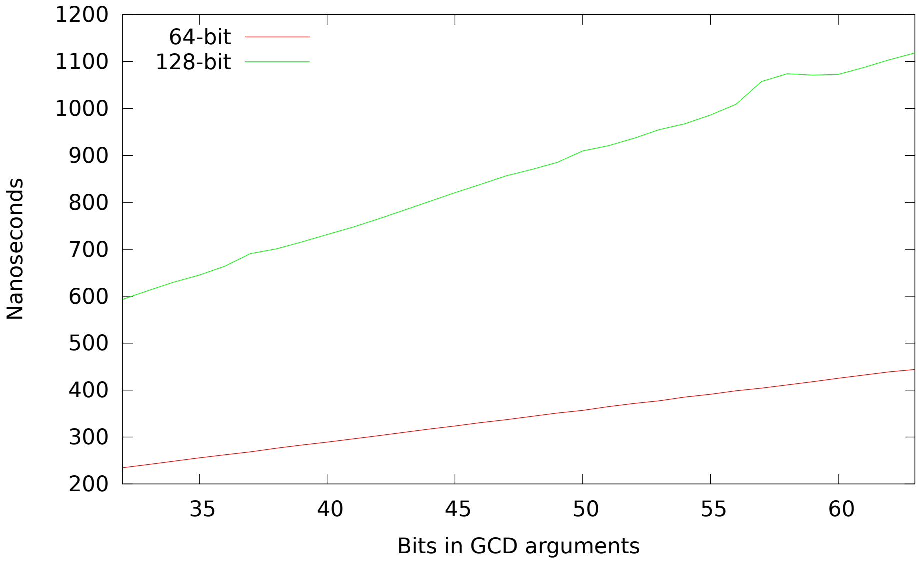

Figure 5.21: Shallit's Left-to-Right Binary Extended GCD for 64-bit Inputs.

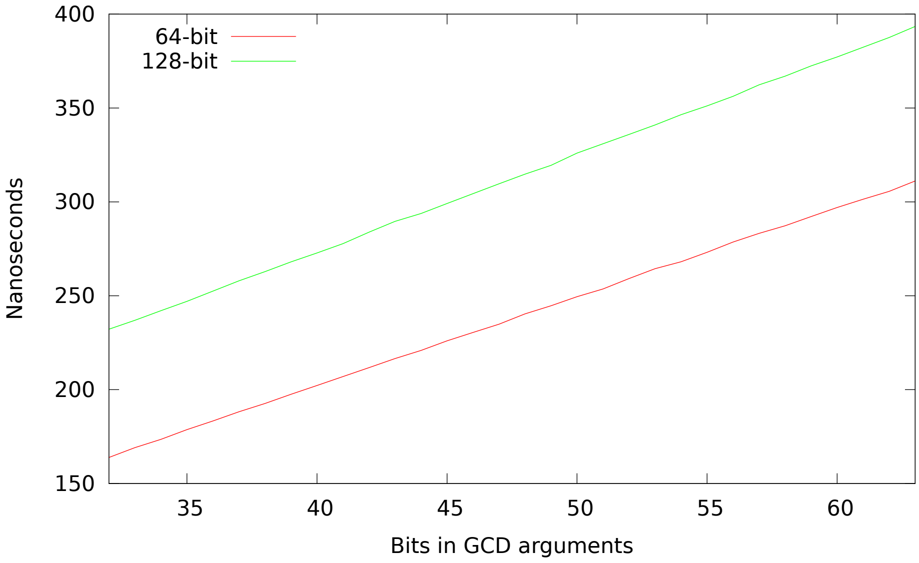

Figure 5.22: Simplified Left-to-Right Binary Extended GCD for 64-bit Inputs.

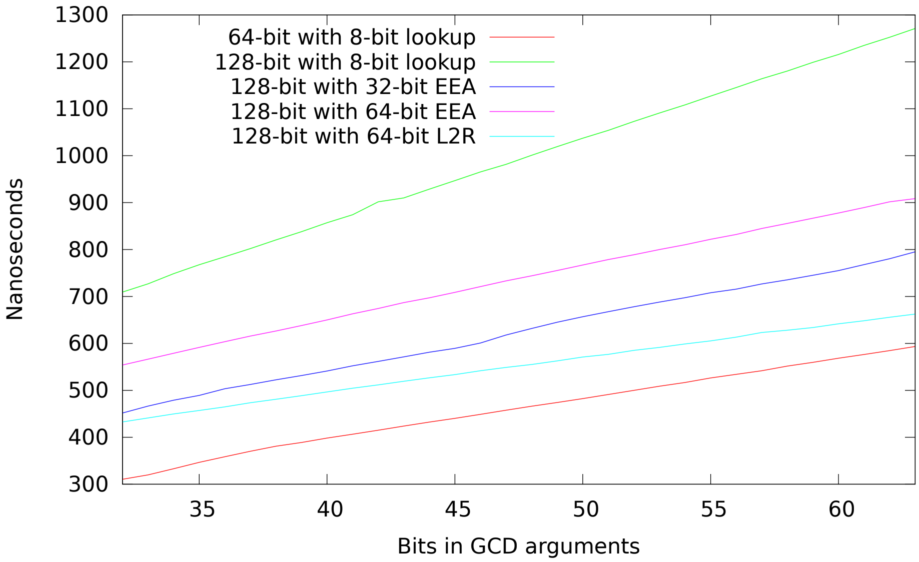

Figure 5.23: Lehmer's Extended GCD for 64-bit Inputs.

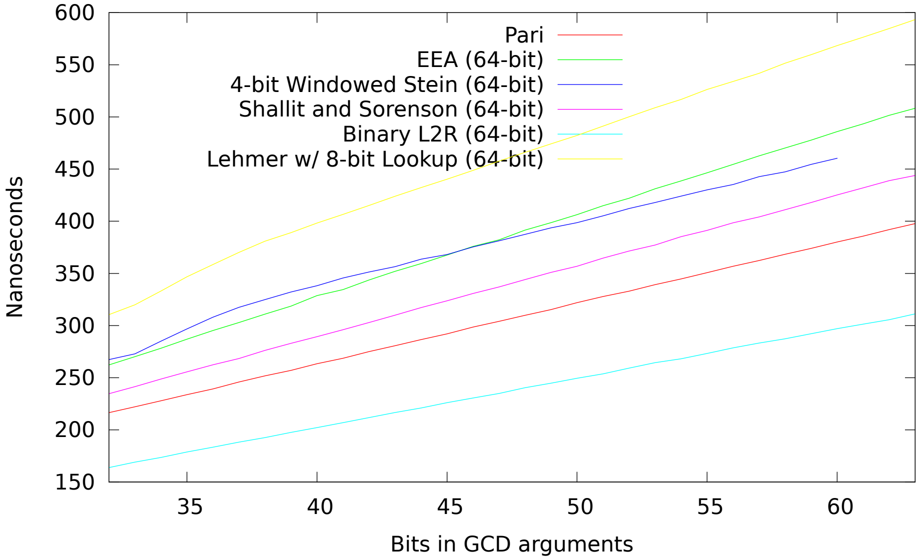

Figure 5.24: Best Extended GCDs for 64-bit Inputs.

5.7.3 128-bit Extended GCDs

Figures 5.25 through 5.28 show the performance timings of each of our 128-bit implementations of the extended GCD, including the reference implementations. Figure 5.25 shows that, of the reference implementations, Pari and GMP are fastest on average. Figure 5.26 show each of our windowed implementations of Stein's right-to-left binary extended GCD, and that our non-windowed implementation is fastest on average. Figure 5.27 shows each of our implementations of a 128-bit Lehmer's extended GCD. Our implementation of a 128-bit Lehmer's extended GCD using a 64-bit implementation of our simplified left-to-right binary extended GCD is the fastest on average among these. Finally, Figure 5.28 shows the best of each of our 128-bit implementations as well as the best of the reference implementations. In this case, our 128-bit implementation of our simplified left-to-right binary extended GCD is the best performing on average for inputs no larger than 118-bits, otherwise our implementation our 128-bit implementation of Lehmer's extended GCD using our 64-bit simplified left-to-right binary extended GCD is fastest on average.

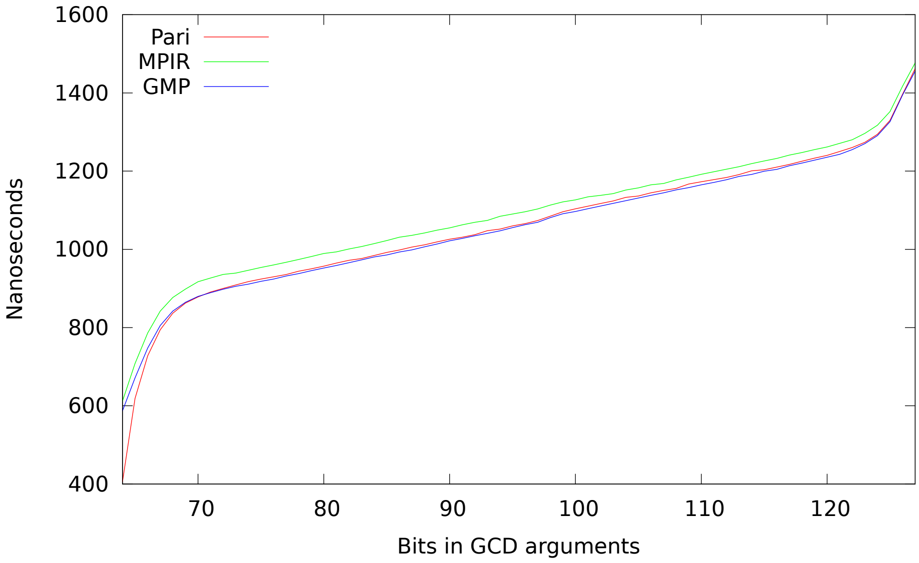

Figure 5.25: Reference Implementations for 128-bit Inputs.

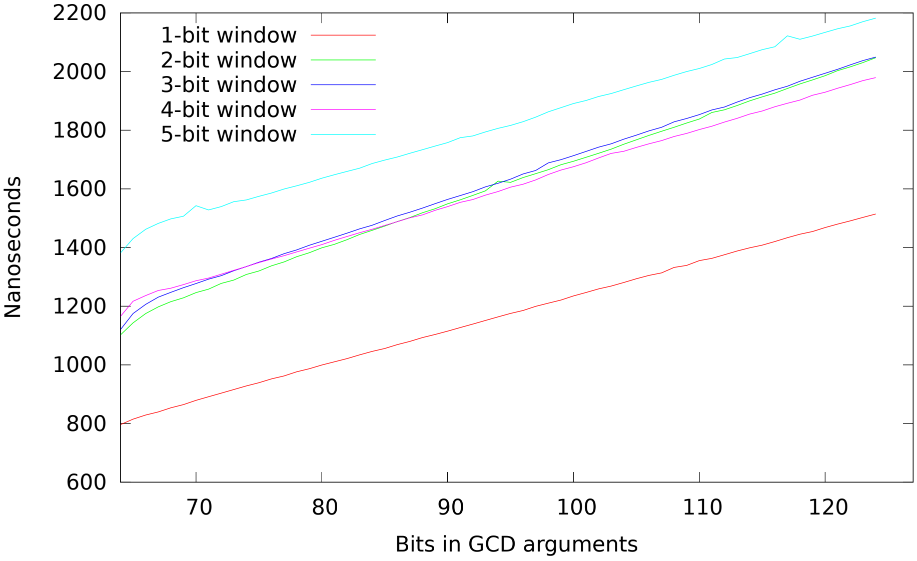

Figure 5.26: All 128-bit Implementations of Stein's Extended GCD.

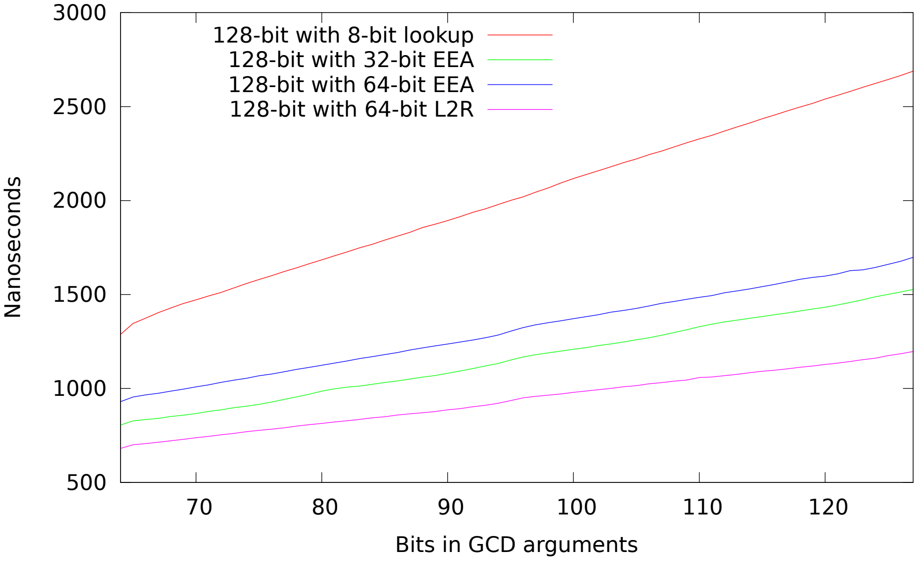

Figure 5.27: Lehmer's Extended GCD for 128-bit Inputs.

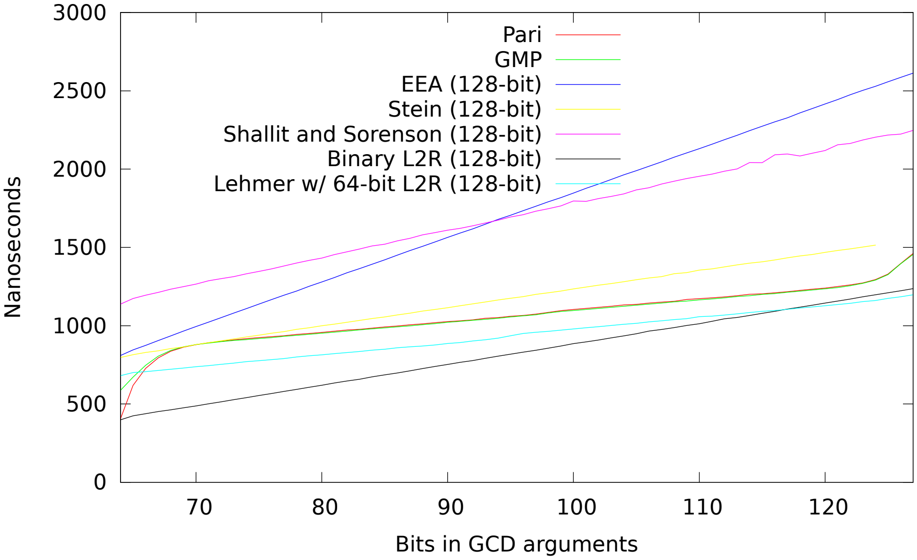

Figure 5.28: Best Extended GCDs for 128-bit Inputs.

5.8 Summary

We briefly restate the results in this section. For inputs less than 32-bits, our 32-bit implementation of the extended Euclidean algorithm is the fastest on average and on average takes roughly 63% of the time of the best reference implementation, which is Pari. For inputs less than 64-bits, our 64-bit implementation of our simplified left-to-right binary extended GCD is the fastest on average and takes on average roughly 77% of the time of the best reference implementation, which, again, is Pari. For inputs less than 119-bits, our 128-bit implementation of our simplified left-to-right binary extended GCD is the fastest on average and typically only takes roughly 95% of the time of either Pari or GMP. For inputs larger than 118-bits, our 128-bit implementation of Lehmer's extended GCD using our simplified left-to-right binary extended GCD for 64-bit words is the fastest on average, taking roughly 88% of the time of Pari or GMP on average.

All of the 32-bit implementations tested in this Chapter produce correct results for inputs up to 31-bits, with the exception of each of the variants of Stein's right-to-left binary extended GCD, which is only correct for inputs up to 29-bits. Similarly, our 64-bit implementations are correct for inputs up to 63-bits, but our implementations of Stein's right-to-left binary extended GCD are only correct for inputs no larger than 60-bits. Finally, our 128-bit implementations are correct for inputs up to 127-bits, with the exception of our 128-bit implementations of Stein's right-to-left binary extended GCD, which is correct for inputs no larger than 124 bits.

For each of our implementations (with the exception of our implementations of Stein's right-to-left binary extended GCD), overflows of intermediates do not occur and we did not have to use arithmetic of a larger word size. However, this is not the case for our implementations of Stein's right-to-left binary extended GCD. In this case, we could have used arithmetic of a larger word size, but this would have made the implementation slower than it currently is and we wanted each of our implementations to be as fast as possible for comparison. Our implementation of Stein's right-to-left binary extended GCD is not the fastest on average for any bit range, and so altering it to handle intermediate overflow did not seem useful.

This chapter discussed our implementations of the extended GCD and demonstrated a practical improvement on average for inputs less than 128-bits. We introduced a simplified left-to-right binary extended GCD that is based on Shallit and Sorrenson [46] method. Our 32-bit implementation of our simplified left-to-right binary extended GCD takes on average 78% of our 32-bit implementation of Shallit and Sorrenson's left-to-right binary extended GCD. Our 64-bit implementation takes on average 71% of the time, and our 128-bit implementation takes on average 45% of the time. As such our implementations of our simplified left-to-right binary extended GCD demonstrate a practical improvement on average over our implementations of Shallit and Sorenson's left-to-right binary extended GCD.

The next chapter studies the performance of our implementations of ideal class arithmetic. Since ideal class arithmetic makes heavy use of the extended GCD, the left-only extended GCD, and the partial extended GCD, we adapted each of our implementations of the full extended GCD to a left-only and partial extended GCD. We then use the results of this chapter to determine which implementation of the full, left-only, and partial extended GCDs to use within our implementations of ideal class arithmetic.

Chapter 6

Ideal Arithmetic Experiments

The previous chapter demonstrated results that lead to practical improvements in computing the extended GCD – the dominating operation of reduced ideal multiplication. This chapter discusses our implementations of ideal arithmetic, which make extension use of the results of the previous chapter. In particular, when computing either the full, left-only, or partial extended GCD, if arguments are less than 32-bits, we use our 32-bit implementation of the extended Euclidean algorithm. If arguments are larger than 31-bits, but smaller than 64-bits, we use our 64-bit implementation of the simplified left-to-right binary extended GCD. When arguments are larger than 63-bits and smaller than 119-bits, we use our 128-bit implementation of the simplified left-to-right binary extended GCD, and when they are larger than 118-bits, we use our 128-bit implementation of Lehmer's extended GCD using our 64-bit implementation of our simplified left-to-right binary extended GCD for the inner extended GCD. Finally, when arguments are larger than 127-bits, we use the implementation of the extended GCD and left-only extended GCD provided by GMP, and for the partial extended GCD we use an implementation of Lehmer's extended GCD using a 64-bit extended Euclidean algorithm for GMP types.

To further improve the performance of ideal arithmetic, specialized implementations using at most a single machine word, i.e. 64-bits, or at most two machine words, i.e. 128-bits were developed. A reference implementation using GMP (or MPIR), that works for unbounded discriminant sizes, was developed for comparison. Our 64-bit implementation of ideal arithmetic is accurate for negative discriminants less than 60-bits in size, while our 128-bit implementation of ideal arithmetic is accurate for discriminants less than 119-bits. This holds for all class group operations such as ideal class multiplication, squaring, and cubing.

As a general rule, in our 64-bit implementation, if intermediate terms fit within a 32-bit word, we use 32-bit arithmetic. If terms do not fit within a 32-bit word, but within a 64-bit word, we use 64-bit arithmetic. Only when terms do not fit within a 64-bit word do we use two machine words and 128-bit arithmetic. Similarly, for our 128-bit implementation, if terms fit within a single 64-bit word, then we use 64-bit arithmetic. Only when terms do not fit within a single 64-bit word do we use two 64-bit words and perform 128-bit arithmetic. In the extreme case that an intermediate term does not fit within two 64-bit words, i.e. the term is larger than 128-bits, only then do we fall back on GMP for multiprecision arithmetic.

The software developed for this chapter uses the GNU C compiler version 4.7.2, and assembly language for 128-bit arithmetic, and was developed on Ubuntu 12.10 using a 2.7GHz Intel Core i7-2620M CPU with 8Gb of memory. Experiments are restricted to using only a single core of the CPU's four cores.

6.1 Specialized Implementations of Ideal Arithmetic

Throughout this thesis, an ideal class $[\mathfrak{a}]$ is represented using the $\mathbb{Z}$-module for the reduced ideal representative $\mathfrak{a} = [a, (b + \sqrt{\Delta})/2]$. Ideal class multiplication and ideal reduction use the value $c = (b^2 - \Delta)/4a$, which is also useful in determining if an ideal is an ambiguous ideal (see Subsection 4.1.1 for additional details). For this reason, our implementation represents an ideal class using the triple (a, b, c). We represent the ideal class group Cl_Δ using the discriminant Δ, and maintain this separately from the representation of each ideal class. We do this to reduce the memory footprint of the representation of each ideal class.

Our implementations ideal class arithmetic use all the improvements discussed in Subsection 2.5.5 (see [29, Algorithm 6]). This includes separating the computation of

$$s = \gcd(a_1, a_2, (b_1 + b_2)/2) = Ya_1 + Va_2 + W(b_1 + b_2)/2$$

into two GCD computations, the second of which is only computed if the first GCD is not equal to 1. Additional optimizations discussed in this section are an approximation of the termination bound, minimizing the size of intermediates, branch free reduction operations, and the computation of prime ideal classes.

In Subsections 2.5.5, 2.5.6, and 2.5.7 we gave algorithms for fast ideal class multiplication, squaring, and cubing. These compute a termination bound for the partial extended GCD useful for computing partially reduced coefficients of the product ideal class. The termination bound for ideal multiplication is $\sqrt{a_1/a_2}|\Delta/4|^{1/4}$, for ideal squaring it is $|\Delta/4|^{1/4}$, and for ideal cubing it is $\sqrt{a_1}|\Delta/4|^{1/4}$. In our implementations of ideal class arithmetic, we approximate each of these termination bounds, however, future work is necessary to demonstrate that this approach actually leads to a performance improvement.

Each of the termination bounds is approximated as follows. In each case, the termination bound contains the coefficient $|\Delta/4|^{1/4}$. Since the coefficients of the partial extended GCD are integers, once the discriminant Δ of the class group Cl_Δ is known, we store both $\lceil|\Delta/4|^{1/4}\rceil$ and $\lceil|\Delta/4|^{1/2}\rceil$ along side Δ in our representation of the class group. These values are computed as accurately as possible, since these are only computed once per class group. The termination bound for ideal class multiplication is approximated as

$$\sqrt{a_1/a_2}|\Delta/4|^{1/4} = \sqrt{(a_1/a_2)|\Delta/4|^{1/2}} \approx \sqrt{(a_1/a_2)\left\lceil|\Delta/4|^{1/2}\right\rceil}.$$

For ideal squaring we simply use $\lceil|\Delta/4|^{1/4}\rceil$, and for ideal cubing we use $\sqrt{a_1\left\lceil|\Delta/4|^{1/2}\right\rceil} \approx \sqrt{a_1}|\Delta/4|^{1/4}$.

We further approximate the computation of integer square roots when computing the termination bound for fast multiplication. We note that

$$\begin{array}{rcccc} & x^{1/2} & = & 2^{(\log_2 x)/2} & \approx & 2^{\lfloor\lfloor\log_2 x+1\rfloor/2\rfloor} \\ \Rightarrow & x/x^{1/2} & = & x/2^{(\log_2 x)/2} & \approx & x/2^{\lfloor\lfloor\log_2 x+1\rfloor/2\rfloor}, \end{array}$$

where $\lfloor\log_2 x+1\rfloor$ is the number of bits in the binary representation of x. Using this approximation, we compute the integer square root of x as roughly $\left\lfloor x/2^{\lfloor\lfloor\log_2 x+1\rfloor/2\rfloor}\right\rfloor$. This is simply a bit shift right by half the number of bits in x.

Other practical considerations are to minimize the size of variables and intermediate values, as smaller operands often lead to faster arithmetic. Recall from Subsection 2.5.2 that $|b| \leq a \leq \sqrt{\Delta/3}$. As such, the variables a and b only require half the memory needed to store Δ.

In our 64-bit implementation, Δ and c take 64-bits of storage, while both a and b require only 32-bits at most. As such, we take advantage of 32-bit operations when possible (such as our 32-bit implementation of the extended Euclidean GCD from Section 5.1). Similarly in our 128-bit implementation, Δ and c fit within 128-bits of storage, and both a and b fit within one 64-bit machine word. Again, we use 64-bit arithmetic when possible (such as our simplified 64-bit left-to-right binary extended GCD from Section 5.3), and even 32-bit when operands are small. However, since intermediates may be larger, we sometimes require the full 64-bits or 128-bits of the implementation. Cubing sometimes requires even larger intermediates, for example, when computing modulo ${a_1}^2$. Since ${a_1}^2$ can be as large as Δ, intermediates before modulo might be larger than Δ in size. As such, our 64-bit implementation of cubing requires some 128-bit arithmetic, while our 128-bit implementation uses GMP when necessary.

To keep intermediates small, we compute using the smallest absolute representative of a residue class. This has the added benefit that it reduces the likelihood of overflow and the necessity of multiple precision arithmetic. For example, Equations 2.7 states that we can compute $U \pmod{a_1/s}$ and Equation 2.9 allows us to compute $b \pmod{2a}$. When an intermediate is known to be close to zero, we do not perform a complete division with remainder, but only add or subtract the divisor as appropriate (often using the signed mask discussed in Section 5.6).

In the specific case of ideal reduction (see Algorithm 2.1), one step is to compute b' such that $-a < b' \leq a$ and $b' \equiv b \pmod{2a}$. To compute $b \bmod 2a$ we effectively compute $b = q2a + r$ for $q, r \in \mathbb{Z}$ where $|r| < 2a$. We note that $q = \lfloor b/(2a) \rfloor = \lfloor (b/a)/2 \rfloor$, and instead compute $b = q'a + r'$ for $q', r' \in \mathbb{Z}$ where $|r'| < a$. Now q' is at least twice q. If q' is even, then $b = q'a + r' = (q'/2)2a + r'$ and we let $b' = r'$. Suppose q' is odd and r' is positive. Then $b = q'a + r' = (q'+1)a + (r' - a) = ((q'+1)/2)2a + (r' - a)$. Since $0 < r' < a$, we have $b' = r' - a$ and $-a < b' \leq a$. The proof is similar when q is odd and r' is negative.

This leads us to the following implementation, which we optimize to be free of conditionals. Using two's complement for fixed sized words, we compute

$$\begin{aligned}
b &= q'a + r' && \{\text{division with remainder}\} \\
q_m &= -(q \texttt{ and}_2\ 1) && \{q_m \text{ has all bits set when } q \text{ is odd}\} \\
r_m &= \neg\texttt{sign\_mask}(r) && \{r_m \text{ has all bits set when } r \geq 0\} \\
a' &= (a \oplus r_m) - r_m && \{\text{negate } a \text{ when } r_m \text{ is all set}\} \\
r &= r' + (a' \texttt{ and}_2\ q_m) && \{\text{move } r \text{ towards zero when } q \text{ is odd}\} \\
d &= r_m \texttt{ or}_2\ 1 && 1 \text{ when } r < 0 \text{ and } -1 \text{ otherwise} \\
q &= (q' - (d \texttt{ and}_2\ q_m))/2 && \{\text{adjust } q \text{ with respect to } r\}
\end{aligned}$$

and finally $b' = r$.

Since we represent an ideal class using the triple (a, b, c), it is necessary to compute a new value $c' = (b'^2 - \Delta)/4a$ after a reduction step. This can be simplified using Tenner's algorithm (see [34, §3.4]) as

$$\begin{aligned}
c &= (b^2 - \Delta)/4a \\
&= ((2aq + r)^2 - \Delta)/4a \\
&= (4a^2q^2 + 4aqr + r^2 - \Delta)/4a \\
&= (r^2 - \Delta)/4a + aq^2 + qr \\
&= c' + q(aq + r) \\
&= c' + q(2aq + r - aq) \\
&= c' + q(b - aq) \\
&= c' + q(b + r)/2.
\end{aligned}$$

As such, we have

$$c' = c - q(b + r)/2.$$

Note, the last step above is obtained by rewriting $b = 2aq + r$ as

$$\begin{aligned} b - 2aq &= r \\ 2b - 2aq &= b + r \\ b - aq &= (b + r)/2. \end{aligned}$$

Since $b - aq \in \mathbb{Z}$, the divide by 2 can be performed with an arithmetic bit shift right.

The final performance improvement is for generating prime ideals. For a given prime p, we would like to find an ideal of the form $[p, (b + \sqrt{\Delta})/2]$. Since $b^2 - 4pc = \Delta$, and p is known, this gives $b^2 \equiv \Delta \pmod p$ and $b \equiv \pm\sqrt{\Delta} \pmod p$. Computing b is a matter of computing a square root modulo a prime, if one exists. There are efficient algorithms for this (see Bach and Shallit [9]), but since we are interested in finding some prime ideal, rather than a specific prime ideal, we instead use a table of all square roots modulo each prime p for sufficiently many small p. Note, the size of the table is $O(n^2/\log n)$ for n primes. For our purposes, all primes $p < 1000$ is enough. Finding $b \equiv \pm\sqrt{\Delta} \pmod p$ is now a matter of looking up $\Delta \bmod p$ from a table.

Having found a value for b, we have not necessarily found a prime ideal $[p, (b + \sqrt{\Delta})/2]$. Since $\Delta = b^2 - 4pc$, this implies that $c = (b^2 - \Delta)/4p$ must have an integral solution. Our implementation maintains the value c for an ideal class, so we compute c and if c is an integer, than we have found a prime ideal. Otherwise, we try again for some other prime (usually the smallest prime greater than p). Note that the table of square roots modulo a prime is part of our library and available as global data. Also, an optimization overlooked in our implementation is to compute the square roots modulo $4p$ for p prime. This way, if a square root exists, c is guaranteed to be integral.

6.2 Average time for operations

This section compares the performance of our implementations of ideal class arithmetic with our reference implementation, Pari 2.5.3 [7]. Let k be the target number of bits in the discriminant Δ. We iterate k from 16 to 140, and for each k we repeat the following 10,000 times: We compute two prime numbers p and q such that p is $\lfloor k/2 \rfloor$ bits and q is $k - \lfloor k/2 \rfloor$ bits. We use the negative of their product $\Delta = -pq$ as the discriminant for a class group, unless $\Delta \not\equiv 1 \pmod 4$. In which case, we try again with a different p and q. We then pick a prime ideal class, $[\mathfrak{a}]$, randomly from among the primes less than 1000. One motivation for limiting this to primes less than 1000 is the size of the lookup table for $\sqrt{\Delta \bmod p} \pmod p$.

For ideal multiplication, squaring, and cubing, we use Algorithms 2.2, 2.3, and 2.4 respectively. For ideal reduction we use Algorithm 2.1. Each of the optimizations mentioned in this chapter are applied to each implementation, as well as the results from Chapter 5 for computing the extended GCD. The general strategy for our 64-bit implementation is to use 32-bit arithmetic when operands are less than 32-bits, otherwise to use 64-bit arithmetic, and only when this fails do we use 128-bit arithmetic. For our 128-bit implementation, we use 64-bit arithmetic when operands are less than 64-bits, otherwise we use 128-bit arithmetic, and only when intermediates are larger than 128-bits do we use GMP arithmetic.

To time the performance of squaring and cubing, we iterate 1000 times either $[\mathfrak{a}] \leftarrow [\mathfrak{a}]^2$ or $[\mathfrak{a}] \leftarrow [\mathfrak{a}]^3$ respectively. Dividing the total cost by 10,000,000 gives the average time to square or cube an ideal. For multiplication, we initially set $[\mathfrak{b}] \leftarrow [\mathfrak{a}]$. We then iterate 1000 times the sequence of operations $[\mathfrak{c}] \leftarrow [\mathfrak{a}] \cdot [\mathfrak{b}]$, $[\mathfrak{a}] \leftarrow [\mathfrak{b}]$, and $[\mathfrak{b}] \leftarrow [\mathfrak{c}]$.

To time the performance of Pari, we statically link with the Pari library. We determine the discriminant and prime ideal class in exactly the same way as with the other implementations. We then convert the ideal class into a Pari `GEN` object using `qfi`, Pari's function to create a binary quadratic form. The only operations timed are calls to `gmul`, in the case of multiplication, `gsqr`, in the case of squaring, and `gpowsg`, for cubing. Pari does not appear to implement ideal class cubing and so we use the general powering function, `gpowsg`. We perform garbage collection in Pari by recording Pari's stack pointer, `avma`, and then restoring it after all the timings are completed.

Figures 6.1, 6.2, and 6.3 show the average time for ideal multiplication, squaring, and cubing respectively, alongside the timings for each operation using the reference implementation from Pari. For all discriminant sizes, our 64-bit implementation performs faster than our 128-bit implementation, which performs faster than our GMP implementation. Table 6.1 shows the relative costs on average for ideal multiplication, squaring, and cubing for each implementation compared with Pari. For our 64-bit implementation, we only average the times for discriminants less than 60-bits in size, for our 128-bit implementation, we use the times for discriminants less than 119-bits in size, and for our GMP implementation, we use the times for discriminants less than 140-bits in size.

In Figures 6.4, 6.5, and 6.6, we compare the average time to cube against the average time to multiply a representative with its square. One thing to notice is that our 128-bit implementation of cubing performs more poorly than that of multiplying an ideal with its square when the discriminant is larger than 69-bits. This is possibly because when a_1 is large, computing modulo ${a_1}^2$ means the intermediate before reduction might be larger than Δ and as such larger than 128-bits. In this case, we rely on GMP for multiple precision arithmetic. Also, when computing the coefficients

$$\begin{aligned} M_1 &\leftarrow (R_i a_1 + C_i U a_1)/{a_1}^2 \\ M_2 &\leftarrow (R_i(b_1 + U a_1) - s C_i c_1)/{a_1}^2 \end{aligned}$$

in Algorithm 2.4, the intermediate before dividing by ${a_1}^2$ may be larger than Δ in size, and as such will require the use of multiple precision arithmetic. On the other hand, none of the intermediates in our 128-bit implementation of ideal multiplication or squaring are ever larger than Δ in size. Contrast this to our 64-bit implementation that does not use GMP at all, and our GMP implementation that only uses GMP for arithmetic – in both of these implementations, cubing is faster than multiplying an ideal with its square.

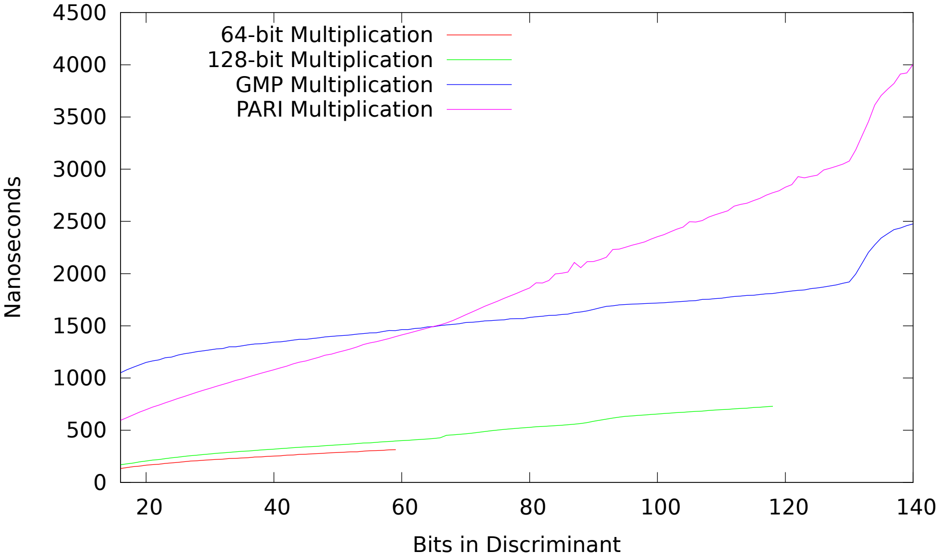

Figure 6.1: Average Time to Multiply Reduced Ideal Class Representatives.

	64-bit Implementation / Pari
Multiplication	0.23405
Squaring	0.26703
Cubing	0.17120

	128-bit Implementation / Pari
Multiplication	0.28567
Squaring	0.26703
Cubing	0.23718

	GMP Implementation / Pari
Multiplication	0.96476
Squaring	0.84217
Cubing	0.55412

Table 6.1: Relative cost of each implementation on average compared with Pari.

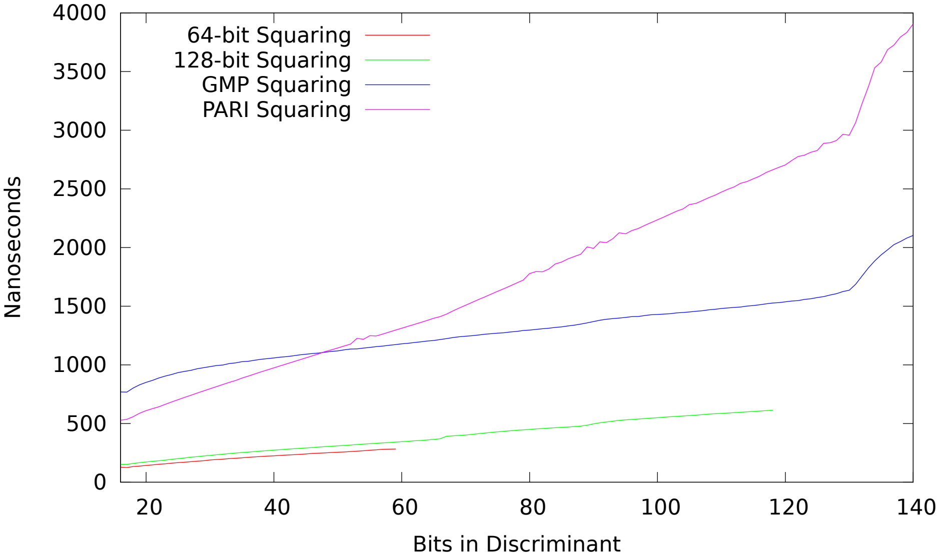

Figure 6.2: Average Time to Square Reduced Ideal Class Representatives.

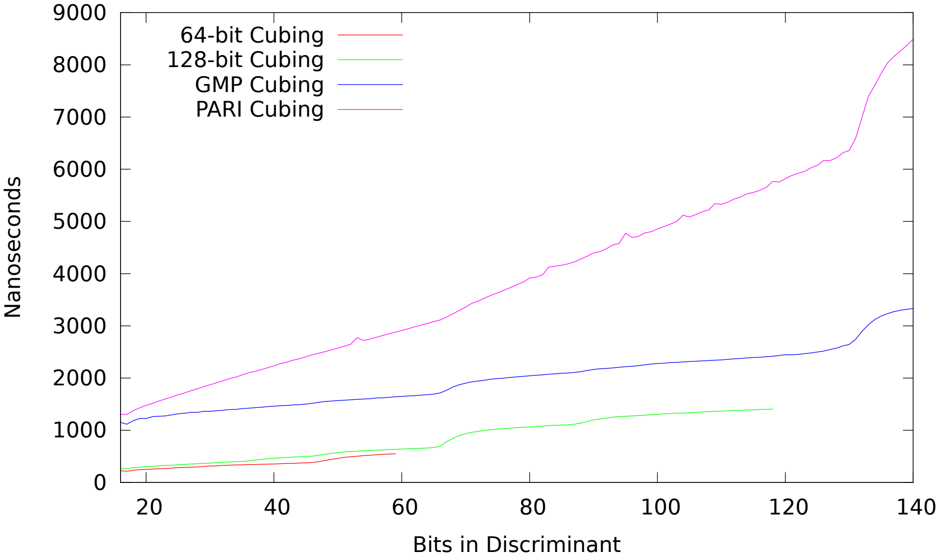

Figure 6.3: Average Time to Cube Reduced Ideal Class Representatives.

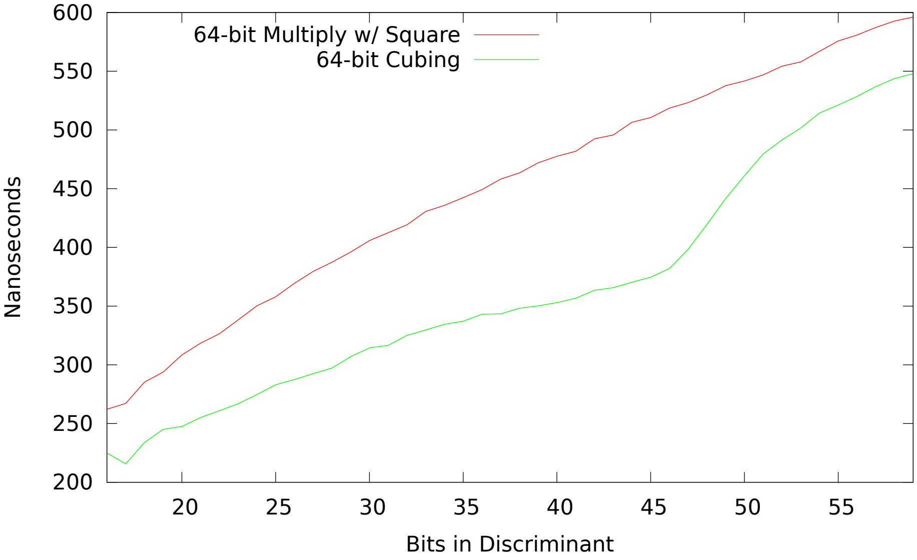

Figure 6.4: Time to compute $[\mathfrak{a}^3]$ in our 64-bit implementation.

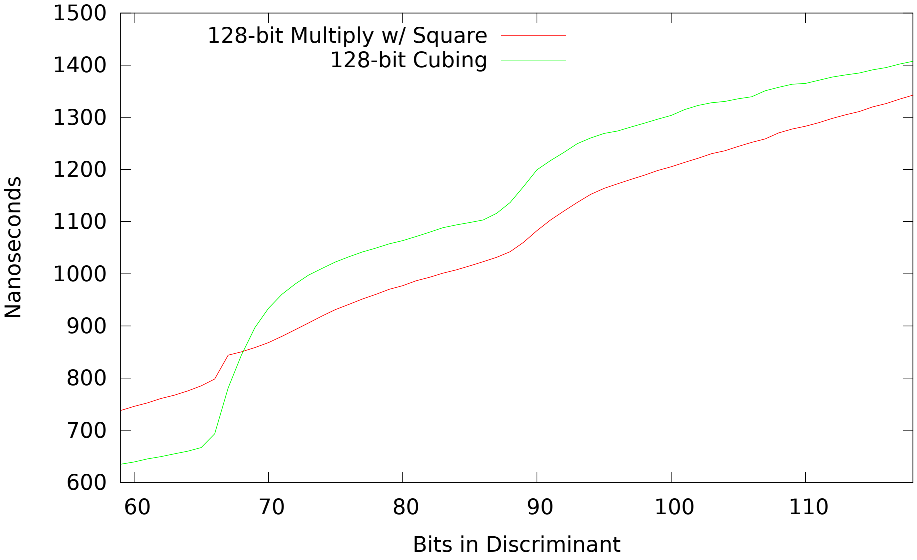

Figure 6.5: Time to compute $[\mathfrak{a}^3]$ in our 128-bit implementation.

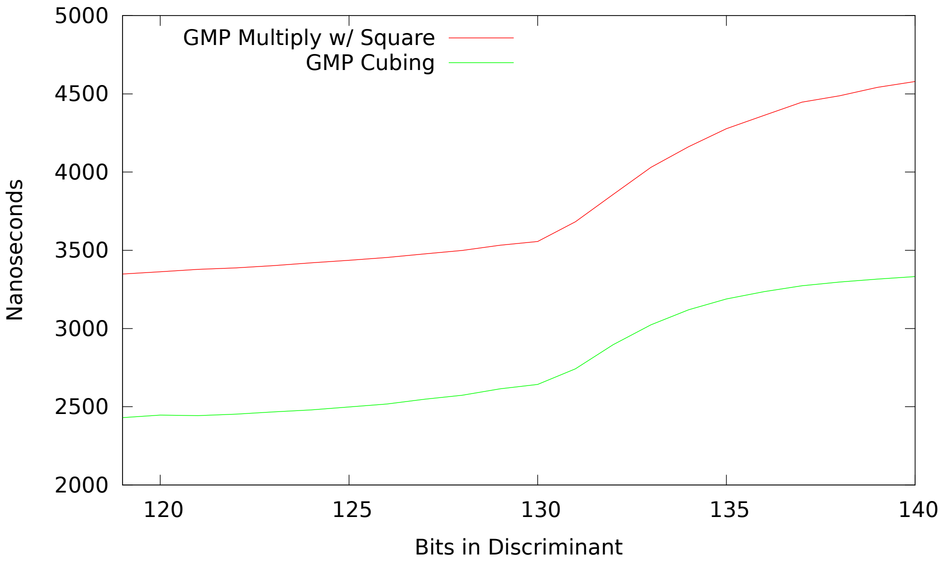

Figure 6.6: Time to compute $[\mathfrak{a}^3]$ in our GMP implementation.

This chapter discussed our implementations of ideal class arithmetic. We then demonstrate the average time for multiplication, squaring, and cubing. We show that for discriminants smaller than 60-bits, our 64-bit implementation of ideal class arithmetic is significantly faster than the reference implementation provided by Pari, then for discriminants larger than 59-bits, but smaller than 119-bits, our 128-bit implementation of ideal class arithmetic is significantly faster than the reference implementation provided by Pari, and finally, for discriminants larger than 118-bits, our GMP implementation is still faster on average than the reference implementation provided by Pari.

Chapter 7

Exponentiation Experiments

Subection 4.2.1 discussed the bounded primorial steps algorithm – an algorithm that finds the order of an element by exponentiating it to the product of many primes. Since this exponent is determined by a given bound, the algorithm benefits from precomputation of representations that lead to efficient exponentiation in practice. Section 4.2 discussed the SuperSPAR integer factoring algorithm, for which a bound on the group size and the order of an element lead to the factorization of an integer. Representations that lead to efficient exponentiation in practice improve the performance of the SuperSPAR factoring algorithm, as well any algorithm based on the bounded primorial steps algorithm, or one that uses exponentiation for fixed exponents.

This chapter discusses several approaches to exponentiating an ideal class representative to a large exponent, and in particular, large exponents that are the product of all primes less than some bound[1] and are known in advance. We begin by recalling exponentiation techniques from Chapter 3. These include techniques that use signed and unsigned base 2 representations, greedy right-to-left and left-to-right double base representations (for bases 2 and 3), and a tree-based approach. We also consider several extensions to these. For small exponents (16-bits or less), we are able to exhaustively compute all representations under certain constraints. This allows us to use larger exponents by partitioning the large exponent into 16-bit blocks or by using its factorization. Finally, we compare each method on a sequence of primorials to find the best method in practice.

In the previous chapter, we discussed our implementations of ideal class arithmetic using 64-bit and 128-bit operations. For our 64-bit implementation, the largest discriminants supported are those that fit within 59-bits, and for our 128-bit implementation, the largest

[1] Such products are called *primorials* and are discussed in Section **??**.

discriminants fit within 118-bits. These represent an upper bound on the average cost of arithmetic operations within each implementation. As such, our exponentiation experiments focus on improving the time to exponentiate assuming the average cost of operations for 59-bit and 118-bit discriminants.

7.1 Binary and Right-to-Left Non-Adjacent Form

In Sections 3.1 and 3.2 we discussed binary exponentiation and non-adjacent form exponentiation in detail. Briefly again, binary exponentiation uses the binary representation of an exponent n. The representation can be parsed from high-order to low-order or from low-order to high-order – we typically refer to this difference as left-to-right or right-to-left respectively. Using either approach, we use $\lfloor \log_2 n \rfloor$ squares and $\lfloor \log_2 n \rfloor / 2$ multiplications on average.

A non-adjacent form exponentiation uses a signed base 2 representation of the exponent such that no two non-zero terms are adjacent[2]. Similarly, when computing a non-adjacent form we can parse the exponent from left-to-right or from right-to-left. Either direction, we use $\lfloor \log_2 n \rfloor + 1$ squares and $(\lfloor \log_2 n \rfloor + 1)/3$ multiplications on average.

7.2 Right-to-Left 2,3 Chains

In Subsection 3.4.1, we described a method for computing 2,3 chains from low-order to high-order that is a natural extension of the binary representation or non-adjacent form of an exponent. In this method, we reduce the exponent n by 2 while it is even, and by 3 while it is divisible by 3. At which point either $n \equiv 1 \pmod 6$ or $n \equiv 5 \pmod 6$ and we add or subtract 1 so that n is a multiple of 6. The resulting partition of n is then reversed such that the number of squares or cubes in successive terms is monotonically increasing and we

[2] Non-adjacent form is written as $n = \prod s_i 2^i$ for $s_i \in \{0, -1, 1\}$. By "non-adjacent" we mean that $s_i \cdot s_{i+1} = 0$ for all i.

can use Algorithm 3.2 from Chapter 3 to compute the exponentiation.

Since this approach evaluates the exponent modulo 3 and may reduce the exponent by 3, efficient methods to compute $n \bmod 3$ and $n/3$ will speed the computation of the chain. Notice that $4 \equiv 1 \pmod 3$. As such, we can write n as

$$n = 4\lfloor n/4 \rfloor + (n \bmod 4) \equiv \lfloor n/4 \rfloor + (n \bmod 4) \pmod 3$$

and then recursively write $\lfloor n/4 \rfloor \pmod 3$ the same way. This provides us with a method to quickly compute $n \bmod 3$ – we partition the binary representation of n into 2-bit blocks and sum each block modulo 4. The resulting sum s is $s \equiv n \pmod 3$. We further improve the performance of this algorithm by noting that $4^m \equiv 1 \pmod 3$. Since our target architecture (and language) has 64-bit words, we partition the binary representation into 64-bit blocks and sum each block modulo 2^{64}. We then add each 32-bit word of the resulting sum modulo 2^{32}, then each 16-bit word of the sum modulo 2^{16}, and finally each 8-bit word modulo 2^8. We look up the final answer modulo 3 from a table of 256 entries (see Algorithm 7.1).

Algorithm 7.1 Fast $n \bmod 3$ (adapted from Hacker's Delight [52]).

```
Input: $n \in \mathbb{Z}$.
Output: $n \bmod 3$.
s ← 0
for $i$ from 0 to $\lfloor \log_2 n \rfloor$ by 64 do
    $t \leftarrow \lfloor n/2^i \rfloor \bmod 2^{64}$
    $s \leftarrow (s + t) \bmod 2^{64}$
$s \leftarrow (s + \lfloor s/2^{32} \rfloor) \bmod 2^{32}$
$s \leftarrow (s + \lfloor s/2^{16} \rfloor) \bmod 2^{16}$
$s \leftarrow (s + \lfloor s/2^{8} \rfloor) \bmod 2^{8}$
return $s \bmod 3$                    {Lookup from a table with 256 entries.}
```

Division by 3 is relatively expensive when compared to computing the remainder. A common approach is to precompute a single word approximation of the inverse of 3, and then to use multiplication by an approximation of the inverse and then to adjust the result (see [27, 52, 41] for additional information). As the GNU Multiple Precision (GMP) library implements exact division by 3, we use this.

7.3 Windowed Right-to-Left Chains

Windowing improves the performance of right-to-left binary exponentiation and right-to-left binary GCD computations. Similarly, we use windowing to improve the performance of a right-to-left 2,3 chain. Specifically, we experimented with a window size of $2^2 3^2$. Again, while the exponent is even, we reduce it by 2, and while it is divisible by 3, we reduce it by 3. In a non-windowed variant, we add or subtract 1 to make the remainder a multiple of 6. In the windowed variant, we evaluate the exponent n modulo the window $2^2 3^2 = 36$, which gives us $n \equiv 1, 5, 7, 11, 13, 17, 19, 23, 25, 29, 31, 35 \pmod{36}$. As there are 12 different residue classes, and for each residue class we could either add or subtract 1 from n, we have 2^{12} different strategies to evaluate. For each strategy, we measured the cost to exponentiate the primorials P_{100k} for $1 \le k \le 50$. For 48 out of 50 of the primorials tested, the same strategy lead to the cheapest cost of exponentiation[3], which was to subtract 1 from n when $n \equiv 1, 5, 13, 17, 25, 29 \pmod{36}$ and to add 1 otherwise.

Since the windowed variant of a 2,3 chain computes $n \bmod 36$, we give a fast method for this. First write $n = 4x + r_4$ for integers x and r_4 such that $0 \le r_4 < 4$. Then write $x = 9y + r_9$ for integers y and r_9 such that $0 \le r_9 < 9$. Substituting the second equation into the first we have

$$\begin{aligned} n &= 4(9y + r_9) + r_4 \\ &= 36y + 4r_9 + r_4. \end{aligned}$$

Notice that $(4r_9 + r_4)$ is $n \bmod 36$ where $r_4 = n \bmod 4$ and $r_9 = \lfloor n/4 \rfloor \bmod 9$. To compute $n \bmod 9$, we point out that $64 \equiv 1 \pmod{9}$. Similar to the case of computing $n \bmod 3$, we write

$$n = 64 \lfloor n/64 \rfloor + (n \bmod 64) \equiv \lfloor n/64 \rfloor + (n \bmod 64) \pmod{9}$$

and recursively write $\lfloor n/64 \rfloor \pmod{9}$. To compute $n \bmod 9$, we partition n into blocks of

[3] For the primorial P_{200}, this strategy was the third most efficient, and for P_{500} it was the second most efficient.

6-bits (since $2^6 = 64$) and sum each 6-bit block modulo 64. We then compute the sum $s \bmod 9$. Again, since our target architecture has 64-bit machine words, we improve upon this by first partitioning n into 192-bit blocks[4] and compute an intermediate sum of each 192-bit block. We then partition the intermediate sum into 6-bit blocks and compute the final solution.

7.4 Left-to-Right 2,3 Representations and Chains

In Subsection 3.4.2 we described a greedy algorithm for computing 2,3 representations from high-order to low-order. Briefly, for a given exponent n and for $a_i, b_i \in \mathbb{Z}_{\geq 0}$ and $s_i \in \{-1, 1\}$, we compute the term $s_i 2^{a_i} 3^{b_i}$ such that $\left| n - s_i 2^{a_i} 3^{b_i} \right|$ is as small as possible. We then recurse on the result $n - s_i 2^{a_i} 3^{b_i}$. As is, this produces unchained representations, but a chain can be generated by restricting each a_i and b_i such that it is no greater than the a_{i-1} and b_{i-1} from the previous term. Placing a bound on the number of squares and cubes can also be done globally, i.e. for every a_i, b_i we have $0 \leq a_i \leq A$ and $0 \leq b_i \leq B$. When a bound is applied globally, the cost of a left-to-right chain or representation varies as the global bound varies.

Figure 7.1 shows the time to exponentiate using a precomputed left-to-right representation for an exponent that is the product of the first 1000 primes (a number with 11271 bits). The horizontal axis indicates a global bound A such that all $a_i \leq A$. A global bound B is chosen for the exponent n such that $B = \left\lceil \log_3(n/2^A) \right\rceil$. The end points of the graph correspond to a representation that uses only cubing and multiplication (on the left side) and a representation that uses only squaring and multiplication (on the right side).

In general, bounding the maximum number of squares and cubes give representations that lead to faster exponentiation in the ideal class group than representations where the number of squares or cubes is left unbound. For this particular exponent, a representation generated without a global bound takes approximately 7.3 milliseconds for a 59-bit discriminant and

[4] We use 192-bit blocks since a 64-bit machine word evenly partitions a 192-bit block and $2^{192} \equiv 1 \pmod 9$.

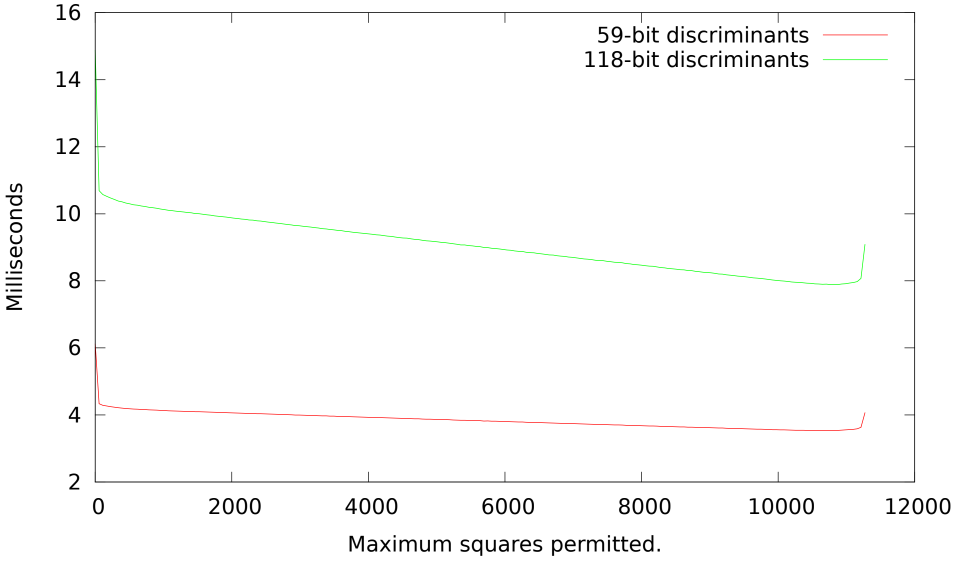

Figure 7.1: The time to exponentiate the 1000th primorial, n, while varying the maximum number of squares permitted, A. The maximum number of cubes is $B = \lceil \log_3(n/2^A) \rceil$.

17.2 milliseconds for a 118-bit discriminant. This is slower than all bounded representations for both implementations.

7.5 Greedy Pruned Trees

A right-to-left 2,3 chain can be generated by repeatedly reducing the exponent by 2 and 3 and then either adding or subtracting 1 and repeating this process until the exponent is 1. In the windowed variant, after reducing the exponent by 2 and 3, we had 12 residues modulo $2^2 3^2$ and this lead to 2^{12} different strategies for adding or subtracting 1 from the residue.

In Subsection 3.4.3, we discussed a tree based method where a set of at most L *partial representations* are maintained (a partial representation can be written as $n = x + \sum s_i 2^{a_i} 3^{b_i}$). At each iteration, the x term from each partial representation generates two new elements $x - 1$ and $x + 1$. After reducing each new element by powers of 2 and 3, only the L smallest are kept. More formally, an element x generates new integral elements $(x \pm 1)/(2^c 3^d)$.

Here we consider two variations to the approach of maintaining the L best[5] partial rep-

[5] We say that a partial representation is *better* when the x term is smaller, or if the x terms are equal,

resentations. The first variation we consider is to generate several new nodes with integer values of the form $(x \pm 2^a 3^b)/(2^c 3^d)$, while the second variation is to include two terms such that each node generates values of the form $(x \pm 2^a \pm 3^b)/(2^c 3^d)$. Notice that the first variation includes the forms $(x \pm 1)/(2^c 3^d)$ and so tends to give representations no worse than the method suggested by Doche and Habsieger [24] (covered in Subsection 3.4.3). The assumption is that by adjusting x by more than just 1, we increase the likelihood of finding a multiple of a large $2^c 3^d$ window.

For both variations, we bound a and b such that $0 \leq a \leq U$ and $0 \leq b \leq U$. Figure 7.2 shows the average time to exponentiate by the 1000^{th} primorial as the bound U increases (using the $k = 4$ best partial representations). As U increases, computing representations using this approach quickly becomes intractable. We found that for our purposes, $U = 16$ is an acceptable trade off between the average time to exponentiate and the time to generate the 2,3 representation.

7.6 The L Best Approximations

The approach of maintaining a set of the L best partial representations of an exponent can be adapted to that of the left-to-right 2,3 representation from Subsections 3.4.2 and 7.4. For an integer x we say that $2^a 3^b$ is a best approximation of $|x|$ when $\left||x| - 2^a 3^b\right|$ is minimal. The algorithm for the left-to-right representation (Subsection 3.4.2) finds a best approximation for x and then repeats on the positive difference. However, instead of only iterating on the best approximation, each value from the set of partial representations generates new values of the form $\left||x| - 2^a 3^b\right|$ (being careful to record the sign of x), and the L best partial representations are retained. In this case, we iterate b from $0 \leq b \leq B$ and let $a_1 = \left\lfloor \log_2(x/3^b) \right\rfloor$ and $a_2 = \left\lceil \log_2(x/3^b) \right\rceil$ bounding both $a_1, a_2 \leq A$. Note that using $a_2 = a_1 + 1$ is sufficient since either $\left\lceil \log_2(x/3^b) \right\rceil = \left\lfloor \log_2(x/3^b) \right\rfloor$ or $\left\lceil \log_2(x/3^b) \right\rceil = \left\lfloor \log_2(x/3^b) \right\rfloor + 1$. We then use

when the cost of the 2,3 summation is less.

$\left||x| - 2^{a_1}3^b\right|$ and $\left||x| - 2^{a_2}3^b\right|$ as candidates for the new set. As before, iterating the bound on the number of squares and cubes varies the cost of the representations generated.

7.7 Additive 2,3 Chains

Imbert and Philippe [30] give a method (see Subsection 3.4.4) to compute the shortest additive strictly chained 2,3 partition, suitable for smaller integers. Such partitions do not permit subtraction and every term is strictly less than and divides all subsequent terms. They take the form

$$n = \sum_{i=0}^{k} 2^{a_i}3^{b_i}$$

for $a_i, b_i \in \mathbb{Z}_{\geq 0}$ where $2^{a_i}3^{b_i}$ divides $2^{a_j}3^{b_j}$ and $a_i < a_j$ and $b_i < b_j$ for all $i < j$.

For our purposes, we modified this algorithm to compute additive 2,3 chains that minimize the average cost of arithmetic in the ideal class group – the resulting additive 2,3 chains give a better average time for exponentiation, while they are not necessarily the shortest possible.

Let M, S, and C be the average cost to multiply, square, and cube respectively. Our modified function is as follows

$$s'(n) = \begin{cases} \min\{S + s'(n/2), C + s'(n/3)\} & \text{when } n \equiv 0 \pmod 6 \\ M + s'(n-1) & \text{when } n \equiv 1 \pmod 6 \\ S + s'(n/2) & \text{when } n \equiv 2 \pmod 6 \\ \min\{C + s'(n/3), M + S + s'((n-1)/2)\} & \text{when } n \equiv 3 \pmod 6 \\ \min\{S + s'(n/2), M + C + s'((n-1)/3)\} & \text{when } n \equiv 4 \pmod 6 \\ M + S + s'((n-1)/2) & \text{when } n \equiv 5 \pmod 6 \end{cases}$$

where $s'(1) = 0$, $s'(2) = S$, and $s'(3) = C$ are the base cases.

We also experimented with computing 2,3 strictly chained partitions that allow for both positive and negative terms. The corresponding function we used is

$$f(n) = \begin{cases} \min\{S + f(n/2), C + f(n/3)\} & \text{when } n \equiv 0 \pmod 6 \\ \min\{M + f(n-1), M + S + f((n+1)/2)\} & \text{when } n \equiv 1 \pmod 6 \\ \min\{S + f(n/2), M + C + f((n+1)/3)\} & \text{when } n \equiv 2 \pmod 6 \\ \begin{array}{l} \min\{C + f(n/3), M + S + f((n-1)/2), \\ \qquad\qquad M + S + f((n+1)/2)\} \end{array} & \text{when } n \equiv 3 \pmod 6 \\ \min\{S + f(n/2), M + C + f((n-1)/3)\} & \text{when } n \equiv 4 \pmod 6 \\ \min\{M + f(n+1), M + S + f((n-1)/2)\} & \text{when } n \equiv 5 \pmod 6 \end{cases}$$

again, where $f(1) = 0$, $f(2) = S$, and $f(3) = C$ are the base cases. One thing to notice is that the function s' computes a subset of the representations computed by the function f. As such, we expect the average cost to exponentiate using a representation computed by f to be no worse than the average cost to exponentiate using a representation computed by s.

7.8 Incremental Searching

For small inputs, the number of additive 2,3 chains is sufficiently reduced that we can compute all such chains in order to find the fastest. Here we consider a different approach to searching for representations – for a function generating a set of representations, we record the cheapest representation for each integer represented by the set.

We first iterate over all single term representations $2^{a_1}3^{b_1}$ for $0 \leq a_1 \leq A$ and $0 \leq b_1 \leq B$, and then all two term representations $2^{a_1}3^{b_1} \pm 2^{a_2}3^{b_2}$ for $0 \leq a_1 < a_2 \leq A$ and $0 \leq b_1 < b_2 \leq B$. In general, we compute the set of representations

$$R = R_1 \cup R_2 \cup ... \cup R_m$$

for some maximum number of terms m where

$$R_k = \left\{ \sum_{i=1}^{k} \pm 2^{a_i} 3^{b_i} \;:\; 0 \leq a_1 < \cdots < a_k \leq A \text{ and } 0 \leq b_1 < \cdots < b_k \leq B \right\}.$$

We iterate over the set R and for each integer represented, we record the lowest cost representation for that integer in R.

In our experiments, we search for representations for 16-bit integers. We initially chose $A = \lceil \log_2 (2^{16}) \rceil = 16$, $B = \lceil \log_3 (2^{16}) \rceil = 11$, and iterated the number of terms k from 1 through 5. For representations of $k = 6$ terms, our implementation did not complete after a week of execution. We then ran our search again using $A = 18$ and $B = 12$ and found that none of our minimal representations were improved.

Since both the incremental search of this section and the two functions from the previous section are computationally expensive, we are only able to compute chains for small exponents. This is still useful when we consider methods that partition large exponents into smaller blocks using their binary representation, or when we consider multiple exponentiations by a list of prime powers.

7.9 Big Exponents from Small Exponents

The techniques of the previous two sections are computationally expensive and as such, we limit our search to representations of 16-bit integers. One way that we use such representations is to write the exponent n in 16-bit blocks as

$$n = \sum_{i=0}^{k} 2^{16i} b_i$$

where $0 \leq b_i < 2^{16}$. Assuming that we can exponentiate a group element g to a 16-bit exponent b_i, Algorithm 7.2 computes the exponentiation g^n using an approach similar to a left-to-right windowed binary exponentiation.

Another way we can use 16-bit representations is when we know the prime factorization of the exponent n and n is the product of primes smaller than 2^{16}. Since we are interested in exponentiation by primorials (i.e. the product of the first w primes), the prime factorization is trivial. Given a primorial $P_w = \prod_{i=1}^{w} p_i$ where p_i is the i^{th} prime, we can compute g^{P_w} as $(((g^{p_1})^{p_2})^{\cdots})^{p_w}$ using a series of w small exponentiations.

Algorithm 7.2 16-bit Blocked Exponentiation.

```
Input: $n \in \mathbb{Z}, g \in G$ and a method to compute $g^{b_i}$ for $0 \leq b_i < 2^{16}$.
Output: $g^n$
$R \leftarrow 1_G$
for $i$ from $\lceil \log_{2^{16}} n \rceil$ downto 0 do
    $R \leftarrow R^{2^{16}}$                                  {By repeated squaring.}
    $b_i \leftarrow \lfloor n/2^{16i} \rfloor \bmod 2^{16}$
    $R \leftarrow R \cdot g^{b_i}$                              {Externally compute $g^{b_i}$.}
return $R$
```

7.10 Experimental Results

One reason to improve the performance of exponentiation in the ideal class group is to improve the performance of order finding in this group. In particular, we exponentiate a random ideal class to a primorial so that the order of the resulting ideal class is likely to be coprime to our exponent.

Two of the techniques in the previous section compute g^n using a series of smaller exponentiations g^b such that $0 \leq b < 2^{16}$. So we begin by determining the best method to exponentiate using 16-bit exponents. To do so, we compute the average time to exponentiate for exponents 1 through 65535 using 59-bit and 118-bit discriminants. Table 7.1 shows the number of exponents for which each method was the fastest[6]. For each exponent, we then normalized the time of each method relative to the best time. Table 7.2 shows the average of these normalized times. The method that maintains the 4 best partial representations of the form $(x \pm 2^a 3^b)/(2^c 3^d)$ was the best performer in general. However, since we were

Method	59-bit Discriminants	118-bit Discriminants
Left-to-Right 2,3 Representation	2536	7097
4 Best $\big\lvert \lvert x \rvert - 2^a 3^b \big\rvert$	9091	7230
4 Best $(x \pm 2^a 3^b)/(2^c 3^d)$	47969	45468
4 Best $(x \pm 2^a \pm 3^b)/(2^c 3^c)$	3333	2756
Recursive $\sum 2^a 3^b$ Chains	2406	2970
Incremental Search	199	13

Table 7.1: The number of 16-bit exponents where an exponentiation technique was fastest.

[6] The methods not listed in this table were not the fastest for any exponents.

Method	59-bit Discriminants	118-bit Discriminants
Binary	1.48864	1.44287
Right-to-Left Non-Adjacent Form	1.39605	1.34547
Right-to-Left 2,3 Chain	1.35869	1.37595
$2^2 3^2$ Windowed Right-to-Left 2,3 Chain	1.38676	1.37415
Left-to-Right 2,3 Representation	1.27775	1.23781
4 Best $\left\|\|x\| - 2^a 3^b\right\|$	1.19508	1.17034
4 Best $(x \pm 1)/(2^c 3^d)$	1.23152	1.22800
4 Best $(x \pm 2^a 3^b)/(2^c 3^d)$	1.03227	1.03886
4 Best $(x \pm 2^a \pm 3^b)/(2^c 3^c)$	1.37963	1.38966
Recursive $\sum 2^a 3^b$ Chains	1.28606	1.26822
Recursive $\sum \pm 2^a 3^b$ Chains	1.21295	1.18712
Incremental Search	1.19495	1.17059

Table 7.2: Normalized time to exponentiate 16-bit exponents.

interested in precomputing 16-bit representations for use with a block exponentiation and exponentiation by a list of prime factors, we used the best representation available for each exponent.

To determine the best method to exponentiate a random ideal by a primorial, we compute the average time to exponentiate for the primorials P_{250k} for $1 \le k \le 100$. We categorized the different exponentiation techniques as those that use only base 2, those that generate 2,3 chains from right-to-left, from left-to-right, that add or subtract to a partial representation and then reduce by $2^c 3^d$, and those that make use of the best 16-bit representations. We then compared the average time to exponentiate each method from a category, and finally compared the best performers of each category to determine the best performer overall.

We found that for both our 64-bit and 128-bit implementations and for all primorials tested, the method of iterating on the 4 best $\left||x| - 2^a 3^b\right|$ approximations leads to representations with the fastest time to exponentiate. This is in contrast to the method of iterating on the 4 best partial representations $(x \pm 2^a 3^b)/(2^c 3^d)$ that lead to the best timings for 16-bit integers. Naturally, we can improve these results by iterating on the L best partial representations for $L > 4$ at the expense of longer precomputation.

Figures 7.3 and 7.4 compare binary exponentiation against right-to-left non-adjacent form representation. The non-adjacent form representation leads to faster exponentiations in all cases. Figures 7.5 and 7.6 compare the non-windowed right-to-left 2,3 chain method to the $2^2 3^2$ windowed method. The $2^2 3^2$ windowed method out performs the non-windowed method for all exponents. Figures 7.7 and 7.8 compare left-to-right 2,3 representations with that of maintaining the 4 best $\left||x| - 2^a 3^b\right|$ approximations. In this case, maintaining the 4 best approximations performs best. Figures 7.9 and 7.10 compare the three different techniques of adding or subtracting a value to a partial representation and then reducing by a power of 2 and 3. Here, computing candidates $(x - 2^a 3^b)/(2^c 3^d)$ leads to representations that exponentiate the fastest. Figures 7.11 and 7.12 compare the two methods that rely on the best found 16-bit representations. In this case, when the factorization of the exponent is easily known, it is faster to exponentiation by the list of prime factors than it is to represent the exponent using 16-bit blocks. Finally, Figures 7.13 and 7.14 compare the best performer from each category.

7.11 Summary

This chapter presented the approach and results used to improve exponentiation in the ideal class group of imaginary quadratic number fields. Several methods for exponentiation including novel extensions to methods for computing 2,3 chains and representations were explored. We found that for 16-bit exponents, the method of iterating on the L best partial representations $x + \sum s_i 2^{a_i} 3^{b_i}$ such that each x generates new terms of the form $(x \pm 2^a 3^b)/(2^c 3^d) \in \mathbb{Z}$ gives the fastest representation on average. For large primorial exponents, generating new terms of the form $\left||x| - 2^a 3^b\right|$ gave the fastest representations on average.

The next chapter brings together the improvements to ideal arithmetic and exponentiation in our implementation of SuperSPAR – an algorithm with the fastest average time to factor integers n in the range $2^{54} \leq n < 2^{63}$ of the integer factoring algorithms that we tested.

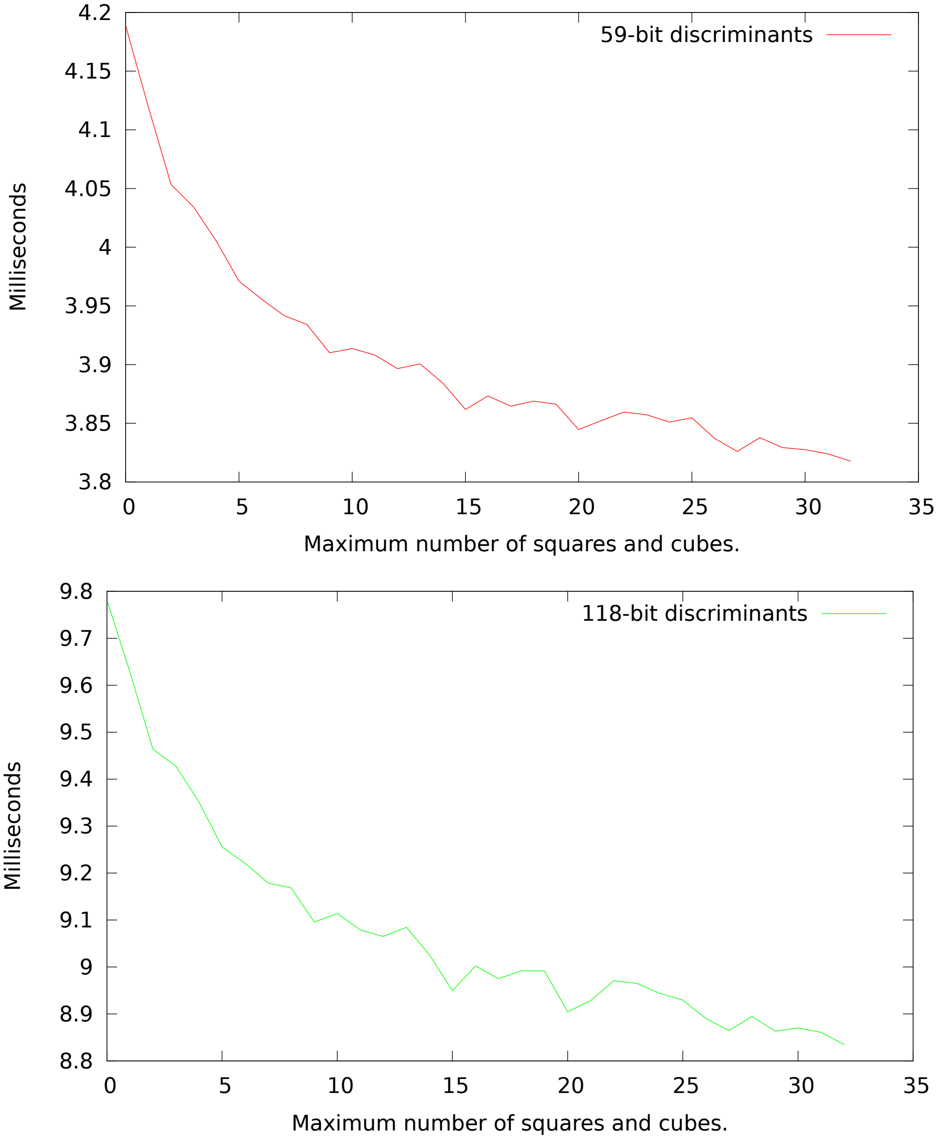

Figure 7.2: Performance of varying the bounds $a, b \leq U$ on $(x \pm 2^a 3^b)/(2^c 3^d)$ when exponentiating by the primorial P_{1000}.

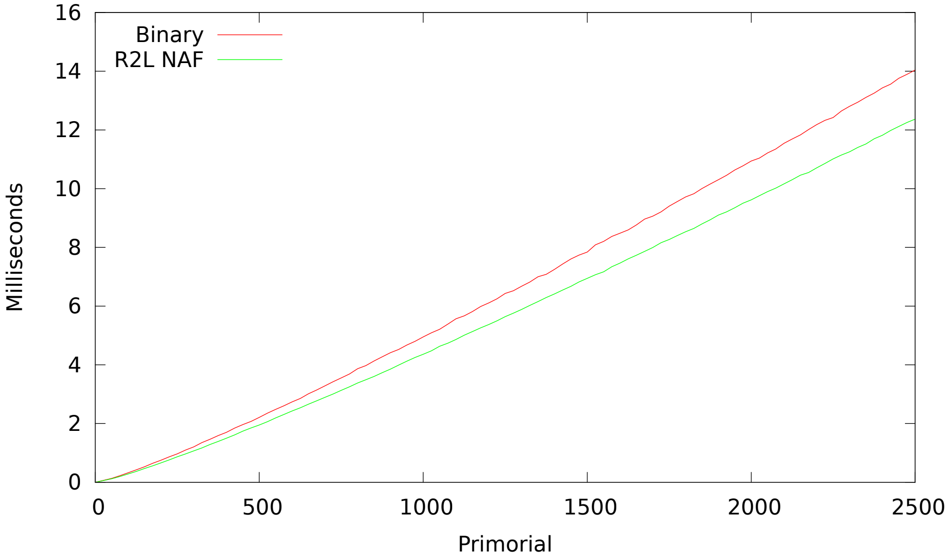

Figure 7.3: Base 2 Exponentiation (59-bit Discriminants).

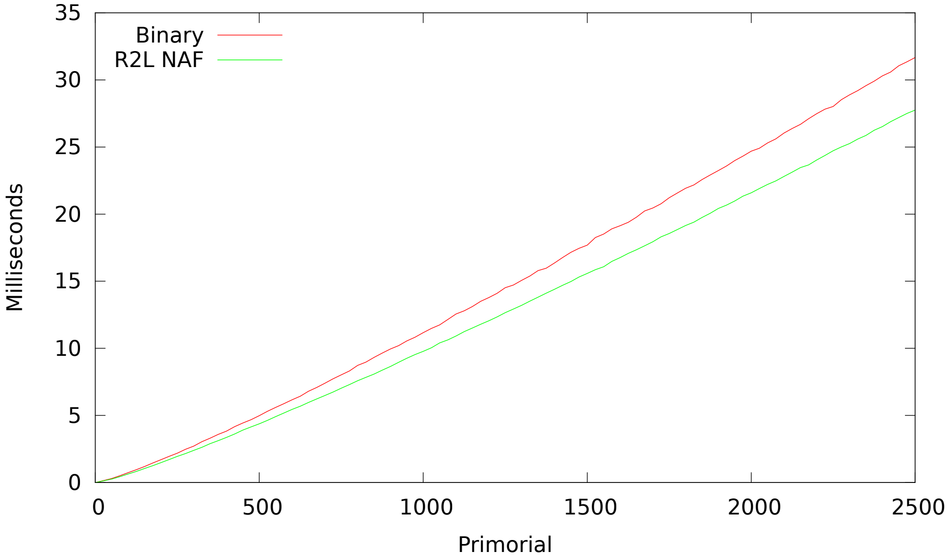

Figure 7.4: Base 2 Exponentiation (118-bit Discriminants).

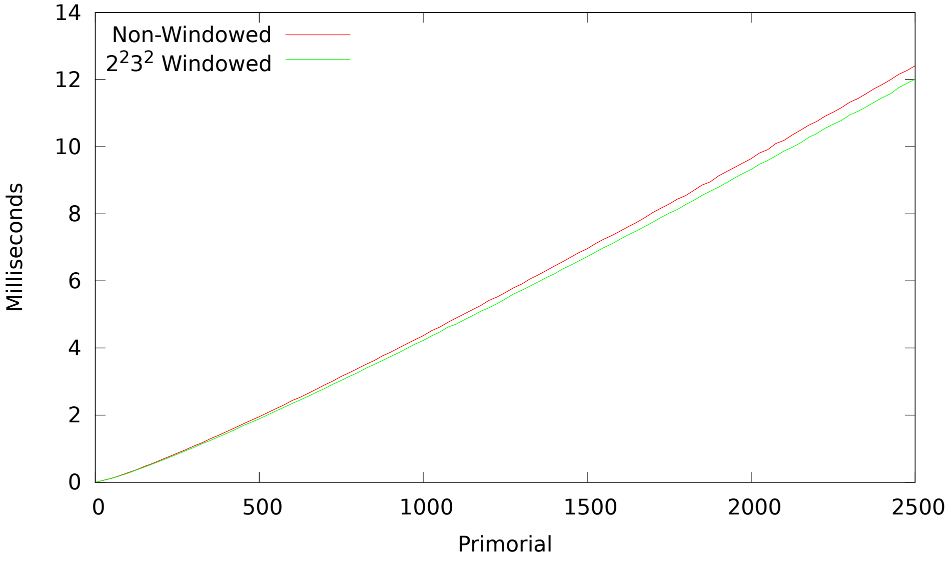

Figure 7.5: Right-to-Left 2,3 Chains (59-bit Discriminants).

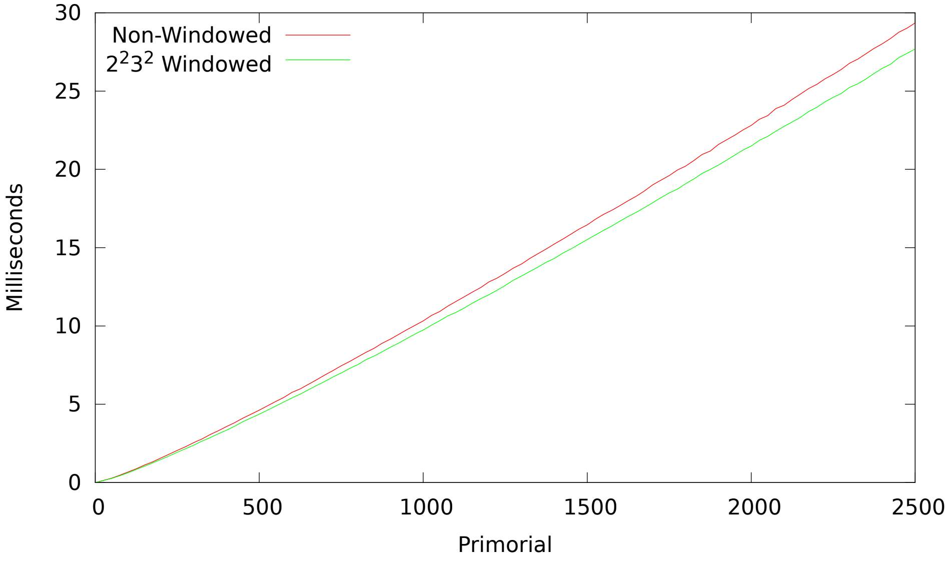

Figure 7.6: Right-to-Left 2,3 Chains (118-bit Discriminants).

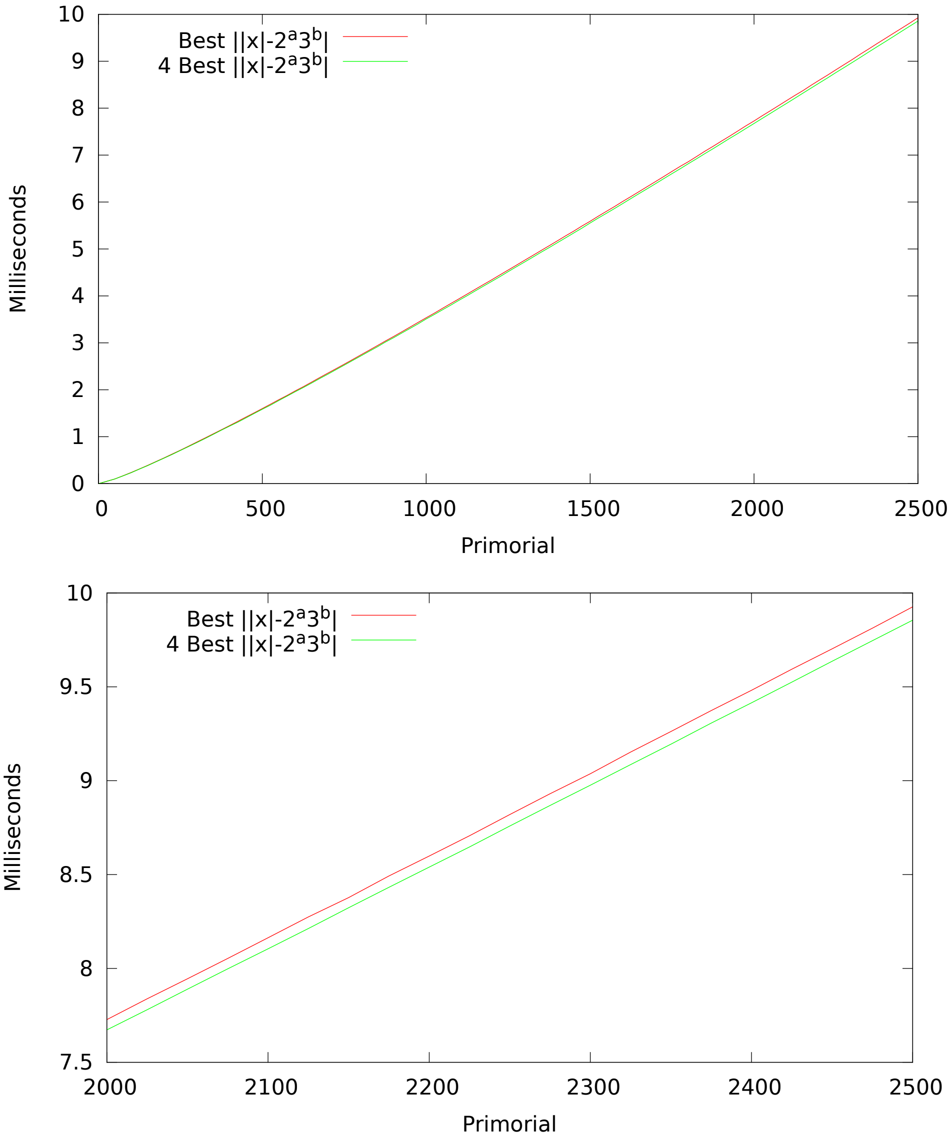

Figure 7.7: Left-to-Right 2,3 Representations (59-bit Discriminants).

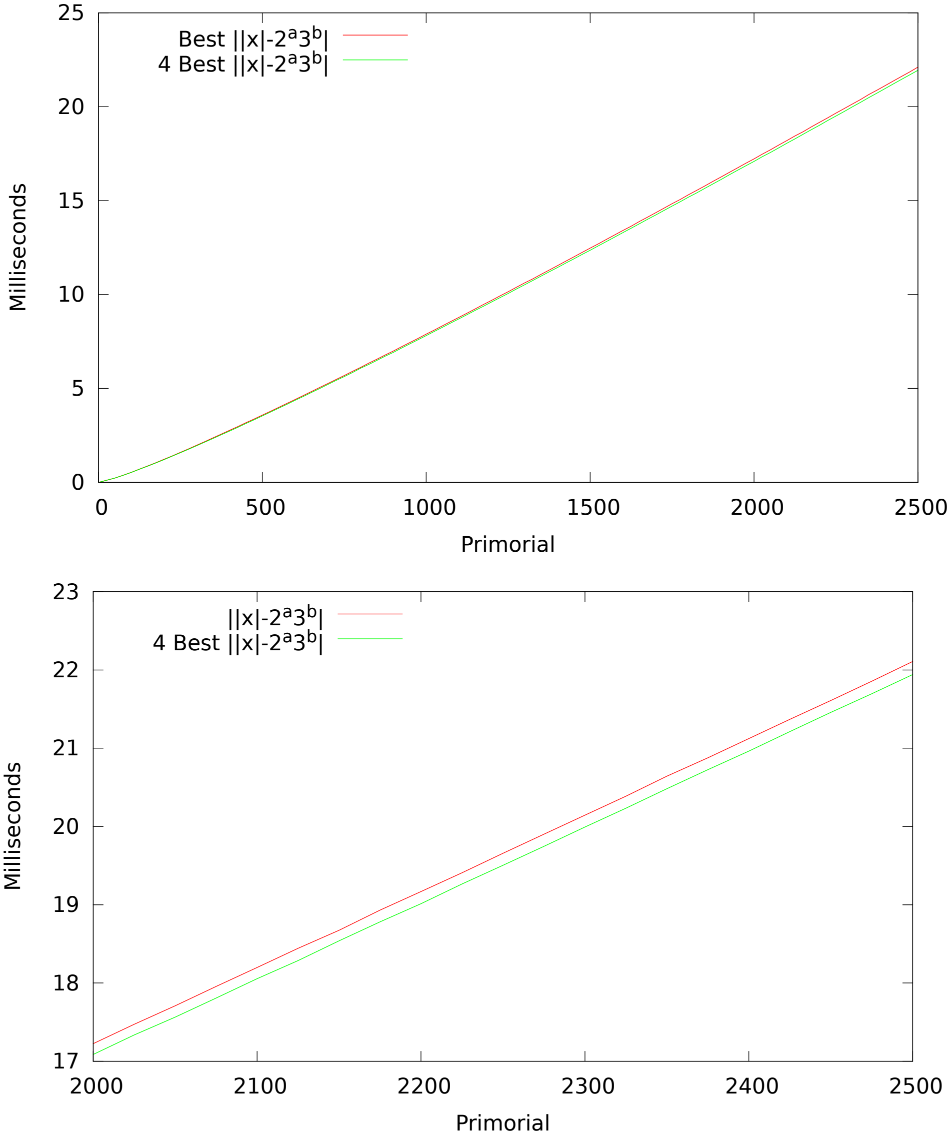

Figure 7.8: Left-to-Right 2,3 Representations (118-bit Discriminants).

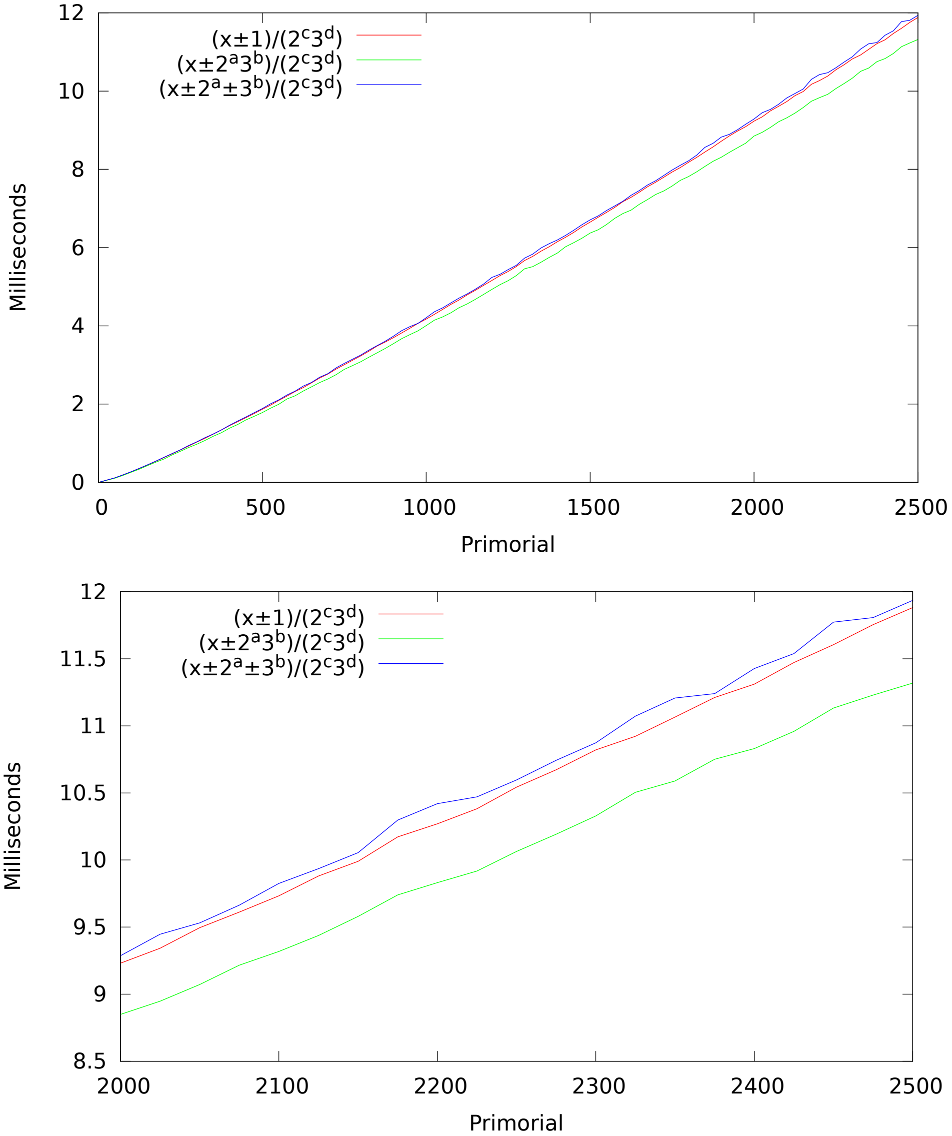

Figure 7.9: 4 Best $(x \pm \cdots)/(2^c 3^d)$ (59-bit Discriminants).

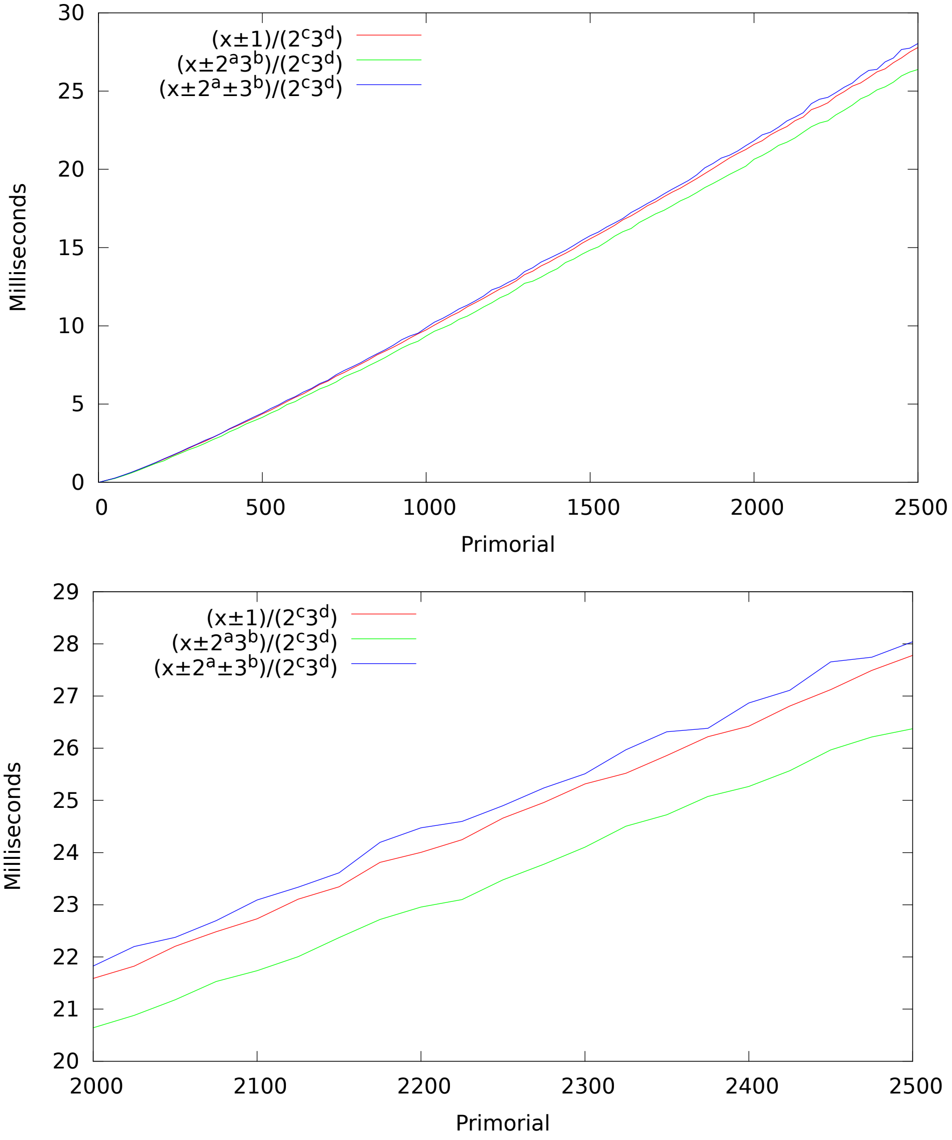

Figure 7.10: 4 Best $(x \pm \cdots)/(2^c 3^d)$ (118-bit Discriminants).

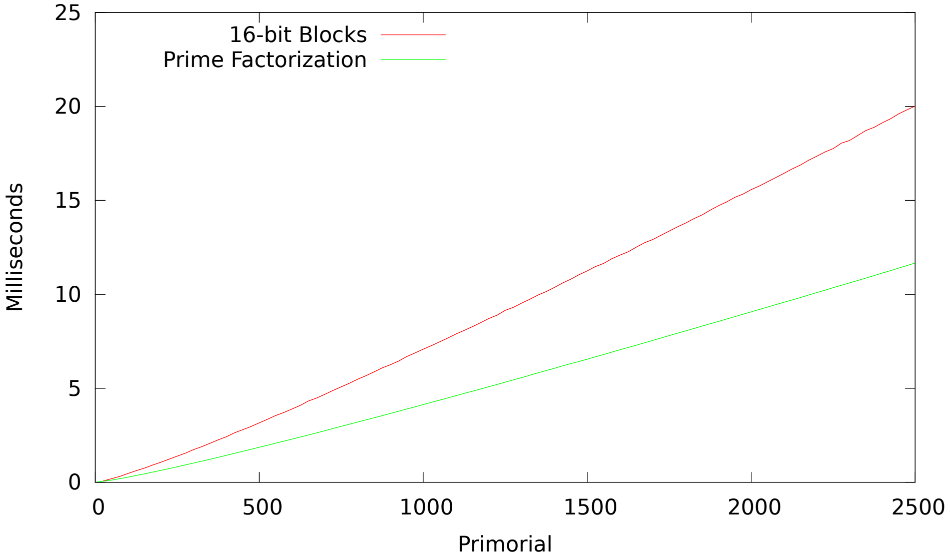

Figure 7.11: Use g^b for 16-bit b (59-bit Discriminants).

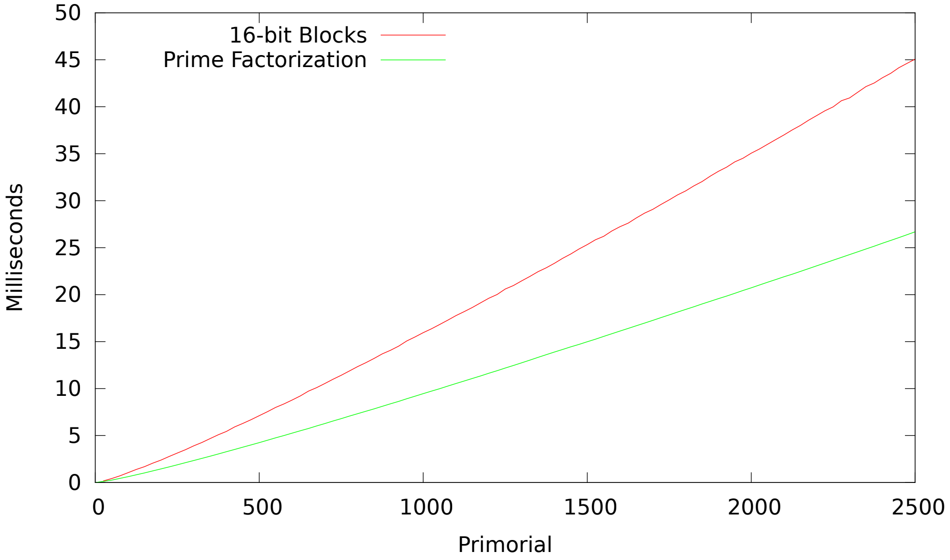

Figure 7.12: Use g^b for 16-bit b (118-bit Discriminants).

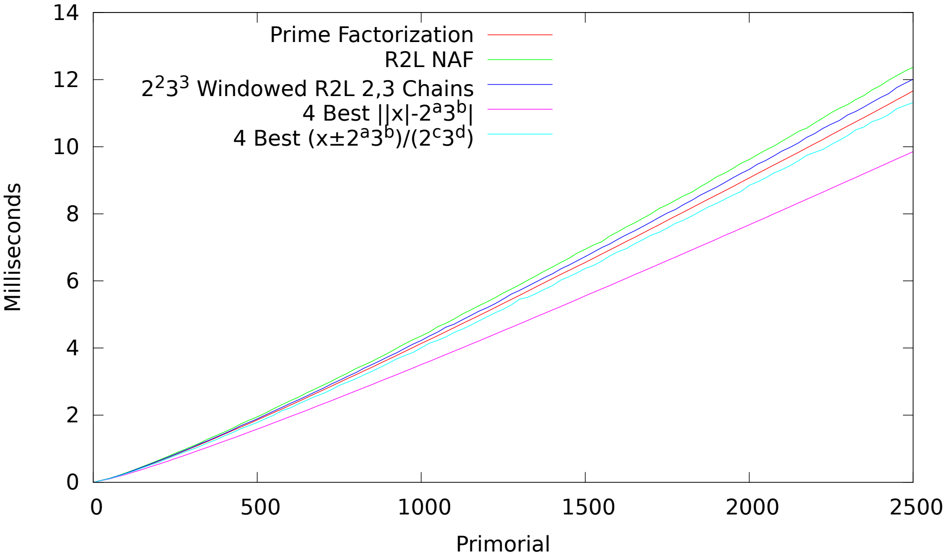

Figure 7.13: The best performers from each category (59-bit Discriminants).

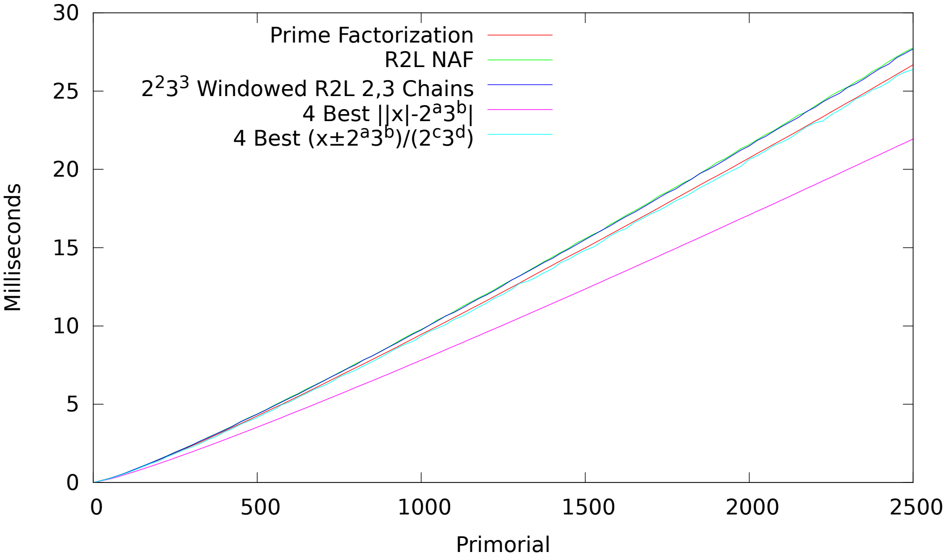

Figure 7.14: The best performers from each category (118-bit Discriminants).

Chapter 8

SuperSPAR Experiments

The goal of this chapter is to demonstrate that the SPAR integer factoring algorithm (Section 4.1) can be modified and implemented so that, on average, it is the fastest available factoring implementation for integers of the size 50-bits to 62-bits – this is an implementation of SuperSPAR (Section 4.2). One way in which factoring integers of these sizes is useful is after the sieve step of more powerful integer factoring algorithms, such as the number field sieve (see [19, §6.2]). Factoring integers of these sizes is also of particular interest in this thesis since Chapter 5 demonstrates performance improvements on average for computing the extended GCD using our 32-bit, 64-bit, and 128-bit implementations, while Chapter 6 demonstrates performance improvements on average for ideal class group arithmetic for orders with discriminant no larger than 118-bits. This is relevant to our implementation of SuperSPAR, since SuperSPAR is based on the SPAR factoring algorithm (discussed in Section 4.1), which is based on arithmetic in the ideal class group.

SPAR factors an integer N by attempting to compute the order of a random ideal class in an order with discriminant $\Delta = -kN$ for some square free integer k. An implementation first exponentiates a random ideal class to the product of many small primes (the exponentiation stage) and then performs a random walk in an attempt to find an ambiguous ideal class (the search stage). Our implementation of SuperSPAR improves upon the search stage of SPAR by using a bounded primorial steps search, due to Sutherland [50] and discussed in Subsection 4.2.1. This is a practical improvement over a random walk search since a bounded primorial steps search has an asymptotically faster worst case (see Subsection 4.2.1), which means that it requires fewer group operations to find the order of an ideal class.

Our implementation of SuperSPAR also improves upon the exponentiation stage of

SPAR. In this stage, SPAR exponentiates a random ideal class to the product of many small primes. Since our implementations of ideal class arithmetic include cubing, we take advantage of 2,3 representations for exponentiation, which is an improvement over binary exponentiation (see Section 7.10). For SuperSPAR, we precompute 2,3 representations using the L best approximations technique of Section 7.6, which Chapter 7 shows to perform the best on average for large exponents that are the product of many primes and for ideal classes for orders with discriminants no larger than 118-bits.

Another way in which our implementation of SuperSPAR improves upon SPAR is by bounding the number of group operations used by each stage independently from each other and separate from the theoretically optimal values. In SPAR, each stage uses $O(p_t)$ group operations where $p_t \approx N^{1/2r}$ and $r = \sqrt{\ln N / \ln \ln N}$ (see Subsection 4.1.2 for additional details). The value for r is chosen to theoretically minimize the expected running time of the algorithm, since the assumption is that the order of a random ideal class $[\mathfrak{a}] \in Cl_\Delta$ with $\Delta = -kN$ is $N^{1/2r}$ smooth with probability r^{-r}. In practice, other values of p_t are more efficient.

Furthermore, balancing both stages to use $O(p_t)$ group operations is not ideal. This is partly because the actual cost of multiplication, squaring, and cubing differ, but also because the success of each stage depends on different properties of the class number. The exponentiation stage is successful when the order of a random ideal class is smooth with respect to the exponent E used in the exponentiation stage, while the search stage is successful when the non-smooth part of the order is sufficiently small. Selecting bounds for each stage independently of r varies the time spent during each stage inversely to the probability of its success. As such, our implementation of SuperSPAR selects the largest prime p_t, used in the exponent $E = \prod {p_i}^{e_i}$ in the exponentiation stage, separately from the multiple of a primorial, mP_w, used in the search stage. Note that the largest prime p_t used during the exponentiation stage is larger than the largest prime p_w used by the search stage, since the search stage assumes that the order of $[\mathfrak{b}] = [\mathfrak{a}]^E$ is coprime to E.

In addition to allowing the largest prime p_t used in the exponent $E = \prod {p_i}^{e_i}$ to differ from the theoretically optimal value, we also bound the prime exponents e_i differently from the theoretical values suggested by Schnorr and Lenstra, who incidentally advise the use of smaller exponents in practice (see [45, p.293] and Subsection 8.3.1 of this thesis).

Since there are several parameters affecting the performance of our implementation of SuperSPAR, we first say a little about the methodology used in this chapter. This chapter, like previous chapters, emphasizes empirical results, however, factoring a single integer is considerably slower than computing a single extended GCD or performing a single arithmetic operation in the ideal class group. In the case of computing the extended GCD, a single computation takes on average between a few nanoseconds to a few hundred nanoseconds. This means that we can empirically test 1,000,000 computations for random integers of each size from 1-bit to 127-bits in a modest amount of time. Even for ideal class arithmetic, multiplication, squaring, and cubing take between a few hundred nanoseconds to a few microseconds. Again, because of this we are able to empirically time each operation on 10,000,000 random ideal classes for discriminants of size 16-bits to 140-bits in a few hours. On the other hand, factoring a single integer with our implementation of SuperSPAR takes less than 100 microseconds on average for properly chosen parameters for 32-bit integers, but over 50 milliseconds on average for poorly chosen parameters for 100-bit integers – this is roughly between 10,000 times to 500,000 times slower than an extended GCD computation. So even for small sets of integers (a few 1000), this is quite slow and may not lead to parameters that work well on average.

To aid in the selection of efficient parameters for our implementation of SuperSPAR, Section 8.3 discusses the statistical analysis of a set of roughly 6,000,000 ideal classes in order to find bounds on several parameters. Subsection 8.3.1 shows that the prime factorization of the order of ideal classes consists of prime powers with exponents that are typically 1 for all but the smallest primes. This informs the value of each e_i in the exponent E for the exponentiation stage. Additionally, we found that the order of an ideal class $[\mathfrak{a}_1]$ was, with high probability (about 97.8% in our experiments), either the same as or a multiple of the order of some other ideal class $[\mathfrak{a}_2]$ in the same class group (see Subsection 8.3.2). For this reason, if the algorithm finds some h' such that $[\mathfrak{a}_1]^{h'\left(2^j\right)}$ is an ambiguous class for some j, but is unsuccessful at factoring N, the algorithm simply attempts to find an ambiguous class by repeated squaring of $[\mathfrak{a}_i]^{h'}$ for several $[\mathfrak{a}_i]$. Since this may fail for several ideal classes, we let c be the maximum number of ideal class classes tried before the algorithm starts over with a different multiplier k for the discriminant. Although the statistical data shows that in expectation a value of $c \approx 2$ is all that is needed (see Subsection 8.3.4), we rely on empirical testing to show that much larger values of c perform better in practice. Similarly, the largest prime p_t used in the exponent E for the exponentiation stage, and the multiple of a primorial mP_w used for the search stage are determined empirically in Section 8.6.

The software used to generate timing information was developed using GNU C compiler version 4.7.2 on 64-bit Ubuntu 12.10. Assembly language was used for 128-bit arithmetic on x64 processors and for processor specific features. The C programming language was used for all other implementation details. The CPU is a 2.7GHz Intel Core i7-2620M CPU with 8Gb of RAM and has four cores, only one of which was used for timing experiments. Unless otherwise specified, the timings associated with factoring integers are the average time to factor one integer from a set of 4,000 semiprime integers, such that each integer is the product of two primes of equal size. All empirical timings for integers of a given size use the same set of 4,000 semiprime integers.

Exponents used for the exponentiation stage were precomputed using the L best approximations technique from Section 7.6, which Chapter 7 shows performs the fastest for exponents that are the products of many small primes and for our implementations of ideal class group arithmetic. The implementation of ideal class arithmetic used is that discussed in Chapter 6, specifically, we use the 64-bit implementation for all class groups with discriminant no larger than 59-bits, and we use the 128-bit implementation for class group with discriminant larger than 59-bits, but no larger than 118-bits. Our implementation of SuperSPAR will work for integers larger than 118-bits, however, empirical tests show that SuperSPAR was not competitive for larger integers. Extended GCD computations used for ideal arithmetic are based on the results of Chapter 5. For an ideal $\mathfrak{a} = [a, (b + \sqrt{\Delta})/2]$, the coefficients a and b are roughly half the size of the discriminant Δ. As such, our 64-bit implementation of ideal arithmetic favours our 32-bit implementations of the extended Euclidean algorithm, however, when arguments are larger than 32-bits, our implementation uses our 64-bit implementations of our simplified left-to-right binary extended GCDs. Our 128-bit implementation of ideal arithmetic favours of 64-bit implementations of our simplified left-to-right binary extended GCDs, but similarly, when arguments are larger than 64-bits, our implementation uses our 128-bit implementation of Lehmer's extended GCD using our 64-bit simplified left-to-right binary extended GCD for 64-bit machine words. In each case, when a left-only or partial extended GCD is needed, our implementation uses the left-only or partial implementation of the corresponding full extended GCD. Our GMP implementation of ideal arithmetic is not utilized during these tests since our implementation of SuperSPAR was not competitive for integers larger than 118-bits, which is the largest size supported by our 128-bit implementation of ideal class arithmetic.

The sections in this chapter are as follows. Section 8.1 discusses two techniques to generate a list of integers (or deltas between integers) coprime to a primorial up to a given bound. This is useful during the search stage of SuperSPAR when we take baby steps coprime to the multiple of a primorial mP_w. The search stage requires the use of a lookup table. Section 8.2 discusses how our implementation hashes an ideal class and then compares two methods of resolving hash collisions, namely chaining with list heads and open addressing. Open addressing performs faster on average. As discussed in Subsection 4.2.1, for the search stage to work, the order of the element should have no factors in common with the primorial used for the search stage. Section 8.3 discusses the data and experiments used to bound the exponents of the prime factors of the exponent used in the exponentiation stage so that (with high probability) the order of the resulting ideal class will have no factors in common with the primorial used during the search stage. This section also discusses the number of ideal classes to try in expectation for a given class group. To enable the discovery of bounds useful for the search stage, Section 8.5 discusses a method of generating suitable candidates. With all this in place, Section 8.6 discusses the method we use to determine the exponent for the exponentiation stage and the bound for the search stage that work well in practice for integers of specific sizes. Finally, a comparison of our implementation of SuperSPAR to several public implementations of factoring algorithms is provided. We show that for pseudorandom semiprime integers between 50-bits and 62-bits inclusive, SuperSPAR performs the best on average of the implementations tested.

8.1 Coprime Finding

The bounded primorial steps algorithm from Subsection 4.2.1 and adapted by the SuperSPAR factoring algorithm (Section 4.2) computes baby-steps $[\mathfrak{a}]^i$ for $1 \leq i \leq s$ where i is coprime to some primorial P_w and s is a multiple of P_w. By considering only values coprime to P_w the running time complexity of the bounded primorial steps algorithm achieves an upper bound of $O(\sqrt{M/\log\log M})$ group operations in the worst case. If an implementation has to test whether each i from 1 to P_w is coprime to P_w, the algorithm cannot achieve this. As such, the algorithm assumes a set of values coprime to P_w is globally available. Of course, one can generate such a set by testing whether $\gcd(i, P_w) = 1$ for i from 1 to P_w, but there are more efficient methods. This section discusses two such methods, namely sieving and wheeling. Additional information on each technique can be found in [19, pp.117–127] and [51, p.494].

8.1.1 Coprime Sieve

Sieving the values coprime to some primorial P_w is straightforward since the factorization $P_w = 2 \times 3 \times \cdots \times p_w$ is known. Let $[1, b]$ denote the range to be sieved. First, create a table mapping the values 1 through b to *true*. Then for each prime factor p_j of P_w, and for every multiple mp_j such that $1 \leq mp_j \leq b$, set the table entry at index mp_j to *false*. At this point, each index with a *true* value is coprime to P_w. To see this, note that an index x is *false* when there was some multiple m of a prime p_j dividing P_w such that $mp_j = x$. As such, the value of x is false if and only if x and P_w share a common factor, namely p_j.

8.1.2 Coprime Wheel

When the value b from the previous section is equal to the primorial P_w, a coprime wheel may be faster in practice. First, suppose that for some primorial P_w, the set of all integers $1 \leq x < P_w$ coprime to P_w is already known. More formally, let

$$\mathcal{C}_w = \left\{x : \gcd(x, P_w) = 1, 1 \leq x < P_w\right\}.$$

As such, there is no $x \in \mathcal{C}_w$ that is divisible by any prime p_j that divides P_w. The values $x + mP_w$ for $x \in \mathcal{C}_w$ and $m \in \mathbb{Z}$ are also coprime to P_w, and so act as a set of representatives of the integers modulo P_w that are coprime to P_w. Let $\mathcal{C}_w + mP_w$ denote the set $\{x + mP_w : x \in \mathcal{C}_w\}$. Computing

$$\bigcup_{0 \leq m < p_{w+1}} \mathcal{C}_w + mP_w$$

generates the set of all positive integers less than P_{w+1} that are coprime to P_w. Removing all multiples of p_{w+1} from this set implies that no element is divisible by a prime $p_j \leq p_{w+1}$. As such, the set

$$\mathcal{C}_{w+1} = \left(\bigcup_{0 \leq m < p_{w+1}} \mathcal{C}_w + mP_w\right) \setminus \{m' p_{w+1} : 1 \leq m' < P_w\}$$

is the set of all values $1 \leq x < P_{w+1}$ that are coprime to P_{w+1}.

The primorial $P_1 = 2$ has the set $\mathcal{C}_1 = \{1\}$ of integers coprime to 2. This acts as a base case, from which the representative set of integers coprime to primorials $P_2, P_3, ..., P_w$ are computed by recursive application of the above steps.

Our implementation of SuperSPAR iterates baby steps for consecutive integers coprime to some primorial P_w. As such, the difference between coprime integers is used instead, and the above technique is adapted to work with lists of deltas between consecutive coprime integers. Each list begins with the difference from the first coprime integer 1 (which is coprime to all integers) to the next integer coprime to some primorial P_w. Let

$$\mathcal{D}_w = d_1, ..., d_{\phi(P_w)}$$

such that $d_i = c_{i+1} - c_i$ for consecutive integers c_i and c_{i+1} coprime to P_w where $c_1 = 1$. The list $\mathcal{D}_1$ consists of the single value 2.

Algorithm 8.1 Compute deltas for P_{w+1} given deltas for P_w.

```
Input: A primorial $P_w$, the delta list $\mathcal{D}_w$, and the next prime $p_{w+1}$.
    Let $P_{w+1} = p_{w+1}P_w$ and $\phi(P_{w+1}) = (p_{w+1} - 1)\phi(P_w)$.
Output: The delta list $\mathcal{D}_{w+1}$.
$i \leftarrow 1, j \leftarrow 1, c \leftarrow 1, d \leftarrow 0$
while $j \le \phi(P_{w+1})$ do
    $c \leftarrow c + d_i$                                   {each $c$ is coprime to $P_w$}
    $d \leftarrow d + d_i$
    if $c \not\equiv 0 \pmod{p_{w+1}}$ then
        $d'_j \leftarrow d, d \leftarrow 0, j \leftarrow j + 1$        {output $d$ when $c$ is coprime to $P_{w+1}$}
    $i \leftarrow i + 1$
    if $i > \phi(P_w)$ then
        $i \leftarrow 1$
return $d'_1, ..., d'_{\phi(P_{w+1})}$
```

Algorithm 8.1 computes $\mathcal{D}_{w+1}$ given $\mathcal{D}_w$, P_w, and the next prime p_{w+1}. The algorithm starts with the current candidate integer, c, coprime to P_{w+1} and a delta counter $d = 0$. The algorithm cycles through the delta list $\mathcal{D}_w$ adding each element encountered to c. On each iteration, if c is not a multiple of p_{w+1}, then the delta counter d is appended to the output list $\mathcal{D}_{w+1}$ and then set to 0. Otherwise, the algorithm continues with the next delta from the input list (cycling to the beginning of the list when the end is reached). Once $\phi(P_{w+1})$ deltas have been appended to the output list, the algorithm terminates.

8.1.3 Comparison

The coprime sieve requires b bits of memory, while the coprime wheel requires $\phi(P_w)$ integers for the list of deltas. Table 8.1 shows the values of $\phi(P_w)$ for the first 12 primorials. The delta list for the tenth primorial, $\phi(P_{10}) = 1021870080$, requires over a billion integers. As such, the coprime sieve is preferred when the primorial is large, while the coprime wheel may be more efficient for large values of b. This is because the coprime wheel generates deltas directly, whereas each entry in the sieve must be inspected to find integers coprime to the primorial.

w	p_w	$P_w = \prod_{p \le p_w} p$	$\phi(P_w) = \prod_{p \le p_w} (p-1)$
1	2	2	1
2	3	6	2
3	5	30	8
4	7	210	48
5	11	2310	480
6	13	30030	5760
7	17	510510	92160
8	19	9699690	1658880
9	23	223092870	36495360
10	29	6469693230	1021870080
11	31	200560490130	30656102400
12	37	7420738134810	1103619686400

Table 8.1: The first 12 primorials.

Our implementation of SuperSPAR computes baby steps coprime to a multiple of a primorial, and does so by iterating a list of deltas from one coprime value to the next. Since this list consists of $\phi(P_w)$ integers, in practice our implementation is limited to coprime delta lists that fit within available memory, and as such to primorials P_w that are not too large. On the other hand, our implementation is still able to use much larger primorials for the exponentiation stage, since the space needed to represent a primorial is $O(\log(P_w))$ while the space needed to represent the list of coprime deltas is $O(\phi(P_w))$. One reason we might want to use a larger primorial in the exponentiation stage is that by doing so, the order of resulting ideal class $[\mathfrak{b}] = [\mathfrak{a}]^E$ is likely to be coprime to E and so any factors of the order of $[\mathfrak{a}]$ that were in common with the factors of E are likely to be removed by the exponentiation stage. The order of the resulting ideal class $[\mathfrak{b}]$ will be smaller and as such, there is a greater probability that the search stage will discover the order of the ideal class $[\mathfrak{b}]$. We note that the largest prime used in the exponent of the exponentiation stage can be no smaller than the largest prime in the primorial of the search stage. If this were the case, the search stage would step coprime to a possible factor of the order of the ideal class $[\mathfrak{a}]$ and as such, the search stage could not possible discover the order of the ideal class $[\mathfrak{b}]$.

8.2 Lookup Tables

The SuperSPAR Algorithm 8.4 (and more generally, the bounded primorial steps algorithm 4.1) requires a lookup table mapping group elements $[\mathfrak{b}]^i$ to exponents i. The baby steps generate group elements $[\mathfrak{b}]^i$ with i coprime to some primorial P_w and store the mapping $[\mathfrak{b}]^i \mapsto i$ in the lookup table. The giant steps lookup group elements $[\mathfrak{b}]^{2js}$ and $[\mathfrak{b}]^{-2js}$ from the lookup table. As such, the performance of the algorithm directly corresponds to the performance of the lookup table. This section describes the experimental results for SuperSPAR using two different implementations of lookup tables, as well as the method used to map group elements to exponents in our implementation.

Our implementation of idea class arithmetic represents an ideal class $[\mathfrak{a}] \in Cl_\Delta$ using the triple (a, b, c) where $\mathfrak{a} = [a, (b + \sqrt{\Delta})/2]$ is a reduced representative for the ideal class and $c = (b^2 - \Delta)/4a$ (see Section 6.1). Since inversion of ideal classes is virtually free, SuperSPAR computes giant steps for ideal classes $[\mathfrak{b}]^{2js}$ and $[\mathfrak{b}]^{-2js}$, but rather than perform two table lookups, we take advantage of the way in which the inverse of an ideal class is computed. Recall form Subsection 2.5.3 that the inverse of an ideal class for a representative

$[a, (b+\sqrt{\Delta})/2]$ is given by the representative $[a, (-b+\sqrt{\Delta})]$. Since $|b| = \pm\sqrt{\Delta + 4ac}$, we use the pair (a, c) for the key and our implementation is able to look up both $[\mathfrak{b}]^{2sj}$ and $[\mathfrak{b}]^{-2js}$ using a single table lookup. However, once a giant step finds a corresponding $[\mathfrak{b}]^i$ in the table, both the exponents $2js - i$ and $2js + i$ must be tested to determine which one is a multiple of the order of $[\mathfrak{b}]$. Figure 8.1 contrasts this with an implementation using two lookups per giant step. Both implementations use identical parameters on the same sets of semiprime integers, i.e. the executions of the two implementations only differ in the number of table lookups performed during the giant step portion of the search stage. In the implementation using two table lookups per giant step, the pair (a, b) is used for the hash key in order to distinguish between $[\mathfrak{b}]^{2js}$ and $[\mathfrak{b}]^{-2js}$, however, when some $[\mathfrak{b}]^i$ is found, a multiple of the order is immediately known.

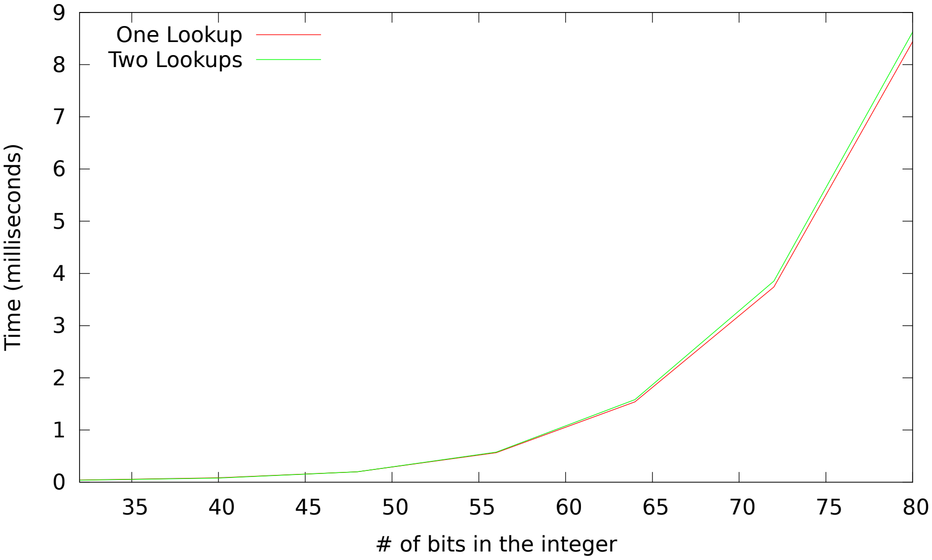

Figure 8.1: Average time to split an integer by either using a single lookup for each giant step, with exponentiation to verify a multiple of the order, or two lookups for each giant step, but without the need to verify a multiple of the order.

We implement the lookup table as a hash table with 32-bit keys and 32-bit values (both as unsigned integers). 32-bit values are sufficient since the theory predicts that for integers as large as 128-bits, that the largest exponent generated by the search stage still fits within 31-bits (see Table 8.7 and Section 8.5 for more details). To generate the 32-bit key, our implementation maps the pair (a, c) associated with an ideal class using

$$\text{hash}_{32}(a, c) = (2654435761 \times (2654435761a + c)) \bmod 2^{32}.$$

The multiplier 2654435761 was chosen because it is the largest prime less than $2^{32} \times (\sqrt{5} - 1)/2$, and $(\sqrt{5} - 1)/2$ has been shown by Knuth [36, Section 6.4] to have good scattering properties. Other multipliers were also tried but had little impact on performance. Since the number of baby steps is a function of the size of the integer to be factored (see Section 8.6), the maximum number of entries in the hash table is known at initialization. We found that a table of 2^k slots for the smallest k such that $2^k \geq 3m\phi(P_2)$, where $m\phi(P_w)$ is the number of baby steps, worked well in practice. Different multiples of the number of baby steps were tried, as well as setting the number of slots to a prime near some multiple of the number of baby steps. There was little difference in performance, with the exception of a table having too few slots (resulting in an insertion failing), or a table having an excessive number of slots (in which case, table initialization dominated). Using a power of 2 for the number of slots has the added benefit that once a hash key is computed, the corresponding slot is found by using a binary $\texttt{and}_2$ operation. However, regardless of the hash function used, multiple elements may hash to the same slot. Two methods of collision resolution are considered here, each being selected for cache coherency properties (see [18, Subsection 11.2] and [37]).

8.2.1 Chaining with List Heads

The first method is chaining with list heads (see [18, Subsection 11.2]). The idea is for the hash table to consist of 2^k slots, such that each slot operates as a linked list, but that the head of the linked lists are separate from the tails. Two areas of memory are used: one for the heads, and the other for the tails (also known as the overflow). In the head, each slot consists of the 32-bit value for a specified key, as well as the 32-bit hash key itself (since multiple keys may resolve to the same slot). A special pair of constants are used to indicate that a slot is unoccupied, e.g. (0xAAAAAAAA, 0x55555555). The overflow area consists of an array of no less than 2^k entries, each entry consisting of the 32-bit value for a specified hash key, the 32-bit hash key itself, and an index into the overflow area of the next node in the list, if any. A special constant, e.g. 0xFFFFFFFF, is used to indicate the end of list. The first 2^k entries in the overflow area are reserved and correspond to the second node in the linked list of each slot. To perform a lookup, the hash key resolves to a slot (using $\text{hash}_{32}(a, c) \bmod 2^k$) and the list is traversed until either the specified hash key or the end of the list is found. To insert or update, again, the list is traversed until either the specified hash key is found, at which point an update is performed, or the end of list is found. If the end of list is found, the next available entry in the overflow area is set appropriately and appended to the end of the list.

8.2.2 Open Addressing

The second method of collision resolution considered is open addressing, and in particular, we implement the same probing function as in CPython [37]. In this method, each slot consists of a 32-bit hash key, as well as the 32-bit value associated with that hash key. Any hash key can occupy any slot, and a special pair of constants are used to indicate that a slot is unoccupied, e.g. (0xAAAAAAAA, 0x55555555). When an element is hashed, the table is probed at slot indices $s_0, s_1, ...$ generated by a probing function until a slot is found where either the hash key occupying the slot is the same as the hash key for the element in question, or the slot is unoccupied. Our implementation defines probe indices using

$$\begin{aligned} s_0 &= \text{hash}_{32}(a, b) \bmod 2^k \\ s_i &= \left(5s_{i-1} + 1 + \left\lfloor \text{hash}_{32}(a, b)/2^{5i} \right\rfloor\right) \bmod 2^k. \end{aligned}$$

8.2.3 Chaining vs. Open Addressing

Figure 8.2 shows the average time to split semiprime integers of the specified size using the collision resolution methods described above. In this context and for each integer range tested, open addressing for collision resolution performs better.

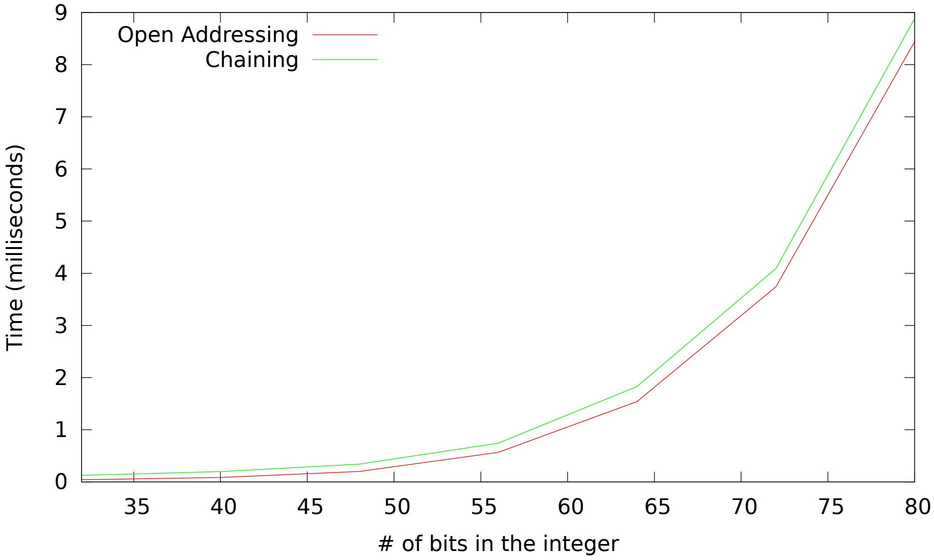

Figure 8.2: The average time to split an integer using either open addressing or chaining with list heads for collision resolution in a hash table.

8.3 Factorization of the Order of Ideal Classes

Although our implementation of SuperSPAR is the fastest of the implementations we tested for integers between 50-bits and 62-bits inclusive, this is for properly chosen parameters. For poorly chosen parameters and larger integers, SuperSPAR takes substantially longer. Also, in the introduction to this chapter, we identified several configuration parameters for our implementation of SuperSPAR based on the size of the input integer. One method for finding configuration parameters that perform well on average is to iterate the range of reasonable values for each parameter for each integer size, but this would be slow even for small sets of input integers. Furthermore, training the configuration parameters on a small set of integers is unlikely to lead to parameters that perform well on average.

Since this chapter is concerned with the performance of SuperSPAR in practice, the following integer sizes were selected to highlight the range in which SuperSPAR is competitive with other factoring algorithms, but also in order to minimize the size of operands. Pari/GP was used to generate as large a data set as possible, but that still fit within the available memory (8Gb) and could be processed reasonably quickly. For bit sizes $n \in \{32, 40, 48, 56, 64, 72, 80\}$, roughly 250,000 unique semiprime integers $N = p \cdot q$ were generated such that p and q are prime and half the size of N, while N is n bits in size. Then for each square free multiplier $k \in \{1, 2, 3, 5, 6, 7, 10\}$, discriminants $\Delta = -kN$ or $\Delta = -4kN$ such that $\Delta \equiv 0, 1 \pmod 4$ were generated. For each corresponding ideal class group Cl_Δ, ideal class representatives $[\mathfrak{p}] \in Cl_\Delta$ with $\mathfrak{p} = [p, (b + \sqrt{\Delta})/2]$ and p prime were generated for the 5 smallest values of p such that $\gcd(p, \Delta) = 1$ and $4p \mid b^2 - \Delta$. Finally, for each ideal class $[\mathfrak{p}]$, the order of $[\mathfrak{p}]$, its corresponding prime factorization, and whether it lead to to a successful splitting of the integer N were recorded. This data set is used throughout the computations in the following subsections and is denoted $\mathcal{D}$.

Subsection 8.3.1 uses data about the prime power factorization of the order of ideal classes to determine values for each e_i in the exponent E used by the exponentiation stage. In Subsection 8.3.2, we justify a strategy of reusing a known multiple of the order of an ideal class to search for an ambiguous class starting with several ideal classes from the same class group. We then provide evidence in Subsection 8.3.3, at least for the integer range studied, that certain multipliers are more likely to lead to a splitting of the input integer. Finally, Subsection 8.3.4 shows that once a multiple of the order of an ideal class is known, that in expectation we only need to try $c \approx 2$ ideal classes before successfully splitting the input integer. However, this is tested empirically and we see that larger values of c perform better in practice.

8.3.1 Prime Power Bound

SPAR (Section 4.1) exponentiates an ideal class $[\mathfrak{a}]$ to an exponent E where $E = \prod_{i=2}^{t} {p_i}^{e_i}$, p_i are prime, and $e_i = \max\{v : {p_i}^v \le {p_t}^2\}$ for some appropriately chosen prime p_t [45, p.290]. In practice however, Schnorr and Lenstra recommend using smaller exponents e_i such that $e_i = \max\{v : {p_i}^v \le p_t\}$ [45, p.293]. Doing so means that the exponentiation stage uses fewer group operations (and so takes less time), at the risk that the order of the resulting ideal class $[\mathfrak{b}] = [\mathfrak{a}]^E$ may still have small prime factors. Here we consider bounds for the exponents e_i for use with our implementation of SuperSPAR, assuming the range in which it is a competitive integer factoring tool.

The exponentiation stage of SuperSPAR computes $[\mathfrak{b}] = [\mathfrak{a}]^{2^\ell E}$ with $\ell = \left\lfloor \log_2 \sqrt{|\Delta|} \right\rfloor$. The search stage then computes baby steps $[\mathfrak{b}]^i$ for i coprime to some primorial P_w. However, if $\operatorname{ord}([\mathfrak{b}])$ and P_w have a common factor p, then p is not coprime to P_w and $[\mathfrak{b}]^p$ will not be added to the lookup table used by the coprime baby steps. If this is the case, the search phase is guaranteed to fail since it cannot find any $[\mathfrak{b}]^{2js}$ such that $2js \equiv 0 \pmod p$ in the lookup table. In order to ensure that the order of $[\mathfrak{b}]$ is coprime to E, the exponents in E would have to be chosen such that $e_i = \left\lfloor \log_{p_i} \sqrt{|\Delta|} \right\rfloor$, where $\sqrt{|\Delta|}$ is a bound on h_Δ (see Subsection 2.5.2). However, in practice it is more efficient to choose smaller values of e_i at the risk that additional class groups are tried. For this reason, the exponent E is chosen to remove, with high probability, the factors from the order of $[\mathfrak{a}]$ that are common with the primorial P_w used by the search stage. In this subsection, we assume that the largest prime p_t dividing E in the exponentiation stage is larger than the largest prime p_w diving P_w in the search stage. Here we are only concerned with determining bounds on e_i, the exponent of each prime factor of E.

For each ideal in our data set $\mathcal{D}$, let the factorization of the order be represented by $\operatorname{ord}([\mathfrak{a}]) = \prod {p_i}^{e_i}$ for p_i prime. For each prime power factor 3^{e_2}, 5^{e_3}, 7^{e_4}, 11^{e_5}, 13^{e_6}, and 17^{e_7}, more than 99% of the ideals studied had either $e_2 \le 4$, $e_3 \le 2$, $e_4 \le 2$, $e_5 \le 1$, $e_6 \le 1$, or

$e_7 \leq 1$. This is captured by Table 8.2. This is not to say that more than 99% of the ideals in the data set were such that $\text{ord}([\mathfrak{a}]^{3^4 5^2 7^2 11^1 13^1 17^1})$ is coprime to $3 \times 5 \times 7 \times 11 \times 13 \times 17$, but only that $\text{ord}([\mathfrak{a}]^{3^4})$ is coprime to 3, and so on.

n	$e_2 = 4$	$e_3 = 2$	$e_4 = 2$	$e_5 = 1$	$e_6 = 1$	$e_7 = 1$
32	0.99598	0.99195	0.99713	0.99177	0.99419	0.99659
40	0.99594	0.99193	0.99709	0.99166	0.99404	0.99660
48	0.99593	0.99193	0.99713	0.99162	0.99406	0.99655
56	0.99593	0.99194	0.99709	0.99170	0.99404	0.99648
64	0.99584	0.99211	0.99713	0.99170	0.99412	0.99652
72	0.99586	0.99201	0.99707	0.99163	0.99415	0.99650
80	0.99580	0.99190	0.99719	0.99169	0.99404	0.99645

Table 8.2: The probability that $\text{ord}([\mathfrak{a}]^{{p_i}^{e_i}})$ is coprime to p_i.

On the other hand, Table 8.3 represents the probability that $\text{ord}([\mathfrak{a}]) = \prod {p_i}^{e_i}$ has $e_i \leq B_i$ for $2 \leq i \leq 5$, $e_i = 1$ for all $i > 5$, and e_1 is left unbound. In other words, this table represents the probability that choosing some exponent E with the above constraints implies that the ideal class $[\mathfrak{b}] = [\mathfrak{a}]^E$ will have order $\text{ord}([\mathfrak{b}])$ coprime to the exponent E.

B_2	B_3	B_4	B_5	32 bits	40 bits	48 bits	56 bits	64 bits	72 bits	80 bits
1	1	1	1	0.40826	0.40693	0.40649	0.40598	0.40638	0.40624	0.40664
2	1	1	1	0.80574	0.80473	0.80484	0.80438	0.80500	0.80436	0.80502
2	2	1	1	0.90863	0.90739	0.90720	0.90718	0.90722	0.90714	0.90738
3	2	1	1	0.93187	0.93061	0.93071	0.93067	0.93057	0.93066	0.93051
3	2	2	1	0.94869	0.94735	0.94720	0.94724	0.94720	0.94734	0.94722
4	2	2	1	0.95669	0.95525	0.95511	0.95510	0.95518	0.95520	0.95499
4	2	2	2	0.96418	0.96256	0.96246	0.96238	0.96243	0.96255	0.96228
4	3	2	2	0.97062	0.96887	0.96871	0.96866	0.96858	0.96877	0.96856

Table 8.3: The probability that the order of an ideal class $[\mathfrak{b}] = [\mathfrak{a}]^E$ is coprime to the product $E = \prod {p_i}^{e_i}$ for primes p_i where $e_i \leq B_i$ for $2 \leq i \leq 5$; let $e_i = 1$ for all $i > 5$ and let e_1 be unbound.

While almost any bounds on e_i work in practice, the purpose of this chapter is to optimize the performance of our implementation of SuperSPAR. To this end, let B denote a prime power bound such that $e_i = \max\left\{\left\lfloor \log_{p_i} B \right\rfloor, 1\right\}$ for all odd primes $p_i \leq B$ and $e_i = 1$ for all primes $p_i > B$; let $e_1 = \left\lfloor \log_2 \sqrt{|\Delta|} \right\rfloor$. We then iterate the bound B over increasing values such that exactly one of the e_i increases by just 1. Table 8.4 shows the average time to split semiprime integers when the exponents e_i of the prime powers ${p_i}^{e_i}$ are bound accordingly. The largest prime p_t in the exponent E and the search stage bound mP_w are fixed for each input size, and were chosen based on the theoretically optimal values from Section 4.2. The sample set consists of 100,000 semiprime integers of each bit size, such that each integer was the product of two primes of equal size. Having determined bounds for the exponents e_i for the prime powers ${p_i}^{e_i}$, determining the exponent $E = \prod {p_i}^{e_i}$ for use with each bit range is simpler since it only involves bounding the largest prime p_t used. This is done in Section 8.6.

Power bound B	32 bits	40 bits	48 bits	56 bits	64 bits	72 bits	80 bits
$9 = 3^2$	0.04483	0.09538	0.24576	0.74423	2.13834	5.14941	11.81556
$25 = 5^2$	0.04469	0.09332	0.23505	0.70051	2.00290	4.82805	11.09770
$27 = 3^3$	$\underline{0.04453}$	$\underline{0.08980}$	0.22673	0.67292	1.90503	4.58491	10.51262
$49 = 7^2$	0.04550	0.09202	0.21968	0.65496	1.84011	4.42733	10.08651
$81 = 3^4$	0.04794	0.09219	$\underline{0.21840}$	0.64810	1.80854	4.32226	9.88501
$121 = 11^2$	0.04988	0.09499	0.22151	0.64957	1.79549	4.27838	9.81882
$125 = 5^3$	0.04861	0.09505	0.22084	0.64985	1.79609	4.28010	9.80192
$169 = 13^2$	0.05103	0.09783	0.22195	0.64568	1.76646	4.21067	9.60217
$243 = 3^5$	0.05126	0.09871	0.22307	$\underline{0.64527}$	$\underline{1.75882}$	4.17222	9.50603
$343 = 7^3$	0.05415	0.10083	0.22761	0.64718	1.76139	4.17266	9.47648
$625 = 5^4$	0.05966	0.10620	0.23601	0.65786	1.76363	4.13652	9.31960
$729 = 3^6$	0.06031	0.11030	0.23750	0.65964	1.76242	$\underline{4.12565}$	9.28392
$1331 = 11^3$	0.06028	0.10836	0.24391	0.66904	1.78810	4.13952	$\underline{9.28274}$
$2187 = 3^7$	0.06274	0.11096	0.25596	0.69223	1.82186	4.18447	9.34884

Table 8.4: Average time (in milliseconds) to factor semiprime integers using the exponent $E = \prod_{p_i > 2} {p_i}^{e_i}$ with $e_i = \lfloor \log_{p_i} B \rfloor$ when $p_i \le B$ and $e_i = 1$ otherwise. The lowest time for each integer range is underlined.

8.3.2 Difference between the Order of Two Ideal Classes

The purpose of this subsection is to justify a strategy where once h', a multiple of the order of an ideal class $[\mathfrak{a}_1]^{2^\ell E}$ for $[\mathfrak{a}_1] \in Cl_\Delta$, is known, our implementation then attempts to find an ambiguous ideal by repeated squaring (up to at most $\ell = \left\lfloor \log_2 \sqrt{|\Delta|} \right\rfloor$ times) of $[\mathfrak{a}_i]^{Eh'}$ for several other ideal classes $[\mathfrak{a}_i] \in Cl_\Delta$. If both stages of the algorithm fail to find h', then a different multiplier k for the discriminant of the class group is tried. Intuitively, the hope is that the factorization of the order of each ideal class $[\mathfrak{a}_i]$ consists of one large prime and several small primes, and that the large prime is the same among each ideal class. The exponentiation stage computes $[\mathfrak{b}_1] = [\mathfrak{a}_1]^{2^\ell E}$ for some exponent E chosen so that the factorization of the order of $[\mathfrak{b}_1]$ is likely to be free of small primes. In which case, either the search stage successfully computes h', which is hopefully also a multiple of the order of some other ideal class $[\mathfrak{a}_i]^{2^\ell E}$, or the search stage was unsuccessful in determining the order of $[\mathfrak{b}_1]$ because of the large prime factor dividing its order, and as such, the search stage will be unlikely to determine the order of some other ideal class $[\mathfrak{a}_i]^{2^\ell E}$.

Suppose the search stage of SuperSPAR determines h', a multiple of the order of an ideal class $[\mathfrak{b}_1] = [\mathfrak{a}_1]^{2^\ell E}$ for $\ell = \left\lfloor \log_2 \sqrt{|\Delta|} \right\rfloor$. Then for some other ideal class $[\mathfrak{a}_i]$ in the same group, there exists a set of primes $\mathcal{P}$ such that the order of $[\mathfrak{a}_i]^{2^\ell E}$ divides $h' \prod_{p \in \mathcal{P}} p$, where the product of the empty set is taken to be 1. Also suppose that an exponent E was chosen for the exponentiation stage such that $\operatorname{ord}([\mathfrak{a}]^E)$ is coprime to $2 \times 3 \times 5 \times 7 \times 11$. We chose 11 as the largest prime here since for over 99% of the ideal classes in our set $\mathcal{D}$, 11^2 does not divide the order (see Table 8.2). Also, notice that $2 \times 3 \times 5 \times 7 \times 11 \mid E$ but that E may be considerably larger than $2 \times 3 \times 5 \times 7 \times 11$.

n	$\lvert\mathcal{P}\rvert = 0$	$\lvert\mathcal{P}\rvert = 1$	$\lvert\mathcal{P}\rvert = 2$	$\lvert\mathcal{P}\rvert = 3$
32	0.97804	0.02177	0.00020	0.00000
40	0.97791	0.02183	0.00026	0.00000
48	0.97785	0.02187	0.00028	0.00000
56	0.97776	0.02196	0.00028	0.00000
64	0.97779	0.02193	0.00028	0.00000
72	0.97782	0.02190	0.00027	0.00000
80	0.97781	0.02191	0.00028	0.00000

Table 8.5: If h', a multiple of the order of an ideal class $[\mathfrak{u}]$ is known, this table shows the probability that the order of the ideal class $[\mathfrak{v}]^{h'}$, for some other ideal class $[\mathfrak{v}]$ in the same group, has exactly $|\mathcal{P}|$ prime factors larger than 11.

Assuming the above constraints, for two ideal classes from the same group $[\mathfrak{u}], [\mathfrak{v}] \in Cl_\Delta$, Table 8.5 shows the probability that the smallest set of primes $\mathcal{P} = \{p \,:\, p \text{ is prime}, p > 11\}$

with $\mathrm{ord}([\mathfrak{v}]^{2^\ell E})$ dividing $\mathrm{ord}([\mathfrak{u}]^{2^\ell E}) \times \prod_{p \in \mathcal{P}} p$ is of a given size. Of the ideal class groups studied and assuming the above constraints, more than 97.7% of the time, if $2^\ell E h'$ is a multiple of the order of an ideal class, then it is also a multiple of the order of some other ideal class in the same group. For SuperSPAR, this means that once a multiple of the order of an ideal class is discovered, if this does not lead to a splitting of N, then with high probability, it is also a multiple of the order of some other ideal class in the same group. Rather than starting over with a different ideal class, let h' be the odd part of the multiple of the order and compute $[\mathfrak{a}_2]^{h'}$ for some other ideal class $[\mathfrak{a}_2] \in Cl_\Delta$. SuperSPAR then tries to find an ambiguous class by repeated squaring of $[\mathfrak{a}_2]^{h'}$ and, if successful, attempts to split N. This process may be repeated for several different ideal classes in the same group. However, if the algorithm is unsuccessful for more than an fixed number of ideal classes within the same class group, Subsection 8.3.4 shows that changing class groups by changing multipliers is beneficial.

8.3.3 Best Multipliers

As in the previous subsection, assume that an exponent E is chosen for the exponentiation stage such that $\mathrm{ord}([\mathfrak{a}_1]^{2^\ell E})$ is coprime to E. Also, suppose that the search stage is successful in determining h', a multiple of the order of $[\mathfrak{a}_1]^{2^\ell E} \in Cl_\Delta$. Our implementation of SuperSPAR will first attempt to split the integer N associated with the discriminant Δ by searching for an ambiguous class via repeated squaring of the ideal class $[\mathfrak{a}_1]^{Eh'}$. Failing this, the algorithm tries again by repeated squaring of the ideal classes $[\mathfrak{a}_2]^{Eh'}, [\mathfrak{a}_3]^{Eh'}, ...$ and so on in sequence. This subsection uses the data set $\mathcal{D}$ to determine the expected number of ideal classes to try before a successful splitting of the integer N.

The discriminant Δ associated with an integer N and multiplier k is chosen to be either $\Delta = -kN$ when $-kN \equiv 0, 1 \pmod 4$, or $\Delta = -4kN$. As such, the data set $\mathcal{D}$ is separated into subsets for $N \equiv 1 \pmod 4$ or $N \equiv 3 \pmod 4$, and again for each multiplier $k \in \{1, 2, 3, 5, 6, 7, 10\}$. Figure 8.3 shows the expected number of ideal classes to try before splitting N, assuming that an appropriate exponent E for the exponentiation stage is chosen and that a multiple of the order h' was found during the search stage.

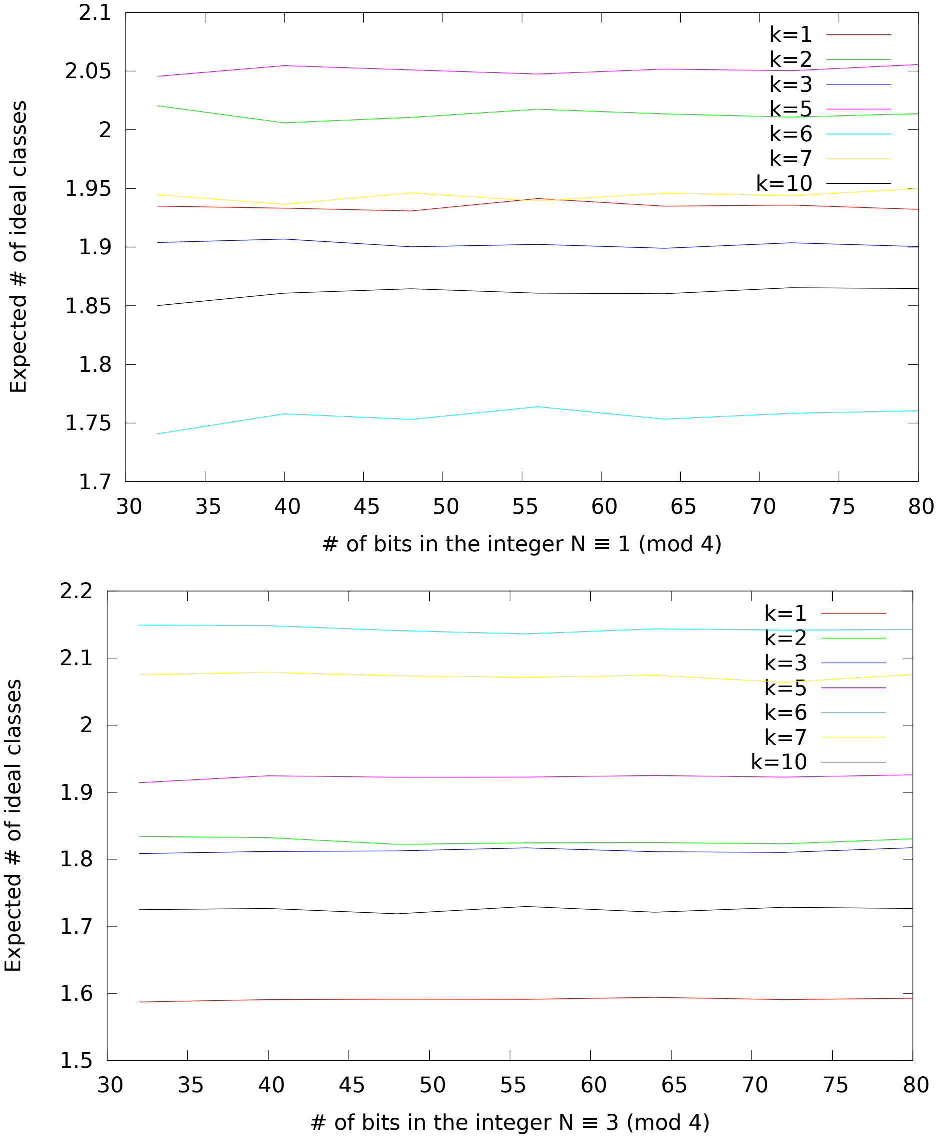

Figure 8.3: Attempts $N \equiv 1 \pmod 4$)

Given this separation of the integer N and the multiplier k in our data set $\mathcal{D}$, the class groups associated with discriminants $\Delta = -kN$ or $\Delta = -4kN$, as appropriate, appear to be separate with respect to the probability that an ideal class in the group will lead to a splitting of the integer N. As such, our implementation of SuperSPAR chooses multipliers of N sequentially such that when $N \equiv 1 \pmod 4$, $k = 6, 10, 3, 1, 7, 2, 5$, and when $N \equiv 3$

(mod 4), $k = 1, 10, 3, 2, 5, 7, 6$. In the event that the algorithm exhausts this list of multipliers without successfully splitting N, square free multipliers $k > 10$ are selected in increasing order.

8.3.4 Expected Number of Ideal Classes

Figure 8.3 indicates that for most multipliers, if the search stage of the algorithm is successful in finding a multiple of the order of an ideal class, then in expectation, roughly 2 ideal classes need to be tried before a successful splitting of the integer N. Although switching multipliers after at most two ideal classes does work in practice, our implementation of SuperSPAR is more efficient if many ideal classes are tried before switching class groups. This is possibly due to the cost of the search phase of the algorithm, which is executed for each class group, in contrast to the exponentiation stage, which is executed for each ideal class tried.

Classes/Group	32 bits	40 bits	48 bits	56 bits	64 bits	72 bits	80 bits
1	0.05570	0.12454	0.31630	0.92448	2.56007	6.55736	14.64756
2	0.04462	0.09425	0.23631	0.68485	1.92037	4.88625	11.07386
3	0.04181	0.08708	0.21465	0.62268	1.74317	4.39471	10.01309
4	0.04104	0.08418	0.20621	0.59624	1.66741	4.19481	9.55292
5	0.04080	0.08306	0.20254	0.58463	1.63373	4.09556	9.31992
6	0.04081	0.08270	0.20113	0.57836	1.61780	4.03869	9.18897
7	0.04098	0.08262	0.20026	0.57618	1.60919	4.01559	9.09742
8	0.04113	0.08266	0.20008	0.57406	1.60498	3.99565	9.05229
9	0.04130	0.08284	0.20005	0.57365	1.60082	3.98321	9.02378
10	0.04146	0.08311	0.20028	0.57390	1.60011	3.97691	9.00030
11	0.04175	0.08327	0.20067	0.57404	1.59738	3.97021	8.98048
12	0.04180	0.08348	0.20092	0.57470	1.59855	3.96747	8.97534
13	0.04200	0.08367	0.20129	0.57520	1.59944	3.96789	8.97226
14	0.04221	0.08391	0.20161	0.57583	1.60174	3.96836	8.97280
15	0.04231	0.08407	0.20208	0.57680	1.60335	3.97339	8.97848
16	0.04251	0.08431	0.20248	0.57780	1.60844	3.97845	8.97971

Table 8.6: The average time (in milliseconds) to split an integer N using no more than a fixed number of ideal classes for each class group. The lowest time for each integer range is underlined.

Table 8.6 shows the average time to split integers given an upper bound on the number of ideal classes tried before switching multipliers. The times here are for the same set of 100,000 semiprime integers as in Table 8.4. Again, the largest prime in the exponent for the exponentiation stage, and the multiple of a primorial for the search stage where chosen according to the theoretically optimal values of Section 4.2. Our implementation of SuperSPAR bounds the number of ideal classes tried before switching class groups to the lowest time for each integer range in the table given. For integer sizes not appearing in the table, they are rounded up to the nearest value appearing in the table, or for integer sizes greater than 80, the are linearly extrapolated by 1 class/group for each 8 bits.

8.4 SuperSPAR Algorithm in Practice

This section describes and gives pseudo-code for our implementation of SuperSPAR. We includes this here as our implementation makes use of the results of the previous sections.

Section 8.1 gives an algorithm to compute a list of coprime deltas for a primorial P_w. Since the baby-step portion of the search phase steps coprime up to a multiple of some primorial, we precompute deltas between neighbouring coprime values. Table 8.1 shows the rate of growth of the first 12 primorials, and the number of integers coprime to each primorial. The 10^{th} primorial has over a billion coprime values less than P_{10}, while the 11^{th} primorial has over 30 billion coprime values less than P_{11}. For this reason, we limit our precomputation of the delta lists to the first 10 primorials. These lists were generated using Algorithm 8.1 and then made available as global data to our implementation.

Section 8.2 discusses a method to simultaneously hash an ideal class and its inverse. Briefly, our implementation from Chapter 6 represents an ideal class using the triple (a, b, c). The inverse is given as $(a, -b, c)$. Since the value of $|b|$ can be determined from the values a, c, and Δ, we compute a hash using only the values a and c. Since two ideals may still hash to the same value, we use open addressing (see Subsection 8.2.2) for collision resolution.

Our implementation of SuperSPAR uses many results from Section 8.3. The prime power bounds e_i for the exponent $E = \prod {p_i}^{e_i}$ in the exponentiation stage are given in Subsection

8.3.1 and in particular, Table 8.4. If either stage determines a multiple of the order of an ideal class, but this does not lead to a splitting of the input integer N, then our implementation reuses the multiple of the order for several ideal classes. However, if either stage fail to determine a multiple of the order of an ideal class, the algorithm tries again using a different square free multiplier. This strategy is justified in Subsection 8.3.2. In addition, our implementation uses the results from Subsection 8.3.4 to limit the number of ideal classes tried before switching multipliers, specifically, we use the results of Table 8.6. Lastly, Subsection 8.3.3 indicates that for the set of ideals tested, the use of some multipliers are more likely to be successful at splitting the input integer. As such, the list of square free multipliers provided to our implementation is ordered according to these results.

The pseudo-code for our implementation of SuperSPAR is separated into three listings. Algorithm 8.2 attempts to find an ambiguous ideal class by repeated squaring of an input ideal class up to at most ℓ times. Bounding the number of times we square is necessary since the order of the input ideal class could be odd. If the algorithm finds an ambiguous ideal, it attempts to split the input integer N. Details are given in Subsection 4.1.1, but we briefly restate them here. For an ideal class representative $\mathfrak{a} = [a, (b + \sqrt{\Delta})/2]$, let $c = (b^2 - \Delta)/4a$. An ideal class is an ambiguous class when either $b = 0$, $a = b$, or $a = c$ (see [45, p.303]). Since the discriminant is $\Delta = b^2 - 4ac$, given an ambiguous form, either $\Delta = 4ac$ when $b = 0$, $\Delta = b(b - 4c)$ when $a = b$, or $\Delta = (b - 2a)(b + 2a)$ when $a = c$. Once an ambiguous class is found, we then compute $d = \gcd(a, N)$ if $b = 0$ or $a = b$, and $d = \gcd(b - 2a, N)$ otherwise.

Once a multiple of the order of an ideal class is known, Algorithm 8.3 is used to try to find an ambiguous ideal class from at most c ideal classes in the same class group. If the algorithm is successful, a non-trivial factor of the input integer N is returned, otherwise 1 is returned to indicate failure. This algorithm makes use of Algorithm 8.2 as a sub-algorithm and is itself a sub-algorithm of the main algorithm given by Algorithm 8.4.

Algorithm 8.4 is the main factoring algorithm. It takes as input an odd composite positive

Algorithm 8.2 Try to find an ambiguous ideal class by repeated squaring (Subsection 4.1.1).

```
procedure RepeatedlySquare($[\mathfrak{b}], \ell$)
    for $i$ from 1 to $\ell$ while $[\mathfrak{b}] \neq [\mathcal{O}_\Delta]$ do
        if $[\mathfrak{b}]$ is an ambiguous ideal class then
            if $[\mathfrak{b}]$ leads to a non-trivial factor of $N$ then
                $d \leftarrow$ a non-trivial factor of $N$
                return ($[\mathfrak{b}]$, $d$)
            else
                return ($[\mathfrak{b}]$, 1)
        $[\mathfrak{b}] \leftarrow [\mathfrak{b}]^2$
    return ($[\mathfrak{b}]$, 1)
```

Algorithm 8.3 Try to factor N using several ideal classes and a multiple of the order of a single ideal class.

```
procedure TryManyIdealClasses($\Delta$, $h$, $\ell$, $c$)
    for $2 \leq i \leq c$ do
        let $[\mathfrak{p}_i]$ be the $i^{\text{th}}$ smallest prime ideal class in $Cl_\Delta$
        ($[\mathfrak{b}]$, $d$) $\leftarrow$ RepeatedlySquare($[\mathfrak{p}_i]^h$, $\ell$)      {$[\mathfrak{b}]$ is ignored}
        if $d \neq 1$ then return d
    return 1
```

integer N, and the configuration parameters. The configuration parameters are determined by the size of the input integer N and are looked up from a skeleton function that invokes the main algorithm. The previous sections in this chapter constrain the majority of the configuration parameters with the exception of the largest prime p_t used in the exponent $E = \prod_{i=2}^{t} {p_i}^{e_i}$ for the exponentiation stage, and the multiple of a primorial mP_w used for the search stage. We give pseudo-code for our implementation of SuperSPAR here, since these two parameters are empirically determined in Section 8.6 by running the main algorithm on sets of semiprime integers for various values of E and mP_w, by using Algorithm **??** to search for a local minimum in 2 dimensions.

The main algorithm first computes $\ell = \left\lfloor \log_2 \sqrt{|\Delta|} \right\rfloor$ where $\sqrt{|\Delta|}$ is a bound on the class number given in Subsection 2.5.2, and then sets the square free multiplier index k to 1. The algorithm then computes a discriminant Δ for N and the multiplier corresponding to the index k, and then begins the exponentiation stage.

The exponentiation stage first finds an ideal class $[\mathfrak{p}_1] \in Cl_\Delta$ such that $\mathfrak{p}_1 = [p, (b+\sqrt{\Delta})/2]$ for the smallest prime p with $\gcd(p, \Delta) = 1$ and $4p \mid b^2 - \Delta$. Next, the algorithm computes $[\mathfrak{b}'] \leftarrow [\mathfrak{p}_1]^E$. In our implementation, E is given as a 2,3 representation precomputed using the L best approximations method of Section 7.6. As such, we use Algorithm 3.3 to perform the exponentiation. The last step of the exponentiation stage is to use Algorithm 8.2 to repeatedly square $[\mathfrak{b}']$, test for an ambiguous form, and attempt to factor the input integer N. Failing this, the exponentiation stage has computed $[\mathfrak{b}] = [\mathfrak{p}_1]^{2^\ell E}$ and the algorithm continues with the search stage.

The search stage begins by precomputing each $[\mathfrak{b}]^{\delta_i}$ and caches these in $[\mathfrak{d}_{\delta_i}]$. These are used to move from one coprime baby-step to the next. Only even deltas are computed since only odd exponents are computed by the baby-steps. Lines 12 through 16 compute baby-steps for $[\mathfrak{b}]^i$ with $1 \le i \le s$ and $\gcd(i, P_w) = 1$, but does so by utilizing δ_i and the cached steps $[\mathfrak{d}_{\delta_i}]$ If any $[\mathfrak{b}]^i = [\mathcal{O}_\Delta]$, then the algorithms records i as the order of $[\mathfrak{b}]$ and jumps to line 28, where we attempt to find an ambiguous ideal and then factor N. The loop exits after the last baby step with $i = s+1$, at which point, line 17 effectively computes $[\mathfrak{c}] = [\mathfrak{b}]^{2s}$ where $[\mathfrak{c}]$ is the first giant step. We choose to compute this by multiplying $[\mathfrak{c}]$ with the inverse of $[\mathfrak{b}]$ and then squaring. An alternative method is to not compute the last baby step such that $i < s$, however, this requires us to then compute $[\mathfrak{c}] \leftarrow [\mathfrak{c}] \cdot [\mathfrak{b}]^{s-i}$ where $s - i$ is guaranteed to be odd. To speed this, we can compute $[\mathfrak{c}] \leftarrow [\mathfrak{c}] \cdot [\mathfrak{b}]^{s-i-1} \cdot [\mathfrak{b}]$ since $[\mathfrak{b}]^{s-i-1}$ is even and will be cached by $[\mathfrak{d}_{s-i-1}]$, however, this is the same number of operations (not counting inversions, which are essentially free) as the method employed in our implementation. Next, the algorithm performs an equal number of giant steps by computing $[\mathfrak{c}] \leftarrow [\mathfrak{c}] \cdot [\mathfrak{d}]$ for each iteration where $[\mathfrak{d}] = [\mathfrak{b}]^{2s}$ from line 18. If any $[\mathfrak{c}]$ is in the table, then either $2js - i$ or $2js + i$ is a multiple of the order of $[\mathfrak{b}]$ and we continue at line 25 where we attempt to find an ambiguous ideal class for both values. If either stage fails, we start over with the next square free multiplier from the list.

Algorithm 8.4 SuperSPAR Integer Factoring Algorithm.

```
Input: $N \in \mathbb{Z}_{\geq 0}$ odd and composite, $E = \prod_{i=2}^{t} {p_i}^{e_i}$ for the exponentiation phase, $s = mP_w$
    a multiple of a primorial for the search phase and $\phi(P_w)$, $c \in \mathbb{Z}_{>0}$ the number of ideal
    classes to try before switching multipliers, a coprime delta list $\mathcal{D}_w = \delta_1, \delta_2, ..., \delta_{\phi(P_w)}$ for
    the primorial $P_w$, $\delta_{\max} = \max \mathcal{D}_w$, and square free multipliers $\kappa_1, \kappa_2, ....$
 1: $\ell = \left\lfloor \log_2 \sqrt{|\Delta|} \right\rfloor$                              {bound for repeated squaring}
 2: $k \leftarrow 1$                                              {square free multiplier index}
 3: $\Delta \leftarrow -\kappa_k N$
 4: if $\Delta \not\equiv 0, 1 \pmod 4$ then $\Delta \leftarrow 4\Delta$
    {– exponentiation stage –}
 5: let $[\mathfrak{p}_1]$ be the smallest prime ideal class in $Cl_\Delta$
 6: $[\mathfrak{b}'] \leftarrow [\mathfrak{p}_1]^E$
 7: $([\mathfrak{b}], d) \leftarrow$ RepeatedlySquare($[\mathfrak{b}']$, $\ell$)
 8: if $d \neq 1$ then return d
 9: if $[\mathfrak{b}]$ is an ambiguous ideal class then $h \leftarrow 1$, and go to line 30
    {– search stage –}
10: compute $[\mathfrak{d}_2] = [\mathfrak{b}]^2$ and $[\mathfrak{d}_{2i}] = \left[\mathfrak{d}_{2(i-1)}\right] \cdot [\mathfrak{d}_2]$ for $2 \leq i \leq \delta_{\max}/2$
11: $i \leftarrow 1, j \leftarrow 1, [\mathfrak{c}] \leftarrow [\mathfrak{b}]$
12: while $i \leq s$ do                                                {coprime baby-steps}
13:     store $[\mathfrak{c}] \mapsto i$ in the lookup table
14:     if $[\mathfrak{c}] = [\mathcal{O}_\Delta]$ then $h \leftarrow i$, and go to line 28
15:     $[\mathfrak{c}] \leftarrow [\mathfrak{c}] \cdot \left[\mathfrak{d}_{\delta_j}\right], i \leftarrow i + \delta_j$
16:     $j \leftarrow (j + 1) \bmod \phi(P_w)$
17: $[\mathfrak{c}] \leftarrow ([\mathfrak{c}] \cdot [\mathfrak{b}]^{-1})^2$                {last baby-step was $s + 1$, first giant-steps is $2s$}
18: $[\mathfrak{d}] \leftarrow [\mathfrak{c}]$
19: for $1 \leq j \leq m\phi(P_w)$ do                                                  {giant-steps}
20:     if $[\mathfrak{c}]$ is in the table, first lookup $i$ then
21:         $h \leftarrow$ odd part of $(2js - i)$, $h_2 \leftarrow$ odd part of $(2js + i)$
22:         go to line 25
23:     $[\mathfrak{c}] \leftarrow [\mathfrak{c}] \cdot [\mathfrak{d}]$
24: go to line 32                                     {failed to find an ambiguous class}
    {– found a multiple of the order –}
25: $([\mathfrak{b}], d) \leftarrow$ RepeatedlySquare($[\mathfrak{b}']^{h_2}$, $\ell$)
26: if $d \neq 1$ then return $d$
27: if $[\mathfrak{b}] = [\mathcal{O}_\Delta]$ then $h \leftarrow h_2$, and go to line 30.
28: $([\mathfrak{b}], d) \leftarrow$ RepeatedlySquare($[\mathfrak{b}']^h$, $\ell$)                          {$[\mathfrak{b}]$ is ignored}
29: if $d \neq 1$ then return $d$
30: $d \leftarrow$ TryManyIdealClasses($\Delta$, $E \cdot h$, $\ell$, $c$)
31: if $d \neq 1$ then return $d$
32: $k \leftarrow k + 1$
33: start over at line 3
```

Before we move on, we reiterate some of the results from previous chapters that are used in our implementation of SuperSPAR. Since SuperSPAR is based on ideal class arithmetic, we use the results from Chapter 6. In particular, when the discriminant Δ is less than 60-bits, we use our 64-bit implementation of ideal class arithmetic, otherwise, when Δ is less than 119-bits, we use our 128-bit implementation, and when Δ is larger than 118-bits, we use our reference implementation based on GMP for multiple precision arithmetic.

Again, our 64-bit and 128-bit implementations of ideal class arithmetic rely heavily on the results of the extended GCD experiments in Chapter 7. In particular, when the inputs are guaranteed to be less than 32-bits, we use our 32-bit implementation of the extended Euclidean Algorithm (see Section 5.1). Otherwise, when inputs are guaranteed to be less than 64-bits, we use our 64-bit implementation of our simplified left-to-right binary extended GCD (see Section 5.3). Failing this, we use our 128-bit implementation our simplified left-to-right binary extended GCD for inputs less than 118-bits, and our 128-bit implementation of Lehmer's extended GCD using our simplified left-to-right binary extended GCD for 64-bit machine words (see Section 5.4) when inputs are greater than 119-bits. These methods were selected since Section 5.7 shows that each method performs the fastest on average for their given input range. For each of these extended GCD methods, we also implement a left-only and partial version and use them as applicable for ideal class arithmetic. For larger inputs, we use GMP for the extended GCD and left-only extended GCD, and we implement our own partial extended Euclidean GCD using GMP arithmetic.

8.5 Search Bounds

This section introduces the method we use to generate a list of candidate search bounds. Each search bound is a multiple of some primorial mP_w such that the search stage of SuperSPAR takes $m\phi(P_w)$ steps for values coprime up to mP_w, and then takes another $m\phi(P_w)$ giant steps up to $2m^2\phi(P_w)P_w$. The idea is to generate a list of pairs (m, w) such that for each pair in the list, there does not exist a pair (m', w') with a larger final giant step requiring the same or fewer total number of steps. This list is useful since in Section 8.6, we empirically determine search bounds that work well on average for inputs of a given size. These empirically determined search bounds use a different number of total steps than the theoretically optimal number predicted in Section 4.2. This list may also be useful for other algorithms deriving from the bounded primorial steps algorithm (see Subsection 4.2.1) where the search bound is determined empirically. We describe the technique here as it does not appear in the literature.

The search stage will successfully compute the order of an ideal class if the order is no larger than the largest exponent $2m^2\phi(P_w)P_w$ generated by the giant steps – we will refer to this as the *search range*. The idea here is to generate a set $\mathcal{S}$ of primorial and multiplier pairs used as candidates for the baby step bound s. For each pair, let δ be a function from some multiple m of the primorial P_w to the total number of baby steps and giant steps taken during the search stage of the algorithm,

$$\delta(m, w) = 2m\phi(P_w). \tag{8.1}$$

Notice we take $m\phi(P_w)$ baby steps plus $m\phi(P_w)$ giant steps. Each giant step is of size $2mP_w$, so let θ be a function that computes the exponent of the final giant step, i.e. the search range,

$$\theta(m, w) = 2m^2\phi(P_w)P_w. \tag{8.2}$$

For a given pair (m, w) representing a search bound, the total number of steps taken and the search range are unique. This idea is more formally expressed by the following theorem.

Theorem 8.5.1. For the pair (m_i, w_i), there does not exist a pair (m_j, w_j) with $m_i \neq m_j$ or $w_i \neq w_j$ such that both $\delta(m_i, w_i) = \delta(m_j, w_j)$ and $\theta(m_i, w_i) = \theta(m_j, w_j)$.

Proof. By contradiction, suppose there exists (m_i, w_i) and (m_j, w_j) with $m_i \neq m_j$ or $w_i \neq w_j$ such that $\delta(m_i, w_i) = \delta(m_j, w_j)$ and $\theta(m_i, w_i) = \theta(m_j, w_j)$. Let $w_i < w_j$. Then by Equation

8.1

$$
\begin{aligned}
2m_i\phi(P_{w_i}) &= 2m_j\phi(P_{w_j})\\
\Rightarrow \frac{m_i}{m_j} &= \frac{\phi(P_{w_j})}{\phi(P_{w_i})}
\end{aligned}
$$

and by Equation 8.2

$$
\begin{aligned}
&2{m_i}^2\phi(P_{w_i})P_{w_i} = 2{m_j}^2\phi(P_{w_j})P_{w_j}\\
\Rightarrow &2m_j\phi(P_{w_j})m_iP_{w_i} = 2{m_j}^2\phi(P_{w_j})P_{w_j} && \{\text{Substituting from above.}\}\\
\Rightarrow &m_iP_{w_i} = m_jP_{w_j}\\
\Rightarrow &\frac{m_i}{m_j} = \frac{P_{w_j}}{P_{w_i}}.
\end{aligned}
$$

Therefore

$$\frac{\phi(P_{w_j})}{\phi(P_{w_i})} = \frac{P_{w_j}}{P_{w_i}},$$

and this expands to

$$(p_{w_j}-1)(p_{w_{j-1}}-1)\cdots(p_{w_{i+1}}-1) = p_{w_j}p_{w_{j-1}}\cdots p_{w_{i+1}}.$$

Since each term on the left hand side is 1 less than the corresponding term on the right hand side, this is only possible when $w_i = w_j$ and then for $\delta(m_i, w_i) = \delta(m_j, w_j)$ and $\theta(m_i, w_i) = \theta(m_j, w_j)$ to be true, $m_i = m_j$ must also be true. □

With this out of the way, the set $\mathcal{S}$ consists of pairs (m, w) such that for all (m', w') either

$$\theta(m', w') < \theta(m, w) \text{ or } \delta(m', w') > \delta(m, w).$$

In other words, the set $\mathcal{S}$ consists of the pairs (m, w) such that for all other pairs (m', w'), either the search range $\theta(m', w')$ is not as large, or the total number of steps $\delta(m', w')$ is greater. The set $\mathcal{S}$ is defined as

$$\mathcal{S} = \{(m, w) : \text{for all } (m', w'),\ \text{either } \theta(m', w') < \theta(m, w) \text{ or } \delta(m', w') > \delta(m, w)\} \tag{8.3}$$

and is infinite. When a bound B on the total number of steps $\delta(m, w)$ is given, a subset $\mathcal{S}_B \subset \mathcal{S}$ is defined as

$$\mathcal{S}_B = \{(m, w) : (m, w) \in \mathcal{S} \text{ and } \delta(m, w) \leq B\} \tag{8.4}$$

and Algorithm 8.5 can be used to generate such a set.

Algorithm 8.5 Compute baby step bound candidates.

```
Input: A bound $B$ on the total number of steps $\delta(m, w)$.
Output: The set of candidate pairs $\mathcal{S}$ (Equation 8.3).
generate the set $\mathcal{T} = \{(m, w) \ : \ \delta(m, w) \leq B\}$
define an ordering on the set $\mathcal{T}$ using
    $(m_i, w_i) < (m_j, w_j) \Leftrightarrow \begin{cases} \delta(m_i, w_i) < \delta(m_j, w_j) \\ \delta(m_i, w_i) = \delta(m_j, w_j) \text{ and } \theta(m_i, w_i) > \theta(m_j, w_j). \end{cases}$
let $(m_1, w_1), (m_2, w_2), ..., (m_n, w_n)$ be the list of elements generated by sorting the set $\mathcal{T}$
    in ascending order
$\mathcal{S}_B \leftarrow \{\}$
$(m, w) \leftarrow (m_1, w_1)$
for $i = 2$ to $n$ do
    if $\theta(m_i, w_i) > \theta(m, w)$ then
        $\mathcal{S}_B \leftarrow \mathcal{S}_B \cup \{(m, w)\}$
        $(m, w) \leftarrow (m_i, w_i)$
$\mathcal{S}_B \leftarrow \mathcal{S}_B \cup \{(m, w)\}$
return $\mathcal{S}_B$
```

First the algorithm generates all pairs (m, w) such that $\delta(m, w) \leq B$. There is a finite number of these pairs, since there are a finite number of primorials no larger than B, and for a given w there are $\lfloor B/2\phi(P_w)\rfloor$ possible values of m. The algorithm then sorts these pairs in ascending order by the total number of steps $\delta(m, w)$, breaking ties by sorting in descending order of the search range $\theta(m, w)$. The first pair in the list is chosen as a candidate (m, w) for the set $\mathcal{S}_B$. This pair is always in the solution set since by Equation 8.3 and the ordering imposed on the list, all other pairs either take more steps, or an equal number of steps, but have a smaller search range. The algorithm then iterates over the remaining pairs, and the search range of each pair is compared with that of the candidate pair. If the search range is greater, then the candidate pair is output to the set $\mathcal{S}_B$ and the current pair becomes the next candidate pair. The correctness of this follows from the fact that pairs in the list are monotonically increasing by the total number of steps, and successive candidate pairs are strictly increasing in their search range. As such, when the candidate pair (m, w) is compared with the pair (m_i, w_i) from the list, if the search range of the list pair is less than that of the candidate pair, the the list pair uses at least as many steps, but does not search as far, and therefore is not a member of the set $\mathcal{S}_B$. Otherwise, when the list pair has a larger search range, this implies that it uses more steps (since ties are broken by sorting in descending order of search range), and the candidate pair is guaranteed to be a member of the set $\mathcal{S}_B$. This holds since all the pairs remaining in the list use more steps, and all the previous pairs from the list either had a smaller search range, or were rejected because they required more steps.

As an example, the set $\mathcal{S}_{512}$ restricted to pairs using no more than 512 steps, and ordered according to Algorithm 8.5, is

$$\begin{aligned}\mathcal{S}_{512} = \{&(1,1),(1,2),(3,1),(2,2),(5,1),(3,2),(1,3),(5,2),(6,2),(7,2),(2,3),(9,2),\\&(10,2),(11,2),(3,3),(14,2),(15,2),(4,3),(18,2),(19,2),(5,3),(23,2),(1,4),\\&(7,3),(8,3),(9,3),(10,3),(11,3),(2,4),(13,3),(14,3),(15,3),(16,3),(17,3),\\&(3,4),(20,3),(21,3),(22,3),(23,3),(4,4),(26,3),(27,3),(28,3),(29,3),(5,4)\}.\end{aligned}$$

The number of steps for each corresponding pair is

$$\begin{aligned}&2,4,6,8,10,12,16,20,24,28,32,36,40,44,48,56,60,64,72,76,80,92,96,112,128,144,\\&160,176,192,208,224,240,256,272,288,320,336,352,368,384,416,432,448,464,480,\end{aligned}$$

and the search range, i.e. the value of the largest exponent generated by the giant steps, for each corresponding pair is

$$4, 24, 36, 96, 100, 216, 480, 600, 864, 1176, 1920, 1944, 2400, 2904, 4320, 4704, 5400, 7680, 7776, 8664, 12000, 12696, 20160, 23520, 30720, 38880, 48000, 58080, 80640, 81120, 94080, 108000, 122880, 138720, 181440, 192000, 211680, 232320, 253920, 322560, 324480, 349920, 376320, 403680, 504000.$$

Table 8.7 provides several values predicted by the theory in Section 4.2 in combination with the results of this section for integers of various sizes. In theory, SuperSPAR takes the number of baby steps and giant steps proportional to the largest prime, p_t, used in the exponent of the exponentiation stage, such that $p_t \leq N^{1/2r}$, p_t is as large as possible, and $r = \sqrt{\ln N / \ln \ln N}$. In the table, the total number of steps, $2m\phi(P_w)$, is the smallest value greater than or equal to $\lfloor N^{1/2r} \rfloor$. The table also shows the search range given by the specified multiple m and primorial P_w. Since we did not expect our implementation of SuperSPAR to be competitive for integers larger than 100-bits, we point out that for integers 112-bits in size, the theory predicts that the search stage should require around 9802 steps. For this reason, in Section 8.6, we restrict the search bound list so that each candidate takes no more than 16384 steps. We choose 16384 since it is larger than 9802 and provides some padding. Another thing of interest is that even for integers of size 128-bits, the largest exponent generated during the search stage, as predicted by the theory, is 1173110400, which fits within 31-bits. This justifies our use of 32-bit values in the lookup table described in Section 8.2.

This technique of generating candidate bounds for the search phase of SuperSPAR is used in the next section where we search for an exponent to use in the exponentiation stage and a corresponding bound for the search stage that perform well in practice for integers of various sizes.

$\log_2 N$	$r = \sqrt{\ln N / \ln \ln N}$	$\lfloor N^{1/2r} \rfloor$	$2m\phi(P_w)$	$m^2\phi(P_w)P_w$	m	w
16	2.14693	13	16	480	1	3
32	2.67523	63	64	7680	4	3
48	3.08112	221	224	94080	14	3
64	3.42016	655	656	806880	41	3
80	3.71609	1738	1824	7277760	19	4
96	3.98139	4258	4320	40824000	45	4
112	4.22355	9802	10080	222264000	105	4
128	4.44745	21473	22080	1173110400	23	5

Table 8.7: The theoretically optimal number of group operations for each stage is given as $O(N^{1/2r})$. A search bound mP_w is selected so that the total number of steps, $2m\phi(P_w)$, is the smallest integer greater than or equal to the theoretically optimal value. The search range, $m^2\phi(P_w)P_w$, is given for the corresponding search bound.

8.6 Empirical Search for Exponent and Step Count

In order to split integers of a fixed size, two parameters remain to be determined for our implementation of SuperSPAR. The first is the largest prime used in the exponent $E = \prod {p_i}^{e_i}$ for the exponentiation stage, and the second is a multiple of a primorial, mP_w, for use with the search stage. This section begins by displaying the average time to factor semiprime integers of a given size when the largest prime in the exponent for the exponentiation stage or the bound on the search stage is iterated. The purpose of the first part is not to find parameters that work well on average, but only to visual the parameter search space. In Subsection 8.6.1, we propose using a 2 dimensional ternary-search in order to find values that perform well on average.

In selecting the exponent E for the exponentiation stage, our implementation uses the results of Chapter 7. Section 7.6 introduces the L best approximations technique for generating 2,3 representations of an integer, and the results in Section 7.10 show that this method performs best in practice when exponentiating an ideal class to the product of many prime numbers. This is the method used in our implementation of SuperSPAR. In practice, for a given exponent, the bound L was chosen to be $L = 2^k$ for some value k. The bound was repeatedly doubled and an L best approximation was computed for a given exponent until the cost of exponentiating an ideal class to that particular exponent did not change between iterations. For the exponents tested in this section, the bound was $L \leq 2048$. Representations for the exponents used here were precomputed and available in memory to the running application.

When selecting a multiple of a primorial for the search stage, candidates were selected from the list generated by Algorithm 8.5 with a bound of 16384 total steps. Since we did not expect our implementation of SuperSPAR to be competitive for integers larger than 100-bits, 16384 provides some padding over the theoretically optimal number of steps for integers this size, i.e. the search stage for 100-bit integers requires roughly 9802 steps (see Table 8.7).

To indicate the effect of iterating the largest prime in the exponent E and the search bounds $(m, w) \in S_{16384}$, three sets of 1000 random semiprime integers $N = p \cdot q$ were generated for p and q prime and half the size of N. The three sets consist of integers of 48-bits, 56-bits, and 64-bits. For 64-bit integers, the theory predicts that the largest prime, p_t, should be approximately 655 (see Table 8.7), and that the total number of steps in the search phase should be roughly the same. We then performed a 2 dimensional iteration of both the largest prime used in the exponent E for the exponentiation stage and the search bound $(m, w) \in S_{16384}$ over the theoretically optimal values. This process was repeated for each set of 48-bit and 56-bit semiprimes.

We hoped to see that the average time to factor integers of a given size would have a minimum as each parameter was iterated. However, the purpose here is not to find values for the exponent E and search bound mP_w that perform well on average, but only to see the shape of the search space. We chose these sets of integer sizes as they are relatively efficient with which to work. In Subsection 8.6.1, we propose using a modified 2 dimensional ternary search in order to find parameters that work well on average. By applying a ternary search technique rather than the linear search used here, we are able to empirically test larger sets of semiprime integers in order to get a better representation of parameters that work well on average. We are also able to do this for every even integer size in the range 16-bits to 100-bits.

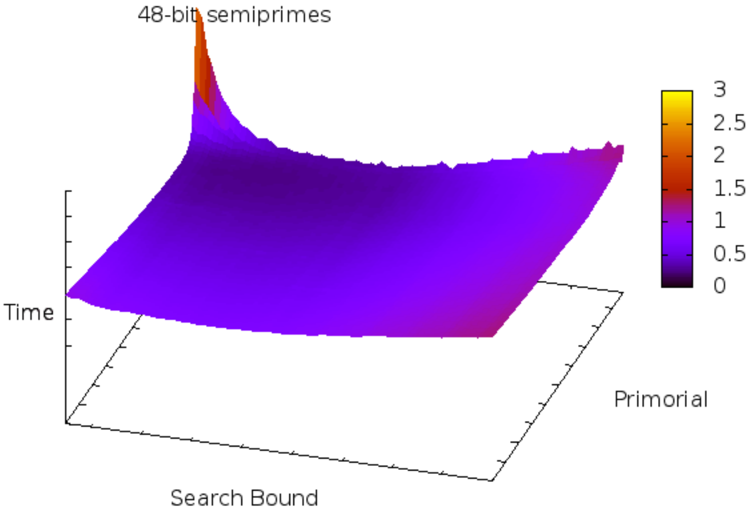

Figure 8.4: Average time to factor 48-bit integers.

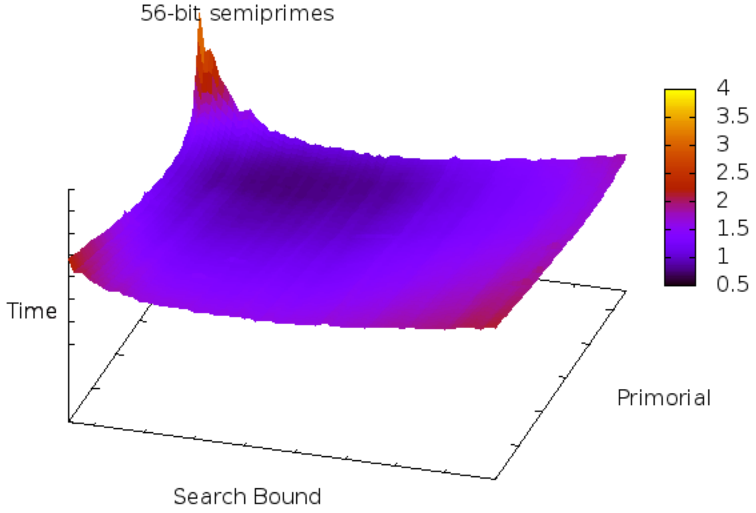

Figure 8.5: Average time to factor 56-bit integers.

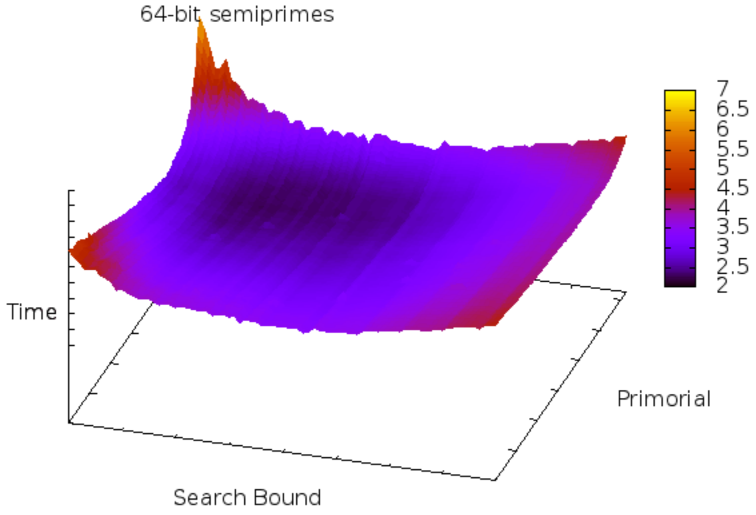

Figure 8.6: Average time to factor 64-bit integers.

The above results were generated on a 3Ghz Intel Xeon 5160 CPU with 16Gb of memory. The CPU has four cores, only one of which was used for all timing tests. For the three sets of 1000 semiprime integers, generate all the timing results took approximately one week. The timing data is noisy, which may be partly due to the inaccuracies inherit in timing very short operations. Even so, these figures indicate that as these two variables are iterated, the search space appears to have a minimum. In the next subsection, we propose a modified 2 dimensional ternary search in order to take advantage of the apparent minimum. Since a ternary search is more efficient than the linear search performed above, this allows us to use larger sets of semiprime integers so that the minimum discovered better represents parameters that work well on average. We are also able to run this method on sets of semiprime integers for every even bit size from 16 to 100 in substantially less time.

8.6.1 Two Dimensional Ternary Search of a Noisy Surface

To find values for the exponentiation stage and search stage that work well in practice, one could iterate all suitable exponents E and multiples of primorials mP_w, but this would conceivably take a long time for even a small sample set of integers. The figures in the previous section suggest a more efficient approach, namely a 2 dimensional ternary search. Using a more efficient approach also allows us to use a larger sample of integers than would not be possible otherwise.

The idea is to perform a 2 dimensional search on the variables E and mP_w by nesting two 1 dimensional ternary searches, but switching to a linear search when the search range is smaller than a specified size. Switching to a linear search when the range is small helps to deal with the inherit noise of the timing data. The 1 dimensional search is given by Algorithm 8.6, which is a subroutine of the 2 dimensional search given by 8.7.

First, recall that the results of Subsection 8.3.1 fix the prime powers of the exponent E and so only the largest prime p_t is free. Let x (with subscripts) denote indices into a list of candidate exponents ordered by increasing largest prime p_t. Candidate search bounds

mP_w are chosen from the list of search bounds $\mathcal{S}_{16384}$ generated by Algorithm 8.5 such that no candidate uses more than 16384 steps. Let y (with subscripts) denote indices into the candidate list. Finally, let $\mathcal{T}(x, y)$ denote the average time to split integers for the exponent E and search bound mP_w denoted by indices x and y.

The 1 dimensional search works as follows. Suppose the exponent E for the exponentiation stage is fixed and is indexed by x. Initially, let y_{lo} take the minimum search index and y_{hi} take the maximum search index. Then sample the time to split integers at one third and two third the value between y_{lo} and y_{hi}. If the time associated with the one third mark is higher, the first third of the interval is thrown away and the algorithm recursively computes using the remaining two thirds. If the time associated with the two third mark is higher, the last third is thrown away and the algorithm recursively computes using the first two thirds. If we assume that by fixing the exponent E and iterating the search bound, that the timings will have a minimum, the third removed at each iteration should contain values generally larger than the minimum in the remaining two thirds. Since timings are noisy, when the interval $[y_{\text{lo}}, y_{\text{hi}}]$ is sufficiently small, the algorithm simply performs a linear search in order to find the $y \in [y_{\text{lo}}, y_{\text{hi}}]$ that gives the best time. This is not guaranteed to be the best time possible for the exponent E, however, this method works well in practice. We give pseudo-code in Algorithm 8.6.

Assuming that Algorithm 8.6 computes some search bound that is near minimal for a given exponent E for the exponentiation stage, we then nest applications of the ternary search in order to search for an exponent E, and corresponding bound mP_w that work well on average. In this case, we perform a ternary search on the index x and invoke Algorithm 8.6 to determine a near minimum time for the fixed exponent associated with the index x. Once the search range is too small, we then perform a linear search on the index x in the same manner. Pseudo-code for the 2 dimensional ternary search is given by Algorithm 8.7.

Table 8.8 shows the values used for the exponentiation and search stage in practice for

Algorithm 8.6 Determine some y such that $\mathcal{T}(x, y)$ is near minimal.

Input: A function $\mathcal{T}(x, y)$ corresponding to the average time to split an integer for an exponent E associated with an index x and a search bound mP_w associated with an index y, a search range $[y_{\min}, y_{\max}]$, a bound k within which to linear search, and the index x of a fixed exponent E to be used by the exponentiation stage.

Output: $y \in [y_{\min}, y_{\max}]$ such that $\mathcal{T}(x, y)$ is near minimal for fixed E.

```
procedure SEARCH1($\mathcal{T}, x, y_{\min}, y_{\max}, k$)
{– ternary search –}
    $(y_{\text{lo}}, y_{\text{hi}}) \leftarrow (y_{\min}, y_{\max})$
    while $y_{\text{hi}} - y_{\text{lo}} > k$ do
        $y_\ell \leftarrow \lfloor (y_{\text{hi}} - y_{\text{lo}})/3 \rfloor + y_{\text{lo}}$
        $y_h \leftarrow \lfloor 2(y_{\text{hi}} - y_{\text{lo}})/3 \rfloor + y_{\text{lo}}$
        if $\mathcal{T}(x, y_\ell) \le \mathcal{T}(x, y_h)$ then
            $y_{\text{hi}} \leftarrow y_h$
        else
            $y_{\text{lo}} \leftarrow y_\ell$
{– linear search –}
    $(y, t) \leftarrow (y_{\text{lo}}, \mathcal{T}(x, y_{\text{lo}}))$        {$(y, t)$ represent smallest $\mathcal{T}(x, y)$ encountered}
    $y_{\text{lo}} \leftarrow y_{\text{lo}} + 1$
    while $y_{\text{lo}} \le y_{\text{hi}}$ do
        if $\mathcal{T}(x, y_{\text{lo}}) < t$ then $(y, t) \leftarrow (y_{\text{lo}}, \mathcal{T}(x, y_{\text{lo}}))$
        $y_{\text{lo}} \leftarrow y_{\text{lo}} + 1$
    return $(y, t)$
```

various sized integers. For each bit size, 10,000 random semiprime integers $N = p \cdot q$ were generated. We emphasize that this set is different from the set of semiprime integers used in Section 8.7 so as to avoid training SuperSPAR on the same set of integers used for comparison with other factoring algorithms. The range of exponents for the exponentiation stage was from a largest prime of 11 in the exponent to a largest prime of 2753. The range for search bounds was given by the set $\mathcal{S}_{16384}$. Algorithm 8.7 was configured to switch to a linear search when the size of the range was ≤ 8. Notice that the largest primorial used for search bounds is P_5, which justifies our approach to bounding the values e_i in the exponent $E = \prod {p_i}^{e_i}$ for values of $i \le 5$ in Section 8.3.

Algorithm 8.7 Determine an exponent E for the exponentiation stage and a search bound mP_w that work well on average.

Input: A function $\mathcal{T}(x, y)$ corresponding to the average time to split an integer for an exponent E associated with an index x and a search bound mP_w associated with an index y, two search ranges $[x_{\min}, x_{\max}]$ and $[y_{\min}, y_{\max}]$, and a bound k within which to linear search.

Output: $x \in [x_{\min}, x_{\max}]$ and $y \in [y_{\min}, y_{\max}]$ such that $\mathcal{T}(x, y)$ is near minimal for exponent E and search bound mP_w corresponding to indices x and y.

```
procedure Search2($\mathcal{T}, x_{\min}, x_{\max}, y_{\min}, y_{\max}, k$)
{– ternary search –}
    $(x_{\text{lo}}, x_{\text{hi}}) \leftarrow (x_{\min}, x_{\max})$
    while $x_{\text{hi}} - x_{\text{lo}} > k$ do
        $x_\ell \leftarrow \lfloor (x_{\text{hi}} - x_{\text{lo}})/3 \rfloor + x_{\text{lo}}$
        $x_h \leftarrow \lfloor 2(x_{\text{hi}} - x_{\text{lo}})/3 \rfloor + x_{\text{lo}}$
        $(\_, t_\ell) \leftarrow$ Search1($\mathcal{T}, x_\ell, y_{\min}, y_{\max}, k$)
        $(\_, t_h) \leftarrow$ Search1($\mathcal{T}, x_h, y_{\min}, y_{\max}, k$)
        if $t_\ell \leq t_h$ then
            $x_{\text{hi}} \leftarrow x_h$
        else
            $x_{\text{lo}} \leftarrow x_\ell$
{– linear search –}
    $x \leftarrow x_{\text{lo}}$                    {$(x, y, t)$ represent smallest $\mathcal{T}(x, y)$ encountered}
    $(y, t) \leftarrow$ Search1($\mathcal{T}, x, y_{\min}, y_{\max}, k$)
    $x_{\text{lo}} \leftarrow x_{\text{lo}} + 1$
    while $x_{\text{lo}} \leq x_{\text{hi}}$ do
        $(y', t') \leftarrow$ Search1($\mathcal{T}, x_{\text{lo}}, y_{\min}, y_{\max}, k$)
        if $t' < t$ then $(x, y, t) \leftarrow (x_{\text{lo}}, y', t')$
        $x_{\text{lo}} \leftarrow x_{\text{lo}} + 1$
    return $(x, y, t)$
```

Bit Size	Largest Prime	Search Bound	Total Steps	Average Time
	in $E = \prod {p_i}^{e_i}$	$s = mP_w$		(milliseconds)
16	11	$120 = 4P_3$	62	0.17992
20	11	$36 = 6P_2$	22	0.21529
24	11	$36 = 6P_2$	22	0.01728
28	11	$90 = 3P_3$	46	0.02461
32	11	$150 = 5P_3$	78	0.03475
36	17	$240 = 8P_3$	126	0.05341
40	17	$420 = 2P_4$	190	0.07951
44	23	$630 = 3P_4$	286	0.12786
48	47	$630 = 3P_4$	286	0.19711
52	53	$1050 = 5P_4$	478	0.32283
56	89	$840 = 4P_4$	382	0.55058
60	149	$1260 = 6P_4$	574	0.93604
64	173	$1260 = 6P_4$	574	1.51579
68	263	$2310 = 1P_5$	958	2.36646
72	347	$1680 = 8P_4$	766	3.52871
76	467	$2310 = 1P_5$	958	5.42861
80	727	$3360 = 16P_4$	1534	8.53706
84	859	$4620 = 2P_5$	1918	11.97196
88	1033	$4620 = 2P_5$	1918	18.00593
92	1597	$6090 = 29P_4$	2782	29.14281
96	1861	$9240 = 4P_5$	3838	40.02095
100	2753	$7140 = 34P_4$	3262	57.97719

Table 8.8: Values for the largest prime used in the exponent $E = \prod {p_i}^{e_i}$ for the exponentiation phase, and the baby step bound mP_w used for the search stage.

8.7 Comparison to Other Factoring Implementations

To demonstrate the performance of our implementation of SuperSPAR, several factoring implementations were compared, namely Pari/GP [7], Msieve [6], GMP-ECM [2], YAFU [8], Flint [1], Maple [4], and a custom implementation of SQUFOF [26] adapted from the source code of Pari/GP.

For each bit size 16, 18, ..., 100, sets of 10,000 random semiprime integers $N = p \cdot q$ for p and q prime and half the size of N were generated and written to disk. The file format was ASCII with one integer per line. In the case of Pari/GP and Flint, C programming libraries were provided, so we wrote a program to load the set of integers and use the library directly. For our implementations of SuperSPAR and SQUFOF, we were also able to load the set of integers and invoke the factoring function directly. Unfortunately, we were not able to directly interface with the factoring functions of Msieve, GMP-ECM, YAFU, or Maple. Our solution to time these implementations was to convert the text file to an implementation specific script and then invoke the application to factor the set.

The hardware platform used for timing is a 2.7GHz Intel Core i7-2620M CPU with 8Gb of memory. The CPU has four cores, only one of which was used during timing experiments. The operating system was 64-bit Ubuntu 12.10. When possible, the most recent version of each factoring implementation was used and built from source using the GNU C compiler version 4.7.2. Timings include forking the process, reading and parsing the test set, and factoring all the integers in the set. The timings do not include the generation of the semiprime integers or the conversion to the implementation specific batch file.

Figures 8.7 through 8.11 were chosen to emphasize different features of the performance of SuperSPAR. Figure 8.7 shows the performance of each factoring implementation for the complete range of integer sizes timed – visually, SuperSPAR is not particularly competitive for integers much larger than 90-bits. Figure 8.8 zooms in on the range of integers 44-bits to 72-bits. This shows that SQUFOF quickly becomes impractical for integers larger than 70-

bits, which is what we would expect from its $O(N^{1/4})$ runtime complexity (see [26, Theorem 4.22]). Maple, Msieve, and GMP-ECM do not perform as efficiently as the other implementations for integers in the range 44-bits to 72-bits. This also shows that SuperSPAR is highly competitive below 64-bits. Figure 8.9 shows the 5 best performing implementations for integers in the range of 44-bits to 68-bits. Figure 8.10 zooms in on the left of this image for integers in the range 44-bits to 58-bits. This shows the same 5 implementations. For integers between 44-bits and around 50-bits, the custom implementation of SQUFOF is the fastest performing, but as the integer size grows, SQUFOF takes longer more quickly. For integers around 50-bits in size, SuperSPAR is typically the fastest performing implementation of the implementations tested. Finally, Figure 8.11 zooms in on the right part of Figure 8.9. This shows the 3 best performing implementations for this range, and the point at which the performance of SuperSPAR decreases and YAFU is better performing. This occurs for integers around 64-bits in size.

Table 8.9 shows the average time in milliseconds to factor integers for each bit size given a particular implementation. The best performing implementation for integers of a given size is underlined. Timings were only recorded for implementations and integer sizes that took less than 100 milliseconds on average. In the case of SQUFOF, this was for integers 74-bits in size and less. Inexplicably, for integers of size 24-bits, 28-bits, and 42-bits, Flint did not complete the test set before crashing. Similarly, Msieve crashed for all sets of integers larger than 84-bits. This table shows that for integers in the range of 50-bits to 62-bits, SuperSPAR performs the fastest on average.

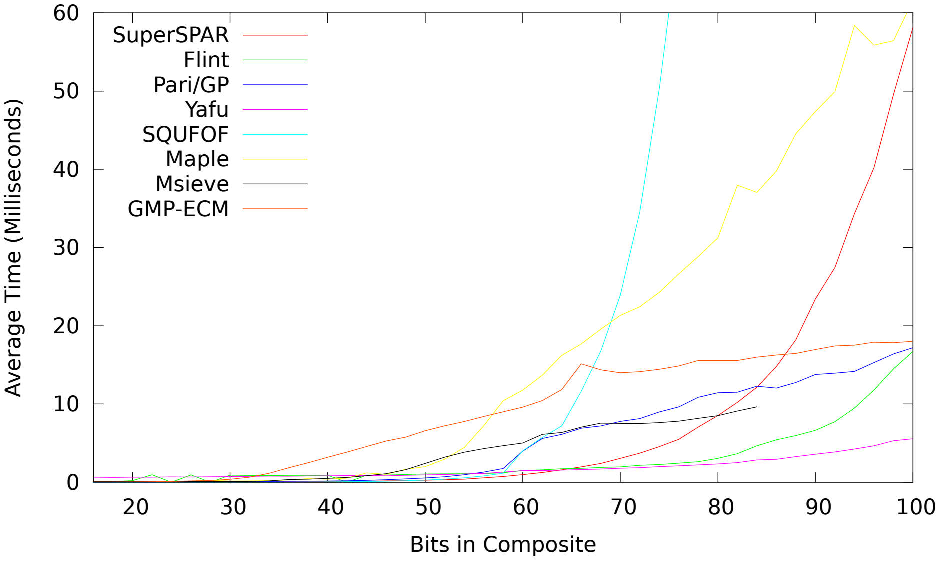

Figure 8.7: The performance of each factoring implementation for integers 16-bits to 100-bits. SuperSPAR appears to be competitive at less than 70-bits.

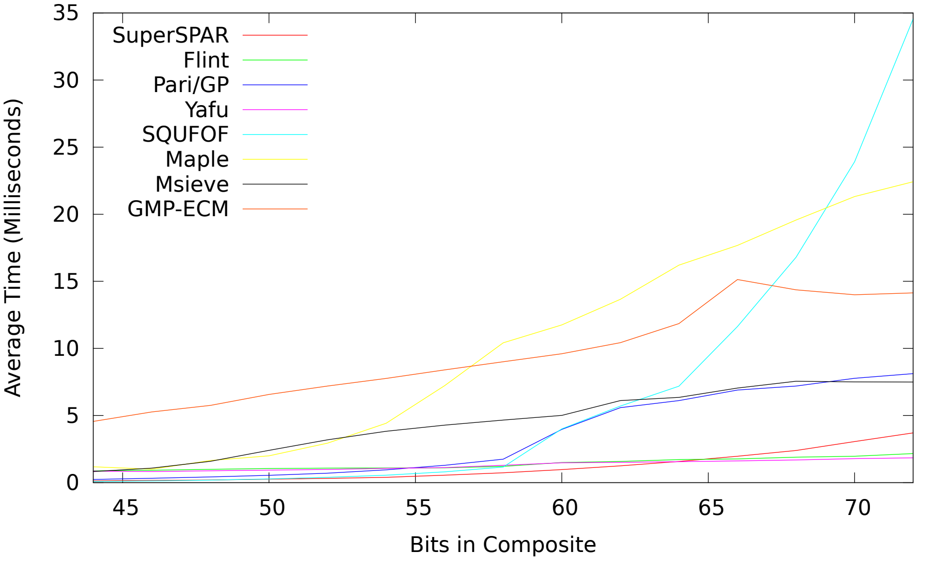

Figure 8.8: The performance of each factoring implementation for integers 44-bits to 72-bits. In this range, Maple, Msieve, and GMP-ECM do not appear to be competitive.

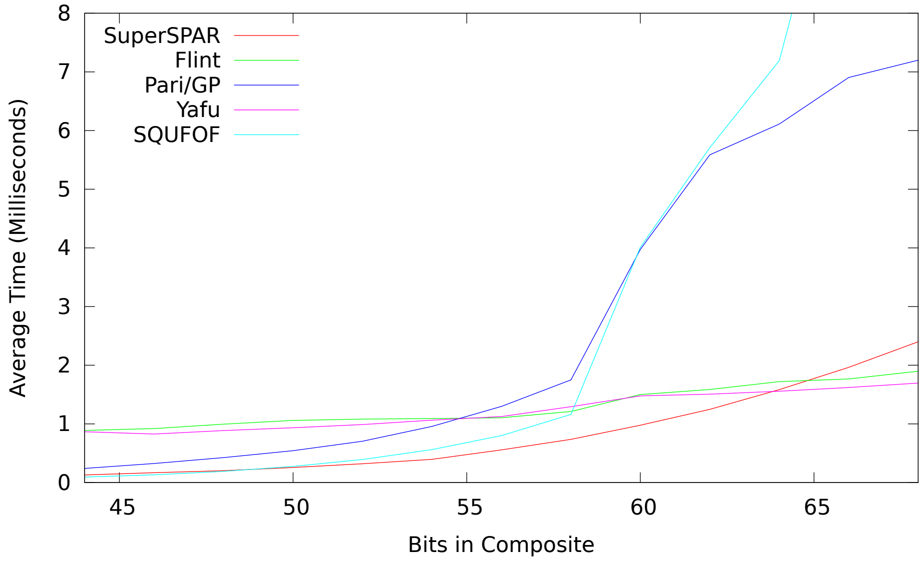

Figure 8.9: The 5 best performing factoring implementations for integers in the range 44-bits to 68-bits. This range shows that SuperSPAR is the fastest performing implementation for some integer sizes.

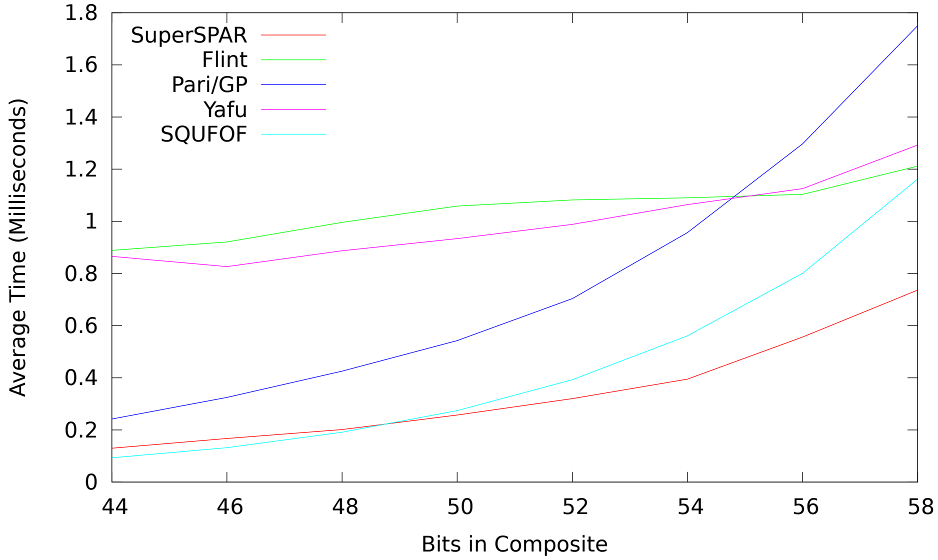

Figure 8.10: The 5 best performing factoring implementations for integers in the range 44-bits to 58-bits. This focuses the range on the lower half of Figure 8.9. Here SQUFOF grows faster than SuperSPAR and at integers around 50-bits in size, SuperSPAR is the best performing implementation.

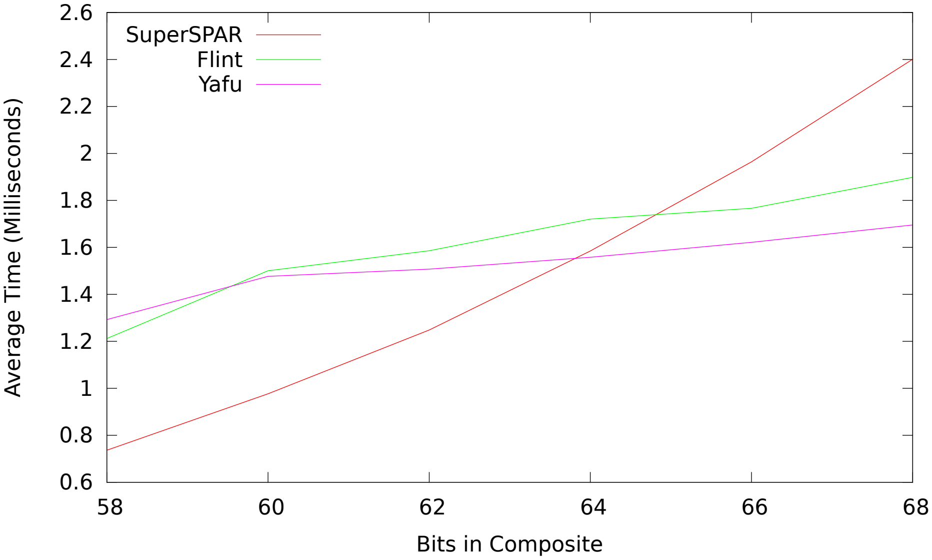

Figure 8.11: The 3 best performing factoring implementations for integers in the range 58-bits to 68-bits. This focuses the range on the upper half of Figure 8.9 and demonstrates that around integers 64-bits in size, YAFU performs better than SuperSPAR.

Bits	ECM	Flint	Maple	Msieve	Pari/GP	SQUFOF	SSPAR	Yafu
16	0.0270	0.0378	0.0505	0.0799	0.0032	0.0008	0.0097	0.6524
18	0.0337	0.0774	0.0920	0.0815	0.0048	0.0010	0.0116	0.6082
20	0.0473	0.1994	0.0637	0.0812	0.0055	0.0018	0.0128	0.6492
22	0.0660	0.9614	0.0795	0.0820	0.0064	0.0021	0.0165	0.6469
24	0.0946	failed	0.1205	0.0829	0.0090	0.0028	0.0195	0.6779
26	0.1740	0.9406	0.1340	0.0832	0.0145	0.0040	0.0221	0.6569
28	0.2235	failed	0.2380	0.0883	0.0235	0.0056	0.0265	0.6975
30	0.3783	0.8982	0.1544	0.0914	0.0677	0.0077	0.0311	0.7193
32	0.6537	0.8725	0.1641	0.1003	0.0782	0.0110	0.0365	0.7511
34	1.1481	0.8591	0.2044	0.1642	0.0981	0.0158	0.0465	0.7827
36	1.8327	0.8310	0.2822	0.3351	0.1128	0.0222	0.0548	0.7661
38	2.4831	0.8327	0.4612	0.4004	0.1371	0.0313	0.0662	0.8022
40	3.1906	0.8796	0.3906	0.4883	0.1607	0.0461	0.0821	0.8167
42	3.8457	failed	0.5226	0.6174	0.1942	0.0648	0.1045	0.8439
44	4.5650	0.8890	1.1850	0.8271	0.2419	0.0928	0.1298	0.8655
46	5.2680	0.9210	0.9994	1.0838	0.3247	0.1320	0.1671	0.8263
48	5.7561	0.9963	1.6464	1.5946	0.4254	0.1911	0.2014	0.8873
50	6.5731	1.0584	2.0015	2.4111	0.5424	0.2741	0.2573	0.9338
52	7.2033	1.0819	2.9428	3.1965	0.7036	0.3929	0.3202	0.9886
54	7.7635	1.0901	4.4271	3.8279	0.9566	0.5608	0.3949	1.0640
56	8.4054	1.1036	7.2331	4.2903	1.2972	0.8010	0.5563	1.1253
58	9.0041	1.2117	10.4179	4.6656	1.7506	1.1622	0.7365	1.2925
60	9.6012	1.5008	11.7576	5.0153	3.9717	4.0100	0.9769	1.4769
62	10.4339	1.5859	13.6654	6.1142	5.5854	5.7092	1.2482	1.5075
64	11.8548	1.7206	16.2116	6.3498	6.1125	7.1933	1.5846	1.5577
66	15.1356	1.7662	17.6785	7.0451	6.9054	11.6313	1.9639	1.6214
68	14.3670	1.8983	19.5723	7.5541	7.1997	16.7912	2.4011	1.6952
70	13.9973	1.9687	21.3196	7.5087	7.7758	23.9112	3.0578	1.7895
72	14.1343	2.1680	22.4251	7.4943	8.1268	34.5667	3.7133	1.8536
74	14.4356	2.2741	24.2755	7.6182	8.9852	50.3791	4.5587	2.0003
76	14.8602	2.4442	26.6470	7.8022	9.6272	70.0198	5.4914	2.0998
78	15.5668	2.6173	28.8623	8.1577	10.8526	–	7.0521	2.2202
80	15.5548	3.0537	31.2528	8.5141	11.4426	–	8.5300	2.3407
82	15.5577	3.6557	37.9693	9.0961	11.5011	–	10.2062	2.5122
84	15.9824	4.6584	37.0573	9.6379	12.2647	–	12.1335	2.8428
86	16.2545	5.4168	39.7901	failed	12.0382	–	14.7879	2.9365
88	16.4600	5.9574	44.5510	failed	12.7549	–	18.2133	3.2695
90	16.9574	6.6372	47.3801	failed	13.7731	–	23.4263	3.5713
92	17.4196	7.7319	49.9509	failed	13.9487	–	27.4630	3.8687
94	17.5219	9.4851	58.3808	failed	14.1614	–	34.3230	4.2393
96	17.9056	11.7830	55.8692	failed	15.2958	–	40.1884	4.6538
98	17.8311	14.4835	56.4315	failed	16.3924	–	49.5837	5.3058
100	18.0168	16.7292	61.6214	failed	17.2042	–	58.1393	5.5532

Table 8.9: The average time (in milliseconds) to factor integers of a given size for a particular implementation.

Chapter 9

Future Work

The work of this thesis is to improve exponentiation in the ideal class group of imaginary quadratic number fields, with an application to integer factoring. This lead to practical improvements when computing the extended GCD for bounded integers (Chapter 5) as well as improvements to ideal class arithmetic for bounded discriminants (Chapter 6). Chapter 7 lead to practical performance improvements when exponentiating an ideal class with bounded discriminant to a large primorial. Finally, this thesis revisits the SPAR factoring algorithm using the contributions to ideal class arithmetic and exponentiation in the context of the primorial steps algorithm. Section 8.7 compares our implementation of SuperSPAR to several integer factoring libraries to show that the performance improvements of this thesis lead to SuperSPAR being the fastest implementation for integers in the range of 54-bits to 64-bits (TODO).

For the work of this thesis to be most useful to the future work of others, the source code for the libraries used throughout is available online, and the hope is for eventual inclusion into common mathematical packages such as Pari/GP, Sage, and Dr. Michael J. Jacobson, Jr.'s algorithmic number theory library.

- A library for 128-bit arithmetic, extended GCD computations, generic group exponentiation, and 2,3 representations.

  `https://github.com/maxwellsayles/liboptarith`

- Ideal class arithmetic library, specialized for discriminants bounded by 59-bits, 118-bits, and unbound.

  `https://github.com/maxwellsayles/libqform`

- Our implementation of the SuperSPAR integer factoring library.

```
https://github.com/maxwellsayles/libsspar
```

The work presented in this thesis is by no means exhaustive. Sections 9.1, 9.2, and 9.3 contain suggestions for future work in ideal class arithmetic, exponentiation, and integer factoring respectively.

9.1 Ideal Arithmetic

The practical improvements to arithmetic in the ideal class group are based on the Algorithms for computing reduced (or almost reduced) representatives for multiplication (Subsection 2.5.5), squaring (Subsection 2.5.6), and cubing (Subsection 2.5.7). While these Algorithms are certainly faster as the size of the discriminant grows, future work could include a comparison of the implementation in this thesis with an implementation of the basic ideal class multiplication presented in Subsection 2.5.4. The idea would be to optimize an implementation as well as make improvements to the ideal reduction algorithm from Subsection 2.5.1. Ideal reduction is similar to the standard Euclidean Algorithm, as such, future work might apply several of the approaches used for computing the extended GCD to that of ideal reduction.

Additional techniques for computing the extended GCD can be explored. A relatively straightforward approach is to apply windowing to the left-to-right binary GCD that performed well for 32-bit to 64-bit integers. Binary GCD computations favour bit shifting over multiplication and division since bit shifting is faster. In groups where squaring and cubing are fast, an interesting combination to explore would be a right-to-left 2,3 GCD, similar to the right-to-left 2,3 chain of Section 7.2.

With future improvements to basic ideal class arithmetic, additional comparisons to other implementations would be useful, such as to Pari/GP and Sage. Finally, one could study practical improvements to other types of ideal arithmetic, such as the class group of real quadratic fields or function fields.

9.2 Exponentiation

The ideal class exponentiation experiments of Chapter 7 assume that the exponents are primorials and that representations can be precomputed. Future work could include a study of the time to generate 2,3 representations against the time to exponentiation using such representations. Our expectation is that representations that lead to faster exponentiations also take longer to generate, and that the rate at which the time to exponentiate improves is slower than the rate at which the time to generate the representation grows. Along this line, one could also rigorously analyse the expected cost to exponentiate using representations generated by a particular technique, or one could analyse the cost of generating representations for each of the techniques described.

The L best approximations technique of Section 7.6 generated 2,3 representations for large primorials that worked well in practice. This algorithm iterated on the L smallest partial representations. Future work could consider other heuristics on which to prune candidates, such as approximating the cost of the complete chain based on the partial chain and retaining the L best approximate costs.

For the techniques of Chapter 7, the cost associated with exponentiating shows general trends as the size of the primorial grows. However, for exponent N and exponent $N + 1$, the cost can vary considerably. Interesting and useful work would be to provide upper bounds (as a function of N) on how much the cost can vary as the exponent increases by a fixed amount.

Finally, in this thesis, we only consider double base number systems using bases 2 and 3. Future work should consider other bases. This would involve additional research with ideal arithmetic to determine the benefit of direct methods for computing $[\mathfrak{a}]^5$, $[\mathfrak{a}]^7$, and so on, for an ideal class $[\mathfrak{a}]$. Other open areas are multiple base number systems, i.e. number systems where an integer is expressed as the sum and difference of powers of several coprime bases $N = \sum_i \left(s_i \prod_j {p_j}^{e_{i,j}} \right)$.

9.3 Super^{2}SPAR

Practical improvements in this thesis to ideal class arithmetic and to exponentiation also lead to improvements in the SuperSPAR integer factoring algorithm. Future work in either area will naturally contribute to future improvements to SuperSPAR. There are, however, many open research areas that could lead to direct practical improvements to SuperSPAR.

Parameters in our implementation were chosen to work well in practice. For a given discriminant associated with the integer to be factored, there is only a probability that the integer N will be split. Often many different multipliers for the discriminant are tried before a successful splitting of N. Future work would include a study of these multipliers, and more formal work would include bounding the probability that a specific number of multipliers are used. Subsection 8.3.3 showed that for $N \bmod 4$ different multipliers were associated with a different number of ideal classes to try in expectation before a splitting of N. Studying the behaviour of multipliers may be useful in the parallelization of SuperSPAR – simply, the algorithm could run on several multiplier in parallel. One possible highly parallel platform for consideration would be an implementation of SuperSPAR for GPUs.

There are many open questions surrounding the running time of SuperSPAR. Incomplete investigations show that the majority of integers only require a single multiplier before a successful splitting of the integer, however, a small number of integers require a very large number of multipliers. These few *hard* numbers tend to skew the average running time. Future work would include quartile timings and 5-number summaries of samples of integers to be split. Additional work could include more rigorous analysis of the median asymptotic complexity of SuperSPAR. Not necessarily related to SuperSPAR would be a theoretical analysis of the factorization of the order of ideal classes, such as the probabilities of certain divisors of the order.

The study of ideal class arithmetic, exponentiation, and our implementation of SuperSPAR given in this thesis was guided by practice. SPAR is theoretically capable of larger interger factorizations, and so is SuperSPAR. Interesting future work would be to generate much larger exponents that would be stored on disk. A method of computing 2,3 representations with low space requirements (such as right-to-left 2,3 chains of Section 7.2) could be used to exponentiate an ideal by a primorial stored on disk. Since the bounded primorial steps algorithm requires a large amount of memory, an alternative Pollard-Brent recursion as suggested by Schnorr and Lenstra [45] would be space efficient and may be able to take steps coprime to the exponent used. More generally, the bounded primorial steps algorithm can be used to compute discrete logarithms, and there are many open questions as to the difficulty of the discrete log problem in the ideal class group of imaginary quadratic number fields.

Bibliography

[1] Flint 2.3. http://www.flintlib.org/. Accessed: 2013-04-10.

[2] GMP-ECM 6.4.4. http://ecm.gforge.inria.fr/. Accessed: 2013-04-10.

[3] GNU Multiple Precision (GMP) Library 5.1.1. http://gmplib.org/. Accessed: 2013-04-10.

[4] Maple 13. http://www.maplesoft.com/. Accessed: 2013-04-10.

[5] MPIR 2.6.0. http://www.mpir.org/. Accessed: 2013-16-10.

[6] Msieve 1.5.1. http://msieve.sourceforge.net/. Accessed: 2013-04-10.

[7] Pari/GP 2.5.3. http://pari.math.u-bordeaux.fr/. Accessed: 2013-04-10.

[8] YAFU (Yet Another Factoring Utility) 1.33. http://yafu.sourceforge.net/. Accessed: 2013-04-10.

[9] E. Bach and J.O. Shallit. *Algorithmic Number Theory: Efficient Algorithms*. Number 1 in Foundations of Computing. Mit Press, 1996.

[10] Valérie Berthé and Laurent Imbert. Diophantine approximation, Ostrowski numeration and the double-base number system. *Discrete Mathematics and Theoretical Computer Science*, 11(1):153–172, 2009.

[11] Richard P. Brent. Analysis of the binary Euclidean algorithm. In J. F. Traub, editor, *New Directions and Recent Results in Algorithms and Complexity*, pages 321–355. Academic Press, 1976.

[12] Richard P. Brent. An improved Monte Carlo factorization algorithm. *BIT*, 20(2):176–184, 1980.

[13] Mathieu Ciet, Marc Joye, Kristin Lauter, and Peter L. Montgomery. Trading inversions for multiplications in elliptic curve cryptography. *Des. Codes Cryptography*, 39(2):189–206, May 2006.

[14] Mathieu Ciet and Francesco Sica. An analysis of double base number systems and a sublinear scalar multiplication algorithm. *Mycrypt*, 3715:171–182, 2005.

[15] H. Cohen and G. Frey, editors. *Handbook of elliptic and hyperelliptic curve cryptography.* Chapman & Hall/CRC, 2006.

[16] H. Cohn. *Advanced Number Theory.* Dover Books on Mathematics. Dover Publications, 1980.

[17] G. E. Collins. *Lecture notes on arithmetic algorithms.* University of Wisconsin, 1980.

[18] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms.* McGraw-Hill Higher Education, 2nd edition, 2001.

[19] R.E. Crandall and C. Pomerance. *Prime Numbers: A Computational Perspective.* Springer, 2001.

[20] V. Dimitrov and T. Cooklev. Hybrid algorithm for the computation of the matrix polynomial $I + A + \cdots + A^{N-1}$. *IEEE Transactions on Circuits and Systems – I: Fundamental Theory and Applications*, 42(7), July 1995.

[21] V. Dimitrov and T. Cooklev. Two algorithms for modular exponentiation using nonstandard arithmetics. *IEICE Trans. Fundamentals*, E78-A(1), January 1995.

[22] V. Dimitrov, K.U. Järvinen, M.J. Jacobson, W.F. Chan, and Z. Huang. Provably sublinear point multiplication on Koblitz curves and its hardware implementation. *IEEE Transaction on Computers*, 57(11), November 2008.

[23] V. S. Dimitrov, L. Imbert, and P. K. Mishra. Fast elliptic curve point multiplication using double-base chains, 2005.

[24] Christophe Doche and Laurent Habsieger. A tree-based approach for computing double-base chains. In *ACISP*, pages 433–446, 2008.

[25] Albrecht Fröhlich and Martin Taylor. *Algebraic Number Theory (Cambridge Studies in Advanced Mathematics)*, volume 27. Cambridge University Press, 1993.

[26] Jason E. Gower and Samuel S. Wagstaff Jr. Square form factorization. *Math. Comput.*, 77(261):551–588, 2008.

[27] Torbjörn Granlund and Peter L. Montgomery. Division by invariant integers using multiplication. In *In Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 61–72, 1994.

[28] L.K. Hua and P. Shiu. *Introduction to Number Theory.* Springer London, Limited, 2012.

[29] L. Imbert, M.J. Jacobson, and A. Schmidt. Fast ideal cubing in imaginary quadratic number and function fields. *Advanced in Mathematics of Communications*, 4(2):237–260, 2010.

[30] Laurent Imbert and Fabrice Philippe. Strictly chained (p,q)-ary partitions. *Contributions to Discrete Mathematics*, 5(2), 2010.

[31] K. Ireland and M.I. Rosen. *A Classical Introduction to Modern Number Theory.* Graduate Texts in Mathematics. Springer, 1990.

[32] M.J. Jacobson. *Subexponential Class Group Computation in Quadratic Orders.* Berichte aus der Informatik. Shaker, 1999.

[33] M.J. Jacobson, R.E. Sawilla, and H.C. Williams. Efficient ideal reduction in quadratic fields. *International Journal of Mathematics and Computer Science*, 1:83–116, 2006.

[34] M.J. Jacobson and H.C. Williams. *Solving the Pell Equation.* CMS books in mathematics. Springer, 2009.

[35] Tudor Jebelean. Improving the multiprecision Euclidean algorithm. In Alfonso Miola, editor, *Design and Implementation of Symbolic Computation Systems*, volume 722 of *Lecture Notes in Computer Science*, pages 45–58. Springer Berlin Heidelberg, 1993.

[36] Donald E. Knuth. *The art of computer programming, volume 3: (2nd ed.) sorting and searching.* Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

[37] Andrew Kuchling. Python's dictionary implementation: Being all things to all people. In Andy Oram and Greg Wilson, editors, *Beautiful Code: Leading Programmers Explain How They Think*, pages 293–318. O'Reilly Media, 2008.

[38] D.H. Lehmer. Euclid's algorithm for large numbers. *American Mathematical Monthly*, 45(4):227–233, April 1938.

[39] Jr. H.W. Lenstra and Carl Pomerance. A rigorous time bound for factoring integers. *Journal of the American Mathematical Society*, 5(3), July 1992.

[40] Nicolas Méloni and M. Anwar Hasan. Elliptic curve scalar multiplication combining Yao's algorithm and double bases. In *Proceedings of the 11th International Workshop on Cryptographic Hardware and Embedded Systems*, CHES '09, pages 304–316, Berlin, Heidelberg, 2009. Springer-Verlag.

[41] Niels Möller and Torbjörn Granlund. Improved division by invariant integers. *IEEE Transactions on Computers*, 60(2):165–175, 2011.

[42] J. M. Pollard. A monte carlo method for factorization. *BIT Numerical Mathematics*, 15:331–334, 1975.

[43] S. Ramachandran. Numerical results on class groups of imaginary quadratic fields. Master's thesis, University of Calgary, Canada, 2006.

[44] George W. Reitwiesner. Binary arithmetic. *Advances in Computers*, 1:231–308, 1960.

[45] C.P. Schnorr and Jr. H.W. Lenstra. A Monte Carlo factoring algorithm with linear storage. *Mathematics of computation*, 43(167), July 1984.

[46] Jeffrey Shallit and Jonathan Sorenson. Analysis of a left-shift binary GCD algorithm. *Journal of symbolic computation*, 17:473–486, 1994.

[47] D. Shanks. Class number, A theory of factorization and genera. In *Symp. Pure Math.*, volume 20, pages 415–440, Providence, R.I., 1971. AMS.

[48] D. Shanks. On Gauss and composition I, II. *Proc. NATO ASI on Number Theory and Applications*, pages 163–179, 1989.

[49] J. Stein. Computational problems associated with racah algebra. *Journal of Computational Physics*, 1(3):397–405, February 1967.

[50] A.V. Sutherland. *Order computations in generic groups.* PhD thesis, M.I.T., 2007.

[51] A.V. Sutherland. A generic approach to searching for Jacobians. *Mathematics of computation*, 78(265):485–507, january 2009.

[52] Henry S. Warren. *Hacker's Delight.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.