Chapter 1

Motivation

1.1 Faster ideal exponentiation on imaginary quadratic fields

1.2 Exponentiation with fixed exponents

1.3 Faster class group computations

1.4 Applications in cryptography

1.5 Examples of fast ideal exponentiation in SuperSPAR

1.6 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)

The software was developed using the GNU C compiler version 4.7.2 on Ubuntu 12.10. The hardware platform was a personal notebook with 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.

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 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 [8], [17], or [20].

2.1 Quadratic Numbers

A *quadratic number* is a root α of a quadratic polynomial $f(x) = ax^2 + bx + c$ with integer coefficients. The roots of $f(x)$ are given by the quadratic formula

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

The *discriminant* of $f(x)$ is $\Delta = b^2 - 4ac$, and we construct a quadratic number field $\mathbb{K}$ as an extension field of the rational numbers $\mathbb{Q}$ as

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

When 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. In this thesis we concern ourselves only with the imaginary case. Notice that if $\Delta = f^2\Delta_0$ for some $f \in \mathbb{Z}_{>0}$ where Δ_0 is square-free, then $\mathbb{Q}(\sqrt{\Delta}) = \mathbb{Q}(\sqrt{\Delta_0})$. For Δ_0 to be square-free, it must be that $\Delta_0 \not\equiv 0 \pmod 4$ since this would imply that Δ_0 is divisible by 4, a perfect square.

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 $bx-a$. The roots of monic degree 1 polynomials with integer coefficients are the integers, $\mathbb{Z}$. When $f(x)$ is a monic quadratic polynomial with integer coefficients, the root α is a *quadratic integer*. By [23, p.77] we have the following theorem.

Theorem 2.2.1. 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

$$\omega_0 = \begin{cases} \sqrt{\Delta_0} & \text{when } \Delta_0 \equiv 2,3 \pmod 4 \\ \frac{1+\sqrt{\Delta_0}}{2} & \text{when } \Delta_0 \equiv 1 \pmod 4. \end{cases}$$

2.3 Maximal Order of Algebraic Integers

The *maximal order* of a field $\mathbb{K}$ is the set of all algebraic integers contained within $\mathbb{K}$. We use the notation of $\mathbb{Z}$-modules to characterize the maximal order of quadratic integers.

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

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

Theorem 2.3.2. A *quadratic order* $\mathcal{O}_\Delta$ of $\mathbb{Q}(\sqrt{\Delta})$ is a sub-ring of the quadratic integers of $\mathbb{Q}(\sqrt{\Delta})$ containing 1. Following Jacobson [23, p.81], we write $\mathcal{O}_\Delta$ as

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

The maximal order $\mathcal{O}_\Delta = [1, \omega_0]$ of $\mathbb{Q}(\sqrt{\Delta_0})$ is the ring of all quadratic integers in $\mathbb{Q}(\sqrt{\Delta_0})$. This order is maximal since any other 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 $\mathfrak{a}$.

For any $\alpha, \beta \in \mathcal{O}_\Delta$, the set $\{x\alpha + y\beta : x, y \in \mathcal{O}_\Delta\}$ is an ideal $\mathfrak{a}$ in the order $\mathcal{O}_\Delta$. We say that $\mathfrak{a}$ is generated by α and β and denote it (α, β). Every ideal $\mathfrak{a}$ of a quadratic order $\mathcal{O}_\Delta$ can be represented by at most two generators [8], but some can be represented by only a single generator. An ideal represented by a single generator, $\alpha \in \mathcal{O}_\Delta$, is denoted (α) and is called *principal* [23, p.87].

Theorem 2.4.2. When $\mathcal{O}_\Delta = [1, f\omega_0]$ is the 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 $s, a, b \in \mathbb{Z}, s > 0, a > 0$, $\gcd(a, b, (b^2 - \Delta)/4a) = 1$, $b^2 \equiv \Delta \pmod{4a}$, and b is unique mod $2a$ [21, p.13]. When $s = 1$, the ideal $\mathfrak{a}$ is *primitive*.

The *identity* ideal is $\mathcal{O}_\Delta = [1, f\omega_0]$ since $\mathfrak{a} = \mathfrak{a}\mathcal{O}_\Delta = \mathcal{O}_\Delta\mathfrak{a}$ [8]. An ideal $\mathfrak{a} = s[a, (b + \sqrt{\Delta})/2]$ is *invertible* if there exists an ideal $\mathfrak{b}$ such that $\mathfrak{a}\mathfrak{b} = \mathcal{O}_\Delta$, and an ideal $\mathfrak{b}$ exists when $\gcd(a, b, (b^2 - \Delta)/(4a)) = 1$ [21, p.14]. The inverse is given by [21, pp.14,15]

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

where $\mathcal{N}(\mathfrak{a}) = s^2 a$ is the norm of $\mathfrak{a}$ and is multiplicative. When $\mathcal{O}_\Delta$ is maximal, all ideals of $\mathcal{O}_\Delta$ are invertible.

Prime ideals allow us to easily create an ideal. For a prime ideal $\mathfrak{p} \in \mathcal{O}_\Delta$ it can be shown ([21, 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 $b = \pm\sqrt{\Delta} \pmod{p}$ and $\mathfrak{p} = [p, (b+\sqrt{\Delta})/2]$ when $4p \mid b^2 - \Delta$. Algorithm 1 gives pseudo-code.

Algorithm 1 Prime Ideal

Input: A prime integer $p \in \mathbb{Z}$.
Output: A representative $\mathfrak{p} = [p, (b+\sqrt{\Delta})/2]$ such that $\mathfrak{p}$ is a prime ideal if one exists.

```
b ← the positive √Δ (mod p)
if 4p | b^2 − Δ then
    return [p, (b + √Δ)/2]
end if
b ← p − b
if 4p | b^2 − Δ then
    return [p, (b + √Δ)/2]
end if
return none
```

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 $(\alpha)\mathfrak{a} = (\beta)\mathfrak{b}$ [23, p.88]. Following Jacobson [23, p.88], we denote by $[\mathfrak{a}]$ the *ideal class* of all ideals equivalent to $\mathfrak{a}$.

2.5 Ideal Class Group

Recall that two ideals $\mathfrak{a}$ and $\mathfrak{b}$ are equivalent if there exists principal ideals (α) and (β) such that $(\alpha)\mathfrak{a} = (\beta)\mathfrak{b}$. An ideal class $[\mathfrak{a}]$ is the set of all ideals that are equivalent to $\mathfrak{a}$. As such, an ideal $\mathfrak{a}$ is a *representative* for the ideal class $[\mathfrak{a}]$. The *ideal class group*, Cl_Δ, is the set of all equivalence classes of invertible $\mathcal{O}$-ideals, with the group operation defined as the product of class representatives.

Our implementation of ideal arithmetic represents an ideal as $\mathfrak{a} = [a, (b + \sqrt{\Delta})/2]$ where $\mathfrak{a}$ is primitive. Additionally, we maintain the value $c = (b^2 - \Delta)/4a$. We represent the class group 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.

In Subsection 2.5.1, we state what it is for an ideal to be in reduced form, and in subsection 2.5.2, we show how to multiply two reduced class representatives. In Subsection 2.5.3 we discuss how to perform multiplication such that the result is a reduced or almost reduced representative, and then extend this to the case of computing the square (2.5.4) and cube (2.5.5) 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 *reduced* when $-a < b \leq a < c$ or when $0 \leq b \leq a = c$ for $c = (b^2 - \Delta)/4a$ [9, p.241].

Algorithm 2 Ideal Reduction

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 $\mathfrak{a} = [a, (b + \sqrt{\Delta})/2]$.

1: $(a, b, c) \leftarrow (a_1, b_1, c_1)$
2: **while** $a > c$ or $b > a$ or $b \leq -a$ **do**
3: **if** $a > c$ **then**
4: swap a with c and let $b \leftarrow -b$
5: **end if**
6: **if** $b > a$ or $b \leq -a$ **then**
7: $b \leftarrow b'$ such that $-a < b' \leq a$ and $b' \equiv b \pmod{2a}$
8: $c \leftarrow (b^2 - \Delta)/4a$
9: **end if**
10: **end while**
11: **if** $a = c$ and $b < 0$ **then**
12: $b \leftarrow -b$
13: **end if**
14: **return** $[a, (b + \sqrt{\Delta})/2]$

In an imaginary quadratic field, every ideal class contains exactly one reduced ideal [29, p.20]. There are several algorithms to compute a reduced ideal, many of which are listed in [22]. Here we present the algorithm we use. We adapt the work presented on [22, p.90] and [23, p.99]. If $\mathfrak{a} = [a, (b + \sqrt{\Delta})/2]$ is a primitive ideal then

$$\mathfrak{b} = \left[-\mathcal{N}((b + \sqrt{\Delta})/2)/a, -(b - \sqrt{\Delta})/2\right] \tag{2.1}$$

is also a primitive ideal, since we can verify that

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

Simplifying Equation 2.1 we get

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

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

$$\mathfrak{b} = \left[c, \frac{-b + \sqrt{\Delta}}{2}\right]. \tag{2.2}$$

As such, the first step is if $a > c$, we can reduce a by setting $\mathfrak{a} = [c, (-b + \sqrt{\Delta})/2]$. Since b is unique mod $2a$, we can also reduce $b \bmod 2a$. We repeat these steps while $\mathfrak{a}$ is not reduced. In the case that $a = c$, we use the absolute value of b, since by Equation 2.2 the ideals $[a, (b + \sqrt{\Delta})/2]$ and $[c, (-b + \sqrt{\Delta})/2]$ are equivalent. Pseudo-code is given in Algorithm 2.

2.5.2 Multiplication of Ideal Classes

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

$$c_2 = ({b_2}^2 - \Delta)/4a_2, \tag{2.3}$$

$$s = \gcd(a_1, a_2, (b_1 + b_2)/2) = Ya_1 + Va_2 + W(b_1 + b_2)/2, \tag{2.4}$$

$$U = (V(b_1 - b_2)/2 - Wc_2) \bmod (a_1/s), \tag{2.5}$$

$$a = (a_1 a_2)/s^2, \tag{2.6}$$

$$b = (b_2 + 2Ua_2/s) \bmod 2a, \tag{2.7}$$

$$\mathfrak{ab} = s\left[a, \frac{b + \sqrt{\Delta}}{2}\right]. \tag{2.8}$$

The remainder of this subsection is used to derive the above equations. We adapt much of the presentation given in [23, pp.117,118]. Component-wise multiplication of $\mathfrak{a}$ and $\mathfrak{b}$ give us

$$\mathfrak{ab} = \left[a_1 a_2, \frac{a_1 b_2 + a_1\sqrt{\Delta}}{2}, \frac{a_2 b_1 + a_2\sqrt{\Delta}}{2}, \frac{b_1 b_2 + (b_1 + b_2)\sqrt{\Delta} + \Delta}{4}\right]. \tag{2.9}$$

By the multiplicative property of the norm we have

$$N(\mathfrak{ab}) = s^2 a = N(\mathfrak{a})N(\mathfrak{b}) = a_1 a_2$$
$$\Rightarrow\ a = \frac{a_1 a_2}{s^2},$$

which gives us Equation 2.6. Now, by the second term of equation (2.9) we know that $(a_1 b_2 + a_1\sqrt{\Delta})/2 \in \mathfrak{ab}$. It follows that there is some $x, y \in \mathbb{Z}$ such that

$$\frac{a_1 b_2 + a_1\sqrt{\Delta}}{2} = xsa + ys\left(\frac{b + \sqrt{\Delta}}{2}\right).$$

Equating irrational parts we have

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

Hence, $s \mid a_1$. Similarly, by the third and fourth terms of equation (2.9) we have $(a_2 b_1 + a_2\sqrt{\Delta})/2 \in \mathfrak{ab}$, which implies that $s \mid a_2$, and $(b_1 b_2 + (b_1 + b_2)\sqrt{\Delta} + \Delta)/4 \in \mathfrak{ab}$, 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 (2.9) 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, we have

$$\frac{sb + s\sqrt{\Delta}}{2} = \left(Xa_1a_2 + Y\frac{a_1b_2}{2} + V\frac{a_2b_1}{2} + W\frac{b_1b_2 + \Delta}{4}\right) + \left(Y\frac{a_1}{2} + V\frac{a_2}{2} + W\frac{b_1+b_2}{4}\right)\sqrt{\Delta}. \tag{2.10}$$

Again, by equating irrational parts we have

$$\begin{aligned} \frac{s\sqrt{\Delta}}{2} &= \left(Y\frac{a_1}{2} + V\frac{a_2}{2} + W\frac{b_1+b_2}{4}\right)\sqrt{\Delta} \\ s &= Ya_1 + Va_2 + W\frac{b_1+b_2}{2}, \end{aligned} \tag{2.11}$$

which is the same as Equation 2.4. Since s divides each of a_1, a_2, and $(b_1+b_2)/2$, we have that $s = \gcd(a_1, a_2, (b_1+b_2)/2)$.

It remains to compute $b \pmod{2a}$. Recall that $a = a_1a_2/s^2$. This time, by equating the rational parts of (2.10) we have:

$$\begin{aligned} \frac{sb}{2} &= Xa_1a_2 + Y\frac{a_1b_2}{2} + V\frac{a_2b_1}{2} + W\frac{b_1b_2 + \Delta}{4} \\ b &= 2X\frac{a_1a_2}{s} + Y\frac{a_1b_2}{s} + V\frac{a_2b_1}{s} + W\frac{b_1b_2 + \Delta}{2s} \\ b &\equiv Y\frac{a_1b_2}{s} + V\frac{a_2b_1}{s} + W\frac{b_1b_2 + \Delta}{2s} \pmod{2a} \end{aligned} \tag{2.12}$$

This gives us b. However, we can rewrite (2.12) with fewer multiplies and divides. By equation (2.11), we have

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

Substituting into equation (2.12) we get

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

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 2.7. Note that the product ideal $\mathfrak{ab}$ is not a reduced representative and that the storage needed can be as much as twice that of the ideal factors $\mathfrak{a}$ and $\mathfrak{b}$.

2.5.3 Fast Ideal Multiplication (NUCOMP)

Shanks [34] 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 fewer (often no) applications of the reduction operator to be converted to reduced form. The description of NUCOMP provided here is a high level description of the algorithm based on [23, pp.119-123].

Equations 2.4, 2.6, and 2.7 from the previous subsection give a solution to the ideal product $\mathfrak{ab} = s[a, (b + \sqrt{\Delta})/2]$. We begin with the observation ([23, p.119]) that the fraction $(b/2)/a$ is roughly equal to the fraction sU/a_1 where U is given by Equation 2.5:

$$\frac{b}{2a} = \frac{b_2 + 2Ua_2/s}{2a_1 a_2/s^2} = \frac{s^2 b_2 + s2Ua_2}{2a_1 a_2} = \frac{s^2 b_2}{2a_1 a_2} + \frac{sU}{a_1} \approx \frac{sU}{a_1}.$$

Following Jacobson [23, 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

$$q_i = \lfloor R_{i-2} \ / \ R_{i-1} \rfloor$$
$$R_i = R_{i-2} - q_i R_{i-1}$$
$$C_i = C_{i-2} - q_i C_{i-1}$$

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

$$R_i < \sqrt{a_1/a_2} \ |\Delta/4|^{1/4} < R_{i-1}. \tag{2.13}$$

Initial values for the recurrence are given by

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

We then compute

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

for the reduced or almost reduced product $\mathfrak{ab} = [a, (b+\sqrt{\Delta})/2]$. 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 2.13 and we compute the product $\mathfrak{ab}$ as in the previous subsection without expanding the simple continued fraction sU/a_1.

Our implementation of fast ideal multiplication includes some optimizations on the above technique. Equation 2.4 requires that we compute $\gcd(a_1, a_2, (b_1 + b_2)/2)$. Since $\gcd(a_1, a_2)$ is often equal to 1, we only need compute $\gcd(a_1, a_2, (b_1 + b_2)/2)$ when

$\gcd(a_1, a_2) \neq 1$. Also, by Equation 2.4, we have that $s \mid a_1$ and $s \mid a_2$, so we reduce both a_1 and a_2 by s throughout. Finally, the first coefficient, Y, from Equation 2.4 is never used, and so we do not compute it. We give pseudo-code in Algorithm 3.

Algorithm 3 NUCOMP – Fast Ideal Multiplication. Based on [23, 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 Δ.
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' and V' such that $s' = \gcd(a_1, a_2) = Y'a_1 + V'a_2$ for $s', Y', V' \in \mathbb{Z}$
$s \leftarrow 1$
$U \leftarrow V'(b_1 - b_2)/2 \bmod a_1$
if $s' \neq 1$ **then**
    compute s, V, and W
    such that $s = \gcd(s', (b_1 + b_2)/2) = Vs' + W(b_1 + b_2)/2$ for $V, W \in \mathbb{Z}$
    $(a_1, a_2) \leftarrow (a_1/s, a_2/s)$
    $U \leftarrow (VU - Wc_2) \bmod a_1$
end if
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 $[a, (b + \sqrt{\Delta})/2]$
end if
$\begin{bmatrix} R_{-2} & R_{-1} \\ C_{-2} & C_{-1} \end{bmatrix} \leftarrow \begin{bmatrix} U & a_1 \\ -1 & 0 \end{bmatrix}$
find i such that $R_i < \sqrt{a_1/a_2}\ |\Delta/4|^{1/4} < R_{i-1}$ using the recurrences:
$q_i = \lfloor R_{i-2} \ / \ R_{i-1} \rfloor$
$R_i = R_{i-2} - q_i R_{i-1}$
$C_i = C_{i-2} - q_i C_{i-1}$
$M_1 \leftarrow (R_i a_2 + C_i(b_1 - b_2)/2)/a_1$
$M_2 \leftarrow (R_i(b_1 + b_2)/2 - sC_i c_2)/a_1$
$a \leftarrow (-1)^{i+1}(R_i M_1 - C_i M_2)$
$b \leftarrow ((2(R_i a_2 - C_{i-1}a)/C_i) - b_2) \bmod 2a$
return $[a, (b + \sqrt{\Delta})/2]$

2.5.4 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. For reduced ideals $\mathfrak{a} = [a_1, (b_1 + \sqrt{\Delta})/2]$ and $\mathfrak{b} =$

$[a_2, (b_2 + \sqrt{\Delta})/2]$, we have $a_1 = a_2$ and $b_1 = b_2$. Equations 2.4 and 2.5 simplify to

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

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

We then compute 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

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

As before, the representative $[a, (b + \sqrt{\Delta})/2]$ is either reduced or almost reduced. Pseudo-code for our implementation is given in algorithm 4.

2.5.5 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 [18].

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

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

Algorithm 4 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 Δ.
Output: A reduced or almost reduced representative $\mathfrak{a}^2$.

compute s and Y such that $s = \gcd(a_1, b_1) = Xa_1 + Yb_1$ for $s, X, Y \in \mathbb{Z}$
$a_1 \leftarrow a_1/s$
$U \leftarrow -Yc_1 \bmod a_1$
if $a_1 < |\Delta/4|^{1/4}$ **then**
  $a \leftarrow {a_1}^2$
  $b \leftarrow (2Ua_1 + b_1) \bmod 2a$
  return $[a, (b + \sqrt{\Delta})/2]$
end if
$\begin{bmatrix} R_{-2} & R_{-1} \\ C_{-2} & C_{-1} \end{bmatrix} \leftarrow \begin{bmatrix} U & a_1 \\ -1 & 0 \end{bmatrix}$
Find i such that $R_i < |\Delta/4|^{1/4} < R_{i-1}$ using the recurrences:
  $q_i = \lfloor R_{i-2}/R_{i-1} \rfloor$
  $R_i = R_{i-2} - q_i R_{i-1}$
  $C_i = C_{i-2} - q_i C_{i-1}$
$M_2 \leftarrow (R_i b_1 - sC_i c_1)/a_1$
$a \leftarrow (-1)^{i+1}({R_i}^2 - C_i M_2)$
$b \leftarrow (2(R_i a_1 + C_{i-1}a)/C_i) \bmod 2a$
return $[a, (b + \sqrt{\Delta})/2]$

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

$$s = \gcd(s'a_1, {b_1}^2 - a_1 c_1) = Xs'a_1 + Y({b_1}^2 - a_1 c_1)$$

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

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

We then 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, we can compute the reduced or almost reduced representative $\mathfrak{a}^3 = [a, (b+\sqrt{\Delta})/2]$

using the equations

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

As always, the ideal $[a, (b + \sqrt{\Delta})/2]$ is reduced or almost reduced. Pseudo-code for our implementation of fast ideal cubing is given in Algorithm 5.

Algorithm 5 NUCUBE – Fast Ideal Cubing. Adapted from [18, 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' and Y' such that $s' = \gcd(a_1, b_1) = X' a_1 + Y' b_1$ for $s', X', Y' \in \mathbb{Z}$
if $s' = 1$ **then**
  $s \leftarrow 1$
  $U \leftarrow Y' c_1 (Y'(b_1 - Y' c_1 a_1) - 2) \bmod {a_1}^2$
else
  compute s, X, and Y such that $s = \gcd(s' a_1, {b_1}^2 - a_1 c_1) = X s' a_1 + Y({b_1}^2 - a_1 c_1)$ for $s, X, Y \in \mathbb{Z}$
  $U \leftarrow -c_1(XY' a_1 + Y b_1) \bmod {a_1}^2/s$
end if
if ${a_1}^2/s < \sqrt{a_1}\ |\Delta/4|^{1/4}$ **then**
  $a \leftarrow {a_1}^3/s^2$
  $b \leftarrow (b_1 + 2U a_1/s) \bmod 2a$
  return $[a, (b + \sqrt{\Delta})/2]$
end if
$\begin{bmatrix} R_{-2} & R_{-1} \\ C_{-2} & C_{-1} \end{bmatrix} \leftarrow \begin{bmatrix} U & ({a_1}^2/s) \\ -1 & 0 \end{bmatrix}$
Find i such that $R_i < \sqrt{a_1}|\Delta/4|^{1/4} < R_{i-1}$ using the recurrences:
$q_i = \lfloor R_{i-2}/R_{i-1} \rfloor$
$R_i = R_{i-2} - q_i R_{i-1}$
$C_i = C_{i-2} - q_i C_{i-1}$
$M_1 \leftarrow (R_i a_1 + C_i U a_1)/{a_1}^2$
$M_2 \leftarrow (R_i(b_1 + U a_1) - s C_i c_1)/{a_1}^2$
$a \leftarrow (-1)^{i+1} R_i M_1 - C_i M_2$
$b \leftarrow (2(R_i a_1/s - C_{i-1} a)/C_i - b_1) \bmod 2a$
return $[a, (b + \sqrt{\Delta})/2]$

In the next chapter, we'll discuss some exponentiation techniques that make use of 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 a group element. Our approach is to exponentiate a group element to the product, P, of several small primes. The result is an element whose order is likely to not be divisible by any of these primes. We then compute the order using a variant of Shanks' baby-step giant-step algorithm where only powers relatively prime to P are computed. Determining the order of an element is useful in computing the structure of a group, and in factoring an integer associated with a group. Faster exponentiation means the entire computation 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 introduce 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.

3.1 Binary Exponentiation

The simplest method of exponentiation is binary exponentiation. Let g be an element in the 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 [7].

Binary exponentiation algorithms require $\lfloor \log_2 n \rfloor$ squarings and on average $\lfloor \log_2 n \rfloor / 2$ multiplications, 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, we 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 [30], that maintains a carry flag c (see Algorithm 6). Instead of adding 2^i to n, we set $c = 1$, and instead of subtracting 2^i from n, we set $c = 0$. When inspecting n two bits at

Algorithm 6 Computes g^n using right-to-left non-adjacent form (Reitwiesner [30]).

Input: $g \in G, n \in \mathbb{Z}_{\geq 0}$

```
1: c ← 0                                              {carry flag}
2: T ← g                                              {invariant: T = g^(2^i)}
3: R ← 1_G
4: i ← 0
5: while n ≥ 2^i do
6:    if ⌊n/2^i⌋ + c ≡ 1 (mod 4) then
7:       R ← R · T
8:       c ← 0
9:    else if ⌊n/2^i⌋ + c ≡ 3 (mod 4) then
10:      R ← R · T^(-1)
11:      c ← 1
12:   end if
13:   T ← T^2
14:   i ← i + 1
15: end while
16: if c = 1 then
17:   R ← R · T
18: end if
19: return R
```

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.

An advantage of non-adjacent form is that it requires at most $\lfloor \log_2 n \rfloor + 1$ squares and on average $(\lfloor \log_2 n \rfloor + 1)/3$ multiplications. 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 $1/3$.

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 introduced by Dimitrov and

Cooklev [10, 11], 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 $a_i, b_i, k \in \mathbb{Z}_{\geq 0}$. We are particularly interested in bases $p = 2$ and $q = 3$ and representations of n such that $n = \sum s_i 2^{a_i} 3^{b_i}$. We refer to such representations 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. The best representation depends on the needs of the application. Later, we shall see some algorithms that take this into account, but many are designed to either find representations quickly, of a special form, or with few terms.

Recall, the binary representation (3.1), non-adjacent form (3.2), and a 2,3 double-base representation (3.5) of 23814216 were

$$23814216 = 2^3 + 2^6 + 2^{13} + 2^{14} + 2^{16} + 2^{17} + 2^{19} + 2^{21} + 2^{22} + 2^{24}, \qquad \text{using binary}$$

$$23814216 = 2^3 + 2^6 - 2^{13} - 2^{15} - 2^{18} - 2^{20} - 2^{23} + 2^{25}, \qquad \text{using NAF}$$

$$23814216 = 2^3 3^2 - 2^{13} 3^2 + 2^{15} 3^6. \qquad \text{using 2,3 DBNS}$$

The binary representation has 10 terms, while the non-adjacent form has only 8 terms. The 2,3 representation given above contains only three terms. While a non-adjacent

form has in expectation $\lfloor \log_2 n \rfloor /3$ terms and a binary representation has in expectation $\lfloor \log_2 n \rfloor /2$ terms, both are bounded by $O(\log n)$ in the number of terms. Both Dimitrov et al [12] and Ciet and Sica [5] give algorithms for 2,3 double-base representations, where they prove that the number of terms is bounded by $O(\log n/ \log \log n)$. For this reason, many algorithms focus on reducing the number of terms in a 2,3 representation. Naturally, the time to compute a 2,3 double-base exponentiation is further reduced if the cost of squaring is less than that of multiplication, or the cost of cubing is less than multiplication with the square.

3.3.1 Chains

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 $g^{23814216} =$

Algorithm 7 Computes g^n given n as a chained 2,3 partition. Adapted from [13].

Input: $g \in G$, $n = \sum_{i=1}^{k} s_i 2^{a_i} 3^{b_i}$,
$s_1, ..., s_k \in \{-1, 1\}$, $0 \leq a_1 \leq ... \leq a_k \in \mathbb{Z}$, $0 \leq b_1 \leq ... \leq b_k \in \mathbb{Z}$.

1: $i \leftarrow 1$
2: $a \leftarrow 0$ {current power of 2}
3: $b \leftarrow 0$ {current power of 3}
4: $T \leftarrow g$ {loop invariant: $T = g^{2^a 3^b}$}
5: $R \leftarrow 1_G$
6: **while** $i \leq k$ **do**
7: **while** $a < a_i$ **do**
8: $T \leftarrow T^2, a \leftarrow a + 1$
9: **end while**
10: **while** $b < b_i$ **do**
11: $T \leftarrow T^3, b \leftarrow b + 1$
12: **end while**
13: $R \leftarrow R \cdot T^{s_i}$ {multiply with T or T^{-1}}
14: $i \leftarrow i + 1$
15: **end while**
16: **return** R

$g^{2^3 3^2 - 2^{13} 3^2 + 2^{15} 3^6}$ can be computed by first computing $x_1 = g^{2^3 3^2}$, $x_2 = {x_1}^{2^{10}}$, $x_3 = {x_2}^{2^2 3^4}$, and finally $g^{23814216} = x_1 \cdot {x_2}^{-1} \cdot x_3$. When n is given as a chained 2,3 representation, Algorithm 7 can be used to compute g^n using exactly a_k squares, b_k cubes, $k - 1$ multiplications, and at most one inverse[1]. Since $x_i \mid x_{i+1}$, we need only retain x_i in order to compute x_{i+1}, and so the algorithm requires only a constant number of group elements.

There is evidence [18], however, 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 provably sublinear in the length of the input [12, 5].

[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.

3.3.2 Exponentiating General 2,3 Representations

Another class of algorithms compute representations without the constraint that larger terms be divisible by smaller terms. Given a general 2,3 representation for an integer $n = \sum s_i 2^{a_i} 3^{b_i}$, Méloni and Hasan [26, Section 3.2] give an algorithm that achieves the same bound on the number of operations as in algorithm 7, but with a bound of $O(\min\{\max a_i, \max b_i\})$ on the memory used. Suppose $\max b_i < \max a_i$. We require the terms to be 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 $s_i 2^{a_i} 3^{b_i}$. We look up $T_{b_i} = g^{3^{b_i}}$ and, after applying the sign s_i, we multiply ${T_{b_i}}^{s_i}$ with the running result. Let $a_\Delta = a_i - a_{i+1}$ when $i < k$ and $a_\Delta = a_k$ when $i = k$. It then squares the running result a_Δ times. The algorithm 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 8 gives pseudo-code for this approach and requires $O(\max b_i)$ memory. 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 8 but with cubing and squaring appropriately swapped.

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 example, we need only store 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 $R_0 = 1_G$. In this

Algorithm 8 Computes g^n given n in 2,3 representation. Méloni [26, Section 3.2].

Input: $g \in G$, $n = \sum_{i=1}^{k} s_i 2^{a_i} 3^{b_i}$,
$s_1, ..., s_k \in \{-1, 1\}$, $a_1 \geq ... \geq a_k \in \mathbb{Z}_{\geq 0}$, $b_1, ..., b_k \in \mathbb{Z}_{\geq 0}$.

1: $T_b \leftarrow g^{3^b}$ for $0 \leq b \leq \max\{b_1, ..., b_k\}$ {by repeated cubing}
2: $R \leftarrow 1_G$
3: $i \leftarrow 1$
4: **while** $i < k$ **do**
5: $\quad R \leftarrow R \cdot {T_{b_i}}^{s_i}$ {multiply with T_{b_i} or ${T_{b_i}}^{-1}$}
6: $\quad a_\Delta \leftarrow a_i - a_{i+1}$
7: $\quad R \leftarrow R^{2^{a_\Delta}}$ {by squaring a_Δ number of times}
8: $\quad i \leftarrow i + 1$
9: **end while**
10: $R \leftarrow R \cdot {T_{b_k}}^{s_k}$
11: $R \leftarrow R^{2^{a_k}}$ {by squaring a_k number of times}
12: **return** R

example, we compute

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

as depicted by figure 3.1. The result of the computation is $R_4 = g^{23814216}$.

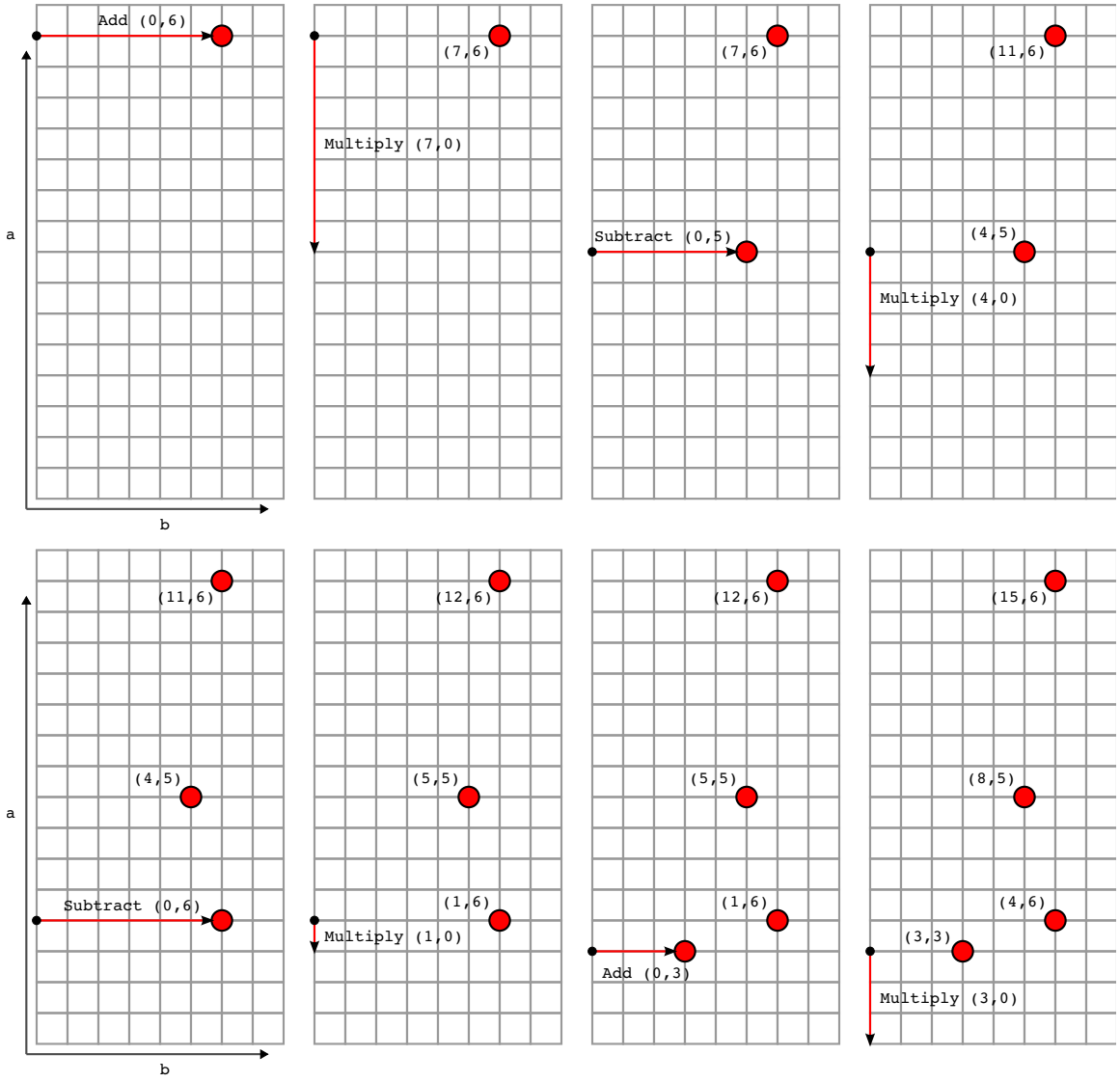

Figure 3.1: The construction of $2^{15}3^6 - 2^8 3^5 - 2^4 3^6 + 2^3 3^3$ using algorithm 8. Steps are executed from left-to-right, top-to-bottom.

3.4 Methods for Computing 2,3 Chains/Representations

In this section, we discuss 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, and 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, we can precompute the chain or representation. None of the methods presented here take into account the relative cost of multiplying, squaring, or cubing ideals. In Chapter 5 we will look at some variations of these algorithms 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 [4]. 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

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

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 $n = 0$.

We extend this concept 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 $n \equiv 5$

(mod 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

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

Algorithm 9 implements 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 7 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

Algorithm 9 Computes a 2,3 strictly chained representation from low order to high order. Ciet [4].

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**
    $n \leftarrow n/2, a \leftarrow a + 1$
  end while
  while $n \equiv 0 \pmod 3$ **do**
    $n \leftarrow n/3, b \leftarrow b + 1$
  end while
  if $n \equiv 1 \pmod 3$ **then**
    $n \leftarrow n - 1, s \leftarrow 1$
  else if $n \equiv 2 \pmod 3$ **then**
    $n \leftarrow n + 1, s \leftarrow -1$
  end if
  $(s_i, a_i, b_i) \leftarrow (s, a, b)$
  $i \leftarrow i + 1$
end while
$k \leftarrow i$
return $(a_1, b_1, s_1), ..., (a_k, b_k, s_k)$

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 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 [2], 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 $\left||n| - 2^a3^b\right|$ and $s = -1$ when $n < 0$ and $s = 1$ otherwise. A recursive function to greedily compute a 2,3 representation 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 $0 \leq a \leq A$, $0 \leq b \leq B$, 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 pseudocode in Algorithm 10. 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 7.

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

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

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

Algorithm 10 Greedy left to right representations. Berthé and Imbert [2].

Input: $n, A, B \in \{\mathbb{Z}_{\geq 0}, +\infty\}$ $\{+\infty$ for unbounded a or $b\}$

1: $L \leftarrow$ empty list
2: **while** $n \neq 0$ **do**
3: compute integers a and b that minimize $\left||n| - 2^a 3^b\right|$ such that $0 \leq a \leq A$ and $0 \leq b \leq B$
4: $s \leftarrow -1$ when $n < 0$ and 1 otherwise
5: push (s, a, b) onto the front of L
6: optionally set $(A, B) \leftarrow (a, b)$ when a chain is desired
7: $n \leftarrow n - s2^a 3^b$
8: **end while**
9: **return** L

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 [2] 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 5.

3.4.3 Pruned Tree of ±1 Nodes

The next technique for finding strictly chained 2,3 partitions was suggested by Doche and Habsieger [14]. 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 11. 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.

Algorithm 11 Chain from ±1 Pruned Tree (Doche and Habsieger [14]).

Input: $n \in \mathbb{Z}_{>0}$ and a bound $L \in \mathbb{Z}_{>0}$.

```
T ← a binary tree on the node n
while no leaf is 1 do
    for all leaf nodes v ∈ 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
    end for
    discard any duplicate leaves
    discard all but the smallest L leaves
end while
return the chained 2,3 partition generated by the path from the root to the first leaf node containing 1
```

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.

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 [19] 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 Coming up

In this chapter, we outlined some exponentiation techniques from the literature. We started with binary exponentiation based on a binary representation of the exponent. Next we introduced non-adjacent form that uses a signed base 2 encoding of the exponent. Since cubing is often faster than combined multiplication with squaring, we introduced 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 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 in Chapter 5, we introduce 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. We let the actual performance of these variations guide our implementation of a factoring algorithm called "SuperSPAR". In the next chapter, we introduce the background for SPAR and the SuperSPAR factoring algorithm.

Chapter 4

SuperSPAR

A 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. In Chapter 2 we introduce the ideal class group, and in Chapter 3 we introduce methods for exponentiation in generic groups. In this chapter, we make a connection between the two and that of integer factoring. In Section 4.1 we introduce an algorithm, called SPAR, that uses a group isomorphic to the ideal class group to factor an integer associated with the discriminant. In Section 4.2, we discuss the primorial steps algorithm for order finding in generic groups that is asymptotically faster than both Pollard's rho method [28] and Shank's baby-steps giant-steps technique [33]. Finally, in Section 4.3 we reconsider 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 [31], but was independently discovered by Atkin and Rickert who named it SPAR after Shanks, Pollard, Atkin, and Rickert [21, p.182].

4.1.1 Ambiguous Forms and the Factorization of the Discriminant

The description of SPAR uses binary quadratic forms. A binary quadratic form is a quadratic form in the variables x and y such that

$$f(x, y) = ax^2 + bxy + cy^2$$

where a, b, and c are integer coefficients. For a given form there is a set of integers represented by $f(x, y)$ for integers x and y. Two forms are equivalent if the sets of integers they represent are equivalent [9, pp.239-240], and these sets are equivalent if and only if there exists an invertible integral linear change of variables that transforms the first form into the second form. Necessarily, two equivalent forms have the same discriminant, which is given as $\Delta = b^2 - 4ac$. The set of all equivalent forms for a given discriminant form an equivalence class, and as shown by Gauß, representatives of equivalence classes of forms can be multiplied together to form a group, $G(\Delta)$. In the case of a negative discriminant, each form is equivalent to a unique reduced form [9, p.241].

The group of equivalence classes of binary quadratic forms is isomorphic to the ideal class group of imaginary quadratic fields, as shown by Frölich and Taylor [15]. Throughout this thesis, we represent elements of the ideal class group using reduced representatives. We denote the equivalence class $[\mathfrak{a}]$ for a reduced representative $\mathfrak{a}$ using the $\mathbb{Z}$-module $\mathfrak{a} = [a, (b + \sqrt{\Delta})/2]$. In our implementation we also maintain a third term $c = (b^2 - \Delta)/4a$. We note that the variables a and b are the same as the representative binary quadratic form $ax^2 + bxy + cy^2$ with discriminant Δ. Furthermore, both necessarily share the same value c. As such, we adapt our discussion of the SPAR factoring algorithm to the language of ideal classes.

Definition 4.1.1. The *ambiguous classes* are the classes $[\mathfrak{a}]$ such that $[\mathfrak{a}]^2$ is the identity class [31]. This holds for both the equivalence class of forms and the equivalence class of

ideals. Notice that the identity ideal class $[\mathcal{O}_\Delta] \in Cl_\Delta$ is an ambiguous class.

According to [31], 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. We have 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, let $\Delta = -N$ when $-N \equiv 1 \pmod 4$ and $\Delta = -4N$ otherwise. This way, $\Delta \equiv 0, 1 \pmod 4$ and $\Delta < 0$. Now, to find a factor of N we find a reduced ambiguous class representative with discriminant Δ. If we are lucky, we get a proper factor of N. If we are unlucky we get the trivial factorization $1N = N$ or a factor 2 or 4 of $4N$. In which case, we try again.

4.1.2 SPAR Algorithm

In Chapter 2, we mentioned that for a negative discriminant $\Delta < 0$, 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 m such that $[\mathfrak{a}]^m = [\mathcal{O}_\Delta]$. We say that m is the *order* of the element $[\mathfrak{a}]$ and denote this $m = \text{ord}([\mathfrak{a}])$. When the order is even, then $[\mathfrak{b}] = [\mathfrak{a}]^{m/2}$ is an ambiguous ideal class. This follows from the fact that $[\mathfrak{b}]^2 = [\mathcal{O}_\Delta]$. Therefore, factoring an integer N reduces to the problem of determining the order of a random ideal class $[\mathfrak{a}] \in Cl_\Delta$ for Δ a square free multiple of $-N$ or $-4N$.

The SPAR algorithm works in two stages. The first stage is to pick a random element $[\mathfrak{a}] \in Cl_\Delta$ and exponentiate it to the product of many small odd prime powers E, such

that $[\mathfrak{b}] = [\mathfrak{a}]^E$. We then attempt to find an ambiguous ideal by repeated squaring of $[\mathfrak{b}]$. If this fails, we perform a random walk on the group generated by the ideal class $[\mathfrak{b}] = [\mathfrak{a}]^E$. We limit the number of class operations performed by each stage, and if both stages fail to find a factorization of N, we try again with a square free multiple $\Delta = -kN$ or $\Delta = -4kN$. Introducing the multiplier k changes the group structure and order of elements – the hope is to find a group with smooth order.

Definition 4.1.2. An integer, x, is *smooth* with respect to a factor base B, if x is the product of elements from the factor base B. Typically B is chosen to be the first t primes such that $B = \{p_1, p_2, ..., p_t\}$ for some value of t.

Following Schnorr and Lenstra [31], we take 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 $e_i = \max\{v : {p_i}^v \leq {p_t}^2\}$ and compute

$$[\mathfrak{b}] = [\mathfrak{a}]^{\prod_{i=2}^{t} {p_i}^{e_i}}.$$

Notice that we exponentiate $[\mathfrak{a}]$ to the product of only *odd* prime powers. The reason for this is that if $\mathrm{ord}([\mathfrak{a}])$ is smooth with respect to the factor base $B = \{{p_i}^{e_i} : 1 \leq i \leq t\}$, then we can compute $[\mathfrak{b}]^{(2^k)}$ for the smallest k such that $[\mathfrak{b}]^{(2^k)} = [\mathcal{O}_\Delta]$. It follows that $[\mathfrak{b}]^{(2^{k-1})}$ is an ambiguous ideal class and we can attempt to factor N.

Since we do not know if $\mathrm{ord}([\mathfrak{a}])$ is smooth, we instead bound k such that 2^k is no larger than the number of elements in the class group. Schnorr and Lenstra [31, p.291] use $k = \left\lfloor \log_2 \sqrt{N} \right\rfloor$. As such, we only need compute $[\mathfrak{b}]^{(2^k)}$ for the smallest $k \leq \left\lfloor \log_2 \sqrt{N} \right\rfloor$ such that $[\mathfrak{b}]^{(2^k)} = [\mathcal{O}_\Delta]$ if such a k exists. If such a k does not exist, then the algorithm continues with the second stage.

In the first stage, we compute $[\mathfrak{b}] = [\mathfrak{a}]^{\prod_{i=2}^{t} {p_i}^{e_i}}$ and $[\mathfrak{c}] = [\mathfrak{b}]^{(2^k)}$. The second stage is a random walk through the cyclic group generated by the ideal class $[\mathfrak{c}]$ in an attempt to find the order $h = \mathrm{ord}([\mathfrak{c}])$. 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 representative 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 $[\mathfrak{c}_k] = [\mathfrak{c}]^v$. The order of $[\mathfrak{c}]$ is then a multiple of $h = v - u$. We compute an ambiguous class representative by computing $[\mathfrak{d}] = [\mathfrak{b}]^h$ and then computing $[\mathfrak{d}]^{\left(2^k\right)}$ for the smallest $k \leq \left\lfloor \log_2 \sqrt{N} \right\rfloor$ as before. Assuming that such a k exists, then $[\mathfrak{d}]^{\left(2^{k-1}\right)}$ is an ambiguous class representative and we attempt to factor N.

4.1.3 SPAR Complexity

The original publication of SPAR by Schnorr and Lenstra [31] 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 [25].

In the first stage of the algorithm, we exponentiate a random ideal class $[\mathfrak{a}] \in Cl_\Delta$ to the product of primes $\prod_{i=2}^{t} {p_i}^{e_i}$ such that $p_t = N^{1/2r}$ where $r = \sqrt{\ln N / \ln\ln N}$. Using binary exponentiation, this will take $O(p_t)$ group operations and for a random composite $m \in [0, N]$ will factor m with probability $\geq r^{-r}$ [31, p.290]. Stage 2 performs a random walk of at most $O(p_t)$ group operations and with probability $\geq (r-2)^{-(r-2)}$ will factor m [31, p.290]. Their claim is that if Stage 1 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 [25, §11]. In short, the original assumption was that for fixed N and variable k, that the class number (h_Δ for $\Delta = -kN$) was as likely to be smooth with respect to some largest prime 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. However, as Lenstra and Pomerance show [25, §11], this assumption is incorrect for a sufficiently dense sequence of integers.

4.2 Primorial Steps

The Bounded Primorial Steps algorithm [36] is an order finding algorithm for generic groups with asymptotic complexity $o(\sqrt{N})$ where N is 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 [3] and Shanks' baby-step giant-step method [33], both of which have complexity $O(\sqrt{N})$.

4.2.1 Shank's Baby-Step Giant-Step Method

The idea is similar to Shanks' baby-step giant-step method, which is as follows. For some element α in a group G and some bound M on the order of the element α, let $b = \left\lceil\sqrt{M}\right\rceil$ and compute $\alpha^1, \alpha^2, ..., \alpha^b$ storing each (α^i, i) in a table – these are the baby steps. If any $\alpha^i = 1_G$, then we know the order of α. Otherwise compute $\alpha^{2b}, \alpha^{3b}, ...$ and for each α^{jb}, if α^{jb} is in the table then $\alpha^{jb} = \alpha^i$ for some i and $jb - i$ is a multiple of the order of α. These are the giant steps. Notice that we choose b such that after an equal number of baby steps and giant steps, the last giant step has an exponent $b^2 \geq M$.

To see that this works, consider the order of the element α. Let $h = \mathrm{ord}(\alpha)$. If $h \leq b$ then we find some $\alpha^i = 1_G$ with $i \leq b$ during the baby steps. Otherwise, $h = jb + i$ for some integer j and $i < b$, and we find $\alpha^{jb} = \alpha^i$, which implies $\alpha^{jb-i} = 1_G$.

Following Sutherland [36, p.50], we point out that if computing the inverse of an element is cheaper than multiplication, we can reduce the number of group operations by letting $b = \left\lceil\sqrt{M/2}\right\rceil$ and computing the giant steps $\alpha^{2b}, \alpha^{-2b}, \alpha^{4b}, \alpha^{-4b}, ...$ instead.

Again, if the order $h \leq b$, we find some $\alpha^i = 1_G$ during the baby steps. Otherwise, either $h = 2jb - i$ or $h = 2jb + i$ for $i \leq b$. In the first case, we find some $\alpha^{2jb} = \alpha^i$ and we have $\alpha^{2jb-i} = 1_G$, and in the second case, we find some $\alpha^{-2jb} = \alpha^i$ and we have $\alpha^{2jb+i} = 1_G$.

4.2.2 Primorial Steps Algorithm

Sutherland observed [36, p.56] that if $h = \text{ord}(\alpha)$ is odd, we need only compute odd powers $\alpha^1, \alpha^3, ..., \alpha^{b-1}$ for the baby steps. We still need to compute giant steps $\alpha^{2b}, \alpha^{3b}, ...,$ for some b that is even since we want to find some $\alpha^{jb} = \alpha^i$ where $jb - i$ is odd. As before, we choose b such that after an equal number of baby steps and giant steps, the last giant step has an exponent $b^2 \geq M$. In this case, we choose $b = \left\lceil \sqrt{2M} \right\rceil$ so that after roughly $\sqrt{M/2}$ baby steps and $\sqrt{M/2}$ giant steps, the last giant step has exponent $b^2 \geq M$.

The problem is that $\text{ord}(\alpha)$ may not be odd. However, by repeated squaring of α it is easy to find an element whose order is guaranteed to be even [36, 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 our modified algorithm on β to find $h' = \text{ord}(\beta)$. The order of α can be found by computing $\gamma = \alpha^{h'}$ and then repeatedly squaring γ until $\gamma^{2^k} = 1_G$ for some k. The order of α is then $2^k h'$.

We can extend this approach 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, we compute baby steps with exponents coprime to 6 and giant steps that are a multiple of 6. More generally, we choose some primorial P_n such that

$$P_n = 2 \times 3 \times \cdots \times p_n = \prod_{i=1}^{n} p_i$$

where p_i is the i^{th} prime. Let $e_i = \left\lfloor \log_{p_i} M \right\rfloor$ for $1 \leq i \leq n$ and $E = 2^{e_2} \times 3^{e_3} \times \cdots \times {p_n}^{e_n}$. Then we compute $\beta = \alpha^E$, and use baby steps coprime to P_n and giant steps that are a multiple of P_n. Following Sutherland [36, p.57], we select the largest $P_n \leq \sqrt{M}$ and then maximize m such that $m^2 P_n \phi(P_n) \leq M$ where $\phi(P_n)$ is the number of integers coprime

Algorithm 12 Primorial Steps (Sutherland [36, p.57]).

Input: $\alpha \in G$, a bound $M \geq \text{ord}(\alpha)$, and a fast order algorithm $\mathcal{A}(\alpha, E)$.

maximize P_n such that $P_n \leq \sqrt{M}$
maximize m such that $m^2 P_n \phi(P_n) \leq M$
$b = mP_n$
$E = \prod_{i=1}^{n} p_i^{\lfloor \log_{p_i} M \rfloor}$
$\beta \leftarrow \alpha^E$
for i from 1 to b where i is coprime to P_n **do**
    compute β^i and store (β^i, i) in the table {baby steps}
    if $\beta^i = 1_G$ **then**
        return $i \cdot \mathcal{A}(\alpha^i, E)$
    end if
end for
for $j = 2b, 3b, \ldots$ **do**
    if β^j is in the table **then**
        lookup i from the table using β^j {giant steps}
        $h' = j - i$
        return $h' \cdot \mathcal{A}(\alpha^{h'}, E)$
    end if
end for

to P_n given as

$$\phi(P_n) = (2-1) \times (3-1) \times \cdots \times (p_n - 1) = \prod_{i=1}^{n} (p_i - 1).$$

The bound on the largest baby step b is $b = mP_n$. We then compute $h' = \text{ord}(\beta)$ and use a fast order finding algorithm $\mathcal{A}(\alpha^{h'}, E)$ (such as one from [36, Chapter 7]) to compute $h = \text{ord}(\alpha)$. The complete Algorithm is in listing 12.

One fast order finding algorithm $\mathcal{A}(\alpha^{h'}, E)$ uses the factorization of $E = \prod {p_i}^{e_i}$. Notice that $\alpha^{h'E} = \beta^{h'} = 1_G$. The idea is to iterate on the factors of E, removing each factor p before computing $\gamma = \alpha^{h'E'}$ for the product $E' = E/p$. If $\gamma = 1_G$, then $\text{ord}(\alpha) \mid h'E'$ and we continue with the next factor in the product E'. Otherwise, p is a factor of $\text{ord}(\alpha)$ and we continue with the next factor of E. There are asymptotically faster algorithms to compute this (see Chapter 7 from [36]), however, our interest is primarily in finding an ambiguous ideal that leads to the factorization of the discriminant.

4.3 SuperSPAR

TODO: The probability that a random integer x is $x^{1/u}$ smooth is $u^{-u+o(1)}$ [36, p.81].

Here we introduce SuperSPAR – an algorithm for factoring integers and our motivation for improving the performance of exponentiation in the ideal class group. First we review SPAR to show how it relates to the primorial steps algorithm of the previous section.

In Stage 1 of SPAR, we take the first t primes $p_i \leq N^{1/2r}$ for $r = \sqrt{\ln N / \ln \ln N}$, and let $e_i = \max\{v : {p_i}^v \leq {p_t}^2\}$ (TODO: Should this be N or Δ?). We then compute

$$[\mathfrak{b}] = [\mathfrak{a}]^{\prod_{i=2}^{t} {p_i}^{e_i}},$$

for a random ideal class $[\mathfrak{a}]$ and $[\mathfrak{c}] = [\mathfrak{b}]^{\left(2^k\right)}$ for $k = \lfloor \log_2 h_\Delta \rfloor$ (where h_Δ is given by Equation 4.1). The purpose of Stage 1 is to find an ideal class $[\mathfrak{b}]$ whose order is likely not have any of the odd primes $p_2, ..., p_t$ as divisors. If the odd part of ord($[\mathfrak{b}]$) divides $\prod_{i=2}^{t} {p_i}^{e_i}$ and the order of $[\mathfrak{b}]$ is even, then computing $[\mathfrak{c}]$ produces an ambiguous ideal and we can attempt to factor the discriminant Δ.

Failing this, Stage 2 attempts to find the order of $[\mathfrak{c}]$ by performing a random walk on the cyclic group that it generates. Schnorr and Lenstra [31, p.294] suggest a Pollard-Brent Recursion [3] based on Pollard's rho method [28]. This approach will yield ord($[\mathfrak{c}]$) within $\sqrt{\pi/2} p_t$ steps if ord($[\mathfrak{c}]$) $\leq {p_t}^2$ and requires only a constant number of group elements. They also remark [31, p.298] that Shanks' baby-step giant-step method [33] could be used to deterministically find ord($[\mathfrak{c}]$) within $\sqrt{2} p_t$ group operations when ord($[\mathfrak{c}]$) $\leq {p_t}^2$, but requiring $O(p_t)$ storage. They point out that this approach can be made faster by exploiting the fact that ord($[\mathfrak{c}]$) is likely to have no prime divisors $p_1, ..., p_t$, and this leads us to the SuperSPAR factoring algorithm.

4.3.1 SuperSPAR Algorithm

SuperSPAR works by finding an ambiguous ideal that leads to the factorization of its discriminant, and we find an ambiguous ideal by applying the relevant parts of the primorial steps algorithm. Let N be the odd integer we wish to factor. We start with a square free integer k and choose a discriminant $\Delta = -kN$ or $\Delta = -4kN$ such that $\Delta \equiv 0, 1 \pmod 4$. We then find a random prime ideal class representative $[\mathfrak{a}] \in Cl_\Delta$. If we were to compute $h = \text{ord}([\mathfrak{a}])$ and h was even, we could compute an ambiguous ideal $[\mathfrak{a}]^{h/2}$, but this is more work than necessary – all we need is the ambiguous ideal and not the order.

To use the bounded primorial steps algorithm, we need some bound M on the order of the element. Cohen [6, p.295] states a bound on the number of elements in the class group Cl_Δ as

$$h_\Delta < \frac{1}{\pi}\sqrt{|\Delta|}\log|\Delta| \text{ when } \Delta < -4. \tag{4.1}$$

Let this be our bound M on the order of $[\mathfrak{a}]$. Following the primorial steps algorithm, we compute the largest primorial P_n such that $P_n \leq \sqrt{M}$. Next, we maximize m such that $m^2 P_n \phi(P_n) \leq M$, and set $b = mP_n$.

If we wanted to determine the order of $[\mathfrak{a}]$, following the primorial steps algorithm we would compute $E = \prod_{i=1}^{n} p_i^{\lfloor \log_{p_i} M \rfloor}$ and $[\mathfrak{b}] = [\mathfrak{a}]^E$. However, since we only want to find an ambiguous ideal, we instead only use the *odd* primes $p_i \leq p_n$ when computing E and $[\mathfrak{b}] = [\mathfrak{a}]^E$.

Similar to the original SPAR algorithm, we then compute $[\mathfrak{c}] = [\mathfrak{b}]^{\binom{2^k}{}}$. In this case, we use the bound on the class group h_Δ from Equation 4.1 such that $k = \lfloor \log_2 h_\Delta \rfloor$. Now we have an ideal class $[\mathfrak{c}]$ whose order is known not to have any of the primes $p_i \leq p_n$ as divisors. We compute the order of $[\mathfrak{c}]$ by taking baby steps $[\mathfrak{c}]^i$ for $1 \leq i \leq b$ where i is coprime to P_n, and giant steps $[\mathfrak{c}]^j$ for $j = 2b, -2b, 4b, -4b...$[1]. Having computed the

[1]We use this sequence of giant steps since computing the inverse in the ideal class group is essentially

order $h' = \text{ord}([\mathfrak{c}])$, we compute an ambiguous ideal by repeated squaring of $[\mathfrak{d}] = [\mathfrak{b}]^{h'}$ assuming that $\text{ord}([\mathfrak{b}])$ is even. This works since $[\mathfrak{b}] = [\mathfrak{a}]^E$ and E has no factors of 2. We have

$$[\mathfrak{c}]^{h'} = [\mathfrak{b}]^{h'\left(2^k\right)} = [\mathfrak{d}]^{\left(2^k\right)} = [\mathcal{O}_\Delta]$$

and for some $k' < k$ it holds that $\left([\mathfrak{d}]^{k'}\right)^2 = [\mathcal{O}_\Delta]$ and $[\mathfrak{d}]^{k'}$ is an ambiguous ideal. If our initial prime ideal class representative $[\mathfrak{a}]$ has odd order, then we try again with a different prime ideal, or a different square free multiplier k.

In practice, the efficiency of the SuperSPAR factoring algorithm partially depends on the smoothness of the order of the ideal class $[\mathfrak{a}]$ with respect to the power primorial E. The larger the divisor that $\text{ord}([\mathfrak{a}])$ and E have in common, the smaller the order of $[\mathfrak{c}] = [\mathfrak{a}]^{\left(2^k\right)E}$ and the smaller the bound b we need for the baby steps and giant steps. Changing the square free multiplier k for the discriminant Δ changes the class group, and in the new class group elements may have orders that are smoother (TODO: cite something). As such, we trade off the number of ideal classes and multipliers k against the largest prime used in the power primorial E.

Furthermore, Schnorr and Lenstra [31, p.293] advise the use of smaller exponents $e_i' = \max\{v : {p_i}^v \leq p_t\}$ for the first t primes $p_i \leq N^{1/2r}$ when computing the exponent E in practice. As the exponent e_i in the prime power ${p_i}^{e_i}$ increases, the likelihood of a random integer being divisible by that prime power decreases. As such, it is generally more useful to use larger primes in the exponent E than larger prime powers. For this reason, we bound the exponents e_i of the prime powers ${p_i}^{e_i}$ separately from the bound on the largest prime used in E.

Finally, we note that selecting a bound $b = mP_n$ for the primorial steps phase implies that the number of group operations performed during this phase of the algorithm dominates over the number of group operations required to exponentiate $[\mathfrak{a}]^E$. In addition

free.

to this, as P_n grows, computing baby steps up to the bound b becomes computational intractable. Therefore, we select the bound b on the largest baby step separately. We point out that after taking $\phi(b)$ baby steps with each step coprime to b, followed by $\phi(b)$ giant steps with each step $2b$ in size, the final giant step may have an exponent less than $\text{ord}([\mathfrak{c}])$. In which case, we will not find an ambiguous ideal from our initial ideal class $[\mathfrak{a}]$ and we try again with a different ideal class $[\mathfrak{a}]$ or multiplier k on the discriminant Δ. Again, this trades off the number of group operations used when searching for an ambiguous ideal against the number of ideals tried per class group or the number of class groups used.

Assuming we know good choices for each of the variables involved, we give the complete SuperSPAR algorithm in listing 13.

4.3.2 SuperSPAR Complexity

Ideally, we want to minimize the expected number of group operations that lead to a factorization of the integer N. The parameters involved are the number of prime ideal classes $[\mathfrak{a}]$ tried before switching multipliers and class groups, the bound on the largest prime in the exponent E, a bound on the exponents e_i of the prime powers ${p_i}^{e_i}$ whose product is the exponents E, and a bound on the largest baby step b.

Algorithm 13 SuperSPAR Integer Factoring Algorithm.

Input: an integer $N \in \mathbb{Z}_{\geq 0}$ to be factored,
$E, b, \mathbb{Z}_{>0}$

1: $k \leftarrow 1$
2: $\Delta \leftarrow -kN$
3: **if** $\Delta \not\equiv 0, 1 \pmod 4$ **then**
4: $\quad \Delta \leftarrow 4\Delta$
5: **end if**
6: select $[\mathfrak{a}]$ at random {typically a prime ideal representative}
7: $[\mathfrak{b}] \leftarrow [\mathfrak{a}]^E$ {big exponentiation by power primorial}
8: **if** $[\mathfrak{b}] = [\mathcal{O}_\Delta]$ **then**
9: $\quad$ go to line 39 {no ambiguous ideal since ord($[\mathfrak{a}]$) is odd}
10: **end if**
11: $[\mathfrak{c}] \leftarrow [\mathfrak{b}]$
12: **for** $1 \leq k \leq \lfloor \log_2 h_\Delta \rfloor$ **do**
13: $\quad [\mathfrak{c}] \leftarrow [\mathfrak{c}]^2$ {compute $[\mathfrak{c}]^{\left(2^{\lfloor \log_2 h_\Delta \rfloor}\right)}$ by repeated squaring}
14: $\quad$ **if** $[\mathfrak{c}]$ is an ambiguous ideal **then**
15: $\quad\quad$ **return** attempt to factor Δ using $[\mathfrak{c}]$
16: $\quad$ **end if**
17: **end for**
18: clear the lookup table
19: **for** $1 \leq i \leq b$ where $\gcd(i, b) = 1$ **do**
20: $\quad$ compute $[\mathfrak{c}]^i$ and store $([\mathfrak{c}]^i, i)$ in the lookup table {coprime baby-steps}
21: $\quad$ **if** $[\mathfrak{c}]^i = [\mathcal{O}_\Delta]$ **then**
22: $\quad\quad h' \leftarrow i$, then go to line 32
23: $\quad$ **end if**
24: **end for**
25: **for** $1 \leq j \leq \phi(b)$ **do**
26: $\quad$ **if** $[\mathfrak{c}]^{2jb}$ is in the table **then**
27: $\quad\quad h' \leftarrow 2jb - i$, then go to line 32 {regular giant-step}
28: $\quad$ **else if** $[\mathfrak{c}]^{-2jb}$ is in the table **then**
29: $\quad\quad h' \leftarrow 2jb + i$, then go to line 32 {inverted giant-step}
30: $\quad$ **end if**
31: **end for**
32: $[\mathfrak{d}] \leftarrow [\mathfrak{b}]^{h'}$
33: **for** $1 \leq k \leq \lfloor \log_2 h_\Delta \rfloor$ **do**
34: $\quad [\mathfrak{d}] \leftarrow [\mathfrak{d}]^2$
35: $\quad$ **if** $[\mathfrak{d}]$ is an ambiguous ideal **then**
36: $\quad\quad$ **return** attempt to factor Δ using $[\mathfrak{d}]$
37: $\quad$ **end if**
38: **end for**
39: choose a different ideal class $[\mathfrak{a}]$ or a different multiplier k and try again at line 2

Chapter 5

Ideal Exponentiation Experiments

A significant contribution of this thesis is to improve the performance of exponentiation in the ideal class group of imaginary quadratic number fields. This chapter focuses on the techniques and experimental data that lead us to our implementation. We specialized much of our implementation for the x64 architecture. Many of our routines benefit when the input is bounded 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.

Improvements to ideal arithmetic also lead to improvements in ideal exponentiation, and so we start there. Arithmetic in the ideal class group uses solutions to equations of the form $s = Ua + Vb$ where s is the largest integer dividing both a and b. In Section 5.1, we introduce several algorithms for computing such solutions and look at their performance in practice. To make arithmetic in the ideal class group as efficient as possible, we specialized our implementation and discuss this implementation in Section 5.2. In Section 5.3, we compare methods to exponentiate ideal representatives to large fixed exponents that are the product of many primes, including several novel approaches to computing double-base representations.

5.1 Extended Greatest Common Divisor

In the Subsection on Fast Ideal Multiplication (2.5.3) we describe an algorithm for the simultaneous multiplication and reduction of two reduced ideal class representatives.

Much of the computational effort of this algorithm 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 (GCD) of both a and b. We refer to this computation as the *Extended GCD*.

5.1.1 The Euclidean Algorithm

The Euclidean Algorithm is an algorithm for computing the greatest common divisor of two integers and is due to Euclid. We start with 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 can reduce the number of steps of this algorithm by subtracting an integer multiple of the smaller number from the larger one.

We can extend the Euclidean Algorithm 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 14. We handle negative

a and b by using $a' = |a|$ and $b' = |b|$ as inputs 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 14 Extended Euclidean Algorithm.

Input: $a, b \in \mathbb{Z}_{\geq 0}$

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

2: **if** $t > s$ **then**

3: $\quad \begin{bmatrix} s & U & V \\ t & X & Y \end{bmatrix} \leftarrow \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} s & U & V \\ t & X & Y \end{bmatrix}$ {Swap rows. Maintain $s \geq t$.}

4: **end if**

5: **while** $t \neq 0$ **do**

6: $\quad q \leftarrow \lfloor s/t \rfloor$

7: $\quad \begin{bmatrix} s & U & V \\ t & X & Y \end{bmatrix} \leftarrow \begin{bmatrix} 0 & 1 \\ 1 & -q \end{bmatrix} \cdot \begin{bmatrix} s & U & V \\ t & X & Y \end{bmatrix}$ {Subtract q times 2^{nd} row and swap.}

8: **end while**

9: **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.

To compute the partially reduced coefficients of the product ideal from the Fast Ideal Multiplication Algorithm (3), we compute the continued fraction expansion of a/b using the recurrences

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

$$R_i = R_{i-2} - q_i R_{i-1}$$

$$C_i = C_{i-2} - q_i C_{i-1}.$$

Notice that these recurrences perform the same operation as a single step in the Extended Euclidean Algorithm, but that the initial and stopping conditions are different. As such, we call this the *Partial Extended Euclidean Algorithm*. The algorithm is listed in Algorithm 15.

Algorithm 15 Partial Extended Euclidean Algorithm.

Input: $a, b, \in \mathbb{Z}$ and a termination bound $B \in \mathbb{Z}$.

1: $\begin{bmatrix} R_0 & C_0 \\ R_1 & C_1 \end{bmatrix} = \begin{bmatrix} a & 0 \\ b & -1 \end{bmatrix}$
2: **if** $R_1 > R_0$ **then**
3: $\quad \begin{bmatrix} R_0 & C_0 \\ R_1 & C_1 \end{bmatrix} \leftarrow \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} R_0 & C_0 \\ R_1 & C_1 \end{bmatrix}$ {Swap rows. Maintain $s \geq t$.}
4: **end if**
5: **while** $R_1 > B$ **do**
6: $\quad q \leftarrow \lfloor R_0/R_1 \rfloor$
7: $\quad \begin{bmatrix} R_0 & C_0 \\ R_1 & C_1 \end{bmatrix} \leftarrow \begin{bmatrix} 0 & 1 \\ 1 & -q \end{bmatrix} \cdot \begin{bmatrix} R_0 & C_0 \\ R_1 & C_1 \end{bmatrix}$ {Subtract q times 2^{nd} row and swap.}
8: **end while**
9: **return** (R_0, R_1, C_0, C_1)

All the operations performed on the matrix

$$\begin{bmatrix} R_0 & C_0 \\ R_1 & C_1 \end{bmatrix}$$

during the Partial Extended Euclidean Algorithm can be represented by an invertible 2×2 matrix with determinant ± 1 (known as a *unimodular* matrix). This is necessarily the case since each operation relates one representative of an ideal equivalence class to another representative[1]. This turns out to be critical as some variations of the Extended Euclidean Algorithm use operations that cannot be represented by an invertible 2×2 matrix and so cannot be used to compute the partially reduced coefficients for ideal multiplication.

[1]Recall our discussion of binary quadratic forms (see subsection 4.1.1). Two forms are related if there exists an invertible linear transformation of variables.

5.1.2 Lehmer's GCD

In the previous section, each iteration subtracts a multiple, $q = \lfloor s/t \rfloor$, of the smaller number from the larger number. Derrick Henry Lehmer 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 [24].

The idea is similar to the Extended Euclidean Algorithm, only that we have an inner loop that performs an Extended GCD computation using values that fit within a single machine word. As before, start by letting

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

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 $k = 0$). Then compute $t' = \lfloor t/2^k \rfloor$ for the same value k. We then perform an Extended GCD computation on the values s' and t' but only for as long as the quotients $q' = \lfloor s'/t' \rfloor$, generated by each step of the single precision GCD computation, are equal to the quotients $q = \lfloor s/t \rfloor$, generated by a full precision GCD computation. Let

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

be our initial matrix for the single precision GCD computation. We can determine when the quotients q' differ from the quotients q by computing $q_1 = \lfloor (s' + A)/(t' + C) \rfloor$ and $q_2 = \lfloor (s' + B)/(t' + D) \rfloor$. Simply let $q' = q_1$ and q' equals q when q_1 equals q_2.

Once $q_1 \neq q_2$, the matrix

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

represents the concatenation of the operations performed during the single precision GCD. If $B \neq 0$, we combine these operation with the outer loop of the larger 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 machine word, and we use a step of the full precision GCD computation to adjust their lengths. The complete algorithm is given in Algorithm 16.

Algorithm 16 Lehmer's GCD ([24]).

Input: $a, b, \in \mathbb{Z}$ and $a \geq b > 0$.

Let m be the number of bits in a machine word.

1: $\begin{bmatrix} s & U & V \\ t & X & Y \end{bmatrix} \leftarrow \begin{bmatrix} a & 1 & 0 \\ b & 0 & 1 \end{bmatrix}$
2: **while** $t \neq 0$ **do**
3: $k \leftarrow \lfloor \log_2 s \rfloor + 1 - m$
4: $s' \leftarrow \lfloor s/2^k \rfloor$ {Shift right for most significant word.}
5: $t' \leftarrow \lfloor t/2^k \rfloor$
6: $\begin{bmatrix} A & B \\ C & D \end{bmatrix} \leftarrow \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$
7: **while** $t' \neq 0$ and $\lfloor (s' + A)/(t' + C) \rfloor = \lfloor (s' + B)/(t' + D) \rfloor$ **do**
8: $q' \leftarrow \lfloor (s' + A)/(t' + C) \rfloor$ {Single precision step.}
9: $\begin{bmatrix} s' & A & B \\ t' & C & D \end{bmatrix} \leftarrow \begin{bmatrix} 0 & 1 \\ 1 & -q' \end{bmatrix} \cdot \begin{bmatrix} s' & A & B \\ t' & C & D \end{bmatrix}$
10: **end while**
11: **if** $B = 0$ **then**
12: $q \leftarrow \lfloor s/t \rfloor$ {Full precision step.}
13: $\begin{bmatrix} s & U & V \\ t & X & Y \end{bmatrix} \leftarrow \begin{bmatrix} 0 & 1 \\ 1 & -q \end{bmatrix} \cdot \begin{bmatrix} s & U & V \\ t & X & Y \end{bmatrix}$
14: **else**
15: $\begin{bmatrix} s & U & V \\ t & X & Y \end{bmatrix} \leftarrow \begin{bmatrix} A & B \\ C & D \end{bmatrix} \cdot \begin{bmatrix} s & U & V \\ t & X & Y \end{bmatrix}$ {Combine step.}
16: **end if**
17: **end while**

Since Lehmer's GCD performs the same operations as the Extended Euclidean Algorithm, both on full precision values and on single precision values, each operation is

unimodular, and as such, Lehmer's GCD can be adapted in a straightforward manner to a Partial Extended GCD computation.

5.1.3 Right-to-Left Binary GCD

The previous extended GCD algorithms use divide with remainder, which is often expensive. Binary GCD algorithms emphasize bit shifting over multiplication and division. Such algorithms may perform better than other GCD algorithms in practice (see Subsection 5.1.7 for results). Here we introduce a binary GCD algorithm 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. The concept was originally published by Stein [35].

To compute the greatest common divisor of two positive numbers, we can 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 can 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 can be 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, we have $\gcd(a, b) = 2 \cdot \gcd(a/2, b/2)$. So the first step of our binary GCD algorithm 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 GCD of the reduced a and b. As such, the final solution to the original input is $s2^r = Ua2^r + Vb2^r$.

As before, we begin with the matrix representation

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

Since we chose a to be odd, s is odd. An invariant of our algorithm will be that s is odd at the beginning of each iteration. As previously, we iterate until $t = 0$.

Since s is odd, we 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.

Theorem 5.1.1. 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

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

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. □

By Theorem 5.1.1, 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. We 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 17. Note that in practice, integer division by 2 is performed using a bit shift right.

Even though this algorithm performs well in practice (see Subsection 5.1.7), the operation of dividing t, X, and Y by 2 and of transforming $X + b$ and $Y - a$ to make them even cannot be expressed as unimodular matrix operations. Therefore, we cannot extend this method to computing the partially reduced coefficients for ideal multiplication.

Algorithm 17 Right-to-left Binary GCD (Based on [35]).

Input: $a, b, \in \mathbb{Z}_{>0}$.

1: let r be the largest integer such that 2^r divides both a and b
2: $a \leftarrow a/2^r, b \leftarrow b/2^r$
3: swap a and b if a is not odd
4: $\begin{bmatrix} s & U & V \\ t & X & Y \end{bmatrix} \leftarrow \begin{bmatrix} a & 1 & 0 \\ b & 0 & 1 \end{bmatrix}$
5: **while** $t \neq 0$ **do**
6: **while** t is even **do**
7: **if** Y is odd **then**
8: $(X, Y) \leftarrow (X + b, Y - a)$
9: **end if**
10: $(t, X, Y) \leftarrow \left(\frac{t}{2}, \frac{X}{2}, \frac{Y}{2}\right)$
11: **end while**
12: **if** $s \geq t$ **then**
13: $\begin{bmatrix} s & U & V \\ t & X & Y \end{bmatrix} \leftarrow \begin{bmatrix} 0 & 1 \\ 1 & -1 \end{bmatrix} \cdot \begin{bmatrix} s & U & V \\ t & X & Y \end{bmatrix}$
14: **else**
15: $\begin{bmatrix} s & U & V \\ t & X & Y \end{bmatrix} \leftarrow \begin{bmatrix} 0 & 1 \\ -1 & 1 \end{bmatrix} \cdot \begin{bmatrix} s & U & V \\ t & X & Y \end{bmatrix}$
16: **end if**
17: **end while**
18: **return** $(s2^r, U, V)$ if a and b were not swapped and $(s2^r, V, U)$ otherwise

5.1.4 Windowed Right-to-Left Binary 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 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.

In the algorithm in the previous section, we repeatedly reduce either the equation $t = Xa + Yb$ or the equation $t = (X + b)a + (Y - a)b$ by 2. When Y is odd, we can 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 $ma \equiv Y$

(mod 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 $(Y - y_j)/2^h$. We give the complete algorithm in Algorithm 18.

Algorithm 18 Windowed Right-to-left Binary 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**
    $j \leftarrow ma \bmod 2^k$
    $x_j \leftarrow mb$
    $y_j \leftarrow ma$
end for
$\begin{bmatrix} s & U & V \\ t & X & Y \end{bmatrix} \leftarrow \begin{bmatrix} a & 1 & 0 \\ b & 0 & 1 \end{bmatrix}$
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
        $j \leftarrow Y \bmod 2^h$
        $(t, X, Y) \leftarrow \left(\frac{t}{2^k}, \frac{X + x_j}{2^h}, \frac{Y + y_j}{2^h}\right)$ {Reduce by 2^h}
    end while
    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
        $\begin{bmatrix} s & U & V \\ t & X & Y \end{bmatrix} \leftarrow \begin{bmatrix} 0 & 1 \\ -1 & 1 \end{bmatrix} \cdot \begin{bmatrix} s & U & V \\ t & X & Y \end{bmatrix}$
    end if
end while
return $(s2^r, U, V)$ if a and b were not swapped and $(s2^r, V, U)$ otherwise

5.1.5 Left-to-Right Binary GCD

Just as exponentiation can be performed from high-order to low-order, so too can an extended binary GCD computation. We term this 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 [32] propose to select $q' = 2^k$ such that $q't \leq s < 2q't$. If $s - q't < 2q't - s$, we 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, we 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, we maintain the invariant that $s \geq t$; if after the above operation we have $s < t$, we then swap the rows of the matrix to restore the invariant. The complete algorithm is given in Algorithm 19.

Algorithm 19 Shallit and Sorenson Left-to-Right binary GCD [32].

Input: $a, b \in \mathbb{Z}$

1: $\begin{bmatrix} s & U & V \\ t & X & Y \end{bmatrix} \leftarrow \begin{bmatrix} a & 1 & 0 \\ b & 0 & 1 \end{bmatrix}$
2: **if** $t > s$ **then**
3: $\quad \begin{bmatrix} s & U & V \\ t & X & Y \end{bmatrix} \leftarrow \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} s & U & V \\ t & X & Y \end{bmatrix}$ {Swap rows. Maintain $s \geq t$.}
4: **end if**
5: **while** $t \neq 0$ **do**
6: $\quad$ find $q = 2^k$ such that $qt \leq s < 2qt$
7: $\quad$ **if** $s - qt < 2qt - s$ **then**
8: $\quad\quad \begin{bmatrix} s & U & V \\ t & X & Y \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & -q \end{bmatrix} \cdot \begin{bmatrix} s & U & V \\ t & X & Y \end{bmatrix}$
9: $\quad$ **else**
10: $\quad\quad \begin{bmatrix} s & U & V \\ t & X & Y \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -1 & 2q \end{bmatrix} \cdot \begin{bmatrix} s & U & V \\ t & X & Y \end{bmatrix}$
11: $\quad$ **end if**
12: $\quad$ **if** $t > s$ **then**
13: $\quad\quad \begin{bmatrix} s & U & V \\ t & X & Y \end{bmatrix} \leftarrow \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} s & U & V \\ t & X & Y \end{bmatrix}$ {Swap rows. Maintain $s \geq t$.}
14: $\quad$ **end if**
15: **end while**
16: **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.

Unlike the right-to-left binary GCD in Subsection 5.1.3, all of the operations in the left-to-right binary GCD are unimodular and so can be used to generate the partially reduced coefficients in the partial extended GCD for fast ideal multiplication.

5.1.6 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 in C99 for the GNU C compiler and often benefit from hand optimized x64 assembler. The use of assembly language is done in line and is conditionally compiled based on the target platform. For each of the GCD algorithms presented here, we implemented 32-bit and 64-bit versions, and when arithmetic overflows did not require us to use GMP, we also implemented 128-bit versions. For appropriately sized inputs, we use these specialized implementations, otherwise we use GMP for the extended GCD computation.

Many of the techniques we use are described in the book "Hacker's Delight" [37]. We use $\texttt{and}_2$ to denote bitwise 'and', $\texttt{or}_2$ for bitwise 'or', '$\oplus$' for bitwise 'exclusive or', and lastly '$\neg$' 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. When we want $x \bmod 2^k$, we 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. We can generate a bit-mask

according to this test by using an arithmetic shift right operation (in the C programming language this is a right shift on a signed integer type). 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. We denote this operation $\texttt{sign\_mask}(x)$. Notice that a word with m set bits, corresponds to the integer -1 under a signed two's complement representation.

Using these operations, we can compute the absolute value of a signed machine word x 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, we can conditionally negate a word x based on a mask y. This is useful since the extended GCD algorithms described previously expect their input, a and b, to be positive integers. We first compute the absolute value of a and b by computing their signed masks and then conditionally negate each respectively. Since the extended GCD algorithm computes a solution to $s = Ua + Vb$, where a and b have been made positive, the solution for our 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. To that effect, when an 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, we can remove the conditional instructions by simply computing the first form and then performing a conditional negation on the triple (t, X, Y) based on the signed

mask of $s - t$ before the operation.

To swap two machine words, x_0 and y_0, without using additional words, we 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.

We can conditionally swap two words, x and y 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 can conditionally swap the triples when $s < t$. The algorithm is given in Algorithm 20. On the x64 architecture we

Algorithm 20 Conditionally swap (s, U, V) with (t, X, Y) when $s < t$.

1: $d \leftarrow s - t$
2: $m \leftarrow \texttt{sign\_mask}(d)$
3: $d \leftarrow d \texttt{ and}_2 m$
4: $s \leftarrow s - d$
5: $t \leftarrow t + d$
6: $d \leftarrow (U - X) \texttt{ and}_2 m$
7: $U \leftarrow U - d$
8: $X \leftarrow X + d$
9: $d \leftarrow (V - Y) \texttt{ and}_2 m$
10: $V \leftarrow V - d$
11: $Y \leftarrow Y + d$

can 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.

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 there is an instruction (`bsr`) that returns the index of the most significant set bit. This allows us to quickly compute that value. On a platform that does not have such an instruction, we can use a logarithmic search to find the most significant set bit[2]. 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 21). 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 21 Return the index of the most significant set bit of x.

```
1: i ← 0
2: k ← ⌊m/2⌋                          {m is the number of bits in a machine word.}
3: while k ≠ 0 do
4:    if x ≥ 2^k then
5:       x ← ⌊x/2^k⌋
6:       i ← i + k
7:    end if
8: end while
9: return i
```

Finally, Lehmer's GCD algorithm is especially useful when the arguments to the GCD are several words in size. However, since our implementations are specialized up to 128-bit words, we consider 8-bit words for the inner extended GCD computation. As such, it is possible to precompute the resulting 2×2 matrix for all possible inputs into the single precision extended GCD computation. Since there are two inputs, each 8-bits in size, there are 65536 possible pairs of inputs. The coefficients of the resulting 2×2 matrix can be bound by 8-bits each, and so the table requires 256Kb of memory. This simplifies the

[2]There is also a similar algorithm to find the *least* significant set bit of a machine word (`bsl` on x64).

GCD computation since the inner loop of Lehmer's GCD becomes a lookup from this table.

5.1.7 Experimental Results

We specialized the implementation of each of the extended GCD algorithms for 32-bit and 64-bit words. When intermediate terms did not overflow 128-bit words, we also implemented 128-bit versions. For each k from 1 to 127, we generated 1,000,000 pairs of integers of k bits each, and measured the time to compute the extended GCD of each pair using each algorithm and its specialized implementation when available.

First, we present evidence that in each case, the 32-bit implementation is no slower than the 64-bit implementation, and in all but one case, always faster. Figure 5.1 shows that a 32-bit Extended Euclidean Algorithm using division with remainder is faster than a 64-bit one, for the same inputs.

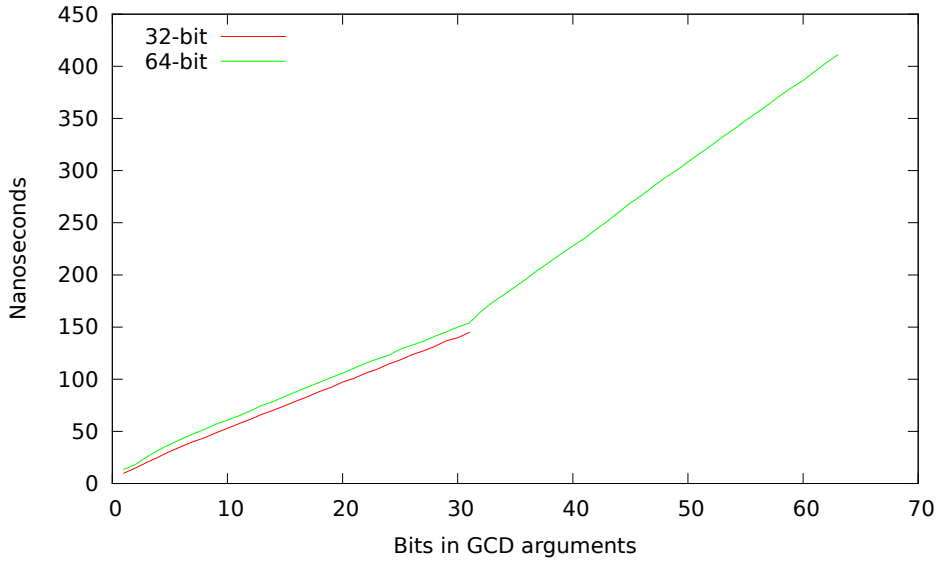

Figure 5.1: Extended Euclidean Algorithm using division with remainder.

For Lehmer's GCD, we precomputed a table of 8-bit inputs for the inner GCD computation – see Figure 5.2. In each case, the 32-bit implementation is faster than the

64-bit implementation.

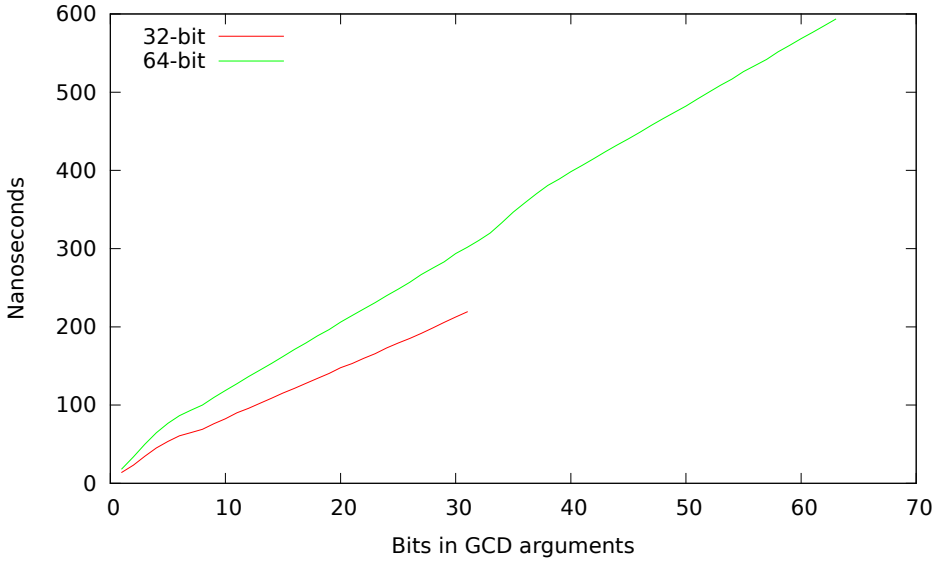

Figure 5.2: Lehmer's GCD with 8-bit precomputed inner GCD.

The graphs in Figures 5.3, 5.4, 5.5, 5.6, and 5.7 show that for all window sizes from 1 to 5 bits, that the 32-bit version of the right-to-left binary GCD is faster than the 64-bit version.

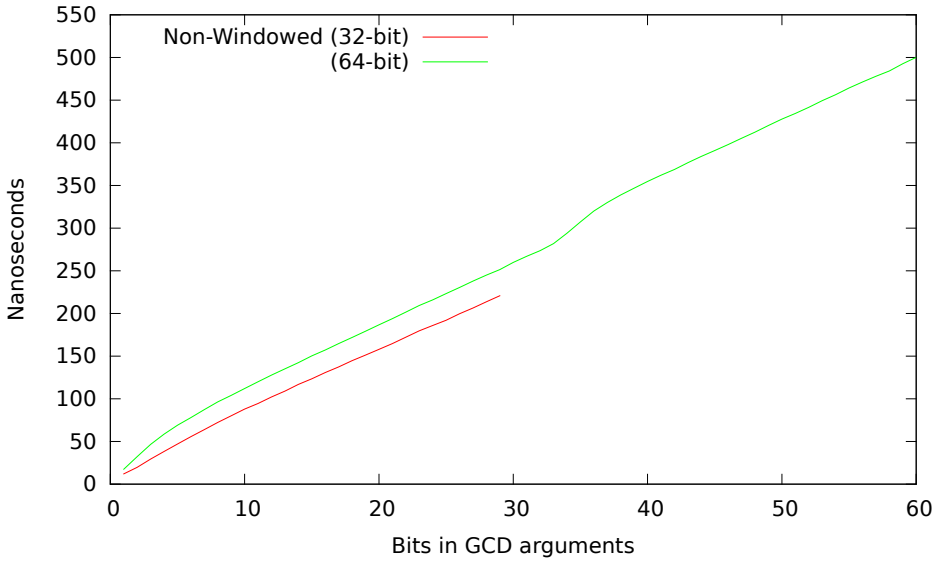

Figure 5.3: Right-to-Left Binary GCD.

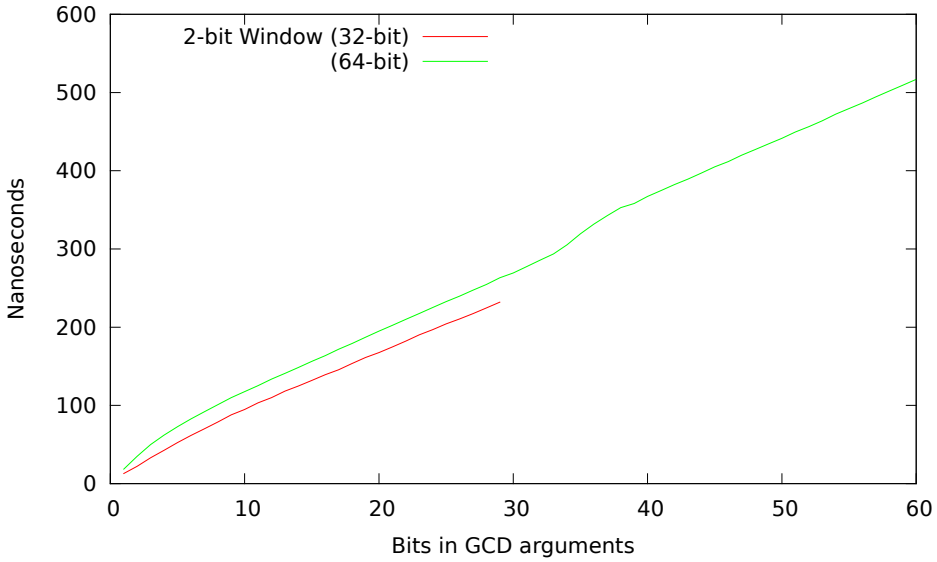

Figure 5.4: 2-bit Windowed Right-to-Left Binary GCD.

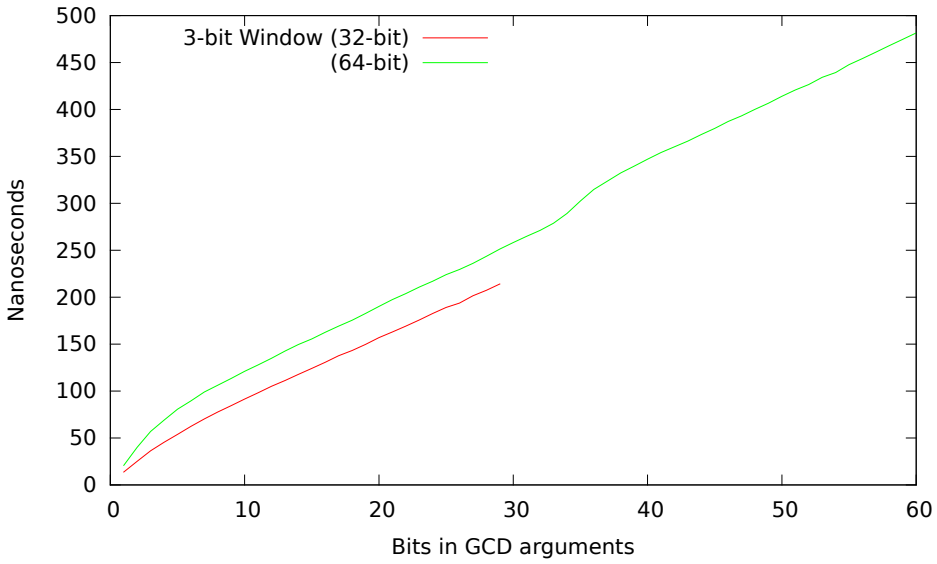

Figure 5.5: 3-bit Windowed Right-to-Left Binary GCD.

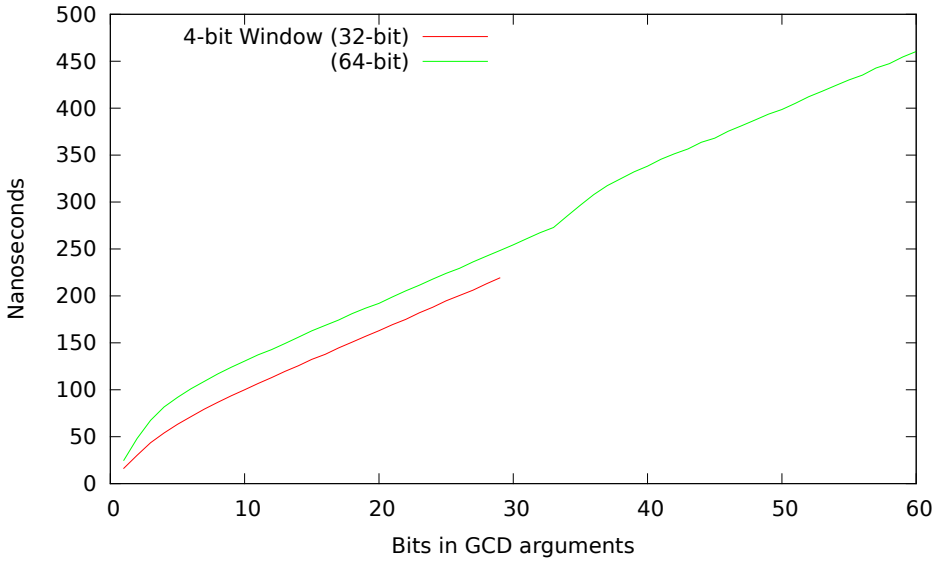

Figure 5.6: 4-bit Windowed Right-to-Left Binary GCD.

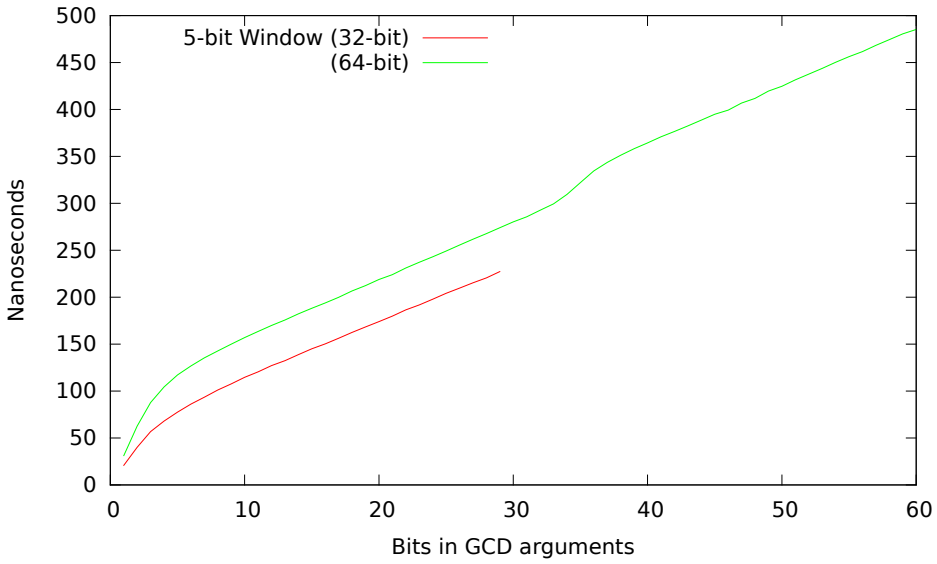

Figure 5.7: 5-bit Windowed Right-to-Left Binary GCD.

The left-to-right variant of the binary GCD proposed by Shallit and Sorenson is shown in figure 5.8. Here the 32-bit implementation is slightly faster than the 64-bit implementation.

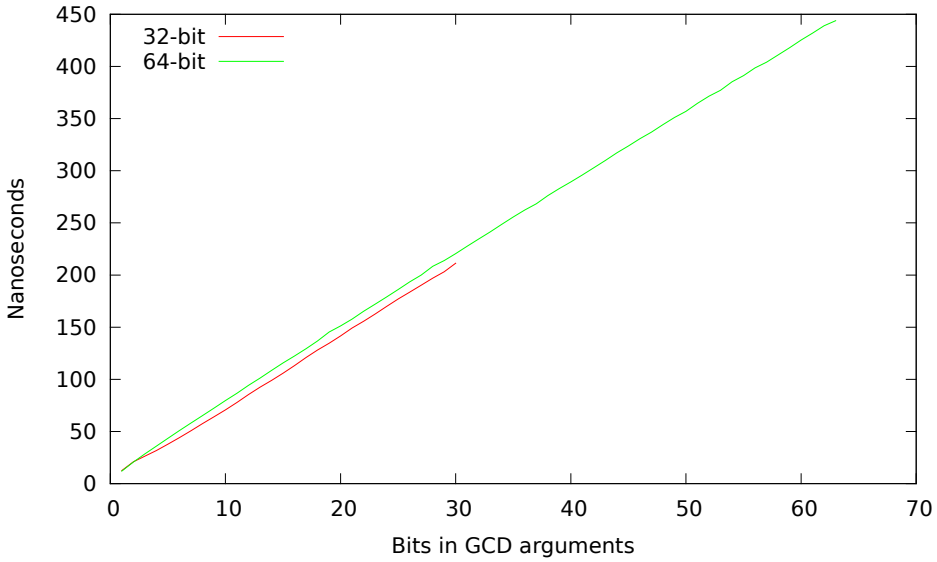

Figure 5.8: Left-to-Right binary GCD of Shallit and Sorenson.

Lastly, our simplified version of the left-to-right binary GCD is shown in figure 5.9. In this case, the benefit, if any, to using a 32-bit implementation over a 64-bit implementation is negligible[3].

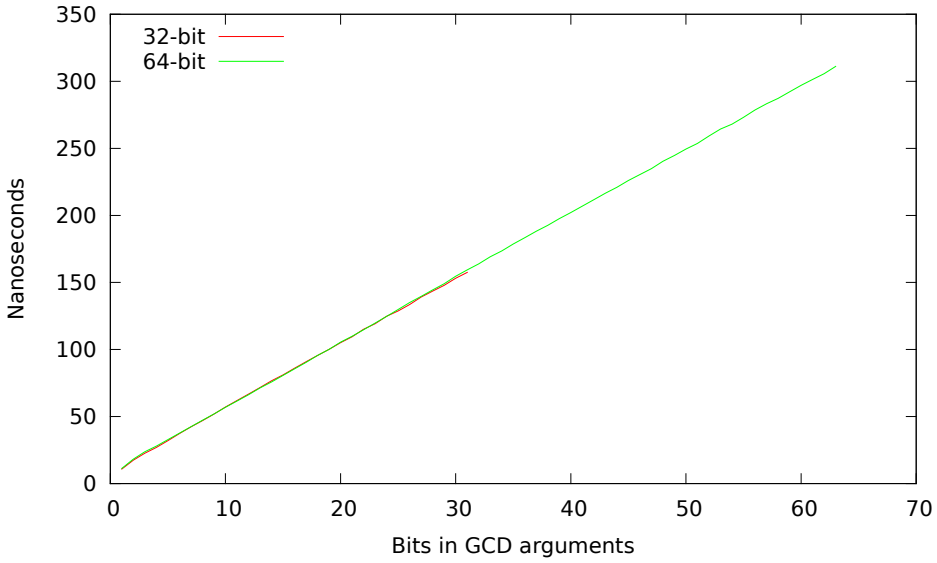

Figure 5.9: Simplified Left-to-Right binary GCD.

[3]Nanosecond differences between the two versions could be attributed to timing inaccuracies.

Note that the 32-bit versions of the right-to-left binary GCD are only accurate up to 29-bits (ignoring sign), while the 64-bit versions are only accurate to 60-bits. All other 32-bit implementations are accurate to 31-bits[4] and all other 64-bit implementations are accurate to 63-bits[5].

Since the 32-bit implementation of each algorithm is at least no slower than the 64-bit implementation, when comparing the different algorithms, we always prefer the 32-bit implementation of each when the algorithm is known to be accurate for the size of the inputs.

Both the Extended Euclidean Algorithm and Lehmer's GCD use division with remainder to compute their result, so we first compare our implementations of both. Figure 5.10 shows that the Extended Euclidean Algorithm is always faster.

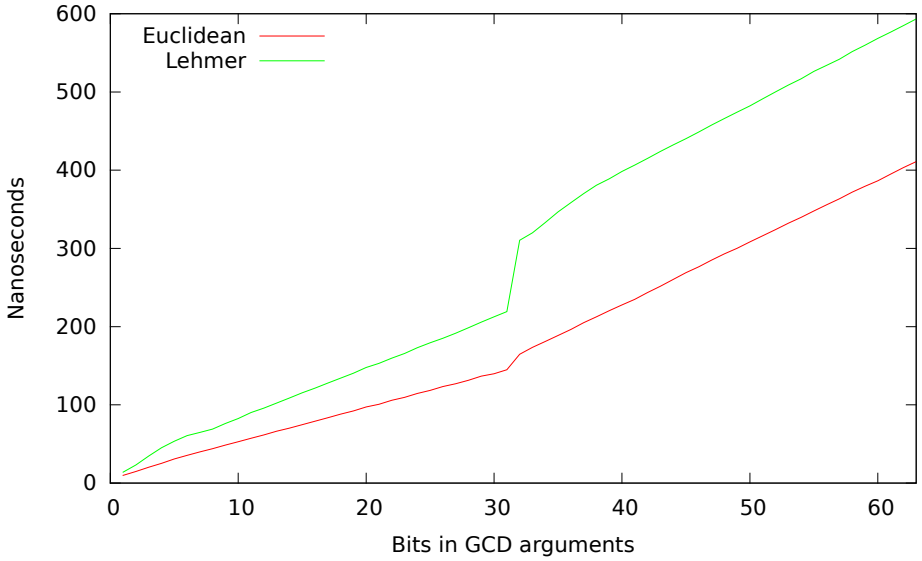

Figure 5.10: Extended Euclidean Algorithm compared to Lehmer's GCD using precomputed 8-bit words.

In Figure 5.11, we compare each of our windowed implementations of the right-to-

[4]The input is less than or equal to 31-bits, since in a two's complement representation the sign of the input variable takes the most significant bit and accounts for all 32-bits.

[5]Similar to the 32-bit implementation, the most significant bit represents the sign of the integer.

left binary GCD. Larger windows trade off precomputation against the likelihood that a number will be divisible by some power of two. You can see that the non-windowed version is most efficient for inputs less than 16-bits in size, at which point the 3-bit windowed version is best until 30 bits. After 30 bits, the 4-bit windowed version out performs other window sizes in the range with which we experimented. In the 64-bit implementation, the slope of the 5-bit windowed version is the lowest and so presumably it would out perform the 4-bit window once inputs were large enough.

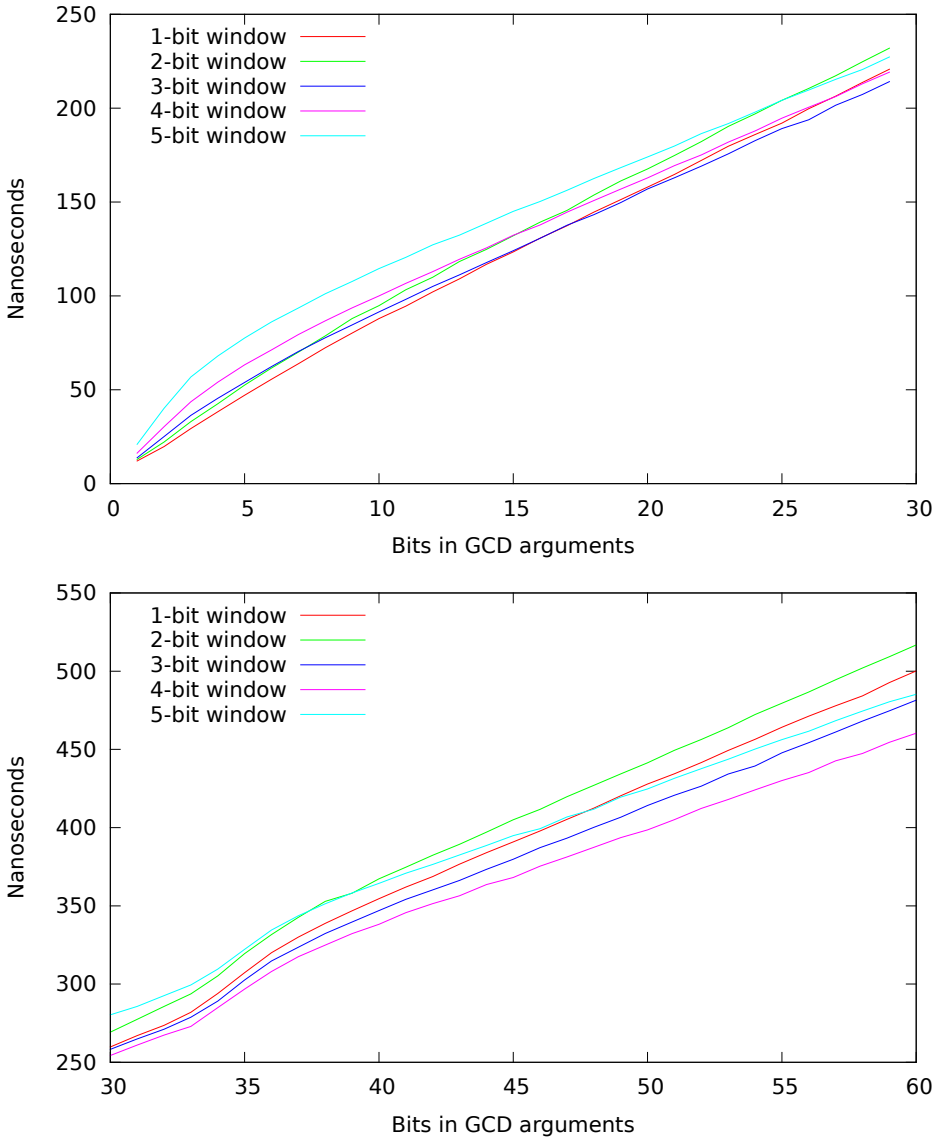

Figure 5.11: Windowed right-to-left binary GCDs.

Figure 5.12 compares the left-to-right binary GCD proposed by Shallit and Sorenson to our own simplified version.

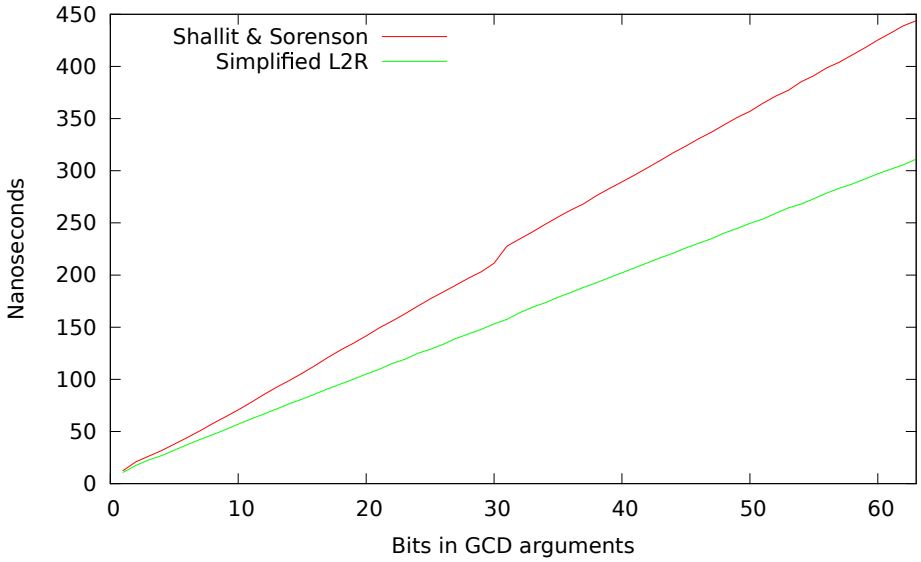

Figure 5.12: Shallit and Sorenson compared to our simplified left-to-right binary GCD.

Our implementation of the Euclidean Algorithm always out performs Lehmer's GCD. For our implementations of the right-to-left binary GCD, among the 32-bit implementations, the non-windowed and the 3-bit windowed versions perform well. For the 64-bit implementation, the 4-bit windowed version out performs the other window sizes. Finally, for the left-to-right variants of the binary GCD, our simplified version always out performs the approach proposed by Shallit and Sorenson. In Figure 5.13, we compare these top performers. For integer inputs less than 32-bits, the standard Euclidean Algorithm performs best, and for inputs between 32-bits and 64-bits, our simplified left-to-right binary GCD is fastest.

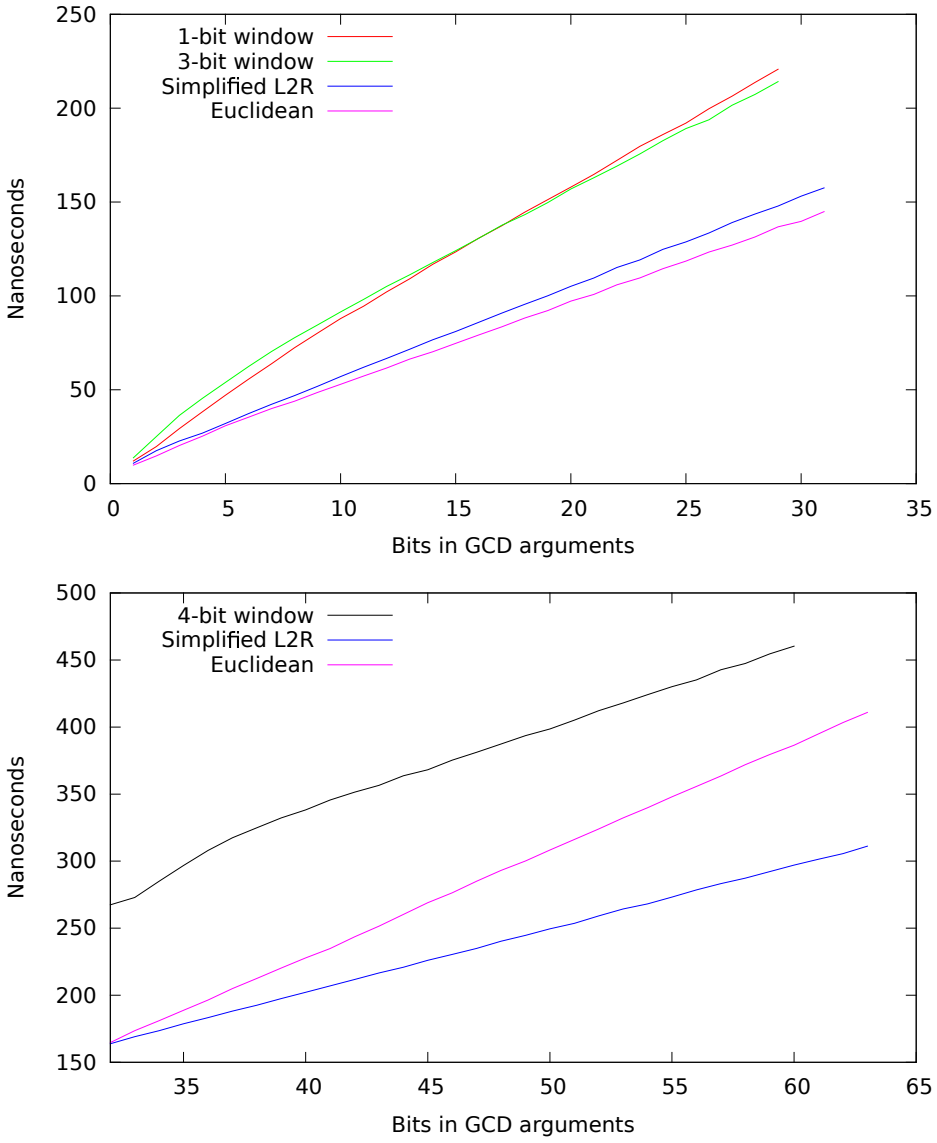

Figure 5.13: Euclidean vs windowed right-to-left binary vs Simplified left-to-right binary.

Since the Euclidean Algorithm and our simplified left-to-right binary GCD are our fastest implementations for inputs less than 64-bits, we extended both to work on 128-bit integers. We also compared these with the implementation of the extended GCD algorithm from the GNU Multiple Precision (GMP) library. Figure 5.14 show that our implementations perform faster than the implementation in GMP for inputs smaller than 64-bits.

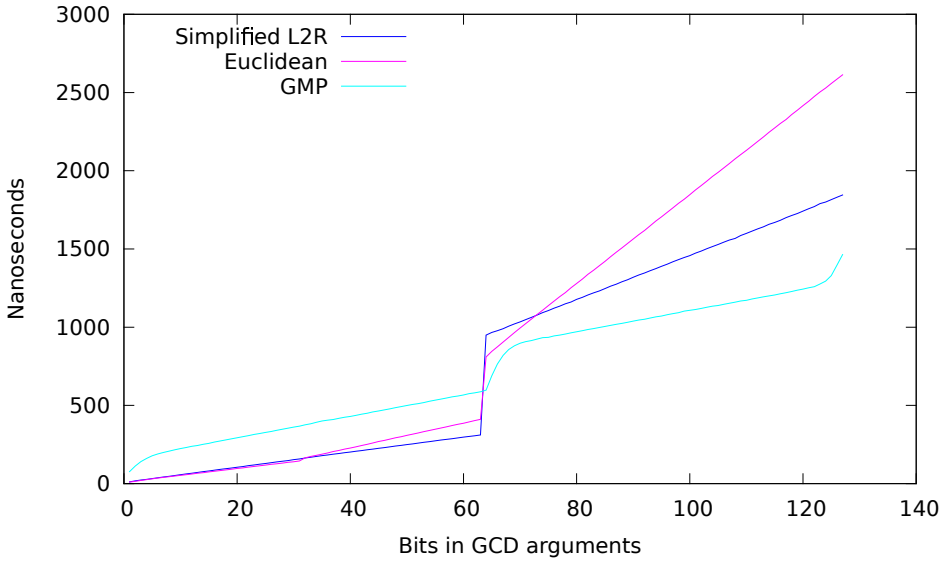

Figure 5.14: Euclidean vs our simplified left-to-right binary vs GNU Multiple Precision library.

5.2 Ideal Arithmetic

In the previous section, we demonstrated results that lead to practical improvements in computing the extended GCD – the dominating operation in reduced ideal multiplication. In this section, we discuss our implementations of ideal arithmetic. To improve the performance of ideal arithmetic, we specialized implementations to use at most a single machine word, i.e. 64-bits, or at most two machine words, i.e. 128-bits. We also wrote a reference implementation using GMP that works for unbounded discriminant sizes. Our 64-bit implementation of ideal arithmetic is accurate for negative discriminants up to 59-bits in size, while our 128-bit implementation of ideal arithmetic is accurate for discriminants up to 118-bits.

5.2.1 Specialized Implementations of Ideal Arithmetic

Throughout this thesis, we represent an ideal class $[\mathfrak{a}]$ 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[6]. For this reason, our implementation represents an ideal class using the triple (a, b, c). We also maintain a representation of the group[7], which includes the discriminant Δ.

In Subsections 2.5.3, 2.5.4, and 2.5.5 we introduced 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. In all cases, the termination bound contains the coefficient $|\Delta/4|^{1/4}$. Since the coefficients of the partial extended GCD are integers, we instead compute $\lceil |\Delta/4|^{1/4} \rceil$ as well as $\lceil |\Delta/4|^{1/2} \rceil$ and store both in our representation of the class group. The reason we store $\lceil |\Delta/4|^{1/2} \rceil$ is that the termination bound for ideal multiplication is

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

For ideal squaring, the bound is simply $|\Delta/4|^{1/4}$ and for ideal cubing the bound is $\sqrt{a_1}|\Delta/4|^{1/4} \approx \sqrt{a_1\,\lceil |\Delta/4|^{1/2} \rceil}$. By approximating the termination bound, we can speed the arithmetic at the expense of additional reduction steps. We note here that we compute the values $\lceil |\Delta/4|^{1/2} \rceil$ and $\lceil |\Delta/4|^{1/4} \rceil$ as accurately as possible, since these are only computed once per class group and then stored with the representation of the group.

We further approximate the computation of integer square roots when computing the

[6]Recall that an ideal is ambiguous if $b = 0$, $a = b$, or $a = c$.

[7]The group representation also includes all intermediate integers needed for use by the GMP library. The reason for this is that GMP integers are managed on the heap, and preallocating them alongside the representation of the group reduces the overhead of managing them during ideal class group operations.

termination bound for fast multiplication. We note that

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

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 that the discriminant $\Delta = b^2 - 4ac$. When $4ac = 0$, then b^2 is as large as possible and so we obtain the bound $b \leq \sqrt{\Delta}$. Similarly, when $b^2 = 0$ we obtain the bound $a \leq \sqrt{\Delta/4}$ (recall that by Definition 2.5.1, a reduced ideal has $a \leq c$). 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 implementations of the extended GCD algorithm. 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, 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 requires even larger intermediates – our 64-bit implementation requires some 128-bit arithmetic, while our 128-bit implementation uses GMP when necessary.

To keep intermediates small, we compute using residue classes. For example, Equations 2.5 states that we can compute $U \pmod{a_1/s}$ and Equation 2.7 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 Subsection 5.1.6). Furthermore, rather than

computing the positive remainder, we typically compute the remainder that is closest to zero. The idea here is to minimize the number of bits required by intermediates in order to avoid overflows as much as possible.

In the specific case of ideal reduction (see Algorithm 2), 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 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. (TODO: Is this method of computing c' published anywhere?)

We further improve ideal arithmetic by noting that multiplication computes

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

(this is Equation 2.4 from Subsection 2.5.2). Our implementations of ideal multiplication (Algorithm 3) performs this in two parts: the first computes $s' = \gcd(a_1, a_2) = Y'a_1 + V'a_2$ and the second only computes $s = \gcd(s', (b_1 + b_2)/2) = Vs' + W(b_1 + b_2)/2$ when $s' \neq 1$.

We do this since checking for $s \neq 1$ is relatively inexpensive when compared to performing an extended GCD computation. In addition, we note that the coefficient Y' is not used in order to derive the product ideal, and so we drop the corresponding term from our GCD computation. Lastly, we use our simplified left-to-right binary GCD to compute partially reduced coefficients of the product ideal when the operands are between 32-bits and 64-bits (since this GCD algorithm is unimodular and performed fastest in this range).

The final performance improvement is for generating prime ideals. For a given prime p, we would like to find an ideal of the form $[p, (b + \sqrt{\Delta})/2]$. We have $b^2 - 4pc = \Delta$, and so we take everything in the residue class modulo p and get $b^2 \equiv \Delta \pmod{p}$ or $b \equiv \pm\sqrt{\Delta} \pmod{p}$. Computing b is a matter of computing a square root modulo a prime, if one exists. There are efficient algorithms for this (see [1]), but since we are interested in finding some prime ideal, rather than a specific prime ideal, we instead precompute all square roots modulo each prime p for sufficiently many small p. For our purposes, all primes $p < 1000$ was enough. Finding $b \equiv \pm\sqrt{\Delta} \pmod{p}$ is now a table lookup. 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).

5.2.2 Average time for operations

In this Subsection, we demonstrate the performance of our implementations of ideal class arithmetic. 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 form, $[\mathfrak{a}]$, randomly from among the primes less than 1000. 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 $[\mathfrak{c}] \leftarrow [\mathfrak{a}\mathfrak{b}]$, $[\mathfrak{a}] \leftarrow [\mathfrak{b}]$, and $[\mathfrak{b}] \leftarrow [\mathfrak{c}]$. Figures 5.15, 5.16, and 5.17 show the results of these timings. For all discriminant sizes, our 64-bit implementation performs faster than our 128-bit implementation, which performs faster than our GMP implementation.

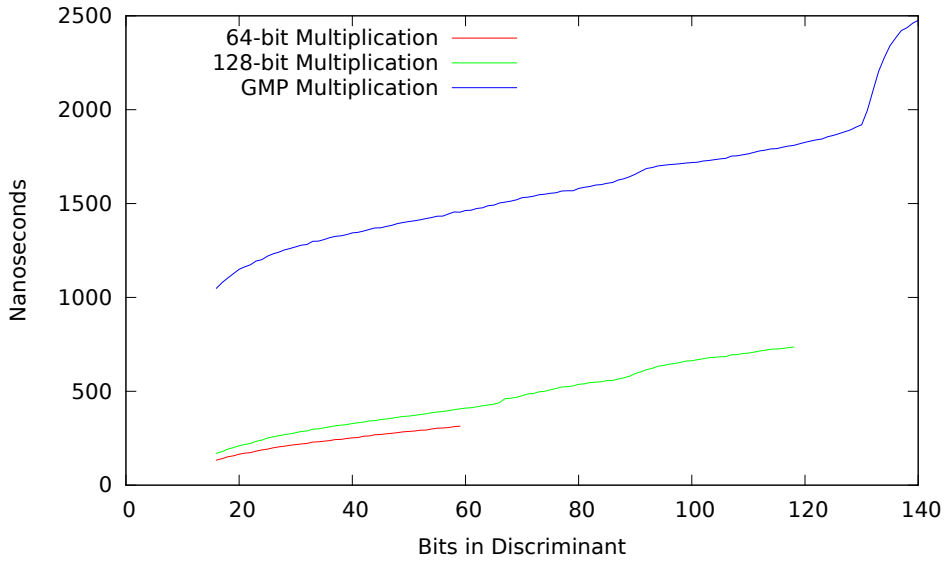

Figure 5.15: Average Time to Multiply Reduced Ideal Class Representatives.

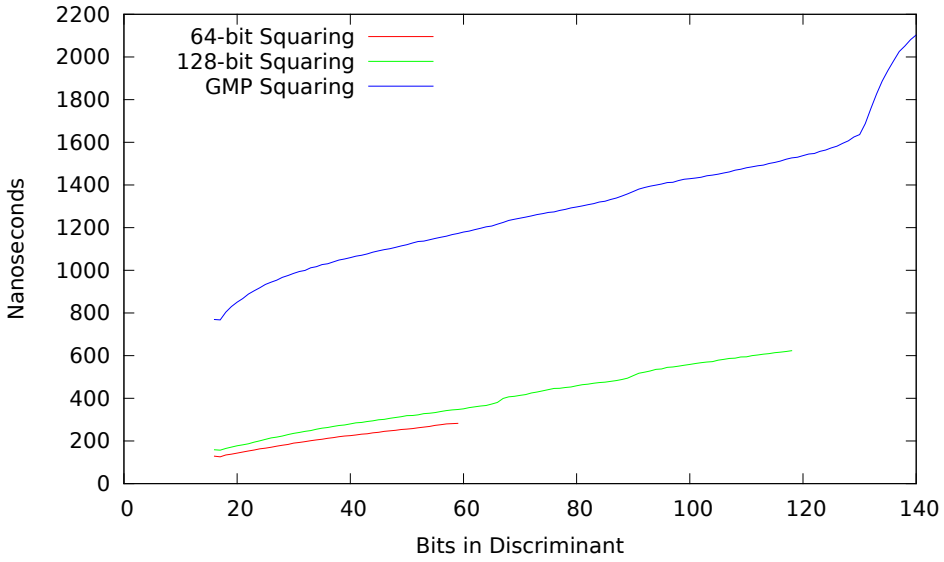

Figure 5.16: Average Time to Square Reduced Ideal Class Representatives.

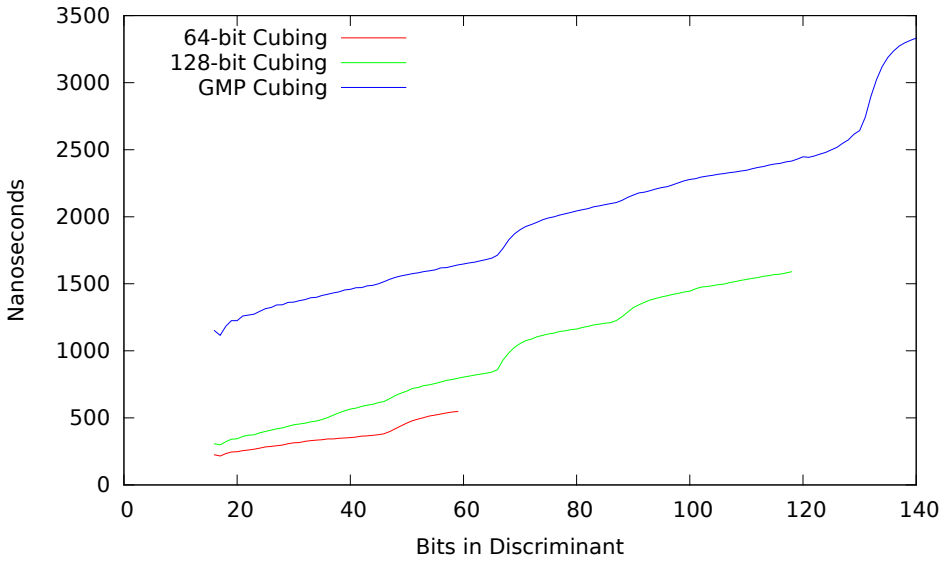

Figure 5.17: Average Time to Cube Reduced Ideal Class Representatives.

In Figures 5.18, 5.19, and 5.20 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. This is likely because we rely on GMP integers for overflow. 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.

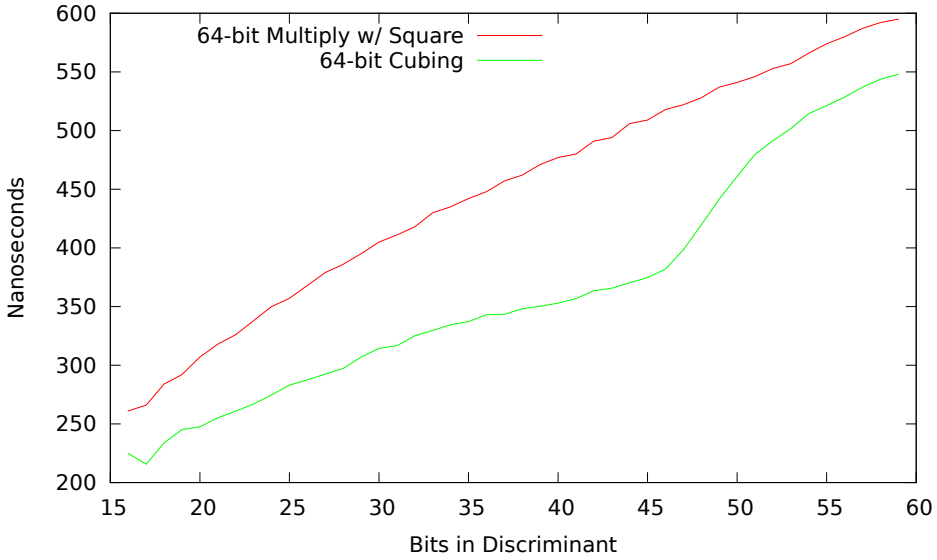

Figure 5.18: Time to compute $[\mathfrak{a}^3]$ in our 64-bit implementation.

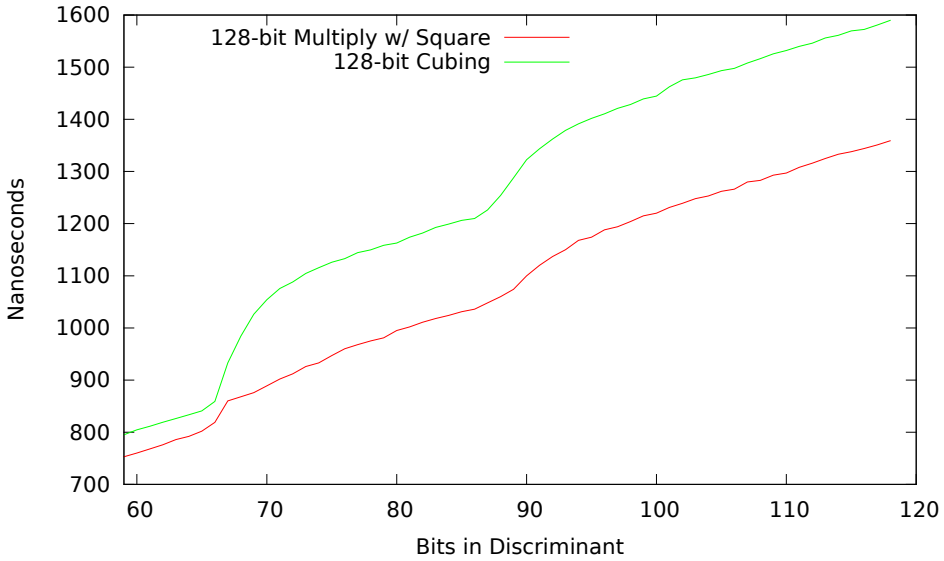

Figure 5.19: Time to compute $[\mathfrak{a}^3]$ in our 128-bit implementation.

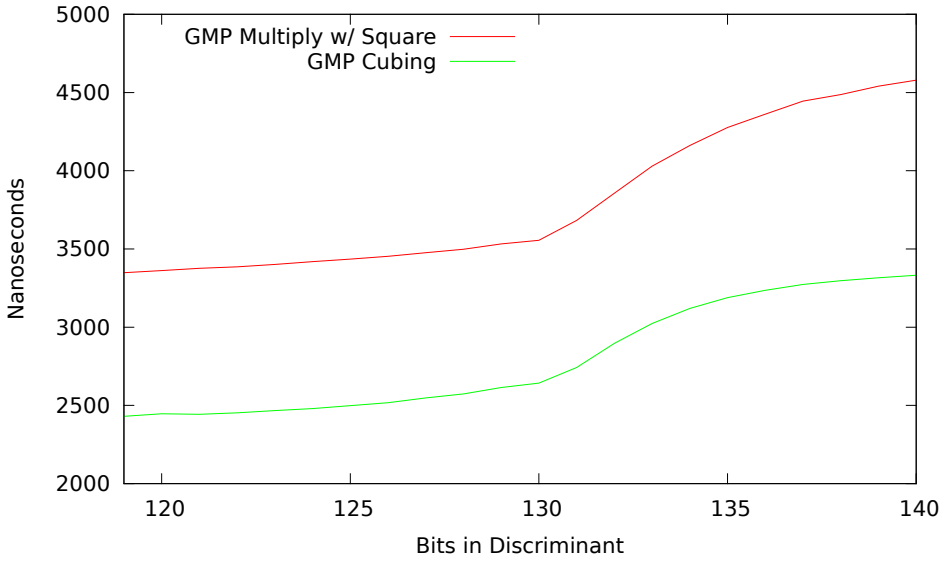

Figure 5.20: Time to compute $[\mathfrak{a}^3]$ in our GMP implementation.

In this Section we discussed methods to improve the performance of arithmetic in the ideal class group and compared the average time of multiplication, squaring, and cubing. In the next section, we look at several methods for exponentiating that we have not discussed previously – many of which take the average time of multiplication, squaring, and cubing into consideration.

5.3 Exponentiation Experiments

In this Section, we discuss several approaches to exponentiating an ideal class representative to a large exponent. In particular, we consider large exponents that are the product of all primes less than some bound[8]. We assume the exponent is known in advance, which allows us to precompute the sequence of operations for the exponentiation.

We begin by recalling exponentiation techniques from Chapter 3. These include

[8]The product of all primes less than a bound B is a *primorial*. See Section 4.2 for the definition and our motivation for using them.

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 Section, 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 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.

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

In Sections 3.1 and 3.2 we introduced binary exponentiation and non-adjacent form exponentiation. 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[9]. Similarly, when computing a non-adjacent form we can parse the exponent from left-to-right or from right-to-left. Ei-

[9]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.

ther direction, we use $\lfloor \log_2 n \rfloor + 1$ squares and $(\lfloor \log_2 n \rfloor + 1)/3$ multiplications on average.

5.3.2 Right-to-Left 2,3 Chains

In Subsection 3.4.1, we introduced 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 can use Algorithm 7 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 22).

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,

Algorithm 22 Fast n mod 3 (adapted from Hacker's Delight [37]).

Input: $n \in \mathbb{Z}$.
Output: n mod 3.

```
1: s ← 0
2: for i from 0 to ⌊log_2 n⌋ by 64 do
3:     t ← ⌊n/2^i⌋ mod 2^64
4:     s ← (s + t) mod 2^64
5: end for
6: s ← (s + ⌊s/2^32⌋) mod 2^32
7: s ← (s + ⌊s/2^16⌋) mod 2^16
8: s ← (s + ⌊s/2^8⌋) mod 2^8
9: return s mod 3                    {Lookup from a table with 256 entries.}
```

and then to use multiplication by an approximation of the inverse and then to adjust the result (see [16, 37, 27] for additional information). As the GNU Multiple Precision (GMP) library implements exact division by 3, we use this.

5.3.3 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 $1 \leq k \leq 50$. For 48 out of 50 of the primorials tested, the same strategy lead to the cheapest cost of exponentiation[10], which

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

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 $0 \leq r_4 < 4$. Then write $x = 9y + r_9$ for integers y and r_9 such that $0 \leq r_9 < 9$. 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 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[11] 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.

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

In Subsection 3.4.2 we introduce 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,

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

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 5.21 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 $B = \lceil \log_3(n/2^A) \rceil$. 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).

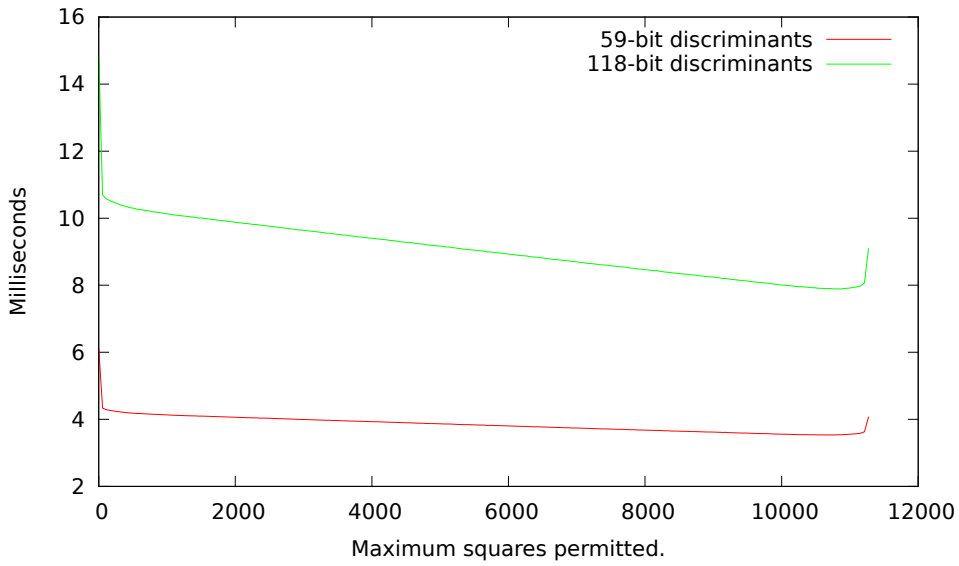

Figure 5.21: The cost to exponentiate the 1000^{th} primorial, n, while varying the maximum number of squares permitted, A. The maximum number of cubes is $B = \lceil \log_3(n/2^A) \rceil$.

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

5.3.5 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[12] partial representations. 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 [14] (covered in Subsection 3.4.3). The assumption is that by adjusting x by more than just 1, we increase the

[12] We say that a partial representation is *better* when the x term is smaller, or if the x terms are equal, when the cost of the 2,3 summation is less.

likelihood of finding a multiple of a large 2^c3^d window.

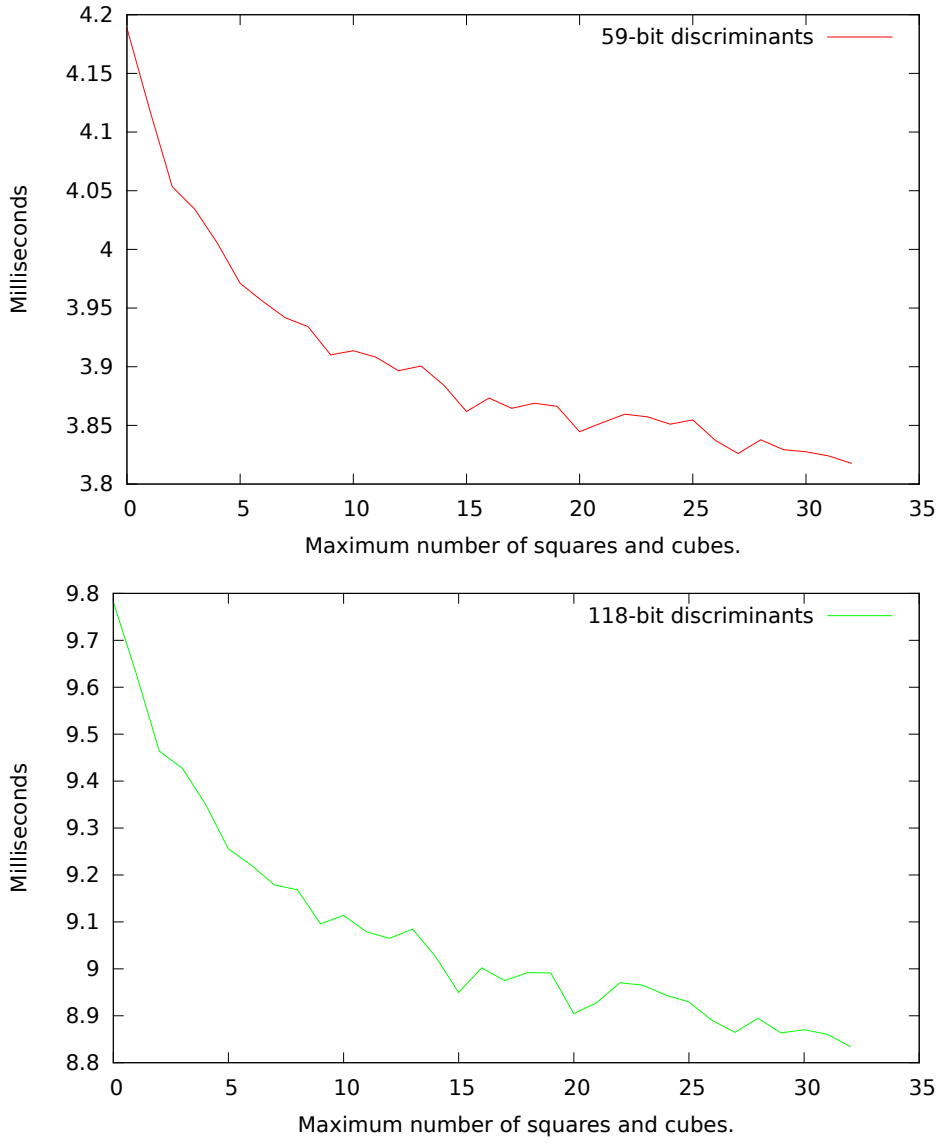

Figure 5.22: Performance of varying the bounds $a, b \leq U$ on $(x \pm 2^a3^b)/(2^c3^d)$ when exponentiating by the primorial P_{1000}.

For both variations, we bound a and b such that $0 \leq a \leq U$ and $0 \leq b \leq U$. Figure 5.22 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.

5.3.6 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 5.3.4. For an integer x we say that 2^a3^b is a best approximation of $|x|$ when $\left||x| - 2^a3^b\right|$ 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 $\left||x| - 2^a3^b\right|$ (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 $\left||x| - 2^{a_1}3^b\right|$ 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.

5.3.7 Additive 2,3 Chains

Imbert and Philippe [19] 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

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

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.

5.3.8 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 intially 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 Subsection and the two functions from the previous Subsection 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.

5.3.9 Big Exponents from Small Exponents

The techniques of the previous two Subsections 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 23 computes the exponentiation g^n using an approach similar to a left-to-right windowed binary exponentiation.

Algorithm 23 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.

1: $R \leftarrow 1_G$
2: **for** i from $\lceil \log_{2^{16}} n \rceil$ downto 0 **do**
3: $\quad R \leftarrow R^{2^{16}}$ {By repeated squaring.}
4: $\quad b_i \leftarrow \lfloor n/2^{16i} \rfloor \bmod 2^{16}$
5: $\quad R \leftarrow R \cdot g^{b_i}$
6: **end for**
7: **return** R

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 n primes), the

prime factorization is trivial. Given a primorial $P_n = \prod_{i=1}^{n} p_i$ where p_i is the i^{th} prime, we can compute g^{P_n} as $(((g^{p_1})^{p_2})^{\cdots})^{p_n}$ using a series of n small exponentiations.

5.3.10 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 5.1 shows the number of exponents for which each method was the fastest[13]. For each

Method	59-bit Discriminants	118-bit Discriminants
Left-to-Right 2,3 Representation	2536	7097
4 Best $\left\|\|x\| - 2^a 3^b\right\|$	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 5.1: The number of exponents for which each method gave the fastest average time to exponentiate.

exponent, we then normalized the time of each method relative to the best time. Table 5.2 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 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.

[13]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 5.2: The normalized time to exponentiation relative to the fastest time 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 $1 \leq k \leq 100$. 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 5.23 and 5.24 compare binary exponentiation against right-to-left non-adjacent form representation. The non-adjacent form representation leads to faster exponentia-

tions in all cases. Figures 5.25 and 5.26 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 5.27 and 5.28 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 5.29 and 5.30 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 5.31 and 5.32 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 5.33 and 5.34 compare the best performer from each category.

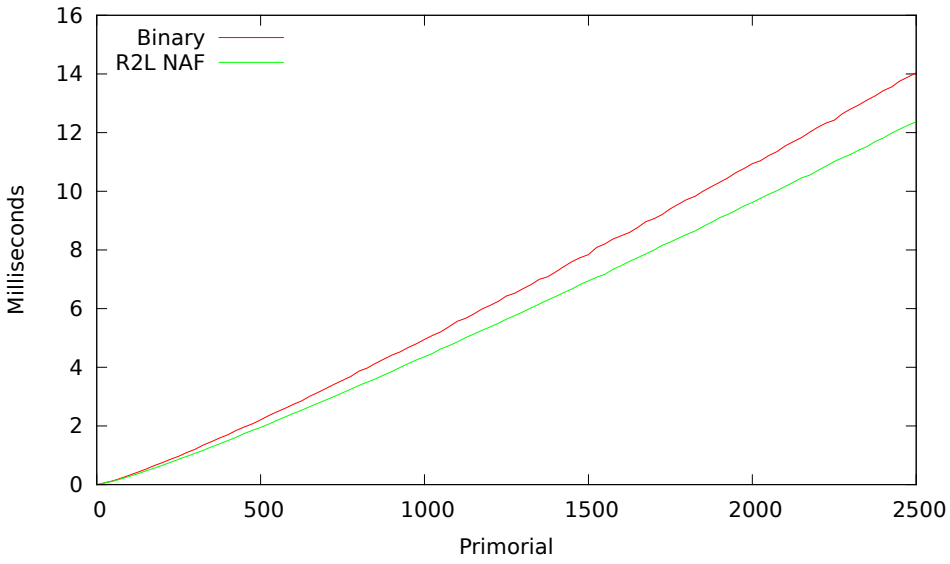

Figure 5.23: Base 2 Exponentiation (59-bit Discriminants).

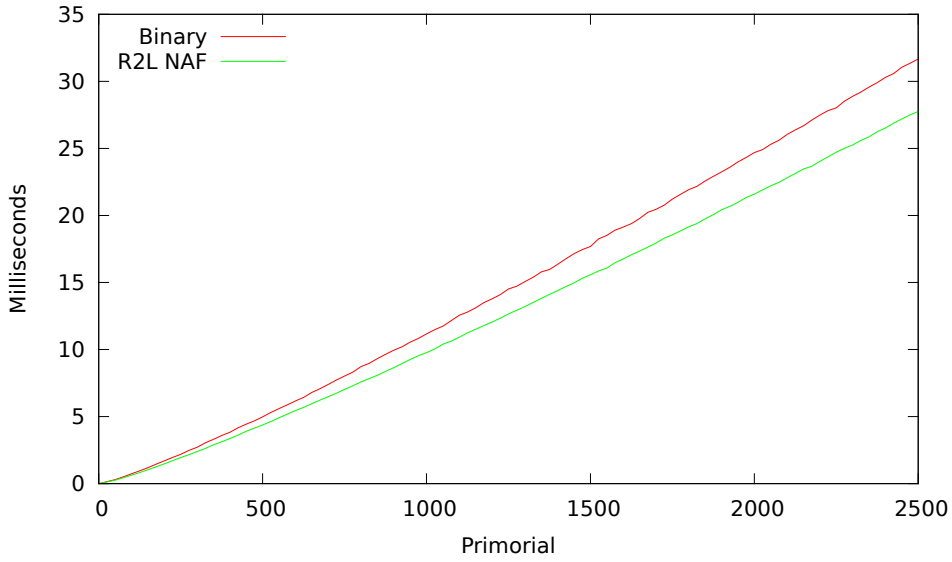

Figure 5.24: Base 2 Exponentiation (118-bit Discriminants).

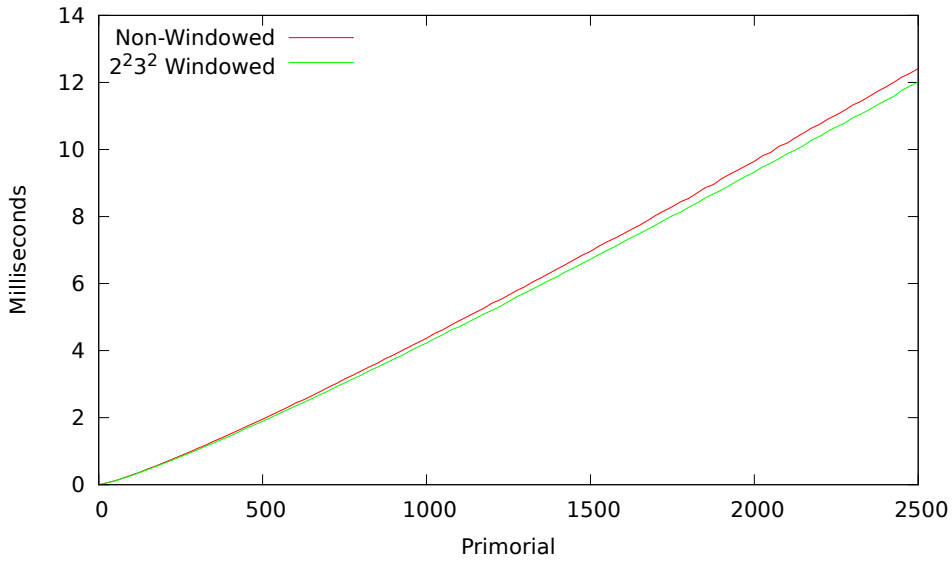

Figure 5.25: Right-to-Left 2,3 Chains (59-bit Discriminants).

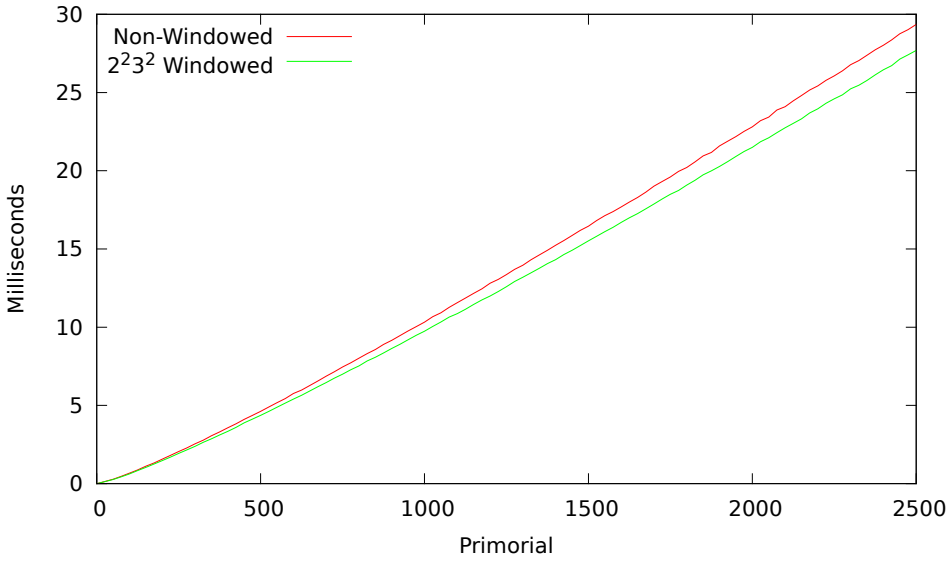

Figure 5.26: Right-to-Left 2,3 Chains (118-bit Discriminants).

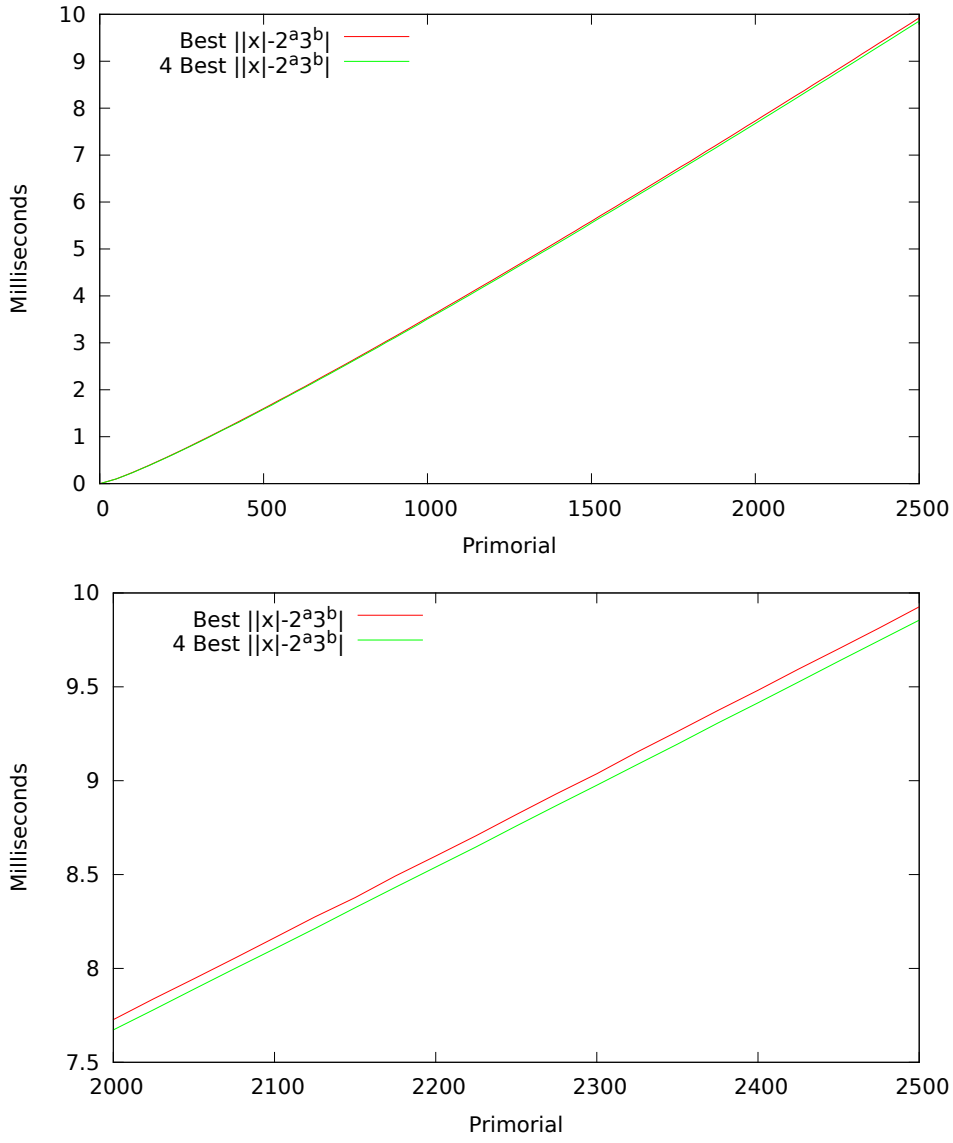

Figure 5.27: Left-to-Right 2,3 Representations (59-bit Discriminants).

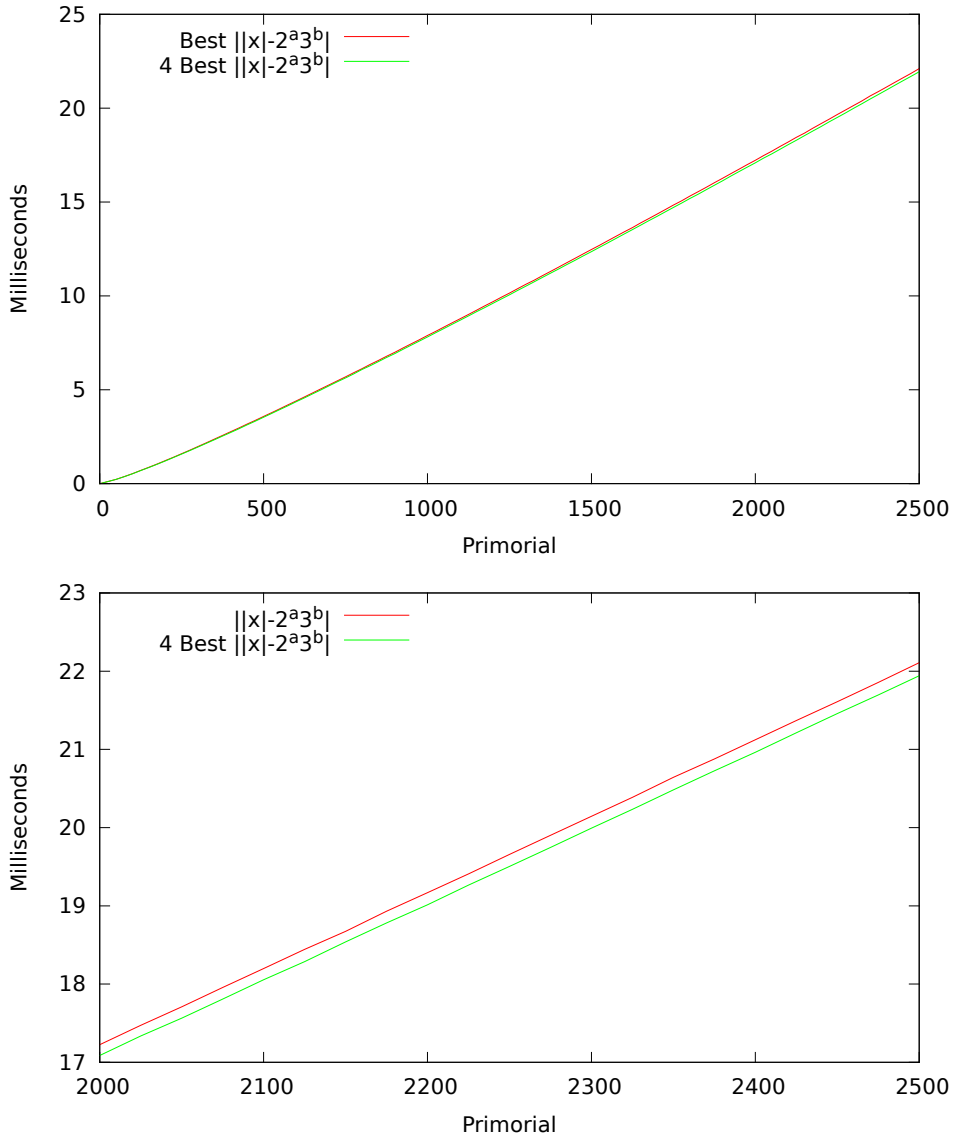

Figure 5.28: Left-to-Right 2,3 Representations (118-bit Discriminants).

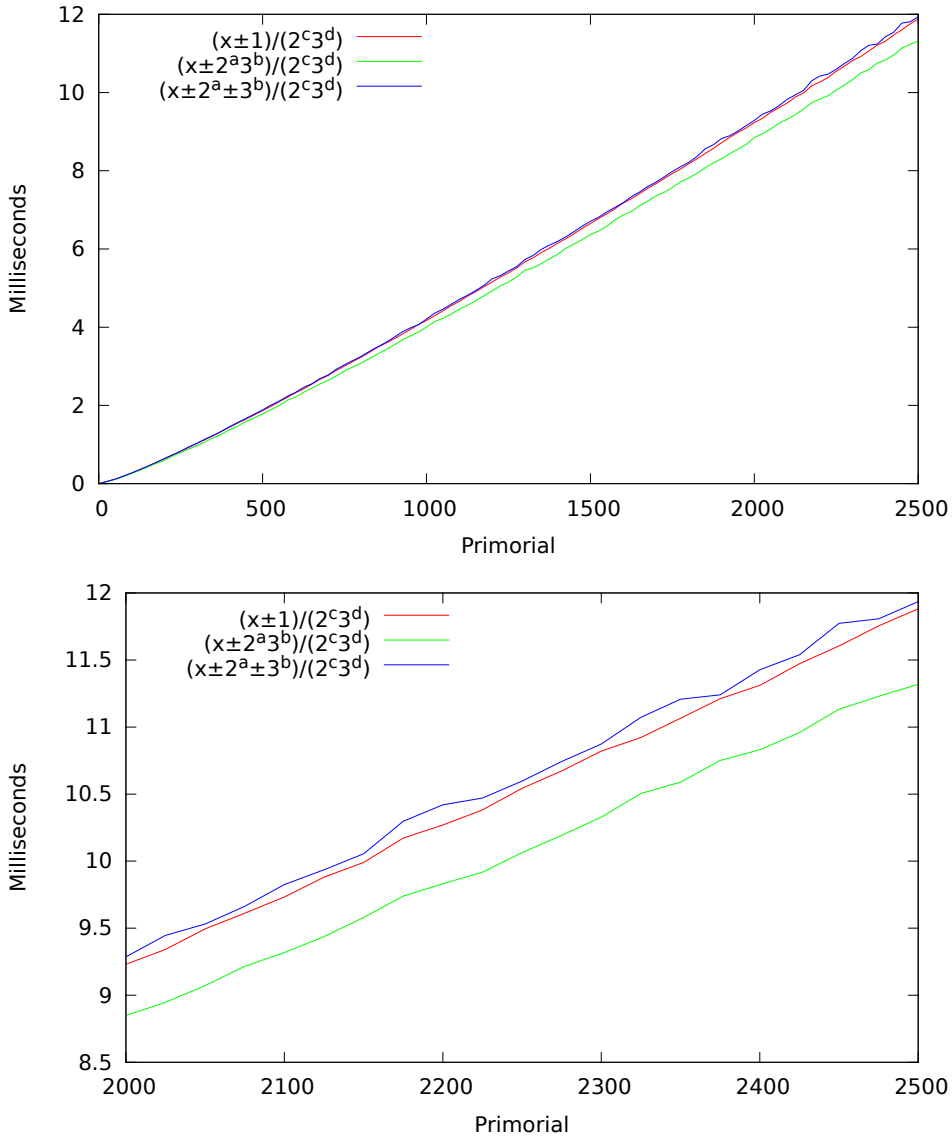

Figure 5.29: 4 Best $(x \pm \cdots)/(2^c 3^d)$ (59-bit Discriminants).

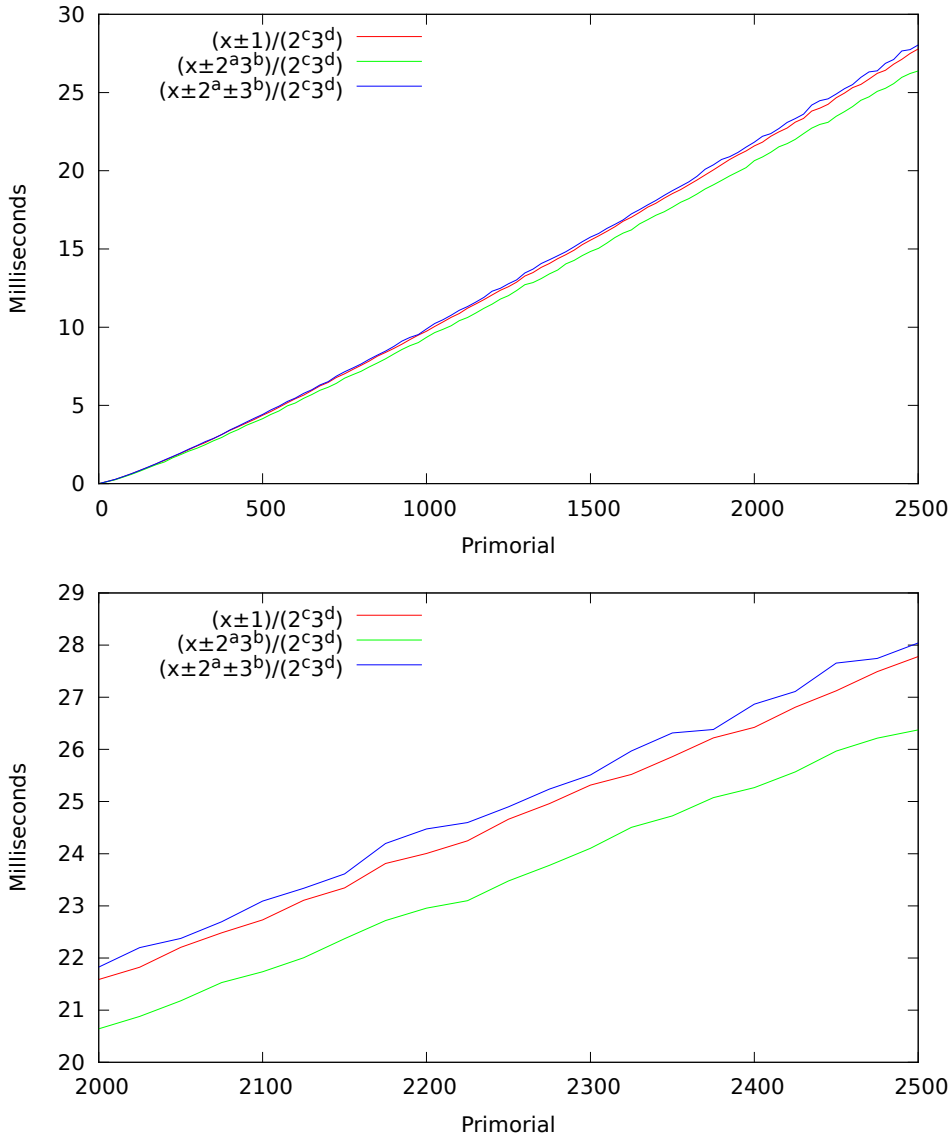

Figure 5.30: 4 Best $(x \pm \cdots)/(2^c 3^d)$ (118-bit Discriminants).

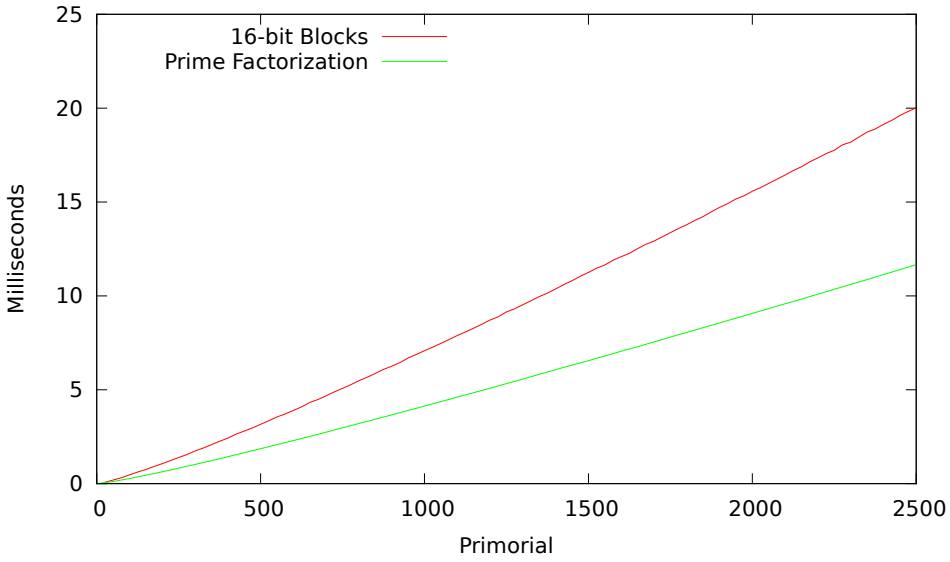

Figure 5.31: Use g^b for 16-bit b (59-bit Discriminants).

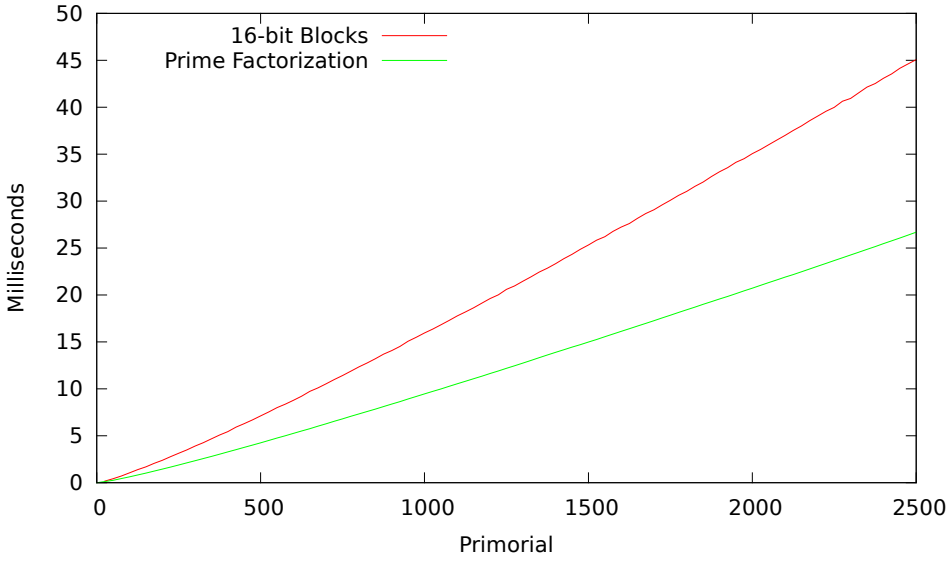

Figure 5.32: Use g^b for 16-bit b (118-bit Discriminants).

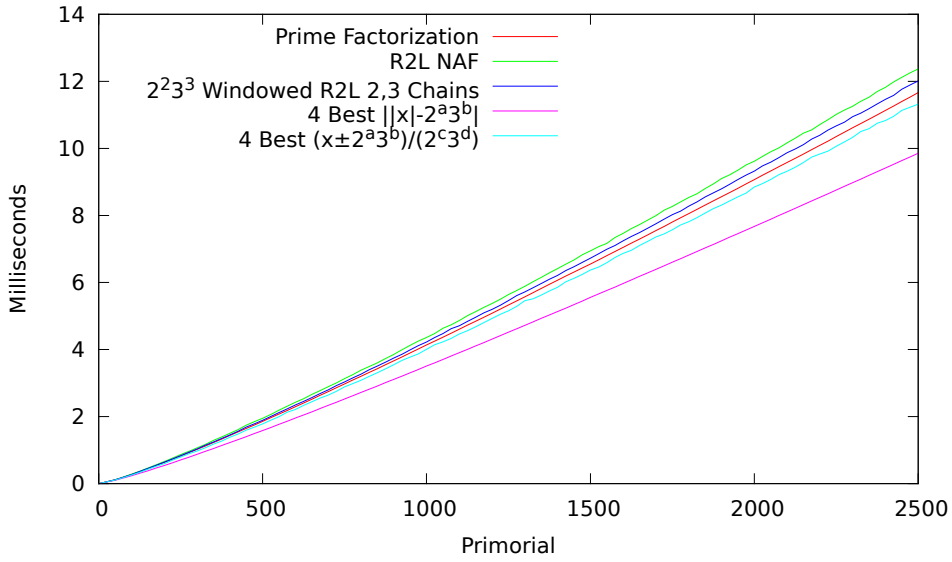

Figure 5.33: The best performers from each category (59-bit Discriminants).

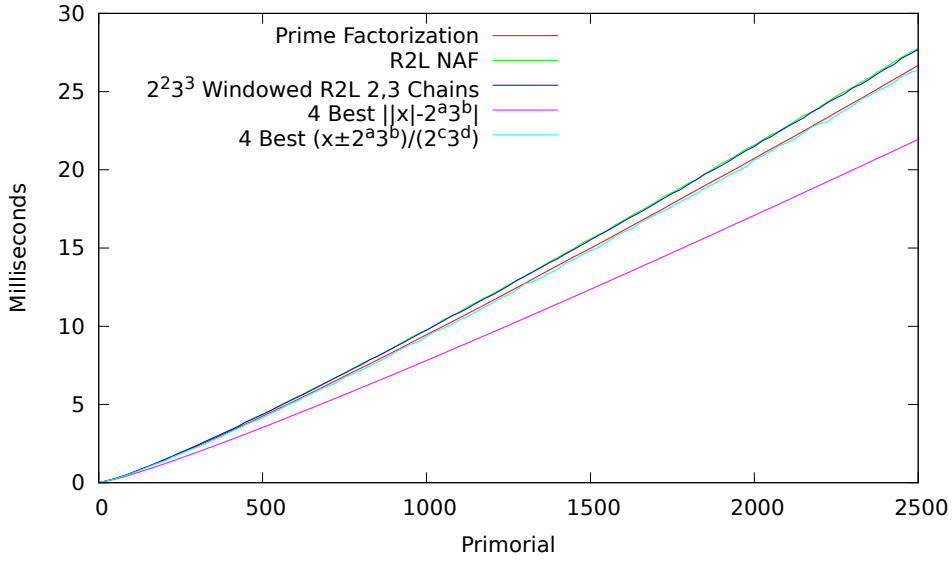

Figure 5.34: The best performers from each category (118-bit Discriminants).

5.4 Summary

This Chapter presented the approach and results we used to improve exponentiation in the ideal class group of imaginary quadratic number fields. We began with several methods for computing solutions to equations of the form $s = Ua + Vb$ where s is the greatest common divisor of both a and b, since arithmetic in the ideal class group is dominated by this operation. We found that when the inputs a and b are bound by 2^{32}, that the standard Extended Euclidean Algorithm performs best. When $2^{32} \leq a, b < 2^{64}$, a left-to-right binary GCD computation is fastest. Finally, for $a, b \geq 2^{64}$ we defer to the implementation in the GNU Multiple Precision (GMP) library.

Following this, we discussed specialized implementations of ideal arithmetic using 64-bit, 128-bit, and unbound arithmetic. Our 64-bit implementation is known to work correctly for negative discriminants $-2^{58} < \Delta < 0$, while our 128-bit implementation works with negative discriminants $-2^{118} < \Delta < 0$. We discussed several low-level optimizations and showed an improvement in the performance of ideal class arithmetic over our reference implementation using GMP.

In the last section, we explored several methods for exponentiation including novel extensions of methods for computing 2,3 chains and representations. 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, we found that generating new terms of the form $\left||x| - 2^a 3^b\right|$ gave the fastest representations on average.

In the next Chapter, we bring all of these improvements together in our implementation of SuperSPAR – an algorithm with the fastest average time to factors integers n in the range $2^{54} \leq n < 2^{63}$ of the integer factoring algorithms that we tested.

Chapter 6

SuperSPAR Experiments

6.1 Motivation: How fast can we make SuperSPAR

6.2 Coprime Finding

- Wheeling
- Sieving
- Delta Tables

6.3 Exponentiation

6.3.1 Bounds for primorial (bound for prime vs exponent)

6.3.2 Primorial vs Prime Powers

i.e. is it faster to use the product and one exponentiation or individual primes and many exponentations

6.3.3 Best primorial bound for n-bit integers

6.4 Time spent on search vs powering

6.5 Sequential prime ideals vs random prime ideals

6.6 Ratio of prime ideals to multipliers

6.7 Best multipliers to use

6.8 Chained Hashing vs Open Address Hashing

Only hash the pair (a, c) so that $[\mathfrak{a}]$ and $[\mathfrak{a}]^{-1}$ hash to the same value.

6.9 Empirical search for primorial and step count

- Linear, grid, 2d quadratic, 1d grid + binary search on step count.
- (Quadratic samples at 1/3 and 2/3 and throws away the larger third)

6.10 Comparison with other algorithms

- Pari/GP
- Custom SQUFOF
- GNU MSieve
- GNU-ECM
- YAFU
- Flint

- Maple

Chapter 7

Further Research

7.0.1 Improvements

Time 2,3 representation generation vs 2,3 exponentiation.

Windowed left-to-right GCD. 2,3 base GCD.

Inclusion into supervisor's ANTL library and hopefully Sage or/and Pari/GP.

Try some extremely large (1Gb+) primorials that are written to disk, powering 2,3 r2l chain of them, and then doing a pollard-rho or something with a smaller primorial for the same number of steps. The idea being to simulate the original SPAR, but with our library and a few dbns tricks. Then we try factoring some much larger composites.

This might be interesting to try a discrete log record too.

Large differences in exponents show general trends to 2,3 representations, however, when we look close up, the variation in 2, 3 representations between n and $n+1$ varies greatly. It might be interesting to measure how much they vary.

7.1 Other types of Ideal Arithmetic

For example, the ideal ring of algebraic integers of real quadratic fields.

7.2 Function Fields

For example Hyperelliptic Curves

7.3 DBNS with other coprime bases

Could try tree based approaches where we approximate the cost of the complete chain based on the partial chain, rather than using the smallest remaining exponent and the cost to break ties.

7.4 Number systems with three, four, or more bases.

7.5 Super^{2}SPAR

SuperSPAR with other corprime bases or multiple bases.

In our implementation we choose a primorial, P, such that baby steps are coprime to P and giant steps are a multiple of P. We can select a product of small primes that need not be consecutive, e.g. $P = 2 \times 3 \times 11 \times 13$ and then use baby steps coprime to this and giant steps that are a multiple of this.

Also consider parallelization of SuperSPAR using multipliers. And possibly implementing it on a GPU.

Bibliography

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

[2] 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.

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

[4] 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.

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

[6] H. Cohen. *A course in computational algebraic number theory*. Springer-Verlag, Berlin, 1993.

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

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

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

[10] 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.

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

[12] 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.

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

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

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

[16] 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.

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

[18] 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.

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

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

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

[22] 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.

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

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

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

[26] 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.

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

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

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

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

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

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

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

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

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

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

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