Order Computations in Generic Groups

by

Andrew V. Sutherland

S.B., Massachusetts Institute of Technology, 1990

Submitted to the Department of Mathematics
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2007

Author .....
Department of Mathematics
May 16, 2007

Certified by .....
Michael F. Sipser
Professor of Applied Mathematics
Thesis Supervisor

Accepted by .....
Alar Toomre
Chairman, Applied Mathematics Committee

Accepted by .....
Pavel I. Etingof
Chairman, Department Committee on Graduate Students

Order Computations in Generic Groups

by

Andrew V. Sutherland

Submitted to the Department of Mathematics on May 16, 2007,
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Abstract

We consider the problem of computing the order of an element in a generic group. The two standard algorithms, Pollard's rho method and Shanks' baby-steps giant-steps technique, both use $\Theta\left(N^{1/2}\right)$ group operations to compute $|\alpha| = N$. A lower bound of $\Omega\left(N^{1/2}\right)$ has been conjectured. We disprove this conjecture, presenting a generic algorithm with complexity $o\left(N^{1/2}\right)$. The running time is $O\left((N/\log\log N)^{1/2}\right)$ when N is prime, but for nearly half the integers $N \in [0, M]$, the complexity is $O\left(N^{1/3}\right)$. If only a single success in a random sequence of problems is required, the running time is subexponential.

We prove that a generic algorithm can compute $|\alpha|$ for all $\alpha \in S \subseteq G$ in near linear time plus the cost of a single order computation with $N = \lambda(S)$, where $\lambda(S) = \operatorname{lcm}|\alpha|$ over $\alpha \in S$. For abelian groups, a random $S \subseteq G$ of constant size suffices to compute $\lambda(G)$, the exponent of the group. Having computed $\lambda(G)$, we show that in most cases the structure of an abelian group G can be determined using an additional $O\left(N^{\delta/4}\right)$ group operations, given an $O\left(N^{\delta}\right)$ bound on $|G| = N$. The median complexity is approximately $O\left(N^{1/3}\right)$ for many distributions of finite abelian groups, and $o\left(N^{1/2}\right)$ in all but an extreme set of cases. A lower bound of $\Omega\left(N^{1/2}\right)$ had been assumed, based on a similar bound for the discrete logarithm problem.

We apply these results to compute the ideal class groups of imaginary quadratic number fields, a standard test case for generic algorithms. The record class group computation by a generic algorithm, for discriminant $-4(10^{30} + 1)$, involved some 240 million group operations over the course of 15 days on a Sun SparcStation4. We accomplish the same task using 1/1000th the group operations, taking less than 3 seconds on a PC. Comparisons with non-generic algorithms for class group computation are also favorable in many cases. We successfully computed several class groups with discriminants containing more than 100 digits. These are believed to be the largest class groups ever computed.

Thesis Supervisor: Michael F. Sipser
Title: Professor of Applied Mathematics

Contents

1 Generic Group Computation **19**

1.1 Generic Group Functions 21

1.2 Generic Group Algorithms 24

1.3 Comparison and Discussion 28

2 The Complexity of Order Computations **31**

2.1 Analyzing Order Algorithms 33

2.2 An Exponential Lower Bound 35

2.3 Factoring Reduction 38

3 Existing Order Algorithms **41**

3.1 The Rho Method 42

3.2 Baby-Steps Giant-Steps 49

4 The Primorial-Steps Algorithm **55**

4.1 Bounded Primorial Steps 57

4.2 Unbounded Primorial Steps 63

5 The Multi-Stage Sieve **73**

5.1 Primorial-Steps Search 76

5.2 Rho Search and Hybrid Search 77

5.3 Complexity Analysis 77

5.4 A Generic Recipe for Subexponential Algorithms 80

6 A Number-Theoretic Interlude **85**

6.1 The Basics 85

6.2 The Distribution of Coarse Numbers 90
6.3 The Distribution of Smooth Numbers 93
6.4 Semismooth Numbers 96
6.5 Applications 99

7 Fast Order Algorithms **107**
7.1 Factored Exponents and Pseudo-Order 108
7.2 A Linear-Time Algorithm for Large Exponents 110
7.3 The Classical Algorithm 112
7.4 Improvements 113
7.5 The Snowball Algorithm 115
7.6 A Fast Algorithm for Multi-Exponentiation 119
7.7 Performance Comparisons 125

8 Computing the Exponent of a Group **127**
8.1 Computing $\lambda(G)$ 128
8.2 Order Complexity Bounds 132
8.3 Computing $\lambda(S)$ 134
8.4 The Order Computation Theorem 135

9 Computing the Structure of an Abelian Group **139**
9.1 Existing Algorithms 141
9.2 Using $\lambda(G)$ to Compute Group Structure 141
9.3 Discrete Logarithms 142
9.4 Abelian Group Structure Algorithm 143
9.5 Expanding a Basis in a p-Group. 145
9.6 Vector Form of the Discrete Logarithm 147
9.7 Complexity Analysis 148
9.8 Computing Group Structure Given an Exponent of G 151

10 A Number-Theoretic Postlude **153**
10.1 The Class Group of a Number Field 153
10.2 Binary Quadratic Forms 154
10.3 Class Groups of Imaginary Quadratic Number Fields 155

11 Performance Results **159**

11.1 Order Computations in Random Groups 161

11.2 Comparison to Existing Generic Algorithms 165

11.3 Generic Order Computations in Class Groups 169

11.4 Comparison to Non-Generic Algorithms 173

11.5 A Trial of the Generic Subexponential Recipe 176

11.6 A Record Setting Computation 178

A Class Group Table Construction Costs **181**

B Imaginary Quadratic Class Groups **185**

List of Tables

4.1 The First Ten Primorials 59

4.2 Bounded Primorial Steps (T_1) vs. Bounded Baby-Steps Giant-Steps (T_2) . . . 63

4.3 Hypothetical Comparison of Order Algorithms with $|\alpha| \in [1, 10^{24}]$ 70

6.1 Asymptotic Lower Bounds on Semismooth and Smooth Probabilities 99

7.1 Comparison of Multi-Exponentiation Packing Functions 124

7.2 Performance Comparison of Fast Order Algorithms 126

8.1 Orders and Exponents of Some Finite Groups 128

11.1 Black Box Performance (thousands of group operations per second) 161

11.2 Order Computations in Random Groups. 164

11.3 Ideal Class Groups Computed by Generic Algorithms, $D = -4(10^n + 1)$ . . . 166

11.4 Ideal Class Groups Computed by Generic Algorithms, $D = -(10^n + 3)$ 168

11.5 Sample Construction Costs for $D = -(2^n - 1)$, $n = 120$ to 129 170

11.6 Median δ-Values of Class Group Structure Computations 172

11.7 Imaginary Quadratic Class Groups with $|D| > 10^{100}$ 179

A.1 Construction Costs for $D = -(2^n - 1)$, $n = 60$ to 95 182

A.2 Construction Costs for $D = -(2^n - 1)$, $n = 96$ to 129 183

A.3 Construction Costs for $D = -(2^n - 1)$, $n = 130$ to 160 184

B.1 Class Groups for $D = -(2^n - 1)$, $n = 2$ to 30 187

B.2 Class Groups for $D = -(2^n - 1)$, $n = 31$ to 60 188

B.3 Class Groups for $D = -(2^n - 1)$, $n = 61$ to 90 189

B.4 Class Groups for $D = -(2^n - 1)$, $n = 91$ to 120 190

B.5 Class Groups for $D = -(2^n - 1)$, $n = 121$ to 150 191
B.6 Class Groups for $D = -(2^n - 1)$, $n = 151$ to 180 192
B.7 Class Groups for $D = -(2^n - 1)$, $n > 180$ 193
B.8 Class Groups for $D = -4(2^n + 1)$, $n = 2$ to 30 194
B.9 Class Groups for $D = -4(2^n + 1)$, $n = 31$ to 60 195
B.10 Class Groups for $D = -4(2^n + 1)$, $n = 61$ to 90 196
B.11 Class Groups for $D = -4(2^n + 1)$, $n = 91$ to 120 197
B.12 Class Groups for $D = -4(2^n + 1)$, $n = 121$ to 150 198
B.13 Class Groups for $D = -4(2^n + 1)$, $n = 151$ to 180 199
B.14 Class Groups for $D = -4(2^n + 1)$, $n > 180$ 200
B.15 Class Groups for $D = -(10^n + 3)$, $n = 2$ to 30 201
B.16 Class Groups for $D = -(10^n + 3)$, $n > 30$ 202
B.17 Class Groups for $D = -4(10^n + 1)$, $n = 2$ to 30 203
B.18 Class Groups for $D = -4(10^n + 1)$, $n > 30$ 204

Introduction

The foundation of this thesis is a new generic algorithm for computing the order of an element in a finite group. This algorithm beats the conjectured lower bound for the problem and is substantially faster than existing methods. For large problems the performance improvement is often several orders of magnitude. Applications of this result include new algorithms for computing the group exponent, and for determining the structure of an abelian group.

We begin with a gentle and somewhat light-hearted introduction. Those desiring a more formal presentation may proceed directly to Chapter 1.

Generic Algorithms

Suppose we are asked to compute:

$$x = 271828182845904523^{53^{31415926535897932384}}.$$

Given the size of the numbers involved, we decide to write a computer program. When implementing our program, we have two basic choices to make: how to multiply, and how to exponentiate. These choices may seem closely related, but they are quite separate. The fastest algorithms to multiply two integers are very sophisticated; they exploit particularly efficient representations of integers. The fastest ways to exponentiate are generally independent of these details, and not that complicated. Russian peasants, we are told, solved this problem in the 19th century [68]. The best modern algorithms have only marginally improved their method.

Let us assume we choose the most advanced multiplication algorithm available and go with the Russian peasant method of exponentiation: squaring and multiplying using

the familiar binary algorithm. We then execute our program with great expectations and unfortunate results. Our computer runs out of memory. Examining the problem more closely, we realize that it is intractable as written (at least at today's memory prices) and ask for a different problem, preferably one whose answer isn't longer than the question. We are given the apparently longer, but much easier problem:

$$x = 271828182845904523^{53141592653589793238^4} \bmod 577215664901532860606512^0.$$

We modify our program and are rewarded with the answer a few microseconds later. We may attribute this to our state-of-the-art multiplication algorithm, but in fact we owe a greater debt to the Russian peasants.[1] They came up with a *generic* algorithm for exponentiation that may be used effectively in any finite group.

The most impressive feature of the Russian peasant method, aside from its speed and simplicity, is its generality. We can use it to exponentiate matrices, permutations, and polynomials, as well as far more abstract objects, provided that in each case we are given a multiplication operation. The algorithm is always the same, regardless of the group. More importantly, it doesn't matter how the group is represented. If we decide to upgrade our polynomial multiplication algorithm to use a point representation rather than storing coefficients, the Russian peasant algorithm remains happily oblivious to this change. Indeed, we would be rightly upset if someone modified the algorithm in a non-generic way that jeopardized this compatibility. The exponentiation algorithm should know nothing about the multiplication algorithm, treating it as a *black box*.

The Order of a Group Element

An interesting thing happens when we exponentiate in a finite group. Pick an element and compute its sequence of powers:

$$\alpha, \alpha^2, \alpha^3, \ldots, \alpha^N, \alpha^{N+1}, \alpha^{N+2}, \ldots$$

It may take a while, but eventually we must get the same element twice. When this happens it is always the element we started with, and the previous element in the sequence

[1] It would be only a slight exaggeration to say that much of our technological infrastructure would be rendered useless if we lacked the ability to exponentiate quickly.

is the identity element, 1_G. For every element α, there is a least positive integer for which $\alpha^N = 1_G$. We call this number the *order* of α, denote it by $|\alpha|$, and use a capital letter N, because this number may be very large. We will generally reserve n for smaller numbers, like the number of bits in N.

In order to communicate with the black box, we need a way to refer to group elements. The black box decides exactly how to do this, since it is responsible for the representation, but in every case a unique identifier is assigned to each group element. This identifier is nothing more than a sequence of bits. It may be an encoding of the element, a pointer into a memory location, or simply a random number, whatever the black box chooses. We cannot, nor should we, attempt to extract any information from these identifiers, but we can tell when two bit-strings are equal. It is then possible for us to determine $|\alpha|$ by computing successive powers of α and noticing when two are the same.

Determining the order of α in this manner is a slow process, requiring N group operations. Fortunately there is a better way; in fact, there are two better ways. They are both based on noticing that we needn't find N such that $\alpha^N = 1_G$ directly. It suffices to find any two powers of α that are equal, say $\alpha^k = \alpha^j$. When this happens we know that $\alpha^{k-j} = 1_G$. This does not necessarily mean that $k - j$ is the least integer with this property, it may be a multiple of $|\alpha|$, but finding a multiple is the hard part. Once a reasonably small multiple is known, computing $|\alpha|$ is easy.

Baby-Steps Giant-Steps

The first better way to compute $N = |\alpha|$ is the baby-steps giant-steps method, due to Shanks [95], and it finds N exactly. It does this by computing two lists of powers, say

$$\alpha^1, \alpha^2, \ldots, \alpha^{999}, \alpha^{1,000} \qquad \text{and} \qquad \alpha^{2,000}, \alpha^{3,000}, \ldots, \alpha^{1,000,000},$$

and then compares them to see if it can find a match. If N is any number less than 1,000,000, then either α^N is in the first list, or there is a power in the first list that is equal to a power in the second list. The smallest difference of two exponents for which this is true will be N. Notice that this method involved fewer than 2,000 powers of α, not 1,000,000.

If N is greater than $1,000,000$, we might appear to be stuck. However, we can always

guess a larger bound and construct two longer lists.[2] Eventually we will find N using just $O\left(N^{1/2}\right)$ group operations.

The Birthday Paradox

The second better way, known as Pollard's rho algorithm [83], is a little more haphazard. It involves guessing a random sequence of powers of α until we find two that are the same. This relies on what is often called the "birthday paradox": if there are at least 23 people in a room, there is a good chance two share the same birthday. If we compute approximately $\sqrt{N}$ random powers of α, there is a good chance that two will be the same. Assuming we know the exponents of these two powers, their difference will be a multiple of N.

The birthday paradox is so named because many find it surprising that such a small number of people suffice: $\sqrt{N}$ is a lot smaller than N. Compared to $n = \lg N$, it is very big. It is impractical to compute $N = |\alpha|$, using either of the two methods that are known, when N is greater than 2^{100}. It is inconvenient for N greater than 2^{50}.

A Lower Bound

Computing the order of α involves finding the least positive k satisfying:

$$\alpha^k = 1_G.$$

There is a closely related problem that looks quite similar. We replace 1_G with β:

$$\alpha^k = \beta.$$

Solving this equation is known as the discrete logarithm problem. This is a very famous problem, partially due to the fact that it is one of very few that is *provably* hard in a model of computation that comes close to how people actually implement algorithms. Given the unsettled nature of the "P versus NP" question, we have no way of knowing whether there is a fast way to solve the discrete logarithm problem in general. There are *non-generic* algorithms that can compute discrete logarithms using specific representations of particular groups to achieve sub-exponential (but still super-polynomial) complexity. Nevertheless,

[2] We will see more elegant solutions to this problem.

the only algorithms that are known to work in every group, generic algorithms, are variants of the two methods discussed above for computing $|\alpha|$.

In 1997, Shoup proved that no generic algorithm could solve the discrete logarithm problem in every group, unless it used $\Omega\left(N^{1/2}\right)$ group operations [96]. Given the similarity between the two problems, it has been generally assumed that computing $|\alpha|$ also requires $\Omega\left(N^{1/2}\right)$ group operations. Indeed, in one sense computing $|\alpha|$ seems harder, since the problem is initially unbounded, whereas a discrete logarithm algorithm is generally told $|\alpha|$ before it starts.

This assumption is incorrect. It is possible to compute $|\alpha|$ in any group using $o\left(N^{1/2}\right)$ group operations. This new result is achieved by the primorial-steps algorithm, presented in Chapter 4.

What's Hidden Behind the Little-o?

Anytime a complexity theorist uses a little-o in front of a big bound, one should be suspicious. This case is no exception. The basic result, while quite surprising from a theoretical point of view, is not a major breakthrough from a practical perspective. In the worst case, the expression hidden by the little-o is very small: $(\log\log N)^{-1/2}$. The constant factors are quite good however, and the net result is a speed-up by a factor of 2 or 3 in the typical worst-case scenario. Hardly earth shattering, but still better than the improvements made on the Russian peasant algorithm over the past 200 years.

Of far more practical relevance is the average case, or, more precisely, the median case. If N is randomly chosen from some large interval, then for nearly half the values of N, $|\alpha| = N$ can be computed using $O\left(N^{1/3}\right)$ group operations. If $N \approx 2^{100}$, this makes it a tractable problem in many cases, one that can be solved on a standard PC in a day or two, depending on the speed of the black box. For about one tenth of the values of N, the running time is $O\left(N^{1/4}\right)$, yielding a problem that can easily be solved in less than an hour. The two algorithms mentioned above have either zero or a negligible chance of solving any 100-bit problem instance on a single computer in less than years.[3]

In the case of the standard baby-steps giant-steps algorithm, it will run out of space long before this, probably on the first day. The rho method, on the other hand, requires very little space. We take advantage of this feature to obtain a space-efficient hybrid that

[3] Assuming current (2007) technology.

effectively matches the median complexity of the primorial-steps algorithm, as described in Chapter 5.

A Subexponential Recipe

One way to handle a problem that is too hard to solve is to simply pass it by and move on to another problem that you hope will be easier. Such a strategy can be very effective if all that is required is a solution to any one of a large set of problems. This is the idea behind some of the fastest algorithms we know for hard problems, ones which achieve subexponential running times. We will see that any problem which requires the solution of just one of a sequence of random order computations can be solved by a generic algorithm in subexponential time. This spells out a recipe for inventing new subexponential algorithms: find a way to transform the problem into a sequence of random order computations and let a generic algorithm do the rest. This recipe is included in Chapter 5.

Send in the Primes

Nearly all the new algorithms in this thesis are based on a simple observation: most numbers aren't prime. They can be broken up into pieces which makes them easier to handle. The ones that cannot, stand out for that very reason. This also makes them easier to handle, because we know where to find them. In the context of order computations, it is easier to compute the order of α if $|\alpha|$ is even: let $\beta = \alpha^2$ and then figure out the order of β. It is also easier to compute the order of α if you know it must be odd. Put these two ideas together and it is always easier.

In order to really capitalize on these ideas, we need to use a lot of primes, not just the prime 2. The most impressive results are not obtained until we call on an astounding number of primes, millions or more. In order to make use of this quantity of primes, we need two things: a fast exponentiation algorithm, which we have, thanks to the Russian peasants,[4] and what is known as a *fast order algorithm*. This is an algorithm that has an easy job and must do it very quickly. It is given a large set of prime numbers known to contain all the divisors of $|\alpha|$ and must first determine the minimal subset with this property, and then $|\alpha|$. The first step can be done in linear time (in the size of the set), and the second step can be done in near linear time (in $\lg |\alpha|$), using algorithms presented in Chapter 7.

[4]We will use a slightly faster version.

What Good Is It?

While a faster order algorithm is certainly useful in its own right, the theme of this thesis is that if you can perform a single order computation in a given group, you can do a whole lot more for about the same computational cost. We start by showing that all the order computations required by a generic algorithm can be completed for essentially the same cost as a single computation on an element with order $N = \lambda(G)$. The integer $\lambda(G)$ is the least N for which $\alpha^N = 1_G$ for every member of the group, known as the *exponent* of the group. Once $\lambda(G)$ is known, it can be used to complete all other order computations very quickly, in near-linear time. If we replace G with any subset of elements in G, an equivalent statement holds. This leads to the Order Computation Theorem, proven in Chapter 8.

We then apply the group exponent to compute the structure of a finite abelian group. We give an effective isomorphism between the group in the black box to a product of cyclic groups, providing a set of independent generators with known order. This algorithm uses $\lambda(G)$ to obtain a complexity that is generally no greater than the cost of a single order computation in the same group (provided a suitable bound on $|G|$ is known). The new algorithm is not only faster, it is also simpler than existing solutions. These results are described in Chapter 9.

In the final chapters and the appendices we reap the benefit of all these improvements. We consider performance comparisons with existing results and put the new algorithms to good use by computing the structure of some groups of particular interest to number theorists (and cryptographers). Tables of ideal class groups for quadratic discriminants of the form $-(10^n + 3)$, $-4(10^n + 1)$, $-(2^n - 1)$, and $-4(2^n + 1)$ are provided, including many previously unknown class groups. We apply the generic subexponential method described in Chapter 5 to compute what may be the largest ideal class groups explicitly determined to date, involving negative discriminants with more than 100 decimal digits.

A Reader's Guide

The first three chapters of this thesis define the model of computation, analyze the complexity of order algorithms in general, and examine the existing methods in some detail. The core of the thesis lies in chapters 4, 5, 7, 8, and 9 mentioned above, with a number-theoretic interlude in the middle. The final section contains a presentation of performance results

along with a brief description of ideal class groups and some conjectures surrounding them.

We assume the reader is familiar with asymptotic notation (see note below), and the most elementary facts from group theory (the theorems of Legendre and Sylow suffice). Ironically, we will have a greater need for number theory rather than group theory. The required results are contained in Chapter 6.

A Note on Notation

Wherever possible we use standard notation and recall definitions as needed. A compilation of number-theoretic definitions and notation can be found in Chapter 6. We use standard big-O and little-o notation for asymptotic expressions, along with two analytic relations which we recall here:

$$f(x) \sim g(x) \iff \lim_{x\to\infty} \frac{f(x)}{g(x)} = 1;$$
$$f(x) \prec g(x) \iff \lim_{x\to\infty} \frac{f(x)}{g(x)} = 0.$$

The variable x may be integer or real. Note that $f(x) \prec g(x)$ is equivalent to $f(x) = o(g(x))$ and in both cases we say that g *dominates* f. An *increasing* function $f(x)$ satisfies $a \leq b \Rightarrow f(a) \leq f(B)$ and is not necessarily *strictly increasing*.

We use big-Ω and big-Θ to specify asymptotic relationships in the usual way and avoid the use of these symbols in a number-theoretic role. We do not use little-ω for asymptotic notation, reserving ω for the Buchstab function, defined in Chapter 6.

Chapter 1

Generic Group Computation

Chapter Abstract

We define a formal model for computation in finite groups based on a synthesis of ideas common to the study of generic group algorithms and black-box groups. We give a rigorous specification of generic group functions (and relations), and consider several examples. We then define correctness criteria for three classes of algorithms, deterministic, probabilistic, and non-deterministic, and give complexity metrics for time and space. We conclude with a discussion and comparisons to prior work.

A *generic group algorithm* (or simply a *generic algorithm*) is based on a "black box" which hides the representation of group elements. An algorithm is initially given an identifier for each group element it takes as an input. Identifiers for other group elements are obtained via calls to the black box, which supports the following operations:

1. Return an identifier for the identity 1_G.
2. Return an identifier for the inverse α^{-1}.
3. Return an identifier for the product $\alpha\beta$.

A randomized black box adds the operation:

4. Return an identifier for a random element of G.

The black box is free to choose any values for the identifiers as long as each group element is assigned a unique bit-string of length $O\left(\lg|G|\right)$. More formally, we define:

Definition 1.1. *A* **black box** $\mathcal{B}(G)$ *for a group* G *is a bijective function* $\mathcal{B} : G \to \mathcal{I}$*, where* $\mathcal{I} \subseteq \{0,1\}^\ell$ *is a set of identifiers of length* $\ell = O\left(\lg|G|\right)$*. For* $\alpha, \beta \in \mathcal{I}$ *the interface to the black box is provided by the functions:*

$$\mathcal{B}_{id}() = \mathcal{B}(1_G), \qquad \mathcal{B}_{inv}(\alpha) = \mathcal{B}\big(\big(\mathcal{B}^{-1}(\alpha)\big)^{-1}\big), \qquad \mathcal{B}_{prod}(\alpha,\beta) = \mathcal{B}\big(\mathcal{B}^{-1}(\alpha)\mathcal{B}^{-1}(\beta)\big).$$

A **randomized black box** $\mathcal{B}(G)$ *is a black box for* G *with a function* $\mathcal{B}_{rand} : \{0,1\}^{2\ell} \to \mathcal{I}$ *with the property that if* $\rho \in \{0,1\}^{2\ell}$ *is a uniform random variable, then* $\Pr[\mathcal{B}_{rand}(\rho) = \mathcal{B}(\alpha)] = 1/|G| \pm \epsilon$ *for all* $\alpha \in G$*, where* $\epsilon \leq 1/|G|^c$ *for some* $c > 1$*.*

The input to $\mathcal{B}_{rand}(\rho)$ represents a source of randomness (or more generally, non-determinism) that may be available to a probabilistic (or non-deterministic) algorithm. The precise role of ρ is explained more fully in Section 1.2. A black box $\mathcal{B}(G)$ can be regarded as a group isomorphism between G and $\mathcal{I}$, where the group operation on $\mathcal{I}$ is derived from G. As a notational convenience, we may refer directly to the group G rather than $\mathcal{I}$, with the understanding that all group elements are actually identifiers from $\mathcal{I}$ and all group operations are performed via the black box. For a randomized black box we may write "let α be a random element of G" to indicate that $\alpha = \mathcal{B}_{rand}(\rho)$.

Before formally defining the notion of a generic algorithm, it may be helpful to consider a few examples.

Algorithm 1.1 (Binary Exponentiation)**.** *Given* $\alpha \in G$ *and an integer* k*, algorithm* $\mathcal{A}(\alpha, k)$ *recursively computes* α^k*:*

1. If $k < 0$, return $\mathcal{A}(\alpha^{-1}, -k)$.
2. If $k = 0$, return 1_G. If $k = 1$, return α.
3. Set $\beta \leftarrow \mathcal{A}(\alpha, \lfloor k/2 \rfloor)$.
4. If k is even, return $\beta\beta$, otherwise, return $\beta\beta\alpha$.

The algorithm above can be used to compute α^k in any group G using any black box $\mathcal{B}(G)$. This is what makes it a *generic* algorithm: it is independent of both G and the black box. The number of calls made to the black box is at least n and at most $2n-2$, where $n = \lceil \lg|k| \rceil$ (for $k \neq 0$), thus the binary exponentiation algorithm uses $\Theta(n)$ group operations.

Algorithm 1.2 (Randomized Abelian Test)**.** *Given a randomized black box and an integer parameter $t > 1$, the following algorithm tests whether G is abelian:*

1. For i from 1 to t:
    a. Let α and β be random elements of G.
    b. If $\alpha\beta \neq \beta\alpha$, return 0.
2. Return 1.

If G is not abelian, the algorithm will return 0 with probability $\geq 1 - \left(\frac{3}{4}\right)^t$, since at least one fourth of all ordered pairs of elements from a non-abelian group do not commute.[1] If G is abelian, the algorithm always returns 1. Thus we have a Monte Carlo algorithm with 1-sided error. The algorithm makes $4t$ calls to the black box: two in line 1a and two in line 1b. Note that the inequality test in line 1b is simply a comparison of identifiers and does not involve the black box. Algorithm 1.2 is a probabilistic generic algorithm which (with high probability) recognizes abelian groups using a constant number of group operations.

1.1 Generic Group Functions

In order to give precise meaning to statements such as "algorithm $\mathcal{A}$ is a generic exponentiation algorithm," it is useful to define a *generic (group) function*, or more generally, a *generic (group) relation*. Intuitively, a generic relation should have two properties:

1. The relation should (in principle) be defined for any finite group.
2. The relation must be invariant under group isomorphisms.

The exponentiation function $f_G : G \times \mathbb{Z} \to G$ defined by $f_G(\alpha, k) = \alpha^k$ is an example of a generic group function. It is well defined for any group G, and if $\varphi : G \to H$ is a group isomorphism, then for all $\alpha \in G$ and all $k \in \mathbb{Z}$ we have

$$\varphi\big(f_G(\alpha, k)\big) = f_H\big(\varphi(\alpha), k\big).$$

[1] The center $C(G)$ of a non-abelian group is a proper subgroup containing at most $|G|/2$ elements. For each $\alpha \notin C(G)$, the centralizer of α is a proper subgroup containing at most $|G|/2$ elements. Thus for at least half the α in G, there is a subset of at least half the elements in G, none of which commute with α.

The square-root relation defined by $R_G(\alpha, \beta) \Leftrightarrow \alpha^2 = \beta$ is an example of a generic group relation, where we may think of α as an input value and $\{\beta : R_G(\alpha, \beta)\}$ as the (possibly empty) set of valid outputs. This relation is also invariant under isomorphisms:

$$R_G(\alpha, \beta) \quad \Longleftrightarrow \quad R_G\big(\varphi(\alpha), \varphi(\beta)\big).$$

Invariance under isomorphisms necessarily holds for any relation that can be computed using a black box. Thus it suffices to define a generic relation for each equivalence class of isomorphic groups. In order to do this explicitly, it will be useful to have a specific enumeration of canonical groups in mind. One way to represent a finite group is by writing down its multiplication table (Cayley table). If $G = \{1_G, \alpha_2, \ldots, \alpha_n\}$ is a group with n elements indexed by the set $\{1, \ldots, n\}$, the Cayley table for G will be an $n \times n$ integer matrix M, where $M_{ij} = k$ whenever $\alpha_i \alpha_j = \alpha_k$. Call such a matrix a *Cayley matrix*. We can enumerate all Cayley matrices by ordering them first by size and then lexicographically. Distinct Cayley matrices may describe isomorphic groups, but among all such matrices, one will occur first in this ordering. Call this matrix a *minimal Cayley matrix*.

Definition 1.2. *The* **canonical family of finite groups** $\mathcal{F} = G_1, G_2, \ldots,$ *is the ordered sequence of all minimal Cayley matrices.*

It is convenient to identify elements of $\mathcal{F}$ with the group they describe, thus we may regard the $n \times n$ matrix $G \in \mathcal{F}$ as a group on the set $\{1, \ldots, n\}$.

The relations we wish to define typically involve a combination of group elements and integers,[2] as in the exponentiation function. To accommodate both types of parameters without risk of confusion, we define the parameter space P_G for a group G.

Definition 1.3. *Given a group G, the set $P_G = \{(0, \alpha) : \alpha \in G\} \cup \{(1, k) : k \in \mathbb{Z}\}$ denotes the disjoint union of the sets G and $\mathbb{Z}$. The set P_G^* denotes all vectors of zero or more elements of P_G. If $\varphi : G \to H$ is any function from G to H, $\varphi' : P_G \to P_H$ is defined by $\varphi'(0, \alpha) = \varphi(\alpha)$ and $\varphi'(1, k) = k$. The function $\varphi^* : P_G^* \to P_H^*$ is defined by letting $\varphi^*(\vec{x})$ be the result of applying φ' to each component of $\vec{x}$.*

We will abuse notation and write φ in place of φ^* where there is no risk of confusion. With these notational conveniences in hand, we can now define a generic relation.

[2] Or any finite objects that can be encoded as integers.

Definition 1.4. *A* **generic group relation** *R is an infinite sequence of relations indexed by $\mathcal{F}$ such that for each $G \in \mathcal{F}$, R_G is a binary relation on P_G^* satisfying*

$$R_G(\vec{x}, \vec{y}) \iff R_G(\varphi(\vec{x}), \varphi(\vec{y}))$$

for all automorphisms $\varphi : G \to G$.

From the perspective of a generic algorithm, $\vec{x}$ represents inputs and $\{\vec{y} : R_G(\vec{x}, \vec{y})\}$ represents the set of possible outputs, or *the* output when R_G is a function.

For the sake of readability, when defining a particular generic relation we may explicitly list the components of $\vec{x}$ and $\vec{y}$, using Greek letters to denote group elements and Latin letters to denote integers, with the understanding that the relation is not defined for parameters of the wrong type. With the same understanding we will use functional notation where appropriate. Thus we may write $f(\alpha, k) = \alpha^k$ to define the exponentiation function rather than the more cumbersome $R(\vec{x}, \vec{y}) = \left\{ \left([(0, \alpha), (1, k)], [(0, \beta)] \right) : \beta = \alpha^k \right\}$. Strictly speaking, we are defining a partial function on P_G^*, since the function $f(\alpha, k)$ is not defined for parameters of the wrong type: α must be a group element and k must be an integer.

We can now define some particular generic functions and relations of interest:

- **order**$(\vec{\alpha})$ is the size of the subgroup $\langle \alpha_1, \ldots, \alpha_k \rangle$ generated by $\vec{\alpha} = [\alpha_1, \ldots, \alpha_k]$. When $k = 1$, this may be denoted $|\alpha|$.

- **DL**(α, β) is the **discrete logarithm** of β with respect to α: the least positive integer e for which $\alpha^e = \beta$, or $\emptyset$ if no such e exists.

- **DL**$(\vec{\alpha}, \beta)$ is the **discrete logarithm** of β with respect to the vector $\vec{\alpha} = [\alpha_1, \ldots, \alpha_k]$: the lexicographically least vector of non-negative integers $\vec{e} = [e_1, \ldots, e_k]$ for which $\beta = \vec{\alpha}^{\vec{e}} = \alpha_1^{e_1} \cdots \alpha_k^{e_k}$, or $\emptyset$ if no such $\vec{e}$ exists.

- The **root**$([\alpha, k], [\beta])$ relation holds for all β such that $\beta^k = \alpha$ and **root**$([\alpha, k], \emptyset)$ when no such β exists.

- **membership**$(\vec{\alpha}, \beta)$ is 1 if β is in the subgroup generated by $\vec{\alpha}$ and 0 otherwise.

- **isomorphic**$(\vec{\alpha}, \vec{\beta})$ is 1 if the subgroups generated by $\vec{\alpha}$ and $\vec{\beta}$ are isomorphic and 0 otherwise.

- **isAbelian**() is 1 if G is abelian and 0 otherwise. Other recognition functions such as **isCyclic()**, **isSimple()**, **isSolvable()**, ..., are defined analogously.
- $\boldsymbol{\lambda(G)}$ returns the **exponent** of G, the least positive integer N s.t. $\alpha^N = 1_G$ for all $\alpha \in G$.
- The **generators**$(\emptyset, \vec{\alpha})$ relation holds for all $\vec{\alpha} = [\alpha_1, \ldots, \alpha_k]$ which generate G.

Each of the definitions above holds for all $G \in \mathcal{F}$. The function $DL(\alpha, \beta) \neq \emptyset$ only when $\beta \in \langle \alpha \rangle$, but it is well defined for any group G and all $\alpha, \beta \in G$.

Note that the function *isAbelian* takes no inputs and is constant (0 or 1) for any particular G, as is the function $\lambda(G)$. An algorithm computing the *generators* relation takes no inputs and outputs a list of generators for G. These three examples must be computed using a randomized black box.

Any of the definitions above can be augmented with additional inputs that may be of use to an algorithm. For example, the function $|\alpha|$ may be extended to a (partial) function $order(\alpha, E)$ where E is intended to specify an exponent of α, an integer such that $\alpha^E = 1_G$. This function is only defined for inputs which satisfy this relationship, and an algorithm computing $order(\alpha, E)$ may assume this to be so.[3] The correctness criteria defined below only hold an algorithm responsible for its behavior on inputs which lie in the domain of the function or relation it is computing.

1.2 Generic Group Algorithms

To complete the model of generic computation it is necessary to define correctness criteria and complexity metrics. We first consider correctness. For a generic algorithm to compute a generic function f, it must correctly compute f_G on all inputs in its domain for every finite group G, using any black box $\mathcal{B}(G)$. More generally, if R is a generic relation, we have the following:

Definition 1.5. *A* **(deterministic) generic algorithm** $\mathcal{A}(\vec{x})$ *for* $R(\vec{x}, \vec{y})$ *satisfies*

$$R_G\left(\vec{x}, \mathcal{B}^{-1}\left(\mathcal{A}(\mathcal{B}(\vec{x}))\right)\right)$$

for all $G \in \mathcal{F}$*, any black box* $\mathcal{B}(G)$*, and all* $\vec{x}$ *in the domain of* R_G*.*

[3] Problems with this characteristic are sometimes called *promise* problems.

For probabilistic and non-deterministic algorithms, the definition above may be extended using an additional input $\tau \in \{0,1\}^t$ to represent a read-once sequence of random bits (or non-deterministic choices). If $\mathcal{A}(\vec{x})$ is a probabilistic (non-deterministic) algorithm, we let $\mathcal{A}'(\vec{x}, \tau)$ denote the deterministic algorithm obtained by letting the bits in τ determine the probabilistic (non-deterministic) choices made during the execution of $\mathcal{A}$, including the values of ρ passed to the randomized black box whenever a random group element is requested. The read-once property of τ means that ρ must contain un-read bits; the algorithm does not have access to the random bits used by the black box. A probabilistic algorithm need not make use of this facility; the definition below also applies to a standard black box as well.

Definition 1.6. *A* **probabilistic generic algorithm** $\mathcal{A}(\vec{x})$ *for* $R(\vec{x}, \vec{y})$ *satisfies*

$$R_G\Big(\vec{x}, \mathcal{B}^{-1}\big(\mathcal{A}'(\mathcal{B}(\vec{x}), \tau)\big)\Big)$$

for all $G \in \mathcal{F}$*, any (randomized) black box* $\mathcal{B}(G)$*, all* $\vec{x}$ *in the domain of* R_G*, and at least* $c2^t$ *values of* $\tau \in \{0,1\}^t$*, where* $c > 1/2$*.*

The definition above corresponds to a Monte Carlo algorithm with 2-sided error, or an Atlantic City algorithm. It is assumed that τ is finite in length, which may force an algorithm to fail simply because it "runs out" of random bits, however t may be chosen to ensure this happens with sufficiently low probability. The choice of the constant $c > 1/2$ is arbitrary. The confidence that an algorithm correctly computes a generic function can be boosted by repeatedly executing the algorithm and choosing a plurality of the output values, or by verifying the results (when practical) and re-executing the algorithm as required.[4]

The success probability is computed over the random choices contained in τ (which may include the selection of random group elements), but not over the particular group, black box, or input vector; the algorithm must achieve the required success probability in every case. Note, however, that this does not require an algorithm to be successful in the face of an "adaptive" black box. When analyzing an algorithm, an "adversary" may choose the black box based on an examination of the algorithm's program and the particular group and input vector, but this choice is fixed prior to any coin flips (the generation of τ). In

[4]For relations, boosting confidence by repetition may not be practical if many correct outputs are possible. In such cases either the output should be verifiable or the algorithm should ensure its output is correct to a specified level of confidence.

the case of a deterministic algorithm, an adversary can simulate the entire execution of the algorithm and thus effectively "adapt" the choice of black box based on the algorithm's simulated behavior, but a probabilistic algorithm's coin flips remain sacrosanct.[5]

Definition 1.7. *A* **non-deterministic generic algorithm** $\mathcal{A}(\vec{x})$ *for* $R(\vec{x}, \vec{y})$ *satisfies*

$$R_G\left(\vec{x}, \mathcal{B}^{-1}\left(\mathcal{A}'(\mathcal{B}(\vec{x}), \tau)\right)\right)$$

for all $G \in \mathcal{F}$*, any randomized black box* $\mathcal{B}(G)$*, all* $\vec{x}$ *in the domain of* R_G*, and at least one value of* $\tau \in \{0,1\}^t$*. For all other values of* τ*, algorithm* $\mathcal{A}$ *terminates and outputs "REJECT".*

Every possible execution of a non-deterministic algorithm must output the correct value or "REJECT". A non-deterministic algorithm is not allowed to make (undetected) mistakes, and must always terminate. As noted above, a non-deterministic algorithm can effectively select group elements non-deterministically via the "randomized" black box by passing in an un-read subsequence ρ from τ.

The classes of probabilistic and non-deterministic generic algorithms clearly both contain the class of deterministic generic algorithms, but neither is obviously contained in the other. This state of affairs is analogous to the relationship between the complexity classes BPP and NP.

Before defining complexity metrics for generic algorithms, there is one aspect of the definitions above that merits further discussion. The behavior of a black box when one of its interfaces is passed an invalid identifier (a bit-string which is not in $\mathcal{I}$) is unspecified. We consider the algorithm's behavior to be incorrect in this scenario (e.g. the black box goes into an infinite loop or self destructs). In light of this, there is an important consequence of the definitions above:

Remark 1.1. *Every identifier used by any type of generic algorithm in a call to a black box interface is either an input to the algorithm or an identifier previously obtained via the black box.*

A generic algorithm must satisfy the correctness criteria for *every* black box. If it tries to "guess" a valid identifier, there exist black boxes for which the guess will be wrong (more than half the time) since the identifier set $\mathcal{I} \subseteq \{0,1\}^\ell$ may be quite sparse and

[5]Not all models grant this assurance, which may impact statements regarding the expected running times of certain algorithms, e.g. those that use Pollard's rho method.

still satisfy $\ell = O(\lg |G|)$. Even in the case of a non-deterministic algorithm, it may non-deterministically guess an unknown identifier, but it still must verify that the guess is correct before it can safely use the identifier in a call to the black box (otherwise the black box might go into an infinite loop causing the algorithm to be incorrect). The only way to verify an unknown identifier is to non-deterministically request a group element from the black box and check if the identifier matches, but then Remark 1.1 applies.

The practical consequence of Remark 1.1 is that it prevents any "cheating" on the complexity metrics we now define. It will be convenient to ignore certain unnecessary or ineffective calls to the black box when measuring a generic algorithm's complexity. Once the identifier for the element 1_G has been obtained via $\mathcal{B}_{id}()$, any group operations involving the identity element can be readily avoided by a careful algorithm. Rather than requiring algorithms to be careful, we will simply ignore all operations involving the identity (including calls to $\mathcal{B}_{id}$), regarding these as *trivial* operations.

Definition 1.8. *The* **time complexity** *of a generic group algorithm is the number of non-trivial interface calls to the black box, referred to as* **group operations**. *The* **space complexity** *of a generic group algorithm is the maximum number of identifiers simultaneously stored by the algorithm, also called* **group storage**.

Asymptotic complexity results are typically stated in terms of $n = \lg |G|$ or some other appropriate measure of the problem size (note that identifiers have length $\ell = \Theta(n)$), but it will often be convenient to use $N = |\alpha|$, or $N = |G|$. The terms "polynomial-time", "exponential-time", "near linear-time", etc., may all be used in conjunction with the complexity metrics defined above with the usual meaning. An algorithm which uses $\Theta\left(N^{1/2}\right)$ group operations is an exponential-time algorithm, while an algorithm which uses $O\left(n^2/\log n\right)$ group operations would be a sub-quadratic algorithm.[6]

These complexity metrics ignore any non-group computations performed by the algorithm. This strengthens the statement of lower bounds, since it effectively grants algorithms unlimited time and space for non-group operations. In the case of upper bounds, the actual running time of most generic algorithms is overwhelming dominated by group operations for large problems. It may generally be assumed that each group operation requires at least as much time as it takes to multiply two n-bit integers (often more will be required).

[6]Throughout this thesis, $\log n$ denotes the natural logarithm while $\lg n$ refers to the binary logarithm. These may be used interchangeably in asymptotic expressions, however, we typically use the more indicative form.

Provided that the number of non-group operations (e.g. integer arithmetic on n-bit integers, comparing two n-bit strings) is less than the number of group operations, the actual running time will be within a constant factor of the time spent performing group operations (similar comments apply to space). For all the algorithms presented here, the time required by group operations will constitute the vast majority of the running time in practice (see Chapter 11).

1.3 Comparison and Discussion

The first formal model of generic group computation was introduced in the seminal 1984 paper of Babai and Szemeredi [7], where they define the concept of a *black-box group*. Since then, numerous models of generic group computation have been considered. We do not attempt a survey here, but note that all models share the common feature that the representation of group elements is hidden. Two major distinguishing features are:

1. Can algorithms compare group elements without consulting the black box?

2. Does the black box provide support for random group elements?

In the black-box groups of Babai and Szemeredi the answer to both questions is "no"; their black box provides a function for comparing a group element with the identity, and algorithms are given a set of generators for the group. In the model defined here, the answer to both questions is "yes". To avoid confusion, we will use the term *generic group* to refer to a group $G \in \mathcal{F}$ together with a black box for G as defined in this chapter, preserving the original meaning of "black-box group" as defined by Babai and Szemeredi.

The approach taken here is a generalization of that used by Shoup in [96]. Following Shoup, we assume that the identification function is bijective. This type of model is also known as a *uniquely encoded* black-box group. Including this capability strengthens the applicability of lower bounds, and provides a better platform for building efficient algorithms in practice. Indeed, many widely-used generic algorithms depend on this feature (e.g. Shanks' baby-steps giant-steps method). The utility of unique encoding in practice, combined with the strong lower bounds that may be proven even when algorithms are granted this capability, make a compelling argument for its inclusion in the model.

Unique encoding does not usually impose an undue burden on the black box. Typically either there is a natural canonical representation for a group element (an integer, a matrix, a permutation, a polynomial, etc...) or a black-box implementation of any type may be impractical. Examples of the latter case include finitely presented groups, in which the task of comparing two elements may be undecidable (the existence of such groups was shown by Novikov [79] and Boone [18]).[7] There are cases that fall between these extremes (e.g. quotient group representations), however, trading unique encoding for a comparison operation may simply shift the burden of work within the black box.

The black box definition above differs from Shoup's oracle in that G may be an arbitrary group and the inputs to the black box are simply identifiers rather than indices into a list of previously acquired identifiers (but see Remark 1.1 above).

The notion of a randomized black box is a natural extension that is rapidly becoming standard [20, 114]. Unless an algorithm is given a set of generators (as in the black box groups of [7]) there is no way for it to access group members not generated by its inputs unless a randomized black box is used. Given that many problems can be solved quite efficiently without a set of generators (e.g. the randomized abelian test of Algorithm 1.2), it seems unnecessary to require a set of generators as part of the model; they can always be specified as inputs for a particular problem. Conversely, given a set of generators, it is not a trivial problem for a generic algorithm to generate a random group element (see [4] for a discussion). In many cases the black box can do this much more effectively then the algorithm because it has direct access to the group representation, and it is certainly no more difficult for the black box in any event. There are efficient heuristic methods for generating nearly uniform random elements that may not be fully independent (the product replacement algorithm of [34], see also [82]), and somewhat less efficient methods that generate fully independent random elements [3].[8]

As noted earlier, the correctness criteria for a probabilistic generic algorithm fix the choice of the black box prior to the algorithm's execution, and, in particular, any coin flips. A probabilistic algorithm can effectively randomize against a worst-case identification map. It may be that for every sequence of coin flips there exists a bad black box, so long as for every black box most sequences of coin flips are good. This approach strengthens

[7] It is worth noting that many finitely presented groups *can* be efficiently implemented as black boxes, e.g. polycyclic groups.

[8] For a recent development in this area see [39].

lower bounds, since it is easier for an algorithm to be correct and/or have a better expected running time. It also represents the actual behavior of most black boxes in practice. If a particular probabilistic algorithm happens to have the admirable virtue of being correct against an adversary who knows every coin flip in advance, so much the better. Note that this doesn't apply to deterministic algorithms, they always have to handle the worst possible scenario.

The formal model defined in this chapter is motivated not only by theory, but also by a desire to make both generic algorithms *and* black boxes easy to implement in practice. There are obvious tradeoffs, but on balance the choices made here seem to work well. As practical examples, all the algorithms presented in this thesis were implemented and tested using black boxes with a wide variety of group representations, including the multiplicative group of integers modulo m, products of additive cyclic groups, ideal class groups, elliptic curves, permutation groups, and matrix groups. While conceived largely for theoretical purposes, in practice, a formal model of group computation is sound software engineering. It enables algorithms and black boxes to be designed, analyzed, and improved independently.

Chapter 2

The Complexity of Order Computations

Chapter Abstract

We consider the complexity of computing $|\alpha|$, proving an $\Omega\left(N^{1/3}\right)$ lower bound and reducing integer factorization to random order computations. Neither result is fundamentally new, but they provide useful illustrations of the model and generalize prior work. We also introduce a number of concepts that facilitate the analysis of order algorithms.

In the most general case, an order algorithm is given a single input $\alpha \in G$, and has essentially no information about $|\alpha|$ or $|G|$. There is an implicit bound on $|G|$ given by the fixed size of identifiers in the set $\mathcal{I} \subseteq \{0,1\}^\ell$, which implies $|G| \leq 2^\ell$. This does give a polynomial bound on $|G|$, since $\ell = O\left(\lg|G|\right)$, but it may be much larger than $|G|$. Practical example of groups with unknown order are ideal class groups and groups of points on elliptic curves. In both cases it is entirely straight forward to implement a black box for a particular group without knowing its size, other than an upper bound on the number of bits required to represent a group element.

In the particular cases of class groups and elliptic curves, we actually know fairly tight bounds on $|G|$: $O\left(|G|\log|G|\right)$ and $|G| + O\left(|G|^{1/2}\right)$ respectively. We will see in Chapter 9 that a bound $|G|^c$ with $c < 2$ will be of great assistance in determining the structure of an abelian group, but even if we know $|G|$ within a constant factor, this is essentially no help in computing $|\alpha|$ efficiently. Our complexity measure for order algorithms is in terms of $|\alpha|$, not $|G|$, and $|\alpha|$ may be much smaller than $|G|$. Computing $|\alpha|$ for an element with order 2

should not be hard in any group, no matter how big.

On the other hand, if we happen to know the prime factors of $|G|$, or any set S containing the possible prime factors of $|\alpha|$, computing $|\alpha|$ is easy. This can be done in time linear in $|S|$ and nearly linear in $\lg |\alpha|$. Algorithms which compute $|\alpha|$, given knowledge[1] of the possible prime factors of $|\alpha|$, are called *fast order algorithms*, and are the subject of Chapter 7. For the moment we assume neither $|G|$ nor its prime factors are known. If an order algorithm can identify a reasonably small superset of the prime factors of $|\alpha|$, say by finding a multiple of $|\alpha|$ and factoring it, a fast order algorithm may then be applied.

A third example of a group with unknown order is the multiplicative groups of integers $\mathbb{Z}_M^*$, where the prime factors of M are unknown. In this case we would be better off factoring M rather than attempting to determine the order of an arbitrary $\alpha \in \mathbb{Z}_M^*$. Given the prime factorization of M, we can easily compute

$$\phi(M) = M \prod_{p|M} \left(1 - \frac{1}{p}\right) = |\mathbb{Z}_M^*|,$$

and then factor $\phi(M)$ to obtain the possible prime factors of $|\alpha|$, since $|\alpha|$ must divide $|G|$. We give two proofs that factoring the integer M is easier than computing $|\alpha| = N$ with a generic algorithm. The first and most direct is an $\Omega\left(N^{1/3}\right)$ lower bound on the number of group operations required to compute $|\alpha|$, which is substantially larger than the complexity of all but the most naive factoring algorithms (in particular the deterministic algorithm of Pollard-Strassen uses $O\left(N^{1/4} \log^2 N\right)$ arithmetic operations [83, 103, 40]). The second proof is a reduction showing that a generic order algorithm can be used to construct a probabilistic algorithm for factoring integers.

Neither of these results is particularly new. An exponential lower bound on order computations in black-box groups is given by Babai in [5], and the factoring reduction is essentially the same as that given by Bach [8] for the discrete logarithm (see also [46]), or one based on the group exponent described by Shoup in [97]. Giving explicit proofs in our model sharpens both results. The details of the lower bound in particular are illuminating: the bound proven is $\Omega\left(N^{1/3}\right)$ rather than $\Omega\left(N^{1/2}\right)$. It seems unlikely that the $\Omega\left(N^{1/3}\right)$ bound is tight, but we shall see that an $\Omega\left(N^{1/2}\right)$ bound does not exist.

[1] We make S more specific by defining a *factored exponent* of α, essentially a multi-set of the prime factors of $|\alpha|$, in Chapter 7.

2.1 Analyzing Order Algorithms

Consider the execution of a generic algorithm $\mathcal{A}$ which receives the identifier of a single group element $\alpha = \alpha_0$ as input. As the algorithm proceeds, it makes a series of calls to the black box, obtaining a sequence of (not necessarily distinct) identifiers $\alpha_0, \alpha_1, \ldots, \alpha_t$, one for each call made to the black box. For convenience, we prefix the sequence with 1_G and ignore all further operations involving the identity element.

Definition 2.1. *An* **identification sequence** *for $\mathcal{A}(\alpha)$ is the sequence $1_G, \alpha_0, \alpha_1, \ldots, \alpha_t$ where $\alpha_0 = \alpha$ and α_i is the value returned by the ith non-trivial call to the black box during an execution of generic algorithm $\mathcal{A}$ on input α.*

Given an identification sequence for a deterministic algorithm $\mathcal{A}(\alpha)$, we define a sequence of ordered pairs by associating (j, k) with the operation $\mathcal{B}_{prod}(\alpha_j, \alpha_k)$ and $(j, -1)$ with the operation $\mathcal{B}_{inv}(\alpha_j)$.

Definition 2.2. *A* **computation chain** *is a sequence of ordered pairs of integers (x_i, y_i) for $1 \leq i \leq t$ which satisfies $0 \leq x_i < i$ and $-1 \leq y_i < i$.*

There may be more than one choice of computation chain compatible with an identification sequence due to duplicated identifiers (j and k above may not be uniquely defined). In such cases we arbitrarily select the first among duplicate values. There are two different reasons why elements in an identification sequence may not be distinct. In the trivial case, the algorithm may have simply computed the same thing twice, or discovered some relation which must hold in every group, such as $(\alpha^2)^3 = (\alpha^3)^2$. In the non-trivial case, the algorithm has found a relation which constrains the possible values of $|\alpha|$. To help extract information contained in non-trivial relations (or to prove no such relations exist), we define the exponent sequence of a computation chain.

Definition 2.3. *The* **exponent sequence** $0, 1, e_1, \ldots, e_t$ *of the computation chain (x_i, y_i) is the sequence of integers defined by $e_{-1} = 0, e_0 = 1$, and*

$$e_i = \begin{cases} -e_j & \text{if} \quad (x_i, y_i) = (j, -1); \\ e_j + e_k & \text{if} \quad (x_i, y_i) = (j, k),\ k \geq 0. \end{cases}$$

More generally, any sequence of integers $0, 1, e_1, \ldots, e_t$, where each e_i is the negation or sum of previous elements, is called an exponent sequence of length t.

An exponent sequence is simply an addition chain with negation steps allowed (or a restricted form of an addition-subtraction chain).

Definition 2.4. *An* **addition chain** *of length t is a sequence of integers $e_0, e_1, \ldots, e_t$ such that $e_0 = 1$ and for $i > 0$, $e_i = e_j + e_k$ for some $j, k < i$. An* **addition-subtraction chain** *of length t is a sequence of integers $e_0, e_1, \ldots, e_t$ such that $e_0 = 1$ and for $i > 0$, $e_i = e_j \pm e_k$ for some $j, k < i$.*

Addition chains play a key role in exponentiation algorithms, see Knuth [68, Section 4.6.3] for an overview. We consider addition chains further in Chapter 7 when we examine fast order algorithms.

In terms of the identification sequence for $\mathcal{A}(\alpha)$, the exponent sequence corresponds to mapping α to the integer 1 and performing operations in the free group $\mathbb{Z}^+$. The element $e_{-1} = 0$ corresponds to the identity element which we implicitly assume is always available. The non-trivial relations in the identification sequence of an order algorithm are precisely those where the corresponding elements of the exponent sequence differ.

Lemma 2.1. *Let $\alpha_{-1}, \alpha_0, \ldots, \alpha_t$ be an identification sequence for a deterministic generic algorithm $\mathcal{A}(\alpha)$, and let $e_{-1}, e_0, \ldots, e_t$ be the exponent sequence of the associated computation chain. Then*

$$(1) \quad \alpha_i = \alpha^{e_i}, \qquad \text{and} \qquad (2) \quad \alpha_i = \alpha_j \iff e_i \equiv e_j \bmod |\alpha|,$$

for all $-1 \leq i, j \leq t$.

Proof. The exponentiation map $k \mapsto \alpha^k$ is a group homomorphism from $\mathbb{Z}^+$ to $\langle \alpha \rangle \subseteq G$. (1) may be proven by noting that $\alpha_{-1} = 1_G = \alpha^0 = \alpha^{e_{-1}}$ and $\alpha_0 = \alpha = \alpha^1 = \alpha^{e_0}$, then inductively applying the exponentiation map to the relations on $e_1, \ldots, e_t$ implied by the computation chain. For (2), note that the exponentiation map has kernel $N\mathbb{Z}$, where $N = |\alpha|$, giving an isomorphism $\varphi : \mathbb{Z}/N\mathbb{Z} \to \langle \alpha \rangle$. We have $\varphi(\overline{e_{-1}}) = \varphi\left(\overline{0}\right) = 1_G = \alpha_{-1}$ and $\varphi(\overline{e_0}) = \varphi\left(\overline{1}\right) = \alpha = \alpha_0$, and it may be shown inductively that $\varphi(\overline{e_i}) = \alpha_i$ by applying the computation chain. □

In specific situations we may extend our definition of an exponent sequence to accommodate probabilistic algorithms. When doing so we will ensure that Lemma 2.1 still holds. This lemma will be useful in proving lower bounds, but for the moment consider it from the perspective of an order algorithm. As long as the exponents do not become too large,

an algorithm can easily compute its own exponent sequence as it proceeds. If at any point the algorithm finds $\alpha_i = \alpha_j$ but $e_i \neq e_j$, then $E = |e_i - e_j|$ is a non-zero multiple of $|\alpha|$ (an *exponent* of α). If E is known to be minimal then $E = |\alpha|$, otherwise, if the factorization of E is known or can be readily computed, a fast order algorithm can quickly determine $|\alpha|$.

On the other hand, Lemma 2.1 also implies that for $e_i \neq e_j$, if $\alpha_i \neq \alpha_j$ then $E = |e_i - e_j|$ is *not* a multiple of $|\alpha|$. This gives an algorithm useful information, as it rules out every divisor of E as a possibility for $|\alpha|$. There is a limit, however, to how rapidly an algorithm can accumulate knowledge in this fashion.

Lemma 2.2. *Let S be a set of n integers, each with absolute value at most 2^t. Let $c(S)$ denote the number of distinct primes which divide the difference of a pair in S. Then $c(S) = O(n^2 t/\log t)$.*

Proof. Let m be the positive difference of a pair in S, and let k be the number of distinct primes dividing m. If $P = \prod_{p \leq p_k} p$ is the product of the first k primes, then $P \leq m \leq 2^{t+1}$. Applying the prime number theorem (see Section 6.1), we have

$$k \log k \leq k \log p_k = \pi(p_k) \log p_k \sim p_k \sim \vartheta(p_k) = \sum_{p \leq p_k} \log p = \log P \leq (t+1) \log 2,$$

hence $k = O(t/\log t)$. There are $\binom{n}{2} = O(n^2)$ distinct pairs in S, and the lemma follows. □

2.2 An Exponential Lower Bound

We are now in a position to prove our first result, an $\Omega\left(N^{1/3}\right)$ lower bound on order algorithms. The statements and proofs of lower bounds require a degree of care; it is important to be clear exactly what needs to be proven.

We typically argue by contradiction, supposing that a generic algorithm $\mathcal{A}(\vec{x})$ computes some generic function or relation $\mathcal{R}(\vec{x}, \vec{y})$, using at most t group operations. We then construct an exceptional problem instance for which the algorithm fails to compute $\mathcal{R}$. Specifically, we must show there exists a particular group G, black box $\mathcal{B}(G)$, and a set of input values $\vec{x}$ for which $\mathcal{A}(\vec{x})$ is incorrect, using the correctness criteria defined in Chapter 1. In the case of a probabilistic algorithm, this means we need to construct a randomized black box, including a specific function $\mathcal{B}_{rand}(\rho)$, such that $\mathcal{A}'(\vec{x}, \tau)$ fails for at least half the values of τ (the sequence of coin-flips). It is *not* sufficient to merely show an

exceptional problem instance for each τ. This distinction is a key feature of the model: it gives algorithms more power and strengthens the impact of lower bounds.

Theorem 2.3. *Let $\mathcal{A}$ be a probabilistic generic order algorithm. For all sufficiently large $M = 2^\ell$ there is a prime p, $\sqrt{M} < p \leq M$, and a randomized black box $\mathcal{B}(C_p)$ for the cyclic group $C_p = \langle \alpha \rangle$, such that the expected number of group operations used by $\mathcal{A}$ on input α is*

$$\Omega\left(N^{1/3}\right),$$

where $N = p = |\alpha|$.

Proof. Fix $M = 2^\ell$. Suppose for the sake of contradiction that $\mathcal{A}$ never uses more than $t = cM^{1/3}$ group operations when computing the order of $\alpha \in C_p$, for $\sqrt{M} \leq p \leq M$, where c is a constant to be determined.

Fix a particular sequence τ of random coin flips used by $\mathcal{A}$ so that $\mathcal{A}'(\alpha, \tau)$ is a deterministic algorithm. We will simulate the execution of $\mathcal{A}'(\alpha, \tau)$, using random identifiers α_0 for α and α_{-1} for 1_G, and construct both an identification sequence $\alpha_{-1}, \alpha_0, \alpha_1, \ldots$, and an exponent sequence $0, 1, e_1, \ldots$, for $\mathcal{A}'(\alpha, \tau)$ as we proceed. We extend our notion of exponent sequences to include random group elements. Let $e_i = \rho_i$ if the ith call to the black box is $\mathcal{B}_{rand}(\rho_i)$, where $\rho_i \in \{0,1\}^{2\ell}$ is a substring of the random bits in τ as described in Definition 1.6. We then carry out the simulation by choosing a distinct random identifier in $\{0,1\}^\ell$ for each new exponent that occurs in the exponent sequence. Note that we can always compute the next exponent using the values of previously defined exponents. We continue the simulation until either $\mathcal{A}'(\alpha, \tau)$ terminates successfully and outputs N, fails in some fashion, or after t steps. In the latter two cases, let $N = 0$. We now pick a random prime p in the interval $(M^{1/2}, M]$, so that $\ell = O(\log |C_p|)$, and construct a randomized black box $\mathcal{B}(C_p)$ for $C_p = \langle \alpha \rangle$ as follows:

1. Simultaneously define $\mathcal{I} \subseteq \{0,1\}^\ell$ and the identification map $\mathcal{B} : C_p \to \mathcal{I}$ by letting $\mathcal{B}(1_G) = \alpha_{-1}$, $\mathcal{B}(\alpha) = \alpha_0$, and letting $\mathcal{B}(\alpha^i) = \alpha_j$, where e_j is the least exponent congruent to i mod p, if any, and otherwise choosing a random unused identifier from $\{0,1\}^\ell$.

2. Define the function $\mathcal{B}_{rand} : \{0,1\}^{2\ell} \to \mathcal{I}$ by $\mathcal{B}_{rand}(\rho) = \mathcal{B}(\alpha^\rho)$.

Note that (2) ensures that Lemma 2.1 applies to our exponent sequence. This defines a particular randomized black box $\mathcal{B}(C_p)$. Moreover, all the choices were made randomly

and independently, and could have been made in any order. For the sake of uniformity, let $\sigma \in \{0,1\}^k$ be a sequence of random bits of some fixed length sufficient to determine the random choices of any particular construction above. We are effectively padding our random choices as required to obtain a uniform distribution over the values of σ. Each σ generates a particular black box, but many σ may produce the same black box.

It is not necessarily the case that our simulation of $\mathcal{A}'(\alpha, \tau)$ actually corresponds to its behavior with the black box $\mathcal{B}(C_p)$. This will only be true if the e_i are all distinct modulo p. The exponent sequence has length at most t, and the maximum size of any exponent is at most $2^{2\ell+t}$: each group operation either returns a random exponent less than $2^{2\ell}$ or an exponent that is at most twice any previous exponent. Since $\ell = O(\log t)$ and the length of the exponent sequence is at most t, Lemma 2.2 implies that $O(t^3/\log t)$ primes divide the difference of some pair of exponents. By the prime number theorem, there are $\pi(M) - \pi(\sqrt{M}) = \Theta(M/\log M)$ primes in $(M^{1/2}, M]$, so we may choose the constant c such that fewer than $1/3$ of these primes divide the difference of any pair of exponents. It follows that with probability at least $2/3$, the exponents are all distinct modulo p and $p \neq N$, where N is zero or the value output by $\mathcal{A}'(\alpha, \tau)$. For these cases, the black box $\mathcal{B}(C_p)$ will cause $\mathcal{A}'(\alpha, \tau)$ to behave as in the simulation above and fail to correctly compute $|\alpha|$.

For any fixed τ, we have shown that, with probability at least $2/3$, a randomly constructed black box will cause $\mathcal{A}'(\alpha, \tau)$ to be incorrect. We now use a pigeon-hole argument to show that some black box causes $\mathcal{A}'(\alpha, \tau)$ to be incorrect for at least $2/3$ of the values of τ. Consider a 0-1 matrix with rows corresponding to values of τ and columns corresponding to values of σ, where a 1 indicates a failure of $\mathcal{A}'(\alpha, \tau)$ when using the black box generated by σ. We have shown that every row is at least $2/3$ full of 1's. It follows that one of the columns is at least $2/3$ full of 1's. For the black box corresponding to this column, $\mathcal{A}(\alpha)$ fails with probability $2/3$, contradicting our hypothesis. Thus $\mathcal{A}(\alpha)$ must use more than t group operations for at least a constant fraction of the possible values of τ, implying an expected running time of $\Omega(N^{1/3})$. □

Corollary 2.4. *Every deterministic generic order algorithm has time complexity* $\Omega\left(N^{1/3}\right)$. *The expected running time of any probabilistic generic order algorithm is* $\Omega\left(N^{1/3}\right)$.

The proof of Theorem 2.3 constructs a black box that simulates a free cyclic group, one for which no non-trivial relations exist. In essence, the proof shows that a generic

algorithm cannot distinguish a random cyclic group with prime order from a free group using $O\left(N^{1/3}\right)$ group operations, because it cannot find a non-trivial relation. This is a stronger statement than Theorem 2.3. It may be expressed using the concept of *pseudo-free groups* introduced by Rivest in [89].

Rivest considers a more general model of "computational groups" in which an algorithm has access to the representation, but notes that the theory may also be applied to black-box groups. In the terminology of [89] we could state our result as:

Cyclic groups are generic pseudo-free groups.

We use the term "generic" to refer to the model defined in Chapter 1 (uniquely encoded black-box groups with random group elements). To make this statement precise, we consider an infinite family of cyclic groups $\{C_1, C_2, \ldots\}$. A random integer $N \in [1, M]$ will contain a prime factor large enough to ensure that no generic algorithm can distinguish C_N from $\mathbb{Z}^+$ in polynomial time with non-negligible probability. See [74] for related work on the notion of pseudo-free groups.

There are two different algorithms commonly used to perform order computations that both have complexity $\Theta\left(N^{1/2}\right)$. It is natural to ask whether the $\Omega\left(N^{1/3}\right)$ lower bound of Theorem 2.3 can be improved to $\Omega\left(N^{1/2}\right)$. The answer is an emphatic "no", as will be made evident in Chapter 4. This is in contrast to the discrete logarithm problem, where we have tight $\Theta\left(N^{1/2}\right)$ bounds. There is a key difference between order computation and the discrete logarithm problem. It is enough for an order algorithm to find any reasonably small exponent E such that $\alpha^E = 1_G$; the order of α is then a divisor of E. If $\alpha^E = \beta$, however, it need not be the case that $DL(\alpha, \beta)$ divides E, rather $E \equiv DL(\alpha, \beta) \bmod |\alpha|$.

2.3 Factoring Reduction

The task of factoring the exponent E may seem daunting, but if a generic order algorithm is able to efficiently find exponents of moderate size, factoring an integer presents no major challenge, as demonstrated by the following algorithm.

Algorithm 2.1. *Let $\mathcal{A}(\alpha)$ be any generic algorithm which computes a multiple of $|\alpha|$, and let N be an odd integer. The following probabilistic algorithm attempts to find a non-trivial factor of N:*

1. Select a random integer $x \in [1, N-1]$. If $\gcd(x, N) > 1$, return $\gcd(x, N)$.

2. Compute $E \leftarrow \mathcal{A}(x)$ by simulating a black box for $\mathbb{Z}_N^*$. Let E' be the odd part of E.

3. Set $y \leftarrow x^{E'} \bmod N$. While $y^2 \not\equiv 1 \bmod N$, set $y \leftarrow y^2 \bmod N$.

4. If $y \equiv \pm 1 \bmod N$ then *FAIL*, otherwise return $\gcd(y+1, N)$.

This algorithm is essentially that used by Bach in [8] to reduce factoring to the discrete logarithm problem.[2] It is not difficult to see that if the algorithm succeeds, its output is a non-trivial factor of N, since $y^2 \equiv 1 \bmod N$ and $y \not\equiv \pm 1 \bmod N$ together imply that $y \equiv -1 \bmod q$ for some, but not every, maximal prime power $q|N$.

Proposition 2.5. *Let N be an odd integer that is not a prime power. The probability that Algorithm 2.1 finds a non-trivial divisor of N is at least $1/2$.*

Proof. Let $\lambda = \lambda(N)$ be the exponent of $\mathbb{Z}_N^*$ (the lcm of $|\alpha|$ over $\alpha \in \mathbb{Z}_N^*$), and let x and y be as in Algorithm 2.1. $\mathbb{Z}_N^*$ contains an element of order 2 (namely -1), so λ is even. We have

$$x^{\lambda/2} \not\equiv 1 \bmod N \quad \Longrightarrow \quad y = x^{\lambda/2},$$

since for such an x, both $x^{\lambda/2}$ and y are the unique element of order 2 in the cyclic subgroup $\langle x \rangle$. Thus y is independent of the particular exponent E computed in step 2, and the algorithm succeeds whenever $x^{\lambda/2} \not\equiv \pm 1 \bmod N$.

Consider the image H of the $(\lambda/2)$-power map. H is a subgroup of $\mathbb{Z}_N^*$ with even order (the minimality of λ ensures H is non-trivial, and every non-trivial element of H must have order 2). It suffices to show H contains a non-trivial element $y \not\equiv -1 \bmod N$, since then at least half the elements in H cannot be congruent to $\pm 1 \bmod N$, implying $x^{\lambda/2} \not\equiv \pm 1 \mod N$ for at least half the $x \in \mathbb{Z}_N^*$.

Let $N = qM$, where q is an odd prime power chosen to maximize the power of 2 which divides $\phi(q) = |\mathbb{Z}_q^*|$, and $M \neq 1$ is odd and coprime to q. Let a generate the cyclic group $\mathbb{Z}_q^*$. By the Chinese remainder theorem there exists $x \in \mathbb{Z}_N^*$ satisfying:

$$\begin{aligned} x &\equiv a \mod q; & x &\equiv 1 \bmod M; \\ x^{\lambda/2} &\equiv -1 \bmod q; & x^{\lambda/2} &\equiv 1 \bmod M. \end{aligned}$$

[2] In [97, pp. 264-265] Shoup describes a similar algorithm for the case where $E = \lambda(N)$ is given. Shoup cites a deterministic algorithm of Miller [75] as the origin and ascribes the probabilistic version to folklore.

Thus $x^{\lambda/2} \not\equiv \pm 1 \bmod N$ as desired. □

The proof above is a simplification of that found in [46, pp. 132-133]. In the case that N is not a prime power, the success probability can be made arbitrarily high by repeating the algorithm. If the algorithm never succeeds, then N is almost certainly a prime power[3] and the base can be easily extracted using a fast perfect-power algorithm: Bernstein's algorithm requires just slightly more than $O\left(\log N\right)$ arithmetic operations [12, 14].

Algorithm 2.1 is an efficient reduction. The black-box simulation may be accomplished with one arithmetic operation on integers modulo N per group operation (modular inversion is slightly more expensive than multiplication but can be done with $O\left(n^2\right)$ bit operations). The remaining gcd, exponentiation, and squaring operations involve a total of $O\left(\lg E\right)$ arithmetic operations.

Corollary 2.6. *Given a generic algorithm $\mathcal{A}(\alpha)$ which computes an exponent E of α with $L(n)$ bits using $T(n)$ group operations (where $n = \lg|\alpha|$), there is a probabilistic algorithm which finds a non-trivial factor of a composite integer N, with high probability, using $O\left(T(n) + L(n)\right)$ arithmetic operations modulo N (where $n = \lg N$).*

Unless the exponent E is $O\left(N\right)$, applying Algorithm 3 to factor E may be expensive, since $O\big(T(L(n)) + L(L(n))\big)$ arithmetic operations would be required. Fortunately there are more efficient factoring algorithms available. One particular method involves order computations in the ideal class group $Cl(-N)$ of the number field $\mathbb{Q}[\sqrt{-N}]$. The same approach taken by Algorithm 2.1 may be applied to probabilistically find an element with order 2 in the class group. Using the quadratic form representation described in Chapter 10, such an element is highly likely to yield a non-trivial factor of N. The key advantage of this approach is that the group $Cl(-N)$ is substantially smaller than the group $\mathbb{Z}_N^*$ used in Algorithm 2.1, typically $\Theta\left(N^{1/2}\right)$.

In practice, if $E = O\left(N^2\right)$, then E can easily be factored in less time than required by the group operations to find E. Alternatively, if an algorithm can prove that E is the *least* exponent of α, then $E = |\alpha|$ and no factorization is required.

[3] Note that Algorithm 2.1 effectively performs a Miller-Rabin primality test [75, 87].

Chapter 3

Existing Order Algorithms

Chapter Abstract

There are two search techniques common to many generic algorithms: Pollard's rho method and Shanks' baby-steps giant-steps approach. We consider the particular challenges that arise when each is applied to order computations, where the unknown value $|\alpha|$ is effectively unbounded. We examine two generic order algorithms that represent the current state of the art, analyze their complexity, and discuss the advantages and disadvantages of each.

We now consider some specific order algorithms. There are two general approaches commonly used: Pollard's rho method [84] and the baby-steps giant-steps algorithm due to Shanks [95]. Both methods are also applicable to the discrete logarithm problem and integer factorization. Many variations of these algorithms have been considered in the literature [112, 102, 40, 37]. We focus here on the specific issues involved in applying these techniques to generic order algorithms.

The major challenge for order algorithms is the need to ensure that the running time is actually a function of $N = |\alpha|$. The upper bound $M = 2^\ell$ implied by the size of identifiers makes it entirely straight forward to obtain $O\left(\sqrt{M}\right)$ algorithms. However, even if M is as tight as possible, it is often the case that $|\alpha| \ll |G| \leq M$. Two typical examples are matrix and permutation groups. If $G = GL_d(\mathbb{F}_q)$ is the general linear group over a finite field, or any large subgroup thereof, it may be shown [78] that $|\alpha| = O\left(\sqrt{|G|}\right)$ for all $\alpha \in G$, and a similar statement applies to the symmetric group [70]. Any meaningful measure of performance for an order algorithm should be a function of N, not M.

3.1 The Rho Method

Pollard's rho algorithm [84] was originally conceived as a space-efficient way to compute discrete logarithms in $\mathbb{Z}_m^*$. The application of Pollard's method to generic group computations has been analyzed by Teske [110, 111, 109]. In [109] she gives a generic algorithm for computing the structure of an arbitrary abelian group specified by a set of generators. This algorithm can be applied to compute, in particular, the order of $\langle\alpha\rangle$ (which is certainly an abelian group, even if G is not). In this restricted case, only part of Teske's algorithm is needed and the constant factors in the expected running time can be improved.

The Main Idea

The rho method is a probabilistic generic algorithm which attempts to simulate a random walk through a cyclic group $G = \langle\alpha\rangle$. This is done by iterating an appropriately constructed function $f : G \to G$ to generate a sequence $\alpha_0, \alpha_1, \alpha_2, \ldots$, where $\alpha_0 = \alpha$ and $\alpha_{i+1} = f(\alpha_i)$. The algorithm also computes its exponent sequence $e_0, e_1, e_2, \ldots$, so that $\alpha_i = \alpha^{e_i}$. Since G is finite, it eventually happens that $\alpha_i = \alpha_j$ for some $i > j$, after which the sequence will repeat with period $i - j$. We denote the least such i by ρ.[1] The algorithm uses a space-efficient cycle detection method to find $\alpha_i = \alpha_j$ for some $i \geq \rho$, at which point $E = |e_i - e_j|$ will be a non-zero multiple of $|\alpha|$. The exponent E can then be factored and a fast order algorithm applied as previously discussed. Appropriately implemented, the resulting algorithm is a Las Vegas algorithm: its output is always correct. The probabilistic choices made in constructing f only impact the expected running time.

Assuming f is a random function, the process of iterating f may (up to the first repetition) be modeled as a sequence of elements $\alpha_1, \alpha_2, \ldots$, sampled independently and uniformly from a set of size $N = |\alpha|$. We then define ρ to be the random variable corresponding to the index of the first repeated element.[2] Sobol shows in [99] that as $N \to \infty$ the random variable $\rho/\sqrt{N}$ has the asymptotic density function: $p(x) = xe^{-x^2/2}$. This implies

$$\mathbf{E}[\rho] \sim \sqrt{\pi N/2}; \qquad \mathbf{var}[\rho] \sim \left(2 - \frac{\pi}{2}\right)\sqrt{N}. \tag{3.1}$$

[1] The letter ρ is meant to be a graphical representation of such a sequence, whence the "rho" method.

[2] This model is analogous to filling a room with people one at a time until there are two with the same birthday. The rho method is often called a "birthday paradox" algorithm for this reason.

The number of iterations before a cycle is detected may be somewhat larger than ρ, depending on the technique used. Brent's method expects to detect a cycle within a constant factor ≈ 1.6 of the optimal value while storing only a single group element [21]; Teske achieves a constant factor of ≈ 1.1 by storing 8 elements [109]. Using linear storage (logarithmic in N), the cycle detection methods described in [93] bring the constant factor arbitrarily close to 1. Henceforth we let X be a random variable representing the number of iterations until a cycle is detected in a random walk, and assume there is a small positive constant ϵ such that

$$\mathbf{E}[X] \sim (1+\epsilon)\sqrt{\pi N/2}; \qquad \mathbf{var}[X] = O\left(\sqrt{N}\right). \tag{3.2}$$

The cycle detection methods cited above all satisfy these conditions, as does the next method we consider.

Distinguished-Point Cycle Detection

Using only slightly more space, a particularly attractive cycle detection technique is the *distinguished-point* method, attributed to Rivest [41]. The group identifier of each iterated element is checked to see whether it has a distinguishing feature of some sort, e.g. contains a certain sequence of trailing bits. More properly, a random hash function should be chosen and the hashed value of the identifier checked, to avoid an unfortunate distribution of identifiers. The distinguished elements are stored in a table, and the second time an attempt is made to insert the same element, a cycle is found. The choice of distinguishing property determines a tradeoff between space and the speed of cycle detection, but a reasonably small table will bring ϵ close to 0.

The main attraction of the distinguished-point method is its suitability for large-scale parallel computation, as described in [113]. Each processor uses the same iteration function, but a different starting point $\alpha_0 = \alpha^{e_0}$, where e_0 is a randomly chosen integer. When a distinguished point is found, the group identifier and the associated exponent are submitted to a central server. As soon as the same distinguished point occurs twice, the difference of the two associated exponents will be a multiple of $|\alpha|$, and almost certainly not zero. It is not necessary to actually find a cycle of f; each processor performs an independent random walk and when a collision is detected, either within a single walk or between two walks,

the order of α can almost always be determined.

One particular issue that arises with an unbounded search is choosing the distinction probability. We want the probability low enough that the table of distinguished points is reasonably small, ideally logarithmic in $|\alpha|$, meaning the distinction probability should by exponentially small. On the other hand, if the probability is too low, we risk never finding any distinguished points. Indeed, if $|\alpha|$ is small, it may be that none of the elements of $\langle\alpha\rangle$ are distinguished, in which case we will never detect a cycle.

This problem may be addressed by gradually decreasing the distinction probability as the search progresses. We define a boolean predicate $s(\alpha, k)$ which takes a group identifier α and an integer k indicating the index of α in the iteration sequence. Let c be an integer constant, say $c \approx 10$, and let $h : \{0,1\}^\ell \to \{0,1\}^\ell$ be a pseudo-random hash function on group identifiers. Let $z(x)$ denote the number of trailing zeros in $x \in \{0,1\}^\ell$. We define:

$$s(\alpha, k) \begin{cases} 1 & \text{if } z\big(h(\alpha)\big) \geq \lfloor \lg k \rfloor - c; \\ 0 & \text{otherwise.} \end{cases} \tag{3.3}$$

Initially every point is distinguished, but $s(\alpha, k)$ becomes gradually more selective as k increases. The key feature of $s(\alpha, k)$ is that no new distinguished points are introduced: if $s(\alpha, k) = 1$, then $s(\alpha, j) = 1$ for all $j \leq k$.

We now argue that if $s(\alpha, k)$ is used to distinguish points then a cycle will be detected shortly after its occurrence, with high probability. Let ρ be the index of the first repeated element in the iteration sequence, so that $\alpha_\rho = \alpha_\lambda$ for some $\lambda < \rho$. For $k < \rho$ in the interval $[2^{n-1}, 2^n)$, either all the points are distinguished, or we expect to find 2^c distinguished points. In the former case we will certainly detect a cycle as soon as it occurs, so we consider the latter. It is shown in [99] that the expectation of the random variable $\rho - \lambda$ is half that of ρ, with similar variance. It follows that with high probability there are many (approximately 2^{c-1}) points α_k with $\lambda < k < \rho$ for which $s(\alpha_k, k) = 1$. Half of these points, on average, will still be distinguished when they are visited again; if $j = k + \rho - \lambda$, then we expect $s(\alpha_j, j) = 1$ for approximately 2^{c-2} of the points on the cycle. These points will all cause a cycle to be detected, thus with high probability we will detect a cycle after $(1 + \epsilon)\rho$ iterations, where ϵ is a small multiple of 2^{-c}. The expected space required for the table is less than $2^c \lg N$, where $N = |\alpha|$.

The argument above can be made more explicit, but it suffices to know that for an appropriate choice of c, we can be almost certain to detect a cycle within $(1+\epsilon)\rho$ iterations. The space required is small, and depends only on c and $N = |\alpha|$.

Limiting Exponent Size

As noted earlier, it is important that the algorithm limit the size of the integers in the exponent sequence, as these will determine the size of E. This issue does not arise in other applications of Pollard's rho algorithm where the order of the group is known and the exponents can be reduced modulo $|G|$ at each step. The original pseudo-random function proposed by Pollard includes a squaring operation, which doubles the size of the exponents each time it is applied. Pollard's function also does not simulate a random walk in a cyclic group as well as might be hoped [111]. Both these issues can be addressed with an *r-adding* walk, as defined by Sattler and Schnorr in [91] (see also [92, 111]).

An r-adding walk is constructed by first choosing r random elements $\beta_1, \ldots, \beta_r \in \langle\alpha\rangle$ with known exponents which will be used by the function f. A randomized black box is of no assistance here: the random elements returned might not lie in $\langle\alpha\rangle$, and even if they did, their associated exponent would be unknown. Instead, we select random integers $e_1, \ldots, e_r \in [1, M]$ where $M \geq N^{\delta_0} \gg |\alpha|$ for some $\delta_0 > 1$, and let $\beta_i = \alpha^{e_i}$. The constraint $M \gg |\alpha|$ ensures the β_i will be nearly uniform over $\langle\alpha\rangle$. Then a random hash function $h : \{0,1\}^\ell \to \{1, \ldots, r\}$ is chosen with the property that any large subset $\mathcal{I} \subseteq \{0,1\}^\ell$ will be partitioned into r groups of roughly equal size, with high probability.[3] The function f is then defined by $f(\alpha) = \alpha\beta_{h(\alpha)}$, where the hash function h is operating on the identifier α as a bit-string in $\mathcal{I}$, not as a group element.

The order algorithm below illustrates the rho method using an r-adding walk and distinguished-point cycle detection. The parameter B is a bound on the number of iterations after which the search will terminate unsuccessfully and return 0.

Algorithm 3.1 (Pollard Rho Method)**.** *Given $\alpha \in G$, positive integers r, M, and B, a hash function $h : \{0,1\}^\ell \to \{1, \ldots, r\}$, a selection predicate $s(\alpha, k)$, and a fast order algorithm $\mathcal{A}(\alpha, E)$, the following algorithm computes $|\alpha|$:*

[3] In practice, simple families of fast hash functions are used that may not be equidistributing. A poor choice of hash function at worst slows down the algorithm.

1. **Initialize**: For i from 1 to r, choose a random integer $e_i \in [1, M]$ and set $\beta_i \leftarrow \alpha^{e_i}$.
   Set $\beta \leftarrow \alpha$, $K \leftarrow 1$, and $k \leftarrow 0$.

2. **Iterate**: If $k \geq B$ return 0, otherwise increment k.
   Set $\beta \leftarrow \beta\beta_i$ and $K \leftarrow K + e_i$, where $i = h(\beta)$. If $s(\beta, k) \neq 1$, repeat step 2.

3. **Cycle?**: Lookup β in the table, and if (β, K') is found, go to step 4.
   Store (β, K) in the table and go to step 2.

4. **Match**: Set $E \leftarrow K - K'$, factor E, and return $\mathcal{A}(\alpha, E)$.

The fast order algorithm $\mathcal{A}(\alpha, E)$ called in the final step requires a *factored exponent* of α, a multiple of $|\alpha|$ whose factorization is known. Any of the fast order algorithms described in Chapter 7 may be used. Considering the parameter r, Sattler and Schnorr show that $r \geq 8$ will generate a pseudo-random walk, provided that $\langle\alpha\rangle$ is large enough [91], and Teske finds $r = 20$ to be sufficient in all cases [109]. Each iteration involves a single group operation, and the exponent K increases by at most M each step, meaning that the expected value of E is $O\left(M\sqrt{N}\right)$. If $M = O\left(N^{\delta}\right)$, with $\delta < 3/2$, then E can be factored using fewer arithmetic operations than the expected $\Theta\left(\sqrt{N}\right)$ group operations.

We would like to choose M so that

$$N^{\delta_0} \leq M \leq N^{\delta_1}, \tag{3.4}$$

where $1 < \delta_0 < \delta_1 < 3/2$, but we don't know $N = |\alpha|$. If we choose M too small, then the β_i used by f will not be uniformly distributed over $\langle\alpha\rangle$, potentially invalidating our assumption that f approximates a random function. If we choose M too large, the resulting exponent may require too much effort to factor. The solution is to start with a small $M = M_0$, fix a constant $\epsilon' > 0$, and once the number of iterations exceeds

$$B = (1 + \epsilon + \epsilon')\sqrt{(\pi/2)M^{1/\delta_0}}, \tag{3.5}$$

replace M by M^{δ_1/δ_0} and restart the search. This can be accomplished via a routine which calls Algorithm 3.1 in a loop, using (3.5) to set the parameter B.

This restart strategy is a minor modification of that used by Teske in [109]. The change allows a tighter bound on the running time to be proven and keeps the exponents smaller.

Proposition 3.1. *Let $N = |\alpha|$, $1 < \delta_0 < \delta_1 < 3/2$, and assume f approximates a random function whenever $M > N^{\delta_0}$. If the restart strategy described above is used with Algorithm 3.1 then:*

(1) The computed exponent $E = O\left(N^{\delta_1 + \frac{1}{2}}\right)$ with high probability.

(2) The expected running time is bounded by $2(1 + \epsilon + \epsilon')\sqrt{\pi N/2} + o\left(\sqrt{N}\right)$.

Proof. (1) We first show that, with high probability, anytime the algorithm restarts we have $M \leq N^{\delta_0}$. If $M > N^{\delta_0}$, then we may assume f has approximated a random function. Let X be the random variable defined in (3.2). If the algorithm decides to restart, (3.5) implies

$$X \;>\; (1 + \epsilon + \epsilon')\sqrt{\pi M^{1/\delta_0}/2} \;>\; (1 + \epsilon + \epsilon')\sqrt{\pi N/2}. \tag{3.6}$$

We may bound the probability of this event by applying Chebyshev's inequality:

$$\begin{aligned}
\Pr\left[X > (1 + \epsilon + \epsilon')\sqrt{\pi N/2}\right] &\leq \Pr\left[\left|X - \mathbf{E}[X]\right| \geq (1 + \epsilon + \epsilon')\sqrt{\pi N/2} - \mathbf{E}[X]\right] \\
&= \Pr\left[\left|X - \mathbf{E}[X]\right| \geq \epsilon'\sqrt{\pi N/2}\right] \\
&\leq \frac{O\left(\sqrt{N}\right)}{\left(\epsilon'\sqrt{\pi N/2}\right)^2} \\
&= O\left(1/\sqrt{N}\right).
\end{aligned} \tag{3.7}$$

The values of $\mathbf{E}[X]$ and $\mathbf{var}[X]$ are taken from (3.2). In particular, at the time of the last restart, $M \leq N^{\delta_0}$, thus the final value of M is at most N^{δ_1} with high probability. A simpler version of the argument above implies that in the last stage X is within a constant factor of its expected value $\Theta\left(N^{1/2}\right)$ with high probability. Then E is the sum of $\Theta\left(N^{1/2}\right)$ terms each bounded by $M \leq N^{\delta_1}$, giving $E = O\left(N^{\delta_1 + 1/2}\right)$ with high probability, proving (1).

(2) The algorithm's execution consists of $k = O\left(\log\log N\right)$ stages, with the value of M being exponentiated by $\delta = \delta_1/\delta_0$ at each stage. If we let $A = (1 + \epsilon + \epsilon')\sqrt{\pi/2}$ and $B_0 = M_0^{1/2\delta_0}$, then the total number of iterations is given by

$$A\left(B_0 + B_0^{\delta} + B_0^{2\delta} + \cdots + B_0^{(k-2)\delta}\right) + F, \tag{3.8}$$

where F is the number of iterations in the final stage. Since the final stage did not reach the restart condition, by (3.6) we have $F \leq (1 + \epsilon + \epsilon')\sqrt{\pi N/2}$, with high probability. The sum of the other terms is dominated by $T = AB_0^{(k-2)\delta}$, which we now consider. In the

penultimate stage, if $M > N^{\delta_0}$ (this happens with low probability) then the expected value of T conditioned on this event is certainly no greater than $\mathbf{E}[X]$, since in this scenario f simulates a random walk. If $M \leq N^{\delta_0}$ then (3.5) implies $T \leq (1 + \epsilon + \epsilon')\sqrt{\pi N/2}$ which serves as a bound on $\mathbf{E}[T]$ in either case. The total number of group operations involved in computing the β_i is negligible: $O\left(\log N \log\log N\right)$. Each iteration requires a single group operation, thus the expected total number of group operations is

$$\mathbf{E}[F] + \mathbf{E}[T] + o\left(\sqrt{N}\right) = 2(1 + \epsilon + \epsilon')\sqrt{\pi N/2} + o\left(\sqrt{N}\right),$$

which completes the proof of (2). □

As desired, the expected running time is bounded by a function of $N = |\alpha|$, independent of any upper bound M. Achieving this independence effectively doubles the expected number of group operations.

Summary

As a generic order algorithm, the rho method has the following advantages:

1. The expected running time is approximately $2.5\sqrt{N}$, using negligible space.
2. It can be efficiently adapted to parallel computation.

These virtues make it the algorithm of choice for very large N. Some disadvantages are:

1. It is a probabilistic algorithm whose expected running time depends on assumptions that are not always met in practice.
2. It requires factoring E.
3. Restricting the search is difficult, even when constraints on N are known.

The last item is the most significant relative to the method we consider next. As examples of where this might apply, suppose that we knew a lower bound on N, or simply that N was an odd number. This information constrains the possible values of N even though it does not constrain its size. It is not obvious how one might take advantage of this knowledge using the rho method.

3.2 Baby-Steps Giant-Steps

Originally developed as an algorithm for determining the class numbers of quadratic number fields, Shanks baby-steps giant-steps method [95] is a generic algorithm well suited to a variety of problems in group computation. It is one of the standard methods for computing discrete logarithms, and is the basis of more general algorithms for determining group structure [27]. Its flexibility lends itself to a wide variety of optimizations in specific circumstances [102]. We focus here on the original problem that motivated the algorithm: order computations.

In its most basic form, the baby-steps giant-steps algorithm is very simple, provided we have an approximate upper bound M, which we can always take to be 2^ℓ.

Algorithm 3.2 (Shanks' Baby-Steps Giant-Steps)**.** *Given $\alpha \in G$ and a positive integer M, let $b = \left\lceil \sqrt{M} \right\rceil$ and compute $|\alpha|$ as follows:*

1. **Baby Steps**: Compute $\alpha^1, \alpha^2, \ldots, \alpha^b$, storing each (α^j, j) in a table.
   If any $\alpha^j = 1_G$, return j.

2. **Giant Steps**: Compute $\alpha^{2b}, \alpha^{3b}, \ldots$, looking up each α^i in the table.
   When $\alpha^i = \alpha^j$, return $i - j$.

Many implementations use α^{-1} in the baby steps and replace $i - j$ by $i + j$. There is no particular advantage to doing this, and keeping the exponents positive simplifies the discussion. The implementation of the lookup table is straight forward and does not require any group operations.[4]

The correctness of the algorithm follows from the fact that $\alpha^i = \alpha^j$ if and only if $i - j$ is a multiple of $|\alpha|$. If $|\alpha| \leq b$ this will be discovered in step 1, otherwise any value of $m > b$ can be written uniquely in the form $m = i - j$ where $i > b$ is a multiple of b and $1 \leq j \leq b$. The first case where $\alpha^i = \alpha^j$ in step 2 will yield the least $m = i - j$ that is a multiple of $|\alpha|$, namely $|\alpha|$.

The enumeration of both baby and giant steps can be accomplished with a single group multiplication per element. The total number of group operations is bounded by $b + |\alpha|/b$, which is at most $2\sqrt{M}$ assuming $|\alpha| \leq M$. As a function of $N = |\alpha|$, however, the running time may be $\Theta(N)$, since when $N \leq \sqrt{M}$ the algorithm performs a sequential search.

[4] This is not true for the black-box groups of Babai and Szemeredi, which effectively require the algorithm to perform an exhaustive search on the lookup table.

Remark 3.1. *If the operation $\mathcal{B}_{inv}$ is faster than $\mathcal{B}_{prod}$, then the elapsed time for Algorithm 3.2 may be improved by letting $b = \lceil \sqrt{M/2} \rceil$ and computing the giant steps $\alpha^{2b}, \alpha^{-2b}, \alpha^{4b}, \alpha^{-4b}, \ldots$. This will decrease the group storage by a factor of $\sqrt{2}$, increase the group operations by a factor of $3\sqrt{2}/4$, and decrease the elapsed time by a factor of up to $\sqrt{2}$. This same optimization may be applied to any order algorithm based on the baby-steps giant-steps approach.*

Unbounded Baby-Steps Giant-Steps

We face a problem similar to that we confronted with the rho method. We would like to pick $M \approx N$, but we don't know N. We adopt a similar solution: start with a provisional M_0, then periodically increase it and "restart". We have the advantage that we need not discard the work we have already done, we simply increase the size of the giant steps by taking some more baby steps. Baby steps are never wasted, and we continue taking (bigger) giant steps wherever we left off. During each iteration, we maintain an optimal 1:1 ratio between baby and giant steps by taking the same number of each. There is some flexibility in choosing the number of steps to take in each iteration. It may be a constant, or it may grow (but not too quickly, as we shall see). We parameterize this with a growth function $d(k)$ which indicates the number of steps to take in iteration $k+1$. The parameter N_0 is an optional lower bound which may be 0, but is useful when it is known that $|\alpha| > N_0$.

Definition 3.1. *A* **growth function** *is an eventually increasing, positive integer function.*

Algorithm 3.3 (Unbounded Baby-Steps Giant-Steps)**.** *Given $\alpha \in G$, an integer $N_0 < |\alpha|$, and a growth function $d(k)$, compute $|\alpha|$ as follows:*

1. **Initialize**: Set $k \leftarrow 0$, $b \leftarrow 0$, and $g \leftarrow \max\{N_0, d(0)\}$.
2. **Baby Steps**: For j from b to $b + d(k)$, compute α^j and store (α^j, j) in a table.
   If any $\alpha^j = 1_G$, return j, otherwise set $b \leftarrow b + d(k)$.
3. **Giant Steps**: For $i = g + b, g + 2b, \ldots, g + d(k)b$, compute α^i and do a table lookup.
   If any $\alpha^i = \alpha^j$, return $i - j$, otherwise, set $g \leftarrow g + d(k)b$, increment k, and go to step 2.

If $|\alpha| \leq d(0)$, this will be discovered in the first set of baby steps. Otherwise, for every integer m between N_0 and g, the algorithm has computed a baby step α^j and a giant step α^i such that $m = i - j$. The values of m are checked in increasing order, hence the least $m = i - j$ such that $\alpha^i = \alpha^j$ will be found and is necessarily $|\alpha|$.

Proposition 3.2. *Let $d(k)$ be a growth function satisfying $\lim_{n\to\infty}\big(d(n+1)-d(n)\big)/d(n) = 0$, that is, $\Delta d \prec d$. Let $|\alpha| = N > N_0 \geq 0$, let $d_0 = d(0)$, and let $T(N)$ be the number of group operations and $S(N)$ the group storage used by Algorithm 3.3.*

(1) If $N - N_0 \leq d_0$ then $T(N) = \min(N, d_0) + 1$ and $S(N) = \min(N, d_0)$.

(2) If $d_0 < N - N_0 < d_0(d_0+1)$ then $T(N) = d_0 + \left\lceil \frac{N-N_0}{d_0} \right\rceil$ and $S(N) = d_0$.

(3) If $N - N_0 \geq d_0(d_0+1)$ then $T(N) \sim 2\sqrt{2(N-N_0)}$ and $S(N) \sim \sqrt{2(N-N_0)}$.

Proof. In case (1) the algorithm terminates during the first set of baby steps if $N \leq d_0$, or on the first giant step. In case (2) the algorithm terminates during the first set of giant steps, and in both cases the proposition is easily verified. We now prove (3) in some detail.[5]

Let $b(n)$ and $g(n)$ respectively denote the values of b and g reached during the nth iteration, where the first iteration has $n = 1$. These functions are defined by

$$b(0) = 0; \qquad b(n+1) = b(n) + d(n);$$
$$g(0) = N_0; \qquad g(n+1) = g(n) + b(n+1)d(n).$$

Assuming $|\alpha| > N_0 + d_0(d_0+1)$, the algorithm terminates in step 3 at some stage t when

$$g(t-1) \;\leq\; |\alpha| \;<\; g(t). \tag{3.9}$$

The number of group operations will be at most twice the number of baby steps $b(t)$ and the group storage will be $b(t)$. We proceed by bounding $b(t)$ in terms of $g(t)$.

Adopting the finite calculus notation of [52], we let $\sum_a^b f(x)\delta x$ denote a sum taken over integers in $[a, b)$ and use the functional operators $\Delta f(n) = f(n+1) - f(n)$ and $Ef(n) = f(n+1)$. In this notation, $\Delta b = d$ and $\Delta g = (Eb)d = (Eb)\Delta b$. For all $n > 0$ we have

$$\begin{aligned} g(n) &= \sum_0^n \Delta g(x)\delta x + g(0) = \sum_0^n Eb(x)\Delta b(x)\delta x + N_0 \\ &= b^2(x)\Big|_0^n - \sum_0^n b(x)d(x)\delta x + N_0 \\ &= b^2(n) - \sum_0^n \big(b(x+1) - d(x)\big)d(x)\delta x + N_0 \\ &= b^2(n) - g(n) + \sum_0^n d^2(x)\delta x + 2N_0, \end{aligned}$$

[5] The proof is straight forward; the detail is intended to illustrate arguments used elsewhere.

where we sum by parts ($\sum Ev\Delta u = uv - \sum u\Delta v$) in the second line. This yields

$$2(g(n) - N_0) = b^2(n) + \sum_0^n d^2(x)\delta x. \tag{3.10}$$

To complete the proof, we show $b^2(n) \sim 2(g(n) - N_0)$ (this allows N_0 to depend on N), and then show $g(n) \sim g(n+1)$, which together with (3.9) implies the desired result. We make use of the fact that for any growth functions f and g,

$$\Delta f \prec \Delta g \quad \Longrightarrow \quad f \prec g. \tag{3.11}$$

By the hypothesis of the theorem, $\Delta d \prec d$, and we are given that d and consequently b and g are growth functions (as are sums or products of such functions). Since $d = \Delta b$, we have $\Delta d \prec \Delta b$, which implies $d \prec b$ and $d^2 \prec bd$. By the product rule ($\Delta(uv) = u\Delta v + Ev\Delta u$),

$$\Delta b^2 = b\Delta b + Eb\Delta b = bd + (b + d)d = 2bd + d^2 \geq bd. \tag{3.12}$$

It follows that $d^2 \prec \Delta b^2$. Since $d^2(n) = \Delta\left(\sum_0^n d^2(x)\delta x\right)$ we may apply (3.11) to obtain

$$\lim_{n\to\infty} \frac{\sum_0^n d^2(x)\delta x}{b^2(n)} = 0, \tag{3.13}$$

which by (3.10) implies $2(g(n) - N_0) \sim b(n)^2$ as desired. To show $g(n) \sim g(n+1)$, we need $\Delta g \prec g$. Since g and Δg are growth functions, by (3.11) it suffices to show $\Delta\Delta g \prec \Delta g$:

$$\begin{aligned}\Delta\Delta g = \Delta[(Eb)d] = \Delta[(b + d)d] &= (b + d)\Delta d + (Ed)(\Delta b + \Delta d) \\ &= b\Delta d + d\Delta d + (d + \Delta d)(d + \Delta d) = b\Delta d + 3d\Delta d + d^2 + (\Delta d)^2.\end{aligned}$$

Every term on the right is $\prec bd \leq (Eb)d = \Delta g$, hence $\Delta\Delta g \prec \Delta g$, as desired. □

If we choose $d_0 \geq \left\lceil \sqrt{N - N_0} \right\rceil$ the algorithm reduces to the standard version of Shanks' algorithm with the upper bound $M = d_0(d_0 + 1)$. In general, however, we will want d_0 to be some small constant, and assume $N_0 = 0$.

Any well-behaved growth function that is subexponential will satisfy the hypothesis of the theorem. In particular, a polynomial function will, but an exponential function will not, as may be seen by the example $d(n) = 2^n$. In this case the sum in equation (3.10) is

not negligible and it is no longer true that $g(n) \sim g(n+1)$. The net result is an $O\left(3\sqrt{N}\right)$ running time using $O\left(2\sqrt{N}\right)$ space (see Remark 3.2 below). In this case the algorithm is essentially equivalent to the method of Buchmann, Jacobson, and Teske [27].[6]

Remark 3.2. *We may abuse notation by using explicit constants within big-O expressions.*[7] *Thus the expression* $O\left(2\sqrt{\pi N/2}\right)$ *may be read as* $2\sqrt{\pi N/2} + o\left(\sqrt{N}\right)$ *or* $\left(2\sqrt{\pi/2} + o(1)\right)\sqrt{N}$. *Any big-O expression which includes an explicit multiplicative constant may be interpreted unambiguously in this fashion.*

In the simplest case, $d(n) = 1$, Algorithm 3.3 reduces to Terr's version of the baby-step giant-step algorithm, also known as the *triangle-number* method [108]. This algorithm represents the fastest generic order algorithm in the literature. By the theorem above, the same $O\left(2\sqrt{2N}\right)$ running time is achieved for any reasonable $d(n)$ whose growth is subexponential, a fact that will be useful in what follows. As a practical matter, $d(n) = 1$ is not the best choice. A linear or quadratic function will result in essentially the same number of group operations but will improve the localization of memory access by reducing the number of transitions between baby steps and giant steps as the table gets large. For large problem sizes, memory usage plays an important role in the performance of Shanks algorithm; processing many baby steps at once is more efficient in most table implementations.

Summary

The Shanks baby-steps giant-steps approach has the following features when applied to order computation:

1. It is a deterministic algorithm with a running time of approximately $2.8\sqrt{N}$.

2. It determines $|\alpha|$ precisely, rather than simply finding a multiple of $|\alpha|$.

3. It is flexible. It can readily benefit from known constraints on $N = |\alpha|$, as well as the availability of a fast inverse operation.

[6] Buchmann et al. give an $O\left(4N^{1/2}\right)$ bound on the running time, but the worst case occurs when the baby-step/giant-step ratio is 2:1, improving the constant to 3.

[7] This is oxymoronic of course, but it improves readability enough to justify the abuse. We will avoid this in the statement of theorems.

The deterministic running time and its flexibility give it an advantage over the rho method when space is not a concern. For large problems however, space quickly becomes the limiting factor, and it is unfortunately not as well suited to parallel implementation as the rho method.

An $\Omega\left(\sqrt{N}\right)$ lower bound?

The expected running time of Algorithm 3.1, based on the rho method, is nearly the same as the running time of Algorithm 3.3 which uses the baby-steps giant-steps approach; roughly $2\sqrt{N\pi/2} \approx 2.5\sqrt{N}$ versus $2\sqrt{2N} \approx 2.8\sqrt{N}$. It is natural to ask whether there is a lower bound close to these values.

In [108], Terr proves a $\sqrt{2N}$ lower bound for all "similar" methods based on an analysis of the number of distinct positive integers which can be represented as the difference of two numbers in a sequence. He considers sequences which are addition chains, but the argument applies more generally.

Lemma 3.3. *Let $a_1, a_2, \ldots$, be a sequence of integers, and for every positive integer N, let $c(N)$ be the least m such that every positive integer $k \leq N$ is the difference $a_i - a_j$ of two numbers in $a_1, \ldots, a_m$. Then for every N, $c(N) > \sqrt{2N}$.*

Proof. Let $m = c(N)$. There are $\binom{m}{2}$ distinct pairs in $a_1, \ldots, a_m$, and each pair represents one positive integer by their difference. Thus $\binom{m}{2} \geq N$, implying $m > \sqrt{2N}$. □

Applying this to a generic algorithm, we may consider the exponent sequence associated with an algorithm's execution, and argue that until the algorithm finds $\alpha^{e_i} = \alpha^{e_j}$ for some pair of exponents $e_i \neq e_j$, it is impossible for the algorithm to correctly output $|\alpha|$. If it is assumed that whenever $\alpha^{e_i} = \alpha^{e_j}$ for some $e_i \neq e_j$, we have $e_i - e_j = |\alpha|$, then the lemma above implies a lower bound of $\sqrt{2N}$ group operations. This assumption is certainly true for all the baby-steps giant-steps algorithms we have considered, and would seem to pose a clear limitation on the baby-steps giant-steps approach.

However, as indicated by Lemma 2.1, the only necessary condition is that $e_i - e_j$ be a *multiple* of $|\alpha|$, it need not equal $|\alpha|$. This apparently minor detail is the enabling factor behind the algorithms we present in the next two chapters.

Chapter 4

The Primorial-Steps Algorithm

Chapter Abstract

We present a new generic order algorithm using the baby-steps giant-steps approach. We begin with a bounded order algorithm with complexity $o\left(M^{1/2}\right)$, then give an unbounded version with comparable worst-case complexity but dramatically better performance in most cases. For $|\alpha| = N$ uniformly distributed over $[1, M]$, the primorial-steps algorithm achieves a median complexity of $O\left(N^{0.344}\right)$. For nearly ten percent of the cases, the performance is $O\left(N^{1/4}\right)$.

None of the order algorithms we have seen break the $\sqrt{2N \log 2}$ barrier suggested by the birthday paradox: a random sampling of large integers expects to accumulate this many values before finding a pair that are congruent modulo $N = |\alpha|$. The best of both the rho method and baby-steps giant-steps algorithms come close, roughly within a factor of 2. In this chapter we see how to beat the birthday paradox. We present an algorithm whose performance is always $o\left(N^{1/2}\right)$, often $O\left(N^{1/3}\right)$, and occasionally even better.

The Basic Idea

Suppose we knew $|\alpha|$ were even. We could easily modify the baby-steps giant-steps algorithm to compute only even powers of α, effectively improving the running time by a factor of $\sqrt{2}$. Alternatively, we could simply run the algorithm as-is on the element α^2, achieving the same improvement. A similar argument could be made if we knew any particular divisor of $|\alpha|$.[1]

[1] Indeed, this is what makes fast order algorithms fast: they know all the divisors of $|\alpha|$.

Unfortunately, there is no way to be sure $|\alpha|$ is even. On the other hand, there is a simple way to find a power of α whose order is odd: square α enough times, and we must eventually get an element with odd order. We may use $\beta = \alpha^{2^\ell}$, since we know $|\alpha| < 2^\ell$ (recall that group identifiers have length ℓ), and computing β uses only ℓ group operations. Once we have β with odd order, we can use a modified version of Algorithm 3.2 in which baby steps land on odd powers of β and giant steps land on even powers of β, so that every odd exponent can be expressed uniquely as the difference of a giant step and a baby step. We can compute $|\beta|$ using at most $O\left(\sqrt{2N}\right)$ group operations,[2] since $|\beta|$ is no larger than $|\alpha|$ (and may be smaller). This doesn't quite give us $|\alpha|$, but we know that $|\alpha| = 2^h N'$ for some $h \leq \ell$, hence 2^ℓ is a multiple of $|\alpha^{N'}|$. We can determine h by squaring $\alpha^{N'}$ until we get 1_G. Note that we do not need to know the prime factors of N'.

The result is an algorithm which works on any α, whether its order is even or not, and a running time bounded by $O\left(\sqrt{2N}\right)$ (assuming $\ell = o\left(\sqrt{N}\right)$, which is almost certainly true, since $\ell = O\left(\lg |G|\right)$). Moreover, if it happens to be the case that $|\alpha|$ *is* even, the running time improves further: $O\left(1\sqrt{N}\right)$ or better depending on the power of 2 in $|\alpha|$.

There is no reason to stop there. We can apply the same idea to obtain $\beta = \alpha^E$ with order coprime to 2 and 3 by letting $E = 2^\ell 3^{\lfloor \ell \log_3 2 \rfloor}$. The baby steps then consist of all powers of β coprime to 6 up to some multiple of 6, say b, while the giant steps land on multiples of b. A few extra group operations are required to cover the gaps between baby steps (computing β^2 and β^4 suffices), and one extra group operation is needed to get the first giant step (the last baby step plus one), but these are negligible. Once $N' = |\beta|$ is known, we can determine $|\alpha|$ by computing $|\alpha^{N'}|$ using a fast order algorithm: we know E is an exponent of $\alpha^{N'}$, and we certainly know the prime factorization of E.[3]

More generally, we should compute β with order coprime to some *primorial*:

$$P_n = 2 * 3 * 5 * \cdots * p_w = \prod_{p \leq p_n} p.$$

The notation $x\# = \prod_{p \leq x} p$ is sometimes used to denote such a product. When $x = p_n$ is the nth prime we adopt the following convention:

Definition 4.1. $\boldsymbol{P_n}$ *denotes the product of the first n primes, where* $P_0 = 1$.

[2] $O\left(\sqrt{2N}\right)$ means $\left(1 + o\left(1\right)\right)\sqrt{2N}$ (see Remark 3.2).

[3] Recall that a fast order algorithm computes $|\alpha|$ given the factorization of a multiple of $|\alpha|$. This can be done very efficiently (see Chapter 7).

4.1 Bounded Primorial Steps

We first give an algorithm for the case where an upper bound M is known. Recall that the Euler function $\phi(n)$ counts the positive integers $m \leq n$ which are coprime to n. For a primorial P_w, the value of $\phi(P_w)$ is simply $\prod_{p \leq p_w}(p-1)$. The notation $m \perp n$ indicates that m and n are coprime (have no common divisor). Assume, for the moment, $M' = M$.

Algorithm 4.1 (Bounded Primorial Steps)**.** *Given $\alpha \in G$, $M \geq |\alpha|$, $M' \leq M$, and a fast order algorithm $\mathcal{A}(\alpha, E)$, compute $|\alpha|$ as follows:*

1. **Compute β**: Maximize P_w s.t. $P_w \leq \sqrt{M'}$, then maximize m s.t. $m^2 P_w \phi(P_w) \leq M'$. Set $b \leftarrow mP_w$, set $E \leftarrow \prod_{p \leq p_w} p^{\lfloor \log_p M \rfloor}$, and compute $\beta \leftarrow \alpha^E$.

2. **Baby Steps**: For j from 1 to b, if $j \perp P_w$, compute β^j and store (β^j, j) in a table. If any $\beta^j = 1_G$, set $N' \leftarrow j$ and go to step 4.

3. **Giant Steps**: For $i = 2b, 3b, \ldots$, compute β^i and do a table lookup. When $\beta^i = \beta^j$, set $N' \leftarrow i - j$ and go to step 4.

4. **Compute $|\alpha|$**: Compute $h \leftarrow \mathcal{A}(\alpha^{N'}, E)$ and return hN'.

The correctness of the algorithm is easily verified. For any p which divides $N = |\alpha|$, the greatest power of p dividing N is at most $\log_p N \leq \log_p M$, hence $|\alpha^E| = N/\gcd(N, E)$ is coprime to P_w. If $|\alpha^E| \leq b$, this will be detected in step 2. Every integer $N' > b$ and coprime to P_w can be written uniquely in the form $N' = i - j$, where $i > b$ is a multiple of $b = mP_w$, and $j < b$ is coprime to P_w. The first case where $\beta^i = \beta^j$ will yield $N' = i - j = |\beta|$. Since $(\alpha^{N'})^E = (\alpha^E)^{N'} = \beta^{N'} = 1_G$, it follows that E is an exponent of $\alpha^{N'}$. Computing $h = |\alpha^{N'}|$ then gives $hN' = |\alpha|$, by the following lemma:

Lemma 4.1. *Let $\alpha \in G$, and let k be an integer. If $n_1 = |\alpha^k|$ and $n_2 = |\alpha^{n_1}|$ then $|\alpha| = n_1 n_2$.*

Proof. The hypothesis $n_2 = |\alpha^{n_1}|$ implies that $n_1 n_2$ is an exponent of α. If $n_1 n_2 \neq |\alpha|$ then $\alpha^{n_1 n_2/p} = 1_G$ for some prime $p | n_1 n_2$. If $p | n_2$ then $\alpha^{n_1 n_2/p} \neq 1_G$, since $n_2 = |\alpha^{n_1}|$ is the minimal exponent of α^{n_1}, so p must divide n_1. Then $\alpha^{k n_1/p}$ has order p, since $n_1 = |\alpha^k|$, but n_2 is not a multiple of p, so $\alpha^{k n_1 n_2/p} \neq 1_G$. This implies $\alpha^{n_1 n_2/p} \neq 1_G$, which contradicts our supposition that $n_1 n_2 \neq |\alpha|$, hence $n_1 n_2 = |\alpha|$. □

Example 4.1.1. *Let* $M = 10,000$ *and suppose* $|\alpha|$ *is the prime* 7883. *Then* $P_w = 30$, $\phi(P_w) = 8$, $m = 6$, $b = 180$, $E = 2^{13} * 3^8 * 5^5$, *and* $|\beta| = 7883$. *The* 48 *baby steps are:*

$$1, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, \ldots, 151, 157, 161, 163, 167, 179, 173, 179.$$

The auxiliary powers β^2, β^4, *and* β^6 *suffice to cover all the gaps, and one additional power* β^{180} *is computed to begin the giant steps:* $360, 540, 720, \ldots, 7740, 7920$. *The algorithm finds* $\beta^{7920} = \beta^{37}$ *and* $N' = 7883$ *after taking* 43 *giant steps. In this case* $\alpha^{N'} = 1_G$, *so* $h = 1$ *and* $hN' = 7883$. *A standard baby-steps giant-steps search for* $|\alpha|$ *would use 100 baby steps and 77 giant steps.*

Suppose instead $|\alpha| = 7566 = 2*3*13*97$. *Then* $|\beta| = 1261$, *and only* 7 *giant steps are required to find* $\beta^{1440} = \beta^{179}$ *and* $N' = 1261$. *Then* $h = \mathcal{A}(\alpha^{N'}, E) = 6$ *and* $hN' = 7566$.

Complexity Analysis

The choice of P_w in step 1 is smaller than optimal. Ideally we want $P_w\phi(P_w) = M$, not $P_w^2 = M$, but pushing P_w too close to the optimal value can make it difficult to maintain a 1:1 ratio of baby steps to giant steps in the worst case. Using a slightly smaller value has negligible impact. The algorithm is optimized for the worst case, where $|\alpha|$ is close to M and coprime to P_w. If $|\alpha|$ has some small factors, as in the second example above, the number of giant steps is much smaller than the number of baby steps, which is not optimal. This illustrates a further limitation of using an upper bound: even if $|\alpha|$ happens to be close to M, $|\beta|$ need not be. This limitation will be addressed by the next algorithm we consider, but first we analyze the complexity of Algorithm 4.1.

The analysis of the primorial-steps algorithm necessarily involves the application of some elementary number theory. The results we need are contained in Chapter 6. Of particular interest is the value $P_w/\phi(P_w)$, the ratio of the size of the giant steps to the number of baby steps. In the standard algorithm this ratio is always 1. In Algorithm 4.1, P_w is chosen so that $P_w \leq \sqrt{M} < P_{w+1}$. Lemma 6.4 (Section 6.1) implies that

$$P_w/\phi(P_w) \sim e^{\gamma} \log\log \sqrt{M} \sim e^{\gamma} \log\log M, \tag{4.1}$$

where $\gamma \approx 0.5772$ is Euler's constant. The expression in (4.1) represents the asymptotic advantage gained by the primorial-steps algorithm. While this approximation is valid as

w	p_w	$P_w = \prod_{p \le p_w} p$	$\phi(P_w) = \prod_{p \le p_w}(p-1)$	$\phi(P_w)/P_w$	$P_w/\phi(P_w)$
1	2	2	1	0.5000	2.0000
2	3	6	2	0.3333	3.0000
3	5	30	8	0.2667	3.7500
4	7	210	48	0.2286	4.3450
5	11	2310	480	0.2078	4.8125
6	13	30030	5760	0.1918	5.2135
7	17	510510	92160	0.1805	5.5394
8	19	9699690	1658880	0.1710	5.8471
9	23	223092870	36495360	0.1636	6.1129
10	29	6469693230	1021870080	0.1579	6.3312

Table 4.1: The First Ten Primorials

M tends toward infinity, for practical values of M it is more appropriate to compute the ratio directly. Ratios for the first ten primorials are listed in Table 4.1. The primorials form sequence A002110 in the OEIS [98], and the values $\phi(P_w)$ form sequence A005867.

Proposition 4.2. *If $|\alpha| < M$ and $M' = M$, then Algorithm 4.1 uses at most*

$$\left(A_1 + o\left(1\right)\right) \sqrt{M/\log\log M} \tag{4.2}$$

group operations, where $A_1 = 2e^{-\gamma/2} \approx 1.4986$. The group storage is at most half of (4.2).

Proof. Let m, $P = P_w$, and $b = mP$ be the values computed in step 1 of Algorithm 4.1. Then

$$m^2 P\phi(P) \le M < (m+1)^2 P\phi(P). \tag{4.3}$$

The number of baby steps is $b' = m\phi(P)$, which may be bounded by

$$b' = m\phi(P) \le \sqrt{M\phi(P)/P}. \tag{4.4}$$

If g is the number of giant steps, step 3 will terminate with $gb \le |\beta| < (g+1)b$. Since $|\beta| \le |\alpha| < M$, we may apply the right inequality in (4.3) to obtain

$$g \le \frac{|\beta|}{b} \le \frac{(m+1)^2 P\phi(P)}{mP} = \left(1 + \frac{2}{m} + \frac{1}{m^2}\right) m\phi(P) \le \left(1 + O\left(\frac{1}{m}\right)\right) \sqrt{M\phi(P)/P}, \tag{4.5}$$

where the last inequality follows from (4.4). By (4.3) and (4.1) we also have

$$(m+1)^2 > \frac{M}{P\phi(P)} \geq \frac{P^2}{P\phi(P)} = P/\phi(P) \sim e^{\gamma} \log\log M, \tag{4.6}$$

hence m is unbounded and $1/m = o(1)$. Adding (4.4) and (4.5) and applying (4.1) yields

$$b' + g \leq (2 + o(1)) \sqrt{M\phi(P)/P} = (2e^{-\gamma/2} + o(1)) \sqrt{M/\log\log M}. \tag{4.7}$$

This essentially completes the proof, as the group operations are dominated by this bound. By Lemma 4.3 below, the gaps between baby steps up to P, and therefore all the gaps, can be covered using $O\left(\sqrt{P}\right) = O\left(M^{1/4}\right)$ additional group operations. Step 1 uses $O\left(\lg E\right) = O\left(\lg^2 M\right)$ group operations, since $\lg E = w \lg M \leq \lg P \lg M \leq \lg^2 M$. Any of the fast order algorithms presented in Chapter 7 can complete step 4 in polylogarithmic time. □

The gaps between baby steps are generally quite small, being bounded by the largest prime gap below $\sqrt{M}$, and a negligible number of group operations suffices to cover them. Rather than give a tight bound, we prove a lemma that is more widely applicable. The proof may be regarded as an application of the baby-steps giant-steps method.

Lemma 4.3. *Let S be a sequence of integers $1 = s_0 < s_1 < \cdots < s_n = N$. There is an addition chain containing S of length at most*

$$|S| + \sqrt{2N} + O\left(N^{1/3} \lg N\right),$$

which can be computed using $O(\sqrt{N} \lg N)$ arithmetic operations.

Proof. Let $T = \{t | t = s_i - s_{i-1}, 1 \leq i \leq n\}$ be the set of gaps of S. The sum of the elements of T is less than N. It follows that

$$|T| < \sqrt{2N},$$

since any sum of k distinct positive integers is at least $\sum_{i=1}^{k} i > k^2/2$. Now let $m = \lceil N^{1/3} \rceil$, and let U to be the addition chain $\{1, 2, 3, \ldots, m, 2m, 3m, \ldots, (m-1)m\}$.[4] Then

$$|U| = O\left(N^{1/3}\right).$$

[4]We say a set is an addition chain if the ordered sequence of its elements is.

Every positive integer up to $N^{2/3} \le m^2$ is either an element of U or the sum of a pair in U. Fewer than $N^{1/3}$ of the elements in T can be greater than $N^{2/3}$, since the sum of the elements in T is less than N. For each of these an addition chain of length $O\left(\lg N\right)$ can be constructed using $O\left(\lg N\right)$ arithmetic operations via the binary exponentiation algorithm. Let V be the union of these chains. Then $T \cup U \cup V$ is an addition chain and

$$|V| = O\left(N^{1/3} \lg N\right),$$

hence $S \cup T \cup U \cup V$ satisfies the lemma. □

The Average Case

We wish to compare the average case performance of the bounded primorial-steps algorithm and the standard baby-steps giant-steps method. We assume that $|\alpha|$ is uniformly distributed over some interval $[1, M]$. Note that the optimal bound to use, knowing this distribution,[5] is not M, but some $M' < M$ that minimizes the average number of group operations over values of $|\alpha|$ in $[1, M]$. For the standard baby-steps giant-steps algorithm the optimal bound is $M' = M/2$ and the average number of group operations is $2\sqrt{M'} = \sqrt{2M}$ (compared to $2\sqrt{M}$). The optimal M' to use in Algorithm 4.1 is given below.

Proposition 4.4. *If $|\alpha|$ is uniformly distributed over $[1, M]$, the average number of group operations used by Algorithm 4.1 is at most*

$$\frac{\left(A_2 + o(1)\right)\sqrt{M}}{\log\log M}, \tag{4.8}$$

provided the input bound $M' = (\zeta(2)/2)M \approx 0.8225M$ is used, where $A_2 = e^{-\gamma}\sqrt{2\zeta(2)} \approx 1.0184$. The average group storage is half of (4.8).

Proof. Assume for the moment that the bound $M' = M$ is used, and let $r = P_w/\phi(P_w)$, where P_w is the value chosen in step 1 of Algorithm 4.1. By (4.1), $r \sim e^{\gamma}\log\log M$. Let $\kappa_y(x)$ denote the y-coarse part of x, the largest divisor of x whose prime factors all exceed y. If we set $y = p_w$ then $|\beta| = \kappa_y(|\alpha|)$. The average number of group operations may be bounded by

$$\frac{1}{M}\sum_{x=1}^{M}\left(\frac{b}{r} + \frac{\kappa_y(x)}{b}\right) \;=\; \frac{b}{r} + \left(\frac{1}{bM}\sum_{x=1}^{M}\kappa_y(x)\right) \;=\; \frac{b}{r} + \frac{\overline{\kappa_y}}{b}, \tag{4.9}$$

[5] In the bounded case it is reasonable to assume the distribution is known. In the unbounded case it is not.

where b is the step size and $\overline{\kappa_y}$ is the mean value of $\kappa_y(x)$ over $x \in [1, M]$. The term b/r is the number of baby steps, and the term $\kappa_y(x)/b$ is the number of giant steps required when $|\alpha| = x$. An asymptotic estimate for $\overline{\kappa_y}$ is derived in Section 6.5. To apply the estimate in its most convenient form we need $y = \Theta\left(\log M\right)$. By the PNT (Theorem 6.2) we have

$$\log P_w = \sum_{p \le p_w} \log p = \vartheta(p_w) = \vartheta(y) \ \sim \ y, \tag{4.10}$$

and since $\log P_w = \Theta\left(\log M\right)$, we can now apply Corollary 6.17 to obtain

$$\overline{\kappa_y} \ \sim \ \frac{\zeta(2)M}{2e^{\gamma} \log\log M} \ \sim \ \frac{\zeta(2)M}{2r}. \tag{4.11}$$

It follows that (4.9) is asymptotic to

$$\frac{b}{r} + \frac{\zeta(2)M}{2br}, \tag{4.12}$$

which is minimized when $b = \sqrt{M\zeta(2)/2}$. If we use $M' = M\zeta(2)/2$ as our initial bound, this will be true to within a factor of $1 + o(1)$, and the value of r will be effectively unchanged, since $r = \Theta\left(\log\log M\right)$. The average number of group operations will then be bounded by $\left(2+o\left(1\right)\right)b/r$, of which half are baby steps, determining the group storage used. Substituting $r \sim (e^{\gamma} \log\log M)$ and $b = \sqrt{M\zeta(2)/2}$ completes the proof. □

Table 4.2 compares the complexity of Algorithm 4.1 to the standard baby-steps giant-steps method, Algorithm 3.2. The ratios compare the dominant terms in the complexity of the two algorithms and apply to both time and space. The worst case occurs when $|\alpha|$ contains no small prime factors, e.g. when $|\alpha|$ is prime.

The average case performance of Algorithm 4.1 may be improved by using a larger value of E in step 1. This increases the value of y in the proof above, effectively decreasing κ_y^*. An average complexity of $O\left(M^{1/2}(\log M \log\log M)^{-1/2}\right)$ can be obtained with this approach, however, the algorithm we next consider surpasses this bound.

The bounded primorial-steps algorithm is severely limited by not taking full advantage of the situation when $|\beta| < |\alpha|$. For larger values of E, it will often be the case that $|\beta| \ll |\alpha|$. Thus even if we know tight bounds on $|\alpha|$, it is still essential to have an unbounded version of the algorithm.

	Worst Case		Average	
M	T_1/T_2	T_2/T_1	T_1/T_2	T_2/T_1
10^6	0.4624	2.1626	0.2742	3.6464
10^9	0.4304	2.3236	0.2376	4.2095
10^{12}	0.4113	2.4313	0.2170	4.6090
10^{15}	0.3981	2.5117	0.2033	4.9189
10^{18}	0.3883	2.5755	0.1933	5.1721

Table 4.2: Bounded Primorial Steps (T_1) vs. Bounded Baby-Steps Giant-Steps (T_2)

4.2 Unbounded Primorial Steps

Algorithm 4.1 depends on M in two ways: M determines the choice of P_w, and M bounds the prime powers in the exponent E. The latter dependence is effectively logarithmic in M. For the moment we will accept this and use the identifier length ℓ to bound the power of any prime factor of E. We will address the dependence on ℓ in the next chapter.

To address the major dependence on M, we can adapt Algorithm 3.3, where we use a growth function $d(k)$ to determine the number of steps to take in each iteration. Here we definitely want $d(k)$ to grow reasonably quickly, as the largest P_w we can use at any stage will be limited by $d(k)$. On the other hand, we must take care not to let $d(k)$ grow too quickly since, as we saw in Proposition 3.2, the optimal performance is achieved when $d(k)$ is sub-exponential. Using a quadratic growth function works well in practice.

One minor annoyance arises when switching from a multiple of P_w to a multiple of P_{w+1}. We need to be sure to end our baby steps just below a multiple of P_{w+1} and to start our giant steps on a multiple of P_{w+1}, which my require a short "stutter step" to handle the transition. We also add the parameter L which bounds the prime factors of the exponent E used to compute $\beta = \alpha^E$. This should be large enough to avoid unnecessarily restricting P_w ($\ell/2$ suffices), but may be much larger.

Algorithm 4.2 (Unbounded Primorial Steps)**.** *Given $\alpha \in G$, $\ell \geq \lg|\alpha|$, $L > 0$, a growth function $d(k)$, and a fast order algorithm $\mathcal{A}(\alpha, E)$, compute $|\alpha|$ as follows:*

1. **Compute β**: Let $E = \prod_{p \leq L} p^{\lfloor \log_p 2^\ell \rfloor}$ and compute $\beta \leftarrow \alpha^E$.

2. **Initialize**: Maximize w s.t. $P_w \leq d(0)$ and $p_w \leq L$. Set $k \leftarrow 0$, $b \leftarrow 0$, and $g \leftarrow 0$.

3. **Adjust P_w**: If $P_{w+1} \leq d(k)$ and $p_{w+1} \leq L$ then

   a. Increment w, minimize $b_1 = mP_w \geq b$, and minimize $g_1 = mP_w \geq g$.

   b. For j from b to b_1, if $j \perp P_w$ compute β^j and store (β^j, j) in the table.
   If any $\beta^j = 1_G$, set $N' \leftarrow j$ and go to step 7, otherwise set $b \leftarrow b_1$.

   c. Set $i \leftarrow g_1$, compute β^i and do a table lookup.
   If $\beta^i = \beta^j$, set $N' \leftarrow i - j$ and go to step 7, otherwise set $g \leftarrow g_1$.

   Maximize m s.t. $mP_w \leq d(k)$, then set $s \leftarrow mP_w$ and $s' \leftarrow m\phi(P_w)$.

4. **Baby Steps**: For j from b to $b + s$, if $j \perp P_w$, compute β^j and store (β^j, j) in a table.
   If any $\beta^j = 1_G$, set $N' \leftarrow j$ and go to step 7, otherwise set $b \leftarrow b + s$.

5. **Giant Steps**: For $i = g + b, g + 2b, \ldots, g + s'b$, compute β^i and do a table lookup.
   If any $\beta^i = \beta^j$, set $N' \leftarrow i - j$ and go to step 7, otherwise set $g \leftarrow g + s'b$.

6. **Iterate**: Increment k and go to step 3.

7. **Compute $|\alpha|$**: Compute $h \leftarrow \mathcal{A}(\alpha^{N'}, E)$ and return hN'.

To gain an understanding of the algorithm, it is helpful to suppose that in step 3a it always happens that b and g are divisible by P_w, allowing steps 3b and 3c to be ignored. With this in mind, the algorithm may be viewed as a simple integration of Algorithm 3.3 and Algorithm 4.1. As shown in the proof below, the number of group operations involved in steps 3b and 3c is negligible.

Proposition 4.5. *Let $N = |\alpha|$, let $\ell \geq \lg N$, and let $L \geq \ell/2$. Let $d(k)$ be a growth function satisfying $\Delta d \prec d$ that is polynomially bounded above and below. If algorithm $\mathcal{A}(\gamma, E)$ has complexity $T(\lg|\gamma|, \lg E)$ at least linear in $\lg E$, then Algorithm 4.2 uses at most*

$$\left(A_3 + o\left(1\right)\right) \sqrt{N/\log\log N} + T(\lg N, \ell\pi(L)) \tag{4.13}$$

group operations and $\left(A_3/2 + o\left(1\right)\right) \sqrt{N/\log\log N}$ group storage. $A_3 = 2e^{-\gamma/2}\sqrt{2} \approx 2.1194$.

Proof. We begin by assuming that stutter steps do not occur, i.e. that step 3a always sets $b_1 = b$ and $g_1 = g$. We then show the asymptotic bounds are unaffected by stutter steps.

Let $w(n)$, $s(n)$, and $s'(n)$ be the values of w, s, and s' derived from $d(n)$, and define

$$r(n) = s(n)/s'(n) = P_{w(n)}/\phi(P_{w(n)}). \tag{4.14}$$

For all n we have $P_{w(n)} \le d(n) < P_{w(n)+1}$, hence $r(n) \sim e^\gamma \log\log d(n)$ by Lemma 6.4. It follows that $r(n) \sim e^\gamma \log\log n$, since $d(n)$ is bounded by polynomial functions of n.

Following Proposition 3.2, let $b(n)$ and $g(n)$ be the values of b and g reached in the nth iteration, so that in the final iteration t we have $g(t-1) \le N' < g(t)$. Using the same notation, we have $b = \sum s$ and $g = \sum (Eb)s'$, with $b(0) = g(0) = 0$. If we define $\tilde{g} = \sum (Eb)s$, then as in Proposition 3.2, we find $b^2 \sim 2\tilde{g}$. We also have

$$\begin{aligned}\tilde{g}(n) &= \sum_{i=0}^{n-1} b(i+1)s(i) = \sum_{i=0}^{n-1} b(i+1)r(i)s'(i) = \sum_{i=0}^{n-1} r(i)\Delta g(i) \\ &= r(n)g(n) - \sum_{i=0}^{n-1} g(i+1)\Delta r(i),\end{aligned} \tag{4.15}$$

where we sum by parts in the last line. The RHS of (4.15) is dominated by the first term, since Δr is almost always zero and $g(n)$ is polynomially bounded.[6] Thus $b^2 \sim 2\tilde{g} \sim 2rg$, and if we define $b' = \sum s'$, a similar argument shows $b \sim rb'$. Therefore $rb' \sim \sqrt{2rg}$ and $b' \sim \sqrt{2g/r}$. But $b'(t)$ is precisely the number of baby steps, $g(t-1) \le N' < g(t)$, and, as in Proposition 3.2, we have $g(n) \sim g(n+1)$. Thus we have

$$b' \sim \sqrt{2N'/r(t)} \sim \left(e^{-\gamma/2}\sqrt{2}\right)\sqrt{N'/\log\log t} \sim \left(e^{-\gamma/2}\sqrt{2}\right)\sqrt{N'/\log\log N'}, \tag{4.16}$$

where the replacement of $\log\log t$ by $\log\log N'$ is justified by the fact that $N' < g(t)$ is polynomially bounded in t. Since $N' \le N$, the number of baby steps is bounded by

$$\left(e^{-\gamma/2}\sqrt{2} + o(1)\right)\sqrt{N/\log\log N}, \tag{4.17}$$

and the number of giant steps is no greater. As in Proposition 4.2, the group operations needed to cover gaps between baby steps are negligible.

We now account for the possibility of stutter steps. Recall that $P_{w(n)} \le d(n)$, thus $w(n)$ is logarithmically bounded, since $2^m \le P_m$, and bounds the number of stutter steps.

[6] This can be rigorously proven using Stieltjes integrals and asymptotic integration (see Section 6.5).

Each stutter step uses no more than $d(t)$ group operations, and the total $d(t)w(t)$ is strictly dominated by $b'(t)$ since the function $\Delta b' = s' = s/r \geq d/(2r)$ is within a polylogarithmic factor of the function dw.

Finally, we account for the group operations used in step 1 and step 7. The size of the exponent E is given by $\lg E = \ell\pi(L)$, thus the exponentiation in step 1 uses $O(\ell\pi(L))$ group operations. The complexity bound on $\mathcal{A}(\alpha^{N'}, E)$ then gives the desired result, since $|\alpha^{N'}| \leq |\alpha| = N$. The group storage is determined by the number of baby steps, which is at most half the time bound. □

Unless L is very large, the first term in (4.13) dominates and the worst-case complexity of Algorithm 4.2 is $O\left(\sqrt{N/\log\log N}\right)$, within a constant factor of the bounded version. As with the unbounded baby-steps giant-steps algorithm, we increased the running time by a factor of $\sqrt{2}$ in order to remove the dependence on M. Thus a comparison of the worst-case complexity of Algorithm 4.2 with Algorithm 3.3 will show the same ratios as listed in Table 4.2. The "average case", on the other hand, is an entirely different story.

Average Complexity

The complexity of the primorial-steps algorithm is essentially determined by the order of $\beta = \alpha^E$. If we suppose $|\alpha|$ is uniformly distributed over $[1, M]$, then $|\beta|$ will be substantially smaller than $|\alpha|$ in many cases. The frequency with which this occurs (and the benefit when it does) depends on the parameter L. In the extreme case, if we set L to $\sqrt{M}$, then $|\beta|$ is necessarily either 1 or the single prime factor of $|\alpha| > \sqrt{M}$. On average, this will be smaller than $|\alpha|$ by a factor of $\log M$, but in nearly 1/3 of the cases $\beta = 1_G$ and no work is required to determine $|\beta|$.

Unfortunately this value of L makes $E = \Theta\big(\ell\pi(L)\big)$ too large: it takes $\Omega\left(\sqrt{M}\right)$ group operations to compute $\beta = \alpha^E$. However, a slightly more modest value of L, say $L = M^{1/3}$, achieves nearly the same result without increasing the average running time. This gives a running time that is "often" $O\left(N^{1/3}\right)$. More formally, an appropriate choice of L gives an algorithm which uses only $O(M^c)$ group operations for a constant fraction $r(c)$ of the possible values of $|\alpha|$ in $[1, M]$. In particular, for $c = 1/3$, we will find that $r(c) \approx 45\%$.

Before analyzing this phenomenon in more detail, we first consider the average performance over uniformly distributed $|\alpha|$ in $[1, M]$. In what follows we assume that $d(k)$ is

always chosen to satisfy the conditions of Proposition 4.5. We also define

$$f_*(x) = A_3 \sqrt{x/\log\log x}, \tag{4.18}$$

for $x \geq 3$. The function $f_*(x)$ represents an asymptotic bound on the number of group operations used by the main loop of Algorithm 4.2.

Proposition 4.6. *If $|\alpha|$ is uniformly distributed over $[1, M]$, $\ell = O\left(\lg M\right)$, and $L = M^{1/u}$ with $u > 2$, then the average number of group operations used by Algorithm 4.2 is bounded by*

$$\frac{\left(uA_4 + o\left(1\right)\right)\sqrt{M}}{\log M(\log\log M)^{1/2}}, \tag{4.19}$$

where $A_4 \approx 2.4$.[7] The average group storage is bounded by half of (4.19).

Proof. By Proposition 4.5, the function $f_*(x)$ defined by (4.18) bounds the running time of the main loop of Algorithm 4.2 in terms of $x = |\beta|$, which is the L-coarse part of $|\alpha|$. As in Section 6.1, we use $\kappa_y(x)$ to denote the y-coarse part of x, the largest divisor of x whose prime factors all exceed y. We wish to bound the mean value of $f_*(\kappa_y(x))$ over the interval $[0, M]$ with $y = L$. Applying Proposition 6.22, we find

$$\frac{1}{M}\sum_{n=1}^{M} f_*(\kappa_y(n)) \leq \left(\frac{u\omega(u)}{s} + o\left(1\right)\right)\frac{f_*(M)}{\log M}, \tag{4.20}$$

where $\omega(u) \leq e^{-\gamma} + 0.005684 \approx 0.5672$, and $s = 1/2$ is the exponent of x in $f_*(x)$. Setting $A_4 = 0.5672 * A_3/s \approx 2.4$, where A_3 is the constant from Proposition 4.5, gives

$$\frac{\left(uA_4 + o\left(1\right)\right)\sqrt{M}}{\log M(\log\log M)^{1/2}}, \tag{4.21}$$

which is the desired bound. Let $\ell = c\lg M$. The size of the exponent E is bounded by

$$\lg E \leq c\pi(L)\lg M = c\pi(M^{1/u})\lg M \sim \frac{cuM^{1/u}}{\log 2}, \tag{4.22}$$

by the prime number theorem. Using any fast exponentiation algorithm, step 1 will use $O\left(M^{1/u}\right)$ group operations. Using Algorithm 7.1 to perform the fast order computation in

[7] As noted in Section 6.5, this constant is not tight. Using the speculated bound (6.40), $A_4 \approx 2.1$.

step 7 will also use $O\left(M^{1/u}\right)$ group operations. These bounds are dominated by (4.21) for $u > 2$. In no case is the group storage more than half the number of group operations. □

The bound above is not tight, but it gives a fairly accurate measure of the average performance. It also hides a great deal. The average is dominated by the worst-case running time of a small fraction of the values in $[1, M]$: large primes and other numbers composed mostly of large primes. The order of β after all the primes up to L have been removed from $|\alpha|$ is reasonably likely to be less than $M^{2/3}$ (about 45% of the time). For these cases the running time of Algorithm 4.2 is $O\left(M^{1/3}\right)$.

More generally, for a given choice of $L = M^{1/u}$, provided that $|\beta|$ is $O\left(M^{2/u} \log\log M\right)$, Algorithm 4.2 will use $O\left(M^{1/u}\right)$ time and space. This will certainly occur whenever the L-coarse part of $|\alpha|$ is smaller than L^2. In this scenario, $|\alpha|$ can contain at most one prime factor bigger than L, and all its prime factors are less than L^2. Such a number is said to be *semismooth* with respect to L^2 and L. Semismooth numbers are discussed further in Section 6.4, but for the moment it suffices to know that there is a function $G(1/u, 2/u)$ which computes the probability that a random integer in $[1, M]$ is semismooth with respect to $M^{2/u}$ and $M^{1/u}$.

Proposition 4.7. *Let $T(N)$ be the number of group operations used by Algorithm 4.2 on input α with $|\alpha| = N$. If $|\alpha|$ is uniformly distributed over $[1, M]$, $\ell = O\left(\lg M\right)$, and $L = M^{1/u}$, there is a constant c such that*

$$\Pr\left[T\left(|\alpha|\right) \leq cM^{1/u}\right] \geq G(1/u, 2/u), \tag{4.23}$$

for all sufficiently large M, where $G(r, s)$ is the semismooth probability function (Definition 6.8).

Proof. Let $L = M^{1/u}$. Choose c' so that $f_*(x) = A_3\sqrt{x/\log\log x} \leq c'x^{2/u}$ for all $x \in [L, L^2]$ and so that steps 1 and 7 of Algorithm 4.2 require less than $c'M^{1/u}$ group operations (both steps have linear complexity for $\ell = O\left(\lg M\right)$ and $L = M^{1/u}$ as in Proposition 4.6). If the L-coarse part of $x = |\alpha|$ is less than L^2, then the complexity of Algorithm 4.2 will be less than $cM^{1/u}$, where $c = 2c'$. This occurs precisely when x is semismooth with respect to $M^{2/u}$ and $M^{1/u}$, and the proposition follows from the definition of $G(r, s)$. □

Asymptotically, the constant c in the proposition can be made arbitrarily small by choosing L slightly less than $M^{1/u}$, say $L = M^{1/u}(\log\log m)^{-1/3}$. Of more practical relevance are the values of $\sigma(u) = G(1/u, 2/u)$ which are listed in Table 6.1 of Chapter 6, including

$\sigma(2.2) \approx 90\%$, $\sigma(2.9) > 50\%$, and $\sigma(4) \approx 10\%$. To better represent the "typical" performance of Algorithm 4.2, we define the median complexity of a generic algorithm:

Definition 4.2. *For a specified set of input values, the* **median complexity** *of a generic algorithm is the least integer N for which the algorithm uses less than N group operations for at least half the input values in the set.*

Noting that $G(1/u, 2/u) > 1/2$ for $u = 2.9$ yields the following corollary:

Corollary 4.8. *For $|\alpha|$ uniformly distributed over $[1, M]$, $\ell = O\left(\lg M\right)$, and $L = M^{1/3}$, the median complexity of Algorithm 4.2 is $O\left(M^{0.344}\right)$.*

Before presenting a more general version of the primorial-steps algorithm which addresses the dependence on ℓ and the optimal choice of L, let us take a moment to reflect on what we have found. Consider a hypothetical scenario in which we are asked to compute $|\alpha|$ for some particular $\alpha \in G$ with unknown order N uniformly distributed over $[1, 10^{24}]$. This scenario is clearly "hypothetical", as any algorithm based on a baby-steps giant-steps approach is likely to require more space than is reasonably available. Or so it would seem.

Table 4.3 lists running times for three unbounded order algorithms in this scenario. For the rho method, the optimized version of Teske's algorithm is used (Algorithm 3.1), with an expected running time of $\approx 2.5\sqrt{N}$. For the sake of simplicity, we assume the expected running time always occurs and only consider the distribution of input sizes, not the coin flips of the randomized algorithm. For the baby-steps giant-steps algorithm we use Terr's triangle method (Algorithm 3.3), with a running time of $\approx 2.8\sqrt{N}$. These two options represent the best choices among existing generic algorithms.

For the primorial-steps algorithm, we use Algorithm 4.2 with $\ell = \lg M$ and $L = M^{1/u}$ where $u = 2.9$. We will see how to remove the explicit dependence on M in the next chapter, but for now we pick values to optimize the median complexity, at the expense of the average performance, which would benefit from a larger value of L. We then apply Propositions 4.5, 4.6, and 4.7 to compute the worst, average, and median complexity for this choice of L. In the median case we can choose $c' \approx 1$ giving $c \approx 2$ in Proposition 4.7. The group storage is half the group operations in the first two cases, and between 1/3 and 1/4 in the median case.

We suppose that the black box performs 10^5 group operations per second, comparable to several of the black boxes described in Chapter 11, and assume any auxiliary computations

Algorithm	Worst	Average	Median
Rho method	289 days	193 days	204 days
Baby-steps giant-steps	327 days	218 days	231 days
Primorial steps	117 days	8 days	**63 min**
Rho method	-	-	-
Baby-steps giant-steps	23 Tb	15 Tb	16 Tb
Primorial steps	8 Tb	0.5 Tb	**1 Gb**

Table 4.3: Hypothetical Comparison of Order Algorithms with $|\alpha| \in [1, 10^{24}]$

done by the algorithms are negligible; this will be true in an efficient implementation of any of these three algorithms. We assume 16 bytes suffice to store a group identifier along with any table overhead for the baby-steps giant-steps and primorial-steps algorithms. This is about 1/3 larger than the absolute minimum required, giving the black box some flexibility and keeping the algorithms honest. We have not assumed fast inverses are available (see Remark 3.1). This optimization could reduce the time and space listed for the baby-steps giant-steps algorithm and the primorial-steps algorithm by up to 30%.

As indicated by the median case in Table 4.3, at least half the time the primorial-steps algorithm computes $|\alpha|$ in about an hour using less than 1Gb of memory, easily achieved on a typical personal computer (2007).

Skepticism in the face of such hypothetical scenarios is completely justified, as they gloss over any number of issues that arise in practice. The reader is invited to peruse the performance results in Chapter 11. Comparisons of computations using the primorial-steps algorithm against published results for other generic algorithms show differences every bit as dramatic as those in the table above.[8] Indeed, the performance of the primorial-steps algorithm on ideal class groups is substantially better than indicated by Table 4.3. The observed median complexity for class groups of comparable size was better than $O\left(N^{0.3}\right)$ (see Chapter 11, Table 11.6), which would give a running time of a few minutes in the scenario above.

While the median-case performance in Table 4.3 looks quite attractive, it is clear that primorial-steps algorithm will run into space limitations on large problems, even in situations where the running time is tractable. This is a problem with baby-step giant-step

[8] Albeit on a necessarily smaller scale, due to the limited availability of results for large groups.

methods in general: they run out of space before they run out of time. We will see how to address this issue in the next chapter, achieving an algorithm with essentially the same median performance, without space limitations.

Chapter 5

The Multi-Stage Sieve

Chapter Abstract

We present a fully generalized order algorithm that nearly matches the performance of the primorial-steps algorithm without requiring a prime bound to be specified. The search phase of each stage is performed by a subroutine which may use either a primorial-steps search or a rho search. In the latter case, the same $O\left(N^{0.344}\right)$ median case performance is achieved using minimal space. We consider similarities with certain algorithms for integer factorization, and show that any problem which can be reduced to solving one of a sequence of random order computations can be solved by a generic algorithm with subexponential complexity.

We wish to generalize the unbounded version of the primorial-steps algorithm to remove the dependence on the prime bound L. For a particular $N = |\alpha|$, the best choice of L depends on the multiplicative structure of the integer N. In the worst case, N is prime, and there is no reason to choose $L > \frac{1}{2} \log N$. This will ensure that an appropriately large primorial $P = \prod_{p \le L} p$ is used by Algorithm 4.2, and a smaller value may suffice (depending on the function $d(k)$). In practice one rarely needs $L > 30$ if N is prime.

When N is composite, it is almost always worth making L big enough to ensure that every prime in N but the largest is removed from $N' = |\beta|$. To simplify the discussion, we will suppose that N is square-free; taking prime-power factors into account does not change the basic argument. To remove all but one prime factor from N, L should be as big as the second largest prime factor of N. The cost of making L this large is felt in steps 1 and 7 of Algorithm 4.2, which use a total of about

$$2\pi(L) \lg L \;\sim\; 2L \lg L / \log L = (2 \log^{-1} 2)L \approx 3L$$

group operations, assuming we use Algorithm 7.1 for the fast order computation in step 7. The benefit of doing so is roughly

$$N^{1/2} - (N/L)^{1/2} \geq N^{1/2} - N^{1/4} \approx N^{1/2},$$

assuming we ignore the factor $A_3(\log\log N)^{-1/2}$, which is reasonably close to 1 and never less than 1/2 for realistic values of N. Thus for any L up to about $\sqrt{N}/3$, we should be prepared to exponentiate α by all the primes up to L in order to reduce N' to a prime number. Note that the argument applies to any particular $N = |\alpha|$; it does not depend on the distribution of $|\alpha|$.

This is a somewhat startling conclusion, as it implies using millions, possibly billions, of primes for large values of N. On the other hand, we would prefer not use any more than the absolute minimum required. Unfortunately, we have no way to know when we have reached that point, other than searching up to approximately L^2 to see if we find $N' = |\beta|$, e.g. by running the main loop of Algorithm 4.2 up to this bound. If the search fails, we know that either the largest prime divisor of N is greater than L^2, or the second largest prime divisor is greater than L. We cannot tell which is the case, so we assume the latter.

We need to work our way up in stages, alternating between exponentiating and searching, never putting too much effort into either in any one stage. This requires some faith on our part; we must be prepared to discard the effort put into searching, starting over with a new β each time. It also means a slight sacrifice in the constant factors: if N is prime, all of the exponentiations past $L \approx 30$ are pointless, as are all the restarted searches. If we are careful, the impact on the worst case will be a factor of about 2. This is a small price to pay to ensure that we are close to the optimal choice of L in every case.

To minimize unnecessary exponentiations, we adopt the following strategy. Using a small provisional value L_1, we compute $\beta_1 = \alpha_1^E$ and search to some initial bound B_1 for $N_1 = |\beta_1|$. If we don't find N_1 we know that $|\beta_1| > B_1$ and should be prepared to increase L_1 to $L_2 = \sqrt{B_1}/c$, for some small constant $c \approx 3$, in the hopes of making $N_2 = |\beta_2|$ smaller than N_1. We then set a new bound $B_2 = b^2 B_1$, where the constant $b \geq 2$ effectively determines the growth rate of both B_i and L_i. We continue in this fashion until we find $N_i = |\beta_i|$. We then apply a fast order computation to determine $|\alpha^{N_i}|$ using a factored exponent E (containing powers of primes bounded by L) and output $|\alpha| = |\alpha^{N_i}|N_i$.

Adopting a staged approach allows us to remove the dependence on the parameter ℓ in Algorithm 4.2. We let K_i denote the maximum prime p_w that may arise during the execution of Algorithm 4.2 up to the bound B_i. The value of K_i can be readily determined from B_i and the growth function $d(k)$.[1] We will be satisfied to remove the prime power

$$\lfloor B_i \rfloor_p = \max\{p^h : p^h \leq B_i\}$$

from $|\alpha|$ for each $p \leq K_i$. This doesn't guarantee $|\beta_i|$ is not divisible by some $p \leq K_i$, but if it is, then $|\alpha|$ is divisible by a large power of p and we will discover this in a later stage. This happens very rarely and when it does we are assured a running time of $O\left(N^{1/3}\right)$.

The main algorithm is given below. It uses the subroutine $\mathcal{S}(\beta, B)$ to search for $|\beta|$ up to the bound B. We use variables subscripted by i for the sake of exposition, but only the values for the current stage need to be maintained.

Algorithm 5.1 (Multi-Stage Sieve)**.** *Given $\alpha \in G$ and constants $b, c, L_1, B_1 \geq 1$, the following algorithm computes $|\alpha|$:*

1. **Initialize**: If $\alpha = 1_G$, return 1, otherwise set $E_0 \leftarrow 1$, $\beta_0 \leftarrow \alpha$, and $i \leftarrow 1$.

2. **Begin Stage i**: If $i > 1$, set $B_i \leftarrow b^2 B_{i-1}$ and $L_i \leftarrow \sqrt{B_i}/c$. Determine K_i and set $E_i \leftarrow \prod_{p \leq L} q$ where $q = \lfloor B_i \rfloor_p$ for $p \leq K_i$ and $q = \lfloor L_i \rfloor_p$ for $p > K_i$.

3. **Exponentiate**: Compute $\beta_i \leftarrow \beta_{i-1}^{E_i/E_{i-1}}$.

4. **Search**: Compute $N_i \leftarrow \mathcal{S}(\beta_i, B_i)$. If $N_i = 0$, increment i and go to step 2.

5. **Compute $|\alpha|$**: Compute $h \leftarrow \mathcal{A}(\alpha^{N_i}, E_i)$ and return hN_i.

The correctness of the algorithm follows from Lemma 4.1 and the correctness of algorithms $\mathcal{S}$ and $\mathcal{A}$. Algorithm $\mathcal{A}$ needs to know the factorization of E_i, so it may be useful to keep track of the prime powers in E_i as they accumulate. Alternatively E_i may be implicitly specified in terms of the bounds L_i and B_i. The latter option is the more space efficient (E_i may get very large). Algorithm $\mathcal{A}$ can then recompute the prime powers p^h of step 2 as it needs them.

[1] There are only about ten values that arise in practice. A lookup table can easily be precomputed.

5.1 Primorial-Steps Search

We now consider the search subroutine $\mathcal{S}(\beta, B)$. Note that even though we have a bound B, the search is still effectively an unbounded search: we hope that $|\beta| \ll B$, and we want the running time to depend on $|\beta|$ not B. It is not necessarily the case that $|\beta|$ is larger than the previous bound; additional exponentiations may have reduced the order of β.[2]

Below is a primorial-steps search algorithm which implements $\mathcal{S}(\beta, B)$. This is essentially Algorithm 4.2 with steps 1 and 7 removed, but we also address the possibility that $|\beta|$ is actually divisible by a prime in P_w. In this scenario we do not really expect to find $|\alpha|$ in the current stage, but we account for the fact that a giant step could land on an integer which is an exponent of β but not equal to $|\beta|$ (if the element $\beta^0 = 1_G$ is in the lookup table, we will find a match). Alternatively, we could ignore this complication and pretend we haven't found $|\beta|$, but it seems foolish not to capitalize on our good luck. A fast order algorithm $\mathcal{A}(\alpha, E)$ is used in step 6 to handle this situation.

Algorithm 5.2 (Primorial Steps Search)**.** *Given $\beta \in G$, a bound B, and a growth function $d(k)$, compute β as follows:*

1. **Initialize**: Maximize w s.t. $P_w \leq d(0)$. Set $k \leftarrow 0$, $b \leftarrow 0$, and $g \leftarrow 0$.
   Store $(1_G, 0)$ in a table.

2. **Adjust P_w**: If $P_{w+1} \leq d(k)$ then

   a. Increment w, minimize $b_1 = mP_w \geq b$, and minimize $g_1 = mP_w \geq g$.

   b. For j from b to b_1, if $j \perp P_w$, compute β^j and store (β^j, j) in the table.
   If any $\beta^j = 1_G$, set $N \leftarrow j$ and go to step 6, otherwise set $b \leftarrow b_1$.

   c. Set $i \leftarrow g_1$, compute β^i, and do a table lookup.
   If $\beta^i = \beta^j$, set $N \leftarrow i - j$ and go to step 6, otherwise set $g \leftarrow g_1$.

   Maximize m s.t. $mP_w \leq d(k)$, then set $s \leftarrow mP_w$ and $s' \leftarrow m\phi(P_w)$.

3. **Baby Steps**: For j from b to $b + s$, if $j \perp P_w$, compute β^j and store (β^j, j) in the table.
   If any $\beta^j = 1_G$, set $N \leftarrow j$ and go to step 6, otherwise set $b \leftarrow b + s$.

4. **Giant Steps**: For $i = g + b, g + 2b, \ldots, g + s'b$, if $i > B$ return 0, otherwise lookup β^i.
   If any $\beta^i = \beta^j$, set $N \leftarrow i - j$ and go to step 6, otherwise set $g \leftarrow g + s'b$.

[2] We do know that $|\beta_i|$ is either 1 or greater than L_i, but this is of minimal use.

5. **Iterate**: Increment k and go to step 3.

6. **Finalize**: Factor N and return $\mathcal{A}(\beta, N)$.

If Algorithm 5.2 uses t group operations, then N will be $O\left(t^2 \log\log t\right)$, so the time required to factor N in step 6 is negligible, and requires no group operations in any event.

5.2 Rho Search and Hybrid Search

It is not necessary to use Algorithm 5.2 to implement $\mathcal{S}(\beta, B)$. We can use Algorithm 3.1 based on the rho method instead. This increases the running time by a factor of $\Theta\left((\log\log N)^{-1/2}\right)$, but addresses the more significant obstacle of space limitations. The exponentiations and fast order computations performed by Algorithm 5.1 can all be done in polylogarithmic space (see Algorithm 7.1). The E_i are exponentially large, but they can be stored implicitly as ranges of prime powers.

The worst-case performance is then no longer $o\left(\sqrt{N}\right)$, but from a practical point of view we have only lost a factor of 2 or 3 and can still achieve $O\left(N^{0.344}\right)$ median complexity with minimal space requirements. The constant factors are improved by the fact that we do not need to implement the restart strategy described in Section 3.1; the multi-stage sieve restarts even more frequently and uses a tighter bound.

To gain the best of both search methods, a hybrid approach may be used. Algorithm 5.1 can use a primorial-steps search in the early stages, and if space becomes limited, switch to a rho search. This is easy to implement and gives good results. Small searches are noticeably faster; the asymptotic behavior of the $\Theta\left((\log\log N)^{-1/2}\right)$ factor means that most of the benefit is realized early, and the primorial-steps search has better constant factors for small values of N. Large searches would be impractical without the rho method, so we don't mind the slight reduction in performance when N is large.

5.3 Complexity Analysis

To analyze the complexity of Algorithm 5.1, we consider the termination conditions. For the sake of simplicity, suppose $N = pqr$, where $p > q$ are prime and r contains no prime

power greater than q (r is q-powersmooth). Algorithm 5.1 terminates in stage t with

$$L_{t-1} < q_* \leq L_t, \tag{5.1}$$

where $q_* = \max(q, \sqrt{p}/c)$. To reach this point will require, at most, approximately

$$(\log^{-1} 2)(L_1 + \ldots + L_t) \leq \left(\frac{b^2}{b-1}\right)(\log 2)^{-1} q_* \tag{5.2}$$

group operations spent on exponentiations, since $L_t = bL_{t-1} \leq bq_*$. If we let $S(B)$ denote the number of group operations used searching to the bound B, the total cost of searching is

$$S(B_1) + \ldots S(B_{t-1}) + S(p).$$

For simplicity, we suppose $S(B) = a\sqrt{B}$ for some "constant" a which may also incorporate the $(\log\log B)^{-1/2}$ factor in the case of the primorial-steps search. Since $B_i = (cL_i)^2$ and $\sqrt{p} \leq cq_*$, we then have approximately

$$ac\left(\frac{2b-1}{b-1}\right) q_* \tag{5.3}$$

group operations spent searching. The fast order computation at the end uses about $(\log 2)^{-1} L_t$ group operations, which may be bounded by

$$b(log2)^{-1} q_*. \tag{5.4}$$

The optimal choice of b is around 2, but depends on N, q_*, and which case one wishes to optimize. The optimal choice of c is between 1 and 3, but depends on a and whether $q^2 > p$, since decreasing c potentially increases $q_* = \max(q, \sqrt{p}/c)$. We will not attempt a precise analysis of the constant factors, other than to note that they are all reasonably small. The values of b and c used in the performance tests are given in Chapter 11. The key point is that the total running time is bounded by a constant factor of q_*.

Definition 5.1. *For any integer x, let $q_*(x)$, be the least y such that the largest prime-power divisor of x is at most y^2 and the second largest prime-power divisor of x (if any) is at most y.*

The value $q_*(N)$ is a convenient upper bound on q_* above. If $q_*(x) = y$ then x has no

prime factors greater than y^2, and its second largest prime factor is at most y. Except for the rare case of numbers containing large powers of small primes, $q_*(x)$ is the least y for which x is semismooth with respect to y^2 and y (see Definition 6.7).[3]

Lemma 5.1. *If the search algorithm $\mathcal{S}(\beta, B)$ has complexity $O\left(B^{1/2}\right)$, then the complexity of Algorithm 5.1 on input $\alpha \in G$ is $O\left(q_*(N)\right)$, where $N = |\alpha|$.*

Proof. For $i > 1$, let $L_i = b^{i-1}\sqrt{B_1}/c$, as computed by step 1 of Algorithm 5.1, and let L_t be the least $L_i \geq q_*(N)$. Then $|\beta_i| \leq q_*^2(N)$, since all the prime powers in $|\alpha|$ for primes less than L_t have been removed. It follows that the bound $B_t = (cL_t)^2 > |\beta_i|$, since $c > 1$, and the search in stage i will find $|\beta_i|$ (assuming $|\alpha|$ has not already been determined). Since $q_*(N)$ satisfies (5.1), it follows from the discussion above that the total number of group operations may be bounded by a constant factor of the sum of (5.2), (5.3), and (5.4), with q_* replaced by $q_*(N)$, yielding an $O\left(q_*(N)\right)$ bound. □

The argument above can easily be extended to incorporate the $(\log\log N)^{-1/2}$ factor in the complexity bound for the primorial-steps algorithm.

Corollary 5.2. *If a primorial-steps search is used by Algorithm 5.1 on input $\alpha \in G$, its complexity is $o\left(q_*(N)\right)$, where $N = |\alpha|$.*

We can now prove the main proposition, showing that the multi-stage sieve achieves essentially the same average and median complexity as the primorial-steps algorithm (see Propositions 4.6 and 4.7).

Proposition 5.3. *Let $T(N)$ denote the complexity of Algorithm 5.1 on input $\alpha \in G$ with $|\alpha| = N$, and assume algorithm $\mathcal{S}(\beta, B)$ has complexity $O\left(B^{1/2}\right)$. If N is uniformly distributed over $[1, M]$, then for all sufficiently large M,*

$$\mathbf{E}[T(N)] = O\left(\frac{\sqrt{M}}{\log M}\right), \tag{5.5}$$

and for any $u > 2$ there exists a constant c such that

$$\Pr\left[T(N) \leq cN^{1/u}\right] \geq G(1/u, 2/u), \tag{5.6}$$

where $G(r, s)$ is the semismooth probability function of Definition 6.8.

[3] If $y = q_*(x)$, we could say x is "semipowersmooth" with respect to y^2 and y.

Proof. Applying Lemma 5.1, we prove (5.5) by bounding the expected value of $q_*(N)$. Let $y = M^{1/3}$. For each $x \in [1, M]$, either $q_*(x) \leq M^{1/3}$ or $q_*(x) \leq \sqrt{\kappa_y(x)}$ (see Definition 6.1). We bound the expectation by summing over both cases, effectively overcounting. Thus

$$\mathbf{E}[q_*(N)] \leq \frac{1}{M} \sum_{1 \leq x \leq M} M^{1/3} + \frac{1}{M} \sum_{1 < x \leq M} \sqrt{\kappa_y(x)} = O\left(M^{1/3}\right) + O\left(\frac{\sqrt{M}}{\log M}\right) = O\left(\frac{\sqrt{M}}{\log M}\right),$$

where we apply Proposition 6.22 to the function $f(x) = x^{1/2}$ in the second sum. To prove (5.6), we first note that the probability that N is divisible by the square of any number greater than $M^{1/3u}$ is at most

$$\frac{1}{M} \sum_{M^{1/3u} < n \leq M^{1/2}} \left\lfloor \frac{M}{n^2} \right\rfloor \leq \sum_{M^{1/3u} < n \leq M^{1/2}} \frac{1}{n^2} = O\left(M^{-1/3u}\right).$$

Therefore, we may assume that if N is semismooth with respect to $N^{2/u}$ and $N^{1/u}$, then $q_*(N) \leq N^{1/u}$ with probability arbitrarily close to 1 $\left(\text{if not then } N \text{ contains a prime power } p^h > N^{1/3u} \text{ with } h > 1 \text{ and is divisible by the square of a power of } p \text{ greater than } N^{3/u}\right)$. The inequality (5.6) then follows from Lemma 5.1 and the definition of $G(1/u, 2/u)$. □

Corollary 5.4. *The median time complexity of Algorithm 5.1 using either a primorial-steps search (Algorithm 5.2) or a rho search (Algorithm 3.1) is* $O\left(N^{0.344}\right)$. *For the primorial-steps search, the median space complexity is* $O\left(N^{0.344}\right)$, *and for the rho search the space complexity in every case is polylogarithmic in* N.

5.4 A Generic Recipe for Subexponential Algorithms

Readers familiar with algorithms for integer factorization will have noticed similarities with the multi-stage sieve. Pollard's $p - 1$ method [83] may be viewed as an order computation in the multiplicative group $\mathbb{Z}_N^*$, where N is the integer to be factored. In this algorithm, a random a is chosen and then exponentiated by all the prime powers up to some bound L to obtain b. If $p - 1$ is L-powersmooth for some p dividing N, then $b \equiv 1 \bmod p$. This implies that $d = \gcd(b - 1, N)$ is not 1, and provided $d \neq N$ (almost certainly the case), d is a non-trivial divisor of N. This is called the first stage. If it fails, that is, $\gcd(b - 1, N) = 1$, then the algorithm moves to a second stage in which specific additional powers $b' = b^p$ are individually computed for primes up to some second stage bound B, typically $B \approx L \log L$.

For each b', $\gcd(b'-1, N)$ is computed in the hopes of finding a non-trivial divisor of N.

The first stage of Pollard's $p-1$ method is analogous to the exponentiation phase in the multi-stage sieve algorithm, and the second stage corresponds to the search phase. In the case of the multi-stage sieve, it effectively alternates between first and second stages, to avoid spending an unnecessary amount of effort in the first stage. This is important because it is able to search much further than the standard $p-1$ algorithm, with $B = L^2$ or $B = L^2 \log\log L$ in the case of the primorial-steps search. This makes it more likely that the search phase is where the algorithm will actually succeed. The exponentiation phase is necessary to make this possible, since it (hopefully) reduces the order of β to bring it within reach of the search.

Subexponential Methods for Integer Factorization

The ideas embodied in Pollard's $p-1$ algorithm are carried further in more sophisticated factoring algorithms, such as Lenstra's elliptic curve method (see [37, 40]), the Schnorr-Lenstra class group algorithm [92], and a similar class group algorithm known as SPAR, after Shanks, Pollard, Atkins, and Rickert (see [31]). These algorithms are able to achieve subexponential running times (in almost all cases) by attempting a random order computation which is related in some way to the integer whose factorization is being attempted. The probability of success in any particular group is low, but by trying to solve many random problems, they eventually succeed in finding one that is easy to solve. In each group both a first and second stage are employed. In effect, they alternate between exponentiating and searching much like the multi-stage sieve algorithm, but switch groups after an unsuccessful search rather than continuing in the same group.

The reason these algorithms are able to achieve subexponential performance has to do with the distribution of smooth numbers discussed in Chapter 6. If they manage to find a group element α with sufficiently smooth order, they will be successful. Assuming that $N = |\alpha|$ is uniformly distributed over some appropriate interval, the probability that N is $N^{1/u}$-smooth is $\rho(u) = u^{-u+o(1)}$, by Theorems 6.13 and 6.14. If we suppose the effort required to find $|\alpha|$ in such a case is $O\left(N^{1/u}\right)$ (this ignores the second stage), then the expected running time is approximately

$$\left(\rho(u)\right)^{-1} O\left(N^{1/u}\right) = u^{u+o(1)} N^{1/u} = \exp\left(\frac{1}{u}\log N + u\log u + o(1)\right). \tag{5.7}$$

This is minimized for $u \approx \sqrt{2 \log N / \log \log N}$, giving an expected running time of

$$L[1/2, \sqrt{2}] = \exp\Big((\sqrt{2} + o(1)) \sqrt{\log N \log \log N} \Big), \tag{5.8}$$

where the notation $L[a, c] = \exp\Big(\big(c + o(1)\big) (\log N)^a (\log \log N)^{(1-a)} \Big)$ is a standard shorthand for such asymptotic expressions. The value $N = |\alpha|$ in (5.8) is typically proportional to the square root of the number being factored; in terms of the input value, the complexity is $L[1/2, 1]$. Note that we did not consider the second stage in the analysis above. Doing so improves the constant factors, but the asymptotic expression remains unchanged. In practice, however, the second stage makes a great deal of difference; an efficient implementation of these algorithms must give it careful consideration. We consider this issue further in the next section.

There are faster algorithms for factoring integers, but the ones mentioned above are most similar to the multi-stage sieve and suggest a general method for obtaining subexponential algorithms for any problem that can be reduced to random order computations.

A Generic Algorithm with Subexponential Complexity

Suppose we are given a sequence of problems to solve, each randomly selected from some large set. We are not required to solve any particular problem, rather we will be measured by the time it takes us to successfully solve one problem.

The factoring algorithms of the previous section construct such a situation by generating a sequence of random groups associated with the integer to be factored, and then attempt to compute the order of a random element in each group. In the case of the class group method, the sequence of groups are ideal class groups in quadratic number fields with discriminants that are multiples of the integer to be factored (see Chapter 10).

We may take the same approach with a generic algorithm using any finite group. The transformation of the original problem into a sequence of random order computations may involve the specific representation of the problem and the construction of associated black boxes, but for any problem where such a transformation is possible, we can obtain a generic solution with subexponential complexity.

Consider a sequence of order computations where both α and G may be random. Provided $|\alpha|$ is uniformly distributed over some interval $[1, N]$, we can apply Proposition

5.3, which doesn't depend on the group G. Suppose that we pick some u and attempt to compute $|\alpha|$ for each problem instance until we have used $O\left(N^{1/u}\right)$ group operations, at which point we give up and try the next problem. Our expected running time will then be

$$(G(1/u, 2/u))^{-1} O\left(N^{1/u}\right)$$

group operations. Now clearly $G(1/u, 2/u) > \rho(u)$, since any number which is $N^{1/u}$ smooth is certainly semismooth with respect to $N^{2/u}$ and $N^{1/u}$, so we may use the approximation $\rho(u) = u^{-u+o(1)}$ as a lower bound on $G(1/u, 2/u)$. If we choose $u = \sqrt{2 \log N / \log \log N}$ as in (5.8), we will obtain a running time of $L[1/2, \sqrt{2}]$, subexponential in $\log N$, and presumably subexponential in the size of the original problem from which N was derived. As noted above, in the case of integer factorization, N is proportional to the square root of the original problem.

Proposition 5.5. *Given a sequence of problem instances $\alpha_i \in G_i$, where each $|\alpha_i|$ is uniformly and independently distributed over $[1, N]$, the expected number of group operations required by the multi-stage sieve algorithm to solve one problem in the sequence is bounded by $L[1/2, \sqrt{2}]$. If a primorial-steps search is used, the group storage is $L[1/2, \sqrt{2}/2]$, and if a rho search is used, it is polylogarithmic in N.*

The proposition follows from the discussion above and Proposition 5.3. The running time is proportional to the square of the number of problem instances attempted, hence the space required is proportional to the square root of the time. This is useful, because it means that a primorial-steps search can usually be applied.

When using Algorithm 5.1 to implement the subexponential method described above, only a single stage is used. The value u is chosen based on the size of the problem and an estimation of the success probability (see Remark 5.1). Once the choice of u has been made, L_1 and B_1 are set appropriately, and the algorithm is instructed to terminate if the search reaches B_1 so that the next problem instance may be attempted. In the nomenclature of factoring algorithms, L_1 corresponds to the first stage boundary, and B_1 corresponds to the second stage boundary.

The Importance of the Search Phase

When using $\rho(u)$ as a lower bound on $G(1/u, 2/u)$ in Proposition 5.5, we effectively discounted the entire search phase of Algorithm 5.1. We could attempt a tighter analysis, but asymptotic bounds are not especially useful in this instance. Implicit in the big-O notation, is the assumption that $u \to \infty$. This is theoretically true of course, since $u \approx \sqrt{2 \log N / \log \log N}$, but in practice u is rarely greater than 10 and essentially never above 20. Of far greater importance than the asymptotic relationship between $G(1/u, 2/u)$ and $\rho(u)$, is their ratio over the range of typical values of u.

Table 6.1 lists values for both functions. We find that $G(1/u, 2/u)$ is nearly 200 times larger than $\rho(u)$ when $u = 10$. Even for more pedestrian values of u which might arise in the median case, $G(1/u, 2/u)$ is nearly ten times $\rho(u)$ when $u = 3$; fewer than 5% of the numbers in a large interval are $N^{1/3}$-smooth, but almost 45% are semismooth with respect to $N^{2/3}$ and $N^{1/3}$.

In the case of subexponential algorithms where the value of u is larger, these differences get magnified and the impact of the search phase becomes more pronounced. This also applies to constant factors and the $(\log \log N)^{-1/2}$ factor in the complexity of the primorial-steps search. These all are effectively squared when solving a random sequence of problems, since the expected running time is proportional to the square of the running time on a single problem instance. The total difference made by the search phase when all these factors are taken into account may be an improvement by a factor of more than 1000. This can easily be the difference between a problem that is tractable and one that is not.

Remark 5.1. *The difference in the probabilities implied by $\rho(u)$ and $G(1/u, 2/u)$ noted above impacts the optimal choice of u. Additionally, the distribution of $|\alpha|$ may not be uniform. The semismooth probabilities observed for large class groups, for example, are noticeably better than predicted by $G(1/u, 2/u)$ (see Chapter 11). In practice it is often necessary to obtain an estimate for u empirically to achieve the best running time. Either underestimating or overestimating u will have a deleterious effect on performance.*

Chapter 6

A Number-Theoretic Interlude

Chapter Abstract

This chapter collects some useful results from classical number theory, and reviews more recent work on the distribution of numbers with no small prime factors, and those with no large prime factors. We also develop practical techniques that simplify the analysis of algorithms whose complexity depends on these and other number-theoretic distributions.

The complexity of generic order computations is intimately related to the multiplicative structure of the natural numbers. It is often not the size of the numbers involved, but the size of their prime factors, that determines the computational resources required. It is thus useful to distinguish numbers based on the size of the primes they contain. Of particular interest are numbers whose prime factors are all "small" (*smooth* numbers), and those composed entirely of "large" primes (*coarse* numbers).

6.1 The Basics

The fundamental theorem of arithmetic states that every natural number has a unique prime factorization. Given a real number y, we may partition the prime factors of a natural number according to their relationship with y.

Definition 6.1. *Let y be a real number. For $n = p_1^{h_1} \cdots p_w^{h_w}$ with $p_1 < \cdots < p_w$ prime, define:*

1. $\boldsymbol{\sigma_y(n)} = \prod_{p_i \le y} p_i^{h_i}$ *is the* **y-smooth part** *of n. If $\sigma_y(n) = n$ then n is* **y-smooth**.
2. $\boldsymbol{\kappa_y(n)} = \prod_{p_i > y} p_i^{h_i}$ *is the* **y-coarse part** *of n. If $\kappa_y(n) = n$ then n is* **y-coarse**.

The function σ_y is completely multiplicative: $\sigma_y(ab) = \sigma_y(a)\sigma_y(b)$ for any natural numbers a and b. The same is true of κ_y, and in every case we have $n = \sigma_y(n)\kappa_y(n)$. Thus each y splits every natural number into a y-smooth and a y-coarse part. We are especially interested in the cases where one of these parts is trivial.

Definition 6.2. *Let n denote a natural number and let x and y be real numbers.*

1. $\boldsymbol{\Psi_y(x)} = \{n \leq x : \kappa_y(n) = 1\}$ *is the set of y-smooth numbers bounded by x. The set of all y-smooth numbers is denoted* $\boldsymbol{\Psi_y}$.

2. $\boldsymbol{\Phi_y(x)} = \{n \leq x : \sigma_y(n) = 1\}$ *is the set of y-coarse numbers bounded by x. The set of all y-coarse numbers is denoted* $\boldsymbol{\Phi_y}$.

The Riemann zeta function, $\zeta(s) = \sum_{n=1}^{\infty} n^{-s}$, plays a key role in the study of prime numbers. We may define a restricted form of the zeta function whose sum is limited to y-smooth numbers:

$$\zeta_y(s) = \sum_{n \in \Psi_y} \frac{1}{n^s}. \tag{6.1}$$

Equivalently, the function $\zeta_y(s)$ is the Dirichlet series for the characteristic function of Ψ_y. Unlike the general zeta function, $\zeta_y(s)$ converges when $s = 1$, since

$$\prod_{p \leq y} \left(\frac{1}{1 - p^{-1}} \right) = \prod_{p \leq y} \left(\sum_{h=0}^{\infty} \frac{1}{p^h} \right) = \sum_{n \in \Psi_y} \frac{1}{n^s} = \zeta_y(1). \tag{6.2}$$

The product on the left is evidently finite, and we have rearranged terms in a finite product of absolutely convergent sums to obtain $\zeta_y(1)$. The equivalence of the product and sum above is an example of an Euler product, and in general we have $\zeta_y(s) = \prod_{p \leq y}(1 - p^{-s})^{-1}$. The fact that $\zeta_y(1)$ converges while $\zeta(1)$ does not is yet another testament to the infinitude of primes.

Definition 6.3. *For $n = p_1^{h_1} \cdots p_w^{h_w}$ with $p_1 < \cdots < p_w$ prime:*

1. $\boldsymbol{\mu(n)} = (-1)^w$ *when n is square-free and 0 otherwise.*

2. $\boldsymbol{\phi(n)}$ *counts the natural numbers $m \leq n$ such that $m \perp n$.*

Note that $\mu(1) = \phi(1) = 1$. The relation $m \perp n$ indicates that m and n are coprime (have no prime factor in common). The function $\phi(n)$ is Euler's "phi" function and represents

the order of the multiplicative group $\mathbb{Z}_n^*$. The function $\mu(n)$, the Möbius function, provides a convenient way to compute $\phi(n)$:

$$\phi(n) = \sum_{d|n} \mu(d)\frac{n}{d} = n\prod_{p|n}\left(1 - \frac{1}{p}\right). \tag{6.3}$$

The central sum may be viewed as counting the numbers up to n that are coprime to n via the inclusion/exclusion principle, justifying both equalities. We are particularly interested in applying (6.3) to the case where $P = \prod_{p\leq y} p$ is a *primorial* number. The proportion of coarse numbers in $[1, P]$ may be expressed as

$$\frac{\phi(P)}{P} = \prod_{p\leq y}\left(1 - \frac{1}{p}\right) = \prod_{p\leq y}\left(\frac{1}{1-p^{-1}}\right)^{-1} = \frac{1}{\zeta_y(1)}. \tag{6.4}$$

An asymptotic value for (6.4) is given by Mertens' theorem.

Theorem 6.1 (Mertens)**.** *For $x \geq 2$,*

$$\prod_{p\leq x}\left(1 - \frac{1}{p}\right) = \frac{1}{e^\gamma \log x} + O\left(\frac{1}{\log^2 x}\right),$$

where $\gamma \approx 0.5772$ is Euler's constant.

See [107, p. 17] for a proof. This implies $\zeta_y(1) \sim e^\gamma \log y$. The appearance of Euler's constant here is not too surprising, as it represents the asymptotic difference between $\log y$ and $\sum_{n\leq y} \frac{1}{n}$, a partial sum of $\zeta(1)$.

The most important classical result for our work is the prime number theorem.

Theorem 6.2 (PNT)**.** *Let x be a real number, define $\vartheta(x) = \sum_{p\leq x} \log p$, and let $\pi(x)$ count the primes less than or equal to x.*

1. $\vartheta(x) = x + O\left(\dfrac{x}{\log x}\right)$.
2. $\pi(x) = \mathrm{li}(x) + O\left(\dfrac{x}{\log^2 x}\right) = \dfrac{x}{\log x} + O\left(\dfrac{x}{\log^2 x}\right)$.

where $\mathrm{li}(x) = \int_2^x \log t\, dt$ is the logarithmic integral.

The function $\vartheta(x)$ is known as Chebyshev's function. The two statements are easily shown to be equivalent; we will have use for both. Various proofs of the prime number

theorem can be found in many standard texts, see [77] for an elementary proof, or [2] for an analytic proof. The error terms above are not the best possible, but suffice for our work. For $P = \prod_{p \le x} p$, the prime number theorem (PNT) implies

$$\log(P) = \sum_{p \le x} \log p = \vartheta(x) \ \sim \ x. \tag{6.5}$$

The last classical result we mention is Bertrand's postulate.

Theorem 6.3 (Bertrand's Postulate)**.** *Let p_n denote the nth prime. Then $p_{n+1} < 2p_n$ for all n.*

Equivalently, the interval $[n, 2n)$ always contains a prime. See [58, Theorem 418] for a proof. We are now in a position to prove a useful corollary of the three theorems above.

Lemma 6.4. *Let $P_n = \prod_{p \le p_n} p$ be the product of the first n primes. If $P_n \le x < P_{n+1}$ then*

$$\frac{\phi(P_n)}{P_n} \ \sim \ \frac{1}{e^{\gamma} \log\log x},$$

where $\gamma \approx 0.5772$ is Euler's constant.

Proof. Applying Mertens' theorem,

$$\frac{\phi(P_n)}{P_n} = \prod_{p \le p_n} \left(1 - \frac{1}{p}\right) \ \sim \ \frac{1}{e^{\gamma} \log p_n}. \tag{6.6}$$

It suffices to show $\log x \sim p_n$. By the hypothesis, $\log P_n \le \log x < \log P_{n+1}$, hence

$$0 \le \log x - \log P_n < \log P_{n+1} - \log P_n = \log p_{n+1} < \log p_n + \log 2, \tag{6.7}$$

where the last inequality uses Bertrand's postulate. By the PNT, $\log P_n = \vartheta(p_n) \sim p_n$, thus

$$0 \le \log x - \log P_n < \log\log P_n + c, \tag{6.8}$$

for any $c > \log 2$. This implies $\log x \sim \log P_n \sim p_n$, as desired. □

This lemma is used in Proposition 4.2 in Chapter 4. We note that the prime number theorem is not required to prove the lemma above; $\vartheta(x) = \Theta(x)$ suffices. We shall not be restrained in our application of the PNT. It is often the most direct approach.

The asymptotic formula for $\pi(x)$ given by the PNT can be used to estimate sums over primes. Since $\pi(x)$ is monotonic, we may express the sum of any continuous function $f(x)$ taken over primes as a Stieltjes integral:[1]

$$\sum_{a<p\leq x} f(p) = \int_a^x f(t)d\pi(t). \tag{6.9}$$

If we now apply the PNT to write $\pi(x) = \mathrm{li}(x)+\varepsilon(x)$, where $\varepsilon(x) = \mathrm{li}(x)-\pi(x) = O\left(x/\log^2 x\right)$, we have an effective means to estimate the integral in (6.9). This technique may be applied to any counting function for which we have an asymptotic formula.

Definition 6.4. *Let S be a set of natural numbers. The* **counting** *function for S is $c_S(x) = \left|S\cap[1,x]\right|$.*

The theorem below encapsulates the technique in a more general form. The case where S is the set of primes appears in [90] and may also be found in [10].

Theorem 6.5. *Let S be a set of natural numbers with counting function $c_S(x) = G(x) + \varepsilon(x)$ where $G(x) = \int_{a_0}^x g(x)dx$ for some $g(x)$ continuous over $[a_0,\infty)$. Let $f(x)$ be continuously differentiable over an open interval containing $[a,\infty)$. Then for $x > a \geq a_0$,*

$$\sum_{\substack{a<n\leq x;\\ n\in S}} f(n) = \int_a^x f(t)g(t)dt \;+\; \left[\epsilon(t)f(t)\right]_a^x \;-\; \int_a^x \varepsilon(t)f'(t)dt.$$

Proof. Expressing the sum as a Stieltjes integral, we have

$$\sum_{\substack{a<n\leq x;\\ n\in S}} f(n) = \int_a^x f(t)dc_S(t) = \int_a^x f(t)d\big(G(t)+\varepsilon(t)\big) = \int_a^x f(t)g(t)dt + \int_a^x f(t)d\varepsilon(t),$$

where the existence of the integrals is guaranteed by the fact that $c_S(x)$ is monotonic and $f(x)$ and $G(x)$ are continuous. Integrating the rightmost term by parts completes the proof. □

Theorem 6.5 is typically applied in cases where it is known that $s(x) \sim G(x)$ and the two terms involving $\varepsilon(x)$ can be bounded, leaving just the first integral.

[1] It suffices for $f(x)$ to share no discontinuities with $\pi(x)$ over the interval. See [1, Ch. 7] for a detailed presentation of the Stieltjes integral.

6.2 The Distribution of Coarse Numbers

Like the prime numbers, the coarse numbers have an associated counting function.

Definition 6.5. *The function* $\boldsymbol{\phi(x, y)} = |\Phi_y(x)|$ *counts the y-coarse numbers bounded by x.*[2]

The function $\phi(x, y)$ may be viewed as counting the natural numbers bounded by x that are coprime to the primorial $P = \prod_{p \le y} p$. Counting these via the inclusion-exclusion principal yields the Legendre formula:

$$\phi(x, y) = \sum_{d|P} \mu(d) \left\lfloor \frac{x}{d} \right\rfloor . \tag{6.10}$$

Equation (6.10) may also be interpreted as counting the number of unmarked entries in a sieve of Eratosthenes on the set of integers in $[1, x]$ after all the primes up to y have been processed.

The primorial-steps algorithm makes use of the fact that every natural number $n \perp P$ can be written uniquely in the form $n = aP + r$, where $r < a$ is coprime to P. Since there are $\phi(P)$ possible values r can take between multiples of P, we have

$$\phi(aP + r, y) = a\phi(P) + \phi(r, y). \tag{6.11}$$

Both equations (6.10) and (6.11) lead to explicit bounds for $\phi(x, y)$.

Theorem 6.6 (Legendre)**.** *For all real numbers x and y,*

$$\frac{x}{\zeta_y(1)} - 2^{\pi(y)} \le \ \phi(x, y) \ \le \frac{x}{\zeta_y(1)} + 2^{\pi(y)},$$

where $\zeta_y(1) = \prod_{p \le y} \left(\frac{1}{1 - p^{-1}} \right) \ \sim \ e^{\gamma} \log y.$

Proof. Let $P = \prod_{p \le y} p$. The Legendre formula (6.10) implies:

$$\phi(x, y) = \sum_{d|P} \mu(d) \left\lfloor \frac{x}{d} \right\rfloor = x \sum_{d|P} \frac{\mu(d)}{d} + \varepsilon,$$

[2]The notation $\phi(x, n)$ is used in [88] to count the p_n-coarse numbers, but the notation used here is generally standard. One also sees $\pi(x, y)$.

where $|\varepsilon| \le 2^{\pi(y)}$. The error term arises from replacing $\lfloor x/d \rfloor$ by x/d; we introduce an error of at most $|\mu(d)| \le 1$ for each of the $2^{\pi(y)}$ divisors of P. Applying (6.3) and (6.4) gives

$$\sum_{d|P} \frac{\mu(d)}{d} = \frac{1}{P} \sum_{d|P} \mu(d) \frac{P}{d} = \frac{\phi(P)}{P} = \frac{1}{\zeta_y(1)},$$

which completes the proof. □

If x happens to be asymptotically larger than $2^{\pi(y)} \log y$ we obtain a tight estimate, but the theorem applies in all cases. The estimate is clearly not correct for y near x, as shown by the example $\phi(x, x) = 1 + \pi(x) \sim x/\log x$. Surprisingly, it is only off by a constant factor of $e^{-\gamma} \approx 0.5615$ in this case, despite the fact that the error term is *much* larger than $x/\log x$. In fact, the estimate in Theorem 6.6 is quite close to the true asymptotic value of $\phi(x, y)$ in general, with $\phi(x, x)$ representing the greatest deviation.

We now consider the case that $y = x^{1/u}$ for some real $u > 1$. For $u < 2$, a y-coarse number bounded by x can have at most one prime factor, hence

$$\phi(x, y) = 1 + \pi(x) - \pi(y) = \frac{x}{\log x} + O\left(\frac{x}{\log^2 x}\right) \qquad (\text{for } \sqrt{x} < y \le x/\log x). \tag{6.12}$$

When u lies between 2 and 3, a y-coarse number bounded by x may have up to two prime factors. To compute $\phi(x, y)$ in this case, we need the following lemma:

Lemma 6.7. *Let $\phi_2(x, y)$ count the numbers in $\Phi_y(x)$ with 2 (not necessarily distinct) prime factors. Uniformly, for $1 < y < \sqrt{x}$,*

$$\phi_2(x, y) = \log(u - 1) \left(\frac{x}{\log x}\right) + O\left(\frac{x}{\log^2 x}\right),$$

where $u = \log x/\log y > 2$.

Proof. Every n counted by $\phi_2(x, y)$ is the product of primes p and q with $x^{1/u} < p \le q \le x/p$. Summing over p gives

$$\phi_2(x, y) = \sum_{x^{1/u} \le p \le x^{1/2}} (\pi(x/p) - \pi(p) + 1) = \sum_{x^{1/u} \le p \le x^{1/2}} \frac{x}{p \log(x/p)} + O\left(\frac{x}{\log^2 x}\right). \tag{6.13}$$

We now apply Theorem 6.5 to $f(t) = x/\big(t\log(x/t)\big)$, with x fixed, summed over primes:

$$\sum_{x^\delta \le p \le x^{1/2}} \frac{x}{p\log(x/p)} = \int_{x^\delta}^{x^{1/2}} \frac{x}{t\log(x/t)\log(t)}dt \;+\; \varepsilon(t)f(t)\Big|_{x^\delta}^{x^{1/2}} - \int_{x^\delta}^{x^{1/2}} f'(t)\varepsilon(t)dt. \tag{6.14}$$

By the PNT, $\varepsilon(t) = O\left(x/\log^2 x\right)$, and the last two terms are readily seen to be $O\left(x/\log^2 x\right)$. The first integral may be evaluated to yield

$$\int_{x^\delta}^{x^{1/2}} \frac{x}{t\log(x/t)\log(t)}dt = \left(\frac{x}{\log x}\right)\big(\log\log t - \log\log(x/t)\big)\Big|_{x^\delta}^{x^{1/2}} = \left(\frac{x}{\log x}\right)\log(u-1). \tag{6.15}$$

Incorporating (6.15) and the error terms from (6.14) into (6.13) gives the desired result. □

Corollary 6.8. *For $x^{1/3} \le y \le x^{1/2}$,*

$$\phi(x,y) = \big(1+\log(u-1)\big)\left(\frac{x}{\log x}\right) + O\left(\frac{x}{\log^2 x}\right),$$

where $u = \log x/\log y$.

Roughly speaking, the corollary implies $\pi(x) \lesssim \phi(x, x^{1/u}) \lesssim 1.7\pi(x)$ when u is between 2 and 3, and $\phi(x, x^{1/u})$ may be brought arbitrarily close to $\pi(x)$ by choosing u close to 2.

The technique used in Lemma 6.7 can be applied inductively to compute $\phi(x,y)$ for successively larger values of u. The result does not lead to a simple closed form. One is led to define the function $\omega(u)$ satisfying the differential-difference equation

$$\big(u\omega(u)\big)' = \omega(u-1) \qquad\qquad (u > 2),$$

with $u\omega(u) = 1$ for $1 \le u \le 2$. The existence and uniqueness of $\omega(u)$ was shown by Buchstab [30], who proved $\phi(x,y) \sim u\omega(u)x/\log x$. He also found that $\omega(u)$ converges rapidly to $e^{-\gamma}$ as $u \to \infty$, matching our expectation from Theorem 6.6. We restate these results more generally in the following theorem.

Theorem 6.9 (Buchstab)**.** *Uniformly for $1 < y \le x/\log x$,*

$$\phi(x,y) = \omega(u)\left(\frac{x}{\log y}\right) + O\left(\frac{x}{\log^2 y}\right),$$

where $u = \log x/\log y \ge 1$, and $\omega(u)$ is Buchstab's function.

A proof of this theorem, along with related results, may be found in Tenenbaum's book [107], which devotes an entire chapter to the distribution of coarse numbers.

For $u \leq 3$, equation (6.12) and Corollary 6.8 give precise asymptotic values for $\phi(x, y)$. When u is greater than 3, the agreement between the bound in Theorem 6.9 and the bound in Theorem 6.6 is extremely close, since $\omega(u)$ converges rapidly to $e^{-\gamma}$. Explicit computations [35] show that for $u > 3$ we have

$$-6.36653 \times 10^{-4} < \omega(u) - e^{\gamma} < 5.68381 \times 10^{-3}, \tag{6.16}$$

and for $u > 8$ the agreement is better than one part in a billion. It is never possible to bound one by the other; Hildebrand shows in [60] that the value of $\omega(u) - e^{-\gamma}$ changes sign infinitely often as $u \to \infty$. However, we may use the upper bound $e^{\gamma} + 0.006$ implied by (6.16), which is valid for all $u \geq 1$.

6.3 The Distribution of Smooth Numbers

The counting function for smooth numbers is analogous to $\phi(x, y)$.

Definition 6.6. *The function $\psi(x, y) = |\Psi_y(x)|$ counts the y-smooth numbers bounded by x.*

The 2-smooth numbers are exactly the powers of 2, hence

$$\psi(x, 2) = \lfloor \lg x \rfloor + 1 = \frac{\log x}{\log 2} + O(1).$$

The case $y = 3$ was considered by Ramanujan [16] who found that

$$\psi(x, 3) \sim \frac{\log 2x \log 3x}{2 \log 2 \log 3}.$$

These bounds can be generalized by a geometric argument. Every number $n \in \Psi_y(x)$ can be written in the form $n = \prod_{p \leq y} p_i^{h_i}$, and we have $\log n = \sum_{p \leq y} h_i \log p_i$. By counting the points with non-negative integer coordinates bounded by an appropriate hyperplane in $\mathbb{R}^{\pi(y)}$, we can estimate $\psi(x, y)$ when y is not too large.

Theorem 6.10 (Ennola)**.** *Uniformly for* $2 \le y \le \sqrt{\log x \log\log x}$*,*

$$\psi(x,y) = \frac{1}{\pi(y)!}\prod_{p\le y}\left(\frac{\log x}{\log p}\right)\left(1+O\left(\frac{y^2}{\log x\log y}\right)\right).$$

A proof of this theorem may be found in [48] and also in [107, III.4].

Tight asymptotic formulae for $\psi(x,y)$ are difficult to obtain in a convenient form when $y \approx \log x$; there is a distinct phase change in the behavior of $\psi(x,y)$ in this region. The following proposition gives a convenient upper bound when $y = O\left(\log x\right)$.

Theorem 6.11. *If* $y = O\left(\log x\right)$ *then* $\psi(x,y) = O\left(x^\epsilon\right)$ *for any* $\epsilon > 0$*.*

Proof. Any y-smooth number $s \le x$ may be written as $s = 2^{h_1}3^{h_2}\cdots p_w^{h_w}$, where $p_w \le y$ is the wth prime and the h_i are non-negative integers, and we have

$$h_1\lg p_1 + h_2\lg p_2 + \cdots + h_w\lg p_w = \lg s \le \lg x.$$

Each s corresponds to a unique sequence of non-negative h_i which satisfy the inequality above. Fix $\epsilon > 0$ and choose a constant k so that $1/k < \epsilon$. There exists a least v such that $\lg p_v \ge k$. We assume $v < w$ since otherwise $p_w \le 2^k$ is bounded by a constant, as is w, and we have $\psi(x,y) = O\left((\lg x+1)^w\right) = O\left(x^\epsilon\right)$. For $v < i \le w$, the h_i satisfy

$$h_{v+1}+\cdots+h_w < h_{v+1}\lg(p_{v+1}-p_v)+\cdots+h_w\lg(p_w-p_v) < \frac{\lg s}{\lg p_v} \le \frac{\lg x}{k},$$

and for $i \le v$ we have $h_i \le \lg x$. Recalling that there are $\binom{a+b}{b}$ distinct ways to express a non-negative integer $n \le a$ as the ordered sum of b non-negative integers, we obtain

$$\psi(x,y) \le (\lg x+1)^v\binom{\left\lfloor\frac{\lg x}{k}\right\rfloor + w - v}{w-v} < (\lg x+1)^v 2^{\frac{\lg x}{k}+w-v} = (\lg x+1)^v x^{1/k}2^{w-v}.$$

Since $w = \pi(y) = O\left(y/\log y\right) = O\left(\log x/\log\log x\right)$, we have $2^{w-v} < e^w < x^{c/\log\log t}$ for some constant c, thus

$$\psi(x,y) = O\left((\log x)^v x^{1/k+c/\log\log x}\right) = O\left(x^\epsilon\right),$$

where $1/k < \epsilon$, and the constants v and c are independent of x. □

We now suppose $y = x^{1/u}$ with $u \ge 1$. For $u \le 2$, $\psi(x,y)$ can be readily computed.

Theorem 6.12. *Uniformly, for* $\sqrt{x} \le y \le x$,

$$\psi(x,y) = (1 - \log u)x + O\left(\frac{x}{\log x}\right)$$

where $u = \log x / \log y \ge 1$.

Proof. For $\sqrt{x} \le y \le x$, each $n \in [1,x] - \Psi_y(x)$ is divisible by exactly one prime in $(y,x]$.

$$\psi(x,y) \;=\; \lfloor x \rfloor - \sum_{y<p\le x} \left\lfloor \frac{x}{p} \right\rfloor \;=\; x\left(1 - \sum_{y<p\le x} \frac{1}{p}\right) + O\left(\pi(x)\right), \tag{6.17}$$

where the sums are over primes. We may evaluate the rightmost sum[3] by applying Theorem 6.5 to the function $f(t) = 1/t$ summed over primes to obtain

$$\sum_{y<p\le x} \frac{1}{p} = \int_y^x \frac{1}{t\log t}dt + \frac{\varepsilon(t)}{t}\Big|_y^x + \int_y^x \frac{\varepsilon(t)}{t^2}dt.$$

By the PNT, $\varepsilon(x) = O\left(x/\log^2 x\right)$ and the two rightmost terms are both $O\left(1/\log x\right)$. The first integral evaluates to $\log\log x - \log\log y = \log u$, where $u = \log x/\log y$. Substituting back into (6.17) and applying the PNT gives the desired result. □

Theorem 6.12 is a special case of Theorem 6.13 below. As with $\phi(x,y)$, one can estimate $\psi(x,y)$ inductively for values of $u > 2$. No closed form for $\psi(x,y)$ exists when $u > 2$ and we are led to define the function $\rho(u)$ satisfying the differential-difference equation

$$u\rho'(u) + \rho(u-1) = 0 \qquad (u > 1),$$

with $\rho(u) = 1$ for $0 \le u \le 1$. The function $\rho(u)$ was first investigated by Dickman [42] and bears his name. Dickman proved the existence and uniqueness of $\rho(u)$ and showed that $\psi(x,y) \sim \rho(u)x$ for $y = x^{1/u}$. The function $\rho(u)$ converges to 0 very rapidly as $u \to \infty$, roughly on the order of u^{-u}, and this behavior makes it difficult to obtain explicit bounds for $\psi(x,y)$. The function $\rho(u)$ can be numerically computed. Efficient algorithms for computing both $\omega(u)$ and $\rho(u)$ are presented in [72], and a table of explicit values for $\rho(u)$ can be found in [54]. It is also possible to numerically approximate $\psi(x,y)$ directly; an efficient algorithm

[3]The asymptotic formula $\sum_{p\le x} \frac{1}{p} \sim \log\log x$ is well known. The advantage illustrated by applying Theorem 6.5 is that it becomes trivial to derive such facts from the PNT even when they are not well known.

may be found in [106]. The theorem below is a more precise version of Dickman's result.

Theorem 6.13 (Hildebrand)**.** *Let $\epsilon > 0$. Uniformly for $x \geq 2$ and $\exp\left((\log\log x)^{5/3+\epsilon}\right) \leq y \leq x$,*

$$\psi(x, y) = \rho(u)x\left(1 + O\left(\frac{\log(u+1)}{\log y}\right)\right).$$

A proof can be found in [107, III.4]. We also have the following theorem due to Canfield, Erdös, and Pomerance [32].

Theorem 6.14 (Canfield et al.)**.** *Let $\epsilon > 0$. Uniformly for $1 \leq u < (1-\epsilon)\log x/\log\log x$,*

$$\psi(x, x^{1/u}) = xu^{-u+o(u)}.$$

For more information on smooth numbers, we refer the reader to the comprehensive work of Hildebrand [61, 62] and the survey by Granville [54].

6.4 Semismooth Numbers

Semismooth numbers are a refinement of smooth numbers that incorporate two bounds.

Definition 6.7. *A positive integer is* **semismooth** *with respect to y and z if all its prime factors are bounded by y and all but at most one are bounded by z. The function $\boldsymbol{\psi(x, y, z)}$ counts the positive integers up to x which are semismooth with respect to y and z.*

A number that is semismooth with respect to y and z is y-smooth, and is at most one prime factor away from being z-smooth. The function $\psi(x, y, z)$ arises in the analysis of "large prime" variants of integer-factorization algorithms. Typically, one is in interested in $\psi(x, x^s, x^r)$ for some $r < s$, most commonly, $r = 1/u$ and $s = 2/u$. Note that an integer which is semismooth with respect to $x^{2/u}$ and $x^{1/u}$ is equivalently an integer whose $x^{1/u}$-coarse part is at most $x^{2/u}$; in both cases it can have at most one prime factor in $(x^{1/u}, x^{2/u}]$ and none greater than $x^{2/u}$.

The function $G(r, s)$ estimates the probability that an integer in $[1, x]$ is semismooth with respect to x^s and x^r.

Definition 6.8. *For $0 < r < s < 1$, the* **semismooth probability function** *$G(r, s)$ is defined by*

$$G(r, s) = \lim_{x\to\infty} \psi(x, x^s, x^r)/x.$$

Note that the order of r and s is reversed. Bach and Peralta show in [11] that $G(r,s)$ always exists, and give formulae that may be used to estimate $G(r,s)$. These estimates are most often applied in situations where r and s are quite small, as little as $1/10$ or less. We are interested in $G(r,s)$ for larger values of r and s, say $r = 1/u \geq 1/3$ and $s = 2/u \geq 2/3$. The case $G(1/u, 2/u)$ is considered by Knuth and Trabb Pardo in [69], where they give tables for some specific values of u. For $u \in (2,3]$, estimating $G(1/u, 2/u)$ is particularly straight forward.

Theorem 6.15. *For $2 < u \leq 3$,*

$$G(1/u/2/u) \geq 1 - \left(\log\frac{u}{2} + \frac{1}{2}\log^2\frac{u}{2} + \log 2 \log\frac{u}{2} + S(u)\right),$$

where $S(u) \leq 0$ may be bounded by

$$S(u) \leq \sum_{k=0}^{n-1} \log\left(\frac{2n + (k+1)s}{2n+ks}\right)\log\left(\frac{n(s+1)-ks}{n(s+2)}\right),$$

for any n, with $s = u - 2$.

Proof. Since $u \leq 3$, no integer in $[1,x]$ contains three primes greater than $x^{1/u}$. We compute $G(1/u, 2/u)$ by counting the $x^{2/u}$-smooth numbers in $[1,x]$ which do not contain two primes in $(x^{1/u}, x^{2/u}]$. By Theorem 6.12, the number of $x^{2/u}$-smooth integers in $[1,x]$ is

$$\psi(x, x^{2/u}) \sim \left(1 - \log\frac{u}{2}\right)x. \tag{6.18}$$

We need to subtract the $x^{2/u}$-smooth integers with two prime factors in $(x^{1/u}, x^{2/u}]$:

$$\sum_{\substack{x^{1/u} < p \leq x^{2/u};\\ p \leq q \leq x/p}} \left\lfloor \frac{x}{pq} \right\rfloor. \tag{6.19}$$

Dropping the floor in the sum introduces an error of $O\left(x/\log x\right)$, since the number of terms in the sum is $\phi_2(x, x^{1/3}) = O\left(x/\log x\right)$ by Lemma 6.7. Dividing by x, we consider the sum

$$\sum_{\substack{x^{1/u} < p \leq x^{2/u};\\ p \leq q \leq x/p}} \frac{1}{pq} = \frac{1}{2}\sum_{\substack{x^{1/u} < p \leq x^{1/2};\\ x^{1/u} \leq q \leq x^{1/2}}} \frac{1}{pq} + \sum_{\substack{x^{1/u} < p \leq x^{1/2};\\ x^{1/2} < q \leq x/p}} \frac{1}{pq} + O\left(\frac{1}{\log x}\right), \tag{6.20}$$

where the error due to overcounting squares of primes is incorporated in the error term. We make use of the approximation[4]

$$\sum_{y<p\le x}\frac{1}{p}=\int_y^x\frac{1}{t\log t}dt+O\left(\frac{1}{\log x}\right)=\log\left(\frac{\log x}{\log y}\right)+O\left(\frac{1}{\log x}\right), \tag{6.21}$$

as in the proof of Theorem 6.12. The first term in the RHS of (6.20) becomes

$$\frac{1}{2}\sum_{x^{1/u}<x\le x^{1/2}}\frac{1}{p}\sum_{x^{1/u}<q\le x^{1/2}}\frac{1}{q}=\frac{1}{2}\log^2\frac{u}{2}+O\left(\frac{1}{\log x}\right). \tag{6.22}$$

The second term in the RHS of (6.20) becomes

$$\sum_{\substack{x^{1/u}<x\le x^{1/2};\\ x^{1/2}<q\le x/p}}\frac{1}{pq}=\sum_{x^{1/u}<x\le x^{1/2}}\frac{1}{p}\log\left(\frac{\log x/p}{\log x^{1/2}}\right)+O\left(\frac{1}{\log x}\right) \tag{6.23}$$

$$=\log 2\log\frac{u}{2}+\sum_{x^{1/u}<x\le x^{1/2}}\frac{1}{p}\log\left(1-\frac{\log p}{\log x}\right)+O\left(\frac{1}{\log x}\right). \tag{6.24}$$

Let $S(u)$ be the limit of the sum on the right as $x\to\infty$. If we divide 6.18 by x and take the limit as $x\to\infty$ we have $\lim_{x\to\infty}\psi(x,x^{2/u})/x=1-\log(u/2)$. Subtracting (6.22) and (6.24) gives the desired expression for $G(2/u,1/u)$.

We now bound $S(u)$ by partitioning the interval $(x^{1/u},x^{1/2})$ into n subintervals of the form $I_k=(x^{(2n+ks)/2nu},x^{(2n+(k+1)s)/2nu}]$, where $s=u-2$. We then have

$$\sum_{x^{1/u}<x\le x^{1/2}}\frac{1}{p}\log\left(1-\frac{\log p}{\log x}\right)\le\sum_{k=0}^{n-1}\sum_{p\in I_k}\frac{1}{p}\log\left(1-\frac{2n+ks}{nu}\right)$$

$$=\sum_{k=0}^{n-1}\log\left(\frac{2n+(k+1)s}{sn+ks}\right)\log\left(1-\frac{2n+ks}{nu}\right)+O\left(\frac{1}{\log x}\right).$$

As $x\to\infty$, the error term fits under the inequality. □

Theorem 6.15 can be used to generate a list of lower bounds for $G(1/u,2/u)$ with $2<u\le 3$ which are compared with lower bounds for $\rho(u)$ in Table 6.1. The values for $u=2.5$ and $u=3.0$ agree with those given by Knuth and Trabb Pardo in [69]. The values of $G(1/u,2/u)$ for u from 3.5 to 6.0 are taken from [69], and from Bach and Peralta [11] for $u>6$. The values of $\rho(u)$ are taken from Granville [54] who credits Bernstein. All numbers

[4]The error term can be made $O\left(1/\log^2 x\right)$, but we don't need to do so.

u	$G(1/u, 2/u)$	$\rho(u)$
2.1	0.9488	0.2604
2.2	0.8958	0.2203
2.3	0.8415	0.1857
2.4	0.7862	0.1559
2.5	0.7302	0.1303
2.6	0.6738	0.1082
2.7	0.6172	0.0894
2.8	0.5605	0.0733
2.9	0.5038	0.0598
3.0	0.4473	0.0486
3.5	0.2238	0.0162
4.0	0.0963	4.910e-03
4.5	0.0365	1.370e-03
5.0	0.0124	3.547e-04
6.0	1.092e-03	1.964e-05
8.0	3.662e-06	3.232e-08
10.0	5.382e-09	2.770e-11

Table 6.1: Asymptotic Lower Bounds on Semismooth and Smooth Probabilities

are rounded down and represent asymptotic lower bounds as $x \to \infty$.

6.5 Applications

We now apply the results above to prove a number of theorems relevant to the analysis of algorithms which depend on the distribution of coarse numbers. More specifically, we investigate the distribution of the coarse parts of numbers and develop tools to estimate sums over these coarse parts.

If we take the natural numbers in the interval $[1, x]$ and remove from each number all prime factors less than or equal to y, we may then ask for the average value of the remaining numbers. Equivalently, we are interested in the average order of $\kappa_y(n)$. When $x = 30$ and $y = 3$ we obtain the sequence:

$$1, 1, 1, 1, 5, 1, 7, 1, 1, 5, 11, 1, 13, 7, 5, 1, 17, 1, 19, 5, 7, 11, 23, 1, 25, 13, 1, 7, 29, 5,$$

with mean value ≈ 7.53. As x tends to infinity we find that the mean value of $\kappa_3(n)$ over $[1, x]$ is asymptotic to $x/4$, or precisely half the mean value of n. More generally, we have the following theorem:

Theorem 6.16. *Uniformly for* $1 \le y \le x$,

$$\frac{1}{x}\sum_{n=1}^{x}\kappa_y(n) = \frac{x}{2}\prod_{p\le y}\frac{p}{p+1} + O\left(2^{\pi(y)}\log x\right).$$

Proof. Every natural number $n \le x$ can be written as $n = \sigma_y(n)\kappa_y(n)$. Thus we have

$$\sum_{n=1}^{x}\kappa_y(n) = \sum_{s\in\Psi_y(x)}\left(\sum_{r\in\Phi_y(x/s)} r\right). \tag{6.25}$$

We estimate the inner sum by using $c(t) = \phi(t, y) = G(t) + \varepsilon(t)$ as a counting function for Φ_y, where $G(t) = t/\zeta_y(1)$ and $|\varepsilon(t)| \le 2^{\pi(y)}$, by Theorem 6.6. Applying Theorem 6.5 with $f(t) = t$,

$$\begin{aligned}\sum_{\substack{0<r\le x/s\\ r\in\Phi_y}} r &= \frac{1}{\zeta_y(1)}\int_0^{x/s} t\,dt + t\varepsilon(t)\Big|_0^{x/s} - \int_0^{x/s}\varepsilon(t)\,dt\\ &= \frac{x^2}{2\zeta_y(1)s^2} + 2^{\pi(y)}O\left(\frac{x}{s}\right).\end{aligned} \tag{6.26}$$

Substituting into (6.25) and dividing by x yields

$$\begin{aligned}\frac{1}{x}\sum_{n=1}^{x}\kappa_y(n) &= \frac{x}{2\zeta_y(1)}\sum_{s\in\Psi_y(x)}\frac{1}{s^2} \;+\; 2^{\pi(y)}\sum_{s\in\Psi_y(x)} O\left(\frac{1}{s}\right)\\ &= \frac{x}{2\zeta_y(1)}\left(\zeta_y(2) - O\left(\frac{1}{x}\right)\right) + 2^{\pi(y)}O(\log x)\\ &= \frac{x\zeta_y(2)}{2\zeta_y(1)} + O\left(2^{\pi(y)}\log x\right),\end{aligned} \tag{6.27}$$

where we have bounded the tail of $\zeta_y(2)$ by the tail of $\zeta(2) = O(1/x)$. Finally, we may express $\zeta_y(1)$ and $\zeta_y(2)$ as Euler products,

$$\zeta_y(2)/\zeta_y(1) = \left(\prod_{p\le y}\frac{1}{1-p^{-2}}\right)\left(\prod_{p\le y}\frac{1}{1-p^{-1}}\right)^{-1} = \prod_{p\le y}\frac{p}{p+1}, \tag{6.28}$$

to complete the proof. □

Corollary 6.17. *Let $0 < c \leq 1$. If $y = \Theta\left(\log^c x\right)$ then*

$$\frac{1}{x}\sum_{n=1}^{x} \kappa_y(n) \;\sim\; \frac{\zeta(2)x}{2e^{\gamma}\log\log x},$$

where $\zeta(2)/(2e^{\gamma}) \approx 0.4618$.

Proof. In the proof of Theorem 6.16 we may substitute $\zeta_y(1) \sim e^{\gamma}\log y \sim e^{\gamma}\log\log x$ using Mertens' theorem and note that $\zeta_y(2) \to \zeta(2) = \pi^2/6$ as $y \to \infty$. □

The application of many of the number-theoretic results of this chapter to the analysis of algorithms can be greatly simplified by a method of *asymptotic integration* for estimating the kind of integrals that frequently arise. Directly evaluating an integral like

$$\int_a^x \frac{\sqrt{t\log\log t}}{\log t}dt$$

may be difficult or impossible, but its asymptotic behavior is easily determined. The technique developed here, an extension of the basic concept described by Bach in [10], uses elementary analytic results, all of which may be found in Dieudonne's text [43]. We begin with the theorem which motivates the method [43, III.10.5].

Theorem 6.18 (Asymptotic Integration)**.** *Let f be a positive function continuously differentiable over $[a, \infty)$. Suppose $f'(x)/f(x) \sim s/x$ for some non-zero real $s > -1$, or $f'(x)/f(x) = o(1/x)$, in which case let $s = 0$. Then*

$$\int_a^x f(t)dt \;\sim\; \frac{xf(x)}{s+1},$$

which is unbounded as $x \to \infty$.

Proof. For $s \neq 0$, the hypothesis implies (by integration) that $\log f(x) \sim s\log x$. Thus $f(x) > x^{s-\epsilon}$ with $s - \epsilon > -1$ and the integral is unbounded. Integrating by parts yields

$$\int_a^x f(t)dt = tf(t)\Big|_a^x - \int_a^x tf'(t)dt \;\sim\; xf(x) - s\int_a^x f(t)dt, \tag{6.29}$$

and solving for $\int_a^x f(t)dt$ gives the desired result. When $s = 0$ the hypothesis implies $|\log f(x)| < \log x$ and thus $f(x) > x^{-\epsilon}$ for any $\epsilon > 0$ and the integral is again unbounded. In this case the first term of (6.29) dominates and the formula holds with $s = 0$. □

The case where $s < -1$ may also be analyzed, but we do not consider it here. As a simple example, note that $\mathrm{li}(x) \sim x/\log x$ is an immediate consequence of Theorem 6.18, and a second application of the theorem to $(\mathrm{li}(x) - x/\log x)$ yields the error term $O\left(x/\log^2 x\right)$.

As a matter of convenience, we adopt the following notation.

Definition 6.9. *For any real number s, $f \in [x^s]_a$ indicates that $f(x)$ is a positive function continuously differentiable over $[a, \infty)$ with $f'(x)/f(x) \sim s/x$ if $s \neq 0$, and $f'(x)/f(x) = o(1/x)$ if $s = 0$. When unspecified, a is assumed to be 1.*

We note some useful properties of functions in $[x^s]_a$ that justify the notation.

Theorem 6.19. *For $f \in [x^s]_a$ and $g \in [x^t]_a$ the following hold:*

1. *If $h(x) = f(x)g(x)$ then $h \in [x^{s+t}]_a$.*
2. *If $h(x) = \big(f(x)\big)^r$ for some real r then $h \in [x^{rs}]_a$.*
3. *If $h(x) = f(x^r)$ for some real r then $h \in [x^{rs}]_a$.*
4. *If $h(x) = \int_a^x f(t)dt$ and $s > -1$ then $h \in [x^{s+1}]_a$.*
5. *If $s < t$ then $f \prec g$.*

Proof. In each case h is clearly positive and continuously differentiable over $[a, \infty)$.

(1) We have $h'/h = f'/f + g'/g$. If $s + t \neq 0$ then we have $h'/h \sim (s+t)/x$ (this is true even when one of s or t is zero). If $s = -t$ then $h'/h = o(1/x)$, and in either case $h \in [x^{s+t}]_a$.

(2) Assume $r \neq 0$. Then $h'/h = r(f'/f)$, hence $h'/h \sim rs/x$ when $s \neq 0$. If $s = 0$ then $h'/h = o(1/x)$, and in either case $h \in [x^{rs}]_a$. If $r = 0$ the statement is trivially true.

(3) Note that $h'(x) = (f(x^r))' = f'(x^r)rx^{r-1}$. Using the substitution $y = x^r$ we obtain

$$\frac{h'(x)}{h(x)} = \left(\frac{f'(y)}{f(y)}\right) r y^{\frac{r-1}{r}} \sim \left(\frac{s}{y}\right) r y^{\frac{r-1}{r}} = \frac{rs}{y^{1/r}} = \frac{rs}{x}.$$

(4) By Theorem 6.18, $h(x) \sim xf(x)/(s+1)$. Thus $h'/h \sim (s+1)/x$ and $h \in [x^{r+1}]_a$.

(5) For $t > 0$ we have $\log g \sim t \log x$ which implies $\log g \geq (t - \epsilon)\log x$ for any $\epsilon > 0$ and suitably large x. If $s \neq 0$ then we similarly obtain $\log f \leq (s + \epsilon)\log x$. For a suitable choice of ϵ we have $f(x) \leq x^{s+\epsilon} \prec x^{t-\epsilon} \leq g(x)$ as $x \to \infty$. If $s = 0$ then we may bound $f(x)$ by x^ϵ and the same holds. If $s < t = 0$, we have $f(x) \leq x^{s+\epsilon} \prec x^{-\epsilon} \leq g(x)$ for suitable $\epsilon > 0$ and sufficiently large x. When $s < t < 0$ we instead show $1/g \prec 1/f$ by applying (2) with $r = -1$ to obtain the positive case already considered. □

We now apply asymptotic integration to prove a more convenient form of Theorem 6.5.

Theorem 6.20. *Let $c(x)$ be a non-negative increasing function over $[a_0, \infty)$. Suppose $c(x) \sim G(x)$, where $G(x) = \int_{a_0}^{x} g(t)dt$ for some $g \in [x^0]_{a_0}$, and let $f \in [x^s]_a$ for some $s > -1$ and $a \geq a_0$.*

$$\int_a^x f(t)dc(t) \sim \frac{f(x)G(x)}{s+1}.$$

Proof. Letting $c(x) = G(x) + \varepsilon(x)$, with $\varepsilon \prec G$, we have

$$\int_a^x f(t)dc(t) = \int_a^x f(t)dG(t) + \int_a^x f(t)d\varepsilon(t). \tag{6.30}$$

Since f is continuous and c and G are increasing, the integrals all exist. To evaluate $\int_a^x f(t)dG(t) = \int_a^x f(t)g(t)dt$, we note that $fg \in [x^s]_a$ and apply Theorem 6.18 to obtain

$$\int_a^x f(t)g(t)dt \sim \frac{xf(x)g(x)}{s+1} \sim \frac{f(x)G(x)}{s+1}, \tag{6.31}$$

where the second relation is given by applying Theorem 6.18 to $G(x) = \int_{a_0}^{x} g(t)dt$.

We complete the proof by showing that $\int_a^x f(t)d\varepsilon(t)$ is dominated by (6.31). Integrating by parts yields

$$\int_a^x f(t)d\varepsilon(t) = f(t)\varepsilon(t)\Big|_a^x - \int_a^x \varepsilon(t)df(t). \tag{6.32}$$

The first term is dominated by the RHS of (6.31), since $\varepsilon \prec G$. The second term is dominated by the LHS of (6.31), since if it diverges we have

$$\int_a^x \varepsilon(t)df(t) = \int_a^x \varepsilon(t)f'(t)dt \sim s\int_a^x \frac{\varepsilon(t)f(t)}{t}dt \prec s\int_a^x \frac{G(t)f(t)}{t}dt \sim s\int_a^x f(t)g(t)dt,$$

and otherwise it is bounded and (6.31) is not. □

Any counting function satisfies the requirements of $c(x)$ in the theorem above, thus we have the following corollary.

Corollary 6.21. *Let S be a set of natural numbers with counting function $c_S(x) \sim G(x)$, where $G(x) = \int_{a_0}^{x} g(t)dt$ for some $g \in [x^0]_{a_0}$, and let $f \in [x^s]_a$ for some $s > -1$ and $a \geq a_0$. Then*

$$\sum_{\substack{a<n\leq x \\ n\in S}} f(n) \sim \frac{f(x)G(x)}{s+1}.$$

As a simple example, the corollary implies $\sum_{p\le x} p^2 \sim x^3/(3\log x)$ when applied to the prime counting function. This is not unexpected, but it is nice to have a rigorous proof of the fact. As a less simple example, the distribution of numbers with exactly two prime factors is given by the counting function $\pi_2(x) \sim \frac{x\log\log x}{\log x}$ [58, Thm 437]. We may apply the corollary using $\pi_2(x) \sim G(x) = \int_3^x \frac{\log\log t}{\log t} dt$ to immediately obtain

$$\sum_{pq\le x} \sqrt{pq} \;\sim\; \frac{2x^{3/2}\log\log x}{3\log x}.$$

We are now ready for the main goal of this section: a proposition which gives approximate bounds on functions summed over the coarse parts of numbers.

Proposition 6.22. *Fix $u > 1$ and let $y = x^{1/u}$. If $f \in [x^s]$ for some $s > 0$, then*

$$\big(1 + o(1)\big)\frac{u\omega(u)f(x)}{(s+1)\log x} \;\le\; \frac{1}{x}\sum_{n=1}^{x} f\big(\kappa_y(n)\big) \;\le \big(1 + o(1)\big)\frac{u\omega(u)f(x)}{s\log x},$$

where $\omega(u)$ is the Buchstab function.

Proof. Let $a = x^{1-\epsilon}$ with $\epsilon < 1/u$. We group the terms by coarse part and split the sum at a:

$$\begin{aligned}\frac{1}{x}\sum_{n=1}^{x} f\big(\kappa_y(n)\big) &= \frac{1}{x}\sum_{r\in\Phi_y(x)} \psi\left(\frac{x}{r}, y\right) f(r) \\ &= \frac{1}{x}\sum_{\substack{r\in\Phi_y(x)\\ r\le a}} \psi\left(\frac{x}{r}, y\right) f(r) \;+\; \frac{1}{x}\sum_{\substack{r\in\Phi_y(x)\\ r> a}} \left\lfloor \frac{x}{r}\right\rfloor f(r). \end{aligned} \tag{6.33}$$

Note that $r > a > y$ implies $x/r < y$, hence every number up to x/r is y-smooth in the second sum. We bound the first sum by replacing $\psi(x/r, y)$ with x/r and applying Corollary 6.21 to the function $\frac{f(t)}{t} \in [x^{s-1}]$ using the trivial counting function $c(t) = t$:

$$\frac{1}{x}\sum_{\substack{r\in\Phi_y(x)\\ r\le a}} f(r)\psi\left(\frac{x}{r}, y\right) \le \sum_{\substack{r\in\Phi_y(x)\\ r\le a}} \frac{f(r)}{r} \;\sim\; \frac{f(a)}{s}. \tag{6.34}$$

By part 3 of Theorem 6.19, $f(a) = f(x^{1-\epsilon}) \in [x^{(1-\epsilon)s}]$, and since $f(x)/\log x \in [x^s]$, part 5 of the same Theorem implies $f(a) \prec f(x)/\log x$. Thus the first sum in (6.33) is $o\left(f(x)/\log x\right)$.

We now consider the second sum in (6.33). Let $\phi_y(t)$ denote $\phi(t, y)$ for fixed y. We then write $\phi_y(t) = u\omega(u)\mathrm{li}(t) + \varepsilon(t)$ for some $\varepsilon(t)$. Note that $u\omega(u)$ does not depend on t. We

evaluate the second sum as

$$\frac{1}{x}\sum_{\substack{r\in\Phi_y(x)\\ r>a}}\left\lfloor\frac{x}{r}\right\rfloor f(r) \le \sum_{\substack{r\in\Phi_y(x)\\ r>a}}\frac{f(r)}{r} = \int_a^x \frac{f(t)}{t}d\phi_y(t) \tag{6.35}$$

$$= u\omega(u)\int_a^x \frac{f(t)}{t\log t}dt + \int_a^x \frac{f(t)}{t}d\varepsilon(t). \tag{6.36}$$

We may estimate the first integral by asymptotic integration of $\frac{f(t)}{t\log t} \in [x^{s-1}]$ to obtain

$$u\omega(u)\int_a^x \frac{f(t)}{t\log t}dt \;\sim\; \frac{u\omega(u)f(x)}{s\log x}, \tag{6.37}$$

which represents the main term in the upper bound of the proposition. If we had replaced $\left\lfloor\frac{x}{r}\right\rfloor$ in (6.35) by 1 instead of $\frac{x}{r}$ we would have derived a lower bound, with s replaced by $s+1$. We now bound the second integral in (6.36).

Theorem 6.9 implies that for $a < t \le x$ we have $\phi_y(t) \sim u(t)\omega\big(u(t)\big)t/\log t$ where

$$u(t) = \frac{\log t}{\log y} = \frac{\big(1-\delta(t)\big)\log x}{\log y} = \big(1-\delta(t)\big)u, \tag{6.38}$$

and $\delta(t)$ satisfies $t = x^{1-\delta(t)}$. For $t \in (a,x]$ we have $\delta(t) \le \epsilon$. Since the choice of $\epsilon > 0$ was arbitrary, we may force $\delta(t)$ arbitrarily close to u as $a < t \le x$ all tend toward infinity. It follows that $\varepsilon(t) = o\left(t/\log t\right)$. Integrating by parts we have

$$\int_a^x \frac{f(t)}{t}d\varepsilon(t) = \frac{f(t)\varepsilon(t)}{t}\Big|_a^x - \int_a^x \varepsilon(t)d\left(\frac{f(t)}{t}\right). \tag{6.39}$$

It is readily verified that both terms are $o\left(f(x)/\log x\right)$ using $(f(t)/t)' \sim (s-1)f(t)/t^2$ (since $f \in [x^s]$) and applying asymptotic integration to the integral on the right. □

Recall that $\omega(u) \approx e^{-\gamma}$ and is never more than $e^{-\gamma} + 0.006$. Additional explicit bounds on $\omega(u)$ may be found in [35].

Neither of the bounds in Proposition 6.22 is tight. We expect, by analogy with

$$\frac{1}{x}\sum_{n=1}^x n^s\left\lfloor\frac{x}{n}\right\rfloor = \frac{1}{x}\sum_{n=1}^x\sum_{m=1}^{x/n} n^s \;\sim\; \frac{1}{x}\sum_{n=1}^x \frac{1}{s+1}\left(\frac{x}{n}\right)^{s+1} = \frac{x^s}{s+1}\sum_{n=1}^x\frac{1}{n^{s+1}} \;\sim\; \frac{x^s\zeta(s+1)}{s+1},$$

that the true asymptotic value in Proposition 6.22 may be

$$\frac{1}{x}\sum_{n=1}^{x} f\big(\kappa_y(n)\big) \;\sim\; \frac{u\omega(u)\zeta(s+1)f(x)}{(s+1)\log x}, \tag{6.40}$$

but we will not attempt to prove this here.

Chapter 7

Fast Order Algorithms

Chapter Abstract

We consider the problem of computing $|\alpha|$ given the prime factorization of an integer N for which $\alpha^N = 1_G$, and generalize this notion to include incomplete factorizations of N with w coprime factors. We present two new algorithms, one with complexity linear in $n = \lg N$, but not in $m = \lg |\alpha|$, for use when $n \gg m$, and one with complexity $O\left(n \log w / \log \log w\right)$. We also give a generic multi-exponentiation algorithm that is particularly well suited for use with fast order algorithms, and provide some performance comparisons.

Fast order algorithms are given additional information to facilitate the computation of $|\alpha|$. Such information is often available when the order of G, or some group containing G, is known, since this determines a multiple of $|\alpha|$. Matrix groups over finite fields and permutation groups are typical examples, as are the integers modulo p. Alternatively, a multiple of $|\alpha|$ may become known as a result of general order computations, as seen in previous chapters. Given sufficient information, it is possible to compute $|\alpha|$ quite efficiently. Indeed, some of the performance tests in Section 7.7 involve groups with more than $10^{10,000}$ elements, and the exponents generated by the multi-stage sieve may include millions of factors.

The efficiency of fast order computations is important in smaller groups as well; such computations lie at the heart of many generic algorithms. Computational group theory often addresses problems of group recognition and membership which may be efficiently solved with the aid of an order oracle. This may involve statistical methods which perform order computations on a large sample of group elements. For many well known groups, the set of prime factors that may divide the order of any element is small, and fast order

algorithms may be applied. See [5, 6, 63, 94, 49, 50] for examples.

There are other situations where fast order algorithms are applicable. The problem of finding a primitive root (a generator) in a cyclic group with known order may be solved with a fast order computation. In some cases, representation-specific information can be applied to assist a generic order computation. The algorithm of Cellar and Leedham-Green for determining the order of a matrix over a finite field is an example [33]. Their algorithm uses the matrix representation of α to determine an exponent of $|\alpha|$, then applies a fast order algorithm which uses this exponent to compute $|\alpha|$.

7.1 Factored Exponents and Pseudo-Order

The complexity of fast order algorithms depends on the specificity of the information available regarding $|\alpha|$. Typically we are given either a set containing the prime factors of $|\alpha|$, or a *factored exponent* of α, an integer N such that $\alpha^N = 1_G$, along with its prime factorization. The latter case may be viewed as a generalization of the former: if S contains the prime factors of $|\alpha|$, the implicit bound given by the size of identifiers implies that

$$N = \prod_{p \in S} p^{\lfloor \log_p 2^\ell \rfloor}$$

is a factored exponent of α. The complexity of fast order computations depends not only on the size of N, but, more critically, on its factorization. For N of a given size, the more factors N has, the more difficult computing $|\alpha|$ will be. This is the reverse of the situation with general order computations, where the prime order case is typically the most difficult. Here we will be particularly concerned with highly composite N.

It is possible that the complete factorization of a known exponent may not be available. For example $|G|$ may be known, but not the prime factorization of $|G|$; if $G = \mathbb{Z}_p^*$, we know that $|G| = p - 1$, but it may not be feasible to completely factor $p - 1$. Another example is a matrix group over a finite field. The order of the general linear group of dimension d over the finite field with $q = p^h$ elements is given by

$$\left|GL_d(\mathbb{F}_q)\right| = \prod_{i=0}^{d-1} \left(q^d - q^i\right).$$

The prime factors of $|G|$ for any subgroup $G \subseteq GL_d(\mathbb{F}_q)$ lie among p and the divisors of $p^n - 1$, for various n. Considerable effort has been expended to find factorizations of such numbers, and for specific values of p complete factorizations are known for many values of n [23]. Even when the complete factorization is unavailable, a partial factorization containing only a few coarse composites is usually known.

These examples motivate a useful generalization. A factorization of N into powers of integers which are (pair-wise) coprime may serve as a useful substitute for a complete prime factorization of N. Typically these coprime factors will either be primes or very coarse composites (numbers with no small prime factors), but for our purposes, coprimality is the essential criterion.

Definition 7.1. *An integer N such that $\alpha^N = 1_G$, together with a factorization $N = n_1^{h_1} \cdots n_w^{h_w}$ satisfying $n_i \perp n_j$ for $i \neq j$, is called a* **factored exponent** *of α. An integer M of the form $M = n_1^{e_1} \cdots n_w^{e_w}$ which divides N is called a* **compatible divisor** *of N. The* **pseudo-order** *of α relative to the factored exponent N is the least compatible divisor M such that $\alpha^M = 1_G$, and is denoted $|\boldsymbol{\alpha}|_{\boldsymbol{N}}$.*

When we use the notation $|\alpha|_N$, a specific factorization of N is assumed to have been given. The set of compatible divisors of N and the set of exponents of α are both closed under the gcd operation. The integer $|\alpha|_N$ is the unique minimal member of both sets.

Definition 7.2. *A* **fast order algorithm** *$\mathcal{A}(\alpha, N)$ is a generic group algorithm which computes $|\alpha|_N$, given a factored exponent N of α.*

For a more formal definition, we may apply the model presented in Chapter 1.[1] We define a generic group relation $R(\vec{x}, \vec{y})$, where $\vec{x}$ encodes α, the n_i, and the h_i, and $\vec{y}$ represents the integer $|\alpha|_N$. The relation $R(\vec{x}, \vec{y})$ holds precisely when $|\alpha|$ divides N, the n_i are pair-wise coprime, and $y = |\alpha|_N$. We then define a fast order algorithm to be a generic algorithm $\mathcal{A}(\vec{x})$ which computes $R(\vec{x}, \vec{y})$ according to the definitions given in Section 1.2. For convenience we may write $\mathcal{A}(\alpha, N)$ instead of $\mathcal{A}(\vec{x})$, with the factorization of $N = n_i^{h_i} \cdots n_w^{h_w}$ implicit.

When the prime factorization of N is given, $|\alpha|_N$ necessarily coincides with $|\alpha|$. If $|\alpha|$ is a compatible divisor of N then we also have $|\alpha|_N = |\alpha|$, but in general they may differ. If the prime factorization of N is not available, computing $|\alpha|_N$ is often just as good as computing $|\alpha|$. As pointed out by Celler and Leedham-Green [33], the pseudo-order of α is

[1] This degree of formality is not generally required, but it is important to know that it can be done.

sufficient to determine whether $|\alpha|$ divides any integer M. If $|\alpha|$ divides M but $|\alpha|_N$ does not, $\gcd(M, |\alpha|_N)$ must split one of the given factors of N, and $|\alpha|_N$ can then be recomputed using a refined factorization of N. Babai and Beals show that a wide range of problems for matrix groups can be solved by generic algorithms using pseudo-orders, provided the factors of N come from an appropriate set of "pretend-primes" [5]. More generally, suppose the given factorization of N consists entirely of primes and composites which have resisted all attempts of factorization. If it is known that $|G|$ contains an element of order n_i for each given factor n_i of N, then for a random $\alpha \in G$ it is highly probable that $|\alpha|_N = |\alpha|$.[2]

7.2 A Linear-Time Algorithm for Large Exponents

We begin our consideration of fast order computations with a specialized algorithm that is particularly relevant to the general order algorithms presented in this thesis. When the size of N, the factored exponent, is much larger than $\lg |\alpha|$, it is possible to compute $|\alpha|$ in time linear in $n = \lg N$. This is the algorithm of choice for use with the primorial-steps and multi-stage sieve algorithms when the bound L is large. In this case, the factored exponent is the product of all the primes (or prime powers) less than L, which may potentially involve millions of factors, even though $|\alpha|$ likely contains no more than ten or twenty of these.

The main task of this algorithm is to identify the minimal subset of the factors of $N = N_1 \dots N_w$ whose product M is an exponent of α. If the N_i are primes, M is the order of α, but in general the N_i will be prime powers (or pseudo-prime powers) and M will be a new factored exponent of α which is much smaller than N. Any of the fast order algorithms presented in the following sections may then be used to compute $|\alpha|$ given M.

We first give a simple version that stores $O(w)$ group elements, then discuss how to make it to more space efficient.

Algorithm 7.1. *Given a factored exponent $N = q_1 \dots q_w$ of $\alpha \in G$, the algorithm below computes an exponent M of α, whose factors are a minimal subset of $\{q_1, \dots, q_w\}$.*

1. **Initialize:** If $\alpha = 1_G$, return 1, otherwise set $\beta_1 \leftarrow \alpha$, $i \leftarrow 1$, and $M \leftarrow 1$. Let $q_0 = 1$.
2. **Exponentiate:** While $\beta_i \neq 1_G$, compute $\beta_{i+1} = \beta_i^{q_i}$ and set $i \leftarrow i + 1$.
3. **Prepare:** Set $M \leftarrow q_{i-1}$ and $t \leftarrow i - 1$.

[2] This is clearly true when G is abelian, but would need to be precisely stated and proven in general.

4. **Find Factor:** If $t = 1$, return M. Otherwise do a binary search on $j \in [1, t-1]$ to find the greatest j s.t. $\beta_j^M \neq 1_G$. If no such j exists, return M.

5. **Include Factor:** Set $M \leftarrow q_j M$ and $t \leftarrow j$, then go to step 4.

The correctness of the algorithm may be shown inductively. After step 3, M is an exponent of β_t, and this remains true each time step 5 is executed. When the algorithm returns, $\beta_t = \beta_1 = \alpha$, thus M is an exponent of α. Each factor q_j in M contains a prime divisor of the order of some element of $\langle\alpha\rangle$, which must divide the order of α. The q_j are coprime, so M contains precisely the minimal subset of $\{q_1, \ldots, q_w\}$ required.

To reduce the space, we modify step 1 to only save β_1 and every kth value of β_i. In step 4 we do a binary search on the saved values of β_i corresponding to $i \in [1, t-1]$, then recompute up to $k-1$ of the discarded β_i values to find the desired $j \in [1, t-1]$ in step 4.

Proposition 7.1. *Let $m = \lg M$ and $n = \lg N$. Let v and w be the number of factors q_i in M and N respectively. Let b be the largest value of $\lg q_i$. Then if k is the parameter described above, Algorithm 7.1 uses at most*

$$\big(1 + o(1)\big)\big(n + v(m + kb)\lg(w/k)\big)$$

group operations, and stores $w/k + O(1)$ group elements.

Proof. Using a fast exponentiation algorithm, the exponentiations in step 2 require a total of $\big(1 + o(1)\big)n$ group operations. Step 4 is executed v times, each binary search involves $\big(1 + o(1)\big)\lg(w/k)$ exponentiations of the exponent M with size at most m, and less than k exponentiations of size b are required to recompute the β_i in the gaps. The storage bound is immediate. □

When $k = 1$, we can drop b from the time bound to obtain

$$\big(1 + o(1)\big)(n + mv \lg w). \tag{7.1}$$

Proposition 7.1 can be used to compute the exact complexity of Algorithm 7.1 in specific instances, but we are most interested in the case where b and m (and necessarily v) are logarithmic or nearly logarithmic in n. This occurs, for example, when N is the product of all the primes (or prime powers) up to some $L \approx (|\alpha|)^c$ for some $c > 0$. In this case we

can choose k to make the space required quite small without significantly impacting the running time, by taking $k = n/(bv \log n)$, for example.

Corollary 7.2. *If m and b are polylogarithmic in n, then for appropriate k, Algorithm 7.1 uses $\big(1 + o(1)\big)n$ group operations and polylogarithmic space.*

7.3 The Classical Algorithm

We now consider more conventional fast order algorithms, beginning with the classical solution.[3]

Algorithm 7.2 (Classical Algorithm)**.** *Given a factored exponent $N = n_1^{h_1} \cdots n_w^{h_w} = N_1 \cdots N_w$ of α, compute $|\alpha|_N$ as follows:*

1. Compute $\alpha_i \leftarrow \alpha^{N/N_i}$ for i from 1 to w.
2. For each i, find the least $q_i = n_i^{e_i}$ such that $\alpha_i^{q_i} = 1_G$ by successively computing $\alpha_i \leftarrow \alpha_i^{n_i}$ until $\alpha_i = 1_G$.
3. Return $\prod_{i=1}^{w} q_i$.

The algorithm above is easily seen to be correct, but it is useful to examine why. We may regard the algorithm as a search through the set of compatible divisors of N for the value $x = |\alpha|_N$. For every compatible divisor $y \neq x$, the algorithm either finds $\alpha^z = 1_G$ for some z that is a multiple of x but not y, or it finds $\alpha^z \neq 1_G$ for some z that is a multiple of y but not x. Given that N is a factored exponent of α, it must then be the case that $x = |\alpha|_N$, even though the algorithm does not necessarily compute α^x. This is the main task of fast order algorithms: they must distinguish $|\alpha|_N$ among the compatible divisors of N.

While perhaps not immediately apparent, the complexity of Algorithm 7.2 is essentially determined by step 1. Each of the exponentiations in step 2 involving the exponent n_i can be performed using no more than $2\lfloor \lg n_i \rfloor$ group operations. Each n_i is used at most h_i times, so the total number of group operations performed in step 2 is bounded by $2 \lg N$. This value is linear in $n = \lg N$ and does not depend on w. The constant 2 may be brought arbitrarily close to 1 by using any of several generic algorithms for fast exponentiation.

[3]The material contained in the remainder of this chapter also appears in the author's paper [105].

All but possibly one of the w exponentiations in step 1 involve an exponent of size $\Theta(n)$. If these exponentiations are performed independently, $\Theta(nw)$ group operations will be required. Note that w may be as large as $n/\log n$, as occurs when N is the product of the first w primes, so in general the complexity of step 1 will be $\Theta\left(n^2/\log n\right)$ in a naive implementation. We will see how to improve the performance of step 1 significantly, but in every case it will dominate the complexity.

7.4 Improvements

We now focus our attention on the specific problem of computing $\left[\alpha^{N/N_1}, \ldots, \alpha^{N/N_w}\right]$, where $N = N_1 \cdots N_w$. We don't require that the N_i be coprime, but in step 1 of the classical algorithm they always will be. As noted above, a naive implementation uses $\Theta(wn)$ group operations, where $w = \Theta\left(n/\log n\right)$ in the worst case. If the prime factorization of a random integer $N \in [1, M]$ is given, we have $w = \Theta\left(\log\log N\right) = \Theta\left(\log n\right)$ [58, Thm. 430-431].[4] Thus the classical algorithm uses $O\left(n^2/\log n\right)$ group operations in general, and $\Theta\left(n \log n\right)$ group operations for a random N (the prime factorization is assumed for random N).

The classical algorithm may be improved by noting that the exponentiations performed in step 1 all involve the base α. There are several generic fixed-base multi-exponentiation algorithms [73, 51, 13] which can compute w exponentiations of a common base using $O\left(n + w\frac{n}{\lg n}\right)$ group operations (including any precomputation). By applying any of these algorithms, the running time of the classical algorithm can be improved to $O\left(n^2/\lg^2 n\right)$ in general, and $\Theta(n)$ for a random value of N.

Remark 7.1. *While convenient, the notion of a "random" value of N should be used carefully in the context of fast order algorithms. If $N = |G|$, where G is a random group chosen uniformly among the non-isomorphic groups with order at most M, then N is most definitely not uniformly distributed over* $[1, M]$. *Indeed, it is conjectured [86] that the order of almost all groups is a power of 2; among the nearly 50 billion non-isomorphic groups with order less than 2000, more than 98% have order* 2^{10}, *and most of the remaining groups have order* $3 * 2^9$ *[15]. In practice, fast order algorithms are typically used in situations where G is known to have some particular structure, which will often mean that N has many small factors. The order of a random subgroup of the symmetric group* S_n, *for example, has many more distinct factors than a random integer in* $[1, n!]$.

[4] Both the average and median value of w are $\Theta\left(\log n\right)$.

Celler and Leedham-Green have an efficient method for computing the order of an invertible matrix over a finite field [33]. Their algorithm takes advantage of the specific structure of the general linear group $GL_d(\mathbb{F}_q)$ to find an exponent of α. At the core of their method is a generic order algorithm which may be used with any group. They use the same basic structure as the classical algorithm, but replace step 1 with the recursive procedure described below. In anticipation of things to come, their "divide-and-conquer" algorithm has been rewritten as a "clumping" algorithm.

Algorithm 7.3 (Celler, Leedham-Green)**.** *Given $\alpha \in G$ and positive integers $[N_1, \ldots, N_w]$, $\mathcal{A}(\alpha, [N_1, \ldots, N_w])$ recursively computes $[\alpha_1, \ldots, \alpha_w]$, where $\alpha_i = \alpha^{N/N_i}$ and $N = \prod_{i=1}^{w} N_i$.*

1. **Base**: If $w = 1$, set $\alpha_1 \leftarrow \alpha$ and return.
2. **Partition**: Put $N_1, \ldots, N_w$ into $v = \lceil \frac{w}{2} \rceil$ blocks of size at most 2.
   Let $M_1, \ldots, M_v$ be the block products.
3. **Recurse**: Set $[\beta_1, \ldots, \beta_v] \leftarrow \mathcal{A}(\alpha, [M_1, \ldots, M_v])$.
4. **Exponentiate**: Set $\alpha_i \leftarrow \beta_j^{M_j/N_i}$, where block j contains N_i.

Example 7.3.1. *$\mathcal{A}(\alpha, [2,3,5,7])$ might compute:*

$$\beta_1 = \alpha^{35}, \quad \beta_2 = \alpha^{6}, \quad \alpha_1 = (\beta_1)^3, \quad \alpha_2 = (\beta_1)^2, \quad \alpha_3 = (\beta_2)^7, \quad \alpha_4 = (\beta_2)^5.$$

A naive implementation would compute: $\alpha_1 = \alpha^{105}$, $\alpha_2 = \alpha^{70}$, $\alpha_3 = \alpha^{42}$, $\alpha_4 = \alpha^{30}$.

The partition chosen by the algorithm does not materially impact the running time. The depth of the recursion is $\Theta(\lg w)$, and at each level the product of all the exponents used in step 4 is less than N. Thus $\Theta(n)$ group operations are used at each level ($n = \lg N$), and $\Theta(n \lg w)$ in all. If we replace step 1 of the classical algorithm with Algorithm 7.3, we obtain a fast order algorithm that uses $O(n \lg n)$ group operations in general, and $\Theta(n \lg \lg n)$ for a random N.

The latter bound is, perhaps surprisingly, worse than the comparable bound for the classical algorithm when fixed-base multi-exponentiation is used. Fixed-based methods can be applied to Algorithm 7.3, but since just two exponentiations are performed for each base, the number of group operations is only reduced by a constant factor.

7.5 The Snowball Algorithm

To develop an algorithm which improves both Algorithm 7.3 and the classical algorithm with multi-exponentiation, we need a recursive structure which maximizes the impact of multi-exponentiation. In essence, any time we exponentiate a group element α using a b-bit exponent, we can perform $O\left(\lg b\right)$ additional exponentiations of α for free. In the classical algorithm, too many exponentiations are done with the same base, while algorithm 7.3 performs too few.

The ideal situation would be to always use $k = \Theta\left(\lg b\right)$ exponentiations of a common base. More precisely, given a constant $c \approx 2$, we wish to choose the largest k such that

$$cost(k, b) \leq cb, \tag{7.2}$$

where $cost(k, b)$ is the number of group operations required to perform k exponentiations of a common base, using exponents containing at most b bits. For a fixed c, we may determine k from b by defining

$$f_c(b) = \max\{k : cost(k, b) \leq cb\}. \tag{7.3}$$

For most multi-exponentiation algorithms, $f_c(n) \sim \lg n$ for an appropriate value of c. The algorithm presented in Section 7.6 has $f_c(n) = \lg n + O(1)$, for all $c > 2$.

Given integers $N_1, \ldots, N_w$, if we pack k integers into a block, say $M = N_1 \cdots N_k$, rather than just two as in Algorithm 7.3, we will exponentiate α using the exponents $M/N_1, \ldots, M/N_k$. These exponents are all bounded by M, so we may take $b = \lceil \lg M \rceil$ in the cost function above. The actual sizes of the exponents will be smaller than b, but this does not materially impact the implementation or analysis of the algorithm. Arranging the N_i in increasing order makes it convenient to pack them into contiguous blocks that produce exponents of roughly the same size.

We want to maximize the value of k, so we will pack exponents into *optimal blocks* whenever possible. An optimal block B is a contiguous block of exponents which cannot be extended without exceeding the cost ratio. That is, every contiguous block containing B as a proper prefix has $cost(k, b) > cb$, where b is the total number of bits in the block. To make this determination for blocks at the end of the input array, we assume a dummy input equal to the last input. If this dummy input can be added to the block without exceeding the cost

ratio, then the block is not considered optimal.[5] Any inputs which cannot be conveniently packed into optimal blocks are placed into singleton blocks, effectively deferring their processing to a later stage of the recursion. As the recursion deepens, the size of the inputs grows, making it possible to use larger and larger values for k. These ideas are captured in the "snowball" algorithm:

Algorithm 7.4 (Snowball)**.** *Let c be a constant and let $cost(k, b)$ be the cost function of Algorithm 7.5. Given $\alpha \in G$, and integers $N_1, \ldots, N_w$ with $N_i > 1$, algorithm $\mathcal{A}\big(\alpha, [N_1, \ldots, N_w]\big)$ recursively computes $[\alpha_1, \ldots, \alpha_w]$, where $\alpha_i = \alpha^{N/N_i}$, $N = \prod_{i=1}^{w} N_i$.*

1. **Base**: If $w = 1$, set $\alpha_1 \leftarrow \alpha$ and return.
2. **Analyze**: Let $N_1 \leq \ldots \leq N_w$, and maximize $b = \lg(N_1 \cdots N_k)$ such that $cost(k, b) \leq cb$. Let N_{max} be the largest N_i such that $\lg N_i < b$.
3. **Partition**: Put $N_1, \ldots, N_k$ in a block, construct optimal blocks using $N_{k+1}, \ldots, N_{max}$, and place unused N_i into singleton blocks. Let $M_1, \ldots, M_v$ be the block products.
4. **Recurse**: Compute $[\beta_1, \ldots, \beta_v] \leftarrow \mathcal{A}\big(\alpha, [M_1, \ldots, M_v]\big)$.
5. **Exponentiate**: Set $\alpha_i \leftarrow \beta_j^{M_j/N_i}$ for N_i in block j, using Algorithm 7.5 with base β_j.

When implementing the algorithm above, it is important to note that in step 2 it is possible to have $a = \lg(N_1 \cdots N_j)$ and $b = \lg(N_1 \cdots N_k)$ with $cost(j, a) > ca$ while $cost(k, b) \leq cb$, even though $j < k$. The process of maximizing b must start with $b = w$ and then reduce b. The same comment applies when constructing optimal blocks: the largest possible block must be considered first. The partition constructed in step 3 will either be a single block containing all the exponents (when $k = w$), or it will consist of one or more optimal blocks and singleton blocks. The singleton blocks will include all $N_i > N_{max}$ and possibly some $N_i \leq N_{max}$ which remain because constructing an optimal block requires going past N_{max}.

Example 7.4.1. *On inputs $\alpha \in G$ and $[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31]$, Algorithm 7.4 might initially determine $k = 3$, $N_{max} = 29$, and partition the N_i into products:*

$$M_1 = 2 \cdot 3 \cdot 5, \quad M_2 = 7 \cdot 11 \cdot 13, \quad M_3 = 17 \cdot 19 \cdot 23 \cdot 29, \quad M_4 = 31.$$

[5] This is a minor optimization; omitting this check and simply considering end blocks optimal does not significantly impact the performance of the algorithm.

The next recursive call sets $k = 4$ and clumps all the N_i into a single block with product N, reaching the base case. The algorithm then computes:

$$\beta_1 = \alpha^{N/M_1}, \quad \beta_2 = \alpha^{N/M_2}, \quad \beta_3 = \alpha^{N/M_3}, \quad \beta_4 = \alpha^{N/M_4},$$

using a single multi-exponentiation with base α. One additional multi-exponentiation operation is performed for each of the β_i to compute the eleven required output values:

$$\alpha_1 = (\beta_1)^{3\cdot5},\ \alpha_2 = (\beta_1)^{2\cdot5},\ \alpha_3 = (\beta_1)^{2\cdot3},\ \alpha_4 = (\beta_2)^{11\cdot13},\ \alpha_5 = (\beta_2)^{7\cdot13},\ \alpha_6 = (\beta_2)^{7\cdot11},$$

$$\alpha_7 = (\beta_3)^{19\cdot23\cdot29},\ \alpha_8 = (\beta_3)^{17\cdot23\cdot29},\ \alpha_9 = (\beta_3)^{17\cdot19\cdot29},\ \alpha_{10} = (\beta_3)^{17\cdot19\cdot23},\ \alpha_{11} = \beta_4.$$

Before analyzing the complexity of the snowball algorithm, there are a few details worth noting. All the group operations occur in step 5, and we need only consider the non-singleton blocks, since if block j is a singleton block, then $M_j/N_i = 1$ and the exponentiation is trivial. With the possible exception of the very last block constructed, all the non-singleton blocks are optimal blocks containing $f_c(b)$ inputs, where b is the total number of bits in the block. Note that the algorithm would be effectively unchanged if only one non-singleton block were constructed in each recursive call.

Proposition 7.3. *Let c be a constant, and let $f_c(b) = \max\{k : cost(k, b) \leq cb\}$ satisfy $f_c(b) \sim \lg b$ and $f_c(b) \geq 2$ for all b. Let $n = \lg N$, where $N = N_1 \cdots N_w$.*

(1) Algorithm 7.4 uses $O\left(n \frac{\lg w}{\lg \lg w}\right)$ group operations and $O(w)$ group storage.

(2) Algorithm 7.4 uses $O(n)$ group operations when w is polylogarithmic in n.

Proof. We assume the algorithm forms just one non-singleton block in each recursive call and that the input array is always in increasing order. We claim we may also assume the N_i are all approximately the same size and thus $n/\lg N_1 = \Theta(w)$. Suppose not, and consider the impact of increasing $\lg N_i$ by δ while decreasing $\lg N_j$ by δ and holding $\lg N$ and w fixed. We may choose i, j, and δ so that the sorted order of the inputs is unchanged. We then have $i < j$, and note that the input N_i will get packed into a non-singleton block at least as many times as N_j. Since the bound on the cost is linear in the size of the exponents, this change can only increase the total cost bound. Thus the cost is maximized when the N_i are approximately the same size, which we now assume to be the case.

To analyze the algorithm, we divide the recursive calls of the algorithm into rounds. We begin by recording the values $E_1 = N_1$, $k_1 = k$, and $F_1 = M_1$ computed in the first recursive call, and start the second round when the smallest input N_1 is greater than or equal to F_1, at which point we record the values $E_2 = N_1$, $k_2 = k$, and $F_2 = M_1$. We proceed in this fashion until the algorithm terminates in round t. Each call in a given round, except possibly the last, forms a non-singleton block out of distinct inputs present at the start of the round. It follows that the total size of all the non-singleton blocks formed during a round is at most $2n$ and the number of group operations is no more than $2cn$ (in fact, close to cn). To complete the proof, we bound the number of rounds, t.

Note that $F_i \geq E_i^{k_i} \geq F_{i-1}^{k_i}$, since F_i is composed of k factors, all $\geq E_i$. At the start of round i we must have $E_i \geq F_{i-1}$, by definition. Thus we have

$$N \geq F_t \geq F_{t-1}^{k_t} \geq F_{t-2}^{k_{t-1}k_t} \geq \cdots \geq F_1^{k_2 \cdots k_t} \geq E_1^{k_1 \cdots k_t} = N_1^{k_1 \cdots k_t}, \tag{7.4}$$

which implies

$$k_1 k_2 \cdots k_t \leq n / \lg N_1 = O(w), \tag{7.5}$$

since we assumed $n/\lg N_1 = \Theta(w)$ above. This proves (2), for if w is polylogarithmic in n we have $w = O\left(\lg^d n\right)$ for some constant d and $k_1 = \Theta\left(\lg n\right)$, implying $t = O(1)$. To prove (1), we note that $k_i = f_c(\lg F_i) \geq f_c(k_i \lg E_i)$, and therefore

$$k_{i+1} \geq f_c(k_{i+1} \lg E_{i+1}) \geq f_c(k_{i+1} \lg F_i) \;\sim\; f_c(\lg F_i) + \lg k_{i+1} = k_i + \lg_{i+1}, \tag{7.6}$$

since $f_c(b) \sim \lg b$. The value $\lg F_i$ grows as i increases, hence so does $f_c(\lg F_i)$, and (7.6) implies that there is a constant s for which the sequence $k_s, k_{s+1}, \ldots k_t$ is strictly increasing. It follows from (7.5) that $(t - s + 1)! \leq k_s \cdots k_t = O(w)$, and thus $t = O\left(\frac{\lg w}{\lg \lg w}\right)$ as desired. The time bound in (1) follows.

To prove the space bound, we forget our assumption that only one non-singleton block is constructed in each call and note that, as written, the algorithm will complete each round within two recursive calls. The top level uses $O(n)$ group storage and the space required decreases by more than a constant factor every two calls, thus the total storage within the recursion is $O(n)$, and none of the calls to Algorithm 7.5 exceed this bound. □

The constant factors contained in the bounds above are small; the number of group

operations is close to $cn\frac{\lg w}{\lg\lg w}$ in practice. With the algorithm presented in the next section, c is just slightly larger than 2. As noted earlier, the size of the exponents used in step 5 of the Algorithm 7.4 is less than the value b used in the cost function, since the exponent M_j/N_i has one factor removed. As a result, the effective constant is less than 2 in practice.

7.6 A Fast Algorithm for Multi-Exponentiation

The exponentiations performed by the snowball algorithm typically involve small-to-moderate values of b and k, and in general $k \approx \lg b$. While many algorithms have good asymptotic performance as either b or k tend to infinity, few are ideal when b and k are small or $k/\lg b$ is constant.

The multi-exponentiation algorithm below is based on the transpose (dual) of an algorithm originally due to Straus [104] for computing a monomial $\alpha_1^{e_1}\cdots\alpha_k^{e_k}$ of multiple group elements (see also [76] or [73, Algorithm 14.88]). The transpose computes multiple powers of a single group element (see [80] or [13] on the transposition of exponentiation algorithms). Rather than transposing the operations of Straus' algorithm, we give a direct implementation.

Algorithm 7.5. *Given $\alpha \in G$, positive integers $E_1, ..., E_k$ with $\lceil \lg E_i \rceil \leq b$, and a positive integer s, the following algorithm computes $[\alpha^{E_1}, \ldots, \alpha^{E_k}]$:*

1. Let $d = \lceil \frac{b}{s} \rceil$ and write $E_i = \sum_{j=0}^{d-1} E_{ij} 2^{sj}$ as a d-digit number in base 2^s.
   Let $v(j) = [E_{1j}, E_{2j}, \ldots, E_{kj}]$, and for all vectors $x \in \{0, \ldots, 2^s - 1\}^k$, set $\beta_x \leftarrow 1_G$.
2. For j from 0 to $d-1$, compute $\alpha^{2^{sj}}$ and if $v(j)$ is non-zero set $\beta_{v(j)} \leftarrow \beta_{v(j)}\alpha^{2^{sj}}$.
3. For m from $k(2^s - 1)$ down to 2, for each z with weight m, if $\beta_z \neq 1_G$ then choose non-zero x and y such that $x + y = z$ and set $\beta_x \leftarrow \beta_x\beta_z$ and $\beta_y \leftarrow \beta_y\beta_z$.
4. Return $[\beta_{e(1)}, \ldots, \beta_{e(k)}]$, where $e(i)$ denotes the ith unit vector.

The weight of a vector is defined to be the sum of its digits, thus the vectors x and y chosen in step 3 will necessarily have lower weight than z. It is preferable to choose x and y so that β_x and β_y are non-trivial (not 1_G), but any choice will work. It is straight-forward to prove that Algorithm 7.5 is correct (see Proposition 7.4), but it is perhaps best understood by example.

Example 7.5.1. *For inputs 795 and 582, $k = 2$ and $b = 10$, and let $s = 2$ so that $d = \lceil \frac{b}{s} \rceil = 5$. Writing the exponents in base $2^s = 4$, we have*

$$E_1 = \mathbf{3\,0\,1\,2\,3};$$
$$E_2 = \mathbf{2\,1\,0\,1\,2}.$$

The vectors $v(j)$ can now be read down the columns corresponding to the powers 4^j, with 4^0 on the right. The $\beta_{v(j)}$ computed in step 2 may be written with base-4 exponents as:

$$\beta_{32} = \alpha^{10001_4}, \quad \beta_{21} = \alpha^{00010_4}, \quad \beta_{10} = \alpha^{00100_4}, \quad \beta_{01} = \alpha^{01000_4}.$$

Step 2 squares α eight times and performs one non-trivial multiplication involving β_{32}. Step 3 computes:

$$\beta_{21} = \beta_{21}\beta_{32} = \alpha^{10011_4}, \qquad \beta_{11} = \beta_{11}\beta_{32} = \alpha^{10001_4},$$
$$\beta_{11} = \beta_{11}\beta_{21} = \alpha^{20012_4}, \qquad \beta_{10} = \beta_{10}\beta_{21} = \alpha^{10111_4},$$
$$\beta_{10} = \beta_{10}\beta_{11} = \alpha^{30123_4}, \qquad \beta_{01} = \beta_{01}\beta_{11} = \alpha^{21012_4},$$

using five non-trivial multiplications (the operation $\beta_{11}\beta_{32}$ is trivial). The output is then $[\beta_{10}, \beta_{01}]$. Written in decimal, the complete addition chain of length fourteen is

$$1, 2, 4, 8, 16, 32, 64, 128, 256, 257, 261, 277, 518, \mathbf{582}, \mathbf{795}.$$

For comparison, the standard binary exponentiation algorithm would use twenty-six group operations in the example above. If we had set $s = 1$, fifteen group operations would have been used.[6]

For larger values of k, unless b is very large we will use $s = 1$ and the vectors will be binary strings. This case is the transpose of what has been referred to as "Shamir's trick" ([47], see also [97, Ex. 3.28]), which we illustrate with a slightly larger example.

Example 7.5.2. *For inputs $[31415, 27182, 13171]$, $k = 3$ and $b = 15$. Let $s = 1$, so $d = \lceil \frac{b}{s} \rceil = 15$. Writing the exponents in binary, we have*

$$E_1 = \mathbf{1\,1\,1\,1\,0\,1\,0\,1\,0\,1\,1\,0\,1\,1\,1};$$
$$E_2 = \mathbf{1\,1\,0\,1\,0\,1\,0\,0\,0\,1\,0\,1\,1\,1\,0};$$
$$E_3 = \mathbf{0\,1\,1\,0\,0\,1\,1\,0\,1\,1\,1\,0\,0\,1\,1}.$$

[6] In general, $s = 1$ is a better choice for exponents of this size, but not in this case.

The vectors $v(j)$ can be read down the columns corresponding to powers of 2^j, with 2^0 on the right. The $\beta_{v(j)}$ computed in step 2 may be written with binary exponents as:

$$\beta_{001} = \alpha^{000000101000000_2}, \quad \beta_{010} = \alpha^{000000000001000_2}, \quad \beta_{100} = \alpha^{000000010000000_2},$$
$$\beta_{101} = \alpha^{001000000010001_2}, \quad \beta_{110} = \alpha^{100100000000100_2}, \quad \beta_{111} = \alpha^{010001000100010_2}.$$

Step 2 squares α fourteen times and performs eight non-trivial multiplications. Step 3 computes:

$$\beta_{110} = \beta_{110}\beta_{111} = \alpha^{110101000100110_2}, \qquad \beta_{001} = \beta_{001}\beta_{111} = \alpha^{010001101100010_2},$$
$$\beta_{100} = \beta_{100}\beta_{110} = \alpha^{110101010100110_2}, \qquad \beta_{010} = \beta_{010}\beta_{110} = \alpha^{110101000101110_2},$$
$$\beta_{100} = \beta_{100}\beta_{101} = \alpha^{111101010110111_2}, \qquad \beta_{001} = \beta_{001}\beta_{101} = \alpha^{011001101110011_2},$$

using six group operations. The output is then $[\beta_{100}, \beta_{010}, \beta_{001}]$. The complete addition chain is

1, 2, 4, 8, 16, 32, 64, 128, 256, 320, 512, 1024, 2048, 4096, 4112, 4113, 8192, 8704, 8736, 8738, 9058, **13171**, *16384, 18432, 18436, 27174,* **27182**, *27302,* **31415**

Individual binary exponentiation of the three exponents in the example above would require sixty-seven group operations rather than twenty-eight. If we had used $s = 2$, thirty group operations would have been performed.

We now prove the correctness of the algorithm and bound its complexity.

Proposition 7.4. *Given inputs $\alpha \in G$, positive integers $E_1, \ldots, E_k$, and a positive integer s, Algorithm 7.5 outputs $[\alpha^{E_1}, \ldots, \alpha^{E_k}]$.*

Proof. We adopt additive group notation in this proof, so we write $k\alpha$ rather than α^k and 0 in place of 1_G. Let b be the number of bits in the largest E_i, and let $d = \lceil \frac{b}{s} \rceil$, where s is any positive integer. The ith desired output may be written as

$$E_i\alpha = \sum_{j=0}^{d-1} E_{ij} 2^{sj} \alpha = \sum_{j=0}^{d-1} [v(j)]_i 2^{sj} \alpha, \tag{7.7}$$

where the $v(j)$ are the k-place vectors defined in step 1 of Algorithm 7.5. We may regroup the terms of the sum according to the values β_x computed in step 2 to obtain

$$E_i\alpha = \sum_x x_i \beta_x, \tag{7.8}$$

where the sum is over all $x \in \{0, \ldots, 2^s - 1\}^k$, since $\beta_x = 0$ for any x not equal to some $v(j)$.

We now show that the sum in (7.8) is effectively unchanged by step 3 of the algorithm. If $z = x + y$, the contribution of the terms involving x, y, and z is

$$x_i\beta_x + y_i\beta_y + z_i\beta_z = x_i\beta_x + y_i\beta_y + (x_i + y_i)\beta_z = x_i(\beta_x + \beta_z) + y_i(\beta_y + \beta_z).$$

Thus if we add β_z to both β_x and β_y and then suppose β_z is set to 0, the sum is unchanged. At the completion of step 4, the only non-zero terms of the sum that remain have x with weight 1 and x_i non-zero. The only such x is the unit vector $e(i)$, thus (7.8) implies $E_i\alpha = \beta_{e(i)}$, which is the value returned by the algorithm. □

Proposition 7.5. *For k inputs with at most b bits and window size s, Algorithm 7.5 uses at most*

$$cost(k, b) = b + \lceil b/s \rceil + 2^{sk} - 2k - 2 \tag{7.9}$$

non-trivial group operations and $2^{sk} + O(1)$ group storage.

Proof. Let $d = \lceil \frac{b}{s} \rceil$. A total of $(d-1)s \leq b - 1$ squarings of α are performed in step 2, along with up to d multiplications involving the elements β_x. Step 3 performs at most two multiplications for each of the $2^{sk} - k - 1$ vectors with weight greater than one. Thus the total number of group operations is bounded by

$$b - 1 + \lceil b/s \rceil + 2(2^{sk} - k - 1). \tag{7.10}$$

We now show that (7.10) overcounts the non-trivial group operations by at least $2^{sk} - 1$. Every β_x which takes on a non-trivial value during the algorithm's execution is at some point the output of a group operation of the form $\beta_x = \beta_x\gamma$, with inputs $\beta_x = 1_G$ and $\gamma \neq 1_G$. Note that unit vectors (vectors with weight one) all must take on a non-trivial value at some point since the input exponents are non-zero and we may assume $\alpha \neq 1_G$ (otherwise every operation is trivial). If the value of β_x is never non-trivial and x has weight greater than one, then we have included two operations in (7.10) which did not occur. Thus, for each of the $2^{sk} - 1$ non-zero vectors, we have included at least one distinct group operation in (7.10) which was either trivial or did not occur. Subtracting $2^{sk} - 1$ yields the desired time bound. The space bound is immediate since there are $2^{sk} - 1$ group elements β_x. □

In the worst case, β_x is non-trivial for every non-zero vector x, and the cost estimate above is tight. In this situation the particular choice of x and y in step 3 of Algorithm 7.5 is immaterial.

For specific values of k and b, the optimal s can be determined via the cost function. In particular, for $s = 1$ we have the following corollary.

Corollary 7.6. *Let $c > 2$ be a constant and let $f_c(n) = \max\{k : cost(k, n) \leq cn\}$, where $cost(k, n)$ is the cost function of Algorithm 7.5 given by (7.9) with $s = 1$.*

(1) $f_c(n) \geq 3$ for all n.

(2) $f_c(n) = \lg n + O(1)$.

(3) If $k = f_c(n)$ then Algorithm 7.5 uses $O(n)$ storage.

Proof. Assume $c > 2$. Setting $s = 1$, Proposition 7.5 implies

$$cost(k, n) = 2n + 2^k - 2k - 2.$$

Thus $cost(3, n) = 2n < cn$ and $f_c(n) \geq 3$ for all n, proving (1). For (2), we note that $f_c(n)$ maximizes k subject to:

$$\begin{aligned} 2n + 2^k - 2k - 2 &\leq cn \\ 2^k \left(1 - \frac{2(k+1)}{2^k}\right) &\leq (c-2)n = \epsilon n, \end{aligned}$$

where $\epsilon = c - 2 > 0$. Note that k is unbounded as $n \to \infty$, and in particular, for sufficiently large n we may assume $k > 3$. Taking logarithms, we have

$$\begin{aligned} k + \lg\left(1 - \frac{2(k+1)}{2^k}\right) &\leq \lg n + \lg \epsilon \\ k &\leq \lg n + \lg \epsilon - o(1) = \lg n + O(1), \end{aligned}$$

which proves (2). For (3), we note that for $k = \lg n + O(1)$ and $s = 1$, the space used by Algorithm 7.5 is, by Proposition 7.5, $2^{sk} = 2^{\lg n + O(1)} = O(n)$. □

By way of comparison, we consider a commonly used multi-exponentiation algorithm: the fixed-base windowing method originally proposed by Yao [115] and generalized by

	$c = 2\frac{1}{4}$		$c = 2\frac{1}{2}$	
n	$f_c(n)$	$f'_c(n)$	$f_c(n)$	$f'_c(n)$
8	1	3	2	3
32	2	4	2	4
128	3	5	4	6
512	4	7	5	8
2048	6	9	7	10

Table 7.1: Comparison of Multi-Exponentiation Packing Functions

Brickell et al. [22] (see also [73, Algorithm 14.109]). Including precomputation, the total number of group operations is approximately

$$cost(k, b) = b + k\left(\lceil b/s \rceil + 2^s - 2\right) - 1. \tag{7.11}$$

The optimal window size is typically $s \approx \lg b - \lg\lg b$. When $b = 32$ and $k = 4$, the fixed-base windowing method requires 99 group operations, while Algorithm 7.5 uses at most 70. For $b = 512$ and $k = 7$ we have 1442 versus 1136.

In terms of the packing function $f_c(n)$, for an appropriate choice of c the cost function in (7.11) gives $f_c(n) = \lg n - \lg\lg n + O(1)$, compared to the bound $f_c(n) = \lg n + O(1)$ given by Proposition 7.5. This results in significant differences in the values of $f_c(n)$ relating to the sizes of exponents that arise in practice. This is shown in Table 7.1, where $f_c(n)$ corresponds to the fixed-base windowing method and $f'_c(n)$ corresponds to Algorithm 7.5. Recall that $f_c(n)$ represents the maximum number of n-bit exponents that can be exponentiated using no more than cn group operations, so larger values of $f_c(n)$ are better, and we want c to be as small as possible.

In the field of exponentiation algorithms, where performance comparisons typically involve small percentages, these are significant differences. In the case of the snowball algorithm, these differences are magnified by the fact that the algorithm is quite sensitive to the values of k in the early stages of the recursion. Once k begins to increase, it does so rapidly. Algorithm 7.5 causes this to happen sooner. If we start the snowball algorithm with an arbitrary number of 16-bit inputs, the size of the inputs at each level of the recursion may be approximated by the recurrence $n_0 = 16$, $n_{i+1} = k * n_i$ where k is maximized subject

to $cost(k, n_i) \leq cn$. The cost function for the standard windowing method gives the sequence

$$16, 32, 96, 384, 2304, 20736, \ldots,$$

whereas the cost function for Algorithm 7.5 produces the sequence

$$16, 80, 560, 5600, 78400, 1411200, \ldots,$$

effectively reducing the depth of the recursion (and the running time) by more than 50%. The snowballs grow a lot faster on a hill that is slightly steeper at the start.

Note that both sequences grow super-exponentially; this is what gives the snowball algorithm its advantage. By comparison, the sequence of exponent sizes in the recursive algorithm of Celler and Leedham-Green is the geometric sequence

$$16, 32, 64, 128, 512, 1024, \ldots.$$

7.7 Performance Comparisons

Table 7.2 lists performance test results for several of the fast order algorithms presented in this chapter. Each test computed $|\alpha|$ for random $\alpha \in G$ given the prime factorization of an exponent N of G. The numbers listed under each algorithm count group operations, where "Classic+" indicates the classical algorithm with multi-exponentiation, and "CLG" indicates the algorithm of Celler and Leedham-Green (Algorithm 7.3). For cyclic groups, $|C_N| = N$ was used as the exponent. In the random case, Bach's algorithm was used to generate random integers with known factorizations [9]. For the symmetric groups S_m, the exponent $N = \prod_p p^{\lfloor \log_p m \rfloor}$ was used. The constant $c = 2.1$ was used in the snowball algorithm.

The major points worth noting are that the snowball algorithm is generally about twice as fast as the algorithm of Celler and Leedham-Green, and both are much faster than the classical algorithm, even for fairly moderate values of w.

Group	n	w	Classic	Classic+	CLG	Snowball
C_N	256	5.8	1,854	613	848	496
(random N)	1,024	6.7	8,743	2,382	3,446	1,914
	4,096	8.2	44,243	9,294	16,619	8,393
	16,384	9.3	204,924	35,742	56,925	32,501
$C_{541^{\#}}$	730	100	108,331	17,684	7,397	3,839
$C_{1223^{\#}}$	1704	200	508,534	69,662	19,890	10,303
$C_{2741^{\#}}$	3,886	400	2,327,008	273,813	51,237	25,148
$C_{6133^{\#}}$	8,716	800	10,446,504	1,078,709	128,099	64,578
$C_{13499^{\#}}$	19,292	1,600	46,273,593	4,255,475	312,786	141,364
$C_{29433^{\#}}$	42,253	3,200	202,764,477	16,662,597	748,904	340,374
S_{100}	136	25	4,891	1,215	750	344
S_{400}	574	78	66,095	11,368	4,924	2,029
S_{1600}	1,144	251	863,727	112,066	24,648	7,954

Table 7.2: Performance Comparison of Fast Order Algorithms

Chapter 8

Computing the Exponent of a Group

Chapter Abstract

We apply order algorithms to compute the group exponent, $\lambda(G)$, of abelian and certain non-abelian groups. We then show how to compute the minimal exponent of a given set of elements in any group, and use this to prove the Order Computation Theorem: all order computations performed by a generic algorithm can be accomplished using essentially the same number of group operations as a single computation on an element with order $\lambda(G)$.

We call an integer N for which $\alpha^N = 1_G$ an exponent of α. For $\alpha \in G$ the order of the group is necessarily an exponent of α, thus $|G|$ is an exponent of every $\alpha \in G$. However $|G|$ is not necessarily the least number with this property.

Definition 8.1. *The* **exponent** *of a finite group G is the least positive integer N for which $\alpha^N = 1_G$ for all $\alpha \in G$. The exponent of G is denoted $\boldsymbol{\lambda(G)}$.*

The exponent of a group is the least common multiple of the orders of all its elements, and therefore divides any common multiple, including $|G|$. It follows from the Sylow theorems that $\lambda(G)$ is divisible by each prime p dividing $|G|$, since G must contain an element of order p. When $G = \mathbb{Z}_m$, the exponent of G may be computed via the Carmichael function, $\lambda(m)$, which is defined recursively for positive integers by letting $\lambda(1) = 1$ and

$$\begin{aligned}
&\lambda(2^h) = 2^{h-2}; && \\
&\lambda(p^h) = \phi(p^h) = p^h - p^{h-1} && \text{for prime } p; \\
&\lambda(ab) = \operatorname{lcm}\big(\lambda(a), \lambda(b)\big) && \text{when } a \perp b.
\end{aligned}$$

Group	$\lvert G\rvert$	$\lambda(G)$	Notes
$\mathbb{Z}_{91}^*$	72	12	Multiplicative group of integers mod 91
S_7	5,040	420	Permutations of seven elements
$GL_2(\mathbb{F}_7)$	2016	336	2x2 non-singular matrices over $\mathbb{F}_7$
$E(x^3+42x+1, \mathbb{F}_{101})$	96	24	Elliptic curve $y^2 = x^3 + 42x + 1$ over $\mathbb{F}_{101}$
$Cl\left(\mathbb{Q}\sqrt{-4004}\right)$	40	10	Ideal class group of $\mathbb{Q}\sqrt{-4004}$

Table 8.1: Orders and Exponents of Some Finite Groups

When G is cyclic, we have $\lambda(G) = |G|$, and for abelian groups, the converse holds. In general, $\lambda(G)$ and $|G|$ may be quite different, as seen in Table 8.1.

8.1 Computing $\lambda(G)$

A simple approach to computing $\lambda(G)$ is to compute the orders of a random sample of group elements and output their least common multiple. Clearly this will work if the sample is large enough, but can we determine the exponent of the group with probability bounded above 1/2 using a small sample, preferably one of constant size? In general, the answer is "no", as may be seen by considering the symmetric group S_p for prime p. The probability of finding a random element with order p is only $(p-1)!/p! = 1/p$, and we need to find all the prime powers up to p. To have a good chance of correctly determining $\lambda(S_p)$, we would need a fairly large sample, certainly more than constant size.

On the other hand, if we only wanted a set of generators, we could pick just three random elements and be fairly confident we could generate the entire group. This is true of many non-abelian groups. The expected number of random elements needed to generate any finite simple group is at most $2 + o(1)$, as conjectured by Dixon [44] and proven by Liebeck and Shalev [71] (see Pak [81] for more general results). Unfortunately the orders of a set of generators may give little information regarding the exponent the group. The group S_n, for example, can be generated entirely by elements of order 2.

Given a set of generators for an abelian group, however, we can easily compute $\lambda(G)$. The least common multiple of their orders will be an exponent of every generator, and thus an exponent of any product of generators, since G is commutative. The minimum number of elements required to generate an abelian group may be as large as $\lg|G|$, as

when G is the direct product of cyclic groups of order 2. In general, the expected number of random elements required to generate an abelian group is $r + \sigma$, where $r \leq \lg |G|$ is the size of the smallest generating set, and the constant $\sigma \approx 2.1184$, as shown by Pomerance in [85]. Thus, $O\left(\lg |G|\right)$ random elements suffice to determine the exponent of an abelian group G. However, only two are necessary, with probability at least $6/\pi^2 > 1/2$.

Theorem 8.1. *Let α and β be uniformly random elements of a finite abelian group G.*

$$\Pr\left[\operatorname{lcm}\left(|\alpha|, |\beta|\right) = \lambda(G)\right] > \frac{6}{\pi^2} \tag{8.1}$$

Proof. Recall that every abelian group is isomorphic to a direct product of cyclic groups. Thus we may represent G using a basis of independent elements of prime-power order $\gamma_1, \ldots, \gamma_n$, so that each $\alpha \in G$ has a unique exponent vector $[e_1, \ldots, e_n]$ whose ith component is an integer modulo $|\gamma_i|$. Let $p_1, \ldots, p_w$ be the primes dividing G. If $p_i^{h_i}$ is the maximum power of p_i equal to some $|\gamma_j|$, then the exponent of G is simply

$$\lambda(G) = p_1^{h_1} \cdots p_w^{h_w}. \tag{8.2}$$

For α uniformly distributed over G, e_i is uniformly distributed over the integers modulo $|\gamma_i|$. In particular, if $|\gamma_i| = p^h$ and $e_i \perp p^h$, then $|\alpha|$ will be divisible by p^h. This occurs with probability

$$\frac{\phi(p^h)}{p^h} = \frac{p^h - p^{h-1}}{p^h} = 1 - p^{-1}. \tag{8.3}$$

Thus for each maximal p^h dividing $\lambda(G)$, the order of α is divisible by p^h with probability at least $1 - p^{-1}$, since there is at least one γ_i with order p^h. The probability that either α or β has order divisible by p^h is then at least $1 - p^{-2}$. If this is true for all $p|\lambda(G)$, then we have $\operatorname{lcm}(\alpha, \beta) = p_1^{h_1} \cdots p_w^{h_w} = \lambda(G)$ as desired. Since the e_i are independently distributed, the probability that p^h divides one of $|\alpha|$ or $|\beta|$ for every $p|\lambda(G)$ is at least

$$\prod_{p|\lambda(G)} \left(1 - p^{-2}\right) > \prod_{p} \left(1 - p^{-2}\right) = \frac{1}{\zeta(2)} = \frac{6}{\pi^2}, \tag{8.4}$$

which completes the proof. □

The theorem implies that we can compute the exponent of an abelian group with an exponentially small probability of error using just a constant number of random order

computations.[1]. While this simple approach will work, it is needlessly inefficient. Even if the sample size is a small constant, as implied by the theorem above, it will still need to be a constant substantially larger than 1. In order to achieve a satisfactory level of confidence in the output, it may be necessary to perform 20 or more order computations. This is unnecessary; essentially only "one" order computation is required.

We start by selecting a random $\alpha_1 \in G$, and compute $E = e_1 = |\alpha_1|$, which is a divisor of $\lambda(G)$. We then select a random $\beta \in G$, but rather than computing the order of β, we compute the order of $\alpha_2 = \beta^E$, and let $e_2 = |\alpha_2|$. By Lemma 8.2 below, $E = e_1 e_2$ is also a divisor of $\lambda(G)$. If α_2 is the identity element, then $e_2 = 1$ and the order computation was trivial. Otherwise our new E is a larger divisor of $\lambda(G)$, and the cost of finding it depends on e_2, not E. In the event that we only make a little progress, we only have to do a little work. In fact, our fondest hope is that we get many small e_i rather than one big one.

The algorithm continues in this fashion until the identity element is obtained repeatedly (as many times as required to achieve the desired level of confidence), at which point $E = \lambda(G)$ with high probability (for suitable G). At most a logarithmic number of nontrivial exponentiations are required, thus the total cost of the exponentiations is negligible compared to the order computations. The result is an algorithm whose computational complexity is effectively no worse than the cost of performing just a single order computation on an element with order $\lambda(G)$. It may be substantially better than this if $\lambda(G)$ is composite and we get lucky.

Algorithm 8.1 (Randomized Exponent Algorithm)**.** *Given an integer parameter $r > 1$ and a randomized black box for the group G, the probabilistic algorithm below outputs a divisor of $\lambda(G)$.*

1. **Initialize**: Let $E \leftarrow 1$ and $t \leftarrow 0$.
2. **Sample**: Set $\alpha \leftarrow \beta^E$ for a random β in G.
3. **Trivial?**: If $\alpha = 1_G$, increment t and if $t \geq r$ return E, otherwise go to step 2.
4. **Compute $|\alpha|$**: Compute $|\alpha|$, set $E \leftarrow |\alpha|E$, and go to step 2.

The correctness of the algorithm follows inductively from the lemma below.

Lemma 8.2. *Let β be an element of a finite group G. If e divides $\lambda(G)$ then $|\beta^e|e$ also divides $\lambda(G)$.*

[1]Similar but weaker statements may be made for some non-abelian groups (see Proposition 8.4)

Proof. Suppose e divides $\lambda(G)$, and let $d = |\beta^e|$. Then d divides $\lambda(G)$, and if de does not, then there is a prime power p^h which divides de but not $\lambda(G)$. We claim that this cannot be so. Such a p must divide both d and e, and $p^h \nmid \lambda(G)$ implies that $p^h \nmid |\beta|$. It follows that de/p is an exponent of β (since de is) and therefore d/p is an exponent of β^e. But this contradicts the fact that $d = |\beta^e|$ is the minimal exponent of β^e. □

Algorithm 8.1 always outputs a divisor of $\lambda(G)$, but we would like to bound the probability that the algorithm fails to output $\lambda(G)$. We first consider the abelian case.

Proposition 8.3. *Let G be a finite abelian group. If E is the value output by Algorithm 8.1 then*

$$Pr[E \neq \lambda(G)] < \frac{1}{2^{r-2}},$$

where $r \geq 2$ is the input bound to the algorithm.

Proof. When Algorithm 8.1 terminates, it has found random elements $\beta_1, \ldots, \beta_r$ for which E is an exponent. If every maximal prime power p^h dividing $\lambda(G)$ divides the order of some β_i, then we have $e = \lambda(G)$. Following the proof of Theorem 8.1, the probability that p^h divides the order of β_i is $1 - p^{-1}$, independently for each β_i. Thus the probability that p^h divides the order of some β_i is $1 - p^{-r}$. The probability that every maximal prime-power divisor of $\lambda(G)$ divides some β_i is then

$$\prod_{p|\lambda(G)} (1 - p^{-r}) > \prod_{p} (1 - p^{-r}) = \frac{1}{\zeta(r)}, \tag{8.5}$$

where we note that $r > 1$, so the infinite product converges. Thus we have

$$Pr[E \neq \lambda(G)] < 1 - \frac{1}{\zeta(r)} = \frac{\zeta(r) - 1}{\zeta(r)} \leq \zeta(r) - 1 = \sum_{1 \leq n} n^{-r} - 1 = 2^{-r} + \sum_{2 < n} n^{-r}, \tag{8.6}$$

since $\zeta(r) > 1$. We now bound the sum using a Stieltjes integral:

$$\sum_{2 < n} n^{-r} = \int_2^\infty t^{-r} d\lfloor t \rfloor = \int_2^\infty t^{-r} dt - \int_2^\infty t^{-r} d\{t\} \leq \int_2^\infty t^{-r} dt = \frac{2^{1-r}}{r-1} \leq 2^{1-r}. \tag{8.7}$$

Substituting into 8.6 and noting that $2^{-r} + 2^{1-r} < 2^{2-r}$ completes the proof. □

A more general, but weaker, proposition can be applied to certain non-abelian groups.

Matrix groups of low dimension are one example. The exponent of the group $GL_2(\mathbb{F}_p)$ is

$$\lambda(GL_2(\mathbb{F}_p)) = p \cdot \mathrm{lcm}(p-1, p^2-1),$$

and a similar formula holds for $GL_d(\mathbb{F}_q)$ [78]. The order of $GL_2(\mathbb{F}_p)$ can be effectively computed by Algorithm 8.1 (which of course doesn't know p), because there is a reasonably high probability of obtaining a random element with order divisible by any particular prime power in $\lambda(G)$.

Proposition 8.4. *Fix $\epsilon > 0$, and suppose G is a group where each prime power dividing $\lambda(G)$ divides a random element of G with probability at least ϵ. Let E be the value output by Algorithm 8.1. There exists an $r = O\left(\log\log(N)\right)$, where $N = \lambda(G)$, such that*

$$\Pr[E \neq \lambda(G)] = O\left(\frac{1}{\lg^k N}\right),$$

for any constant k.

Proof. The probability that a maximal prime power that divides $\lambda(G)$ does not divide the order of any of the r random elements whose order divides E is at most $(1-\epsilon)^r$. Noting that there are at most $\lg \lambda(G) = \lg N$ prime powers that divide $\lambda(G)$ and applying a union bound yields

$$\Pr[E \neq \lambda(G)] \leq \lg N (1-\epsilon)^r.$$

For any constant k, we may choose $r = c \log\log N$ (where c depends on k and ϵ) to satisfy the proposition. □

The proposition above can be generalized by considering non-constant ϵ, e.g. ϵ inversely proportional to $\log |G|$, requiring larger values of r.[2]

8.2 Order Complexity Bounds

We now consider the running time of Algorithm 8.1, which is the principal item of interest. We will show that the complexity is essentially dominated by the equivalent of a single

[2] It might be interesting to classify finite groups based on the divisibility of element orders by prime powers. For a given group G, what is the minimum probability that $p^h \| \lambda(G)$ divides a random element?

order computation. Given the variety of complexity bounds proven in Chapters 4 and 5, it is useful to have a generic bound $T(N)$ that may represent any of these results.

A key feature of all these bounds is that it is always easier (or at least never harder) to compute $|\alpha|$ and $|\beta|$ separately than to compute the order of a single element γ with $|\gamma| = |\alpha||\beta|$. Thus we have $T(a) + T(b) \leq T(ab)$. This can be derived from the analytic properties of the functions used in many of the bounds (e.g. $\sqrt{a} + \sqrt{b} = O\left(\sqrt{ab}\right)$), but this is misleading. If $T(N)$ is the number of group operations required to compute $|\alpha| = N$, it may well be the case that $T(N+1) \ll T(N)$, e.g. if N is prime and $N+1$ highly composite, which is certainly not true of $\sqrt{N}$. The fact that $T(a) + T(b) \leq T(ab)$ indicates a fundamental connection between the complexity of order computations and the multiplicative structure of integers. This illustrates another distinctive property of generic order computations. Assuming an algorithm has no information about G or α, the complexity of computing $|\alpha|$ is entirely a function of the integer $N = |\alpha|$, independent of the group G.[3]

Definition 8.2. *An* **order complexity bound** $T(N)$ *is a function that bounds the complexity of computing the order of α in the cyclic group $\langle\alpha\rangle \cong C_N$, and satisfies $T(a) + T(b) \leq T(ab)$ for all integers $a, b > 1$ when the product ab is sufficiently large.*

Proposition 8.5. *Let $T(n)$ be any order complexity bound. The complexity of Algorithm 8.1 is bounded by*

$$T\big(\lambda(G)\big) + O\left(r \lg \lambda(G) + \lg^2 \lambda(G)\right), \tag{8.8}$$

where r is the input bound on the number of times the algorithm may compute 1_G.

Proof. Let $e_1, \ldots, e_k$ be the sequence of non-trivial values of e_i computed by Algorithm 8.1. Then each e_i is an integer greater than 1 and $e_1 \cdots e_k$ divides $\lambda(G)$, by Lemma 8.2. Thus $k \leq \lg \lambda(G)$, and the number of exponentiations is bounded by $r + k \leq r + \lg \lambda(G)$. The cost of each exponentiation is $O\left(\lg E\right) = O\left(\lg \lambda(G)\right)$, since $E \leq \lambda(G)$, which implies the right-hand term in 8.8. For sufficiently large $\lambda(G)$, the cost of the order computations is bounded by

$$T(e_1) + \cdots + T(e_k) \leq T(e_1 \cdots e_k) \leq T(\lambda(G)),$$

which is the left-hand term in 8.8, completing the proof. □

[3]The complexity of computing $|\alpha|$ in the cyclic group generated by α is never any easier (in terms of group operations or group storage) than computing $|\alpha|$ in G.

Corollary 8.6. *If r is polylogarithmic in $\lambda(G)$ and $T(N)$ is an order bound satisfying $T(N) = O(N^c)$ for some $c > 0$, then the complexity of Algorithm 8.1 is bounded by $\big(1 + o(1)\big)T\big(\lambda(G)\big)$.*

The appropriate complexity bound $T(N)$ to use in Corollary 8.6 depends on whether one is interested in the worst, average, or median complexity of Algorithm 8.1. All of the bounds given in Chapters 4 and 5 may be applied, to both time and space, including bounds of the form $T(N) = O\left(N^{1/u}\right)$ implied by Propositions 4.7 or 5.3.

Remark 8.1. *In the expression $T\big(\lambda(G)\big)$, a particular integer $N = \lambda(G)$ is referred to. If $\lambda(G)$ happens to be highly composite, Algorithm 8.1 will benefit. This is true even though it may never compute the order of an element α with $|\alpha| = \lambda(G)$. Indeed, in non-abelian groups no such element may exist. Similar comments apply in the next section to $\lambda(S)$.*

8.3 Computing $\lambda(S)$

We now give a deterministic exponent algorithm which computes the least common multiple of the orders of a set of group elements S. Its output is the minimal exponent of the input set, which we denote $\lambda(S)$. If S generates an abelian group G, we have $\lambda(S) = \lambda(G)$, but the algorithm correctly computes $\lambda(S)$ in any case, whether G is abelian or not.

Algorithm 8.2 (Set Exponent Algorithm)**.** *Given a set of group elements $S = \{\beta_1, \ldots, \beta_k\} \subseteq G$, the algorithm below computes $E = \lambda(S)$, the least common multiple of $|\beta_i|$ over $\beta_i \in S$.*

1. **Initialize**: Set $E \leftarrow 1$ and $i \leftarrow 1$.
2. **Exponentiate**: Set $\alpha \leftarrow \beta_i^E$.
3. **Compute $|\alpha|$**: Compute $|\alpha|$ and set $E \leftarrow |\alpha|E$.
4. **Loop**: If $i < k$, set $i \leftarrow i + 1$ and go to step 2, otherwise return E.

The correctness of the algorithm follows from Lemma 8.2.

Proposition 8.7. *Let $T(n)$ be any order complexity bound. The complexity of Algorithm 8.2 is bounded by*

$$T\big(\lambda(S)\big) + O\left(|S| \lg \lambda(S)\right),$$

where S is the set of input elements in G.

The proof of the proposition is a simplification of the argument in Proposition 8.5. Unless S is exponentially large, the running time is dominated by $T\big(\lambda(S)\big)$. This effectively means that the running time of the algorithm is the same as if it were given a single element with order $\lambda(S)$, even though such an element may not exist in G. An order complexity bound $T(N)$ is defined for all positive integers N.

Corollary 8.8. *Suppose $|S|$ is polylogarithmic in $\lambda(S)$. If $T(N)$ is an order bound with $T(N) = O(N^c)$ for some $c > 0$, the complexity of Algorithm may be bounded by $\big(1 + o(1)\big)T\big(\lambda(S)\big)$.*

8.4 The Order Computation Theorem

Corollary 8.8 has significant implications. Once the exponent of the group is known, any subsequent order computations can be accomplished in near linear time using a fast order algorithm such as the snowball algorithm (Algorithm 7.4). In fact, an even stronger statement can be made, which leads us to the central theorem of this thesis. We first define a fast order complexity bound.

Definition 8.3. *A* **fast order complexity bound** *$T(N)$ is a function that bounds the complexity of computing the order of $\alpha \in C_N$ given the prime factorization of N.*

Theorem 8.9 (Order Computation)**.** *Let $S \subseteq G$ be the set of all group elements whose order is computed by a generic algorithm at some point during its execution. The total complexity of all order computations may be bounded by*

$$\big(1 + o(1)\big)T_1\big(\lambda(S)\big) \;+\; \big|S\big|T_2\big(\lambda(S)\big)$$

where $T_1(N)$ is any order complexity bound and $T_2(N)$ is any fast order complexity bound.

Proof. Any generic algorithm can implement Algorithm 8.2 in a dynamic fashion by maintaining a factored exponent $E = \lambda(S_0)$, where $S_0 \subseteq S$ contains the elements whose order has been determined. Set $E \leftarrow 1$ initially, and each time an order computation for a new $\beta \in S$ is required, first compute $\alpha = \beta^E$. If $\alpha = 1_G$ then E is an exponent of β which can be used for a fast order computation. Otherwise, compute $m = |\alpha|$ using a general order algorithm, factor m, and set $E \leftarrow mE$, updating its factorization appropriately. The new E

is a factored exponent of $S_0 \cup \{\beta\}$ and can be used in a fast order computation to determine $|\beta|$. The algorithm then sets $S_0 \leftarrow S_0 \cup \{\beta\}$ and continues.

The total cost of all the general order computations is $T_1(|\alpha_1|) + \cdots + T_1(|\alpha_k|)$, where the α_i are elements with order greater than 1, and $E = \lambda(S)$ is the product of their orders. We then have

$$T_1\big(|\alpha_1|\big) + \cdots T_1\big(|\alpha_k|\big) \le T_1\big(|\alpha_1| \cdots |\alpha_k|\big) = T_1\big(\lambda(S)\big),$$

for all sufficiently large values of $\lambda(S)$.

We may assume that no fast order computations are repeated, since the results can be saved using no group storage (and in any event less space than required by the elements of S). The complexity of all fast order computations is then $\big|S\big|T_2\big(\lambda(S)\big)$. □

Remark 8.2. *The cost of factoring N deterministically is $O\left(N^{1/4}\log^2 N\right)$ arithmetic operations, and many faster probabilistic algorithms exist. Even when $T(N)$ is less than this bound, the property of N that makes order computations easier (divisibility by small primes) also makes factoring easier. One way to see this is to note that an order computation in the group $\mathbb{Z}_N^*$, which has order $\phi(N)$, is easier (for all the algorithms presented in Chapters 4 and 7) than an order computation in C_N, since $\phi(N) = \prod_{p^h \| N} p^{h-1}(p-1)$ has more small prime factors than N does.*

Corollary 8.10. *Let $T(N)$ be an order complexity bound satisfying $T(N) = O\left(N^c\right)$ for some $c > 0$. If S is a set of group elements with $|S|$ polylogarithmic in $\lambda(S)$, the total cost of computing the order of every element in S is $\big(1 + o\left(1\right)\big)T(N)$.*

If an algorithm is working in just a subgroup $H \subseteq G$, the theorem and corollary imply that the complexity of its order computations depends only on $\lambda(H)$, which is an upper bound on $\lambda(S)$ for any $S \subseteq H$, and does not depend on $\lambda(G)$. This can be important in many situations, particularly with non-abelian groups. Note that the computation of $\lambda(S)$, as in Algorithm 8.2, does not depend on G being abelian, nor do the fast order computations.

The implication of the Order Computation Theorem is that all the order computations of a generic algorithm can be satisfied for essentially the cost of a single order computation on an element with order $\lambda(G)$. In fact, this is the worst-case scenario. In many non-abelian groups there is no single element with order $\lambda(G)$, and we will at most compute the order of several elements whose product is $\lambda(G)$.

In abelian groups, we can say that the total cost of all order computations is essentially the same as the expected cost of a single random order computation. In any abelian group

where $\lambda(G)$ is large, a random element will have order N large enough that the cost of all subsequent order computations, which will involve elements with order at most $\lambda(G)/N$, will be dominated by the cost of the first. If $\lambda(G)$ is not large, neither is the cost of order computations.

The practical consequence of the Order Computation Theorem is that the remarkable performance of the primorial-steps and multi-stage sieve algorithms given in Chapters 4 and 5 may be exploited to solve a wide variety of problems based on order computations. This leads to much faster algorithms not only for computing $\lambda(G)$, but also for the next problem we consider, determining the structure of an abelian group.

Chapter 9

Computing the Structure of an Abelian Group

Chapter Abstract

We use the group exponent $\lambda(G)$ to compute the structure of an abelian group G. In most cases the group structure can be computed using no more than $\left(1+o\left(1\right)\right)\sqrt{2|G|}$ group operations. Given a bound $M = |G|^{\delta}$ on the size of the group, we can almost always compute the structure of G using $O\left(|G|^{\delta/4}\right)$ group operations once $\lambda(G)$ is known. Combined with the results of the previous chapter, this gives a total complexity that is essentially the same as a single order computation in G.

From the structure theorem for finitely generated abelian groups, we know that every finite abelian group G is isomorphic to a direct product of cyclic groups of prime-power order. This product is unique up to the order of the factors. For the multiplicative group of integers modulo 105 we have

$$\mathbb{Z}_{105}^{*} \cong C_2 \otimes C_4 \otimes C_6 \cong C_2 \otimes C_2 \otimes C_4 \otimes C_3.$$

Alternatively, we may express G uniquely as a product of cyclic groups $C_1 \otimes \cdots \otimes C_k$ with the order of C_i dividing the order of C_{i+1}, as in

$$\mathbb{Z}_{105}^{*} \cong C_2 \otimes C_2 \otimes C_{12}.$$

It is straight forward to convert between these two forms of representation. The divisor representation minimizes the number of factors, while the prime-power representation

minimizes the size of the factors. Typically the divisor representation is favored, especially by algorithms whose complexity depends on the number of generators required to represent the product of cyclic groups. We will find it more useful to work with the prime-power representation, however. Note that we can easily construct a set of generators for the divisor representation from a set of generators for the prime-power representation by combining generators appropriately.

The ultimate challenge for a generic algorithm working in an abelian group is to explicitly construct the representation of G as a product of cyclic groups. By this we require the algorithm to not only identify the order of each factor group, but to also output a corresponding set of generators which form a *basis* for the group.

Definition 9.1. *A* **basis** *for a finite abelian group G is an ordered set (vector) of group elements, $\vec{\alpha} = [\alpha_1, \ldots, \alpha_k]$ with the property that every $\beta \in G$ can be uniquely expressed in the form $\beta = \vec{\alpha}^{\vec{e}} = \alpha_1^{e_1} \cdots \alpha_k^{e_k}$, with $0 \le e_i < |\alpha_i|$ for $1 \le i \le k$. The vector $\vec{e}$ is the* **exponent vector** *of β.*

Note that, by definition, $\alpha^0 = 1_G$. It will be convenient to use set or vector notation interchangeably, thus we may write $\langle \vec{\alpha} \rangle$ to denote the subgroup generated by the elements of the vector $\vec{\alpha}$. This subgroup may also be considered the *span* of $\vec{\alpha}$.

Given any element of the form $\beta = \vec{\alpha}^{\vec{e}} = \alpha^{e_1} \cdots \alpha^{e_k}$, where the e_i are arbitrary integers, we can express β uniquely by reducing the e_i modulo $|\alpha_i|$. A basis provides an isomorphism between G and the corresponding $\mathbb{Z}$-module of exponent vectors. An algorithm which finds a basis for G has effectively determined the structure of G.

Definition 9.2. *A generic algorithm is said to* **compute the structure** *of an abelian group G if it outputs a basis for G, along with the order of each element in the basis.*

Every basis for G generates G, but not every generating set is a basis. Any $\vec{\alpha}$ which generates G does have the property that every $\beta \in G$ can be expressed in the form $\beta = \vec{\alpha}^{\vec{e}}$, since G is abelian, however, this representation is unique only when $\vec{\alpha}$ is a basis. Non-unique representations imply the existence of a *relation* on $\vec{\alpha}$.

Definition 9.3. *A (non-trivial)* **relation** *on a vector of group elements $\vec{\alpha} = [\alpha_1, \ldots, \alpha_k]$ is an exponent vector $\vec{e} = [e_1, \ldots, e_k]$ with some $e_i \not\equiv 0 \bmod |\alpha_i|$ for which $\vec{\alpha}^{\vec{e}} = 1_G$. A vector $\vec{\alpha}$ for which no (non-trivial) relations exist is said to be* **independent**.

A basis, then, is an independent vector which generates the group, and any independent vector is a basis of the subgroup it generates. We will assume that each entry in an exponent vector is reduced modulo the order of the corresponding element. Thus a non-trivial relation is simply a non-zero exponent vector, and all relations under discussion are assumed to be non-trivial.

9.1 Existing Algorithms

There are a number of existing generic algorithms that compute the structure of an abelian group. Most are based on Shanks' baby-steps giant-steps algorithm (see [27, 28] and also [37, Algorithm 5.4.1]), but Pollard's rho method can be used to obtain a more space efficient solution [109]. These algorithms are typically given a set S of generators for G, or they may construct a set of generators by choosing random elements from G. Their main task is to find a set of relations on S which can be used to determine a basis. Given a sufficient set of relations, these can be arranged in a matrix and the Smith normal form (SNF) can be computed using standard methods, as in [37, Algorithm 2.4.14]. The SNF of a non-singular integer matrix A is the unique diagonal matrix $D = UAV$, where U and V are unimodular (determinant ± 1) and the diagonal entries of D correspond to the orders of the cyclic factors in the divisor representation of G. The transformations to A represented by U and V can then be used to determine a basis using combinations of elements in S.

These algorithms generally have a running time of $\Theta\left(|S|\sqrt{|G|}\right)$. Some algorithms also have an exponential dependency on the size of the minimal basis for G, which can cause the running time to be substantially worse in rare cases [27], but more recent developments have addressed this issue [28].

9.2 Using $\lambda(G)$ to Compute Group Structure

We use some of the same basic tools as existing algorithms, but take a slightly different approach. Our starting point will be to compute $\lambda(G)$, the exponent of G. This can be done by either Algorithm 8.1 or Algorithm 8.2 using approximately $T\big(\lambda(G)\big)$ group operations, where $T(N)$ represents the cost of computing the order of α when $|\alpha| = N$ (see Definition 8.2). If the primorial-steps algorithm is used, $T\big(\lambda(G)\big) = o\left(\sqrt{|G|}\right)$, and may be much smaller.

Our challenge is to use $\lambda(G)$ to efficiently construct a basis for G, ideally using no more than $T(\lambda(G))$ group operations. Given a bound on the size of $|G|$, we will be able to accomplish this in almost every case, resulting in a dramatically faster generic algorithm for abelian group structure (see performance results in Chapter 11). Even without a bound on $|G|$, for almost all groups the expected running time will be, at worst, approximately $\sqrt{2|G|}$, independent of the size of the basis.

The exponent $\lambda(G)$ gives us a wealth of information regarding the structure of G. It tells us which primes divide $|G|$, and allows us to readily obtain elements in the p-Sylow subgroups of G, which we denote H_p. The subgroup H_p consists of the elements in G whose order is a power of p. The decomposition of G into cyclic factors of prime-power order may be partitioned into decompositions of the H_p. The utility of $\lambda(G)$ is that it not only tells us which H_p exist, it gives us $\lambda(H_p)$ (the largest power of p dividing $\lambda(G)$), and it allows us to generate uniformly random elements of H_p by computing $\alpha^{\lambda(G)/\lambda(H_p)}$ for random $\alpha \in G$. Alternatively, given a set of generators for G, we can use the same method to obtain a set of generators for H_p. To compute the structure of G, it suffices to compute the structure of each H_p.

Breaking the problem down in this way has two distinct advantages. The first is that working in a group of prime-power order makes the process of computing a basis substantially simpler: it is easier to find minimal relations, and computing the Smith normal form of the relation matrix is easier when all the elements involved have orders that are powers of p. In fact, we will find we can avoid matrix operations entirely. The second advantage is that the groups H_p will often be much smaller than G. Note that this is true even when G is cyclic, provided it has composite order.

9.3 Discrete Logarithms

The main obstacle that remains is the discrete logarithm problem. In order to determine the structure of H_p, we need to be able to test when a set of elements is independent, and when it is not, to find relations. This will typically involve computing the vector form of the discrete logarithm, $DL(\vec{\alpha}, \beta)$.

In [96] Shoup shows that for a generic algorithm, the complexity of computing $DL(\alpha, \beta)$ in a group of prime order p is $\Omega(\sqrt{p})$. In the same paper, Shoup proves a similar bound for

$DL\left(\vec{\alpha}, \beta\right)$. When $\vec{\alpha}$ consists of $r > 1$ elements, all with order p, the complexity of computing $DL\left(\vec{\alpha}, \beta\right)$ is $\Omega\left(p^{r/2}\right)$. Somewhat surprisingly, the latter bound is the more problematic of the two for our purposes. Happily, this case arises infrequently, as G rarely contains a large subgroup H_p whose order is divisible by p^2. Let us consider how we may avoid computing $DL(\alpha, \beta)$ for elements with large prime order, given just a little information about $|G|$.

If $\lambda(G)$ contains a large prime factor p, then either p^2 divides $|G|$, in which case $\sqrt{p} = O\left(|G|^{1/4}\right)$, or $\lambda(G)$ and $|G|$ are approximately the same size. If we have even a weak upper bound on $|G|$, say, $M = O\left(G^{3/2}\right)$, then by comparing p^2 to M we can determine whether it is possible for p^2 to divide $|G|$. If p is large relative to $|G|$, say $p > |G|^{2/3}$, then p^2 will be greater than M, in which case H_p must be a cyclic group with order p, and any non-trivial element will serve as a generator.

It is possible that a suitable bound M may be determined simply from the size of the identifiers, if the black box is reasonably efficient in its encoding of group elements. In practice it is typically possible to obtain M by other means, usually with a bound tighter than $|G|^{3/2}$. In the case of ideal class groups, we know $M = O\left(|G| \log |G|\right)$ (see Chapter 10), and for groups of points on elliptic curves over the field $\mathbb{F}_q$, we have the tighter bound given by Hasse's theorem [57]: $|G| \leq M = q + 2\sqrt{q} + 1$.

9.4 Abelian Group Structure Algorithm

We now present a probabilistic generic algorithm for computing a prime-power basis of an abelian group G (see Remark 9.1 regarding a deterministic version). Whenever a random element of H_p is called for, a random element of G is obtained from the black box and $\alpha^{\lambda(G)/\lambda(H_p)}$ is computed. The parameter c specifies the confidence level requested in the output for the probabilistic case. The algorithm is designed to ensure that the output is correct with probability at least $1 - 2^{-c}$, provided the black box returns uniformly random elements of G. The bound M is optional and may be set to ∞. If at any point during the algorithm's execution it determines that the existence of any new independent elements would necessarily result in $|G| > M$, the algorithm terminates and returns the basis it has computed. The functions $Expand(\vec{\alpha}, \beta, \vec{r})$ and $DL\left(\vec{\alpha}, \beta\right)$ are computed by Algorithms 9.2 and 9.3 which follow. The notation $\vec{a} \leftarrow \vec{b} \circ \vec{c}$ indicates that the j-place vector $\vec{b}$ and the k-place vector $\vec{c}$ should be concatenated to obtain a $(j + k)$-place vector $\vec{a}$. We use the same notation

when $\vec{c}$ is a single element.

Algorithm 9.1 (Abelian Group Structure)**.** *Given a randomized black box for an abelian group G, an integer $c > 1$, and a bound $M \geq |G|$, the algorithm below returns a basis $\vec{\gamma} = [\gamma_1, \ldots, \gamma_n]$ and a vector $\vec{m} = [m_1, \ldots, m_n]$, where $m_i = |\gamma_i|$ is a prime power.*

1. **Exponent**: Use Algorithm 8.1 to compute $\lambda(G)$ with parameter $c + 2$, then factor the integer $\lambda(G)$. Choose p to minimize $\lambda(H_p)$ and set $\vec{\gamma} \leftarrow \emptyset$.

2. **Start H_p**: Let $\vec{\alpha} = (\alpha_1)$ be a non-trivial random element of H_p. Set $c_p \leftarrow \lceil \log_p 2^c \rceil$.

3. **Sample**: If $p|\langle \vec{\gamma} \circ \alpha \rangle| \leq M$, attempt to compute $DL\left(\vec{\alpha}, \beta\right)$ for c_p random elements $\beta \in H_p$, and whenever a failure occurs, proceed to step 4. Otherwise go to step 6.

4. **Get Relation**: Determine the least $q = p^j > 1$ for which $\vec{e} = DL(\vec{\alpha}, \beta^{-q})$ exists. If $q = |\beta|$, set $\vec{\alpha} \leftarrow \vec{\alpha} \circ \beta$ and go to step 3, otherwise set $\vec{r} \leftarrow \vec{e} \circ q$ and continue.

5. **Expand Basis**: Set $\vec{\alpha} \leftarrow Expand(\vec{\alpha}, \beta, \vec{r})$ and go to step 3.

6. **End H_p**: Set $\vec{\gamma} \leftarrow \vec{\gamma} \circ \vec{\alpha}$. Update p to correspond to the next largest value of $\lambda(H_p)$, if any, and go to step 2, otherwise return $\vec{\gamma}$ and $\vec{m}$ with $m_i = |\gamma_i|$.

The first step of the algorithm computes the factored exponent $\lambda(G)$. Thus all element orders required by the algorithm or its subroutines (in particular $DL\left(\vec{\alpha}, \beta\right)$) may be obtained via fast order computations. In practical implementations it may be appropriate to store computed orders with the corresponding group elements to avoid recomputing them unnecessarily. Random elements from H_p are sampled in step 2 using $\lambda(G)$ as described above. Note the computation of c_p; this parameter effectively determines the number of discrete logarithm operations that must be performed in order for the algorithm to be confident that it has a basis that generates (spans) the entire subgroup H_p. If p is large, there is no need to retry c times, the much smaller value c_p will suffice. If $\vec{\alpha}$ does not generate H_p, then the subgroup $\langle \vec{\alpha} \rangle$ contains at most a $1/p$ fraction of the elements in H_p. A random $\beta \in H_p$ is unlikely to be a member of $\langle \vec{\alpha} \rangle$, resulting in the failure of $DL\left(\vec{\alpha}, \beta\right)$ and an enlargement of $\langle \vec{\alpha} \rangle$.

The function $Expand(\vec{\alpha}, \beta, \vec{r})$ returns a basis for the subgroup $\langle \vec{\alpha}, \beta \rangle$, which is necessarily larger than $\langle \vec{\alpha} \rangle$ by a factor of at least p (although the new basis need not contain more

elements). The total number of iterations of the algorithm is at most $\lg|G|$. The running time of the algorithm is essentially dominated by the cost of the largest discrete logarithm operation which depends on the size of the largest H_p and on the bound M. We consider this further in the next section (see Proposition 9.5), but first we complete the presentation of the algorithm.

Remark 9.1. *If a set of generators $S \subseteq G$ is available, Algorithm 9.1 can be made deterministic by replacing random elements with an enumeration of the set S for each H_p. This is worth doing if it is believed that*

$$\sum_{p|\lambda(G)} \left|\left\{t : t = s^{\lambda(G)/\lambda(H)} \neq 1_G, s \in S\right\}\right| < \sum_{p|\lambda(G)} (c_p + r_p), \tag{9.1}$$

where r_p is the rank (number of cyclic factors) of H_p. On the other hand, if S is large it may be more efficient to use random elements (unless a deterministic result is required), either from a randomized black box, or by taking random combinations of elements in S.

9.5 Expanding a Basis in a p-Group.

The function $Expand(\vec{\alpha}, \beta, \vec{r})$ takes an independent set $\vec{\alpha}$ in H_p and a group element $\beta \in H_p$ not in $\langle\vec{\alpha}\rangle$, and uses the relation $\vec{r}$ to construct a basis for the subgroup $\langle\vec{\alpha}, \beta\rangle$. Note that $\vec{\alpha}$ is necessarily a basis for $\langle\vec{\alpha}\rangle$, so this may be viewed as expanding the basis to a larger subgroup of H_p.

This process can be accomplished by computing the Smith normal form of an appropriate matrix, however, because we are working in a p-group (every element's order is a power of p), a simpler and more efficient approach is possible.

Any relation $\vec{r}$ on a vector $\vec{\alpha}$ of elements of H_p (in an abelian group) can easily be converted into a relation $\vec{r_*}$ on some $\vec{\alpha_*}$, where $\vec{\alpha_*}$ generates the same subgroup as $\vec{\alpha}$ and the non-zero entries in $\vec{r_*}$ are all powers of p. This is useful because it means that for any non-zero pair of entries in $\vec{r_*}$, one divides the other, and there is some non-zero entry which divides every entry. This effectively eliminates most of the transformations typically required when computing the Smith normal form, enabling us to work directly with $\vec{r_*}$.

We assume that the orders of all group elements are known, and that relations contain non-negative integers reduced modulo the orders of the corresponding elements (if not, this can be done in step 1).

Algorithm 9.2 (Expand Basis)**.** *Given a basis $\vec{\alpha} = [\alpha_1, \ldots, \alpha_{k-1}]$ for a p-group in an abelian group G, an element $\beta \in G - \langle\vec{\alpha}\rangle$ with order $p^h > 1$, and a k-place relation $\vec{r}$ on $\vec{\alpha} \circ \beta$ with $r_k > 1$ minimal, the recursive algorithm Expand$(\vec{\alpha}, \beta, \vec{r})$ returns a basis $\vec{\gamma}$ for the subgroup $\langle\vec{\alpha}, \beta\rangle$.*

1. **Construct p-relation**: For each non-zero r_i not a power of p, let q be the largest power of p dividing r_i and set $\alpha_i \leftarrow \alpha_i^{r_i/q}$ and $r_i \leftarrow q$.
2. **Adjust β?**: If r_k divides every r_i, set $\beta \leftarrow \beta \prod_{1 \le i < k} \alpha^{r_i/r_k}$ and return $\vec{\gamma} = \vec{\alpha} \circ \beta$.
3. **Adjust $\vec{\alpha}$**: Choose a least r_j which divides every r_i and set $\alpha_j \leftarrow \beta^{r_k/r_j} \prod_{1 \le i \le k} \alpha^{r_i/r_j}$.
4. **Reduce?**: If $\alpha_j = 1_G$, remove α_j from $\vec{\alpha}$. If $\vec{\alpha} = \emptyset$, return $\vec{\gamma} = [\beta]$.
5. **New Relation**: Determine the least $q = p^h > 1$ for which $\vec{e} = DL(\vec{\alpha}, \beta^{-q})$ exists. If $q = |\beta|$, return $\vec{\gamma} = \vec{\alpha} \circ \beta$.
6. **Recurse**: Set $\vec{r} \leftarrow \vec{e} \circ q$ and return *Expand*$(\vec{\alpha}, \beta, \vec{r})$.

The algorithm is guaranteed to terminate in less than k recursive calls, since any time it recurses it has reduced k by 1 and will not recurse when $k = 2$. To prove the correctness of Algorithm 9.2, and to prove that Algorithm 9.1 actually minimizes r_k, we need the following lemma.

Lemma 9.1. *If $\vec{\alpha}$ is a basis for a p-group in G with $k-1$ elements and $\beta \in H_p - \langle\vec{\alpha}\rangle$, then the least integer r_k for which a relation $\vec{r}$ on $\vec{\alpha} \circ \beta$ exists, is a power of p.*

Proof. Let $\vec{r}$ be a relation on $\vec{\alpha} \circ \beta$ with r_k minimal. Suppose for the sake of contradiction that $r_k = p^j s$ for some $s \perp p$. Let t be the multiplicative inverse of s modulo q, where $q = |\beta|$ is a power of p. Then for some integer m,

$$(\beta^{r_k})^t = \beta^{p^j st} = \beta^{p^j(mq+1)} = (\beta^q)^{p^j mq}\beta^{p^j} = \beta^{p^j}.$$

The relation $\vec{r}$ implies that $\beta^{r_k} \in \langle\vec{\alpha}\rangle$, and so is β^{p^j}, since it is a power of β^{r_k}. But then there exists a relation on $\vec{\alpha} \circ \beta$ with $p^j < r_k$ as its k-th entry, since $\vec{\alpha}$ is a basis for $\langle\vec{\alpha}\rangle$, contradicting the minimality of r_k. □

Proposition 9.2. *Given inputs $\vec{\alpha}$, β, and $\vec{r}$ satisfying the conditions of Algorithm 9.2, the algorithm returns a basis for the subgroup $\langle\vec{\alpha}, \beta\rangle$.*

Proof. We consider the algorithm step by step.

Step 1: Note that r_i/q is relatively prime to $|\alpha_i|$, since it is not divisible by p, hence $\alpha_i^{r_i/q}$ has the same order as α_i and $\langle \alpha_i^{r_i/q} \rangle = \langle \alpha_i \rangle$. It follows that at the end of step 1, $\vec{\alpha}$ is still a basis for $\langle \vec{\alpha} \rangle$, and the relation is updated correctly, since $\left(\alpha_i^{r_i/q}\right)^q = \alpha_i^{r_i}$. Lemma 9.1 implies that r_k must be a power of p, so r_k and β are unaffected by step 1.

Step 2: Let β_* be the updated value of β. Then $\beta \in \langle \vec{\alpha}, \beta_* \rangle$, and $\vec{\gamma} = \vec{\alpha} \circ \beta_*$ generates the subgroup $\langle \vec{\alpha}, \beta \rangle$. The vector $\vec{\gamma}$ is independent, since $\vec{\alpha}$ is and any relation $\vec{s}$ on γ would have $0 < s_k < |\beta_*| = r_k$, but then $\beta^{s_k} \in \langle \vec{\alpha} \rangle$, which contradicts the minimality of r_k.

Step 3: Let α_{j*} be the updated value of α_j and $\vec{\alpha_*}$ the updated value of $\vec{\alpha}$. We have $\alpha_j \in \langle \vec{\alpha_*}, \beta \rangle$ so $\langle \vec{\alpha_*}, \beta \rangle = \langle \vec{\alpha}, \beta \rangle$. The vector $\vec{\alpha_*}$ is independent, since a relation $\vec{s}$ on $\vec{\alpha_*}$ would have $1 < s_j < r_j$, but this implies $(\beta^{r_k/r_j})^{s_j} \in \langle \vec{\alpha} \rangle$ contradicting the minimality of r_k.

Step 4: Removing 1_G from $\vec{\alpha}$ does not change the subgroup $\langle \vec{\alpha}, \beta \rangle$, and if $\vec{\alpha} = \emptyset$ then $\gamma = [\beta]$ is clearly a basis.

Step 5: Lemma 9.1 implies that it suffices to check for relations on β^q with q a prime power. If $q = |\beta|$ then $\gamma = \vec{\alpha} \circ \beta$ is independent and clearly generates $\langle \vec{\alpha}, \beta \rangle$.

Step 6: The recursive call is correct by induction; we have effectively handled the base cases above. □

9.6 Vector Form of the Discrete Logarithm

A generic algorithm computing $DL\left(\vec{\alpha}, \beta\right)$ attempts to find an exponent vector $\vec{e}$ for β relative to the independent vector $\vec{\alpha}$. It will be successful if and only if $\beta \in \langle \vec{\alpha} \rangle$, and if it succeeds then $\vec{e} \circ 1$ is a relation on $\vec{\alpha} \circ \beta^{-1}$.

We use a variant of the baby-steps giant-steps method to compute $DL\left(\vec{\alpha}, \beta\right)$, as shown in Algorithm 9.3. The rho method may also be used, as in [109] (see Remark 9.3).

The algorithm is based on stepping through the set of possible exponent vectors for β by considering them as numbers in a mixed-radix representation. The low order digits count baby steps, and the high order digits count giant steps. The only minor inconvenience occurs in the "center" digit, which is at position c in Algorithm 9.3 below. The possible values of the digit e_c are also searched in standard baby-step giant-step fashion, with each giant step advancing by s. It is critical that the maximum value $m_c - 1$ occurs as a giant step each time the digit "rolls-over", so this must be handled explicitly.

Algorithm 9.3 (Vector Discrete Logarithm)**.** *Given a basis $\vec{\alpha} = [\alpha_1, \ldots, \alpha_k]$ for an abelian subgroup of G, with $m_i = |\alpha_i|$, and $\beta \in G$, the following algorithm computes $\vec{e} = DL\left(\vec{\alpha}, \beta\right)$:*

1. **Initialize**: Let $M = \prod m_i/2$ and choose c such that $\prod_{i<c} m_i \leq \sqrt{M} < \prod_{i \leq c} m_i$. Choose s such that $s \prod_{i<c} m_i \leq \sqrt{M} < (s+1) \prod_{i<c} m_i$.

2. **Baby Steps**: Compute $\delta \leftarrow \alpha_1^{e_1} \cdots \alpha_c^{e_c}$ with integers $e_i \in [0, m_i)$ for $i < c$ and $e_c \in [0, s)$. Store each δ in a lookup table together with the vector $\vec{e_\delta} = [e_1, \ldots, e_c]$.

3. **Giant Steps**: Compute $\gamma \leftarrow \beta^{-1} \alpha_c^{e'_c} \alpha_{c+1}^{e_{c+1}} \cdots \alpha_k^{e_k}$ where e'_c ranges over integer multiples of s in $[0, m_c)$ and the value $m_c - 1$, and the e_i range over integers in $[0, m_i)$ for $i > c$. Lookup each γ in the table, and if $\gamma = \delta$, return $\vec{e} = [e_1, \ldots, e_{c-1}, e^*_c, e_{c+1}, \ldots, e_k]$, where $\vec{e_\delta} = [e_1, \ldots, e_c]$ and $e^*_c = e'_c + e_c \bmod m_c$. If no such γ is found, return $\emptyset$.

The giant steps can almost all be accomplished with a single multiplication by $\delta = \alpha_c^s$. The element δ should be computed just once, along with a δ' for handling the step to $\alpha_c^{m_c - 1}$ at the end of the digit range. It is best to arrange the inputs so that the m_i are in increasing order to minimize the number of "carry" operations, since each carry requires an extra group operation.

Remark 9.2. *Algorithm 9.3 optimizes performance over uniformly distributed exponent vectors, since it is designed for use by the randomized group structure algorithm. To optimize worst-case performance, set $M = \prod m_i$ rather than $M = \prod m_i/2$ in step 1.*

Remark 9.3. *We present a baby-steps giant-steps algorithm because it is simpler and slightly faster than the rho method. When an upper bound on $|G|$ is available, as in Proposition 9.5, we can generally avoid large discrete log computations and the space requirements do not pose a problem. When this is not true and the problem is large, the rho method should be used.*

9.7 Complexity Analysis

We now consider the complexity of Algorithm 9.1, starting with the worst case scenario. This occurs when $G = H_2$, that is, every element of G has order a power of 2. In this situation, computing $\lambda(G)$ is easy, requiring at worst $O\left(|G|^{1/4}\right)$ group operations using the multi-stage sieve, and $O\left(\log |G|\right)$ group operations if the primorial-steps algorithm is used directly. Unfortunately, in this scenario there may be no way to avoid some large discrete

log operations, even when a tight bound on $|G|$ is known. If $M \leq 2|G|$ we may only need one large discrete log operation, but in general we will need to perform c unsuccessful discrete log operations to be satisfied that we have found all of G, thus the complexity is $O\left(\sqrt{|G|}\right)$, but the constant factor will be roughly c.[1] A similar situation applies whenever G consists almost entirely of a single Sylow subgroup H_p, however, the constant factor quickly improves as p grows.

In the propositions below, we use an order complexity bound $T(N)$, defined in 8.2, to bound the complexity of all order computations performed by Algorithm 9.1 by $T\big(\lambda(G)\big)$, applying Theorem 8.9. The total number of fast order computations will be logarithmic in $|G|$, making their cost negligible.

Proposition 9.3. *Let $T(N)$ be an order complexity bound and let H_p be the largest Sylow subgroup of the abelian group G. Then the expected number of group operations used by Algorithm 9.1 with input parameter c is bounded by*

$$\left(\left\lceil \frac{c}{\lg p} \right\rceil + \frac{1}{\sqrt{p}-1} + \frac{1}{\sqrt{p}(p-1)}\right)\sqrt{2|H_p|} + \big(1+o(1)\big)T\big(\lambda(G)\big) + O\left(|G|^{1/4}\right).$$

Proof. It follows from Theorem 8.9 that the complexity of all the order computations performed by Algorithm 9.1 may be bounded by

$$\big(1+o(1)\big)T\big(\lambda(G)\big) + O\left(|G|^{1/4}\right). \tag{9.2}$$

Thus we need only consider the cost of computing discrete logarithms (the other group operations are negligible). Let H_p be the largest Sylow subgroup of G. Every other Sylow subgroup must have size less than $|G|^{1/2}$, and the total cost of all the discrete logarithms computed in these subgroups is $O\left(|G|^{1/4}\right)$, thus we only consider H_p.

Each unsuccessful call to $DL(\vec{\alpha}, \beta)$ enlarges the subgroup spanned by $\vec{\alpha}$ by a factor of at least p. The total cost of all successful calls may be bounded by

$$\sqrt{2|H_p|/p} + \sqrt{2|H_p|/p^2} + \cdots + 1 \leq \left(\frac{1}{\sqrt{p}-1}\right)\sqrt{2|H_p|}. \tag{9.3}$$

Algorithm 9.1 makes c_p successful calls to $DL(\vec{\alpha}, \beta)$ with $\vec{\alpha}$ a basis for H_p, and at most an expected $1/p^k$ successful calls with $\vec{\alpha}$ a basis for a subgroup of size $|H_p|/p^k$ for $1 \leq k \leq \log_p H_p$.

[1] If a set of generators is used the complexity will be $O\left((|S|-r)\sqrt{|G|}\right)$ where r is the rank of H_p.

The total cost of all unsuccessful calls is thus bounded by

$$c_p \sqrt{2|H_p|} + \frac{1}{p}\sqrt{2|H_p|/p} + \frac{1}{p^2}\sqrt{2|H_p|/p^2} + \cdots + 1 \leq \left(c_p + \frac{1}{\sqrt{p}(p-1)}\right)\sqrt{2|H_p|} \tag{9.4}$$

From step 2 of Algorithm 9.1, we have $c_p = \lceil log_p 2^c \rceil = \lceil c/\lg p \rceil$, and the proposition follows by summing (9.2), (9.3), and (9.4). □

In most situations, the largest prime dividing $|G|$ is not bounded by a constant, and the bound in Proposition 9.3 is at most

$$\big(1 + o(1)\big)\sqrt{2|G|}, \tag{9.5}$$

since $T(\lambda(G)) = o\left(\sqrt{|G|}\right)$ when the primorial-steps algorithm is used for order computations. We also have the following explicit bounds:

Corollary 9.4. *Let G by an abelian group and assume $c \geq 3$. If $|G|$ is divisible by a prime greater than 2^c then Algorithm 9.1 uses less than $\big(2 + o(1)\big)\sqrt{2|G|}$ group operations. If $|G|$ is divisible by two primes greater than c^2, then Algorithm 9.1 uses less than $\big(1 + o(1)\big)\sqrt{|G|}$ group operations.*

We now consider the more attractive scenario where we have a rough bound M on the size of G. Any bound $M = O\left(|G|^{2-\epsilon}\right)$ will be useful, provided that G does not contain a large p-Sylow subgroup for some small p.

Proposition 9.5. *Let $T(N) = O\left(N^\delta\right)$ be an order complexity bound with $1/4 \leq \delta \leq 1/2$. Given $M = O\left(|G|^{4\delta}\right)$, the complexity of Algorithm 9.1 is either polynomial in $\lg|G|$ or bounded by*

$$\big(1 + o(1)\big)T\big(\lambda(G)\big),$$

provided that $|H_p| = o\left(T^2\big(\lambda(G)\big)\right)$ for all $p \leq |G|^{2\delta}$, where H_p denotes a p-Sylow subgroup of the abelian group G.

Proof. By Theorem 8.9, all the order computations fit within the specified bound, assuming that $T\big(\lambda(G)\big) = \Omega\left(\lg^2|G|\right)$ covers all the fast order computations. If this is not the case then $\lambda(G)$ is polylogarithmic in $|G|$, as is every p dividing $|G|$ and every $|H_p|$, by the hypothesis of the proposition. In this scenario, the complexity of Algorithm 9.1 is polynomial in $\lg|G|$.

If $p \leq |G|^{2\delta}$, then the hypothesis of the proposition ensures that the cost of discrete logarithm computations in H_p is dominated by $T\big(\lambda(G)\big)$, so assume $p > |G|^{2\delta}$. Then $p^2 > M$,

and Algorithm 9.1 never needs to perform a discrete logarithm operation in H_p, since the first non-trivial element found will be known to generate H_p. □

Remark 9.4. *In almost every "naturally" arising distribution of abelian groups, including ideal class groups, elliptic curves, and the integers modulo m, the conditions of Proposition 9.5 are met (or believed to be met) by all but an exponentially small fraction of the groups with orders of a given size. For a "random" group from any of these distributions, the proposition will almost certainly apply. See Chapter 11 for empirical evidence to support this claim.*

Any of the complexity bounds for the primorial-steps and multi-stage sieve algorithms derived in Chapters 4 and 5 may be used for $T(N)$ in Proposition 9.5. If the rho method is used, both as the search algorithm in the multi-stage sieve and to compute discrete logarithms, the total space required by Algorithm 9.1 is polylogarithmic in $|G|$. The computationally intensive tasks can then be performed in parallel using the distinguished-point cycle detection method described in Section 3.1, enabling very large problem sizes to be addressed.

9.8 Computing Group Structure Given an Exponent of G

Proposition 9.5 shows that, given a bound on $|G|$, the complexity of Algorithm 9.1 is essentially determined by the complexity of a single order computation in most situations. It is of interest to consider the complexity of Algorithm 9.1 independently, as there are many circumstances where $\lambda(G)$ or a multiple of $\lambda(G)$ is known and one would like to compute the structure of G. Algorithm 9.1 will work correctly without modification given any multiple of $\lambda(G)$, and, in particular, the value $|G|$ may be used. When $|G|$ is known, this also gives a tight value for M. In many circumstances there may be (non-generic) methods of efficiently determining $|G|$. It is then possibly to compute the structure of G using $O\left(|G|^{1/4}\right)$ group operations in most cases.

Proposition 9.6. *Given E, a multiple of $\lambda(G)$ satisfying $\lg E = O\left(\lg |G|\right)$, and M, an upper bound on $|G|$ with $M = O\left(|G|^{\delta}\right)$ for some $r \geq 1$, the number of group operations required to compute the structure of the abelian group G is*

$$O\left(|G|^{\delta/4}\right),$$

provided that for all $p \leq \sqrt{M}$, the p-Sylow subgroup H_p satisfies $|H_p| = O\left(M^{1/2}\right)$.

Proof. Let p be the largest prime divisor of E for which a non-trivial element $\alpha \in H_p$ is found by Algorithm 9.1. If $p^2 > M$ then α generates H_p and no discrete logarithms need be computed. If $p \leq \sqrt{M}$ then, by hypothesis, $|H_p| = O\left(M^{1/2}\right)$, which implies that the cost of the largest discrete logarithm computed in H_p will use $O\left(M^{1/4}\right)$ group operations. The cost of this discrete logarithm will represent a constant fraction of all the group operations spent on discrete logarithms, and dominates the cost of all other group operations. □

Corollary 9.7. *Let G be an abelian group with known order $|G|$ divisible by a prime larger than $|G|^{1/2}$. The structure of G can be computed using $O\left(|G|^{1/4}\right)$ group operations.*

Computing the structure of G, once $|G|$ is known, is a strictly easier problem than order computation in most cases. Typically, the performance of Algorithm 9.1 will be dictated by the size of the second largest prime dividing $|G|$, which will likely be substantially less than $|G|^{1/2}$. This leads to a better than $|G|^{1/4}$ running time, as seen in many of the group structure computations for ideal class groups presented in Chapter 11.

Chapter 10

A Number-Theoretic Postlude

Chapter Abstract

This chapter briefly recollects some facts and conjectures about the ideal class groups of imaginary quadratic number fields.

10.1 The Class Group of a Number Field

A *number field* is a subfield of the complex numbers which has finite degree as an extension of $\mathbb{Q}$. In any number field K, the elements of K which are algebraic integers (zeroes of a monic polynomial with integer coefficients) form a commutative ring R which is a Dedekind domain.[1] Recall that an *ideal* of a ring R is an additive subgroup which is closed under multiplication by elements of R. A product of ideals AB is the ideal generated by $\{\alpha\beta : \alpha \in A, \beta \in B\}$. We may define an equivalence relation on ideals by

$$A \equiv B \qquad \Longleftrightarrow \qquad \langle\alpha\rangle A = \langle\beta\rangle B,$$

where $\langle\alpha\rangle$ denote $\langle\beta\rangle$ denote principal ideals (ideals generated by a single element). In a Dedekind domain (and in particular, in the ring of algebraic integers of any number field) the induced product operation on equivalence classes of ideals is a well defined group operation with identity $\overline{\langle 1\rangle}$ corresponding to the class of all principal ideals. This defines the *ideal class group* of a Dedekind domain, or we may simply refer to the *class group* of a

[1] A Dedekind domain may be defined as a commutative ring without zero divisors (an integral domain) in which every proper ideal can be factored (uniquely) into prime ideals.

number field, with the ring of algebraic integers implicit.

10.2 Binary Quadratic Forms

In the case of quadratic number fields $\mathbb{Q}[\sqrt{D}]$ with $D < 0$, there is a particularly simple representation of the class group via quadratic forms

$$ax^2 + bxy + cy^2.$$

If D is congruent to 0 or 1 modulo 4, the class group of $\mathbb{Q}[\sqrt{D}]$ may be represented by equivalence classes of forms with discriminant $b^2 - 4ac = D$. Two forms are said to be equivalent if the set of integers represented by the form as x and y range over $\mathbb{Z}$ are the same. There is a one-to-one correspondence between quadratic forms and ideals of $\mathbb{Q}[\sqrt{D}]$ for which the two notions of equivalence correspond, and the group operation on ideal classes may be realized directly by integer operations on the coefficients of representative forms.

The key feature of forms with negative discriminant is that each class of forms has a unique representative with coefficients (a, b, c) where $-a < b \leq a \leq c$ and $\gcd(a, b, c) = 1$. Moreover, any form can be efficiently reduced to its canonical representative, allowing the class group to be easily implemented in a black box. Efficient algorithms for the reduction of forms and the group operation on forms can be found in [95] and in [37] (see also [31] or [40]). The only potential issue that arises is the efficient generation of random group elements. An expedient solution is to use random *primeforms*. A random prime a is chosen for which D is a quadratic residue so that b may be computed as the square root of D modulo $4a$ (c is then determined by $D = b^2 - 4ac$). It is known that the primeforms generate the class group, but not every canonical representative of the class group is necessarily a primeform, so this does not always give a uniform distribution over G. Using random primeforms is much more efficient than generating a truly random element of G, and it seems to work well in practice.[2]

[2]It may, however, be appropriate to repeat a probabilistic algorithm a greater number of times in order to obtain the same level of confidence in its results.

10.3 Class Groups of Imaginary Quadratic Number Fields

The investigation of the class groups of imaginary quadratic number fields is a fascinating topic which dates back to Gauss and remains an active field of research today. We list here some of the more significant results, along with a brief sampling of the many open conjectures and unsolved problems that remain. To simplify the discussion, we restrict ourselves to negative *fundamental discriminants* D, those which are either square free and congruent to 1 modulo 4, or $D = 4D'$ where D' is square free and congruent to 2 or 3 modulo 4. Results for non-fundamental discriminants can be easily derived using the corresponding fundamental discriminant. We use the notation $Cl(D)$ to denote the class group of the number field $\mathbb{Q}[\sqrt{D}]$, and let $h(D) = |Cl(D)|$ denote the class number.

1. There are exactly nine imaginary quadratic fields whose class group is trivial, corresponding to the discriminants -3, -4, -7, -8, -11, -19, -43, -67, and -163. The rings of algebraic integers corresponding to these fields are unique factorization domains (and the only such). This was conjectured by Gauss but not finally settled until 1967 by Stark [101] based on earlier work by Heegner [59].

2. The Dirichlet class number formula can be used to estimate (or actually compute) $h(D)$. For $D < -4$ the formula states

$$h(D) = \frac{1}{\pi} L(1, \chi_D) \sqrt{|D|}, \tag{10.1}$$

   where $L(1, \chi_D) = \prod_p (1 - \chi_D(p)/p)^{-1}$ and $\chi_D(p) = \left(\frac{D}{p}\right)$ is given by the Kronecker symbol. The upper bound $h(D) \leq \pi^{-1} \sqrt{|D|} \log |D|$ can be derived from this formula, and it is known that the average value of $h(D)$ is asymptotic to $C\pi^{-1} \sqrt{|D|}$, where $C \approx 0.8815$ [37]. Tighter bounds can be obtained using the extended Riemann Hypothesis (ERH), within an $O\left(\log \log |D|\right)$ factor of $\sqrt{|D|}$ [31]. This has practical implications, since (10.1) can be computed for all the primes up to a chosen bound to get an estimate of $h(D)$ (see Section 11.4).

3. There is a one-to-one correspondence between the elements of order 2 in the group $Cl(D)$ and pairs of relatively prime factors of D. These elements are represented by "ambiguous forms" for which the coefficients a, b, and c can be used to factor D. This

fact is the basis of a number of integer-factorization algorithms [95], including one of the first subexponential algorithms [92] (see also the SPAR algorithm).

4. While the 2-Sylow subgroup of $Cl(D)$ is intimately related to the multiplicative structure of D, the odd part of $Cl(D)$, $Cl_o(D)$, is conjectured to be a "random" abelian group G with odd order. The Cohen-Lenstra heuristics are based (roughly speaking) on the assumption that G is chosen from a distribution in which each isomorphism class is weighted by the factor $1/|\mathrm{Aut}(G)|$ (see [38]). This has a number of interesting consequences which match the observed data quite well:

   a. $Cl_o(D)$ is cyclic with probability ≈ 0.977575.

   b. $h(D)$ is divisible by an odd prime p with probability $p^{-1} + p^{-2} - p^{-5} - p^{-7} + \ldots$.

5. Under the extended Riemann Hypothesis, it is known that the exponent of the group $Cl(D)$ tends to infinity [19] as $D \to -\infty$. This implies that for any integer e, $\lambda\big(Cl(D)\big) = e$ for only finitely many D and that, in particular, there is some largest $|D|$ for which this holds. This has been proven unconditionally for $e = 2$, 3, and 4 [19, 36, 45]. More generally, it has been shown unconditionally that there are only finitely many D for which the exponent of $Cl(D)$ is a power of 2 [45]. These proofs are generally ineffective; they do not give explicit bounds, hence we have no way of knowing for certain when we have found the "last" class group with a given exponent.

The list above is but a sample. We refer the reader to [38, 37, 31] for more information and further references. Class groups have also been proposed as the basis for a number of cryptographic schemes, see [25, 56].

Class groups of imaginary quadratic number fields make an excellent test bed for generic algorithms. They admit an efficient black box implementation, yet for any large discriminant, the precise group structure is generally unknown and can only be determined via group operations. Indeed, the problem of computing the class group was the motivation for some of the very first generic algorithms, including the baby-steps giant-steps method of Shanks [95]. Class groups have been extensively studied, so the results of computations in class groups can be externally verified and performance comparisons to existing algorithms can be made.

The odd part of the class group appears to give an appropriate random distribution of finite abelian groups with odd order. Note that if the discriminant is prime, the class group necessarily has odd order (by (3) above), so it is possible to sample class groups with purely odd part. For composite discriminants, the 2-Sylow subgroup tends to have large rank, a useful test case for group structure algorithms.

We should note that there are non-generic methods of computing both the class number and the class group [37, 26, 95]. These are considered in Section 11.4. The performance results discussed in Chapter 11, and the tables listed in Appendix B were obtained using generic algorithms with no knowledge of class groups other than an explicit unconditional bound on the size of the group based on (2) above.

Chapter 11

Performance Results

Chapter Abstract

Tests of various distributions of random finite groups find the median performance of the multi-stage sieve to be comparable to or better than $O\left(N^{0.34}\right)$ in most cases. Comparisons to existing generic algorithms based on both the baby-steps giant-steps and rho methods show dramatic improvement. For large class groups of size N with negative quadratic discriminant D, the median performance was close to $O\left(N^{0.27}\right)$, approximately $O\left(|D|^{1/7}\right)$. Comparisons to non-generic subexponential algorithms for class group computation are also favorable in many cases. Finally, we consider an application of the multi-stage sieve with subexponential complexity, and use it compute several class groups with 100-digit discriminants.

Before presenting the performance results, some introductions are in order. The generic algorithms presented in this thesis were developed and tested using seven different black boxes. The black boxes were implemented using the GNU Multi-Precision library [53], which provides an efficient implementation of arithmetic operations on arbitrarily large numbers, as well as various number-theoretic functions and the generation of pseudo-random numbers. The black boxes are listed below.

- **Cyclic**: The group C_n, represented by the additive group of integers $\mathbb{Z}_n^+$. The main virtue of this black box is its speed, useful for large test cases.
- **Integer**: The multiplicative group of integers $\mathbb{Z}_n^*$. This black box permits both cyclic and non-cyclic groups to be tested, according to whether n is prime or not.
- **Product**: This black box implements an arbitrary finite abelian group as a direct product of cyclic groups, each represented as an additive group of integers. This black

box is useful in exercising specific test cases for abelian group structure algorithms and general debugging.

- **Permutation**: The group S_n, represented by arrays containing permutations of the integers from 1 to n. This black box is useful for testing algorithms in a highly non-commutative group.

- **2-d Matrix**: The group $GL_2(\mathbb{F}_p)$, represented as 2x2 non-singular matrices over the integers modulo p. This black box was used as a non-abelian test case for group exponent computations.

- **Elliptic Curve**: The group $E(\mathbb{F}_p)$ of points on the elliptic curve $y^2 = x^3 + ax^2 + 1$, represented using modified projective coordinates [40, Algorithm 7.2.3] over the integers modulo the prime p.

- **Class Group**: The class group $Cl(D)$ of the number field $\mathbb{Q}[\sqrt{D}]$ for negative D, represented using binary quadratic forms as described in Chapter 10. The specific algorithm for composition of forms used [37, Algorithm 5.4.7] is not the fastest, but is reasonably efficient.[1]

The software was developed using the GNU C compiler [100] on a Microsoft Windows operating system. The hardware platform was a personal computer with a 2.5GHz AMD Athlon 64 X2 4800+ processor and 2GB of memory. The AMD chip has two processors, only one of which was used during the tests, except as noted.

The performance results in the following sections are reported in terms of group operation counts, not elapsed times, so the speed of the black boxes is not a factor. It is useful to have a rough idea of the performance of the black boxes, as in most cases the elapsed time can then be estimated from the count of group operations. For most of the tests nearly all group operations are multiplications. The one exception was the large class group computations, where a modified version of the primorial-steps algorithm optimized for fast inverses was used (see Remark 3.1). This increased the group operation counts slightly but reduced the elapsed times, since nearly 1/4 of the group operations were inverses.

Table 11.1 gives performance metrics for each of the black boxes over a range of groups and sizes. For comparison, the hash function used by both the rho and primorial-steps

[1] The NUDPL and NUCOMP algorithms of Shanks and Atkin, also listed in [37], are faster and may be used in future tests (see [25] and [67] for further improvements).

Black Box	Operation	$\|G\| \approx 10^{10}$	$\|G\| \approx 10^{20}$	$\|G\| \approx 10^{30}$	$\|G\| \approx 10^{40}$	$\|G\| \approx 10^{50}$
Cyclic	Multiply	4400	3400	3300	3100	3100
	Invert	9900	9100	9100	9000	9000
Integer	Multiply	3500	2400	2000	1700	1500
	Invert	950	410	270	200	160
Product	Multiply	540	270	180	140	110
	Invert	770	390	270	200	160
2-d Matrix	Multiply	480	480	470	380	330
	Invert	1400	520	470	350	290
Permutation	Multiply	350	180	130	93	80
	Invert	490	250	190	130	110
Elliptic Curve	Multiply	460	240	150	110	93
	Invert	4800	4800	4100	3900	3600
Class Group	Multiply	210	110	70	50	38
	Invert	13000	13000	13000	13000	13000

Table 11.1: Black Box Performance (thousands of group operations per second)

search algorithms can perform about 2,300,000 operations per second on group identifiers of similar size on the same platform.

11.1 Order Computations in Random Groups

The algorithms presented in Chapters 4, 5, and 7 were used to compute the order of a random $\alpha \in G$ using a variety of distributions on G. The computations were performed by the multi-stage sieve algorithm (Algorithm 5.1) using a primorial-steps search (Algorithm 5.2). Both of the fast order algorithms (Algorithms 7.4 and 7.1) are used as subroutines, so effectively all the new order algorithms presented in this thesis were exercised.

In each test, the group G was first selected from a distribution of finite groups, then a random element $\alpha \in G$ was uniformly selected.[2] The multi-stage sieve using a primorial-steps search is a deterministic algorithm, so the statistical data listed in the Table 11.2 only reflect randomness in the inputs. The distributions of groups are listed below.

1. C_p for a random prime $p \in [1, 10^d]$.

2. C_n for a random integer $n \in [1, 10^d]$.

[2] In the case of ideal class groups, the random elements were primeforms (see Chapter 10).

3. $\mathbb{Z}_p^*$ for a random prime $p \in [1, 10^d]$.[3]

4. $\mathbb{Z}_n^*$ for a random integer $n \in [1, 10^d]$.

5. $Cl(D)$, the ideal class group of $\mathbb{Q}[\sqrt{D}]$. A random odd integer $n \in [1, 10^d]$ is chosen and $D = -n$ ($n \equiv 3 \bmod 4$) or $D = -4n$ ($n \equiv 1 \bmod 4$).

6. $Cl(P)$, where a random prime $p \in [1, 10^d]$ is chosen and $P = -p$ ($p \equiv 3 \bmod 4$) or $P = -4p$ ($p \equiv 1 \bmod 4$).

7. $E(\mathbb{F}_p)$, the group of points on the elliptic curve $y^2 = x^3 + x + 1$ over the field $\mathbb{F}_p$, where p is a random prime in $[1, 10^d]$.

8. $GL_2(\mathbb{F}_p)$, the group of non-singular 2×2 matrices over the field $\mathbb{F}_p$, where p is a random prime in $[1, 10^d]$.

In Table 11.2 each row gives statistics for 100 random tests. The first column lists d, indicating that the interval used was $[1, 10^d]$. The remaining columns give a performance metric δ, defined to be the least integer for which

$$T(N) \leq aN^{\delta}$$

is satisfied for the proportion of test cases indicated by the column heading. The value $T(N)$ is the number of group operations used to compute $|\alpha|$, and the constant $a = 4\sqrt{2}$ combines the constant factor $2\sqrt{2}$ for an unbounded primorial search and an assumed constant factor of 2 introduced by the multi-stage sieve. Group operations were counted by the black box to compute $T(N)$, after which δ was computed separately for each test instance via the formula

$$\delta = \frac{\log T(N) - \log a}{\log N}. \tag{11.1}$$

Note that the complexity is expressed in terms of $N = |\alpha|$, not in terms of $|G|$ or the interval over which $|G|$ might range. This is the appropriate complexity metric for a generic order algorithm performing an unbounded search, and accurately reflects Proposition 5.3. The values of δ would have been significantly lower (reflecting better performance) if a weaker metric had been used.

[3] $\mathbb{Z}_p^*$ and C_n both represent random cyclic groups, but the probability distribution is not the same.

The constants $c = 2.5$, $L_1 = 6$, and $B_1 = 2048$ were used in Algorithm 5.1. The parameter b was initially set to 5, then gradually reduced to 3 as the number of stages increased. These choices were not optimized for any particular distribution. The random cyclic groups would have benefited from a smaller value of both b and c, while the random class groups favor a slightly higher value of c (they are relatively more likely to be divisible by small primes than large ones). The initially large choice of b was made to improve the constant factors in smaller test cases and then lowered to reduce the size of the jumps between stages as the bounds get large. The decision was made to adopt a fixed setting for these constants that was retained throughout all tests, including the class group computations, in order to demonstrate the generality of the algorithms. The constants could certainly be optimized to improve performance in a particular application.

In Table 11.2, the ranges of d were chosen to give roughly similar problem sizes for each type of group. Given the volume of tests, the range of d was necessarily restricted. Tests in larger groups indicate that performance tends to improve for larger N, suggesting the impact of some additive constant factors in the data listed in Table 11.2. Slightly smaller problem sizes were used for the prime cyclic groups C_p, where the performance was entirely as expected, with δ just slightly less than $1/2$, representing the worst case scenario.

In nearly every other case, the values of δ occurring in the 10%, 50%, and 90% columns match the values predicted by Proposition 5.3 fairly closely. The semismooth probability function $G(1/u, 2/u)$ would predict δ-values of roughly 0.25, 0.34, and 0.45 respectively for $|\alpha|$ uniformly distributed over some large interval. Although none of the tests actually generated this distribution on $|\alpha|$, the results were comparable or slightly better in most cases.

The δ values for the groups $G = \mathbb{Z}_n^*$ are notably lower. These groups are more acyclic than the other abelian groups listed, meaning that $|\alpha|$ is usually smaller than $|G|$, but the performance metric is in terms of $|\alpha|$ not $|G|$. Effectively, a random cyclic subgroup of $G = \mathbb{Z}_n^*$ is selected when α is chosen, giving rise to a different distribution of cyclic groups. This distribution evidently gives a higher probability to composite orders. A similar phenomenon is apparent to a lesser extent in the matrix groups $GL_2(\mathbb{F}_p)$.

G	d	1%	10%	**50%**	90%	100%
C_p	8	0.44	0.44	**0.47**	0.47	0.48
	9	0.45	0.45	**0.45**	0.48	0.48
	10	0.45	0.46	**0.46**	0.47	0.48
	11	0.45	0.46	**0.47**	0.48	0.48
	12	0.46	0.46	**0.47**	0.48	0.48
C_n	11	0.23	0.28	**0.34**	0.43	0.47
	12	0.23	0.28	**0.34**	0.46	0.48
	13	0.24	0.28	**0.37**	0.45	0.47
	14	0.22	0.27	**0.36**	0.47	0.48
	15	0.21	0.25	**0.34**	0.45	0.49
$\mathbb{Z}_p^*$	11	0.25	0.28	**0.35**	0.44	0.48
	12	0.22	0.27	**0.35**	0.44	0.47
	13	0.22	0.27	**0.33**	0.44	0.48
	14	0.23	0.27	**0.34**	0.44	0.47
	15	0.22	0.25	**0.33**	0.43	0.48
$\mathbb{Z}_n^*$	11	0.23	0.25	**0.29**	0.35	0.43
	12	0.20	0.25	**0.29**	0.35	0.47
	13	0.21	0.24	**0.28**	0.36	0.46
	14	0.20	0.23	**0.27**	0.35	0.45
	15	0.20	0.22	**0.27**	0.37	0.44
$Cl(D)$	22	0.24	0.27	**0.33**	0.41	0.47
	24	0.23	0.27	**0.34**	0.41	0.47
	26	0.19	0.27	**0.35**	0.42	0.46
	28	0.21	0.25	**0.32**	0.43	0.46
	30	0.20	0.26	**0.33**	0.43	0.47
$Cl(P)$	22	0.24	0.28	**0.37**	0.45	0.47
	24	0.21	0.27	**0.36**	0.46	0.48
	26	0.23	0.28	**0.37**	0.46	0.48
	28	0.22	0.27	**0.36**	0.45	0.48
	30	0.22	0.26	**0.33**	0.43	0.48
$E(\mathbb{F}_p)$	11	0.24	0.28	**0.34**	0.43	0.48
	12	0.22	0.27	**0.34**	0.43	0.46
	13	0.20	0.25	**0.35**	0.42	0.47
	14	0.19	0.26	**0.35**	0.42	0.47
	15	0.24	0.26	**0.32**	0.44	0.48
$GL_2(\mathbb{F}_p)$	5	0.21	0.25	**0.34**	0.39	0.43
	6	0.21	0.24	**0.31**	0.40	0.44
	7	0.20	0.23	**0.32**	0.40	0.48
	8	0.18	0.24	**0.30**	0.38	0.45

Table 11.2: Order Computations in Random Groups.

11.2 Comparison to Existing Generic Algorithms

Generic algorithms for computing the structure of an abelian group have been tested against the problem of computing ideal class groups by several practitioners [95, 27, 108, 109]. We consider two particular examples: a baby-steps giant-steps approach due to Buchmann et al. [27], and Teske's algorithm based on the rho method [109]. Their tests were conducted using generic algorithms on the same groups, and they report counts of group operations, making them directly comparable to our results, independent of the technology used. Tables 11.3 and 11.4 list the number of group operations used by three different generic algorithms to compute the structure of ideal class groups for quadratic discriminants of the form $-4(10^n + 1)$ and $-(10^n + 3)$. The entries in columns headed by T represent group operations.

The data for the baby-steps giant-steps algorithm are taken from [27]. The results for the rho method are listed in [109] and represent the average number of iterations over several runs. This excludes group operations used to construct the pseudo-random function; for the larger groups, these should be negligible.

The group operations listed for Algorithm 9.1 are each the median of five runs (the variance between runs was minimal, less than 5%). The confidence parameter c was set to 20 for the smaller groups (less than 1,000,000 elements) and 40 for the rest. In every case the results agreed with those computed by the other two algorithms. The group operation counts for Algorithm 9.1 listed in the first several rows of each table are dominated by the repeated random tests performed to reach the confidence level specified. The other two algorithms were provided with a set of generators, making the comparisons less meaningful. For larger groups this effect is negligible and the confidence parameter can be set arbitrarily high without impacting the results.

The multi-stage sieve (Algorithm 5.1) used a primorial-steps search (Algorithm 5.2) for all order computations involved in computing the group exponent. The peak virtual memory usage did not exceed 30Mb during any of these tests. This contrasts with the standard baby-steps giant-steps algorithm of [27] which was limited by memory constraints.

All three algorithms used the upper bound $h(D) \leq \sqrt{D} \log D$ on the size of the group. For Algorithm 9.1, nearly equivalent results were obtained when using the weaker bound $D^{2/3}$, corresponding to $|G|^{3/2}$.

		Baby/Giant		Rho Method		Algorithm 9.1	
n	$Cl[\sqrt{-(10^n+3)}]$	T	$\frac{T}{\sqrt{\lvert G\rvert}}$	T	$\frac{T}{\sqrt{\lvert G\rvert}}$	T	$\frac{T}{\sqrt{\lvert G\rvert}}$
10	[2,2,48396]	3388	7.7	4988	11.3	1967	4.5
11	[2,2,2,56772]	4891	7.3	7509	11.1	1824	2.7
12	[2,4,117360]	6680	6.9	11137	11.5	2503	2.6
13	[2,2,742228]	12037	7.0	21389	12.4	2747	1.6
14	[2,2,4,1159048]	27615	6.4	42194	9.8	2422	0.56
15	[2,2,2,2,2,4,257448]	57387	10.0	43956	7.7	3956	0.69
16	[2,2,2,2,11809616]	120027	8.7	118944	8.7	3143	0.23
17	[2,2,2,46854696]	134990	7.0	237291	12.2	4793	0.25
18	[2,2,264135076]	233244	7.2	405015	12.5	26665	0.83
19	[2,1649441906]	572162	10.0	800177	13.9	33427	0.58
20	[2,2,2,1856197104]	979126	8.0	1037680	8.5	11127	0.091
22	[2,2,2,2,2,2,678293202]	-	-	1810912	8.7	24569	0.12
22	[2,2,2,19870122100]	-	-	3650074	9.1	13346	0.03
23	[2,2,2,2,23510740696]	-	-	6210273	10.1	141653	0.23
24	[2,4,144373395240]	-	-	12736451	11.9	13528	0.013
25	[2,2,2,2,186902691564]	-	-	19749328	11.4	13536	0.0078
26	[2,4,2062939290744]	-	-	45882067	11.3	707054	0.17
27	[2,2,2,2,2,2,596438010456]	-	-	46568150	7.5	354953	0.057
28	[2,4,4,4987045013072]	-	-	-	-	117628	0.0093
29	[2,2,109151360534920]	-	-	-	-	232533	0.011
30	[2,2,2,2,2,8,4591263001512]	-	-	243207644	7.1	250247	0.0073

Table 11.3: Ideal Class Groups Computed by Generic Algorithms, $D = -4(10^n + 1)$

The elapsed clock times for the algorithms are not directly comparable due to differences in both hardware and software. In [109] Teske reports that computation for the largest discriminant $D = -4(10^{30} + 4)$ took more than two weeks of continuous operation on a Sun SPARCStation4. The computation of the same group structure by Algorithm 9.1 took less than three seconds on a 2.5MHz AMD Athlon processor, representing a speed-up by a factor of well over 100,000. In terms of group operations, the improvement was a factor of about 1,000.

Comparisons for larger groups are limited due to a lack of available results for other algorithms, however, one can readily extrapolate the performance of both the existing methods listed above. They consistently use between $5\sqrt{|G|}$ and $15\sqrt{|G|}$ group operations over a wide range of group sizes. Based on this assumption, the performance difference between existing generic algorithms and the multi-stage sieve becomes more dramatic as the size of the group increases. For the larger discriminants listed in the tables that follow, the implied speed-up is well over 10^9 in many cases.

As we will see in the next section, the median performance of Algorithm 9.1 is substantially better than $|G|^{1/3}$ when computing large ideal class groups, although in the worst case it may be close to $|G|^{1/2}$. For any particular group, the complexity is dictated largely by the multiplicative structure of the exponent of the group, not its size. This accounts for the variability of the performance of Algorithm 9.1 seen in Tables 11.3 and 11.4.

		Baby/Giant		Rho Method		Algorithm 9.1	
n	$Cl[\sqrt{-(10^n+3)}]$	T	$\frac{T}{\sqrt{\lvert G\rvert}}$	T	$\frac{T}{\sqrt{\lvert G\rvert}}$	T	$\frac{T}{\sqrt{\lvert G\rvert}}$
10	[10538]	1038	10.1	1329	12.9	1458	14.2
11	[31057]	2213	12.6	2527	14.3	879	5.0
12	[2.62284]	3223	9.1	4567	12.9	1544	4.4
13	[2,2,124264]	5794	8.2	9611	13.6	1935	2.7
14	[2,2,356368]	9233	7.7	13257	11.1	1804	1.5
15	[3929262]	23564	11.9	27867	14.1	4305	2.2
16	[12284352]	37249	10.6	45624	13.0	2490	0.71
17	[38545929]	67130	10.8	88168	14.2	6109	0.98
18	[102764373]	103039	10.2	137517	13.6	18096	1.8
19	[2,2,2,78425040]	149197	6.0	287486	11.5	5139	0.21
20	[2,721166712]	343423	9.0	522644	13.8	4426	0.17
21	[3510898632]	-	-	826743	13.9	28320	0.48
22	[2,2,2,1159221932]	-	-	1073395	11.1	14418	0.15
23	[2,16817347642]	-	-	2594912	14.1	25254	0.14
24	[2,2,37434472258]	-	-	4355120	11.2	11906	0.031
25	[2,245926103566]	-	-	8562256	12.2	111732	0.16
26	[2,656175474498]	-	-	14301324	12.5	363341	0.32
27	[3881642290710]	-	-	25751685	13.1	93223	0.047
28	[2,2,2,1607591023742]	-	-	41832563	11.7	35479	0.0099
29	[2,17634301773068]	-	-	71532179	12.0	152150	0.026

Table 11.4: Ideal Class Groups Computed by Generic Algorithms, $D = -(10^n + 3)$

11.3 Generic Order Computations in Class Groups

To gain a better understanding of the performance of both the multi-stage sieve and the exponent-based algorithm for abelian group structure, they were applied to the task of computing a large set of ideal class groups. Tables 11.3 and 11.4 were extended, and similar tables were constructed for discriminants of the form $-(2^n - 1)$ and $-4(2^n + 1)$. Building the new tables was a useful exercise, as they present a smoother distribution of problem sizes, and give rise to a wide range of group structures, including a variety of prime, near-prime, and highly composite discriminants.

Recall from Chapter 10 that the rank of the 2-Sylow subgroup of $Cl(D)$ is essentially determined by the factorization of the discriminant D, while the odd part of the class group is conjectured to be a "random" finite abelian group. In particular, a prime discriminant will have $h(D) = |Cl(D)|$ odd, but this does not mean that $h(D)$ will be prime. In fact, this is quite rarely the case.

We consider here the construction of the table for discriminants $D = -(2^n - 1)$ in some detail; the construction of the other tables followed a similar pattern. Appendix A contains tables that show the cost of computing the structure of the class group for values of n between 60 and 160. This covers discriminants ranging from 18 to 48 decimal digits. Class groups were constructed for many larger discriminants (see Appendix B) but this is a convenient range with a complete set of meaningful data points. Table 11.5 contains a short interval from the middle of the range which is an illustrative sample. It happens to include the worst performance for this table, as well as several excellent running times.

Each row starts with the value of n for the discriminant $D = -(2^n - 1)$, and the remaining entries are all logarithms in base 2. The column headed $\lg h(D)$ lists the binary logarithm of the size of the class group, $h(D) = |Cl(D)|$. This is typically slightly less than $n/2$, as predicted by the class number formula. Skipping the column $\lg q_*$ for the moment, the next two columns give the binary logarithms of T_1 and T_2, which represent the number of group operations required to compute $\lambda\big(Cl(D)\big)$ and the structure of $Cl(D)$ respectively. The value T_1 is the cost incurred by Algorithm 8.1, including calls to the multi-stage sieve using a hybrid-search to perform order computations. The value T_2 counts all the group operations used by Algorithm 9.1 after $\lambda\big(Cl(D)\big)$ has been computed, including all discrete logarithms. As in the previous section, the upper bound $h(D) \leq \sqrt{D} \log D$ was used, enabling most

n	$\lg h(D)$	$\lg q_*$	$\lg T_1$	$\lg T_2$	δ_1	δ_2
120	**59.45**	**6.60**	**12.24**	**15.01**	**0.16**	**0.21**
121	59.56	14.81	18.39	13.08	0.27	0.18
122	**59.95**	**24.78**	**27.39**	**15.46**	**0.42**	**0.22**
123	59.90	15.80	19.70	13.05	0.29	0.18
124	61.10	12.23	14.78	13.18	0.20	0.17
125	61.68	16.06	17.37	13.25	0.24	0.17
126	61.92	11.51	13.47	14.16	0.18	0.19
127	62.32	18.78	22.13	12.01	0.31	0.15
128	63.76	16.67	17.74	13.51	0.24	0.17
129	63.91	17.54	20.61	12.61	0.28	0.16

Table 11.5: Sample Construction Costs for $D = -(2^n - 1)$, $n = 120$ to 129

large discrete logarithm operations to be avoided. The last two columns, headed by δ_1 and δ_2, give δ-values analogous to (11.1), that is

$$\delta_i = \frac{\lg T_i - \lg a}{\lg h(D)}, \tag{11.2}$$

where the constant $a = 4\sqrt{2}$ implies $\lg a = 2.5$. The δ-values approximate the exponent in the running time of the algorithm given by

$$T_i(N) \approx aN^{\delta_i}, \tag{11.3}$$

where $N = h(d) = |Cl(D)|$ is the size of the group.

In Table 11.5 the first row, corresponding to $D = -(2^{120} - 1)$, indicates that the size of the class group was approximately $2^{59.45}$, the number of group operations used to compute the exponent of the class group was $T_1 = 2^{12.24} \approx 5,000$, and the number of additional group operations required to compute the structure of the group was $T_2 = 2^{15.01} \approx 33,000$. This was one of the 20 out of 100 cases where T_2 dominated the running time. This typically occurs for the smaller discriminants, and when it does, the running time tends to be very good. In this case the total running time was on the order of $N^{0.21} = N^{\delta_2}$.

The actual elapsed time can be inferred from the number of group operations by noting that roughly 110,000 to 120,000 group operations per second were typically performed during these computations. In this case the actual elapsed time was 0.33 seconds, indicating

that $\approx 115,000$ group operations per second were performed.

Two rows down, the entry for $D = -(2^{122} - 1)$ contains the worst performance over the entire range (asymptotically speaking) with $\delta_1 = 0.42$. In this case $T_1 = 2^{27.39} \approx 176,000,000$ dominated $T_2 = 2^{24.78} \approx 29,000,000$ and the performance was on the order of $N^{0.42}$. The actual elapsed time was 1743 seconds, indicating that $\approx 117,000$ group operations per second were performed during the computation.

The row in between, for a class group of almost identical size, had $\delta_1 = 0.27$ and an elapsed time of 5 seconds. To explain the rather dramatic variance in performance, we now consider the column $\lg q_*$. This column lists the binary logarithm of

$$q_*\big(h(D)\big) = \max\{\sqrt{p}, q\},$$

where p and q are the largest and second largest prime powers dividing $h(D) = |Cl(D)|$. This value corresponds to Definition 5.1 which was given in Chapter 5 when analyzing the complexity of the multi-stage sieve. In most cases, the value of T_1 is a small multiple of q_*, typically $\lg T_1 \approx \lg q_* + 2$. For small values of $\lg q_*$ the difference may be greater; these situations typically occur when q is greater than $\sqrt{p}$ (this happens about 1/3 of the time) and generally mean good performance. The values of $\lg q_*$ corresponding to $q > \sqrt{p}$ are listed in bold in Appendix A.

Given the rather erratic behavior of q_* and the consequent variation in running times, it is more useful to look at typical values of δ_1 and δ_2 over a range of n. As the δ-values represent ratios of logarithms, they are comparable over the entire set of problem sizes. Overall, the value of δ_1 typically dominated and had a median value of 0.265. The behavior of δ_1, while locally erratic, was fairly consistent over intervals of 10 or more. The medians in each subinterval of 10 for both δ_1 and δ_2 are shown in Table 11.6.

The median complexity of T_1 would appear to be around $O\left(N^{0.27}\right)$, which is substantially better than the $O\left(N^{0.34}\right)$ bound of Proposition 5.3. This is somewhat expected for two reasons: (1) as noted in Chapter 10 the order of an ideal class group is more likely to contain small divisors than a random integer, and (2) the computation of δ_1 is relative to the size of the group, not the order of a particular element.[4]

The behavior of δ_2 is quite different. It is much smoother, and clearly declining through-

[4]This is the appropriate metric for an algorithm computing the structure of the entire group.

Range of n	Median δ_1	Median δ_2
60-69	0.285	0.315
70-79	0.265	0.270
80-89	0.250	0.250
90-99	0.255	0.220
100-109	0.240	0.210
110-119	0.285	0.190
120-129	0.270	0.170
130-139	0.265	0.170
140-149	0.300	0.160
150-159	0.290	0.140

Table 11.6: Median δ-Values of Class Group Structure Computations

out the range, suggesting that the true growth rate of T_2 may be subexponential.

Finally, we note the remarkable variance in running times that results from the dependence on q_*. The entry listed for $n = 159$ has $\delta_1 = 0.37$, corresponding to nearly 3×10^9 group operations and an elapsed time of about eight hours, by far the longest of any entry in the table. The entry just three rows above for $n = 156$ has $\delta_1 = \delta_2 = 0.18$, used less than 2×10^5 group operations, and had an elapsed time of just over two seconds. The entry two rows earlier for $n = 154$ was even faster at close to one second and under 100,000 group operations.

The values $n = 159$ and $n = 156$ correspond to 48 and 47 digit discriminants with class groups of size 2.8×10^{24} and 2.1×10^{24} respectively. They would each require over 10^{13} group operations if performed by either of the existing generic algorithms considered in the previous section, assuming a complexity of $\approx 10\sqrt{|G|}$. The implied running time on the hardware platform used in these tests would be over four years for each group.

The two examples above were not the most extreme cases that occurred during the tests. The computation for $D = 2^{246} - 1$, a 74 digit discriminant, used less than 2.7×10^7 group operations and took about seven minutes. The value of $10\sqrt{|G|}$ exceeds 2.4×10^{19}, larger by a factor of nearly 10^{12}. By contrast, the computation performed for $D = -(10^{39} + 3)$, a much smaller class group, used 3.2×10^9 group operations, and had the largest δ-value occurring during the construction of any of the tables, 0.47. The class group in this case is cyclic with order 3 times a large prime, representing nearly the worst case. Even in this

case, the performance was better than existing generic algorithms by a factor of about 20.

Remark 11.1. *While our attention has been focused on the computational complexity of constructing the tables of class groups, we should pause to remark on an interesting result. The Cohen-Lenstra heuristics suggest that approximately 97.75% of the time, the odd part of a random class group should be cyclic. This pattern was observed in most of the tables that were constructed, however the tables for* $D = -(2^n - 1)$ *contain a surprising number of class groups with non-cyclic odd part, close to 15% of the entries. For* $n \leq 170$*, there are 27 such class groups, 21 of which correspond to fundamental discriminants, including the Mersenne primes* $2^{89} - 1$ *and* $2^{107} - 1$*.*

11.4 Comparison to Non-Generic Algorithms

There are a variety of algorithms used to compute ideal class groups. We consider just a few, restricting ourselves to the case of imaginary quadratic number fields.

Shanks' Algorithm

The first practical algorithm for computing large class groups was developed by Shanks using the baby-steps giant-steps method [95]. Assuming the extended Reimann hypothesis (ERH), the Dirichlet class number formula (10.1) may be used to obtain a fairly tight bound on the class number $h(D)$ by evaluating the terms involving primes less than $d^{1/5}$, where $d = |D|$. This reduces the interval that must be searched to find $h(D)$ to a range of size $O\left(d^{2/5} \log d\right)$. The result is an algorithm which uses $O\left(d^{1/5} \log d\right)$ operations in the class group to find $h(D)$. This approach can also be used to compute the structure of the group. The only non-generic aspect of this algorithm is the use of the class number formula to bound the search.

In terms of the size of the class group, $N = |Cl(D)| = h(D) \approx \sqrt{d}$, the running time is $O\left(N^{0.4} \log N\right)$ group operations. Algorithm 9.1 was substantially faster than this during the course of constructing the tables in Appendix B, which involved hundreds of class groups with discriminants ranging from 20 to 70 decimal digits. The observed median δ-value was about 0.27, suggesting a median complexity close to $O\left(N^{.27}\right)$. In terms of d this is better than $O\left(d^{1/7}\right)$. Even for the worst case encountered, $D = -(10^{39} + 3)$ with $\delta = 0.47$, we still find $N^{0.47} < N^{0.4} \log N$, since $N \approx 5 \times 10^{19}$ in this case.

Subexponential Algorithms

The best known algorithms for computing class groups of imaginary quadratic number fields are strictly non-generic; they depend on a specific representation of ideal class groups and perform non-group operations directly on this representation. Appropriately implemented, a subexponential running time can be achieved. The first such algorithm was developed in 1988 by McCurley and Hafner [55] and is described in [37]. Assuming the extended Reimann hypothesis (ERH), the running time of their algorithm, in terms of d, is

$$L(1/2,\ \sqrt{2}) = \exp\left(\left(\ \sqrt{2} + o\,(1)\right)\ \sqrt{\log d \log\log d}\right).$$

Buchmann and Düllmann implemented a parallel version of this algorithm [24] which they used to compute the class group for $D = -4(10^{54} + 1)$ in a total of about 282 hours of computation time on a distributed network of Sun Sparc1 and SparcSLC computers in 1991. This algorithm was later improved by Buchmann et al. and Jacobson [26, 65] by applying techniques borrowed from the multiple-polynomial quadratic sieve method (MPQS) for factoring integers (see [40]). They were then able to compute the class group for the same discriminant in under nine hours on a Sun SparcUltra1 in 1997, and computed class groups for a number of larger discriminants, up to 65 decimal digits. They completed the computation of the required relation matrix for a 70-digit discriminant, but did not determine the class group due to difficulties in computing the Hermite normal form of such a large matrix.

Further refinements of the algorithm along with increased hardware capacity have enabled the computation of class groups with 80-digit discriminants. Jacobson reports computing the class groups for the discriminants $-4(10^{80}+1)$ and $(-10^{80}+3)$ in 5.4 and 10.0 days respectively using a 296 MHz Sun UltraSparc-II processor [66]. These results stand as the largest computations for imaginary quadratic class groups reported in the literature.[5]

Buchmann et al. and Jacobson do not give precise asymptotic estimates for the running time of their algorithm, but in [29] it is conjectured to be $L(1/2,\ \sqrt{9/8}+o\,(1)]$. Cohen suggests that an $L(1/2,\ \sqrt{9/8})$ bound is possible, and that $L(1/2, 1)$ probably represents the limit on what is achievable with this approach [37, p. 259]. While this is an attractive asymptotic

[5] Computations for real quadratic class groups with larger discriminants are reported in [64], but in the real case the class group is generally quite small, often trivial.

result, computing the Hermite normal form of the relation matrix may be a limiting factor for very large discriminants.

One concern with the subexponential algorithms mentioned above is their dependence on the ERH, not just for their running times, but also for their correctness. Booker addresses this issue, giving a modified version of the algorithm which is either fast (assuming the ERH is true), or slow (if not), but always correct [17]. This is done by using the class number formula to estimate $h(D)$ closely enough to be able to ascertain whether the class number output by Buchmann's algorithm could be off by a factor of two. The algorithm always outputs a subgroup of the class group, so if it is incorrect, it must err by at least this amount. The same technique may be applied to verify the output of Algorithm 9.1, which of course does not depend on the ERH, but may simply get unlucky in its selection of random elements.

Comparisons

The computation of the class group for $D = -4(10^{54} + 1)$ computed by Buchmann and Düllmann and later by Buchmann et al. took less than 9 seconds for Algorithm 9.1 running on a 2.5 GHz AMD processor. This hardware is certainly faster than the Sun SparcUltra1 used by Buchmann et al., but not by a factor of 4,000, which is the approximate ratio of the running times. More recently, Jacobson reports computing the class group for $D = -4(10^{55} + 1)$ in 27 minutes on a 296 MHz Sun SparcUltra-II using an improved version of the MPQS algorithm [66]. The improved running time is due in part to faster algorithms for the reduction and composition of quadratic forms [67] which could also be applied to the black box used by Algorithm 9.1.

Without a side-by-side performance test on the same platform it is impossible to make precise comparisons with non-generic algorithms. Extrapolating from the most recent data available [66], it is reasonable to speculate that the median performance of Algorithm 9.1 is at least as good as the best non-generic algorithms for negative discriminants with up to 50 or 60 digits. We should emphasize, however, that the worst case performance of the non-generic algorithms is far superior. The computation for discriminant $-(10^{39} + 3)$ took Algorithm 9.1 over eight hours while the MPQS-based algorithms would almost certainly compute this class group in under a minute.

Conversely, Algorithm 9.1 may occasionally outperform the subexponential algorithms

on larger discriminants. The computations of the class groups for discriminants $-4(10^{68}+1)$, $-4(2^{233}+1)$, $-(2^{236}-1)$, and $-(2^{246}+1)$ each took less than 10 minutes. The available data suggest that the best subexponential algorithms would require an hour or more for discriminants of this size ($\approx$ 70 digits). This advantage can be exploited in a systematic fashion by searching for groups that happen to be easy to compute, yielding a generic algorithm with subexponential complexity, as considered in the next section.

11.5 A Trial of the Generic Subexponential Recipe

Section 5.4 describes an efficient method for solving one of a sequence of random order computations by performing a bounded search on successive problem instances. By choosing the bound $N^{1/u}$ to appropriately balance the probability of success with the cost of an unsuccessful search, a solution with subexponential complexity may be achieved. By Proposition 5.3, the running time $T(N)$ of Algorithm 5.1 on a random input with $|\alpha| = N$ uniformly distributed over some large interval satisfies

$$\Pr\left[T(N) \le cN^{1/u}\right] \le G(1/u, 2/u),$$

where $G(1/u, 2/u)$ is the semismooth probability function defined in Section 6.4.

To validate this approach, a program was implemented to compute the structure of a large ideal class group with discriminant close to a specified value. The program starts at the given value and attempts to compute the structure of the class group for an increasing sequence of discriminants. For each attempt, the program uses a single-stage version of the multi-stage sieve (Algorithm 5.1) to attempt to compute the order of a random group element. There is a single exponentiation phase up to the prime bound L, and then a search phase to the bound $B = cL^2$, with $c \approx 3$ chosen so that the group operations are split evenly between exponentiation and searching. If the program is able to successfully compute the order of a random element, it then goes on to compute the entire class group.

The value of the bound L was determined by the formula $L = N^{1/u}$, where $N = \sqrt{D}$ is a convenient estimate of the size of the class group and u is a chosen parameter.[6]

For the initial trial, the program used discriminants of the form $D = -(10^{80} + 4k + 3)$.

[6] A more accurate estimate would be $N = C\sqrt{D}/\pi$ where $C \approx 0.8815$ (see Chapter 10). This would lower the effective value of u slightly.

The parameter u was set to $6\frac{2}{3}$. Each attempt used about 14 million group operations, taking just over 60 seconds. The program succeeded after 346 attempts using a total of approximately 1.3×10^9 group operations in under 6 hours. The structure of the class group for $D = -(10^{80} + 1387)$ is

$$[2, 2, 2, 3744808318496179963899732666211114420 02].$$

This was an encouraging result, as the median complexity bound $aN^{0.27}$ would imply 4.8×10^{11} group operations and a running time of nearly three months. A running time of 5 to 10 days using the non-generic MPQS algorithm is reported by Jacobson [66] for discriminants of comparable size using a Sun SparcUltra-II.

Once a successful discriminant is known, the class group can be recomputed for at most the cost of a single attempt, or under a minute in this case. This makes it easy to verify the results. Each attempt is completely independent, so the same test could have been completed by 346 computers in one minute. No communication between computers is required, other than assigning a range of discriminants to attempt and collecting successful results.

It is notable that the particular group computed did not have an especially large 2-Sylow subgroup. In fact, the discriminant D evidently only has five factors, as implied by the 2-rank of the class group (one less than the number of factors). The factors are easily extracted from the generators of the 2-Sylow subgroup output by Algorithm 9.1.[7] The factorization of D is

$$2069 \times 8271323 \times 814914901349 \times 1828149326773919 \times$$

$$392222981453242226140731247842458595750044 71.$$

There are many algorithms that can factor D more quickly. The point to be made is that it was not at all evident *a priori* that the class group of D should be particularly easy to compute; it looks like a completely ordinary number. D has almost exactly the $\log\log N$ factors it should have, and the size of its largest prime factor is just what we should expect. One could construct a highly composite D whose class group would necessarily have a

[7] Recall that the representation of an element with order 2 may be used to factor D (see Chapter 10).

large 2-Sylow subgroup, but that was not the case here.

According to Proposition 5.5, the complexity of the procedure described above, assuming a uniform distribution on $N = |\alpha|$, is $L(1/2, \sqrt{2})$. In terms of $|D|$ this is $L(1/2, 1)$. This expression hides the fact that $G(1/u, 2/u)$ may be substantially greater than $\rho(u)$. In particular, when $u = 6.67$, $G(1/u, 2/u)$ is about 100 times larger than $\rho(u)$, roughly 2×10^{-4} versus 2×10^{-6}. Moreover, the distribution of orders in ideal class groups is not uniform, resulting in an observed median complexity of $N^{0.27}$ rather than the predicted $N^{0.34}$.

This perhaps explains why the program was successful after only 346 attempts, rather than the ≈ 5000 attempts implied by $G(1/u, 2/u)$. A second trial found the class group for the 78 digit discriminant $D = -(2^{256} + 735)$ after only 185 attempts with $u = 6.4$, using a similar number of group operations. Additional tests suggest that for random discriminants in the range of 60-90 digits, the probability that $|\alpha|$ is semismooth for random $\alpha \in Cl(D)$ may be more than ten times greater than $G(1/u, 2/u)$.

11.6 A Record Setting Computation

Based on the success of the trial, a larger test was conducted with the aim of computing the class group of an imaginary quadratic number field for a discriminant with more than 100 decimal digits. Eight identical PCs were assembled with the same configuration as used in previous tests, a 2.5GHz AMD Athlon 64 X2 4800+ processor and 2GB of memory. Unlike previous tests, both processors were used, giving a total of 16 parallel threads of computation. The machines worked independently and did not use a network. Each processor was assigned a series of discriminants of the form $4(10^{100} + 2500n + k)$ where $n \in [1, 16]$ identified the processor.

The parameter u was set to 6.9. This choice was based on empirical tests with smaller discriminants which suggested that this value of u should give roughly a 1 in 2000 chance of success in each attempt. Insufficient data was collected to assign a meaningful confidence level to this estimate. It could easily be off by a factor of two, given the observed variance and the limited sample size, and assumes that statistics gathered for smaller discriminants (about 60 digits) can be directly applied to larger ones (over 100 digits).

With this value of u, each attempt used approximately 69 million group operations, taking about 25 minutes elapsed time. The total duration of the test was limited to one

D	$Cl(D)$
$-4(10^{100}+12817)$	[534515673497134927919344774623347870484612874406 60]
$-4(10^{100}+12862)$	[2,2,2,2,375060177747257415231524630621565854960829581526 4]
$-4(10^{100}+17712)$	**[2,2,2,2,3,127945949281378786568395760242745391491470232811 6]**
$-4(10^{100}+20225)$	[2,2,2,2,2,2,4,38363300707956174757207494292618174251703011578 4]

Table 11.7: Imaginary Quadratic Class Groups with $|D| > 10^{100}$

week, implying that each processor would be able to make approximately 400 attempts, 6400 attempts in all. Assuming an independent uniform success probability of 1/2000, this gives a better than 95% chance of successfully computing at least one class group, and we should expect to find three.[8]

At the end of the week, a total of four class groups had been successfully computed. The results are listed in Table 11.7. The first two discriminants are both fundamental ($D/4$ is square-free), and $10^{100} + 12817$ is actually prime. This is somewhat remarkable, as one would expect this to be the most difficult case for the algorithm; it means the 2-Sylow subgroup is as small as possible. It is also worth noting that the odd part of the class group for $D = -4(10^{100} + 17712)$ is non-cyclic.

Based on the published results available, these are believed to be the largest class groups ever computed.

[8] Attempts on consecutive discriminants in an arithmetic sequence are not completely independent, since the 2-Sylow subgroup depends on the factorization of D. We may hope that the odd part of the class group is reasonably independent, based on the Cohen-Lenstra heuristics.

Appendix A

Class Group Table Construction Costs

The tables in this appendix give performance data relating to the construction of the class group tables for discriminants of the form $D = -(2^n - 1)$ which appear in Appendix B.

The value $q_* = q_*\big(h(D)\big)$ is defined by $q_* = \max\{\sqrt{p}, q\}$ where p and q are the largest and second largest prime powers dividing $h(D)$ (see Definition 5.1 and Section 11.3). Values listed in bold indicate that $q > \sqrt{p}$.

The value T_1 is the number of group operations used to compute the exponent of the class group, $\lambda\big(Cl(D)\big)$, and T_2 is the number of additional group operations required to then compute the structure of $Cl(D)$. The values δ_1 and δ_2 are defined by

$$\delta_i = \frac{\lg T_i - 2.5}{\lg h(D)}.$$

See Section 11.3 for a more detailed explanation.

D	$\lg h(D)$	$\lg q_*$	$\lg T_1$	$\lg T_2$	δ_1	δ_2
60	29.19	5.67	10.94	13.20	0.29	0.37
61	29.46	**9.57**	12.88	10.84	0.35	0.28
62	30.32	**10.17**	13.28	12.28	0.36	0.32
63	29.87	**6.07**	11.41	12.27	0.30	0.33
64	30.91	**4.95**	10.67	14.36	0.26	0.38
65	31.78	**5.88**	10.98	12.24	0.27	0.31
66	32.02	**9.00**	11.09	14.36	0.27	0.37
67	32.43	**7.22**	12.48	12.34	0.31	0.30
68	33.41	7.10	10.70	12.76	0.25	0.31
69	33.57	9.63	12.01	11.62	0.28	0.27
70	33.97	8.28	11.50	12.77	0.26	0.30
71	33.95	10.84	12.93	11.97	0.31	0.28
72	35.38	**6.19**	10.63	13.75	0.23	0.32
73	35.98	8.98	11.84	12.16	0.26	0.27
74	36.14	**7.40**	11.16	12.54	0.24	0.28
75	35.85	11.33	13.19	11.60	0.30	0.25
76	37.53	7.32	11.54	12.70	0.24	0.27
77	38.05	14.41	15.53	12.48	0.34	0.26
78	37.93	10.44	12.81	12.75	0.27	0.27
79	38.18	**10.19**	14.29	12.09	0.31	0.25
80	39.59	9.63	11.93	12.91	0.24	0.26
81	39.80	10.83	13.04	12.20	0.26	0.24
82	39.54	**5.93**	11.64	12.56	0.23	0.25
83	40.33	13.95	15.25	12.41	0.32	0.25
84	41.28	6.20	11.45	14.09	0.22	0.28
85	41.55	10.89	13.10	12.33	0.26	0.24
86	42.17	9.12	12.01	13.14	0.23	0.25
87	42.40	**10.17**	14.29	12.42	0.28	0.23
88	43.01	6.76	11.61	13.06	0.21	0.25
89	43.58	14.59	15.68	11.65	0.30	0.21
90	44.00	**8.71**	12.65	13.98	0.23	0.26
91	43.85	9.62	12.94	12.11	0.24	0.22
92	45.56	**8.14**	11.98	13.12	0.21	0.23
93	45.83	12.73	14.73	12.72	0.27	0.22
94	45.84	12.68	14.68	12.71	0.27	0.22
95	46.30	16.59	17.67	12.40	0.33	0.21

Table A.1: Construction Costs for $D = -(2^n - 1)$, $n = 60$ to 95

D	$\lg h(D)$	$\lg q_*$	$\lg T_1$	$\lg T_2$	δ_1	δ_2
96	47.35	**9.30**	14.18	13.49	0.25	0.23
97	48.02	12.77	14.78	13.22	0.26	0.22
98	48.52	12.98	14.95	13.09	0.26	0.22
99	47.78	**10.24**	13.76	12.79	0.24	0.22
100	49.19	**9.68**	14.22	13.49	0.24	0.22
101	49.85	17.75	19.16	13.43	0.33	0.22
102	49.74	9.33	12.92	13.28	0.21	0.22
103	50.35	**15.03**	17.27	12.86	0.29	0.21
104	51.18	12.79	14.76	13.33	0.24	0.21
105	51.74	17.23	18.92	12.95	0.32	0.20
106	51.91	9.91	12.92	13.39	0.20	0.21
107	52.30	11.11	14.44	12.77	0.23	0.20
108	53.56	14.06	15.35	14.11	0.24	0.22
109	53.51	10.91	14.44	12.94	0.22	0.20
110	54.47	**13.39**	16.56	13.50	0.26	0.20
111	53.84	17.55	19.06	12.41	0.31	0.18
112	55.05	16.12	17.39	13.46	0.27	0.20
113	56.32	18.94	20.78	13.13	0.32	0.19
114	55.87	13.77	15.20	13.97	0.23	0.21
115	55.95	**16.11**	20.28	12.51	0.32	0.18
116	57.79	18.64	19.68	13.07	0.30	0.18
117	57.72	13.21	14.92	13.50	0.22	0.19
118	57.89	**18.53**	22.02	13.22	0.34	0.19
119	58.54	15.49	17.07	13.14	0.25	0.18
120	59.45	**6.60**	12.24	15.01	0.16	0.21
121	59.56	**14.81**	18.39	13.08	0.27	0.18
122	59.95	**24.78**	27.39	15.46	0.42	0.22
123	59.90	**15.80**	19.70	13.05	0.29	0.18
124	61.10	12.23	14.78	13.18	0.20	0.17
125	61.68	16.06	17.37	13.25	0.24	0.17
126	61.92	11.51	13.47	14.16	0.18	0.19
127	62.32	18.78	22.13	12.01	0.31	0.15
128	63.76	16.67	17.74	13.51	0.24	0.17
129	63.91	17.54	20.61	12.61	0.28	0.16

Table A.2: Construction Costs for $D = -(2^n - 1)$, $n = 96$ to 129

D	$\lg h(D)$	$\lg q_*$	$\lg T_1$	$\lg T_2$	δ_1	δ_2
130	63.84	13.25	15.00	15.40	0.20	0.20
131	63.98	**11.80**	15.76	13.65	0.21	0.17
132	65.36	**12.91**	16.45	14.75	0.21	0.19
133	65.68	**20.81**	24.08	13.23	0.33	0.16
134	66.17	23.88	25.64	13.24	0.35	0.16
135	65.93	**12.59**	16.41	13.92	0.21	0.17
136	67.16	18.99	20.81	14.15	0.27	0.17
137	68.14	**16.12**	20.31	13.87	0.26	0.17
138	68.00	21.34	22.75	13.43	0.30	0.16
139	67.96	**18.17**	20.77	13.56	0.27	0.16
140	69.33	**18.00**	20.85	15.63	0.26	0.19
141	69.44	**20.08**	22.83	13.81	0.29	0.16
142	69.97	**15.24**	18.51	13.75	0.23	0.16
143	70.53	12.03	16.39	13.61	0.20	0.16
144	71.08	23.17	24.46	15.46	0.31	0.18
145	71.70	26.01	27.20	13.26	0.34	0.15
146	72.79	**19.25**	22.42	13.95	0.27	0.16
147	72.13	27.28	29.19	13.65	0.37	0.15
148	73.26	26.37	28.11	13.68	0.35	0.15
149	73.80	26.17	27.53	13.47	0.34	0.15
150	73.59	14.48	16.92	15.87	0.20	0.18
151	74.08	24.36	25.84	12.66	0.32	0.14
152	75.58	14.40	18.44	14.38	0.21	0.16
153	76.24	20.78	22.51	13.43	0.26	0.14
154	75.66	**13.19**	16.51	14.39	0.19	0.16
155	76.09	25.81	27.27	13.49	0.33	0.14
156	77.51	12.89	16.58	16.11	0.18	0.18
157	77.79	31.97	30.12	13.55	0.36	0.14
158	78.29	28.49	29.54	13.59	0.35	0.14
159	77.88	31.00	31.51	13.24	0.37	0.14
160	79.13	**24.95**	27.28	15.83	0.31	0.17

Table A.3: Construction Costs for $D = -(2^n - 1)$, $n = 130$ to 160

Appendix B

Imaginary Quadratic Class Groups

The tables that follow give the group structure for the ideal class group $Cl(D)$ of the number field $\mathbb{Q}[\sqrt{D}]$. The tables are organized according to the form of D:

1. $D = -(2^n - 1)$
2. $D = -4(2^n + 1)$
3. $D = -(10^n + 3)$
4. $D = -4(10^n + 1)$

A typical group structure listing is

$$[2, 2, 2, 4, 312],$$

for $D = -(2^n - 28)$, indicating that the group is isomorphic to

$$C_2 \otimes C_2 \otimes C_2 \otimes C_4 \otimes C_{312}.$$

In this example, as in most cases, the odd part of the class group is cyclic and the group structure appears in standard divisor form. In order to highlight groups with non-cyclic odd part, all but the largest cyclic subgroup of order p^h is listed separately for each p, as in the entry for $D = -(2^n - 29)$, which is written as **[4,3,1080]**, rather than the equivalent divisor form [12,1080]. Such cases are highlighted in bold.

Where space permits, the class number $h(D) = |Cl(D)|$ is listed along with the value of $L(1, \chi_D)$. When absent, $h(D)$ can be computed as the product of the orders of the cyclic

factors of $Cl(D)$, and $L(1, \chi_D)$ can be derived from the class number formula

$$h(D) = \frac{1}{\pi} L(1, \chi_D) \sqrt{|D|},$$

valid for $D < -4$.

The groups were computed using Algorithm 9.1 in its probabilistic form. This algorithm always outputs a subgroup of the desired group. Assuming it is provided with uniformly random elements by the black box implementing the group, it will succeed in finding the entire group with probability greater than $1 - 2^{-c}$. The parameter c was set to 40 in these computations, and the black box used random primeforms (see Chapter 10).

The output was then independently checked by computing $L(1, \chi_D)$ using the class number formula for all primes up to 10^8, sufficient in each case to give nearly 4 digits of agreement with the value of $L(1, \chi_D)$ implied by the output by Algorithm 9.1. Note that since the algorithm always outputs a subgroup, any error would cause the implied value of $L(1, \chi_D)$ to be off by at least a factor of two.

The procedure described above is not sufficient to unconditionally guarantee the results. The algorithm itself did not use the class number formula, however, and it is unlikely that an error in the algorithm would happen to coincide with a case of remarkably slow convergence. Those requiring an unconditional result may see [17] for a more efficient method of verification.

A more likely source of error in the numerous tables that follow is an inadvertent transcription error during the process of preparing this manuscript for publication. The author would be grateful for any corrections.[1]

[1]He may be contacted via the perpetual e-mail address: AndrewVSutherland@alum.mit.edu.

n	$Cl(D)$	$h(D)$	$L(1, \chi_D)$
2	[1]	1	0.6046
3	[1]	1	1.1874
4	[2]	2	1.6223
5	[3]	3	1.6927
6	[4]	4	1.5832
7	[5]	5	1.3939
8	[2,6]	12	2.3608
9	[14]	14	1.9457
10	[2,8]	16	1.5716
11	[18]	18	1.2499
12	[2,2,10]	40	1.9637
13	[55]	55	1.9092
14	[2,36]	72	1.7672
15	[2,26]	52	0.9025
16	[2,2,28]	112	1.3745
17	[285]	285	2.4731
18	[2,2,48]	192	1.1781
19	[255]	255	1.1064
20	**[2,2,2,3,36]**	864	2.6507
21	[2,456]	912	1.9785
22	[2,2,220]	880	1.3499
23	[1554]	1554	1.6856
24	[2,2,2,2,132]	2112	1.6199
25	[2,1886]	3772	2.0457
26	[2,2424]	4848	1.8592
27	[2,1950]	3900	1.0576
28	[2,2,2,4,312]	9984	1.9144
29	**[4,3,1080]**	12960	1.7572
30	[2,2,2,2,980]	15680	1.5033

Table B.1: Class Groups for $D = -(2^n - 1)$, $n = 2$ to 30

n	$Cl(D)$	$h(D)$	$L(1,\chi_D)$
31	[19865]	19865	1.3467
32	[2,2,8,1320]	42240	2.0249
33	[2,2,13764]	55056	1.8662
34	[2,31712]	63424	1.5202
35	**[2,2,3,4752]**	57024	0.9665
36	[2,2,2,2,2,4,1428]	182784	2.1905
37	[213780]	213780	1.8116
38	**[2,3,59148]**	354888	2.1265
39	[2,2,69708]	278832	1.1814
40	[2,2,2,2,2,17176]	549632	1.6467
41	**[5,190320]**	951600	2.0160
42	[2,2,4,8,6680]	855040	1.2809
43	[2,589908]	1179816	1.2497
44	[2,2,2,2,2,105756]	3384192	2.5348
45	[2,2,2,2,222396]	3558336	1.8846
46	[2,2,883124]	3532496	1.3229
47	**[2,3,1125270]**	6751620	1.7879
48	[2,2,2,2,2,2,2,103868]	13295104	2.4896
49	[12773754]	12773754	1.6914
50	[2,2,2,2,4,299760]	19184640	1.7962
51	[2,2,2,1526448]	12211584	0.8085
52	[2,2,2,2,2,955200]	30566400	1.4309
53	[4,15994416]	63977664	2.1178
54	[2,2,2,4,1816620]	58131840	1.3607
55	[2,2,2,4,2189536]	70065152	1.1597
56	[2,2,2,2,2,2,3059532]	195810048	2.2916
57	[2,2,62777550]	251110200	2.0781
58	[2,2,2,2,17140032]	274240512	1.6048
59	**[3,110142126]**	330426378	1.3672
60	[2,2,2,2,2,2,2,2,4,598328]	612687872	1.7926

Table B.2: Class Groups for $D = -(2^n - 1)$, $n = 31$ to 60

n	$Cl(D)$	$h(D)$	$L(1,\chi_D)$
61	[740609005]	740609005	1.5322
62	[2,668431740]	1336863480	1.9557
63	[2,2,2,2,61476654]	983626464	1.0175
64	[2,2,2,2,4,31569408]	2020442112	1.4779
65	**[2,3,7,87750936]**	3685539312	1.9062
66	[2,2,2,2,2,2,68055552]	4355555328	1.5930
67	[5788240250]	5788240250	1.4969
68	**[2,2,2,2,8,3,29688912]**	11400542208	2.0848
69	[2,2,3184521524]	12738086096	1.6471
70	[2,2,2,2,2,2,4,65509160]	16770344960	1.5334
71	**[2,3,2767877316]**	16607263896	1.0737
72	[2,2,2,2,2,2,2,2,4,21799260]	44644884480	2.0410
73	**[2,3,11323615056]**	67941690336	2.1963
74	[2,2,2,9493972488]	75951779904	1.7361
75	[2,2,2,2,2,1934973916]	61919165312	1.0008
76	[2,2,2,2,2,6180379360]	197772139520	2.2603
77	[2,2,71365677450]	285462709800	2.3070
78	[2,2,2,2,4,4088034832]	261634229248	1.4951
79	[2,156144357908]	312288715816	1.2619
80	**[2,2,2,2,2,2,2,3,2161806504]**	830133697536	2.3719
81	[2,2,2,2,59992961030]	959887376480	1.9393
82	**[2,2,2,5,19983331600]**	799333264000	1.1420
83	[1377626540508]	1377626540508	1.3917
84	[2,2,2,2,2,2,2,2,2,4,1299302792]	2660972118016	1.9008
85	[2,1604445038960]	3208890077920	1.6208
86	[2,2,2,619494115968]	4955952927744	1.7701
87	[2,2,2,4,181506436060]	5808205953920	1.4669
88	**[2,2,2,2,2,2,2,2,3,11523447450]**	8850007641600	1.5804
89	**[3,4391379741531]**	13174139224593	1.6636
90	[2,2,2,2,2,2,2,2,2,34327463280]	17575661199360	1.5693

Table B.3: Class Groups for $D = -(2^n - 1)$, $n = 61$ to 90

n	$Cl(D)$	$h(D)$	$L(1, \chi_D)$
91	[2,2,2,1976804272796]	15814434182368	0.9985
92	[2,2,2,2,2,2,2,406370538900]	52015428979200	2.3222
93	[2,31278307292050]	62556614584100	1.9748
94	[2,2,2,4,1969979302000]	63039337664000	1.4072
95	**[2,2,2,3,3606156640236]**	86547759365664	1.3661
96	[2,2,2,2,2,2,2,2,2,2,174702625204]	178895488208896	1.9967
97	[284843279318520]	284843279318520	2.2480
98	[2,2,2,50283229132320]	402265833058560	2.2449
99	[2,2,2,2,2,2,3779471343700]	241886165996800	0.9545
100	[2,2,2,2,2,2,2,2,2,2,626248865900]	641278838681600	1.7894
101	**[3,337860394107390]**	1013581182322170	1.9998
102	[2,2,2,2,2,2,2,2,3680639512600]	942243715225600	1.3146
103	[1437340144676956]	1437340144676956	1.4180
104	[2,2,2,2,2,2,2,2,9997156170204]	2559271979572224	1.7853
105	[2,2,2,2,2,2,2,2,14679888346902]	3758051416806912	1.8537
106	[2,2,4,4,65898353560000]	4217494627840000	1.4710
107	**[3,1850273912356905]**	5550821737070715	1.3690
108	[2,2,2,2,2,2,2,2,2,4,6464844094380]	13240000705290240	2.3090
109	[12856914971890294]	12856914971890294	1.5854
110	**[2,2,2,2,2,2,2,2,4,3,8132904454932]**	24984282485551104	2.1785
111	[2,2,2,2,1008917675494250]	16142682807908000	0.9953
112	[2,2,2,2,2,2,2,2,2,73028921293900]	37390807702476800	1.6302
113	[2,2,2,11230609518274800]	89844876146198400	2.7698
114	[2,2,2,2,2,4,514023516508992]	65795010113150976	1.4343
115	[2,2,2,2,4353489739454866]	69655835831277856	1.0737
116	**[2,2,2,2,2,2,2,3,647708620088748]**	248720110114079232	2.7109
117	[2,2,2,2,2,2,2,1859403877070220]	238003696264988160	1.8343
118	[2,2,2,2,16689133069770224]	267026129116323584	1.4552
119	[2,2,2,2,26259617278827738]	420153876461243808	1.6191
120	**[2,2,2,2,2,2,2,2,2,2,2,2,2,3,31997504201592]**	786370663258324992	2.1428

Table B.4: Class Groups for $D = -(2^n - 1)$, $n = 91$ to 120

n	$Cl(D)$	$h(D)$
121	[2,2,212084789514945746]	848339158059782984
122	[2,557695829452263720]	1115391658904527440
123	[2,2,2,134836018044688070]	1078688144357504560
124	[2,2,2,2,2,2,38733480508822160]	2478942752564618240
125	**[2,2,2,3,1536212020897742 70]**	3686908850154582480
126	**[2,2,2,2,2,2,2,4,4,3,712133545944084]**	4375348506280452096
127	[5737275303524805875]	5737275303524805875
128	**[2,2,2,2,2,2,4,3,20347522750969596]**	15626897472744649728
129	[2,2,2,2160940573696772708]	17287524589574181664
130	[2,2,2,2,2,2,8,32205352982135040]	16489140726853140480
131	[18228056819821440792]	18228056819821440792
132	[2,2,2,2,2,2,2,2,2,2,4,5761802547957820]	47200686472870461440
133	[2,29461361815996545308]	58922723631993090616
134	[2,2,4,5192833198893763632]	83085331182300218112
135	[2,2,2,2,2,2,2,2,274403095499287608]	70247192447817627648
136	[2,2,2,2,2,2,2,8,160854231236508944]	164714732786185158656
137	[324341937184959921330]	324341937184959921330
138	[2,2,2,2,2,8,1153110914626226648]	295196394144314021888
139	[287087800395852071302]	287087800395852071302
140	[2,2,2,2,2,2,2,2,2,2,2,4,45208918446430584]	740702919826318688256
141	[2,2,2,2,50033479397233383264]	800535670355734132224
142	[2,2,2,2,72142038277753795260]	1154272612444060724160
143	[2,2,2,4,53321674221753474576]	1706293575096111186432
144	[2,2,2,2,2,2,2,2,2,2,2,2,2,2,76001362727247948]	2490412653846460760064
145	[2,2,2,479716525401017260070]	3837732203208138080560
146	[2,2,2,8,127997201290334098608]	8191820882581382310912
147	[2,2,2,643885875052825773480]	5151087000422606187840
148	[2,2,2,2,2,2,2,4,22047386783619762680]	11288262033213318492160
149	[16477686585042268595016]	164776865850422685950 16
150	[2,2,2,2,2,2,2,2,2,4,4,1735682730115349792]	14218712925104945496064

Table B.5: Class Groups for $D = -(2^n - 1)$, n = 121 to 150

n	$Cl(D)$
151	[2,2,2,2501021688083157230456]
152	[2,2,2,2,2,2,2,2,2,1104785057239126932 00]
153	[2,2,2,2,2,2,1391196855921714605480]
154	[2,2,2,2,2,2,2,2,2338907103700721104 32]
155	[2,2,2,2,2,25170713295360513180 48]
156	[2,2,2,2,2,2,2,2,2,2,2,2,2,2,131033490557878893 60]
157	[2,2,6546347420555693361372 0]
158	[2,2,2,4618053278949211447788 0]
159	[2,2,2,2,2,2,4335847341451318238982]
160	[2,2,2,2,2,2,2,2,4,8,80894875660895214584]
161	**[2,2,2,2,2,3,9402268582726043291484]**
162	**[2,2,2,2,2,2,2,2,2,4,3,1480751650895140358 40]**
163	[2,2,4,825755260405618671490 88]
164	[2,2,2,2,2,2,2,4,72304207188926530168 44]
165	[2,2,2,2,2,2,2,2,15609181023830804209066]
166	[2,2,2,2,2,4,32453238609785631144276]
167	[62421647258732351346002 40]
168	[2,2,2,2,2,2,2,2,2,2,2,2,2,2,4,204958770574343826504]
169	[4,4,10664603335085876568771 36]
170	**[2,2,2,2,2,3,2938643081943626104419 12]**
171	[2,2,2,2,2,5313080622348419877652 80]
172	[2,2,2,2,2,2,2,2,4,71048120969337346667300]
174	[2,2,2,2,2,2,2,2,4,14946812457112981224660 8]
175	[2,2,2,2,2,2,2,2,74601756936813572163908 8]
176	**[2,2,2,2,2,2,2,2,2,2,2,2,2,3,1545783428891237651149 2]**
177	[2,2,2,2,17407419801308548290217050]
178	[2,2,16,4970946258775991749125024]
180	**[2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,3,4958316423551427177 60]**

Table B.6: Class Groups for $D = -(2^n - 1)$, $n = 151$ to 180

n	$Cl(D)$
182	[2,2,2,2,2,2,2,2,2,331853001743354971776588O]
186	[2,2,2,2,4,738780319784144643263224 40]
188	[2,2,2,2,2,2,2,4,32345533066029597397234176]
189	[2,2,2,2,2,2,4,55122946112866947997364232]
192	[2,2,2,2,2,2,2,2,2,2,2,2,570948084230419597420174 0]
194	[2,2,2,2,2,2333988113592806608377992448]
204	[2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,259492976671514051349535 92]
206	**[2,2,4,3,13421324202077670348801667046 4]**
207	[2,2,2,2,2,2,2,475140436166339561311542071 20]
208	[2,2,2,2,2,2,2,2,4,4,284212417598784960184167701 2]
211	[2,778749525306692221616157312291 0]
216	[2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,4,202055965651810724300315976 0]
222	[2,2,2,2,2,2,2,2,2,4,455299124081182319102752740900]
236	[2,2,2,2,2,2,2,2,2,2,2,12181774191065689917248524740279 6]
246	[2,2,2,2,2,2,2,2,2,11578142709273899955204755179174400]

Table B.7: Class Groups for $D = -(2^n - 1)$, $n > 180$

n	$Cl(D)$	$h(D)$	$L(1, \chi_D)$
2	[2]	2	1.4050
3	[2]	2	1.0472
4	[4]	4	1.5239
5	[2,2]	4	1.0938
6	[2,4]	8	1.5587
7	[2,6]	12	1.6596
8	[16]	16	1.5677
9	[2,6]	12	0.8322
10	[2,16]	32	1.5700
11	[2,18]	36	1.2493
12	[2,16]	32	0.7853
13	[2,26]	52	0.9024
14	[2,2,32]	128	1.5707
15	[2,2,38]	152	1.3190
16	[320]	320	1.9635
17	[2,106]	212	0.9198
18	[2,2,4,24]	384	1.1781
19	[2,406]	812	1.7615
20	[2,624]	1248	1.9144
21	[2,2,148]	592	0.6421
22	[2,2,448]	1792	1.3744
23	[2,902]	1804	0.9784
24	[2,4,424]	3392	1.3008
25	[2,2,2,480]	3840	1.0413
26	[2,4,4,380]	12160	2.3317
27	[2,2,2868]	11472	1.5554
28	[2,7600]	15200	1.4573
29	[2,2,4238]	16952	1.1492
30	[2,2,2,2,2016]	32256	1.5463

Table B.8: Class Groups for $D = -4(2^n + 1)$, $n = 2$ to 30

n	$Cl(D)$	$h(D)$	$L(1, \chi_D)$
31	[2,21450]	42900	1.4542
32	[4,11784]	47136	1.1298
33	[2,2,2,4366]	34928	0.5920
34	[2,2,4,8256]	132096	1.5831
35	[2,2,2,2,10290]	164640	1.3952
36	[2,2,4,12744]	203904	1.2218
37	[2,2,89360]	357440	1.5145
38	[2,2,2,52800]	422400	1.2655
39	[2,2,156556]	626224	1.3267
40	[2,638016]	1276032	1.9115
41	[2,2,196508]	786032	0.8326
42	[2,2,2,2,4,25968]	1661952	1.2448
43	[2,1539306]	3078612	1.6305
44	[2,4,585712]	4685696	1.7548
45	[2,2,2,2,151644]	2426304	0.6425
46	[2,2,4,4,139624]	8935936	1.6733
47	[2,2,2198988]	8795952	1.1647
48	[2,2,3259024]	13036096	1.2205
49	[2,2,4131898]	16527592	1.0942
50	[2,2,2,4,1101520]	35248640	1.6501
51	[2,2,2,2,2635110]	42161760	1.3956
52	[2,2,12483136]	49932544	1.1688
53	[2,2,11814224]	47256896	0.7822
54	[2,2,2,2,4,1827920]	116986880	1.3691
55	[2,2,2,2,12984624]	207753984	1.7193
56	[2,2,75882256]	303529024	1.7762
57	[2,2,2,20421860]	163374880	0.6760
58	[2,4,51089472]	408715776	1.1958
59	[2,2,2,102674024]	821392192	1.6994
60	[2,2,4,46534344]	744549504	1.0892

Table B.9: Class Groups for $D = -4(2^n + 1)$, $n = 31$ to 60

n	$Cl(D)$	$h(D)$	$L(1, \chi_D)$
61	[2,633601130]	1267202260	1.3108
62	[2,2,2,2,140455792]	2247292672	1.6438
63	[2,2,2,2,127245180]	2035922880	1.0530
64	[2,3428667792]	6857335584	2.5079
65	[2,2,2,2,2,95587504]	3058800128	0.7910
66	[2,2,2,2,8,60224152]	7708691456	1.4096
67	[2,2,2979427330]	11917709320	1.5410
68	[4,4,923702784]	14779244544	1.3513
69	[2,2,2,1518226350]	12145810800	0.7853
70	[2,2,2,2,2,1045055616]	33441779712	1.5288
71	[2,2,10010371370]	40041485480	1.2944
72	[2,2,4,4,674809000]	43187776000	0.9872
73	[2,2,15960420426]	63841681704	1.0319
74	[2,2,2,32,533687296]	136623947776	1.5615
75	[2,2,2,2,4,2578785180]	165042251520	1.3338
76	[2,2,8,9338631600]	298836211200	1.7077
77	[2,2,2,2,2,8323363576]	266347634432	1.0763
78	[2,2,2,2,2,2,2,4,7,106306872]	381003829248	1.0886
79	[2,410126301550]	820252603100	1.6572
80	**[2,2,3,99200142912]**	1190401714944	1.7006
81	**[2,2,2,2,2,3,6275946084]**	602490824064	0.6086
82	[2,2,4,4,38068113240]	2436359247360	1.7403
83	[2,2,2,2,4,36792104616]	2354694695424	1.1893
84	[2,2,2,2,240537671328]	3848602741248	1.3746
85	[2,2,2,484236776908]	3873894215264	0.9783
86	[2,2,2,4,285962173240]	9150789543680	1.6341
87	[2,2,2,1168389134586]	9347113076688	1.1803
88	[2,2,2,1862490766144]	14899926129152	1.3304
89	[2,2,2,1697720849616]	13581766796928	0.8575
90	[2,2,2,2,2,2,2,4,4,18785810760]	38473340436480	1.7176

Table B.10: Class Groups for $D = -4(2^n + 1)$, $n = 61$ to 90

n	$Cl(D)$	$h(D)$	$L(1, \chi_D)$
91	[2,2,2,2,2,2025525498564]	64816815954048	2.0462
92	[2,31601684017200]	63203368034400	1.4108
93	[2,2,4,2770647758128]	44330364130048	0.6997
94	[2,2,4,8407351827936]	134517629246976	1.5014
95	[2,2,2,2,10447724225336]	167163587605376	1.3193
96	[2,4,27674723147544]	221397785180352	1.2355
97	[2,2,2,4,9689701386536]	310070444369152	1.2236
98	[2,2,2,4,8,1980140340256]	506915927105536	1.4144
99	[2,2,2,2,2,2,12561002823684]	803904180715776	1.5861
100	[2,2,2,2,8,10211112610912]	1307022414196736	1.8235
101	[2,419090842148170]	838181684296340	0.8269
102	[2,2,2,2,2,2,2,4,3103559367648]	1589022396235776	1.1085
103	[2,2,700107982242348]	2800431928969392	1.3813
104	[4,1341443132751104]	5365772531004416	1.8715
105	[2,2,2,2,2,2,2,2,2,4704134978016]	2408517108744192	0.5940
106	[2,4,8,152591277074664]	9765841732778496	1.7031
107	[2,2,2609568510549062]	10438274042196248	1.2872
108	[2,2,2,2,4,163268175265848]	10449163217014272	0.9111
109	[2,2,5181374289180720]	20725497156722880	1.2779
110	[2,2,2,2,16,148929089444608]	38125846897819648	1.6622
111	[2,2,2,2,2,948919973014500]	30365439136464000	0.9361
112	[2,2,2,2,4068370851009552]	65093933616152832	1.4190
113	[2,2,2,2,3340473133525084]	53447570136401344	0.8239
114	[2,2,2,2,2,2,4,594947682110888]	152306606620387328	1.6601
115	[2,2,2,2,2,7238180096185432]	231621763077933824	1.7851
116	[2,4,39611096763862792]	316888774110902336	1.7270
117	[2,2,2,2,12081594462913008]	193305511406608128	0.7449
118	[2,2,2,2,2,2,8761812716136304]	560756013832723456	1.5280
119	[2,2,2,2,48231335426727154]	771701366827634464	1.4869
120	[2,2,2,2,8,7409202264602512]	948377889869121536	1.2921

Table B.11: Class Groups for $D = -4(2^n + 1)$, $n = 91$ to 120

n	$Cl(D)$	$h(D)$	$L(1, \chi_D)$
121	[2,2,2,106986725624391840]	855893804995134720	0.8246
122	[2,2,2,4,4,15429095956233960]	1974924282397946880	1.3454
123	[2,2,2,2,2,81686971271376968]	2613983080684062976	1.2592
124	[2,2,4,301131950113345208]	4818111201813523328	1.6411
125	[2,2,2,2,4,64360238847005884]	4119055286208376576	0.9921
126	[2,2,2,2,2,2,2,2,8,4521727384331808]	9260497683111542784	1.5771
127	[2,7776985522239155510]	15553971044478311020	1.8731
128	[2,8893572465111730704]	17787144930223461408	1.5146
129	[2,2,2,4,403528264911335220]	12912904477162727040	0.7775
130	[2,2,2,2,2,2,2,8,37505634724057368]	38405769957434744832	1.6352
131	[2,2,2,5356650012168439898]	42853200097347519184	1.2901
132	[2,2,2,2,2,2,690317396902750896]	44180313401776057344	0.9405
133	[2,2,2,2,4271991595853353274]	68351865533653652384	1.0289
134	[2,2,2,2,2,6471891555875423904]	207100529788013564928	2.2044
135	[2,2,2,2,2,2,2,2,487733827436403996]	124859859823719422976	0.9398
136	[2,2,4,21509942342495407896]	344159077479926526336	1.8316
137	[2,2,2,2,13443277789380251386]	215092444630084022176	0.8095
138	[2,2,2,2,2,2,8,955905797806401016]	489423768476877320192	1.3024
139	[2,2,237935127228957421290]	951740508915829685160	1.7908
140	[2,2,4,67434855746105073768]	1078957691937681180288	1.4356
141	[2,2,2,2,2,23976374495548803838]	767243983857561722816	0.7218
142	[2,2,2,2,2,56296488161850062976]	1801487621179202015232	1.1985
143	[2,2,2,2,2,2,34027056337138672248]	2177731605576875023872	1.0244
144	[2,2,2,2,4,8,6629442817932122176]	3394274722781246554112	1.1290
145	[2,2,2,2,2,181896074944420809834]	5820674398221465914688	1.3690
146	**[2,2,4,16,3,118802470417283788 80]**	9124029728047394979840	1.5175
147	[2,2,2,2,2,409390362389228326180]	13100491596455306437760	1.5406
148	[2,8474887168533260362128]	16949774337066520724256	1.4095
149	[2,2,2,4,648390089042262571112]	20748482849352402275584	1.2200
150	[2,2,2,2,2,2,2,2,2,2,2,167858927842151483 20]	34377508422072623759360	1.4294

Table B.12: Class Groups for $D = -4(2^n + 1)$, $n = 121$ to 150

n	$Cl(D)$
151	[2,2,13565987331423544350222]
152	[2,4,4,2486625235897103930008]
153	**[2,2,2,2,2,2,2,2,3,44489929282897921062]**
154	[2,2,2,2,2,4,4,4,87335984474738098188]
155	[2,2,2,2,2,585174515552655400 4008]
157	[2,2,2,2,1829066685291916330 3526]
158	[2,2,2,5964080910360672512 7424]
160	[2,4,8,24791704971910586357136]
161	[2,2,2,1079971927231140324239 90]
162	[2,2,2,2,2,2,2,4,8,440924224969465891704]
163	[2,2,2,4431718539125026793771 08]
164	[2,2,2,5790685967227833679857 28]
166	[2,2,2,2,16,37522567121602510218672]
167	[2,5756971480532199656383074]
168	[2,2,2,2,2,2,4,47134941245247353027584]
169	[2,2,2,2805948712285703921662296]
172	[2,2,4,41508976796300902269987 12]
173	[2,2,2,2,2,19658940638422164370 52542]
174	[2,2,2,2,2,2,4,64642622119225593646 4352]
176	[2,2,8705726626646885643767 1696]
177	[2,2,2,2,2,2,27864619584216895287 91692]
178	[2,2,4,33122413767213912941942520]
179	[2,2,21942164298800883615067 4934]
180	[2,2,2,2,2,2,2,74054076021930233882 31504]

Table B.13: Class Groups for $D = -4(2^n + 1)$, $n = 151$ to 180

n	$Cl(D)$
182	[2,2,2,2,2,2,2,2,2,406133399563253688408 2800]
183	[2,2,635454703803486297188966172]
184	[2,2,8,132134814442840130455148416]
186	**[2,2,2,2,2,4,4,8,3,7936595145917672035935 84]**
189	**[2,2,2,2,2,2,2,2,4,3,526414607889671287481 0100]**
190	[2,2,2,2,2,2,2,4,871665272330626520592423 68]
191	[2,212446771655964150938733795 30]
192	[2,2,4,344118658888008989186342 5240]
194	[2,2,2,2,2,2,4,4,170134824093753153808053424]
216	[2,2,2,2,2,2,2,223110273011353883077281507 4976]
218	[2,2,2,2,8,429948782403269688460522192 2336]
224	[2,2,2,919247478909699589095795656584640]
233	[2,2,2,708262015421968724331748587873 8602]

Table B.14: Class Groups for $D = -4(2^n + 1)$, $n > 180$

n	$Cl(D)$	$h(D)$	$L(1, \chi_D)$
2	[5]	5	1.5478
3	[4]	4	0.3968
4	[12]	12	0.3769
5	[39]	39	0.3874
6	[105]	105	0.3299
7	[706]	706	0.7014
8	[1702]	1702	0.5347
9	[2,1840]	3680	0.3656
10	[10538]	10538	0.3311
11	[31057]	31057	0.3085
12	[2,62284]	124568	0.3913
13	[2,2,124264]	497056	0.4938
14	[2,2,356368]	1425472	0.4478
15	[3929262]	3929262	0.3904
16	[12284352]	12284352	0.3859
17	[38545929]	38545929	0.3829
18	[102764373]	102764373	0.3228
19	[2,2,2,78425040]	627400320	0.6233
20	[2,721166712]	1442333424	0.4531
21	[3510898632]	3510898632	0.3488
22	[2,2,2,1159221932]	9273775456	0.2913
23	[2,16817347642]	33634695284	0.3341
24	[2,2,37434472258]	149737889032	0.4704
25	[2,245926103566]	491852207132	0.4886
26	[2,656175474498]	1312350948996	0.4123
27	[3881642290710]	3881642290710	0.3856
28	[2,2,2,1607591023742]	12860728189936	0.4040
29	[2,17634301773068]	35268603546136	0.3504
30	[125355959329602]	125355959329602	0.3938

Table B.15: Class Groups for $D = -(10^n + 3)$, $n = 2$ to 30

D	$Cl(D)$
31	[4,4,4,8240136101284]
32	[2,955448419266184]
33	[2,2,2,460137245957140]
34	[2,5634726462792888]
35	[2,15989145880745520]
36	[2,2,30129084013196016]
37	[557309419766348976]
38	[2,847971386413711124]
39	[5044956409536984867]
40	[2,2,2884234075561655648]
41	[2,2,2,3783297264385115820]
42	[106475021887373986546]
43	[543203506831569725730]
44	[2,2,367520648573728999642]
45	[2,2,4,220637056338713949948]
46	[2,2,2662932604570235968950]
47	[2,20555090011213992064 74]
48	[2,2,35422488947049442222270]
49	[2,2,2,62670237779959002150728]
51	[2,4,39506816087578383249 4708]
52	[2,56366248169651633507 23552]
53	[2,19485863439714150244338844]
54	[96986139815112762181153038]
55	[2,2,154720759596540679388678070]
57	[2,19595660528265735919078031 70]
60	[2,2,339379394009992997634908 20322]
61	[2,2,2,2,2,2,866869651982910165280018 7856]

Table B.16: Class Groups for $D = -(10^n + 3)$, $n > 30$

n	$Cl(D)$	$h(D)$	$L(1, \chi_D)$
2	[14]	14	2.1882
3	[2,2,10]	40	1.9859
4	[4,40]	160	2.5131
5	[2,230]	460	2.2849
6	[2,516]	1032	1.6211
7	[2,1446]	2892	1.4365
8	[4,4104]	16416	2.5786
9	[2,2,2,2,2560]	40960	2.0346
10	[2,2,48396]	193584	3.0408
11	[2,2,2,56772]	454176	2.2560
12	[2,4,117360]	938880	1.4748
13	[2,2,742228]	2968912	1.4747
14	[2,2,4,1159048]	18544768	2.9130
15	[2,2,2,2,2,4,257448]	32953344	1.6369
16	[2,2,2,2,11809616]	188953856	2.9681
17	[2,2,2,46854696]	374837568	1.8619
18	[2,2,264135076]	1056540304	1.6596
19	[2,1649441906]	3298883812	1.6387
20	[2,2,2,1856197104]	14849576832	2.3326
21	[2,2,2,2,2,2,678293202]	43410764928	2.1563
22	[2,2,2,19870122100]	158960976800	2.4970
23	[2,2,2,2,23510740696]	376171851136	1.8686
24	[2,4,144373395240]	1154987161920	1.8142
25	[2,2,2,2,186902691564]	2990443065024	1.4854
26	[2,4,2062939290744]	16503514325952	2.5924
27	[2,2,2,2,2,2,596438010456]	38172032669184	1.8961
28	[2,4,4,4987045013072]	159585440418304	2.5068
29	[2,2,109151360534920]	436605442139680	2.1687
30	[2,2,2,2,2,8,4591263001512]	1175363328387072	1.8463

Table B.17: Class Groups for $D = -4(10^n + 1)$, $n = 2$ to 30

D	$Cl(D)$
31	[2,2056399348229026]
32	[2,2,4,1090942379155176]
33	[2,2,2,2,2,2,4,116002287037860]
34	[2,2,2,23706573772146012]
35	[2,2,2,2,29186559377772872]
36	[2,2,2,2,4,15799409333364584]
37	[2,2,2,315505817497815868]
38	[2,4,1995813614136301876]
39	[2,2,2,2,2,2,2,4,58778103675632604]
40	[2,2,8,5664549002512638320]
41	[2,2,90072072732125828328]
42	[2,2,2,2,2,2,14534047479909591484]
43	[2,2,2,380286840645249345372]
44	[2,4,8,256628302056496579584]
45	[2,2,2,2,2,2,2,2,2,2,2,17607541700271233832]
46	[2,2,2,21889597164067703269780]
48	[2,2,2,2,2,2,2,4,2054406711911308268064]
49	[2,2,2,3668343430337587343438888]
50	[2,2,2,2,2,4,144862768812972247716000]
52	[2,2,8,44917130652138448734271 52]
54	[2,2,2,2,2,33005468815882949353682176]
55	[2,2,2,2,2,2,2,2,11858517462807308421507372]
58	[2,2,2,2,92115760137865449165200171 88]
60	[2,2,2,2,2,37409900158832732901785147824]
65	[2,2,2,2,2,13043323078148845674146634826224]
68	[2,4,4,44308832343257551793954040643407 2]

Table B.18: Class Groups for $D = -4(10^n + 1)$, $n > 30$

Bibliography

[1] Tom M. Apostol, *Mathematical Analysis*, second ed., Addison-Wesley, 1974.

[2] ———, *Introduction to Analytic Number Theory*, Springer, 1976.

[3] László Babai, *Local expansion of vertex-transitive graphs and random generation in groups*, Proceedings of the 23rd ACM Symposium on the Theory of Computing, 1991, pp. 117–126.

[4] ———, *Randomization in group algorithms: Conceptual questions*, Groups and Computation II, DIMACS Workshops on Groups and Computation (L. Finkelstein and W. M. Kantor, eds.), American Mathematical Society, 1997.

[5] László Babai and Robert Beals, *A polynomial-time theory of black-box groups I.*, Groups St. Andrews 1997 in Bath, I, London Mathematical Society Lecture Notes Series, vol. 260, Cambridge University Press, 1999, pp. 30–64.

[6] László Babai, William M. Kantor, Péter P. Pálfy, and Ákos Seress, *Black-box recognition of finite simple groups of Lie type by statistics of element orders*, Journal of Group Theory **5** (2002), 383–401.

[7] László Babai and Endre Szemerédi, *On the complexity of matrix group problems*, Proceedings of the 25th IEEE Symposium on Foundations of Computer Science, 1984, pp. 229–240.

[8] Eric Bach, *Discrete logarithms and factoring*, Technical Report UCB/CSD 84/186, University of California, Berkeley, 1984.

[9] ———, *Analytic Methods in the Analysis and Design of Number-Theoretic Algorithms*, ACM Distinguished Dissertation 1984, MIT Press, 1985.

[10] ———, *Algorithmic Number Theory, Vol. 1: Efficient Algorithms*, MIT Press, 1996.

[11] Eric Bach and René Peralta, *Asymptotic semismoothness probabilities*, Mathematics of Computation **65** (1996), 1701–1715.

[12] Daniel J. Bernstein, *Detecting perfect powers in essentially linear time*, Mathematics of Computation **67** (1998), 1253–1283.

[13] ———, *Pippenger's exponentiation algorithm*, Available at `http://cr.yp.to/papers/pippenger.pdf`, 2001.

[14] Daniel J. Bernstein, Hendrik W. Lenstra, Jr., and Jonathan Pila, *Detecting perfect powers by factoring into coprimes*, Mathematics of Computation **76** (2007), 385–388.

[15] Hans Ulrich Besche, Bettina Eick, and E. A. O'Brien, *The groups of order at most 2000*, Electronic Research Announcements of the American Mathematical Society (2001).

[16] Richard Blecksmith, Michael McCullum, and J. L. Selfridge, *3-smooth representations of integers*, The American Mathematical Monthly **105** (1998), 529–543.

[17] Andrew R. Booker, *Quadratic class numbers and character sums*, Mathematics of Computation **75** (2006), 1481–1492.

[18] William W. Boone, *The word problem*, Annals of Mathematics **70** (1959), 207–265.

[19] D. W. Boyd and H. Kisilevsky, *On the exponent of the ideal class group of complex quadratic fields*, Proceedings of the American Mathematical Society **31** (1972), 433–436.

[20] Sergey Bratus and Igor Pak, *Fast constructive recognition of a black-box group isomorphic to S_n or A_n using Goldbach's conjecture*, Journal of Symbolic Computation **29** (2000), 33–57.

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

[22] Ernest F. Brickell, Daniel M. Gordon, Kevin S. McCurley, and David B. Wilson, *Fast exponentiation with precomputation*, Advances in Cryptology–EUROCRYPT '92, Lecture Notes in Computer Science, vol. 658, Springer-Verlag, 1992, pp. 200–207.

[23] J. Brillhart, D. Lehmer, J. Selfridge, B. Tuckerman, and S. S. Wagstaff, Jr., *Factorizations of $b^n \pm 1$, $b = 2, 3, 5, 6, 7, 10, 11, 12$ up to High Powers*, third ed., Contemporary Mathematics, vol. 22, American Mathematical Society, 2002.

[24] Johannes Buchmann and Stephan Düllmann, *Distributed class group computation*, Informatik **1** (1992), 69–79.

[25] Johannes Buchmann and Safuat Hamdy, *A survey on IQ cryptography*, Public-key Cryptography and Computational Number Theory (K. Alster, J. Urbanowicz, and H.C. Williams, eds.), Springer-Verlag, 2000, pp. 234–247.

[26] Johannes Buchmann, Michael J. Jacobson, Jr., Stefan Neis, Patrick Theobald, and Damian Weber, *Sieving methods for class group computation*, Proceedings of Algorithmic Algebra and Number Theory (Heidelberg, 1997), Springer, 1999, pp. 3–10.

[27] Johannes Buchmann, Michael J. Jacobson, Jr., and Edlyn Teske, *On some computational problems in finite abelian groups*, Mathematics of Computation **66** (1997), 1663–1687.

[28] Johannes Buchmann and Arthur Schmidt, *Computing the structure of a finite abelian group*, Mathematics of Computation **74** (2005), 2017–2026.

[29] Johannes Buchmann, Tsuyoshi Takagi, and Ulrich Vollmer, *Number field cryptography*, High Primes & Misdemeanors: Lectures in Honour of the 60th Birthday of Hugh Cowie Williams (van der Poorten and Stein, eds.), Fields Institute Communications, vol. 41, American Mathematical Society, 2004, pp. 111–125.

[30] Aleksandr A. Buchstab, *Asymptotic estimates of a general number theoretic function*, Matematicheskiĭ Sbornik **44** (1937), 1239–1246.

[31] Duncan A. Buell, *Binary Quadratic Forms: Classical Theory and Modern Computations*, Springer-Verlag, 1989.

[32] E. Canfield, P. Erdös, and C. Pomerance, *On a problem of Oppenheim concerning "factorisatio numerorum"*, Journal of Number Theory **17** (1983), 1–28.

[33] Frank Celler and C. R. Leedham-Green, *Calculating the order of an invertible matrix*, Groups and Computation II, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 28, American Mathematical Society, 1997, pp. 55–60.

[34] Frank Celler, C. R. Leedham-Green, Scott H. Murray, Alice C. Niemeyer, and E. A. O'Brien, *Generating random elements of a finite group*, Communications in Algebra **23** (1995), 4931–4948.

[35] A.Y. Cheer and D.A. Goldston, *A differential delay equation arising from the sieve of Eratosthenes*, Mathematics of Computation **55** (1990), 129–141.

[36] S. Chowla, *An extension of Heilbronn's class number theorem*, Quarterly Journal of Mathematics, Oxford **5** (1934), 304–307.

[37] H. Cohen, *A Course in Computational Algebraic Number Theory*, Springer, 1996.

[38] H. Cohen and H. W. Lenstra, Jr., *Heuristics on Class Groups of Number Fields*, Number Theory, Noordwijkerhout 1983, Lecture Notes in Mathematics, vol. 1068, Springer-Verlag, 1984, pp. 33–62.

[39] Gene Cooperman, *Towards a practical, theoretically sound algorithm for random generation in finite groups*, 2007, Available at `http://arxiv.org/PS_cache/math/pdf/0205/0205203v1.pdf`.

[40] Richard Crandall and Carl Pomerance, *Prime Numbers: A Computational Perspective*, second ed., Springer, 2005.

[41] D. E. Denning, *Cryptography and Data Security*, Addison-Wesley, 1982.

[42] K. Dickman, *On the frequency of numbers containing prime factors of a certain relative magnitude*, Arkiv för Mathematik, Astronomi, och Fysik, 22A **10** (1930), 1–14.

[43] Jean Dieudonné, *Infinitesimal Calculus*, Hermann, 1971, Translated from the French text *Calcul infinitésimal* (1968).

[44] J. D. Dixon, *The probability of generating the symmetric group*, Math. Z. **110** (1969), 199–205.

[45] A. G. Earnest and O. H. Körner, *On ideal class groups of 2-power exponent*, Proceedings of the American Mathematical Society **86** (1982), 196–198.

[46] Alfred J. Menezes (editor), Ian F. Blake, XuHong Gao, Ronald C. Mullin, Scott A. Vanstone, and Tomik Yaghoobian, *Application of Finite Fields*, Kluwer Academic Publishers, 1993.

[47] T. ElGamal, *A public-key cryptosystem and a signature scheme based on discrete logarithms.*, IEEE Transactions on Information Theory **31** (1985), 469–472.

[48] V. Ennola, *On numbers with small prime divisors*, Annales Academiae Scientiarum Fennicae Series A I **440** (1969), 1–16.

[49] Larry Finklestein and William Kantor (eds.), *Groups and Computation I: Workshop on Groups and Computation October 7-10, 1991*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 11, American Mathematical Society, 1993.

[50] Larry Finklestein and William Kantor (eds.), *Groups and Computation II: Workshop on Groups and Computation June 7-10, 1995*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 28, American Mathematical Society, 1997.

[51] Daniel M. Gordon, *A survey of fast exponentiation methods*, Journal of Algorithms **27** (1998), 129–146.

[52] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik, *Concrete Mathematics: A Foundation for Computer Science*, second ed., Addison-Wesley, 1994.

[53] Torbjörn Granlund et al., *GNU Multiple Precision Arithmetic Library 4.2.1*, May 2006, `http://swox.com/gmp/`.

[54] Andrew Granville, *Smooth numbers: Computational number theory and beyond*, 2006, Available at `http://www.dms.umontreal.ca/~andrew/PDF/msrire.pdf`.

[55] J. Hafner and K. McCurley, *A rigorous subexponential algorithm for computation of class groups*, Journal of the American Mathematical Society **2** (1989), 837–850.

[56] Safuat Hamdy and Bodo Möller, *Security of cryptosystems based on class groups of imaginary quadratic orders*, Advances in Cryptology–ASIACRYPT 2000, LNCS, vol. 1976, 2000, pp. 234–247.

[57] Darrel Hankerson, Alfred Menezes, and Scott Vanstone, *Guide to Elliptic Curve Cryptography*, Springer, 2004.

[58] G. H. Hardy and E. M. Wright, *An Introduction to the Theory of Numbers*, fifth ed., Oxford Science Publications, 1979.

[59] Kurt Heegner, *Diophantische analysis und modulfunktionen*, Mathematische Zeitschrift **56** (1952), 227–253.

[60] Adolf Hildebrand, *The asymptotic behavior of the solutions of a class of differential-difference equations*, Journal of the London Mathematical Society **42** (1990), 11–31.

[61] Adolf Hildebrand and Gerald Tenenbaum, *On integers free of large prime factors*, Transactions of the American Mathematical Society **296** (1986), 265–290.

[62] ———, *Integers without large prime factors*, Journal de Théorie des Nombres **5** (1993), 411–484.

[63] P.E. Holmes, S.A. Linton, E.A. O'Brien, A.J.E. Ryba, and R.A. Wilson, *Constructive membership in black-box groups*, 2004, revised 2006, available at `http://www.maths.qmul.ac.uk/7Eraw/pubs_files/rybalg.pdf`.

[64] M. J. Jacobson, Jr., R. Scheidler, and H. C. Williams, *The efficiency and security of a real quadratic field based-key exchange protocol*, Public-Key Cryptography and Computational Number Theory (Warsaw, Poland), de Gruyter, 2001, pp. 89–112.

[65] Michael J. Jacobson, Jr., *Applying sieving to the computation of quadratic class groups*, Mathematics of Computation **68** (1999), 859–867.

[66] _____, *Computing discrete logarithms in quadratic orders*, Journal of Cryptography **13** (2000), 473–492.

[67] Michael J. Jacobson, Jr. and Alfred J. van der Poorten, *Computational aspects of NUCOMP*, Algorithmic Number Theory, ANTS-V (Claus Fieker and David R. Kohel, eds.), Lecture Notes in Computer Science, vol. 2369, Springer-Verlag, 2002, pp. 120–133.

[68] Donald E. Knuth, *The Art of Computer Programming, Volume II: Seminumerical Algorithms*, third ed., Addison-Wesley, 1998.

[69] Donald E. Knuth and Luis Trabb Pardo, *Analysis of a simple factorization algorithm*, Theoretical Computer Science **3** (1976), 321–348.

[70] E. Landau, *Über die maximalordnung der permutationen gegebenen grades*, Archiv der Math. und Phys. (1903), 92–103.

[71] M. W. Liebeck and A. Shalev, *The probability of generating a finite simple group*, Geom. Dedicata **56** (1995), 103–113.

[72] George Marsaglia, Arif Zaman, and John C. W. Marsaglia, *Numerical solution of some classical differential-difference equations*, Mathematics of Computation **53** (1989), 191–201.

[73] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone, *Handbook of Applied Cryptography*, CRC Press, 1997, revised reprint.

[74] Daniele Micciancio, *The RSA group is pseudo-free*, Advances in Cryptology–EUROCRYPT 2005, Lecture Notes in Computer Science, vol. 3494, Springer, 2005, pp. 387–403.

[75] Gary L. Miller, *Riemann's hypothesis and tests for primality*, Journal of Computer and System Sciences **13** (1976), 300–317.

[76] Bodo Möller, *Algorithms for multi-exponentiation*, Tech. Report TI-8/01, TU Darmstadt, Fachbereich Informatik, 2001.

[77] Melvyn B. Nathanson, *Elementary Methods in Number Theory*, Springer, 2000.

[78] Ivan Niven, *Fermat's theorem for matrices*, Duke Mathematics Journal **15** (1948), 823–826.

[79] Pyotr Sergeyevich Novikov, *On the algorithmic unsolvability of the word problem in group theory*, Trudy Mat. Inst. Steklov 44 **44** (1955), 1–143, English text in Russian Translations, 9(1958).

[80] Jorge Olivos, *On vectorial addition chains*, Journal of Algorithms **2** (1981), 13–21.

[81] Igor Pak, *On probability of generating a finite group*, 1999, preprint.

[82] _____, *The product replacement algorithm is polynomial*, Proceedings of the 41st IEEE Symposium on Foundations of Computer Science, 2000, pp. 476–485.

[83] John M. Pollard, *Theorems on factorization and primality testing*, Proceedings of the Cambridge Philosophical Society **76** (1974), 521–528.

[84] _____, *Monte Carlo methods for index computations mod p*, Mathematics of Computation **32** (1978), 918–924.

[85] Carl Pomerance, *The expected number of random elements to generate a finite abelian group*, Periodica Mathematica Hungarica **43** (2001), 191–198.

[86] L. Pyber, *Enumerating finite groups of given order*, Annals of Mathematics **137** (1993), 203–220.

[87] Michael Rabin, *Probabilistic algorithms for testing primality*, Journal of Number Theory **12** (1980), 128–138.

[88] Paulo Reibenboim, *The Book of Prime Number Records*, second ed., Springer-Verlag, 1989.

[89] Ronald L. Rivest, *On the notion of pseudo-free groups*, Proceedings of the First Theory of Cryptography Conference–TCC 2004 (M. Naor, ed.), 2004.

[90] J. B. Rosser and L. Schoenfeld, *Approximate formulas for some functions of prime numbers*, Illinois Journal of Mathematics **6** (1962), 64–94.

[91] J. Sattler and C.P. Schnorr, *Generating random walks in groups*, Ann.-Univ.-Scie.-Budapest.-Sect.-Comput. **6** (1985), 65–79.

[92] C.P. Schnorr and H.W. Lenstra, Jr., *A Monte Carlo factoring algorithm with linear storage*, Mathematics of Computation **43** (1984), 289–311.

[93] Robert Sedgewick and Thomas G. Szymanski, *The complexity of finding periods*, Proceedings of the 11th ACM Symposium on the Theory of Computing, ACM Press, 1979, pp. 74–80.

[94] Ákos Seress, *Permutation Group Algorithms*, Cambridge Tracts in Mathematics, no. 152, Cambridge University Press, 2003.

[95] Donald Shanks, *Class number, a theory of factorization and genera*, Analytic Number Theory, Proceedings of Symposia on Pure Mathematics, vol. 20, American Mathematical Society, 1971, pp. 415–440.

[96] Victor Shoup, *Lower bounds for discrete logarithms and related problems*, Advances in Cryptology–EUROCRYPT '97, Springer-Verlag, 1997, revised version, pp. 256–266.

[97] _____, *A Computational Introduction to Number Theory and Algebra*, Cambridge University Press, 2005.

[98] N. J. A. Sloane, *The On-line Encyclopedia of Integer Sequences*, 2007, `www.research.att.com/~njas/sequences/`.

[99] I. M. Sobol, *On periods of pseudo-random sequences*, Theory of Probability and its Applications **9** (1964), 333–338.

[100] Richard Stallman et al., *GNU Compiler Collection 3.4.2*, May 2006, `http://gcc.gnu.org/index.html`.

[101] Harold M. Stark, *A complete determination of complex quadratic fields of class-number one*, Michigan Mathematics Journal **14** (1967), 1–27.

[102] Andreas Stein and Edlyn Teske, *Optimized baby step-giant step methods*, Journal of the Ramanujan Mathematical Society **20** (2005), 1–32.

[103] V. Strassen, *Einige resultate über berechnungskomplexität*, Jahresber Deutsch. Math.-Verein. **78** (1976), 1–8.

[104] Ernst G. Straus, *Addition chains of vectors*, The American Mathematical Monthly **70** (1964), 806–808.

[105] Andrew V. Sutherland, *Faster generic group algorithms for order determination and the discrete logarithm*, 2007, preprint.

[106] Koji Suzuki, *Approximating the number of integers without large prime factors*, Mathematics of Computation **75** (2005), 1015–1024.

[107] Gérald Tenenbaum, *Introduction to analytic and probabilistic number theory*, Cambridge Studies in Advanced Mathematics, vol. 46, Cambridge University Press, 1995, Originally published in French as *Introduction à la theórie analytique et probabiliste des nombres*, 1990.

[108] David C. Terr, *A modification of Shanks' baby-step giant-step algorithm*, Mathematics of Computation **69** (2000), 767–773.

[109] Edlyn Teske, *A space efficient algorithm for group structure computation*, Mathematics of Computation **67** (1998), 1637–1663.

[110] ———, *Speeding up Pollard's rho method for computing discrete logarithms*, Algorithmic Number Theory Seminar ANTS-III, Lecture Notes in Computer Science, vol. 1423, Springer-Verlag, 1998, pp. 541–554.

[111] ———, *On random walks for Pollard's rho method*, Mathematics of Computation **70** (2001), 809–825.

[112] ———, *Square-root algorithms for the discrete logarithm problem (a survey)*, Public Key Cryptography and Computational Number Theory, Walter de Gruyter, 2001, pp. 283–301.

[113] Paul C. van Oorschot and Michael J. Wiener, *Parallel collision search with cryptanalytic applications*, Journal of Cryptology **12** (1999), 1–28.

[114] Robert Wilson et al., *ATLAS of Finite Group Representations*, 2006, `http://brauer.maths.qmul.ac.uk/Atlas/v3/`.

[115] Andrew C. Yao, *On the evaluation of powers*, SIAM Journal of Computing **5** (1976), 100–103.