ALGORITHMS IN NUMBER THEORY

by

Jonathan Paul Sorenson

Computer Sciences Technical Report #1027

June 1991

Algorithms in Number Theory

by

Jonathan Paul Sorenson

A thesis submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy
(Computer Sciences)

at the
University of Wisconsin-Madison
1991

Abstract

This thesis discusses four sets of results on number theoretic algorithms.

A positive integer n is a perfect power if there exist integers $a, b > 1$ such that $n = a^b$. We give a new sieve algorithm to recognize perfect powers. This algorithm has a linear median running time, and it has a subquadratic average running time assuming the extended Riemann hypothesis. Previous algorithms to recognize perfect powers require at least cubic average time. We also show unconditionally that the set of perfect powers is in the parallel complexity class $\mathcal{NC}^2$. This is joint work with Eric Bach.

We give the first randomized parallel algorithms that run in polylog time using a subexponential number of processors for factoring integers and computing discrete logarithms over finite fields of prime order. This solves a problem posed by Adleman and Kompella in 1988.

We present the new k-ary greatest commmon divisor (GCD) algorithm, a generalization of the binary algorithm of Stein. In a sequential implementation, this algorithm outperformed many previous algorithms, including the binary algorithm and the Euclidean algorithm. In addition, the complexity of a parallel version of the k-ary GCD algorithm matches that of the best previous parallel GCD algorithms.

And finally, we analyze the number of integers factorable in random polynomial time using Pollard's $p-1$ factoring algorithm. We show that, in random polynomial time, the $p-1$ method can factor more integers than trial division but fewer than Lenstra's elliptic curve method. This is joint work with Carl Pomerance.

Acknowledgements

There are many people who have helped me through the study and research that led to this thesis, and I wish to express my deepest gratitude by acknowledging them.

Eric Bach, both advisor and friend, has taught me a great deal. Working with him has been truly enjoyable. Anne Condon, Debby Joseph, and Prasoon Tiwari have offered me excellent advice, encouragement, and taught me much as well. Marty Isaacs provided me excellent instruction in Algebra.

Jeffrey Shallit taught an exciting course on number theoretic algorithms in the fall of 1989, from which I learned much. A project I did as part of that course led to the work on greatest common divisor algorithms presented in Chapter 4.

Working with Carl Pomerance has been a priviledge and a pleasure. His insightful ideas for improving my earlier work led to a fruitfull collaboration, and some of our results appear in Chapter 5.

Joao, Narendran, Sam, Marty, Victor, Meera, Das, Kirk, Giri, Judy, Gary, and Ghoshal, sat through many practice talks, helped me work out ideas, and read drafts of papers.

My friends and family, especially my parents Norma and John, provided encouragement and support.

The faculty at Valparaiso University, particularly the Mathematics/Computer Science and Physics departments, gave me a good undergraduate education.

Most importantly, my wife Chelle provided support, understanding, tolerance, and love.

I also acknowledge the support of the National Science Foundation under grant CCR-8552596, a Teaching Incentive Grant provided by the General Electric Foundation, and a fellowship provided by the Wisconsin Alumni Research Foundation.

Contents

Chapter 1

Introduction

Number theory is the study of the additive and multiplicative properties of integers. One basic concept in number theory is that of a *prime number*, which is a number with no divisors aside from itself and 1. The *fundamental theorem of arithmetic* says that every positive integer can be expressed in precisely one way as the product of powers of prime numbers.

Algorithmic number theory is the design and analysis of algorithms for solving problems that arise in number theory. Naturally, one of the more important such problems is the *complete factorization problem* which arises from the fundamental theorem of arithmetic: given an integer, factor it into a product of powers of primes. Other problems include determining if an integer is prime, computing the greatest common divisor of two integers, and computing discrete logarithms over finite fields.

The study of number theoretic algorithms has a long history. For example, over 2000 years ago Euclid described what is now called the Euclidean algorithm for computing greatest common divisors (see Knuth [Knu81b, Section 4.5.2]). This algorithm withstood the test of time; even today, in many circumstances it is the algorithm of choice.

The relatively recent invention of the digital computer has sparked new interest and new developments in this field. For example, the binary algorithm of Stein [Ste67] for computing greatest common divisors is, on most computers, more efficient than Euclid's algorithm. In addition, number theoretic algorithms now have applications in the areas of cryptography, algebraic and symbolic computation, pseudo-random number generation, and complexity theory.

In this thesis we will present:

- new algorithms for determining if an integer is a perfect power,
- new parallel algorithms for factoring integers and computing discrete logarithms,
- a new algorithm for computing greatest common divisors,
- and a new analysis of an integer factoring algorithm.

The rest of this introduction is organized as follows. In the next section we give a detailed overview of the results outlined above. We then review some notation and background material in number theory and computation theory. Finally, we present several fundamental number theoretic algorithms.

1.1 An Overview

As we mentioned above, this thesis discusses four sets of results on number theoretic algorithms. These results are presented in four chapters beginning with chapter 2, and each of these chapters may be read independently of the others. Below we give an overview of this material.

An integer n is a perfect power if there exist integers a and b, both greater than 1, such that $n = a^b$. The correctness of many integer factoring algorithms and primality tests requires that the input not be of the form p^e where p is prime and $e \geq 2$. Since all numbers of this form are perfect powers, an algorithm to recognize perfect powers can be used to screen out such inputs for factoring and primality testing algorithms.

In chapter 2, we give several algorithms to recognize perfect powers, including both an efficient sieve algorithm and the first unconditional $\mathcal{NC}$ algorithm ($\mathcal{NC}$ is Nick's class, a parallel complexity class; see section 1.3). The sieve algorithm has a subquadratic average running time, assuming the extended Riemann hypothesis (ERH), a cubic worst-case running time, and a linear median running time. Previous $\mathcal{NC}$ algorithms rely on the ERH for correctness [Mil89], and the best previous sequential algorithm (which seems to be folklore) has a supercubic running time.

Our new sieve algorithm requires some precomputation, but it is practical and very efficient.

This is joint work with Eric Bach, and appeared previously in [BS89b].

Let G be a cyclic group written multiplicatively, and let g generate G. The discrete logarithm problem is, given a element a of G, compute an integer x such that $g^x = a$. Typically G is taken to be the multiplicative group of a finite field.

In 1988, L. Adleman and K. Kompella asked if there are unbounded fan-in, probabilistic boolean circuits for factoring integers or computing discrete logarithms having both polylog depth and subexponential size [AK88]. We answer this question in the affirmative in chapter 3. We also give sublinear depth, subexponential size circuits with bounded fan-in for both problems.

Previous boolean circuits of subexponential size to solve either problem require a superlinear depth (see [AK88]).

This work appeared previously in [Sor89].

The greatest common divisor (GCD) of two integers u and v is the largest integer d such that d divides both u and v. In chapter 4, we describe the new k-ary greatest common divisor algorithm, a generalization of the binary algorithm of Stein. This algorithm is notable for two reasons, which are discussed below.

First, in a sequential, multiple-precision implementation, the k-ary GCD algorithm outperformed many of the best previous algorithms, including the binary algorithm and the

Euclidean algorithm. We show that this new algorithm has an $O(n^2/\log k)$ worst-case running time on n-bit inputs when k is a prime power. When k is chosen to be roughly n, this yields an algorithm with a subquadratic running time. Most previous GCD algorithms, such as the Euclidean and binary algorithms, require quadratic time.

Second, it leads to several interesting parallel algorithms. For the concurrent read, concurrent write (CRCW) parallel random access machine (PRAM) model, the k-ary GCD algorithm gives both an $O_\epsilon(n/\log n)$ time, $O(n^{1+\epsilon})$ processor algorithm and an $O(\log^d n)$ time $\exp[O(n/\log^d n)]$ processor algorithm, for $d \geq 1$. Both of these algorithms match the best previous parallel GCD algorithms due to Chor and Goldreich [CG90]. There are also exclusive read, exclusive write (EREW) and concurrent read, exclusive write (CREW) versions of the algorithm.

This work appeared previously in [Sor90c].

Many integer factoring algorithms have exponential worst-case running times, and yet for inputs of a certain form they run in polynomial time. One simple example is the trial division algorithm, which can quickly factor integers with small prime divisors. The question, then, is how many integers can an algorithm factor in polynomial time?

In chapter 5, we answer this question for the $p-1$ algorithm of Pollard. In the process we show that, in random polynomial time, the $p-1$ method factors more integers than trial division but fewer than the elliptic curve method.

Trial division and Lenstra's elliptic curve method were analyzed previously by J. Hafner and K. McCurley [HM89].

This is joint work with Carl Pomerance, and is an improvement of [Sor90a].

1.2 Number Theory Background

In this section we review some concepts and facts from number theory. Throughout this thesis, we use $\log x$ to denote the natural logarithm of x, and $\lg x$ to denote the binary logarithm of x. We also write $\log^k x$ to mean $(\log x)^k$. If S is a set, we write $\#S$ to denote the cardinality of S. If x is a real number, we write $\lfloor x \rfloor$ to denote the largest integer less than or equal to x, and $\lceil x \rceil$ to denote the smallest integer greater than or equal to x.

1.2.1 Order Notation

Let f and g be real-valued functions. We say $f = O(g)$ if there exists an absolute constant $c > 0$ such that $f(x) < c \cdot g(x)$ for all $x > x_0$. Equivalently, $f \ll g$ means $f = O(g)$. $f = \Theta(g)$ if $f = O(g)$ and $g = O(f)$. $f = \Omega(g)$ if $g = O(f)$. Equivalently, $f \gg g$ means $f = \Omega(g)$. $f = o(g)$ if $\lim_{x\to\infty} f(x)/g(x) = 0$. We say $f \sim g$ if $\lim_{x\to\infty} f(x)/g(x) = 1$.

If O, Ω, Θ, $\gg$, or $\ll$ is subscripted by a variable, for example $f = O_\epsilon(g)$, then the implied constant is a function of that variable.

Notice that $\lg x = \Theta(\log x)$, so the two are interchangeable with respect to O, Θ, and Ω.

1.2.2 Divisibility and Primality

Let $\mathbf{Z}$ denote the ring of integers.

For positive integers u and v, $u \bmod v$ is the remainder when u is divided by v, that is, $u \bmod v = u - \lfloor u/v \rfloor \cdot v$. We say $v \mid u$ (v divides u) if $u \bmod v = 0$, and $v^e \parallel u$ (v^e fully divides u) if $v^e \mid u$ and $v^{e+1} \not| u$.

An integer p is *prime* if $p > 1$ and the only positive integers than divide p are itself and 1. An integer that is not prime is *composite.* Throughout this thesis, p will always denote a prime, and q will always denote a prime or a power of a prime.

Theorem 1.2.1 *There are infinitely many primes.*

Theorem 1.2.2 (Fundamental Theorem of Arithmetic) *For every integer $n > 1$ there corresponds a unique prime factorization*

$$n = p_1^{e_1} \cdots p_r^{e_r}$$

where the $p_1, \ldots, p_r$ are distinct prime numbers, and the $e_i > 0$.

The largest number of distinct prime divisors an integer n can have is $O(\log n/\log\log n)$. In fact, the average number of prime divisors is $O(\log\log n)$; see Hardy and Wright[HW79]. See also Ireland and Rosen [IR82].

1.2.3 The Ring $\mathbf{Z}/(n)$

We say a is congruent to b modulo n, written $a \equiv b \pmod{n}$, if $n \mid a - b$, or equivalently if $a \bmod n = b \bmod n$. Congruence modulo n for any positive integer n is an equivalence relation, and the set of equivalence classes, the integers modulo n, is denoted by $\mathbf{Z}/(n)$. Typically, $\mathbf{Z}/(n)$ is represented as the integers from 0 to $n-1$. $\mathbf{Z}/(n)$ is a cyclic group of order n under addition modulo n. Other common notations for the integers modulo n include $\mathbf{Z}/n\mathbf{Z}$ and $\mathbf{Z}_n$.

The greatest common divisor (GCD) of two integers u and v is the largest integer d such that $d \mid u$ and $d \mid v$. This is written $\gcd(u,v) = d$. Note that $\gcd(u,0) = u$. If $\gcd(u,v) = d$, then there exist integers x, y such that $xu + yv = d$. We say that u and v are relatively prime if $\gcd(u,v) = 1$.

Let $(\mathbf{Z}/(n))^*$ denote the subset of integers in $\mathbf{Z}/(n)$ that are relatively prime to n. Euler's totient function $\phi(n)$ is the cardinality of $(\mathbf{Z}/(n))^*$. So $\phi(n) = n-1$ if and only if n is prime.

Theorem 1.2.3 (Euler) *For every $a \in (\mathbf{Z}/(n))^*$, $a^{\phi(n)} \equiv 1 \pmod{n}$.*

Corollary 1.2.4 (Fermat's Little Theorem) *If p is prime, then for every $a \in (\mathbf{Z}/(p))^*$, $a^{p-1} \equiv 1 \pmod{p}$.*

If n and m are relatively prime, then $\phi(nm) = \phi(n)\cdot\phi(m)$. For p a prime, $\phi(p^e) = p^{e-1}(p-1)$. $(\mathbf{Z}/(n))^*$ is an abelian group under multiplication modulo n. For p a prime, $e \geq 1$, $(\mathbf{Z}/(p^e))^*$ is cyclic, and $\mathbf{Z}/(p)$ is a finite field of order p, sometimes written as $GF(p)$.

Theorem 1.2.5 (Chinese Remainder Theorem) *Let n and m be relatively prime integers. Then we have*

$$\mathbf{Z}/(nm) \quad \cong \quad \mathbf{Z}/(n) \oplus \mathbf{Z}/(m).$$

This may be applied recursively so that, if $n = p_1^{e_1} \cdots p_r^{e_r}$, then

$$\mathbf{Z}/(n) \quad \cong \quad \mathbf{Z}/(p_1^{e_1}) \oplus \cdots \oplus \mathbf{Z}/(p_r^{e_r}).$$

For a reference, see Hardy and Wright[HW79] or Ireland and Rosen[IR82].

1.2.4 The Distribution of Primes

Let $\pi(x)$ denote the number of primes up to x, and let $Li(x) = \int_2^\infty (\log t)^{-1} dt$. Note that $Li(x) \sim x/\log x$. The following version of the prime number theorem is due to Vinogradov.

Theorem 1.2.6 (Prime Number Theorem) *There exists an absolute constant $c > 0$ such that*

$$\pi(x) \quad = \quad Li(x) + O(x \exp[-c(\log x)^{3/5}]).$$

The following is equivalent to the prime number theorem:

$$\sum_{p \le x} \log p \quad \sim \quad x.$$

Sums over the variables p and q are always assumed to be sums over primes. We also have the following estimates, the second of which is Mertens's Theorem:

$$\sum_{p \le x} \frac{1}{p} \quad = \quad \log\log x + B + o(1);$$

$$\prod_{p \le x} \left(1 - \frac{1}{p}\right) \quad \sim \quad \frac{e^{-\gamma}}{\log x}.$$

Here $\gamma = 0.57721 \cdots$ is Euler's constant and $B = 0.26149 \cdots$. See Rosser and Schoenfeld [RS62].

Let k and l be positive, relatively prime integers with $l < k$. Let $\pi_{k,l}(x)$ denote the number of primes $p \le x$ such that $p \equiv l \,(\bmod\, k)$. The following theorem was first proved by Dirichlet.

Theorem 1.2.7 (Prime Number Theorem for Arithmetic Progressions) *If l, k are fixed, relatively prime, positive integers with $l < k$, then*

$$\pi_{k,l}(x) \quad \sim \quad \frac{1}{\phi(k)} Li(x).$$

The Riemann hypothesis (RH) is the assumption that the nontrivial zeros of the Riemann zeta function lie on the line $\Re(s) = 1/2$ in the complex plane. It is equivalent to the (unproven) statement that $\pi(x) = Li(x) + O(\sqrt{x} \log x)$. The extended Riemann hypothesis (ERH) is the same assumption about the zeros of Dirichlet L-functions, and is equivalent to a similarly strong version of the prime number theorem for arithmetic progressions. For more see Bach, Giesbrecht, and McInnes [BGM91], Davenport [Dav80], and Ingham [Ing32].

1.2.5 Smooth Numbers

A positive integer n is y-smooth if every prime divisor of n is at most y. Let $\Psi(x,y)$ denote the number of integers $\leq x$ that are y-smooth.

The Dickman ρ-function is defined as the continuous solution to the equations

$$\begin{aligned} \rho(u) &= 1 \quad \text{for } 0 \leq u \leq 1, \\ \rho'(u) &= -\rho(u-1)/u \quad \text{for } u \geq 1. \end{aligned}$$

De Bruijn [dB51] proved that

$$\rho(u) = \exp[-u\log u - u\log\log u + O(u)] = u^{-u(1+o(1))}.$$

For large x, y, it is well known that $\Psi(x,y)$ is roughly equal to $x\rho(u)$ where $u = \log x/\log y$; see for example Hildebrand [Hil86] and Canfield, Erdős, and Pomerance [CEP83].

1.2.6 Number Theoretic Problems

Below we define four of the more important problems of algorithmic number theory.

Greatest Common Divisors. Given as input positive integers u and v, compute the greatest common divisor of u and v.

The extended greatest common divisor problem is to compute integers x and y so that $xu + yv = \gcd(u,v)$. Notice if $\gcd(u,v) = 1$, then $x \equiv u^{-1} \pmod{v}$.

Primality Testing. Given as input an integer $n > 1$, determine whether n is prime or composite.

Integer Factoring. Given as input an integer $n > 1$, compute a list of powers of primes $p_1^{e_1}, \ldots, p_r^{e_r}$ such that their product is exactly n.

Often this problem is stated as finding a non-trivial divisor d of n if it exists; clearly any algorithm that solves this problem can be iterated to give the complete factorization. To avoid confusion, we will refer to finding a non-trivial divisor of n as *splitting n*, and we will refer to finding the complete prime power factorization of n as *completely factoring n*.

Discrete Logarithms. Given a cyclic group G, a generator g for G, and a group element $a \in G$, compute an integer x such that $g^x = a$.

In number theory, often G is taken to be $(\mathbf{Z}/(p))^*$ for p a prime.

For many more problems, see the survey paper by Adleman and McCurley [AM87].

1.3 Computation Theory Background

In this section, we review several standard models of computation and informally define several complexity classes.

1.3.1 Models of Computation

Let n be the length of the input to an algorithm. For example, if the input to an algorithm is a positive integer x, then $n = \lfloor \lg x \rfloor + 1$, the number of bits in the binary representation of x. Zero has a 1 bit representation. The goal of algorithm analysis is to determine the resources used by an algorithm, such as time and space, as a function of n. To do this rigorously requires a model of computation for describing the algorithm. Below we review several models used in this thesis.

Sequential Models

Perhaps the most popular model for number theoretic algorithms is the random access machine (RAM). This model consists of an infinite number of registers each of which represents an integer in binary, and a finite list of instructions. The model computes the value of a function by executing the instructions in the obvious way, using the registers for work space. Allowed instructions include reading or writing the contents of a register, conditional transfers of control (which implies a comparison), copying the contents of one register to another (with indirect addressing allowed), the testing or setting of individual bits in a register, and a fixed set of arithmetic operations. The usual arithmetic operations allowed are addition, subtraction, multiplication, and division with remainder.

We calculate the running time of a RAM program to be the sum of the times of each instruction executed in the computation. We calculate the space used by a RAM program to be some function of the total number of registers used.

Let u and v be two integers (stored in registers) with $|u| \geq |v|$, and let n and m be the number of bits in their binary representation. In the *naive bit complexity* model, we assign a cost to each instruction as follows:

- To compute $u \pm v$ costs $n + m$.
- To compute uv costs nm.
- To compute $\lfloor u/v \rfloor$ and $u \bmod v$ costs $(n - m + 1)m$.
- To compare u to v costs $n + m$.
- To read or write u or to copy u to another register costs n.

All other instructions, including bit operations, have a cost of 1. The running time is the sum of the costs of all instructions executed. The space used is the sum, over all registers used, of the maximum number of bits stored in each register. See Knuth [Knu81b] for descriptions of the classical algorithms for basic arithmetic operations.

In the *arithmetic complexity* model, all operations listed above have cost 1. The running time is the total number of instructions executed, and space is the number of registers used. It is assumed that register lengths are bounded by a polynomial in n.

For a more rigorous definition of the RAM model, see Bach and Shallit [BS90] or Aho, Hopcroft, and Ullman [AHU74]. Note that they charge for indexing and indirect addressing, where we do not.

We will also mention the Turing machine, which consists of a finite state control and an infinite, two-way tape with a read/write head. For this model, the running time is the number of states visited in the control, and space is the number of tape cells written. For more, see Hopcroft and Ullman [HU79].

Parallel Models

The parallel random access machine (PRAM) consists of an infinite number of processors, each of which is a sequential RAM. These processors execute instructions in lockstep and utilize a globally shared infinite number of memory registers. Since each register is essentially an array of bits, multiple processors may work together, each on a different part of the same register. This model has several variations depending on how read/write conflicts to shared memory are resolved:

- Exclusive-Read, Exclusive-Write (EREW): Processors are not allowed to conflict, or in other words, a conflict causes the machine to "crash."
- Concurrent-Read, Exclusive-Write (CREW): Multiple processors may read the same memory bit concurrently.
- Concurrent-Read, Concurrent-Write (CRCW): Concurrent writes are allowed. We assume that, if multiple processors write to the same memory bit at the same time, the resulting value of that bit is arbitrarily chosen to be one of the values written.

A *boolean circuit* is a directed, acyclic graph, where each node of the graph is called a *gate*. There are three different kinds of gates:

- Gates of indegree zero are the input gates. The value computed by the ith input gate is the ith bit of input to the circuit.
- Gates of outdegree zero are the output gates. These gates have indegree one. The input bit to the ith output gate becomes the ith bit of the output string.
- All other gates are labelled with one of the symbols $\vee$, $\wedge$, or $\neg$. The value computed by one of these gates is the logical *or*, *and*, or *not* of the input bits.

A *boolean circuit family* is a sequence of circuits, one for each input length.

A circuit family has *bounded fan-in* if the indegree of all gates of all circuits in the family is at most 2. Otherwise the circuit family is said to have *unbounded fan-in.*

We say a boolean circuit family is $S(n)$-space uniform if the circuit in the family for inputs of length n can be constructed by a deterministic, $O(S(n))$ space-bounded Turing machine.

The *size* of a circuit is the number of gates. The *depth* of a circuit is the length of the longest path from an input gate to an output gate.

Any "reasonable" algorithm described in terms of a boolean circuit family can be ported to the PRAM model. The basic idea is to assign one processor to each gate of the circuit, and the bits that travel on the edges or wires of the circuit are stored in the global shared memory. Conversely, a PRAM algorithm can be simulated on a boolean circuit. Here the idea is to construct subcircuits to perform each instruction and then "paste" the subcircuits together to form a circuit.

Informally, the depth of a bounded fan-in circuit corresponds (to within a constant factor) to the time of an EREW PRAM algorithm, and the depth of an unbounded fan-in circuit corresponds (to within a constant factor) to the time of a CRCW PRAM algorithm. The size of a circuit corresponds (to within a polynomial factor) to the number of processors used by a PRAM algorithm.

The complexity of the basic arithmetic operations for the PRAM model are as follows.

- The sum of two n-bit integers can be computed using an $O(\log n)$ time, $O(n)$ processor EREW PRAM algorithm or an $O(1)$ time, $O(n \log\log n)$ processor CRCW PRAM algorithm (see Chandra, Fortune, and Lipton [CFL85]).

- The product of two n-bit integers can be computed using an $O(\log n)$ time, $O(n \log n \log\log n)$ processor EREW PRAM algorithm (see Schönhage and Strassen [SS71]).

- The quotient and remainder of two n-bit integers can be computed using either $O(\log n \log\log n)$ time and $O(n \log n \log\log n)$ processors (see Reif and Tate [RT89]) or $O(\log n)$ time and $n^{O(1)}$ processors (see Beame, Cook, and Hoover [BCH86]).

Beame, Cook, and Hoover's division algorithm, when realized as a boolean circuit, is not known to be $O(\log n)$-space uniform.

For more on parallel models and algorithms, see Cook [Coo85], Karp and Ramachandran [KR90], and Greenlaw, Hoover, and Ruzzo [GHR91].

Probabilistic Computation

We also consider models of computation which use randomness as follows:

- For the sequential RAM, we add an instruction to generate a uniformly chosen integer from the interval $0, \ldots, a-1$. In the naive bit complexity model, the cost is $\lfloor \lg a \rfloor + 1$, and in the arithmetic complexity model, the cost is 1.

- For the PRAM, we allow each processor to generate one random bit (with a value of 0 or 1 with probability 1/2 each) at a time cost of 1. Several processors can be combined to generate larger random numbers.

- For boolean circuits, we add gates of indegree zero which produce a random bit as their output value.

1.3.2 Complexity Classes

Let S be a (potentially infinite) set of binary strings. An algorithm A computes the *characteristic function* for S if on input x, A outputs a 1 when $x \in S$, and A outputs a 0 when $x \notin S$. Typically, a complexity class is defined as a class of sets whose characteristic functions have algorithms that satisfy certain properties. Below we give some standard definitions of well-known complexity classes.

To classify problems that involve calculating the output of a function, such as computing greatest common divisors, we state a corresponding decision problem by specifying the bits of the binary representation of the function value. For example, we can define the set GCD to be the set of all triples (u, v, i) such that the ith bit of $\gcd(u, v)$ is equal to 1. If we have an algorithm to compute the characteristic function for such a set, then we can compute the actual function value by making multiple calls to the characteristic function. For a sequential computation model, these calls can be performed in a loop. For a parallel model, they may be done in parallel.

We say $f(n)$ is *polynomial* if there exists an integer c such that $f(n) = O(n^c)$. Let n be the length of the binary encoding of x. As mentioned above, if x is a positive integer then $n = \lfloor \lg x \rfloor + 1$.

Nick's class ($\mathcal{NC}$). For $k \geq 1$ an integer, $S \in \mathcal{NC}^k$ if there exists an EREW PRAM algorithm that runs in $O(\log^k n)$ time using a polynomial number of processors which accepts x if and only if $x \in S$.

We may also define $\mathcal{NC}$ in terms of boolean circuits: $S \in \mathcal{NC}^k$ if there exists an $O(\log n)$-space uniform, boolean circuit family of depth $O(\log^k n)$ and polynomial size which accepts x if and only if $x \in S$. By our observations about the equivalence of the PRAM and boolean circuit models, both definitions of $\mathcal{NC}^k$ are equivalent.

We define $\mathcal{NC} = \cup_{k=1}^{\infty} \mathcal{NC}^k$.

All the basic arithmetic operations $(+, -, \times, /)$ are in $\mathcal{NC}$. Computing integer square roots via Newton's method is also in $\mathcal{NC}$.

Polynomial Time ($\mathcal{P}$). $S \in \mathcal{P}$ if there exists a RAM or Turing Machine algorithm that runs in polynomial time that accepts x if and only if $x \in S$. Since a PRAM can be efficiently simulated sequentially by a RAM or Turing Machine, we have $\mathcal{NC} \subseteq \mathcal{P}$.

Greatest common divisors can be computed in $\mathcal{P}$, but no $\mathcal{NC}$ algorithm is known for this problem. If the ERH is true, then the set of primes is in $\mathcal{P}$ (see Miller [Mil76]).

Random Polynomial Time ($\mathcal{RP}$). $S \in \mathcal{RP}$ if there exists a randomized polynomial time RAM or Turing machine algorithm such that, if $x \in S$, the algorithm accepts x with probability at least 1/2, and if $x \notin S$, the algorithm always rejects. Clearly $\mathcal{P} \subseteq \mathcal{RP}$.

$S \in Co\text{-}\mathcal{RP}$ if the complement of S is in $\mathcal{RP}$.

The set of primes is in $\mathcal{RP} \cap Co\text{-}\mathcal{RP}$ (see Adleman and Huang [AH87]).

Non-deterministic Polynomial Time ($\mathcal{NP}$). $S \in \mathcal{NP}$ if there exists a set of ordered pairs $A \in \mathcal{P}$ such that $x \in S$ if and only if there exists a string y of length bounded by a fixed polynomial in n such that $(x, y) \in A$. In other words, $x \in S$ if there exists a "proof" y of x's membership in S that can be checked in polynomial time. Notice that $\mathcal{RP}$ can be formulated the same way, with the additional constraint that random coin tosses can be used to guess some appropriate value of y with probability at least 1/2. Thus $\mathcal{RP} \subseteq \mathcal{NP}$.

$S \in Co\text{-}\mathcal{NP}$ if the complement of S is in $\mathcal{NP}$.

Both integer factoring and computing discrete logarithms are in $\mathcal{NP}$. No polynomial time algorithm, even probabilistic or heuristic, is known for either problem.

Exponential Time ($\mathcal{EXP}$). $S \in \mathcal{EXP}$ if there exists a RAM or Turing Machine algorithm that runs in time $\exp[O(n)]$ which accepts x if and only if $x \in S$.

All problems discussed in this thesis are in $\mathcal{EXP}$.

Aside from the fact that $\mathcal{P} \neq \mathcal{EXP}$, it is unknown whether any of the class containments mentioned above are proper. For more, see Hopcroft and Ullman [HU79], Garey and Johnson [GJ79], Gill [Gil77], Karp and Ramachandran [KR90], and Greenlaw, Hoover, and Ruzzo [GHR91].

1.4 Basic Algorithms

We conclude our background material by briefly describing, in Pascal-like pseudocode, several fundamental, sequential number theoretic algorithms.

The first algorithm, the Sieve of Eratosthenes, finds all primes up to a bound n using $O(n \log\log n)$ arithmetic operations. This gives a running time of $O(n \log n \log\log n)$ in the naive bit complexity model (see section 1.3). The algorithm uses an array $s[\,]$ of n bits.

Sieve of Eratosthenes
Input: positive integer n

```
For $x := 2$ to $n$ do: $s[x] := 1$;
For $p := 2$ to $\sqrt{n}$ do:
    If $s[p] = 1$ then
        For $x := p^2$ to $n$ step $p$ do:
            $s[x] := 0$;
For $x := 2$ to $n$ do:
    If $s[x] = 1$ then output($x$);
```

Note that $\lfloor\sqrt{n}\rfloor$ can be found using binary search in at most $O(\log^3 n)$ bit operations. For more, see Bach and Shallit [BS90] and Sorenson [Sor90b].

The second algorithm, the Euclidean algorithm, computes the GCD of two positive integers u and v in $O(\log u \log v)$ bit operations.

Euclidean GCD Algorithm

Input: positive integers u, v

```
While $v \neq 0$ do:
    $r := u \bmod v$;
    $u := v$; $v := r$;
Output($u$);
```

For more, see Collins [Col74], Knuth [Knu81b], and Norton [Nor90].

The third algorithm computes $a^b \bmod m$ using the binary or repeated squaring method. It runs in $O(\log b)$ arithmetic operations, which is $O(\log b \log^2 m)$ bit operations.

Modular Exponentiation

Input: positive integers a, b, m

```
$x := 1$;
While $b > 0$ do:
    If $b$ is odd then $x := x \cdot a \bmod m$;
    $b := \lfloor b/2 \rfloor$;
    $a := a^2 \bmod m$;
Output($x$);
```

For more, see Bach and Shallit [BS90] and Knuth [Knu81b].

The fourth algorithm is a probabilistic compositeness test due to Miller and Rabin [Mil76, Rab80]. On input a positive integer n, if n is prime it outputs "Prime"; if n is composite it outputs "Prime" with probability $\leq 1/4$, and it outputs "Composite" with probability $\geq 3/4$. Note that the algorithm may be repeated to reduce the probability of error.

Miller-Rabin Primality/Compositeness Test

Input: positive integer n

```
Write $n - 1 = 2^e \cdot m$ with $m$ odd;
Choose $a \in \mathbf{Z}/(n)$ uniformly at random;
$x := a^m \bmod n$;
If $x \equiv \pm 1 \pmod{n}$ then Output("Prime") and Halt;
For $i := 1$ to $e - 1$ do:
    $x := x^2 \bmod n$;
    If $x \equiv -1 \pmod{n}$ then Output("Prime") and Halt;
Output("Composite");
```

This algorithm uses $O(\log n)$ arithmetic operations, or $O(\log^3 n)$ bit operations. For details, see Miller [Mil76], Rabin [Rab80], or Bach and Shallit [BS90].

The fifth and final algorithm, trial division, removes prime divisors below a bound y from an integer n.

Trial Division
Input: positive integer n and bound y

```
    Find the primes up to $y$ using the Sieve of Eratosthenes;
    For each prime $p \leq y$ do:
        While $p \mid n$ do:
            Output($p$);
            $n := n/p$;
    Output($n$);
```

This algorithm requires $O(y \log\log y + y \log n + \log^2 n)$ bit operations. If one is interested in arithmetic complexity, the use of the sieve of Eratosthenes should be removed, and all integers up to y should be used as trial divisors; the number of arithmetic operations used by this modified algorithm is $O(y + \log n)$.

A primality test may be added so that the algorithm will halt if n is discovered to be prime. Possible choices for a primality test include those of Adleman, Pomerance, and Rumely [APR83], Goldwasser and Kilian [GK86], Miller-Rabin (discussed above) [Mil76, Rab80], or Solovay and Strassen [SS77].

Trial Division with Primality Test
Input: positive integer n and bound y

```
    If $n$ is prime then Output($n$) and Halt;
    Find the primes up to $y$ using the Sieve of Eratosthenes;
    For each prime $p \leq y$ do:
        While $p \mid n$ do:
            Output($p$);
            $n := n/p$;
            If $n$ is prime then Output($n$) and Halt;
    Output($n$);
```

Let $t(n)$ denote the cost of one primality test. Then the number of bit operations is at most $O(y \log\log y + y \log n + \log^2 n + t(n) \log n)$ since n has $O(\log n)$ prime divisors (with duplication). For more, see Knuth and Trabb Pardo [KTP76] and Knuth [Knu81b].

Other general references for number theoretic algorithms include Angluin [Ang82], Bach [Bac90], Bach and Shallit [BS90], Dixon [Dix84], Lehmer [Leh69], Lenstra and Lenstra [LL90], Koblitz [Kob87], and Pomerance [Pom87].

Chapter 2

Sieve Algorithms for Perfect Power Testing

2.1 Introduction

This chapter presents fast and practical algorithms for deciding if a positive integer n is a perfect power, that is, can be expressed as x^k for integers $x, k > 1$. By trying all possible powers, this problem is solvable in $O(\log^3 n \log\log\log n)$ steps in the worst case. Unfortunately, the average running time for this method is not much better than the worst-case running time. We give algorithms in this chapter that perform much better on typical inputs. One of our methods has an average running time of $O(\log^2 n)$, and another runs in $O(\log^2 n/\log^2 \log n)$ average time, with a median running time of $O(\log n)$. Our average-case results assume that a table of certain small primes is precomputed; as a practical matter this table is quick and easy to construct, and once constructed it uses only $O(\log n)$ space, but we need to assume the extended Riemann hypothesis (ERH) to bound the largest prime in the table. We also improve the worst-case running time to $O(\log^3 n)$ steps; parallelizing this leads to an $\mathcal{NC}^2$ algorithm.

In number theory, most analyses of algorithms address worst case complexity, although there are some studies of average behavior [Col69, GK86, HM89, KTP76, Sho90]. However, we are unaware of any average-case results for this problem.

Before describing our methods, we indicate some applications for them. The fastest known methods for integer factorization [Dix81, Pom87] find non-obvious solutions to the congruence $x^2 \equiv y^2$ modulo n; if x and y are solutions with x different from $\pm y$, then $\gcd(x-y, n)$ splits n. However, if n is odd, such x and y will only exist if n is not a prime power, and this condition should be checked before attempting to factor n. It is simplest to check that n is not a perfect power, for if it is, then we have a factorization.

Similar comments apply to many other factoring algorithms [BMS86, BS89a, Len87, Pol74]; it is common when analyzing them to make the assumption that the input is not a perfect power.

We also mention an application where the average case behavior of a perfect power algorithm is significant; in fact, it is the source of the present problem. Eric Bach published an efficient algorithm for generating random integers in factored form [Bac88]; we will not discuss this algorithm in detail except to say that it repeatedly draws integers (according to some distribution) and rejects them when they are not prime powers, saving the prime powers for further processing. Since the perfect powers are rare, any perfect power test that performs well on the average will be useful in this context.

Our sieve algorithm is based on the following idea: if a number n is not a pth power mod q for some small prime q, then it cannot be a pth power. The time to check this condition is roughly proportional to the length of n, much less than the time needed to compute a pth root of n to high precision. Of course, this advantage is offset somewhat by the fact that a pth power modulo q need not be a pth power. Hence, tests using more than one q are necessary; our algorithm does enough of these tests to ensure that a pth root computation will be rare.

We also present an algorithm that combines perfect power testing and trial division; this algorithm performs even better on the average than the method outlined above. This latter algorithm is useful as a preprocessing step for factorization, since factoring programs usually start by performing trial division. Indeed, in practical factoring one often learns that a number is not a perfect power as a by-product of trial division. This perfect power algorithm is the most efficient one known to us.

Sieving ideas for testing pth powers have been suggested by Cobham [Cob66] and V. Miller [Mil89]. Cobham, assuming the ERH, showed that a sieve method for testing if a number is a perfect square is optimal to within a constant factor in its use of work space. To prove this, he showed that if a number is not a perfect square, then it must fail to be a square mod q for some small q. Miller generalized this last result to pth powers of algebraic numbers; for certain fields, he found that a sieve algorithm outperforms methods based on root approximation in the worst case.

In contrast, we do not attempt to make the sieve part of the algorithm completely foolproof. Instead, we use pth power tests modulo q to make the relatively expensive root computations rare on the average. Hence our algorithms are always correct, and our probabilistic results only apply to the running time.

Throughout this chapter we discuss algorithms to solve the decision problem for the set of perfect powers. Any of them could be easily modified to output roots and exponents when the input is a perfect power, or compute the value of Golomb's arithmetic function $\gamma(n)$ (which counts the positive integral solutions to $n = a^b$) [Gol73], but we leave such modifications to the reader.

2.2 Notation and Definitions

Let k be a positive integer. We call a positive integer n a *perfect kth power* if $n = x^k$ for some integer $x > 1$. In this case, we refer to k as the *exponent*, and x as the *root*. If n is a perfect kth power for some integer $k > 1$, then we say n is a *perfect power*.

In our analysis, we will use the naive bit complexity model. This implies that when a and b are positive integers, computing $c = a^b$ takes $O(\log^2 c)$ steps.

In the sequel, p and q will always be prime. For $x > 0$, $\pi(x)$ denotes the number of primes less than or equal to x, and $\pi_p(x)$ the number of primes less than x and congruent to 1 modulo p.

2.3 Root Computation Algorithms

In this section, we review the usual root-finding algorithm for perfect powers, which seems to be folklore. We present a slight variation of one given and analyzed by Shallit [Sha89].

The idea is very simple. If n is a perfect kth power, $k \leq \lg n$. So, we compute an integer approximation x of $n^{1/k}$ for each such k, starting with $k = 2$. If for some k, $x^k = n$, we accept n and halt. If we reach $k = \lg n$ without finding an exact root x, we reject n and halt. We can do a little better by noting that only prime values of k need be tried. Putting this together we get Algorithm A.

Algorithm A.
Input: positive integer n

```
For each prime $p \leq \lg n$ do:
    Compute $x := \lfloor n^{1/p} \rfloor$
    If $n = x^p$ then Accept and Halt
Reject and Halt.
```

To compute $x = \lfloor n^{1/p} \rfloor$, we first use the value of $\lg n$ to get a rough upper limit for x, say $2^{\lfloor \lg n/p \rfloor + 1}$. Then we do a binary search between 2 and this limit. To test each possible pth root y encountered in the search, we can compute y^p using any reasonable powering algorithm.

Theorem 2.3.1 *Given as input a positive integer n, Algorithm A will decide if n is a perfect power in time $O(\log^3 n \log\log\log n)$.*

Proof: Clearly Algorithm A is correct.

Each time a pth root is computed, binary search will iterate $O(\log n/p)$ times. During each iteration, an approximation y of the pth root of n is raised to the pth power. This takes time $O(\log^2 n)$. So then for a fixed exponent p, $\lfloor n^{1/p} \rfloor$ can be computed in time $O(\log^3 n/p)$. Summing over all prime exponents p gives

$$\sum_{p \leq \lg n} \frac{1}{p} \log^3 n = O\left(\log^3 n \log\log\log n\right).$$

All that remains is the time to find the primes up to $\lg n$. Using the Sieve of Eratosthenes, this can be done in $O(\log n \log\log\log n)$ additions. Hence the overall running time is $O(\log^3 n \log\log\log n)$, as we claimed. □

Note that there are more efficient algorithms than the Sieve of Eratosthenes for finding all the primes below a bound m. Pritchard [Pri83] discusses some of these, and presents an algorithm which uses only $O(m)$ additions and $O(\sqrt{m})$ space. See also Sorenson [Sor90b].

By replacing binary search with Newton's method, we can improve the running time of Algorithm A. However, without a good starting point, Newton's method is no better than binary search. So we first prove that if the first $\lg p$ bits of $n^{1/p}$ are provided, Newton's method converges quadratically.

Lemma 2.3.2 *Let $f(x) = x^p - n$, and let $r > 0$ satisfy $f(r) = 0$. Suppose x_0 satisfies $0 \le (x_0/r) - 1 \le 1/p$. Then Newton's method, using x_0 as an initial estimate for r, will obtain an estimate of absolute error less than 1 in $O(\log(\log n/p))$ iterations.*

Proof: Newton's method computes successive approximations $x_1, x_2, \ldots$ to r using the iteration $x_{m+1} = g(x_m)$, where $g(x) = x - f(x)/f'(x)$. Note that $g(r) = r$ and $g'(r) = 0$. Using a Taylor expansion for g around $x = r$, the mean value theorem implies that for some α, $r \le \alpha \le x_m$,

$$g(x_m) - r \quad = \quad \frac{g''(\alpha)(x_m - r)^2}{2!} \quad = \quad \frac{(p-1)n}{2\alpha^{p+1}}(x_m - r)^2 \quad \le \quad \frac{p}{2\alpha}(x_m - r)^2.$$

Since $g(x_m) = x_{m+1}$, dividing by r gives

$$\frac{x_{m+1}}{r} - 1 \quad \le \quad \frac{p}{2(\alpha/r)}\left(\frac{x_m}{r} - 1\right)^2 \quad \le \quad \frac{p}{2}\left(\frac{x_m}{r} - 1\right)^2.$$

Thus, if $(x_m/r) - 1 \le 2^{-(\lg p + i)}$ for some $i \ge 0$, then $(x_{m+1}/r) - 1 \le 2^{-(\lg p + 2i + 1)}$. In other words, at each step we double the number of bits of r we have found. To get an error less than 1 means we only have to match $O(\lg n/p)$ bits, so the number of iterations is $O(\log(\lg n/p))$.
□

Note that this result still holds if we use $\lfloor g(x) \rfloor$ for the iteration function, which is easy to compute. Now we assume that $\lfloor n^{1/p} \rfloor$ is computed by obtaining the first $\lg p$ bits of $n^{1/p}$ with binary search, and the rest with Newton's method. Calling this the *Modified Newton Method*, the following theorem holds.

Theorem 2.3.3 *Using the Modified Newton Method to find roots, Algorithm A runs in time $O(\log^3 n)$.*

Proof: For each p, $\lg p$ iterations of binary search and $O(\log(\lg n/p))$ iterations of Newton's method are required for a total of $O(\log\log n)$. Since the cost of each iteration is $O(\log^2 n)$, the total running time is

$$\sum_{p \le \lg n} O(\log^2 n \log\log n) \quad = \quad O(\log^3 n).$$

□

Corollary 2.3.4 *Computing integer approximations of kth roots (for $k > 1$ an integer) and solving the recognition problem for the set of perfect powers are both in logspace-uniform $\mathcal{NC}^2$.*

Proof: Let n be the input. From Lemma 2.3.2 we know that $O(\log\log n)$ iterations of binary search, followed by $O(\log\log n)$ iterations of the modified Newton method, suffice to compute an integer approximation to the kth root of n. Each iteration requires a power computation and possibly a long division. By methods of Beame, Cook, and Hoover [BCH86] (see also [KR90]), these two tasks can be done by a circuit of depth $O(\log\log n)$ and size polynomial in $\log n$.

These circuits are not known to be logspace uniform, but the only nonuniformity is a requirement for certain products of small primes, which can be generated easily by logspace uniform circuits of $O(\log^2\log n)$ depth. So we use an $\mathcal{NC}^2$ circuit for generating these, followed by $O(\log\log n)$ levels of binary search and $O(\log\log n)$ levels of Newton's method, to compute kth roots. The total depth is $O(\log^2\log n)$.

For recognizing perfect powers, we note that the circuit size for computing pth roots can be bounded independently of p, and simply compute pth roots of the input for every prime $p \leq \lg n$, in parallel. □

Of course, finding an approximate pth root of n is a special case of finding roots of polynomials over the integers. This more general problem has efficient sequential methods (see, for example, Pan [Pan85]), and recently Neff [Nef90] showed finding the roots of a polynomial is in $\mathcal{NC}$.

One might ask whether the modified Newton method helps in practice. The answer is yes; in section 2.7 we support this with timing results comparing binary search with the modified Newton method when used in Algorithm A.

2.4 A Simple Sieve Algorithm

In this section we present a second algorithm that improves on Algorithm A's average running time. Our main idea is the following. Most numbers are not perfect powers, so the algorithm will run better if it can quickly reject them. We will do this using the following lemma.

Lemma 2.4.1 *Let p and q be primes, with $q \equiv 1 \pmod{p}$. Further suppose that n is a perfect pth power, and* $\gcd(n,q)=1$. *Then $n^{(q-1)/p} \equiv 1 \pmod{q}$.*

Proof: If $n = x^p$, then by Fermat's theorem, $n^{(q-1)/p} = x^{q-1} \equiv 1 \pmod{q}$. □

In other words, if $n^{(q-1)/p} \not\equiv 1 \pmod{q}$, then n cannot be a perfect pth power, assuming all the other hypotheses are satisfied. We will call this computation a *sieve test for exponent* p, and the prime modulus q the *sieve modulus.* We say n *passes* the test if $n^{(q-1)/p} \equiv 0, 1 \pmod{q}$; it *fails* the test otherwise. (Note that we are evaluating the pth power residue symbol mod q; see [IR82].)

We modify Algorithm A as follows. Before computing an approximate pth root, we do a certain number of sieve tests on n for exponent p. The number of tests, $\lceil 2\log\lg n/\log p\rceil$, will be justified later; it results from balancing the goals of accuracy (few non-perfect powers should pass all the sieve tests) and efficiency (not too many sieve tests should be done). These modifications give the following procedure.

```
Algorithm B.
Input: positive integer $n$

    For each prime $p \leq \lg n$ do:
        Perform up to $\lceil 2 \log \lg n / \log p \rceil$ sieve tests on $n$
        If $n$ passed all the tests then:
            Compute $x := \lfloor n^{1/p} \rfloor$
            If $n = x^p$ then Accept and Halt
        End if
    Reject and Halt.
```

To analyze this procedure, two questions must be answered.

1. Do enough sieve moduli exist, and if they do, how large are they? Moreover, how do we find them?
2. What are the chances n will pass all the sieve tests for a fixed p?

Regarding the first question, Dirichlet showed that any "reasonable" arithmetic progression contains infinitely many primes (see [IR82]). Although this guarantees the existence of enough sieve moduli, it says nothing about their size. To reasonably bound this we will have to assume the ERH; in section 2.6 we will prove the following result.

Lemma 2.4.2 (ERH) *For an input less than n to Algorithm B, the largest sieve modulus needed is $O(\log^2 n \log^4 \log n)$.*

This suggests that the required sieve moduli are all small, and our experience with the algorithm confirms this. In fact, we believe that the above result is still an overestimate, and offer in section 2.7 a heuristic argument and numerical evidence for a sharper estimate of $\lg n \log^2(\lg n)$. In practice, the sieve of Eratosthenes will suffice to quickly generate the primes less than this bound, and hence the required list of sieve moduli.

Regarding the second question, we first argue informally. The chance an integer n is a pth power modulo q is about $1/p$. If we perform $\lceil 2 \log \lg n / \log p \rceil$ sieve tests whose results are independent, the chance n passes them all is about $(1/p)^{2 \log \lg n / \log p} = (1/\lg n)^2$. However, the tests are not quite independent and we must modify this rough argument. In section 2.6 we will prove the following result, which uses the ERH only to bound the magnitude of the sieve moduli.

Lemma 2.4.3 (ERH) *Let n be an integer chosen uniformly from an interval of length L, and assume for every such n, $L \geq (\log n)^{3 \log \log \log n}$. Then the probability n passes $\lceil 2 \log \lg n / \log p \rceil$ different sieve tests for a fixed exponent p in Algorithm B is bounded above by*

$$O\left(\frac{\log \log n}{\log^2 n}\right).$$

We expect that any implementation of our algorithm will have available a list of sieve moduli. For this reason, we will analyze Algorithm B under the assumption that they have been precomputed. In particular, we assume that Algorithm B has available a table, called the *sieve table*, which contains, for each prime $p \leq \lg n$, a list of the first $\lceil 2 \log\lg n / \log p \rceil$ sieve moduli for p. Note that the number of entries in this table is at most

$$\sum_{p \leq \lg n} \left\lceil \frac{2 \log \lg n}{\log p} \right\rceil \quad = \quad O\left(\frac{\log n}{\log\log n}\right)$$

and so the total space used by the table, once computed, is $O(\log n)$. Using the Sieve of Eratosthenes, or a variant, the table can be constructed in $(\log n)^{2+o(1)}$ time. In practice this is pessimistic, as we demonstrate in section 2.7.

We also note that computing the exact values of $\lceil 2 \log\lg n / \log p \rceil$ for each prime $p \leq \lg n$ can be done using the methods of Brent [Bre76b] easily within the above time bound. In practice, of course, this is not a concern.

Assuming the above lemmas, we now present our average-case result.

Theorem 2.4.4 (ERH) *Let n and L be as in Lemma 2.4.3, and assume that a sieve table is available. Then Algorithm B will decide if n is a perfect power, and the running time of Algorithm B, averaged over all inputs in L, is $O(\log^2 n)$.*

Proof: The correctness of Algorithm B follows immediately from Lemma 2.4.1; it is independent of the ERH.

To get an upper bound on the running time, we may assume that all the possible sieve tests are actually done. By Lemma 2.4.2, $\log q = O(\log\log n)$. Since the sieve table is precomputed, the time to find each sieve modulus q is $O(1)$. Computing $n^{(q-1)/p} \bmod q$ can be done using one division and then modular exponentiation in time $O(\log n \log q + \log^3 q) = O(\log n \log\log n + \log^3 \log n) = O(\log n \log\log n)$. If $\lceil 2 \log\lg n / \log p \rceil$ sieve tests are performed, the total time spent is at most $O(\log n \log^2 \log n / \log p)$ for each prime exponent p.

From the proof of Theorem 2.3.3, the time needed to approximate the pth root of n and compute its pth power is $O(\log^2 n \log\log n)$. However, by Lemma 2.4.3, the probability that we even have to make the computation is $O(\log\log n / \log^2 n)$. Thus, the average time spent is $O(\log^2 \log n)$.

Hence, for each prime exponent p, the average time spent is
$O(\log n \log^2 \log n / \log p)$. Since we have

$$\sum_{p \leq B} \frac{1}{\log p} \quad = \quad O\left(\frac{B}{\log^2 B}\right)$$

by splitting the sum at $\sqrt{B}$ and using the prime number theorem, this gives the average time of $O(\log^2 n)$, and the proof is complete. □

In connection with Algorithm B, the following question is also of interest: "Can we guarantee that n is a perfect power by only performing sieve tests?" By applying quadratic reciprocity and Ankeny's theorem, Cobham [Cob66] showed that $O(\log^2 n)$ sieve tests suffice to check that n is a square, if the ERH is true. V. Miller [Mil89] has recently extended this result: if one assumes the Riemann hypothesis for certain Hecke L-functions then $O(p^2 \log^2(np))$ sieve tests for the exponent p will prove that n is a pth power. Since we are only interested in $p \leq \lg n$, this would imply that $O(\log^{5+\epsilon} n)$ tests suffice to check perfect powers. Note that this leads to another $\mathcal{NC}$ algorithm for perfect power testing.

It is also of interest to ask what can be proved without assuming the ERH. The main difficulty with this seems to be in finding a large number of sieve moduli efficiently. When the sieve moduli are not bounded by a small polynomial in $\log n$, then the Sieve of Eratosthenes and its variants are no longer practical for finding them. As an alternative, one might use probabilistic search, but then the sieve moduli found must be *proved* prime. It is an interesting theoretical question as to how this might be done, but such an approach is unnecessary in practice, and we discuss it no further here.

2.5 A Sieve Algorithm with Trial Division

In the previous section we discussed Algorithm B, a substantial improvement over Algorithm A, which used a simple sieving idea to weed out non-perfect powers. In this section, we will use trial division by small primes to further improve Algorithm B. The resulting method may be of use in situations, such as factorization, where trial division is done anyway.

Our basic idea is the following. To test n, we see if n has any small prime divisors by checking all primes $\leq b$. The trial division bound b is much smaller than n; we take b to be about $\log n$. There are two ways this modification can help:

1. If we find a prime r that divides n, we can compute the integer e such that $r^e \parallel n$. Then if n is a perfect pth power, p must divide e. Since e will typically be quite small, this will greatly reduce the number of possible prime exponents p that must be checked.

2. If we do not find any divisors of n below b, then if n is a perfect pth power, its pth root x must be larger than b. Hence $p \leq \log_b n = \log n / \log b$, which will also save time.

Our new algorithm does trial division up to some bound b, and then applies either 1. or 2. above, depending on whether or not a divisor is found. It then uses an appropriately modified version of Algorithm B. Algorithm C gives the details.

```
Algorithm C.
Input: positive integer $n$

    Compute the smallest integer $b$ such that $b\log^2 b \geq \log n$
    $S := \{p : \ p \leq \log n/\log b\}$
    Trial divide $n$ by each prime $r \leq b$:
        If a divisor $r$ is found then:
            Find $e$ such that $r^e \mid\mid n$
            $S := \{p : p \mid e\}$
            Stop trial division
        End if
    While $S \neq \emptyset$ do:
        $p :=$ the smallest element of $S$
        Perform up to $\lceil 2\log\lg n/\log p \rceil$ sieve tests on $n$ for $p$:
            If for some sieve modulus $q$, $q \mid n$, then:
                Find $e$ such that $q^e \mid\mid n$
                $S := S \cap \{p : p \mid e\}$
            End if
        If $n$ passed all the tests and $p \in S$, then:
            Compute $x := \lfloor n^{1/p} \rfloor$
            If $n = x^p$ then Accept and Halt
        End if
        $S := S - \{p\}$
    Reject and Halt.
```

To analyze this algorithm we will need a technical lemma.

Lemma 2.5.1 *Let P be a set of primes, and for a positive integer n, let $e(n)$ denote the largest e such that $p^e \mid n$ for some $p \in P$. If n is chosen uniformly from an interval of length L, then the expected value of $e(n)$ is at most $(\#P)\lg n/L + O(1)$.*

Proof: At most $L/p^e + 1$ integers in the interval are divisible by p^e. Since $e(n) \geq k$ if and only if $p^k \mid n$ for some $p \in P$, $\Pr[e(n) \geq k] \leq \sum_{p\in P} 1/p^k + (\#P)/L$. Using this,

$$\mathrm{E}[e(n)] \quad = \quad \sum_{k=1}^{\lfloor \lg n \rfloor} \Pr[e(n) \geq k] \quad \leq \quad \frac{(\#P)\lg n}{L} + 1 + \sum_{p\in P}\sum_{k=2}^{\infty} \frac{1}{p^k} \quad < \quad \frac{(\#P)\lg n}{L} + 2.$$

□

Theorem 2.5.2 (ERH) *Let n be an integer chosen uniformly from an interval satisfying the hypotheses of Lemma 2.4.3. Then Algorithm C will decide if n is a perfect power in time*

$$O\left(\frac{\log^2 n}{\log^2 \log n}\right)$$

on the average.

Proof: Correctness follows from Theorem 2.4.4 and from the two observations made at the beginning of this section. All that remains is to prove the average running time bound.

First we note that b is $\Theta(\log n / \log^2 \log n)$, from which it follows that $\log b = \Theta(\log\log n)$.

By a large sieve estimate of Jurkat and Richert [JR65, 4.2] the probability that no prime below b divides n is

$$\prod_{p \le b} \left(\frac{p-1}{p}\right)\left(1 + O\left(\frac{1}{\log L}\right)\right),$$

provided that L, the interval length, satisfies $\log b \le (\log L)/(2 \log\log(3L))$. By our choice of L, this holds for sufficiently large n, so by Mertens's theorem (see [HW79], p. 351), the probability of escaping trial division is $O(1/\log b)$, a fact we will need later.

We break the running time into four parts: the time spent on trial division, the time spent computing maximal exponents e for various primes, the time spent on approximating pth roots and computing pth powers of these approximations, and the time spent performing sieve tests.

The time spent on trial division is $O(\sum_{p \le b} \log n \log p)$. Combining this with our estimate for b shows that the expected time for trial division is $O(\log^2 n / \log^2 \log n)$.

By Lemma 2.4.2 no sieve modulus or trial divisor is larger than $O(\log^{2+\epsilon} n)$, so the time spent finding a maximal exponent e for any such base is $O(e \log n \log\log n)$. By Lemma 2.5.1 the expected value of e is $O(1)$, so the expected work for finding maximal exponents is negligible.

To estimate the third part, we note that the time spent computing pth roots and their pth powers is no more than the time Algorithm B would spend in the same task, on a given input. Hence the argument used to prove Theorem 2.4.4 applies, and we find that the total expected time for this part is $\sum_{p \le \log n} O(\log^2 \log n)$, which is $O(\log n \log\log n)$.

To estimate the expected time for sieve tests, we condition on whether or not a divisor is found. If a divisor r of n is found, its maximal exponent e is at most $\lg n$. Thus e has at most $\lg\lg n$ prime divisors, which is how many exponents remain to be tested. From the proof of Theorem 2.4.4, we know the maximum time spent on each possible prime exponent p is $O(\log n \log^2 \log n / \log p)$. Thus the average sieve time, given that a divisor is found, is $O(\log n \log^3 \log n)$. If no divisor is found, the average sieve time is similarly found to be $O(\log^2 n / \log b)$. Multiplying this by the probability that no divisor is found and using the asymptotic value of $\log b$ gives the result. □

The distribution of Algorithm C's running time exhibits the following anomaly. Although its expected running time is high (consider the fraction $1/\log b$ of inputs for which all trial divisions are performed), its median running time is much lower, in fact $O(\log n)$. We prove this below.

Theorem 2.5.3 *Let n be chosen uniformly from an interval of length L, and assume for every such n, $L \ge f(n)$, where $\lim_{x\to\infty} f(x) = \infty$. Then the median running time of Algorithm C is $O(\log n)$.*

Proof: We must show that there is a set of inputs, of probability at least $1/2$, on which the running time is $O(\log n)$. First consider the asymptotic probability that some prime less than B divides n, and the least such prime divides n exactly once. This is

$$\sum_{p<B} \Pr[\text{ no } q < p \text{ divides } n \text{ and } p \,||\, n\ ] \quad = \quad \sum_{p<B} \left(\frac{1}{p} \prod_{q \leq p} \left(1 - \frac{1}{q}\right)\right).$$

If $B = 20$, this is some constant α, greater than $1/2$. Hence with probability α, $e = 1$ at the end of the trial division phase. Given that this happens, no further work will be required. Furthermore, the work of trial division is $O(\log n)$, since no r greater than 20 and no exponent larger than 2 was ever used on these inputs. □

2.6 Technical Results

In this section we will prove Lemmas 2.4.2 and 2.4.3 with the aid of the ERH.

Let $\pi_p(x)$ denote the number of primes less than or equal to x which are congruent to 1 modulo p; these are the possible sieve moduli for p. Below we give an estimate for the density of such primes. This result is due to Titchmarsh [Tit30, Theorem 6]. See also Davenport [Dav80, p. 125].

Theorem 2.6.1 (ERH) *Let x be a positive integer, and p a prime. There is a constant $A > 0$, independent of p and x, such that*

$$\pi_p(x) \quad \geq \quad \frac{1}{p-1} Li(x) - A\sqrt{x}\log x.$$

Corollary 2.6.2 (ERH) *Let p be prime. There is a constant $B > 0$, independent of p, such that for any $v \geq 2$, if $x \geq Bp^2v(\log pv)^4$ then $\pi_p(x) \geq pv(\log pv)^3$.*

Proof: Let A be the constant from Theorem 2.6.1, and choose $B \geq 3$ so that $\sqrt{B}/(\log B)^2 \geq 12 + 36A$. Since $Li(x) \geq (x-2)/\log x$, Theorem 2.6.1 and some algebra gives us

$$\begin{aligned} \pi_p(x) &\geq \pi_p\left(Bp^2v(\log pv)^4\right) \\ &\geq pv(\log pv)^3 \left(\frac{B}{6\log B} - 1 - 6A\sqrt{B}\log B\right). \end{aligned}$$

Our choice for B guarantees that the coefficient of $pv(\log pv)^3$ is at least 1. □

We now prove the sieve modulus bound.

Lemma 2.4.2 (ERH) *For an input less than n to Algorithm B, the largest sieve modulus needed is $O(\log^2 n \log^4 \log n)$.*

Proof: We must show that, for each prime $p \leq \lg n$, the $\lceil 2\log\lg n/\log p\rceil$th sieve modulus is $O(\log^2 n \log^4 \log n)$. In other words, we must find an x for each prime p such that $x = O(\log^2 n \log^4 \log n)$ and $\pi_p(x) \geq \lceil 2\log\lg n/\log p\rceil$.

If $p \geq \sqrt{\log n}$, we choose $x = Bp^2 v \log^4(pv)$ with $v = 2$, where B is the constant from Corollary 2.6.2. Clearly $x = O(\log^2 n \log^4 \log n)$, and Corollary 2.6.2 gives us $\pi_p(x) \geq 2\sqrt{\log n} \geq \lceil 2 \log \lg n / \log p \rceil$.

If $p \leq \sqrt{\log n}$, by choosing $x = Bp^2 v \log^4(pv)$ with $v = 2 \log \log n$, Corollary 2.6.2 gives us $\pi_p(x) \geq v \geq \lceil 2 \log \lg n / \log p \rceil$ with $x = O(\log n (\log \log n)^5)$. □

Now we will prove Lemma 2.4.3, which states that an input n to Algorithm B is unlikely to pass multiple sieve tests.

Lemma 2.4.3 (ERH) *Let n be an integer chosen uniformly from an interval of length L, and assume for every such n, $L \geq (\log n)^{3 \log \log \log n}$. Then the probability n passes $\lceil 2 \log \lg n / \log p \rceil$ different sieve tests for a fixed exponent p in Algorithm B is bounded above by*

$$O\left(\frac{\log \log n}{\log^2 n}\right).$$

Proof: Let T be the set of sieve moduli from the sieve table for p, and let $t = \#T$, the cardinality of T. Define $m = \prod_{q \in T} q$, and note that $t = \lceil 2 \log \lg n / \log p \rceil$. Write $L = dm + r$ where d and r are integers satisfying $d > 0$, $0 \leq r < m$. The chance that n passes all $\lceil 2 \log \lg n / \log p \rceil$ sieve tests is, by the Chinese Remainder Theorem, at most

$$\frac{dm}{L} \prod_{i=1}^{t} \left[\frac{q_i - 1}{q_i} \left(\frac{1}{p} \right) + \frac{1}{q_i} \right] + \frac{r}{L} \quad = \quad \frac{dm}{L} \prod_{i=1}^{t} \frac{1}{p} \left(1 + \frac{p-1}{q_i} \right) + \frac{r}{L},$$

where $q_1, \ldots, q_t$ are the primes in T in increasing order. Since $q_i > ip$, $1 + (p-1)/q_i < (i+1)/i$, so the first term is at most

$$\frac{1}{\lg^2 n} \prod_{i=1}^{t} \frac{i+1}{i} \quad = \quad O\left(\frac{\log \log n}{\log^2 n}\right).$$

To estimate the second term, let x be the largest sieve modulus in T. By Corollary 2.6.2, $x = O(p^2 t (\log pt)^4)$. Since $r < m$, it suffices to show $m/L = o(1/\log^2 n)$, which is true since

$$m \quad = \quad \prod_{q \in T} q \quad \leq \quad x^t \quad = \quad (\log n)^{2 \log \log \log n (1 + o(1))}.$$

□

It would be interesting to show that this result holds for intervals of "polynomial size," that is, $L \geq \log^c n$ for some $c > 0$. We require slightly larger intervals, in which only the last $\Omega(\log \log n \log \log \log n)$ bits of the input vary.

Finally, we remark that these results hold for the interval $[1, n]$.

2.7 Implementation Results

In this section we give empirical results on our algorithms. As the performance of Algorithms B and C is sensitive to the size of the entries in the sieve table, we also give a heuristic argument, backed up by experimental data, that this table is efficiently computable.

Lemma 2.4.2 indicates that the sieve table will have entries bounded by $O(\log^{2+\epsilon} n)$. In practice, we have found this bound to be pessimistic, and believe that $\lg n \log^2 \lg n$ is a more accurate estimate. Below we give a heuristic argument, patterned after one by Wagstaff [Wag79], that results in this bound.

Let p be the largest prime less than or equal to $B = \lg n$. We consider a sieve modulus bound $m > B$ and estimate the probability that this suffices to obtain enough sieve moduli for all primes up to B. Call a prime "small" if it is less than $(\lg n)^{1/3}$, and "large" otherwise. If n is sufficiently large, then Lemma 2.4.2 guarantees enough sieve moduli for small primes. For large primes, we need at most t sieve moduli, where $t = 6$ (the particular constant does not matter). We now make the heuristic assumption that an integer x is prime with probability $1/\log x$, and estimate the probability that more than t sieve moduli exist for all large primes. This probability is at least

$$\Pr[\ t \text{ sieve moduli exist for } p\ ]^B,$$

assuming that the chances of success for each p are independent and monotonically decreasing. Now consider a sequence of m/p integers $p+1, 2p+1, \ldots, m$. The probability that there are at most t primes in this sequence is at most

$$\binom{m/p}{t}\left(1 - \frac{1}{\log m}\right)^{m/p-t} \quad \leq \quad (m/p)^t e^{(t-m/p)/\log m}.$$

Write the right-hand expression as e^y where $y = t\log(m/p) + (t - m/p)/\log m$; then

$$\Pr[\text{at least } t \text{ sieve moduli exist for all } p \leq B] \quad \geq \quad (1 - e^y)^B.$$

We wish this estimate to be e^{-c} for some small positive c. Setting these equal we find that $e^y = 1 - e^{-c/B} \sim c/B$; taking logarithms of both sides and simplifying we find that

$$\frac{m}{\log m} \quad \sim \quad B\log B - B\log c + \frac{Bt}{\log m} + Bt\log\left(\frac{m}{B}\right)$$

(note that $p \sim B$). If c is a constant, this implies that

$$m \quad \sim \quad B\log^2 B \quad = \quad \lg n \log^2(\lg n).$$

(We can also take $c = 1/(\log B)^k \to 0$ and get the same result.)

Although its derivation is suspect, this bound seems reasonably precise, as the table below shows.

Building the Sieve Table

Decimal Digits	Maximum Modulus	Heuristic Bound	Table Memory	CPU Time in sec.
10	373	407	48	0.003
25	1609	1622	94	0.004
50	3769	4341	147	0.012
100	9767	11197	245	0.031
250	24229	37525	498	0.113
500	78577	91327	865	0.293
1000	152017	218398	1510	0.707
2500	527591	676371	3289	2.262
5000	1449271	1568522	5981	5.402
10000	2839301	3600522	10977	12.710
25000	9731863	10655493	24781	38.790
50000	21021569	23998967	46169	89.064

In this table, the first column lists various values of d for $n = 10^d$. *Maximum Modulus* is the largest sieve modulus, where the first $\lceil 2 \log \lg n / \log p \rceil$ were taken for each $p \leq \lg n$. *Heuristic Bound* is the value of $\lg n \log^2 \lg n$. *Table Memory* gives the number of integers needed to store all the sieve moduli (the space in words was about double this). Finally, *CPU Time* indicates the number of CPU seconds used to construct the sieve moduli from scratch and store them in a table.

In conclusion, we needed the ERH to give provable bounds on the size of the sieve moduli, and the resulting bounds forced us to use precomputation to construct the table. In practice this is not a problem, as the results above demonstrate.

Next we give the results of our implementations of Algorithms A, B, and C. We coded all three algorithms using the same multiple precision arithmetic routines on a DECstation 3100. The table below gives these results.

Running Time Results

Decimal Digits	A-BS CPU sec	A-MN CPU sec	B CPU sec	C CPU sec
10	0.0289	0.0282	0.0016	0.0007 (8)
25	0.1106	0.0866	0.0056	0.0012 (8)
50	0.3863	0.2454	0.0147	0.0025 (7)
100	1.8744	0.9210	0.0394	0.0033 (10)
250	21.9904	7.7954	0.1487	0.0119 (8)
500	163.0009	47.8809	0.3997	0.0208 (9)
1000	1283.4628	323.7992	1.4937	0.0638 (8)

Decimal Digits is the size of the inputs; we ran each algorithm on the same 10 pseudo-random integers and tabulated the average running time for each algorithm. There are two columns for algorithm A: the first gives timings using binary search, and the second gives timings using the modified Newton method. For Algorithm C we also give the number of times (out of 10) a divisor was found during trial division.

For Algorithm B we did not precompute the sieve table. Instead, a modified version of the Sieve of Eratosthenes found all the primes below the heuristic bound, which were then stored in a bit vector. When sieve moduli were needed, we searched for them sequentially in this bit vector.

For Algorithm C, we first found the primes below b, the trial division bound. Only when trial division failed did we find all the primes below the heuristic bound.

The system's timing clock has unreliable low order digits, so we ran each algorithm several times on the same input and timed the whole group. The number of times per input was inversely proportional to an algorithm's average running time.

Chapter 3

Polylog Depth Circuits for Integer Factoring and Discrete Logarithms

3.1 Introduction

In this chapter, we develop polylog depth, subexponential size probabilistic boolean circuits to factor integers and compute discrete logarithms over prime finite fields. These are the first results of this type, and they answer a previously open question posed by Adleman and Kompella [AK88], who asked whether such circuits exist for these two problems.

Recall that the *complete integer factorization problem* is to write the input N as a product of powers of primes. The *discrete logarithm problem* is, given a finite group G written multiplicatively, a generator g, and an element $a \in G$, to find a solution x to $g^x = a$. In this chapter we consider only multiplicative groups of finite fields.

Our results on these two problems are motivated by work done on the rigorous analysis of subexponential time sequential algorithms to factor integers and compute discrete logarithms [Adl79, Dix81, HR82, Odl85, Pom82, Pom87, Val91, LP91] and also by the recent work done on parallel number theoretic algorithms [AK88, BS89b, BCH86, BvzGH82, BK83, CG90, FT88, KMR87, PR85, Mul87, vzG87]. Our proofs rely heavily on a replication technique. This allows us to reduce the error probability of probabilistic boolean circuits at the cost of increased circuit size, much as iteration reduces the error probability of randomized sequential algorithms at the cost of a longer running time.

We base our integer factoring circuit on Dixon's algorithm [Dix81], which uses the difference of squares method. Our discrete logarithm circuit is based on the index-calculus method as described by Adleman [Adl79]. See also Pomerance [Pom87]. Faster methods exist for both of these problems, but we discarded them for one of two reasons, which are discussed below.

The first reason is that some of these methods, such as the Quadratic Sieve of Pomerance [Pom85], Coppersmith's algorithm [Cop84], and Number Field Sieve techniques [LLMP90, Adl91, Gor91] rely on unproven conjectures in their running time analyses. Such gaps in the theory behind these algorithms are acceptable if one is primarily interested in practical algorithms. However, our results are largely of theoretical interest, so using fully rigorous algorithms as the basis of our circuits is essential. By choosing Dixon's and Adleman's algorithms, which are both fully rigorous, we satisfy this criterion.

The second reason is that those faster methods which are fully rigorous, such as Vallée's algorithm [Val91] or the class group based algorithm of Lenstra and Pomerance [LP91], do not improve enough on Dixon's or Adleman's algorithms to be useful to us. Choosing such a method would only change the implied constants in the exponents of the sizes of our circuits.

We say a positive integer is smooth if its prime divisors are small. Adleman's and Dixon's algorithms use smooth numbers in a similar fashion. They start by choosing a bound B and finding all primes below that bound. Then they generate random elements with special properties until they get at least B of them which factor completely over the primes below B. For integer factoring, these special elements are random squares modulo N, where N is the number to be factored, and for discrete logarithms, these elements are random powers of the generator. Using these elements, they construct a system of linear equations whose solution leads to either a proper divisor of the input or the discrete logarithm of the input.

In optimizing the running times of these algorithms, one analyzes the tradeoff involved in choosing a value for B. If B is small, both finding all the primes and solving the system of equations are fast, but many random elements must be generated, since the probability an element factors over the primes below B is low. If B is large, then generating elements which factor completely over the primes below B is fast, however more of these elements must be generated and solving the system of equations takes a long time.

Let n be the length of the input, and let $L(n) = \exp[\sqrt{n \log n}]$. It is widely known that for most sequential algorithms, B should be $L(n)^c$ for c a small positive constant. The resulting running times are proportional to a small power of $L(n)$. Pomerance [Pom87] showed that an improved version of Dixon's algorithm runs in time $L(n)^{\sqrt{2}+o(1)}$. He also achieved the same running time for an improvement to Adleman's algorithm for discrete logarithms. Vallée's algorithm [Val91] runs in time $L(n)^{\sqrt{4/3}+o(1)}$. Lenstra and Pomerance have the fastest known rigorous factoring algorithm [LP91] with a running time of $L(n)^{1+o(1)}$.

Our idea is to set $B = n^{\delta(n)}$, where $\delta(n) = \log^d n$ for $d > 0$ a constant. We then prove the existence of:

- Probabilistic boolean circuits of depth $O(\log^{2d+2} n)$ and size $\exp[O(n/\log^d n)]$ for completely factoring a positive integer with probability $1 - o(1)$, and

- Probabilistic boolean circuits of depth $O(\log^{2d+2} n + \log^3 n)$ and size $\exp[O(n/\log^d n)]$ for computing discrete logarithms in the finite field $GF(p)$ for p a prime, with probability $1 - o(1)$.

Considering the size of these circuits, it is clear that unbounded fan-in is necessary to achieve a polylog depth. However, we also give sublinear depth circuits with bounded fan-in of subexponential size for these problems; we use $\delta(n) = n^{1/3+o(1)}$ to get bounded fan-in depth $O(n^{2/3+o(1)})$ and size $\exp[n^{2/3+o(1)}]$.

For an introduction to factoring and discrete logarithm algorithms, see Koblitz [Kob87]. For surveys of some of the currently best algorithms, along with detailed analyses and further references, see Pomerance [Pom82, Pom87] and Odlyzko [Odl85].

The rest of this chapter is organized as follows. In section 3.2, we discuss some notation and definitions. We introduce the replication technique and present our circuits for completely factoring integers in section 3.3. In section 3.4 we give our circuits for computing discrete logarithms. We conclude with some remarks in section 3.5.

3.2 Preliminaries

In this section we discuss some notation and definitions.

All our algorithms are given as probabilistic boolean circuits. We define probabilistic boolean circuits similarly to standard boolean circuits, but we also allow gates which can toss a fair coin and use the result for their output. In addition, we also have two special bits of output; one called the *correctness* bit, and one called the *output* bit. The output bit is 1 if the rest of the circuit's output "means" something, or in other words, if the circuit computes something for the function value. The correctness bit signals whether this output is certain to be correct. The output may be correct without the correctness bit being set. If the output bit is 0, then the value of the correctness bit is irrelevant.

We will say that a probabilistic circuit has *zero-sided error* if, whenever the output bit is on, then the correctness bit is also on. An example is a circuit that attempts to find a non-trivial divisor of the input. If one is found, then it is always "correct." If one is not found, then the output bit is off.

We say a circuit has *one-sided error* if the correctness bit is on at least some of the time when the output bit is on. An example is a circuit to test primality by attempting to factor the input. If a divisor is found, then the circuit is certain the input is not prime, but if no divisor is found, then the input is only probably prime.

Finally, a circuit has *two-sided error* if the correctness bit is never on. An example is a circuit that attempts to completely factor its input; it may not be sure that the divisors it outputs are prime.

For more information on parallel computation and circuits, see Cook [Coo85] and Karp and Ramachandran [KR90].

$GF(q)$ is the finite field of q elements, where $q = p^e$ for a prime p and $e \geq 1$ an integer. The prime p is called the *characteristic* of the field. A finite field is *prime* or has *prime order* if $e = 1$. In this case, $GF(q) = GF(p) \cong \mathbf{Z}/(p)$. $GF(q)^*$ is the multiplicative group of the field. An element $g \in GF(q)^*$ is a *generator* if every nonzero field element can be written as a power of g. Let $a \in GF(q)^*$ and assume that g is a generator for $GF(q)^*$. Then we can write $g^x = a$ for x an integer; x is called the *index* or *discrete logarithm* of a, x is unique modulo the order of the group $GF(q)^*$ which is $q-1$, and we write $\operatorname{ind}_g(a) = x$.

3.3 Integer Factoring

In this section we present our polylog depth probabilistic boolean circuit for completely factoring integers. We begin by reviewing Dixon's random squares method for splitting (finding a proper divisor) of a composite integer. We then discuss the replication technique, and we present a circuit for splitting integers. Finally, we use the circuit for splitting integers to construct a circuit that completely factors integers.

3.3.1 Dixon's Random Squares Method

Dixon's algorithm is based on the following idea. Given a composite integer N which is not a prime power, if we can find integers a and b, $a \not\equiv \pm b \pmod N$, such that $a^2 \equiv b^2 \pmod N$, then $\gcd(a+b, N)$ is a proper divisor of N. To find a and b, we first find some smooth squares modulo N. Then we find a linear dependency among them in factored form, which leads to a perfect square. Dixon [Dix81] proved that this method works with probability at least 1/2. To make this more precise, here is an outline of Dixon's factoring algorithm.

Dixon's Algorithm

Input: positive integer N, not a prime power.

1. Choose a positive bound B, find all the primes below B, and call this the prime base. Let $p_1, \ldots, p_k$ be the primes of the prime base, with $k = \pi(B)$.

2. Repeat until at least $k+1$ B-smooth squares are found:
   a) Choose a random integer r uniformly from $[2, N-1]$.
   b) Let $s = r^2 \bmod N$. Factor s over the prime base to see if it is smooth.

3. For each smooth square s_j, $1 \le j \le k+1$, compute the indices e_{ij} such that $s_j = \prod_{i=1}^{k} p_i^{e_{ij}}$. Let $\mathbf{v}_j = [e_{1j}, \ldots, e_{kj}]^T \bmod 2$, a column vector over $GF(2)$.

4. Form $\mathbf{A} = [\mathbf{v}_1, \ldots, \mathbf{v}_{k+1}]$ and solve the system $\mathbf{Ax} = 0$ for a nonzero vector $\mathbf{x}$ over $GF(2)$. Since $\mathbf{A}$ has more columns than rows, such a solution must exist. This can be found using Gaussian elimination.

5. Writing $\mathbf{x} = [x_1, \ldots, x_{k+1}]^T$, form $a = \prod_{j=1}^{k+1} r_j^{x_j}$ and $b = \sqrt{\prod_{j=1}^{k+1} s_j^{x_j}}$ modulo N. Then $a^2 \equiv b^2 \pmod N$ because $s_j \equiv r_j^2$, so $\gcd(a+b, N)$ is a proper divisor of N with probability at least 1/2.

For a proof of correctness, see Dixon [Dix81] or Pomerance [Pom87]. One important question to ask at this point is: What is the probability that an integer is smooth? The following answers this question.

Lemma 3.3.1 (Canfield, Erdős, Pomerance [CEP83]) *Let $u = \log x / \log y$. Then there exists a constant $c > 0$ such that if $u \ge c$, $x \ge 1$, then $\Psi(x,y) > xu^{-3u}$.*

Essentially, this lemma says that if w is chosen uniformly between 1 and x, then w is y-smooth with probability at least u^{-3u}, where $u = \log x / \log y$. Better estimates are available for this probability, but this will suffice for our purposes.

Let $n = \lfloor \lg N \rfloor + 1$, the length of the input N in binary. We can now give an estimated running time for Dixon's algorithm. Roughly speaking, if we set $u = n / \lg B$, then the expected running time of the algorithm is about $Bu^{3u} + B^3$, ignoring polynomial time factors. The first term reflects the time needed to find about B distinct B-smooth squares, since the probability a number is B-smooth is at least u^{-3u} by Lemma 3.3.1. The second term reflects the time for Gaussian elimination to find our solution $\mathbf{x}$. Optimizing gives $B = \exp[O(\sqrt{n \log n})]$. For a more exact run time analysis see Pomerance [Pom82, Pom87].

We, however, are interested in an efficient parallel algorithm, or in other words, a shallow circuit. Since the random squares can be generated simultaneously, the dominant cost is the depth needed to solve the system of linear equations. Using known methods [BvzGH82, Mul87, PR85] a system of m linear equations in m unknowns over $GF(2)$ can be solved in depth $O(\log^2 m)$ using $m^{O(1)}$ gates. For our application, $m \leq B$, so this can be done in depth $O(\log^2 B)$ using at most $B^{O(1)}$ gates. If we set $B = \exp[O(\sqrt{n \log n})]$ as above, we get depth $O(n \log n)$, which is clearly not good enough. If we set $B = n^{O(1)}$, a polynomial in n, the depth is fine, but then we must generate $u^{3u} n^{O(1)}$ squares, which is no longer subexponential in n. We compromise by setting $B = n^{\delta(n)}$ where $\delta(n)$ is a non-decreasing function of n. In our analysis later, we will leave $\delta(n)$ as a parameter, which we will optimize for both unbounded fan-in circuits and bounded fan-in circuits. As we mentioned earlier, to get the polylog depth circuit for factoring, we use $\delta(n) = \log^d n$ for d a positive constant.

3.3.2 The Replication Technique

Before we prove our main result, we must first develop a simple *replication technique.* The idea is to increase the probability with which a probabilistic circuit computes its output correctly by duplicating that circuit several times in parallel.

Let C_n be the circuit for inputs of length n in the probabilistic circuit family C. C_n computes a function probabilistically, so let $p(n)$ be the probability that C_n's output is a correct function value.

Let $q(n)$ be our target error probability. In other words, we want to duplicate C_n enough times so that with probability at least $1 - q(n)$, at least one of the C_n subcircuits computes its output correctly.

We construct a new probabilistic circuit family R, where each circuit R_n is as follows. First, R_n runs $m(n)$ copies of C_n in parallel, where $m(n) \geq (1/p(n)) \log(1/q(n))$. Then we use a simple fan-in structure to select a C_n subcircuit whose output and correctness bits are both set, if there is one. Numbering these subcircuits from 1 to $m(n)$, we choose the lowest numbered such subcircuit for the output of R_n. If no subcircuit generates a correct function value as output with certainty, then R_n either uses the output of the first circuit which generates an output, if there is one, or chooses some "best" circuit to copy. How this choice is made we leave to the specific application of the technique. If no C_n subcircuit generated an output, then R_n's output bit is turned off.

The probability that all the C_n subcircuits fail is at most

$$(1 - p(n))^{m(n)} \quad \leq \quad \left(1 - \frac{1}{p(n)^{-1}}\right)^{-p(n)^{-1} \log q(n)} \quad \leq \quad e^{\log q(n)} = q(n),$$

as desired.

The size of the circuit R_n is $O(m(n) \cdot \text{size}(C_n))$, and the depth of R_n is $O(\log m(n) + \text{depth}(C_n))$ for the bounded fan-in case or the $\text{depth}(C_n) + O(1)$ for the unbounded fan-in case.

Notice if C_n has zero- or one-sided error, then so does R_n. This technique can also be applied to circuits with two-sided error, but not in all cases; R_n must be modified to somehow choose which C_n subcircuit has the "best" output, if that is possible. If it is, then the resulting circuit R_n has two-sided error.

For example, if a circuit C_n produces correct output with probability $1/n$, we can copy C_n $2n \log n$ times using this technique to get a new circuit which produces correct output with probability $1 - 1/n^2$.

We will apply this replication technique repeatedly throughout the rest of this chapter.

3.3.3 A Circuit for Splitting N

Theorem 3.3.2 *Let $\delta(n)$ be a non-decreasing function of n, $\delta(n) \leq O(\sqrt{n/\log n})$. There exist both*

1. *an $n/\delta(n)$-space uniform probabilistic circuit family with bounded fan-in of depth $O(\sqrt{n}\log^{2.5} n + \delta(n)^2 \log^2 n + n/\delta(n))$ and size $\exp[O(n/\delta(n))]$, and*
2. *an $n/\delta(n)$-space uniform probabilistic circuit family with unbounded fan-in of depth $O(\delta(n)^2 \log^2 n)$ and size $\exp[O(n/\delta(n))]$*

which on input N, a positive composite integer of n bits, produce as output a proper divisor of N with probability at least $1/2 - o(1)$.

Proof: We will describe how to parallelize each step of Dixon's algorithm.

We can safely assume that N is not a prime power, since we can compute integer roots in $\mathcal{NC}^2$ (see chapter 2).

Let $B = n^{\delta(n)}$.

Step 1.

To find all the primes in the prime base, we perform trial division in parallel. This can be done in depth $O(\log B) = O(\delta(n) \log n)$ and size $B^{O(1)}$.

Steps 2 and 3.

Next, we construct a subcircuit to do the following. First, choose $r \in [2, N-1]$ uniformly and set $s = r^2 \bmod N$. For each i, compute e_i such that $p_i^{e_i}$ fully divides s. If $\prod_{i=1}^{k} p_i^{e_i} = s$, then s is B-smooth, so output s, r, and $\mathbf{v} = [e_1, \ldots, e_k]^T$.

The total depth of this subcircuit is at most $O(\delta(n) \log^2 n)$, and the probability that it generates an output is at least u^{-3u} where $u = n/(\delta(n) \lg n)$, by Lemma 3.3.1, for n sufficiently large.

We then duplicate this subcircuit creating $k+1$ groups of $(2 \log B) u^{3u}$ subcircuits. Using the replication technique, the probability this results in at least $k+1$ smooth squares is at least $1 - (k+1)/B^2 = 1 - o(1)$.

The size is now $\exp[O(n/\delta(n))]$ since

$$u^{3u} \quad \leq \quad n^{3n/(\delta(n)\lg n)} \quad \leq \quad e^{O(n/\delta(n))}$$

and the k and B factors are absorbed in the constant of proportionality by the bound on $\delta(n)$. The unbounded fan-in depth is still $O(\delta(n)\log^2 n)$, but the bounded fan-in depth has grown to $O(n/\delta(n) + \delta(n)\log^2 n)$.

Step 4.

Now we construct the matrix $\mathbf{A} = [\mathbf{v}_1, \ldots, \mathbf{v}_{k+1}]$. We then search for a non-zero solution to the equation $\mathbf{Ax} = 0$ over $GF(2)$, where we have reduced the entries in $\mathbf{A}$ modulo 2. This can be done by solving the $k+1$ linear equations $\mathbf{A}_j\mathbf{y} = \mathbf{v}_j$ in parallel, where $\mathbf{A}_j$ is simply $\mathbf{A}$ with the jth column removed. One of these equations must give a solution. If we find one for some index j, then $\mathbf{x} = [y_1, \ldots, y_{j-1}, 1, y_j, \ldots, y_k]^T$ is a solution to the original system, where $\mathbf{y} = [y_1, \ldots, y_k]^T$. This can be done in depth $O(\delta(n)^2\log^2 n)$ and size $B^{O(1)}$ (see [BvzGH82, Mul87, PR85]).

Step 5.

Now, using our solution $\mathbf{x}$, we form

$$a = \prod_{j=1}^{k+1} r_j^{x_j} \qquad \text{and} \qquad b = \left(\prod_{j=1}^{k+1} s_j^{x_j}\right)^{\frac{1}{2}} = \prod_{i=1}^{k} p_i^{\frac{1}{2}\sum_{j=1}^{k+1} x_j e_{ij}} \qquad \text{modulo } N.$$

Since a and b are random with $a^2 \equiv b^2 \pmod N$, by Dixon [Dix81], we have $\gcd(a+b, N)$ as a proper factor of N with probability at least 1/2. To compute the greatest common divisor, we use the probabilistic circuit of Adleman and Kompella [AK88] which has unbounded fan-in depth of $O(\log^2 n)$, bounded fan-in depth of $O(\sqrt{n}\log^{2.5} n)$ and size $\exp[O(\sqrt{n\log n})]$. It only succeeds with probability 1/2, but it has zero-sided error, so we can copy it $2\log n$ times and apply the replication technique to decrease its error probability to $1/n$.

The total size of this integer splitting circuit is $\exp[O(n/\delta(n))]$. The unbounded fan-in depth is dominated by solving the linear equations (Step 4) at $O(\delta(n)^2\log^2 n)$. The bounded fan-in depth is $O(\sqrt{n}\log^{2.5} n + \delta(n)^2\log^2 n + n/\delta(n))$.

Since our circuit is composed of large numbers of relatively small circuits together with simple fan-in structures, our circuit is clearly $O(n/\delta(n))$-space uniform. □

If we set $\delta(n) = \log^d n$ for $d > 0$ a constant, we get an $O(\log^{2d+2} n)$ depth circuit of size $\exp[O(n/\log^d n)]$. The bounded fan-in version has optimal depth for $\delta(n) = n^{1/3}/\log^{2/3} n$ giving a depth of $O((n\log n)^{2/3})$ and size $\exp[O((n\log n)^{2/3})]$.

Also notice that this circuit has zero-sided error. If a proper divisor is found, then that is the output, and there is no doubt about its correctness. If only a trivial divisor is found, then the circuit simply turns off its output bit.

3.3.4 A Circuit to Completely Factor N

Now we are interested in constructing a probabilistic circuit which takes as input an n-bit integer N and computes as output the complete factorization of N into powers of primes.

Once we have an algorithm to find a nontrivial divisor of a composite number, creating a sequential algorithm to completely factor that number is relatively easy. Our task is a bit more complex; we want to find a complete factorization in parallel. One approach might be to attempt to show that the factoring algorithm splits N into two divisors of roughly the same size; then we can factor these two divisors in parallel and repeat. The goal would be to show that this process would halt in $O(\log n)$ levels, where each level is the depth of one splitting circuit. This approach might work, however we opt for something a little simpler; by replicating the splitting circuit many times in parallel, there is a good chance that each prime power divisor appears as the output of one of these circuits.

Let $N = \prod_{i=1}^{m} q_i^{e_i}$ be the prime factorization of N. Let $B = n^{\delta(n)}$ as before. Assume that we removed all prime factors below B from N. Then, the number of distinct primes left is $m \leq \log_B N = n/(\delta(n)\lg n)$.

Let s be a quadratic residue modulo N. Then modulo each prime power divisor $q_i^{e_i}$, s has exactly 2 square roots (we can safely assume $B > 2$ so that N is now odd). Thus, modulo N, s has exactly 2^m square roots. Let a and b be arbitrary square roots of s. Then $\gcd(a-b, N)$ will be the product of those prime powers $q_i^{e_i}$ for which $a \equiv b \pmod{q_i^{e_i}}$. Similarly, $\gcd(a+b, N)$ gives those prime powers where $a \equiv -b$.

Let d be the output of a splitting circuit. Then the probability d or N/d is a prime power is at least 2^{-m}, which is no less than $2^{-n/(\delta(n)\lg n)}$. This observation is merely an extension of Dixon's theorem [Dix81].

Corollary 3.3.3 *Let $\delta(n)$ be a non-decreasing function of n, $\delta(n) \leq O(\sqrt{n/\log n})$. There exist both*

1. *an $n/\delta(n)$-space uniform probabilistic circuit family with bounded fan-in of depth $O(\sqrt{n}\log^{2.5} n + \delta(n)^2 \log^2 n + n/\delta(n))$ and size $\exp[O(n/\delta(n))]$, and*
2. *an $n/\delta(n)$-space uniform probabilistic circuit family with unbounded fan-in of depth $O(\delta(n)^2 \log^2 n)$ and size $\exp[O(n/\delta(n))]$*

which, when given as input a positive integer N of n bits in length, output the complete factorization of N as a product of prime powers, with probability $1 - o(1)$.

Proof: We will now describe how to construct the circuit. Correctness will follow from the discussion above.

First we will construct a subcircuit which produces a prime divisor of N with nonzero probability. This subcircuit is composed of four steps.

1. We remove all prime factors smaller than B from N.
2. We factor N using the splitting circuit from Theorem 3.3.2. By the discussion above, the probability that the result is a prime power is at least $2^{-n/(\delta(n)\lg n)}$.

3. We compute the smallest exact integer root of each of the two divisors generated in the previous step using the $\mathcal{NC}$ root computation algorithm of Bach and Sorenson [BS89b] (see chapter 2). (If one of those divisors was a prime power, we now have a prime divisor.)

4. We now take the integer roots from the previous step, which we hope are primes, and perform a prime test on them. To do this, we attempt to factor each $4n^2$ times in parallel. By the replication technique and Theorem 3.3.2, if we do not have a prime, then with probability at least $1 - e^{-n^2}$, for sufficiently large n, we will discover this fact by finding a proper divisor.

So this subcircuit will give a prime divisor of N as its output with probability $2^{-n/(\delta(n)\lg n)}$. Notice that this subcircuit has two-sided error. The probability that it says it has found a prime divisor and is wrong is at most e^{-n^2}. This error is small enough that we can essentially ignore it.

Second, we copy this subcircuit $(3 \log n)2^{n/(\delta(n)\lg n)}$ times using the replication technique to get a circuit which finds some prime divisor of N with probability at least $1 - 1/n^3$. Our "best" subcircuit whose output gets copied is the first one which generates an output.

Third, we copy this circuit n^2 times to get n^2 prime divisors of N, obviously with some of them duplicated. By the above discussion and Dixon's theorem [Dix81], each prime is equally likely to occur, so clearly the probability that one of them is omitted from this list of n^2 primes is $o(1)$. We then sort this list and remove duplicates. We finish by raising the remaining primes to appropriate powers and computing their product. This should equal N. The probability that this results in a complete factorization is $1 - o(1)$.

The size and depth of the resulting circuit are dominated by the size and depth of the integer splitting circuits, with larger constants of proportionality. $\square$

As for the splitting circuit, setting $\delta(n) = \log^d n$ for some constant $d > 0$ gives a polylog depth, subexponential size circuit. The bounded fan-in version has optimal depth for $\delta(n) = n^{1/3}/\log^{2/3} n$ for a total depth of $O((n \log n)^{2/3})$ and size $\exp[O((n \log n)^{2/3})]$. Notice that this circuit has two-sided error in the sense that it never knows if its output is indeed a complete factorization or not, so its correctness bit is always off.

3.4 Discrete Logarithms

In this section we give probabilistic circuits of polylog depth and subexponential size to compute discrete logarithms over certain finite fields. First, we will outline the index-calculus method for computing discrete logarithms as given by Adleman [Adl79]. Second, we will present our circuit for prime finite fields. And finally, we will sketch the construction of a circuit for finite fields of small characteristic.

The index-calculus method for computing discrete logarithms has many algorithmic similarities to the random squares method for factoring integers. Therefore we expect that most of our ideas for factoring integers will apply to computing discrete logarithms. Below we outline Adleman's index-calculus algorithm for the field $GF(p)$.

Adleman's Discrete Logarithm Algorithm

Input: p a prime, g a generator for $GF(p)^*$, and $a \in GF(p)^*$.

Phase I:

1. Find all the primes below B. Call this set of primes the prime base. Let $k = \pi(B)$. Let $p_1, \ldots, p_k$ be the primes of the prime base, and write $p_0 = 1$. Let $l = 2 \log k + 3$.

2. For each i, $1 \leq i \leq k$, repeat (a) and (b) until at least l distinct B-smooth numbers are found, and for $i = 0$ until at least kl such numbers are found:
   a) Choose an integer r uniformly from $[1, p-1]$.
   b) Let $s = g^r p_i \bmod p$. Factor s over the prime base to see if it is smooth.

3. For each smooth number s_j, $1 \leq j \leq 2kl$, compute the indices e_{ij} such that $s_j = \prod_{i=1}^{k} p_i^{e_{ij}}$. Let $\mathbf{v}_j = [e_{1j}, \ldots, e_{kj}]$. For $s_j = g^{r_j} p_i$ with $i > 0$, subtract 1 from the ith entry in $\mathbf{v}_j$.

4. Form $\mathbf{A} = [\mathbf{v}_1, \ldots, \mathbf{v}_{2kl}]^T$ and $\mathbf{r} = [r_1, \ldots, r_{2kl}]^T$. With probability at least $1 - 1/2k$, $\mathbf{A}$ is a rectangular matrix of full rank (see [Pom87]). Find a solution to the equation $\mathbf{Ax} = \mathbf{r} \,(\bmod\, p-1)$. This can be done by factoring $p - 1$ and using the Chinese remainder theorem. Solutions are found modulo each prime power factor and combined. Writing $\mathbf{x} = [x_1, \ldots, x_k]^T$, we have $\operatorname{ind}_g(p_i) = x_i$.

Phase II:

1. Choose $\rho \in [1, p-1]$ until $\alpha = ag^\rho \bmod p$ is B-smooth.

2. Factor α over the prime base to get the equation $\alpha = \prod_{i=1}^{k} p_i^{e_i}$. Taking logs of both sides then gives $\rho + \operatorname{ind}_g(a) = \sum_{i=0}^{k} e_i \operatorname{ind}_g(p_i)$, which determines $\operatorname{ind}_g(a)$.

For a rigorous analysis of the running time and correctness see Adleman [Adl79], Odlyzko [Odl85], or Pomerance [Pom87]. If $B = L(n)^c$ for some small constant c, then the sequential running time of this algorithm is $\exp[O(\sqrt{n \log n})]$, just as for integer factoring.

Theorem 3.4.1 *Let $\delta(n)$ be a non-decreasing function of n, $\delta(n) \leq O(\sqrt{n/\log n})$. There exist both*

1. *an $n/\delta(n)$-space uniform probabilistic bounded fan-in circuit family with depth $O(\delta(n)^2 \log^2 n + \sqrt{n} \log^{3.5} n + n/\delta(n))$ and size $\exp[O(n/\delta(n))]$, and*

2. *an $n/\delta(n)$-space uniform probabilistic unbounded fan-in circuit family with depth $O(\delta(n)^2 \log^2 n + \log^3 n)$ and size $\exp[O(n/\delta(n))]$*

which, on inputs p a prime, g a generator of $GF(p)^*$, and $a \in GF(p)^*$ of total length n in binary, solves the discrete logarithm problem in $GF(p)$ with probability $1 - o(1)$.

Proof: (sketch) The structure of the proof of this theorem strongly resembles that of Theorem 3.3.2, so we merely sketch the differences. Again, let $B = n^{\delta(n)}$.

Steps 2 and 3 compute random powers of the generator and also multiply by primes from the prime base. This uses Adleman and Kompella's modular exponentiation circuit for prime modulus [AK88], which has unbounded fan-in depth $O(\log^3 n)$, bounded fan-in depth $O(\sqrt{n}\log^{3.5} n)$, size $\exp[O(\sqrt{n \log n})]$, and has zero-sided error. We apply the replication technique to reduce its error probability from 1/2 to $o(1)$.

Once the system of linear equations is assembled, in Step 4 a solution is found modulo $p-1$. Since the ring $\mathbf{Z}/(p-1)$ is *well-endowed* as defined by Borodin, Cook, and Pippenger [BCP83], finding a solution to the system is in $\mathcal{NC}^2$. For more details in the sequential case, see Pomerance [Pom87] or Odlyzko [Odl85].

Finally, we add Phase II, which is simply choosing $\rho \in [1, p-1]$ until $g^\rho a$ is B-smooth. We can do this in parallel using techniques similar to what we use to generate random powers of the generator. Finally, we factor over the prime base to get a solution.

The overall size of the circuit is $\exp[O(n/\delta(n))]$. For the unbounded fan-in case, the depth is dominated by the depth of the subcircuit for solving the system of linear equations modulo $p-1$, and modular exponentiation, which is $O(\delta(n)^2 \log^2 n + \log^3 n)$. For the bounded fan-in case, the total depth is at most $O(\delta(n)^2 \log^2 n + n/\delta(n) + \sqrt{n}\log^{3.5} n)$. $\square$

As for the factoring circuits, setting $\delta(n) = \log^d n$ for some constant $d > 0$ gives a polylog depth, subexponential size circuit for computing discrete logarithms. The bounded fan-in version has optimal depth for $\delta(n) = n^{1/3}/\log^{2/3} n$ for a total depth of $O((n \log n)^{2/3})$ and size $\exp[O((n \log n)^{2/3})]$, the same bounds as for integer factoring. Also note that this circuit has zero-sided error.

We can also apply these methods to finite fields of small characteristic; we will now outline the changes necessary.

Our finite field is $GF(q) = GF(p^m)$. We will assume that we are given a representation for the field of the form $GF(p)[x]/(f)$, where p is the characteristic of the field, and $f(x)$ is a monic irreducible polynomial of degree m over $GF(p)$. Elements of the field, then, are represented as polynomials of degree less than m over $GF(p)$.

Our algorithm is based on Hellman and Reyneri's [HR82]. There two main differences from the proof of Theorem 3.4.1. The first is that an element is smooth if it factors as a polynomial into irreducibles of small degree. Thus, the prime base is the set of all monic irreducible polynomials with degree less than some bound $\leq m$. We need something analogous to Lemma 3.3.1 to bound from below the probability that a randomly chosen polynomial is smooth. Odlyzko [Odl85] does give such a result. See also Pomerance [Pom87]. The second difference is that we need modular exponentiation in the field. This problem was shown to be in $\mathcal{NC}$ by Fich and Tompa [FT88], and so is not a difficulty.

Solving the system of linear equations is largely the same. Thus, we have a result similar to Theorem 3.4.1 for finite fields of small characteristic.

The computation of discrete logarithms for finite fields of moderate size characteristic is still largely an open problem in the sequential case. It seems that any new results there would imply parallel algorithms of the type given here, as long as the underlying methods use the index-calculus technique.

3.5 Remarks

We have presented polylog depth probabilistic circuits to factor integers and compute discrete logarithms over finite fields of prime size or small characteristic, but there are still many unresolved questions.

Is it possible to extend our discrete logarithm results to fields of intermediate size characteristic? El Gamal gave heuristic sequential algorithms for the case $GF(p^m)$ for m fixed [EG85b, EG85a]. It might be possible to parallelize these algorithms using our methods, once they are made rigorous.

Our circuits, though subexponential in size, are still quite large. Might it be possible to reduce these very large circuits to something around $\exp[O(\sqrt{n \log n})]$ and still maintain a polylog depth?

Finally, Adleman and Kompella [AK88] suggested their results might lead to sublinear space bounds for computing integer GCD's and modular exponentiation. Similarly, our results might also lead to sublinear space bounds for factoring and discrete logarithms. Intuitively, the parallel computation thesis, which asserts that sequential space and parallel depth (or time) are equivalent in some sense, implies that all these problems might have some kind of sublinear space bounds. And in fact, if these circuits were deterministic instead of probabilistic, this would be the case. Unfortunately, the fact that these circuits are probabilistic seems to be more than simply an inconvenience in proving sublinear space bounds. We will now outline how a sublinear space bound proof would proceed if these circuits were in fact deterministic, followed by an explanation of the difficulty involved in applying this approach to probabilistic circuits.

Borodin [Bor77] proved that, given an $O(S(n))$-space uniform circuit of depth $O(S(n))$ to compute a function f for inputs of length n, then in fact f is computable in space $O(S(n))$. The basic construction is to evaluate the circuit using a Turing machine, and the uniformity condition is used to give a subroutine for constructing the needed pieces of the circuit. The total space used is proportional to the stack space needed for evaluating the circuit plus the space needed to construct pieces of the circuit. The stack space is proportional to the circuit's depth, and the construction space is bounded by the uniformity condition.

Adleman and Kompella's circuits for integer GCD and modular exponentiation are $n^{1/2+o(1)}$-space uniform of depth $n^{1/2+o(1)}$, and our factoring and discrete logarithm circuits are $n^{2/3+o(1)}$-space uniform of depth $n^{2/3+o(1)}$. If these were deterministic circuits, we could apply Borodin's methods to get corresponding sequential space bounds of $n^{1/2+o(1)}$ and $n^{2/3+o(1)}$.

Converting Borodin's results to the standard probabilistic Turing machine model as given by Gill [Gil77] apparently does not work. The reason is that, during the evaluation of the circuit by the Turing machine, the output of any one gate may be needed more than once. Each time the gate's output is evaluated, we must get the same answer. Thus, if that gate performs a coin toss, the same result must be used each time, which seems to imply that the result of the toss must be saved, which in turn means storing the result on tape. This is fine, but the number of coin tosses used by these circuits is much too large for this approach to give sublinear space bounds. One way around this problem is to redefine the probabilistic Turing machine which evaluates the circuit to use a read-only, two-way tape for its source of random coin tosses. But then we have transformed the original problem to that of analyzing the differences between two models of probabilistic Turing machines.

Attempts to prove a sublinear nondeterministic space bound will run into a similar situation. So using this approach to prove sublinear space bounds for these problems seems difficult at best, whether it be on a probabilistic or nondeterministic Turing machine.

Currently, no bound better than linear, deterministic space is known for any of these problems.

Chapter 4

The k-ary GCD Algorithm

4.1 Introduction

Given two positive integers u and v, the greatest common divisor (GCD) of u and v is the largest integer that evenly divides both u and v. In this chapter, we present the k-ary GCD algorithm, a generalization of the binary algorithm of Stein. Interestingly, this new algorithm has both a sequential version that is very practical and parallel versions that rival the best previous parallel GCD algorithms.

Applications for greatest common divisor algorithms are numerous, and include computer algebra systems, symbolic computation, cryptography, and integer factoring. Perhaps the most famous and well-studied GCD algorithm is attributed to Euclid and has a $\Theta(n^2)$ worst-case running time on n-bit inputs. Other algorithms with a $\Theta(n^2)$ running time include the least-remainder Euclidean algorithm, the binary algorithm of Stein (see Brent [Bre76a]), the binary algorithm using left shifts [Bre76a], Purdy's algorithm which exploits carry-free arithmetic [Pur83], and Norton's shift-remainder algorithm [Nor87]. In addition, there are variants of many of these algorithms; for instance Lehmer designed a variant of the Euclidean algorithm for multiple precision inputs [Leh38]. The fastest known sequential GCD algorithm is due to Schönhage [Sch71]. It uses Fast Fourier Transform methods to achieve an $O(n \log^2 n \log \log n)$ running time, but it is not considered practical for reasonably sized inputs. Bshouty [Bsh89] modified Stein's binary algorithm by "compressing" multiple iterations and using precomputed tables. The resulting algorithm uses only $O(n/\log n)$ arithmetic operations and is optimal for certain models of computation, though the algorithm does not appear to be practical. For multiple precision inputs, many consider the binary and left shift binary algorithms to be the best in practice. For general references on GCD algorithms, see Knuth [Knu81b] and Bach and Shallit [BS90].

In this chapter, we show that the k-ary GCD algorithm has a $\Theta(n^2/\log k)$ worst-case running time when k is a prime power. If k is chosen to be roughly n, this gives a subquadratic algorithm. In terms of the number of operations used, this new algorithm matches Bshouty's algorithm. More importantly, we demonstrate that this new algorithm is very practical. We implemented a multiple-precision version of the algorithm, and for k carefully chosen, the k-ary GCD algorithm outperformed several other algorithms, including the Euclidean algorithm by a factor of 12, the binary algorithm by a factor of 2, and the left shift binary algorithm by 35%.

The parallel complexity of computing greatest common divisors is a long-standing open problem. Deng [Den89] showed that certain GCD related problems are $\mathcal{NC}$-equivalent to finding the optimal solution of a two-variable linear program. Although there are several sublinear time parallel GCD algorithms, no $\mathcal{NC}$ algorithm is known, and computing GCD's is not known to be $\mathcal{P}$-complete. See also the paper by Greenlaw, Hoover, and Ruzzo [GHR91].

In this chapter we discuss parallel algorithms in terms of the parallel random access machine (PRAM) model. Previous parallel GCD algorithms include those by Kannan, Miller, and Rudolph [KMR87], who gave the first sublinear time algorithm, Adleman and Kompella [AK88], who gave a polylog time randomized algorithm using a subexponential but superpolynomial number of processors, and Chor and Goldreich [CG90], who currently have the fastest deterministic parallel algorithms. Chor and Goldreich's algorithms are based on the linear time algorithm of Brent and Kung [BK83] for systolic arrays. For a general reference on the PRAM model of computation and parallel algorithms, see Karp and Ramachandran [KR90].

We give four new parallel algorithms based on the k-ary method:

- An EREW PRAM algorithm with a running time of $O(n)$ using $O(n \log n \log\log n)$ processors.
- A CRCW PRAM algorithm with a running time of $O_\epsilon(n/\log n)$ and a CREW PRAM algorithm with a running time of $O_\epsilon(n \log\log n/\log n)$, both using $O(n^{1+\epsilon})$ processors. These algorithms match the complexity bounds obtained by Chor and Goldreich.
- A CRCW PRAM algorithm with a running time of $O(\log^d n)$ using $\exp[O(n/\log^d n)]$ processors for any $d \geq 1$. This deterministic, polylog time algorithm with a subexponential number of processors matches a result of Chor and Goldreich. Adleman and Kompella's algorithm has a running time of $O(\log^2 n)$ using $\exp[O(\sqrt{n \log n})]$ processors, but requires randomness.

Although the k-ary GCD algorithm is more complicated than, say, the Euclidean or the binary algorithms, a straight-forward parallelization of the algorithm is sufficient to rival the best previous algorithms. We believe the k-ary GCD algorithm is simpler than either Bshouty's algorithm or Chor and Goldreich's algorithm, both of which "compress" multiple iterations of simpler sequential algorithms into one iteration using table lookup.

The rest of this chapter is organized as follows: In the next section, we present the k-ary GCD algorithm in detail. In section 4.3, we discuss the sequential version of the algorithm, proving a subquadratic worst-case running time when k is a prime power and presenting the implementation timing results. In section 4.4, we discuss the four parallel algorithms. We conclude with some open problems.

4.2 A Description of the Algorithm

In this section we describe the k-ary GCD algorithm. Although our point of view is sequential in nature, the ideas presented here apply to parallel versions of the algorithm as well.

We begin by presenting the binary algorithm. From that we generalize to the k-ary algorithm, which we describe in detail. We then analyze the number of iterations required by the algorithm.

4.2.1 The Binary Algorithm

Since the k-ary GCD algorithm is a generalization of the binary algorithm, we begin with a review of the latter. Let u and v be the inputs to the algorithm, and assume they are both odd. Then the binary algorithm consists of the following steps:

Binary GCD Algorithm
Input: odd positive integers u, v

```
while $u \neq 0$ and $v \neq 0$ do:
    if $u$ is even, $u := u/2$
    else if $v$ is even, $v := v/2$
    else
        $t := |u - v|/2$;
        if $u > v$ then $u := t$ else $v := t$;
if $u = 0$ then $t := v$ else $t := u$;
Output($t$);
```

Notice that division by 2 does not affect the $\gcd(u,v)$, as u and v are both initially odd. Since each iteration reduces the product uv by a factor of 2 or more, the number of iterations of the algorithm is at most $\lg(uv) + O(1)$.

The binary algorithm is very practical because a multiple precision division subroutine is not required. In addition, the divisions by 2 can often be implemented with a shift operation. For multiple precision inputs, this makes the binary algorithm significantly faster than the Euclidean algorithm on many computers.

There is much literature devoted to the analysis of the binary algorithm, and it has many variations. For more, see Brent [Bre76a], Knuth [Knu81b], or Bach and Shallit [BS90]. For an extended version of the binary algorithm, see Norton [Nor85] or Knuth [Knu81b].

4.2.2 The Main Ideas

We now wish to generalize the binary algorithm to use any positive integer k in place of 2. Let u and v be the inputs to the algorithm, and assume that they are relatively prime to k. Then the *main loop* of the algorithm is as follows:

```
while $u \neq 0$ and $v \neq 0$ do:
    if $\gcd(u,k) > 1$, $u := u/\gcd(u,k)$
    else if $\gcd(v,k) > 1$, $v := v/\gcd(v,k)$
    else
        find nonzero integers $a, b$ satisfying $au + bv \equiv 0 \pmod{k}$;
        $t := |au + bv|/k$;
        if $u > v$ then $u := t$ else $v := t$;
```

Notice that if we set $k = 2$, $a = 1$, and $b = -1$ we have the binary algorithm as a special case.

The integer pairs a, b can be precomputed and stored in a table, as can the $\gcd(x,k)$ for $0 \leq x < k$. The tables are then indexed using $u \bmod k$ and $v \bmod k$. We will postpone the discussion of how these tables are constructed.

If we are to build a GCD algorithm around our main loop, we must address the following two questions:

1. Can the main loop be modified to correctly compute the $\gcd(u,v)$?

2. How large are the values of a and b, and can they be found efficiently?

In answer to the first question, notice that initially u and v are relatively prime to k, so divisions by divisors of k will not affect $\gcd(u,v)$. However, replacing (u,v) by $(\min\{u,v\}, |au+bv|/k)$ does not preserve the greatest common divisor of u and v in general. But we prove the following:

Lemma 4.2.1 *For u, v, a, and b integers, if $g = \gcd(u,v)$ and $h = \gcd(v, au+bv)$, then $g \mid h$ and $h/g \mid a$.*

Proof: The lemma follows by noticing $h = \gcd(v, au)$. □

For the k-ary GCD algorithm to work, then, *before* the main loop we must remove and save common divisors of u and v that may divide possible values of a and b, and *after* the main loop, we must remove the factors introduced by the divisors of a and b. For this to be practical, we must show that a and b are always small, which leads us to the second question above and its answer, the following lemma:

Lemma 4.2.2 *Let $k > 1$ be an integer. For every pair of integers (u,v), there exists a pair of integers (a,b) with $0 < |a| + |b| \leq 2\lceil\sqrt{k}\rceil$ such that $au + bv \equiv 0 \pmod{k}$.*

Proof: We use a simple pigeon-hole argument. Consider the mapping $(a,b) \mapsto au + bv \bmod k$. There can only be k distinct images to this map. If we let a and b range over $1 \leq a, b \leq \lceil\sqrt{k}\rceil + 1$, we have more than k pairs in the domain. Thus there must be two pairs (a_1, b_1) and (a_2, b_2), with $a_1 \neq a_2$ or $b_1 \neq b_2$, that map to the same residue class modulo k. This means the pair $(a_1 - a_2, b_1 - b_2)$ maps to 0, and further it satisfies $0 < |a_1 - a_2| + |b_1 - b_2| \leq 2\lceil\sqrt{k}\rceil$. □

Kannan, Miller, and Rudolph [KMR87] prove results similar to Lemmas 4.2.1 and 4.2.2. Their algorithm searches in parallel for an integer $a \leq n$ such that $au - qv = O(u/n)$, where $q = \lfloor au/v \rfloor$.

Lemma 4.2.2 implies that for k a prime, uv is reduced by a factor of $\Omega(\sqrt{k})$ each time through the main loop, which implies only $O(n/\log k)$ iterations are needed.

The complete algorithm will consist of four phases:

1. The precomputation phase constructs a table of a and b values needed in the main loop and a table containing the $\gcd(x, k)$ for $0 \leq x < k$.
2. The first trial division phase removes and saves common divisors d of u and v satisfying $d \leq \sqrt{k} + 1$ or $d \mid k$.
3. The main loop, which is described above.
4. The second trial division phase removes divisors $d \leq \sqrt{k} + 1$ introduced by the main loop and then restores the divisors saved in phase 2.

We give more details in the next subsection. At this point an example is in order.

An Example

Let $u = 263$, $v = 151$, and $k = 7$. We assume the precomputation phase has been performed, and the first trial division phase finds no common divisors. We proceed to the main loop:

1. We find that $u \equiv v \equiv 4 \pmod 7$, so we use $a = 1$ and $b = -1$, and calculate $|au + bv|/7 = |263 - 151|/7 = 16$ and set $u = 16$ since u was larger.
2. We have $u = 16 \equiv 2$ and $v = 151 \equiv 4 \pmod 7$, so we use $a = 2$ and $b = -1$ to give $|au + bv|/7 = |32 - 151|/7 = 17$, which we assign to v since v was larger.
3. We now have $u = 16 \equiv 2$ and $v = 17 \equiv 3 \equiv -4 \pmod 7$. So we use $a = 2$ and $b = 1$ this time and calculate $|au + bv|/7 = |32 + 17|/7 = 7$ which we assign to v again since it was larger.
4. Now v is divisible by $k = 7$, so we divide to get $v = 1$.
5. We have $u = 16 \equiv 2$ and $v = 1 \equiv 1 \pmod 7$, so we use $a = 1$ and $b = -2$ to give $|au + bv|/7 = |16 - 2|/7 = 2$ which we assign to u.
6. For the last step we have $u = 2$ and $v = 1$ so $a = 1$ and $b = -2$ again to give $|au + bv|/7 = 0$, which we assign to u, and we are done since u is zero.

Since v has no divisors, the final trial division phase has no effect, and we conclude that the GCD of 263 and 151 is 1, the value of v. As both the inputs are in fact prime, this is correct.

4.2.3 The Details

Let u and v be the inputs to the k-ary GCD algorithm. We now give a detailed description of the algorithm by phase.

Phase I: Precomputation

The goal in the precomputation phase is to construct a table A of pairs (a, b) such that, when given integers u and v, we can quickly look in the table to find a and b satisfying $au + bv \equiv 0 \pmod{k}$ with $|a| + |b|$ minimized. We will also compute a table P of prime divisors used in the trial division phases, a table G of GCDs, and a table I of inverses modulo k. All these tables can be implemented using one-dimensional arrays with a total of $O(k)$ storage locations or $O(k \log k)$ bits.

Tables I and G:

We store $\gcd(x, k)$ in $G[x]$, and if $G[x] = 1$ we store $1/x \bmod k$ in $I[x]$, for $0 \le x < k$. Both can be computed using the extended Euclidean algorithm in $O(k \log k)$ arithmetic operations. Alternatively, GCD computations can be avoided as follows: Using the sieve of Eratosthenes, factor all integers up to k and compute the entries for table G from this. Table I can be computed using the identity $x^{-1} \equiv x^{\phi(k)-1} \pmod{k}$.

Table A:

Given u and v relatively prime to k, let $x = u/v \pmod{k}$. Then $u - xv \equiv 0 \pmod{k}$. If we can find a value of a such that both $|a|$ and $|ax|$ are small, we then use $b = -ax \pmod{k}$.

In the main loop we can easily compute $x \equiv u/v$ using table I described above. So in table A we store the optimal value of a for each x.

Table A can be constructed using exhaustive search in $O(k\sqrt{k})$ arithmetic operations, since by Lemma 4.2.2 the largest value of a we ever need is $O(\sqrt{k})$. The table may also be constructed as follows: (1) Loop over all values of a and b up to $\sqrt{k} + 1$. (2) For each pair a, b find all solutions x to the equation $b \equiv -ax \pmod{k}$. Note that not every pair a, b has a solution x. (3) For every solution x with $\gcd(x, k) = 1$, store a in $A[x]$. This requires only $O(k \log k)$ arithmetic operations using an extended GCD algorithm.

Table P:

The prime divisors needed in phases 2 and 4 are stored in table P. These are just the primes below $\sqrt{k} + 1$ and the prime divisors of k. Using trial division and the Sieve of Eratosthenes, all can be found in $O(\sqrt{k} \log k)$ arithmetic operations.

For example, for $k = 7$ we have the following tables:

	x:	0	1	2	3	4	5	6
Table I	x^{-1}:		1	-3	-2	2	3	-1
Table A	a:		1	1	2	2	1	1
Table G	$\gcd(x, k)$:	7	1	1	1	1	1	1
Table P		2	3	7				

Suppose $u \equiv -3$ and $v \equiv 2 \pmod{7}$. We begin by using table I to find $v^{-1} \equiv -3$. We then compute $x = u/v \equiv 2$. Using table A we find $a = 1$, and we compute $b = -ax = -2$. Sure enough, $au + bv \equiv 1 \times (-3) + (-2) \times 2 \equiv 0 \pmod{7}$.

Note that if the k-ary GCD algorithm is to be used more than once, the precomputation phase need not be repeated, as the tables constructed depend only on k and not on the inputs.

Phase II: Trial Division

The purpose of the first trial division phase is to remove all common divisors d of u and v where $d \leq \sqrt{k}+1$ or $d \mid k$. Possible values of d are listed in table P. We save the common divisors found in g.

```
$g := 1$;
for every $d$ in table $P$ do:
    while $d \mid u$ and $d \mid v$ do:
        $u := u/d$; $v := v/d$; $g := g \times d$;
```

Phase III: Main Loop

The main loop of the k-ary GCD algorithm was discussed earlier. Below we show how the tables are used to compute GCDs and find the a, b pairs:

```
while $u \neq 0$ and $v \neq 0$ do:
    $u' := u \bmod k$; $v' := v \bmod k$;
    if $G[u'] > 1$ then $u := u/G[u']$
    else if $G[v'] > 1$ then $v := v/G[v']$
    else
        $x := u' \times I[v'] \bmod k$;
        $a := A[x]$; $b := -a \times x \bmod k$;
        $t := |au + bv|/k$;
        if $u > v$ then $u := t$ else $v := t$;
```

Phase IV: Trial Division

The final trial division phase removes the extraneous divisors introduced during the main loop, and restores those removed in the first trial division phase.

```
if $v = 0$ then $t := u$ else $t := v$;
for every $d$ in table $P$ do:
    while $d \mid t$, $t := t/d$;
$g := t \times g$;
Output($g$);
```

4.2.4 Counting Iterations

We conclude this section by proving upper and lower bounds on the number of iterations of the main loop for all values of $k > 1$. The difficulty here is to bound the number of iterations which perform the transformation $u := u/\gcd(u,k)$. The idea is to show if we repeat this transformation twice, the first time we must get a large GCD. We begin with some definitions.

Let $Q(n) = \min\{\, p^e : p^e \parallel n$, and p is prime $\}$. For example, if n is a prime power, $Q(n) = n$. Also $Q(40) = 5$ and $Q(20) = 4$.

Although not important for our application, note that computing the value of $Q(n)$ is polynomial-time equivalent to factoring n. We also have $Q(n) < \sqrt{n}$ unless n is a prime power, $(1/x)\sum_{n\leq x} Q(n) \sim x/(2\log x)$, and $\#\{n \leq x : Q(n) \leq 23\} > x/2$ for x sufficiently large.

Interestingly, the number of iterations of the main loop of the k-ary GCD algorithm depends on $Q(k)$.

Theorem 4.2.3 *If $k > 1$, $k = o(\log^2(uv))$, and $q = Q(k)$, then on positive integers u and v as input, the k-ary GCD algorithm computes $g = \gcd(u, v)$, and in the worst case the number of iterations is $\Theta(\log(uv)/\log q)$.*

We prove this using the following lemmas:

Lemma 4.2.4 *Let u and k be positive integers, p a prime, and $e > 0$ such that $p^e \parallel k$. Also let $g = \gcd(u, k)$ and $h = \gcd(u/g, k)$. If $p \mid h$ then $p^e \mid g$.*

Proof: Suppose $p \mid h$ and $p^f \parallel g$ with $f < e$. Since $h \mid u/g$, $gh \mid u$ and $p^{f+1} \mid u$. But $p^{f+1} \mid k$ as well, implying $p^{f+1} \mid g$ since $g = \gcd(u, k)$, a contradiction. □

Lemma 4.2.5 *If $k > 1$ and $q = Q(k)$ then the number of iterations of the main loop of the the k-ary GCD algorithm is $O(\log(uv)/\log q)$.*

Proof: Consider any two consecutive iterations of the main loop of the algorithm. It suffices to show that the product uv decreases by a factor d with $\log d = \Omega(\log q)$.

Case 1:

Suppose one of these two iterations performs the transformation

$$(u, v) \rightarrow (\min\{u, v\}, |au + bv|/k).$$

By Lemma 4.2.2, this reduces the product uv by a factor of $d \geq k/(2\lceil\sqrt{k}\rceil) > \sqrt{k}/2 - 1/2$. This gives $\log d = \Omega(\log k)$, and $k \geq q$.

We assumed that $\sqrt{k}/2 - 1/2 > 1$, or $k > 9$. However, one can show only $O(\log(uv))$ iterations are needed for all values of k from 2 to 9; simply construct the tables of a and b values to see that the product uv decreases by a factor larger than 1 at each iteration.

Case 2:

Alternatively, suppose both iterations perform divisions by $\gcd(u, k)$ or $\gcd(v, k)$. Since one of u and v must be relatively prime to k at each iteration, without loss of generality we assume $\gcd(v, k) = 1$. Let $g = \gcd(u, k)$, the divisor used for the first of the two iterations, and $h = \gcd(u/g, k)$, the second divisor used. Let p be a prime divisor of h; p must exist, for g,$h > 1$. By Lemma 4.2.4, $p^e \mid g$, where $p^e \parallel k$, and so $d = gh \geq p^{e+1} > q = Q(k)$. Thus $\log d \geq \log q$, as desired. □

Lemma 4.2.6 *Let $k > 1$ and $q = Q(k)$. There exist infinitely many inputs u, v such that the number of iterations of the main loop of the k-ary GCD algorithm is $\Omega(\log(uv)/\log q)$.*

Proof: Let f be a positive integer, let l be the product of all primes below $\sqrt{k}+1$, and let $u = q^f kl + 1$ and $v = 1$. If f is sufficiently large, we may assume $k = o(\log^2(uv))$. By the prime number theorem (see Hardy and Wright [HW79]), $\log l \sim \sqrt{k} = o(\log(uv))$. Since f is at least $(1/\log q)(\log u - \log l - \log k - 1)$, this means $f = \Omega(\log(uv)/\log q)$. We will show that the algorithm requires at least f iterations.

As u and v are both relatively prime to k and all primes below $\sqrt{k}$, the trial division phase does not alter u or v. The first iteration of the main loop will assign $a = 1$ and $b = -1$, changing u to $|u-v|/k = q^f l$. The next f iterations will each remove a factor q from u, plus common divisors of k and l if there are any. A full f iterations are needed because the GCD computations cannot remove a divisor of q^e larger than q at each iteration as $q = Q(k) \parallel k$.
□

Proof of Theorem 4.2.3: Lemma 4.2.5 implies the algorithm halts, and so correctness follows from Lemma 4.2.1. The number of iterations follows from Lemmas 4.2.5 and 4.2.6.
□

4.3 A Sequential Version

In this section we look at a sequential version of the k-ary GCD algorithm. We start by proving that, on n-bit inputs u and v, the algorithm has an $O(n^2/\log Q(k))$ worst-case running time according to the naive bit complexity model. We then derive an optimal value of k to use in the k-ary GCD algorithm as a function of the input size. This results in an algorithm with a subquadratic running time for k a prime power. Finally, we examine the results of a multiple-precision implementation that compares the k-ary GCD algorithm with several others, including the Euclidean and binary algorithms.

4.3.1 Complexity Results

We will use the naive bit complexity model for our sequential algorithm.

Lemma 4.3.1 *Let $n \geq m$. Multiplication or division of n-bit integers by m-bit integers can be done in $O(n)$ time using precomputed tables of size at most $O(m2^{2m})$ bits.*

Proof: We construct two, two-dimensional arrays M and D.

Let $M[i,j]$ contain the $2m$-bit product $i \times j$ for all integers $0 < i,j < 2^m$. To multiply in $O(n)$ time, view an n-bit integer in base 2^m, use the ordinary left-to-right multiplication algorithm, and at each step the product of m-bit integers is computed through table lookup.

Let $D[i,j]$ contain the m-bit quotient $(i \times 2^m)/j$ for all integers $0 < i,j < 2^m$. The quotient of a $2m$-bit integer by an m-bit can be computed in $O(m)$ time using one table lookup and a subtraction. To divide in $O(n)$ time, again view an n-bit integer in base 2^m, and use the ordinary left-to-right division algorithm.

The total number of entries in each table is $O(2^{2m})$, and each entry is at most $2m$ bits long. □

This also implies that multiplication or division of an n-bit integer by a cm-bit integer, where c is constant, can also be done in $O_c(n)$ time.

Theorem 4.3.2 *Let u and v be the inputs to the k-ary GCD algorithm, and let $n = \lg(uv)$. If $k = O(n^2/\log^2 n)$, then the algorithm takes at most $O(n^2/\log Q(k))$ bit operations.*

Proof: We calculate the complexity of the algorithm by phase:

1. *Precomputation.* This phase can be done using $O(k \log^2 k)$ bit operations.

   We also will apply Lemma 4.3.1 with $m = \lfloor \log k/4 \rfloor$, so that the tables needed are of size $O(\sqrt{k} \log k)$ bits and can be constructed in $O(\sqrt{k} \log^2 k)$ time.

2. *Trial Division.* The two trial division phases have roughly the same cost.

   We alter these phases slightly as follows: For each possible prime divisor p, let p^e be maximal such that $p^e \leq k$. To trial divide u by p, we compute $r = u \bmod p^e$ while saving the corresponding quotient at a cost of $O(n)$ by Lemma 4.3.1. If r is zero, we repeat the process with the quotient. If r was not zero, r must be the largest power of p that evenly divides u.

   By the prime number theorem, there are at most $O(\sqrt{k}/\log k)$ divisions that give a non-zero remainder, one for each value of $d = p^e$, for a cost of $O(n\sqrt{k}/\log k)$. The total cost for divisions with a remainder of zero is $O(n^2 \log k)$, for if $d_1 \ldots d_r$ are the associated divisors, $\sum_{i=1}^{r} \log d_i \leq n$. But each $d_i \geq \sqrt{k}$, which implies $\log d_i = \Omega(\log k)$. Thus there are at most $O(n/\log k)$ divisions.

   So the cost for both trial division phases is $O(n^2/\log k + n\sqrt{k}/\log k)$.

3. *Main Loop.* Each iteration costs at most $O(n)$, and by Theorem 4.2.3 there are $O(n/\log Q(k))$ iterations, giving a cost of $O(n^2/\log Q(k))$.

Thus the total cost is $O(n^2/\log Q(k) + k \log^2 k + n\sqrt{k}/\log k) = O(n^2/\log Q(k))$. □

For k a prime power we have $Q(k) = k$, and the algorithm has an $O(n^2/\log k)$ running time.

Corollary 4.3.3 *Let u, v, and n be as in Theorem 4.3.2, and let k be a power of a prime. Then the optimal value for k is $\Theta(n^2/\log^4 n)$ giving a running time of $O(n^2/\log n)$ for the k-ary GCD algorithm.*

Proof: Differentiate the running time $n^2/\log k + k \log^2 k + n\sqrt{k}/\log k$ with respect to k, set to zero, and solve for k. □

In practice, the product or quotient of 32-bit integers can be computed in $O(1)$ time on most computers. As a result, the tables mentioned in Lemma 4.3.1 are not necessary to give $O(n)$ time operations for $m \leq 32$. Note that we cannot choose k larger than about $10^6 \approx 2^{20}$ in practice because of the space needed for the tables of $O(k)$ entries that arise in the precomputation phase.

4.3.2 Implementation Results

We will now discuss the results from a multiple-precision implementation of several GCD algorithms. In the following discussion, we will present four tables of timing results.

All four tables have the same format: The leftmost column gives the names of the algorithms. Across the top are the input sizes, which range from 10 to 1000 decimal digits in length. Each algorithm was run on 5 pseudo-random pairs of integers of each size. In the box indexed by the algorithm name and input size are two numbers. The top number gives the average time spent by that algorithm on inputs of that size in CPU seconds. The times for the k-ary algorithms include precomputation on every input. The bottom number is the average number of iterations of the main loop for that algorithm.

All the algorithms used the same library of multiple-precision arithmetic routines, written in Pascal, and run on a DECstation 3100 using the Ultrix operating system. The base used for multiple-precision integers was 32768.

Below are the results of timing trials for several previous GCD algorithms.

	Decimal Digits						
Algorithm	*10*	*25*	*50*	*100*	*250*	*500*	*1000*
Euclidean Algorithm	0.005 18	0.019 49	0.049 90	0.178 195	1.070 480	4.972 967	25.948 1929
Least-Remainder Euclidean Alg.	0.004 13	0.015 34	0.039 65	0.135 136	0.791 338	3.591 673	18.488 1340
Purdy's Algorithm	0.005 88	0.016 225	0.044 477	0.130 938	0.652 2385	2.389 4824	9.156 9650
Binary Algorithm	0.003 44	0.009 116	0.024 239	0.066 469	0.313 1167	1.121 2342	4.264 4709
Left-Shift Binary Algorithm	0.004 22	0.011 55	0.025 105	0.069 223	0.280 551	0.916 1087	3.316 2196

If $u > v$, the left shift binary algorithm used here computes an integer e at each iteration such that $2^e v \leq u < 2^{e+1} v$ and then assigns to u the smaller of $u - 2^e v$ and $2^{e+1} v - u$.

We now illustrate how the best value for k increases with the input size. In the table below we have results for the k-ary algorithm using the 10th, 25th, 50th, 100th, 250th, and 500th primes.

	Decimal Digits						
Algorithm	*10*	*25*	*50*	*100*	*250*	*500*	*1000*
29-ary	0.007 15	0.012 38	0.027 77	0.070 151	0.335 378	1.213 754	4.625 1515
97-ary	0.014 12	0.018 28	0.030 59	0.066 121	0.285 304	1.008 609	3.827 1213
229-ary	0.034 10	0.039 27	0.050 55	0.082 107	0.277 268	0.916 530	3.400 1065
541-ary	0.079 10	0.083 24	0.092 48	0.122 95	0.296 237	0.888 482	3.139 967
1583-ary	0.244 9	0.247 21	0.256 43	0.285 87	0.447 217	0.976 430	2.978 854
3571-ary	0.557 8	0.560 19	0.568 40	0.598 81	0.752 202	1.241 398	3.107 790

From this data, we deduce that a reasonable value for k is roughly $8n^2/\log^4 n$ where n is the length of the inputs in base 32768.

Next we investigate whether the running time of the k-ary algorithm is indeed sensitive to the factorization pattern of k as Theorem 4.2.3 suggests. Below we have results for four values of k near 620, each with a different factorization pattern. We also list $Q(k)$ under the name of each algorithm in the table.

	Decimal Digits						
Algorithm	*10*	*25*	*50*	*100*	*250*	*500*	*1000*
619-ary $Q(619) = 619$	0.091 10	0.095 23	0.105 47	0.134 92	0.309 236	0.878 473	3.082 945
621-ary $Q(621) = 23$	0.071 11	0.075 28	0.085 54	0.115 112	0.298 279	0.894 565	3.189 1122
624-ary $Q(624) = 3$	0.058 14	0.062 34	0.073 71	0.106 136	0.301 346	0.947 687	3.440 1375
625-ary $Q(625) = 625$	0.080 10	0.084 25	0.094 51	0.123 103	0.301 264	0.880 518	3.134 1047

For larger inputs, the algorithms' running times are ordered according to the values of $Q(k)$, with the exception of 619 beating 625. In fact, this is what we expect; the 625-ary algorithm often performs divisions by 5.

Finally, we have two adaptive versions of the k-ary algorithm to match against the binary and left shift binary algorithms.

In light of the sensativity of the k-ary algorithm to the factorization of k, we had our adaptive algorithm choose k a prime to insure $Q(k)$ is large.

We also use a *base-adaptive* algorithm, which is an adaptive algorithm that chooses a value of k of near-optimal size which also divides the multiple precision base. In this case this means choosing k a power of 2. The results are below.

	Decimal Digits						
Algorithm	*10*	*25*	*50*	*100*	*250*	*500*	*1000*
Binary	0.003	0.009	0.024	0.066	0.313	1.121	4.264
Algorithm	44	116	239	469	1167	2342	4709
Left-Shift	0.004	0.011	0.025	0.069	0.280	0.916	3.316
Binary Algorithm	22	55	105	223	551	1087	2196
Adaptive k-ary	0.006	0.012	0.025	0.068	0.270	0.861	2.905
	15	36	70	128	271	489	884
$k =$	29	31	43	79	211	463	1151
Base-Adaptive	0.005	0.011	0.023	0.051	0.202	0.612	2.067
k-ary	17	45	81	162	334	601	1125
$k =$	32	32	64	64	256	512	1024

Note that the k-ary algorithms in general are faster than previous algorithms, and adapting k to the multiple precision base is very beneficial.

We also implemented these algorithms on a VAXstation 3200 II running Unix. This machine has a slower processor with a much larger instruction set, and on this machine the k-ary algorithm does slightly better compared to previous algorithms than on the DECstation. One unusual point is that on the VAX, the binary algorithm consistently outperformed the left shift binary algorithm.

That concludes our discussion of sequential implementations for the k-ary GCD algorithm.

4.4 Some Parallel Algorithms

As we mentioned in the introduction, in this section we give four different PRAM algorithms based on the k-ary method discussed in section 4.2. Let n denote the length of the inputs in bits, and let $M(n) = n \log n \log \log n$.

We prove two lemmas concerning the parallel complexity of the precomputation and trial division phases, and then give our PRAM algorithm results.

Lemma 4.4.1 *The preprocessing phase takes $O(\log k)$ time on an EREW PRAM and $O(\log \log k)$ time on a CRCW PRAM, both using $O(k^2 M(\log k))$ processors.*

Proof: Completely factor all the integers up to k in parallel. From this information tables G and P are easily computed. Table I is computed by multiplying together all pairs of integers up to k to see if the result is 1 mod k. Finally, table A is computed using exhaustive search. $\square$

Lemma 4.4.2 *On an EREW PRAM, the trial division phases of the k-ary GCD algorithm take either*

- $O(\sqrt{k} \log^3 n)$ *time and* $O(M(n))$ *processors, or*
- $O(\log n)$ *time and* $(nk)^{O(1)}$ *processors.*

Proof: For the first one, we sequentially test each of the $O(\sqrt{k})$ divisors d by performing a binary search to compute the largest e such that $d^e \mid u$ or $d^e \mid v$. Each binary search will require $O(\log n)$ iterations, constructing d^e will take $O(\log^2 n)$ time, and the division takes $O(\log n \log\log n)$ time, all using at most $O(M(n))$ processors. That proves the first half of the lemma.

For the second one, we simply test all powers of all possible prime divisors in parallel using Beame, Cooke, and Hoover's division algorithm. We then in parallel find the largest power of each divisor that divided both u and v. These powers of divisors are then multiplied together using the iterated product $\mathcal{NC}^1$ reduction to division, and this product is divided out of both u and v. □

Theorem 4.4.3 *Let $\epsilon > 0$. If $k = 2^r$ with $r = \Omega(\log n)$ and $r \le (1/2 - \epsilon)\log n$, then the k-ary GCD algorithm has a running time of $O(n)$ using $O(M(n))$ processors on an EREW PRAM.*

Proof: By Theorem 4.2.3 there are $O(n/\log n)$ iterations of the main loop. Division by k takes $O(1)$ time and $O(n)$ processors, since integers are represented in binary. Thus each iteration of the main loop requires only $O(\log n)$ time and $O(M(n))$ processors. The main loop then takes $O(n)$ time and $O(M(n))$ processors, and so the result follows from Lemma 4.4.1 and the first part of Lemma 4.4.2. □

Theorem 4.4.4 *Let $\epsilon > 0$. If $k = 2^r$ with $r = \lfloor \epsilon \log n \rfloor + O(1)$ such that r is even, then the k-ary GCD algorithm has running times of $O_\epsilon(n/\log n)$ and $O_\epsilon(n \log\log n/\log n)$ on CRCW and CREW PRAMs respectively, using $O(n^{1+\epsilon})$ processors.*

Proof: By Theorem 4.2.3 there are $O_\epsilon(n/\log n)$ iterations of the main loop. We will show how each iteration takes $O(1)$ time on a CRCW PRAM. We use many ideas from Chor and Goldreich [CG90].

As in Theorem 4.4.3, division by k takes $O(1)$ time and $O(n)$ processors since integers are represented in binary. We also precompute a table of $O(k)$ entries to contain the products of all integers $\le O(\sqrt{k})$, allowing the product of integers with $O(\log k)$ bits to be computed in $O(1)$ time using a table lookup. This additional precomputation time is $O(\log\log k)$ using $O(kM(\log k))$ processors, easily within our claimed bound. We now can compute each iteration in $O(1)$ time and $O(M(n))$ processors aside from computing the products au and bv. For this we use the following idea in Chor and Goldreich.

To compute au, do the following. First write u in base $\sqrt{k} = 2^{r/2}$ as $u = \sum_{i=0}^{l} u_i(\sqrt{k})^i$ where l is roughly $2\log u/\log k$. Note that since r is even, the u_i can be computed in $O(1)$ time. Second, compute x and y such that $u = x + y$ where x is the sum of the odd terms in the above expansion for u, and y is the sum of the even terms. For example, if $\sqrt{k} = 10$ (for human readability) and $u = 123456$, then $x = 103050$ and $y = 020406$. The products ax and ay can then be computed in $O(1)$ time with table lookups using $O((n/\log\sqrt{k})\cdot k)$ processors, since there are no carries. We then compute $au = ax + ay$. Following our example, if $a = 5$ then $ax = 515250$ and $ay = 102030$, giving $au = 617280$. By our choice of k, this uses $O(n^{1+\epsilon})$ processors. We compute bv the same way.

Since $k = O(n^\epsilon)$, the theorem now follows from Lemma 4.4.1 and the first part of Lemma 4.4.2 for the CRCW case.

The CREW PRAM algorithm is much the same, however a multiplicative factor of $O(\log\log k) = O(\log\log n)$ time is needed for table lookups, and the additions must be pipelined. □

The results of this last theorem match those by Chor and Goldreich [CG90]. Note that they use a common CRCW PRAM; it allows concurrent writes only if the same value is being written. Theorem 4.4.4 can be easily modified to use the same model.

Corollary 4.4.5 *Let $d \geq 1$. If $k = 2^r$ for $r = \lfloor n/\log^d n \rfloor$, then the k-ary GCD algorithm takes $O(\log^d n)$ time and $\exp[O(n/\log^d n)]$ processors on a CRCW PRAM.*

Proof: Follows from the proof Theorem 4.4.4, only use the second part of Lemma 4.4.2. □

4.5 Remarks

We have show that the k-ary GCD algorithm gives both practical sequential algorithms and several interesting parallel algorithms. We close with a few questions and comments, and we mention some current and future work:

- Another special case of the k-ary GCD algorithm is the trinary algorithm ($k = 3$) which was given by Knuth [Knu81b, Section 4.5.2, Exercise 5].

- One important application for GCD algorithms is integer factoring. The k-ary GCD algorithm is especially well-suited for factoring algorithms that use many GCD calculations, for in this case the precomputation step need only be done once, and in many cases the first trial division phase can be skipped altogether. Examples of such factoring algorithms include the various cyclotomic methods [BS89a, Pol74, Wil82] and Pollard's ρ-method [Pol75]. See also the paper by Montgomery [Mon87].

- Current work on the k-ary GCD algorithm includes the developement of a left-shift version of the algorithm that makes use of Farey fractions (see Hardy and Wright [HW79]) and the extension of both algorithms to compute multiplicative inverses.

- Sam Pottle, Rob Winter, and the author are implementing several parallel versions of the k-ary GCD algorithm. Two implementations are planned for the Sequent Symmetry, one of which uses a software pipeline for the main loop. An additional implementation is planned for the Connection Machine.

- We proved that the number of iterations of the main loop is $\Theta(n/\log Q(k))$. Is it possible to compute the implied constant? Does the average number of iterations depend on the factorization of k as well?

- In the worst case, the second trial division phase might involve the complete factorization of an integer with $\Omega(\log(uv))$ bits. Is this indeed true? Is this the average case, or can one show this integer is small on the average? If this number is large on the average, it may be that periodic trial division steps within the main loop to remove the extraneous divisors accumulated so far will speed the algorithm.

- D. H. Lehmer gave a multiple precision variant of the Euclidean algorithm [Leh38], and Knuth mentions a similar variant for the binary algorithm due to R. W. Gosper (see Knuth [Knu81b]). Is such a variant possible for the k-ary algorithm?

Chapter 5

An Analysis of Pollard's $p-1$ Factoring Algorithm

5.1 Introduction

Given a positive integer n, the *complete factorization problem* is to write n as a product of prime powers. In this chapter, we estimate the number of integers that can be completely factored in random polynomial time using Pollard's $p-1$ factoring algorithm.

Traditional algorithm analysis focuses on finding the worst-case running time of an algorithm. This information often predicts the behavior of an algorithm in practice. It also provides a basis for comparing different algorithms that solve the same problem.

However, many integer factoring algorithms with an exponential worst-case running time have the ability to factor numbers of a special form in polynomial time. One example of this is the trial division algorithm, which can quickly factor integers that are composed entirely of small primes. Note that practitioners often use trial division as their first choice when trying to factor a number, resorting to algorithms with a better worst-case running time, such as the elliptic curve method or quadratic sieve, when it appears trial division will take too long.

So in addition to the worst-case running time, it is useful to know the number of integers an algorithm can factor in random polynomial time. To be precise, we wish to compute $F(x,t,A)$, the number of integers $n \leq x$ that algorithm A can completely factor with probability at least $1/2$ using at most t arithmetic operations. By arithmetic operations, we mean additions, subtractions, multiplications, divisions, gcd computations, and comparisons of $O(\log x)$-bit integers. Using classical methods such as those described in Knuth [Knu81b] these operations have bit complexity $O(\log^2 x)$. Using fast multiplication techniques, this reduces to $(\log x)^{1+o(1)}$ [SS71, Sch71].

Thus, $F(x,(\log x)^{O(1)},A)$ gives the number of integers that algorithm A can factor in random polynomial time. Also, the limit of $F(x,(\log x)^{O(1)},A)/x$ as $x \to \infty$ gives the density of this set of integers. Previously, $F(x,t,A)$ was estimated for trial division and the elliptic curve method by Hafner and McCurley [HM89]. In this chapter, we estimate $F(x,t,A)$ for Pollard's $p-1$ integer factoring algorithm. As a consequence, we show that the $p-1$ algorithm factors more integers than trial division, but fewer than the elliptic curve method. As an immediate consequence, the density of integers factorable by the $p-1$ method is zero. We discuss both this previous work and our new results below.

Previous Work

Knuth and Trabb Pardo [KTP76] analyzed the trial division algorithm (TD) and gave estimates for $F(x, x^{1/u}, TD)$ when $u \geq 1$ is fixed.

Hafner and McCurley [HM89] analyzed a trial division algorithm that uses a probabilistic compositeness test (TDC) and a version of the elliptic curve algorithm that has an initial trial division phase and a probabilistic compositeness test (ECM). For trial division, they showed that if $\log t = o(\log x)$ and $t/(\log^2 x \log\log x) \to \infty$ then

$$F(x,t,TDC) \sim e^{\gamma}\frac{x}{\log x}\log t, \tag{5.1}$$

where γ is Euler's constant, and $e^{\gamma} = 1.78107241\cdots$. This implies TDC can completely factor $\Theta(x\log\log x/\log x)$ integers below x in random polynomial time. For the elliptic curve method, they showed that for any $C < 6/5$, if $t \geq \log^4 x$ and $\log t = o((\log x)^{1/C})$ then

$$F(x,t,ECM) \gg \frac{x}{\log x}(\log t)^C. \tag{5.2}$$

This implies that for any $\delta > 0$, ECM can completely factor at least $(x/\log x)(\log\log x)^{6/5-\delta+o(1)}$ integers in random polynomial time.

Assuming the Riemann hypothesis, Hafner [Haf90] improved on this by showing that if $a \geq 4$ then

$$F(x, \log^a x, ECM) \gg_a \frac{x}{\log x}\frac{(\log\log x)^2}{\log\log\log x}.$$

New Results

We analyze Pollard's $p-1$ integer factoring algorithm. We assume this algorithm uses a probabilistic compositeness test. Our results imply that the $p-1$ algorithm factors more integers than trial division, but fewer than the elliptic curve method. We also show that the density of integers factorable by the $p-1$ method is zero. We elaborate below.

Let $\delta > 0$. We show that if $t \geq (\log x)^{2+\delta}$ and $\log t \leq (\log x)^{1-\delta}$ then

$$\liminf \frac{F(x,t,p-1)}{e^{\gamma}(x/\log x)\log t} > 1. \tag{5.3}$$

Under the same conditions, we also show that

$$\limsup \frac{F(x,t,p-1)}{e^{\gamma}(x/\log x)\log t} < \infty. \tag{5.4}$$

Thus the $p-1$ algorithm factors $\Theta(x\log\log x/\log x)$ integers in random polynomial time. If we assume that for a prime p, $p-1$ factors completely over small primes with the same probability as a randomly chosen integer near p, we show that

$$F(x,t,p-1) \sim c_o \cdot e^{\gamma}\frac{x}{\log x}\log t \tag{5.5}$$

where $c_o = 1.685\cdots$.

Note that (5.1) and (5.3) imply that for a given t, the $p-1$ algorithm factors more integers than trial division, which is interesting considering they have approximately the same worst-case running time. Also (5.2) and (5.4) imply that the elliptic curve method outperforms the $p-1$ algorithm. These findings are not surprising; in fact they substantiate the common belief among researchers that the $p-1$ method falls between trial division and the elliptic curve method in its effectiveness.

The rest of this chapter is organized as follows. In section 5.2 we review some facts about the $p-1$ algorithm, characterize those integers it can factor, and introduce a purely number theoretic function to count them. In sections 5.3 and 5.4 we approximate this function. We conclude in section 5.5 with a few remarks.

5.2 Pollard's $p-1$ Factoring Algorithm

We begin by reviewing Pollard's $p-1$ algorithm.

Let p and q be primes, and we wish to factor $n = pq$. Further suppose $p-1$ factors completely over prime powers below a bound y. Let $p_1, \ldots, p_r$ be the primes below y, and let e_i be the largest integer such that $p_i^{e_i} \leq y$. Let $E = \prod_{i=1}^{r} p_i^{e_i}$, and let M be odd and $e \geq 0$ such that $E = 2^e M$.

Choose $a \in (\mathbf{Z}/(n))^*$ uniformly at random, and compute $x = a^E \bmod n$. Since $p-1 \mid E$, we have $x \equiv 1 \pmod{p}$ by Fermat's Theorem. If $q-1 \nmid E$, then with probability at least $1/2$, the order of a modulo q does not divide E, and so $x \not\equiv 1 \pmod{m}$. In this case $d := \gcd(x-1, n)$ is a proper divisor of n.

If $q-1 \mid E$, then with probability at least $1/2$, there exists an integer j, $1 \leq j < e$, such that $a^{2^j M} \not\equiv \pm 1 \pmod{n}$ but $a^{2^{j+1} M} \equiv 1 \pmod{n}$. In this case, $d := \gcd(a^{2^j M} - 1, n)$ is a proper divisor of n.

For proofs, see Bach, Miller, and Shallit [BMS86] or Bach, Giesbrecht, and McInnes [BGM91]. Note that this method will work when n is the product of more than two primes.

The procedure described above will find a divisor of n with probability at least $1/2$. By repeating the procedure $\lceil 2\lg\lg n \rceil$ times, we can reduce the probability of error to $O(1/\log^2 n)$. In addition we add a probabilistic compositeness test such as Solovay-Strassen [SS77] or Miller-Rabin [Mil76, Rab80], to determine if the divisors found by the above method are prime (with some chance for error) and to know when n has been completely factored. We also use a perfect power testing algorithm such as one by Bach and Sorenson [BS89b] to insure that n is not a prime power (see chapter 2). Below is a pseudocode description of how the $p-1$ algorithm can be implemented to completely factor an integer n.

```
The $p-1$ Factoring Algorithm
Input: a positive integer $n$ and bound $y$

    Function Split($n$,$E$):
        Compute $M$ odd such that $E = 2^e \cdot M$;
        Repeat up to $\lceil 2\lg\lg n \rceil$ times:
            Choose $a \in (\mathbf{Z}/(n))^*$ uniformly at random;
            $x := a^M \bmod n$;
            Repeat $e+1$ times:
                $d := \gcd(x-1, N)$;
                If $1 < d < n$ then Return($d$); { Divisor Found }
                $x := x^2 \bmod n$;
        Report failure and Halt;

    Procedure Factor($n$,$E$):
        If $n$ is probably a prime power then
            Output($n$);
        Else
            $d$ :=Split($n$,$E$);
            Factor($d$,$E$);
            Factor($n/d$,$E$);

    Mainline:
        Find all the primes $p_i \le y$, and compute the $e_i$.
        Let $E = \prod p_i^{e_i}$;
        Factor($n$,$E$);
```

Note that E and M do not have to be explicitly calculated; a list of primes may be passed to the function Split and the calculation $x := a^M \bmod n$ can be done in a loop:

```
$x := a$;
For each $p_i \ne 2$ do:
    $v := p_i^{e_i}$;
    $x := x^v \bmod n$;
```

Let us informally compute the number of arithmetic operations performed by the $p-1$ algorithm.

Function Split essentially computes $a^E \bmod n$ at most $O(\log\log n)$ times for a total of $O(\log E \log\log n)$ operations. Since $\log E \sim y$ by the prime number theorem, this is $O(y \log\log n)$.

Let $\omega(n)$ denote the number of prime divisors of n. Note that $\omega(n) = O(\log n)$. Procedure Factor invokes function Split at most $\omega(n)$ times for $O(y\omega(n)\log\log n)$ operations. Procedure

Factor also performs up to $\omega(n)$ prime power and compositeness tests. At $O(\log n \log\log n)$ operations apiece, this is $O(\log^2 n \log\log n)$ operations. Let $\delta > 0$. If $y \geq (\log n)^{2+\delta}$ then the cost of compositeness and perfect power tests is negligible. So in this case procedure Factor uses $O(y\omega(n)\log\log n)$ arithmetic operations total.

Finding the primes up to y can be done with the sieve of Eratosthenes in $O(y \log\log y) = O(y \log\log n)$ operations. The e_i can be found in $O(1)$ operations for each prime p_i by using the approximation $e_i = \lfloor \log y / \log p_i \rfloor$, for a total of $O(y/\log y)$ operations by the prime number theorem.

Thus, the total number of operations used by the $p-1$ algorithm is $O(y\omega(n)\log\log n)$.

Obviously the algorithm can not compute $\omega(n)$ before factoring n. So we will assume that $\omega(n) \leq 5\log\log x$, and if n has more divisors than that, the algorithm reports failure.

Note that in the worst case, the $p-1$ algorithm uses at most $n^{1/2+o(1)}$ arithmetic operations, the same as trial division, since y must be chosen close to $\sqrt{n}$ to insure all divisors are found with high probability.

To summarize, the $p-1$ algorithm uses the primes below a bound y to attempt to remove prime divisors from n. If we give the algorithm t arithmetic operations, then

- $y = \Theta(t/(\omega(n)\log\log n)) = \Theta(t/(\log\log n)^2)$ and
- the $p-1$ algorithm can completely factor n if $n = p^e \cdot m$ where p is prime and for every prime q dividing m, $q-1$ factors completely over the primes below y.

We now make some definitions. Let $P(m)$ denote the largest prime divisor of m. Let $G(y)$ be the set of all integers $m > 0$ such that, for every prime q dividing m, $P(q-1) \leq y$. Let $\Lambda(x,y)$ be the number of integers $n \leq x$ such that $n = p^e m$ where p is prime, e is a positive integer, and $m \in G(y)$.

We have the following:

Lemma 5.2.1 *Let $\delta > 0$. If $t \geq (\log x)^{2+\delta}$ then*

$$F(x,t,p-1) \quad = \quad \Lambda(x, t^{1+o(1)}) + o(x/\log x).$$

Proof: The algorithm chooses $y = \Theta(t/(\log\log x)^2) = t^{1+o(1)}$ when $t \geq (\log x)^{2+\delta}$. So we need to show that $n = p^e \cdot m$ for p prime and $m \in G(y)$ if and only if the $p-1$ algorithm can completely factor n with probability $\geq 1/2$, with a number of exceptions not to exceed $o(x/\log x)$.

Suppose the $p-1$ algorithm completely factors the integer n with probability $\geq 1/2$. Let q be a prime divisor of n removed by the algorithm. Then the algorithm can find an integer a such that $a^E \equiv 1 \pmod{q}$ with probability at least $1/2$. If there is a prime divisor $r > y$ with $r \mid q$, clearly $r \not| E$. Since a is chosen with order dividing $q-1$ but not dividing r, the number of such a is at most $(q-1)/r$. Thus the probability of finding such an a is $O(1/r) = O(1/y)$. This is a contradiction; therefore there exists no divisor $r > y$ of $q-1$, and hence $q \in G(y)$. Thus n is of the form $p^e \cdot m$ with p prime and $m \in G(y)$.

Suppose $n = p^e \cdot m$ with p prime and $m \in G(y)$. By our discussion above, if $q^f \parallel m$, then the $p-1$ algorithm can remove the divisor q^f with high probability if $\phi(q^f) = q^{f-1}(q-1) \mid E$. Thus n falls into one of the following classes:

1. the $p-1$ method factors n with probability $\geq 1/2$,
2. n has more than $5 \log\log x$ prime factors,
3. there is a prime $q \mid m$ and a prime r and $g > 1$ such that $r \leq y$, $r^g \mid q-1$, and $r^g > y$,
4. or there is a prime $q \mid n$ with $q > y$ and $q^2 \mid n$.

It suffices to show that the number of integers n in classes (2) (3) and (4) is $o(x/\log x)$.

From Erdős and Sarközy [ES80], the number of $n \leq x$ in class (2) is $o(x/\log x)$.

Let $B(x,y)$ be the set of primes $q \leq x$ with $q \in G(y)$ such that there exists a prime $r < y$ with $r^g \mid q-1$, $r^g > y$. By the prime number theorem for arithmetic progressions,

$$\#B(x,y) \ll \sum_{r \leq y,\ r^g \geq y} \frac{x}{r^g \log x}.$$

Let $v(r)$ be the smallest integer such that $r^{v(r)} > y$. Since $\sum_{k=g}^{\infty} 1/r^k = O(1/r^g)$, by splitting the sum according the the values for $v(r)$ we have

$$\begin{aligned}
\#B(x,y) &\ll \sum_{i=2}^{\lg y} \left(\sum_{y^{1/i} < r \leq y^{1/(i-1)}} \frac{x}{r^i \log x} \right) \\
&\ll \sum_{i=2}^{\lg y} \frac{x}{y^{1-1/i} \log x} \quad = \quad O\left(\frac{x}{\sqrt{y} \log x} \right).
\end{aligned}$$

Then the number of integers $n \leq x$ in class (3) is at most

$$\begin{aligned}
\sum_{q \in B(x,y)} \frac{x}{q} &= O(1) + \int_2^x \frac{x}{t} d(\#B(t,y)) \\
&\ll \int_2^x \frac{x}{\sqrt{y} t \log t} dt + O\left(\frac{x}{\sqrt{y} \log x} \right) \quad \ll \quad \frac{x \log\log x}{\sqrt{y}}
\end{aligned}$$

using integration by parts. Since $y \gg (\log x)^{2+\delta/2}$, this is $o(x/\log x)$.

Finally, the number of integers $n \leq x$ in class (4) is at most

$$\sum_{q \geq y} \frac{x}{q^2} \ll \frac{x}{y} = o(x/\log x).$$

□

There exist several extensions and improvements to the $p-1$ algorithm, which have little effect on our results, and we will not discuss them here. For details, see Montgomery [Mon87], Montgomery and Silverman [MS90], and Pollard [Pol74].

5.3 Estimating $\Lambda(x,y)$

Now we concentrate on estimating $\Lambda(x,y)$.

Let $\Psi(x,y)$ be the number of integers $n \le x$ such that $P(n) \le y$. Let $G(y,z)$ be the set of all integers m such that $m \in G(y)$ and $P(m) \le z$. Let $\pi'(x)$ be the number of prime powers $p^e \le x$. Also let

$$s(y) \;=\; \prod_{q>y,\ q\in G(y)} \left(1+\frac{1}{q-1}\right).$$

Theorem 5.3.1 *Let $\delta > 0$. If $\log y \le (\log x)^{1-\delta}$ and $y > (\log x)^{2+\delta}$ then*

$$\Lambda(x,y) \;=\; s(y)\cdot e^{\gamma}\frac{x}{\log x}(\log y)\left(1+O\left(\frac{1}{\log y}+\frac{1}{\log^{\delta/3} x}\right)\right).$$

To prove this, we use the following lemma.

Lemma 5.3.2 *Let $\delta > 0$. If $\log y \le (\log z)^{1-\delta}$, then*

$$\sum_{q>z,\ q\in G(y)} \frac{1}{q} \;\le\; \sum_{m>z-1,\ P(m)\le y} \frac{1}{m} \;\le\; \exp\left[-(\log z)^{\delta}\right]$$

for all $z > z_0(\delta)$.

Proof: The first inequality is clear since if $q > z$ and $q \in G(y)$, then $1/q < 1/(q-1)$ where $q-1$ is an integer m with $m > z-1$ and $P(m) \le y$. For the second sum we use partial summation, obtaining that it is at most

$$\int_{z-1}^{\infty} \frac{1}{t^2}\Psi(t,y)dt.$$

Using the estimate from Canfield, Erdős, and Pomerance [CEP83] on $\Psi(x,y)$, we get the upper bound $\exp[-(\delta+o(1))(\log z)^{\delta}\log\log z]$ from which the result follows. □

Proof of Theorem 5.3.1: Let $z = \exp[(\log x)^{1-\delta/2}]$. If n is counted by $\Lambda(x,y)$, then n is in at least one of the following classes:

1. n is divisible by a prime $q \in G(y)$ with $q > z$,
2. n has more than $5\log\log x$ prime factors,
3. $n \in G(y,z)$,
4. all other n counted by $\Lambda(x,y)$.

We first show that the contribution to $\Lambda(x,y)$ from n in the first 3 classes is $o(x/\log^2 x)$, so is negligible.

The number of $n \le x$ in the first class is at most

$$\sum_{q>z,\ q\in G(y)} \frac{x}{q} \quad \ll \quad x\cdot\exp\left[-(\log z)^{\delta/2}\right] \quad = \quad o(x/\log^2 x),$$

by Lemma 5.3.2. From Erdös and Sarközy [ES80] it follows that the number of $n \le x$ in class (2) is $o(x/\log^2 x)$. Finally, the number of $n \le x$ in class (3) is at most $\Psi(x,z) = o(x/\log^2 x)$ from [CEP83].

Let $w = \exp[(\log x)^{1-\delta/3}]$. If n is in class (4), then n has a unique representation as $p^e m$, where $m \in G(y,z)$, $m \le w$, and $p \notin G(y,z)$. Indeed, we only need to prove $m \le w$. But since n is not in classes (1) or (2), we have $m \le z^{5\log\log x} \le w$ for all large x.

We conclude that

$$\Lambda(x,y) \quad = \quad \sum_{\substack{m\le w \\ m\in G(y,z)}} \ \sum_{\substack{p^e\le x/m \\ p\notin G(y,z)}} 1 \ + \ o(x/\log^2 x).$$

Since $m \le w$, the inner sum is

$$\begin{aligned}
\pi'(x/m) - O\left(\sum_{p\in G(y,z)} \log(x/m)\right) \quad &= \quad \pi'(x/m) + O(z\log(x/m)) \\
&= \quad \frac{x/m}{\log(x/m)} + O\left(\frac{x/m}{\log^2(x/m)}\right) \\
&= \quad \frac{x}{m\log x}\left(1 + O(1/\log^{\delta/3} x)\right).
\end{aligned}$$

Thus the theorem will follow if we show

$$\sum_{m\le w,\ m\in G(y,z)} \frac{1}{m} \quad = \quad s(y)\cdot e^{\gamma}(\log y)\left(1 + O(1/\log y)\right). \tag{5.6}$$

First we note that the restriction $m \le w$ in (5.6) may be dropped, since Lemma 5.3.2 implies

$$\sum_{m>w,\ m\in G(y,z)} \frac{1}{m} \quad = \quad o(1).$$

But

$$\begin{aligned}
\sum_{m\in G(y,z)} \frac{1}{m} \quad &= \quad \prod_{q\in G(y,z)} \left(1 + \frac{1}{q} + \frac{1}{q^2} + \cdots\right) \quad = \quad \prod_{q\in G(y,z)} \left(1 + \frac{1}{q-1}\right) \\
&= \quad \prod_{q\le y}\left(1+\frac{1}{q-1}\right) \prod_{q>y,\ q\in G(y)} \left(1+\frac{1}{q-1}\right) \prod_{q>z,\ q\in G(y)} \left(1+\frac{1}{q-1}\right)^{-1}.
\end{aligned}$$

The first product is $e^{\gamma}(\log y) + O(1)$; this is Mertens's Theorem (see Hardy and Wright [HW79]). The second product is $s(y)$ by definition. Finally, the logarithm of the last product is

$$-\sum_{q>z,\ q\in G(y)} \log\left(1+\frac{1}{q-1}\right) \ll \sum_{q>z,\ q\in G(y)} \frac{1}{q} = o(1/\log y),$$

by Lemma 5.3.2. We have (5.6). □

5.4 Results on $s(y)$

Now we need to bound $s(y)$.

Theorem 5.4.1 $1 < s(y) \ll 1$.

From Theorem 2.2 in Pomerance [Pom89] we have $\log s(y) \sim \sum_{q\in G(y),\ q>y} 1/q = O(1)$ so that $s(y) \ll 1$. It remains to show that $s(y) > 1$, which we do with the following lemma. Let $\Pi(x,y)$ denote the number of primes $q \le x$ such that $q \in G(y)$.

Lemma 5.4.2 *If $a > 1$ and $b > 0$ are such that $\Pi(x, x^{1/a}) > b\pi(x)$ for sufficiently large x, then* $\liminf s(y) \ge a^b > 1$.

Proof: Simply integrate by parts twice and use the prime number theorem:

$$\begin{aligned}\log s(y) &\sim \sum_{q\in G(y),\ q>y} \frac{1}{q} = \int_y^\infty \frac{1}{t}d\Pi(t,y)\\ &= \left.\frac{\Pi(t,y)}{t}\right|_y^\infty + \int_y^\infty \frac{\Pi(t,y)}{t^2}dt \ \ge\ O(1/\log y) + \int_y^{y^a} \frac{b\pi(t)}{t^2}dt\\ &= O(1/\log y) + \sum_{y<p<y^a} \frac{b}{p} \ \sim\ b\log a.\end{aligned}$$

□

Pomerance [Pom80] proves the lower bound $\Pi(x,\sqrt{x}) \ge b\pi(x)(1+o(1))$ for $b = 1 - 4\log(5/4) \approx 0.10742$. Using this, Lemma 5.4.2 gives $s(y) \ge 1.0773$ for large y.

Lemma 5.2.1 and Theorems 5.3.1 and 5.4.1 combine to prove (5.3) and (5.4).

We conclude this section by giving a heuristic approximation for $s(y)$, which leads to the estimate $s(y) \sim 1.685\cdots$. Our heuristic is the following: for q a prime, $q-1$ factors completely over the primes below y with the same probability as a randomly chosen integer below q.

Let $\rho(u)$ denote Dickman's rho function. The probability a randomly chosen number below x factors completely over primes below y is asymptotically $\rho(\log x/\log y)$. (See, for example, Hildebrand [Hil86].) So we are assuming the following.

Hypothesis 5.4.3 *Let $u = \log x/\log y$. Then $\Pi(x,y) \sim \pi(x)\rho(u)$ when $1 \le u \le \log\log y$.*

Theorem 5.4.4 *Hypothesis 5.4.3 implies* $s(y) \sim \exp\left[\int_1^\infty (\rho(u)/u)du\right]$.

Proof: Let $\delta > 0$ and let $z = \exp[(\log y)^{2+\delta}]$. By Lemma 5.3.2 we have

$$\begin{aligned}
\log s(y) &= o(1) + \sum_{q \in G(y),\ q>y} \frac{1}{q} = o(1) + \sum_{q\in G(y,z),\ q>y} \frac{1}{q} \\
&= o(1) + \int_y^z \frac{1}{t} d\Pi(t,y) \\
&= o(1) + \int_y^{y^{\log\log y}} \frac{1}{t} d\Pi(t,y) + \int_{y^{\log\log y}}^z \frac{1}{t} d\Pi(t,y) \\
&= o(1) + E_1 + E_2,
\end{aligned}$$

say. For E_1, Hypothesis 5.4.3, the prime number theorem, and integration by parts gives

$$E_1 = o(1) + \int_1^{\log\log y} \frac{\rho(u)}{u} du = o(1) + \int_1^\infty \frac{\rho(u)}{u} du.$$

For E_2, integration by parts and noticing $y \le t^{1/\log\log y}$ gives

$$E_2 \ll \int_{y^{\log\log y}}^z \frac{\Pi(t, t^{1/\log\log y})}{t^2} dt \le \int_{y^{\log\log y}}^z \frac{\Psi(t, t^{1/\log\log y})}{t^2} dt.$$

Using Hildebrand [Hil86] this gives

$$E_2 \ll \int_{y^{\log\log y}}^z \frac{\rho(\log\log y)}{t} dt \le \rho(\log\log y)\log z = o(1).$$

□

Using Simpson's rule to approximate $\rho(u)$ and the above integral leads to the estimate $s(y) = 1.685\cdots$.

Let $s^*(y)$ be the same as $s(y)$, but with the infinite product truncated to 10^9. We wrote a program to calculate actual values of $s^*(y)$, and the table below compares these actual values to what our hypothesis predicts for $s^*(y)$.

	$s^*(10)$	$s^*(100)$	$s^*(1000)$	$s^*(10^4)$	$s^*(10^5)$
Predicted Value	1.685	1.685	1.674	1.620	1.514
Actual Value	1.887	1.817	1.760	1.677	1.548

The fact that $p-1$ is even makes it slightly more likely to be smooth than a random number. Also $p-1$ is divisible by 3 with probability $1/2$, which is higher than for a random number. These observations fade into the background for large p's, but are more important for small p's. Thus smaller primes p have a higher chance of having $p-1$ be $p^{1/u}$-smooth than larger primes, for a fixed value of u. Thus we guess that $s(y)$ converges to its conjectured limit, as $y \to \infty$, from above, and our data above supports this.

In summary, Lemma 5.2.1 and Theorems 5.3.1 and 5.4.4 prove (5.5), and that concludes our results for Pollard's $p-1$ algorithm.

5.5 Remarks

We conclude with some remarks.

Let us examine a subtle point in the definition of $F(x, t, A)$. Notice that $F(x, t, A)/x$ gives the probability a randomly chosen integer below x can be factored by algorithm A with probability $> 1/2$ in t arithmetic operations. This is different from the probability a randomly chosen integer below x can be factored by algorithm A in t arithmetic operations; there may be many integers below x that algorithm A can factor with a very small positive probability, and $F(x, t, A)$ does not count these. However, if we alter our definition of $F(x, t, A)$ to denote the sum over integers $n \leq x$ of the probability that n can be factored by algorithm A in t steps, we can show the result is asymptotically the same for the $p-1$ algorithm.

Much of the work presented here on the $p-1$ algorithm generalizes to the higher cyclotomic factoring algorithms such as the $p+1$ method (see Williams [Wil82] and Bach and Shallit [BS89a]). Carl Pomerance and the author are currently working on this generalization and on computer calculations to determine whether Hypothesis 5.4.3 and similar heuristics are reasonable in practice.

Bibliography

[Adl79] L. M. Adleman. A subexponential algorithm for the discrete logarithm problem with applications to cryptography. In *20th Annual IEEE Symposium on Foundations of Computer Science*, pages 55–60, 1979.

[Adl91] L. M. Adleman. Factoring numbers using singular integers. In *23rd Annual ACM Symposium on Theory of Computing*, pages 64–71, 1991.

[AH87] L. M. Adleman and M.-D. Huang. Recognizing primes in random polynomial time. In *19th Annual ACM Symposium on Theory of Computing*, pages 462–469, 1987.

[AHU74] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[AK88] L. M. Adleman and K. Kompella. Using smoothness to achieve parallelism. In *20th Annual ACM Symposium on Theory of Computing*, pages 528–538, 1988.

[AM87] L. M. Adleman and K. S. McCurley. Open problems in number theoretic complexity. In D. S. Johnson, A. Nishizeki, A. Nozaki, and H. S. Wilf, editors, *Discrete Algorithms and Complexity: Proceedings of the Japan-US Joint Seminar.* Academic Press, Boston, 1987. Perspectives in Computing Series, volume 15.

[Ang82] D. Angluin. Lecture notes on the complexity of some problems in number theory. Technical Report 243, Department of Computer Science, Yale University, August 1982.

[Apo57] T. M. Apostol. *Mathematical Analysis: A Modern Approach to Advanced Calculus.* Addison-Wesley, Reading, Mass., 1957.

[APR83] L. M. Adleman, C. Pomerance, and R. Rumely. On distinguishing prime numbers from composite numbers. *Annals of Mathematics*, 117:173–206, 1983.

[Bac88] E. Bach. How to generate factored random numbers. *SIAM Journal on Computing*, 2:179–193, 1988.

[Bac90] E. Bach. Number-theoretic algorithms. *Annual Review of Computer Science*, 4:119–172, 1990.

[BCH86] P. W. Beame, S. A. Cook, and H. J. Hoover. Log depth circuits for division and related problems. *SIAM Journal on Computing*, 15:994–1003, 1986.

[BCP83] A. Borodin, S. Cook, and N. Pippenger. Parallel computation for well-endowed rings and space-bounded probabilistic machines. *Information and Control*, 58:113–136, 1983.

[BGM91] E. Bach, M. Giesbrecht, and J. McInnes. The complexity of number theoretic problems. Technical Report 247/91, Department of Computer Science, University of Toronto, January 1991.

[BK83] R. P. Brent and H. T. Kung. Systolic VLSI arrays for linear-time GCD computation. In F. Anceau and E. J. Aas, editors, *Proceedings of VLSI '83*, pages 145–154. Elsevier, 1983.

[BMS86] E. Bach, G. Miller, and J. Shallit. Sums of divisors, perfect numbers, and factoring. *SIAM Journal on Computing*, 4:1143–1154, 1986.

[Bor77] A. Borodin. On relating time and space to size and depth. *SIAM Journal on Computing*, 6(4):733–744, 1977.

[Bre76a] R. P. Brent. Analysis of the binary Euclidean algorithm. In J. F. Traub, editor, *Algorithms and Complexity*, pages 321–355. Academic Press, 1976.

[Bre76b] R. P. Brent. Multiple precision zero-finding methods and the complexity of elementary function evaluation. In J. F. Traub, editor, *Analytic Computational Complexity*, pages 151–176. Academic Press, 1976.

[BS89a] E. Bach and J. Shallit. Factoring with cyclotomic polynomials. *Mathematics of Computation*, 52(185):201–219, 1989.

[BS89b] E. Bach and J. Sorenson. Sieve algorithms for perfect power testing. Technical Report #852, Computer Sciences Department, University of Wisconsin-Madison, 1989. To appear in *Algorithmica*.

[BS90] E. Bach and J. Shallit. Algorithmic number theory. Manuscript, 1990.

[Bsh89] N. H. Bshouty. The Euclidean GCD algorithm is not optimal. Manuscript, 1989.

[BvzGH82] A. Borodin, J. von zur Gathen, and J. Hopcroft. Fast parallel matrix and GCD computations. *Information and Control*, 52:241–256, 1982.

[CEP83] E. R. Canfield, P. Erdős, and C. Pomerance. On a problem of Oppenheim concerning "Factorisatio Numerorum". *Journal of Number Theory*, 17:1–28, 1983.

[CFL85] A. K. Chandra, S. Fortune, and R. Lipton. Unbounded fan-in circuits and associative functions. *Journal of Computer and System Sciences*, 30, 1985.

[CG90] B. Chor and O. Goldreich. An improved parallel algorithm for integer GCD. *Algorithmica*, 5:1–10, 1990.

[Cob66] A. Cobham. The recognition problem for the set of perfect squares. In *Proceedings of the 7th Annual Symposium on Switching and Automata Theory*, pages 78–87, 1966.

[Col69] G. E. Collins. Computing multiplicative inverses in $GF(p)$. *Mathematics of Computation*, 23:197–200, 1969.

[Col74] G. E. Collins. The computing time of the Euclidean algorithm. *SIAM Journal on Computing*, 3(1):1–10, 1974.

[Coo85] S. A. Cook. A taxonomy of problems with fast parallel algorithms. *Information and Control*, 64:2–22, 1985.

[Cop84] D. Coppersmith. Fast evaluation of logarithms in fields of characteristic two. *IEEE Transactions on Information Theory*, 30:587–594, 1984.

[Dav80] H. Davenport. *Multiplicative Number Theory.* Springer-Verlag, New York, 1980.

[dB51] N. G. de Bruijn. The asymptotic behavior of a function occurring in the theory of primes. *Journal of the Indian Mathematical Society (New Series)*, 15:25–32, 1951.

[Den89] X. Deng. On the parallel complexity of integer programming. In *1st Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 110–116, 1989.

[Dix81] J. Dixon. Asymptotically fast factorization of integers. *Mathematics of Computation*, 36(153):255–260, 1981.

[Dix84] J. Dixon. Factorization and primality tests. *American Mathematical Monthly*, 6:333–352, 1984.

[EG85a] T. El Gamal. On computing logarithms over finite fields. In *Proceedings of Crypto '85*, pages 396–402, 1985.

[EG85b] T. El Gamal. A subexponential-time algorithm for computing discrete logarithms over $GF(p^2)$. *IEEE Transactions on Information Theory*, 31(4):473–481, 1985.

[ES80] P. Erdős and A. Sarközy. On the number of prime factors of integers. *Acta Scientiarium Mathematicarum*, 42:237–246, 1980.

[FT88] F. Fich and M. Tompa. The parallel complexity of exponentiating polynomials over finite fields. *Journal of the ACM*, 35(4):651–667, 1988.

[GHR91] R. Greenlaw, H. J. Hoover, and W. L. Ruzzo. A compendium of problems complete for P. Manuscript, March 1991.

[Gil77] J. Gill. Computational complexity of probabilistic Turing machines. *SIAM Journal on Computing*, 6:675–695, 1977.

[GJ79] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* Freeman, 1979.

[GK86] S. Goldwasser and J. Kilian. Almost all primes can be quickly certified. In *18th Annual ACM Symposium on Theory of Computing*, pages 316–329, 1986.

[GK90] D. H. Greene and D. E. Knuth. *Mathematics for the Analysis of Algorithms.* Birkhäuser, Boston, 3rd edition, 1990.

[Gol73] S. W. Golomb. A new arithmetic function of combinatorial significance. *Journal of Number Theory*, 5:218–223, 1973.

[Gor91] D. Gordon. Discrete logarithms using the number field sieve. Manuscript, 1991.

[Haf90] J. Hafner. On smooth numbers in short intervals under the Riemann hypothesis. Technical Report RJ 7728, IBM Research Division, October 1990.

[Hil86] A. Hildebrand. On the number of positive integers $\leq x$ and free of prime factors $> y$. *Journal of Number Theory*, 22:289–307, 1986.

[HM89] J. L. Hafner and K. S. McCurley. On the distribution of running times of certain integer factoring algorithms. *Journal of Algorithms*, 10(4):531–556, 1989.

[HR82] M. E. Hellman and J. M. Reyneri. Fast computation of discrete logarithms in $GF(q)$. In *Proceedings of Crypto '82*, pages 3–13. Plenum Press, 1982.

[HU79] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[HW79] G. H. Hardy and E. M. Wright. *An Introduction to the Theory of Numbers.* Oxford University Press, 5th edition, 1979.

[Ing32] A. E. Ingham. *The Distribution of Prime Numbers.* Cambridge Tract No. 30. Cambridge University Press, 1932.

[IR82] K. Ireland and M. Rosen. *A Classical Introduction to Modern Number Theory.* Springer-Verlag, New York, 1982.

[JR65] W.B. Jurkat and H.-E. Richert. An improvement of Selberg's sieve method I. *Acta Arithmetica*, 11:217–240, 1965.

[KMR87] R. Kannan, G. Miller, and L. Rudolph. Sublinear parallel algorithm for computing the greatest common divisor of two integers. *SIAM Journal on Computing*, 16(1):7–16, 1987.

[Knu81a] D. E. Knuth. *The Art of Computer Programming: Fundamental Algorithms*, volume 1. Addison-Wesley, Reading, Mass., 2nd edition, 1981.

[Knu81b] D. E. Knuth. *The Art of Computer Programming: Seminumerical Algorithms*, volume 2. Addison-Wesley, Reading, Mass., 2nd edition, 1981.

[Kob87] N. Koblitz. *A Course in Number Theory and Cryptography.* Springer-Verlag, New York, 1987.

[KR90] R. Karp and V. Ramachandran. Parallel algorithms for shared-memory machines. In J. van Leeuwen, editor, *Algorithms and Complexity.* Elsevier and MIT Press, 1990. Handbook of Theoretical Computer Science, volume A.

[KTP76] D. E. Knuth and L. Trabb Pardo. Analysis of a simple factorization algorithm. *Theoretical Computer Science*, 3:321–348, 1976.

[Lan71] S. Lang. *Algebra.* Addison-Wesley, 1971.

[Leh38] D. H. Lehmer. Euclid's algorithm for large numbers. *American Mathematical Monthly*, 45:227–233, 1938.

[Leh69] D. H. Lehmer. Computer technology applied to the theory of numbers. In W. J. LeVeque, editor, *Studies in Number Theory*, pages 117–151. Mathematic Association of America, 1969.

[Len87] H. W. Lenstra, Jr. Factoring integers with elliptic curves. *Annals of Mathematics*, 126:649–673, 1987.

[LL90] A. K. Lenstra and H. W. Lenstra, Jr. Algorithms in number theory. In J. van Leeuwen, editor, *Algorithms and Complexity.* Elsevier and MIT Press, 1990. Handbook of Theoretical Computer Science, volume A.

[LLMP90] A. K. Lenstra, H. W. Lenstra, Jr., M. S. Manasse, and J. M. Pollard. The number field sieve. In *22nd Annual ACM Symposium on Theory of Computing*, pages 564–572, 1990.

[LP91] H. W. Lenstra, Jr. and C. Pomerance. A rigorous time bound for factoring integers. Manuscript, 1991.

[Mil76] G. Miller. Riemann's hypothesis and tests for primality. *Journal of Computer and System Sciences*, 13:300–317, 1976.

[Mil89] V. Miller. Private communication, 1989.

[Mon87] P. L. Montgomery. Speeding the Pollard methods of factorization. *Mathematics of Computation*, 48(177):243–264, 1987.

[MS90] P. L. Montgomery and R. D. Silverman. An FFT extension to the $p-1$ factoring algorithm. *Mathematics of Computation*, 54, 1990.

[Mul87] K. Mulmuley. A fast parallel algorithm to compute the rank of a matrix over an arbitrary field. *Combinatorica*, 7(1):101–104, 1987.

[Nef90] C. A. Neff. Specified precision polynomial root isolation is in NC. In *31st Annual IEEE Symposium on Foundations of Computer Science*, pages 152–162, 1990.

[Nor85] G. Norton. Extending the binary GCD algorithm. In J. Calmet, editor, *Proceedings of the 3rd International Conference on Applied Algebra, Algebraic Algorithms and Error-Correcting Codes*, pages 363–372. Springer-Verlag, 1985. LNCS 229.

[Nor87] G. Norton. A shift-remainder GCD algorithm. In L. Huguet and A. Poli, editors, *Proceedings of the 5th International Conference on Applied Algebra, Algebraic Algorithms and Error-Correcting Codes*, pages 350–356. Springer-Verlag, 1987. LNCS 356.

[Nor90] G. Norton. On the asymptotic analysis of the Euclidean algorithm. *Journal of Symbolic Computation*, 10, 1990.

[Odl85] A. M. Odlyzko. Discrete logarithms in finite fields and their cryptographic significance. In *Proceedings of Eurocrypt '84*, Berlin, 1985. Springer-Verlag.

[Pan85] V. Pan. Fast and efficient algorithms for sequential and parallel evaluation of polynomial zeros and of matrix polynomials. In *26th Annual IEEE Symposium on Foundations of Computer Science*, pages 522–531, 1985.

[PB85] P. Purdom, Jr. and C. Brown. *The Analysis of Algorithms.* Holt, Rinehart, and Winston, 1985.

[Pol74] J. M. Pollard. Theorems on factorization and primality testing. *Proceedings of the Cambridge Philosophical Society*, 76:521–528, 1974.

[Pol75] J. M. Pollard. A Monte Carlo algorithm for factorization. *BIT*, 15:331–334, 1975.

[Pom80] C. Pomerance. Popular values of Euler's function. *Mathematika*, 27:84–89, 1980.

[Pom82] C. Pomerance. Analysis and comparison of some integer factoring algorithms. In H. W. Lenstra Jr. and R. Tijdeman, editors, *Computational Methods in Number Theory*, pages 89–141. Math. Centre, Amsterdam, 1982. Math. Centre Tract 154.

[Pom85] C. Pomerance. The quadratic sieve factoring algorithm. In *Proceedings of Eurocrypt '84*, pages 169–182, Berlin, 1985. Springer-Verlag.

[Pom87] C. Pomerance. Fast, rigorous factorization and discrete logarithm algorithms. In D. S. Johnson, A. Nishizeki, A. Nozaki, and H. S. Wilf, editors, *Discrete Algorithms and Complexity: Proceedings of the Japan-US Joint Seminar*, pages 119–143. Academic Press, Boston, 1987. Perspectives in Computing Series, volume 15.

[Pom89] C. Pomerance. On the composition of the arithmetic functions σ and ϕ. *Colloquium Mathematicum*, 58:11–15, 1989. Fasc. 1.

[PR85] V. Pan and J. Reif. Efficient parallel solution of linear systems. In *17th Annual ACM Symposium on Theory of Computing*, pages 143–152, 1985.

[Pri83] P. Pritchard. Fast compact prime number sieves (among others). *Journal of Algorithms*, 4:332–344, 1983.

[Pur83] G. B. Purdy. A carry-free algorithm for finding the greatest common divisor of two integers. *Computers & Mathematics with Applications*, 9(2):311–316, 1983.

[Rab80] M. O. Rabin. Probabilistic algorithm for testing primality. *Journal of Number Theory*, 12:128–138, 1980.

[RS62] J. B. Rosser and L. Schoenfeld. Approximate formulas for some functions of prime numbers. *Illinois Journal of Mathematics*, 6:64–94, 1962.

[RT89] J. H. Reif and S. R. Tate. Optimal size integer division circuits. In *21st Annual ACM Symposium on Theory of Computing*, pages 264–273, 1989.

[Sch71] A. Schönhage. Schnelle Berechnung von Kettenbruchentwicklungen. *Acta Informatica*, 1:139–144, 1971.

[Sha89] J. Shallit. Course notes for number theory and algorithms. Dartmouth College, 1989.

[Sho90] V. Shoup. On the deterministic complexity of factoring polynomials over finite fields. *Information Processing Letters*, 33, 1990.

[Sor89] J. Sorenson. Polylog depth circuits for integer factoring and discrete logarithms. Technical Report #872, Computer Sciences Department, University of Wisconsin-Madison, August 1989.

[Sor90a] J. Sorenson. Counting the integers cyclotomic methods can factor. Technical Report #919, Computer Sciences Department, University of Wisconsin-Madison, March 1990.

[Sor90b] J. Sorenson. An introduction to prime number sieves. Technical Report #909, Computer Sciences Department, University of Wisconsin-Madison, January 1990.

[Sor90c] J. Sorenson. The k-ary GCD algorithm. Technical Report #979, Computer Sciences Department, University of Wisconsin-Madison, November 1990.

[SS71] A. Schönhage and V. Strassen. Schnelle Multiplikation großer Zahlen. *Computing*, 7:281–292, 1971.

[SS77] R. Solovay and V. Strassen. A fast Monte Carlo test for primality. *SIAM Journal on Computing*, 6:84–85, 1977. Erratum in vol. 7, p. 118, 1978.

[Ste67] J. Stein. Computational problems associated with Racah algebra. *Journal of Computational Physics*, 1:397–405, 1967.

[Tit30] E. C. Titchmarsh. A divisor problem. *Rendiconti del Circolo Matematico di Palermo*, 54:414–429, 1930.

[Val91] B. Vallée. Generation of elements with small modular squares and provably fast integer factoring algorithms. *Mathematics of Computation*, 56(194):823–849, 1991.

[vzG87] J. von zur Gathen. Computing powers in parallel. *SIAM Journal on Computing*, 16:930–945, 1987.

[Wag79] S. S. Wagstaff, Jr. Greatest of the least primes in arithmetic progressions having a certain modulus. *Mathematics of Computation*, 33:1073–1080, 1979.

[Wil82] H. C. Williams. A $p+1$ method of factoring. *Mathematics of Computation*, 39(159):225–234, 1982.