A Survey of Techniques Used in Algebraic and Number Theoretic Algorithms

Manindra Agarwal

National University of Singapore
and
IIT Kanpur

Kunming Tutorial, May 2005

Overview

Introduction

Two Applications
- Coding Theory Application: Reed-Solomon Codes
- Cryptography Application: RSA Cryptosystem

Complexity of Basic Operations

Tools for Designing Algorithms for Basic Operations

Overview of the Tools

Outline

Introduction

Two Applications
- Coding Theory Application: Reed-Solomon Codes
- Cryptography Application: RSA Cryptosystem

Complexity of Basic Operations

Tools for Designing Algorithms for Basic Operations

Overview of the Tools

Algebraic Algorithms

- Algorithms for performing algebraic operations.
- Examples:
  - Matrix operations: addition, multiplication, inverse, determinant, solving a system of linear equations, ...
  - Polynomial operations: addition, multiplication, factoring, ...
  - Abstract algebra operations: order of a group element, discrete log, ...

Algebraic Algorithms

- Algorithms for performing algebraic operations.
- Examples:
  - Matrix operations: addition, multiplication, inverse, determinant, solving a system of linear equations, ...
  - Polynomial operations: addition, multiplication, factoring, ...
  - Abstract algebra operations: order of a group element, discrete log, ...

Number Theoretical Algorithms

- Algorithms for performing number theoretic operations.
- Examples:
  - Operations on integers and rationals: addition, multiplication, gcd, square roots, primality testing, integer factoring, ...

Number Theoretical Algorithms

- Algorithms for performing number theoretic operations.
- Examples:
  - Operations on integers and rationals: addition, multiplication, gcd, square roots, primality testing, integer factoring, ...

Applications

- In coding theory for efficient coding/decoding.
- In cryptography for design and analysis of cryptographic schemes.
- In computer algebra systems.

Applications

- In coding theory for efficient coding/decoding.
- In cryptography for design and analysis of cryptographic schemes.
- In computer algebra systems.

Applications

- In coding theory for efficient coding/decoding.
- In cryptography for design and analysis of cryptographic schemes.
- In computer algebra systems.

This Talk

- Discusses two major applications where algebraic and number theoretic algorithms are used.
- Surveys some of the important tools for designing these algorithms.
- Designs algorithms for some basic operations using these tools.

This Talk

- Discusses two major applications where algebraic and number theoretic algorithms are used.
- Surveys some of the important tools for designing these algorithms.
- Designs algorithms for some basic operations using these tools.

This Talk

- Discusses two major applications where algebraic and number theoretic algorithms are used.
- Surveys some of the important tools for designing these algorithms.
- Designs algorithms for some basic operations using these tools.

Outline

Introduction

Two Applications
Coding Theory Application: Reed-Solomon Codes
Cryptography Application: RSA Cryptosystem

Complexity of Basic Operations

Tools for Designing Algorithms for Basic Operations

Overview of the Tools

Outline

Introduction

Two Applications
Coding Theory Application: Reed-Solomon Codes
Cryptography Application: RSA Cryptosystem

Complexity of Basic Operations

Tools for Designing Algorithms for Basic Operations

Overview of the Tools

Reed-Soloman Codes

- One of the most important and popular class of codes.
- Used in several applications including encoding data on CDs and DVDs.
- Uses polynomial evaluations for coding, linear system solving and polynomial factorization for decoding.

Reed-Soloman Codes

- One of the most important and popular class of codes.
- Used in several applications including encoding data on CDs and DVDs.
- Uses polynomial evaluations for coding, linear system solving and polynomial factorization for decoding.

Reed-Soloman Codes: Coding

- Let m be a string that is to be coded.
- Fix a finite field F, $|F| \geq n$, and split m as a sequence of $k < n$ elements of F: $(m_0, \ldots, m_{k-1})$.
- Let polynomial $P_m(x) = \sum_{i=0}^{k-1} m_i \cdot x^i$.
- Let $c_j = P_m(e_j)$ for $0 \leq j < n$ with e_0, …, e_{n-1} distinct elements of F. [Requires polynomial evaluation]
- The sequence $(c_0, \ldots, c_{n-1})$ is the codeword corresponding to m.

Reed-Soloman Codes: Coding

- Let m be a string that is to be coded.
- Fix a finite field F, $|F| \geq n$, and split m as a sequence of $k < n$ elements of F: $(m_0, \ldots, m_{k-1})$.
- Let polynomial $P_m(x) = \sum_{i=0}^{k-1} m_i \cdot x^i$.
- Let $c_j = P_m(e_j)$ for $0 \leq j < n$ with e_0, ..., e_{n-1} distinct elements of F. [Requires polynomial evaluation]
- The sequence $(c_0, \ldots, c_{n-1})$ is the codeword corresponding to m.

Reed-Soloman Codes: Coding

- Let m be a string that is to be coded.
- Fix a finite field F, $|F| \geq n$, and split m as a sequence of $k < n$ elements of F: $(m_0, \ldots, m_{k-1})$.
- Let polynomial $P_m(x) = \sum_{i=0}^{k-1} m_i \cdot x^i$.
- Let $c_j = P_m(e_j)$ for $0 \leq j < n$ with e_0, ..., e_{n-1} distinct elements of F. [Requires polynomial evaluation]
- The sequence $(c_0, \ldots, c_{n-1})$ is the codeword corresponding to m.

Reed-Soloman Codes: Decoding

- Let $(d_0, \ldots, d_{n-1})$ be a given, possibly corrupted, codeword.
- Assume that the number of un-corrupted elements is at least t.
- Let $D_0 = \lceil \sqrt{kn} \rceil$ and $D_1 = \lfloor \sqrt{n/k} \rfloor$.
- Find a non-zero bivariate polynomial $Q(x, y)$ with x-degree D_0 and y-degree D_1 such that $Q(e_j, d_j) = 0$ for every $0 \leq j < n$.
- Such a Q can always be found since Q has $(1 + D_0) \cdot (1 + D_1) > n$ unknown coefficients that need to satisfy n homogeneous equations. [Requires solving a system of linear equations]

Reed-Soloman Codes: Decoding

- Let $(d_0, \ldots, d_{n-1})$ be a given, possibly corrupted, codeword.
- Assume that the number of un-corrupted elements is at least t.
- Let $D_0 = \lceil \sqrt{kn} \rceil$ and $D_1 = \lfloor \sqrt{n/k} \rfloor$.
- Find a non-zero bivariate polynomial $Q(x, y)$ with x-degree D_0 and y-degree D_1 such that $Q(e_j, d_j) = 0$ for every $0 \leq j < n$.
- Such a Q can always be found since Q has $(1 + D_0) \cdot (1 + D_1) > n$ unknown coefficients that need to satisfy n homogeneous equations. [Requires solving a system of linear equations]

Reed-Soloman Codes: Decoding

- Let $(d_0, \ldots, d_{n-1})$ be a given, possibly corrupted, codeword.
- Assume that the number of un-corrupted elements is at least t.
- Let $D_0 = \lceil \sqrt{kn} \rceil$ and $D_1 = \lfloor \sqrt{n/k} \rfloor$.
- Find a non-zero bivariate polynomial $Q(x, y)$ with x-degree D_0 and y-degree D_1 such that $Q(e_j, d_j) = 0$ for every $0 \leq j < n$.
- Such a Q can always be found since Q has $(1 + D_0) \cdot (1 + D_1) > n$ unknown coefficients that need to satisfy n homogeneous equations. [Requires solving a system of linear equations]

Reed-Soloman Codes: Decoding

- Consider the polynomial $\hat{Q}(x) = Q(x, P_m(x))$.
- We have $\hat{Q}(e_j) = 0$ for at least t different e_j's by assumption.
- The degree of $\hat{Q}(x)$ is less than $D_0 + D_1 \cdot k \leq 2\lceil\sqrt{kn}\rceil$.
- Therefore, if $t \geq 2\lceil\sqrt{kn}\rceil$, $\hat{Q}(x) = 0$.
- If $\hat{Q}(x) = Q(x, P_m(x)) = 0$, then polynomial $y - P_m(x)$ must divide polynomial $Q(x, y)$.
- Therefore, $y - P_m(x)$ divides $Q(x, y)$ whenever $t \geq 2\lceil\sqrt{kn}\rceil$.

Reed-Soloman Codes: Decoding

- Consider the polynomial $\hat{Q}(x) = Q(x, P_m(x))$.
- We have $\hat{Q}(e_j) = 0$ for at least t different e_j's by assumption.
- The degree of $\hat{Q}(x)$ is less than $D_0 + D_1 \cdot k \leq 2\lceil\sqrt{kn}\rceil$.
- Therefore, if $t \geq 2\lceil\sqrt{kn}\rceil$, $\hat{Q}(x) = 0$.
- If $\hat{Q}(x) = Q(x, P_m(x)) = 0$, then polynomial $y - P_m(x)$ must divide polynomial $Q(x, y)$.
- Therefore, $y - P_m(x)$ divides $Q(x, y)$ whenever $t \geq 2\lceil\sqrt{kn}\rceil$.

Reed-Soloman Codes: Decoding

- Consider the polynomial $\hat{Q}(x) = Q(x, P_m(x))$.
- We have $\hat{Q}(e_j) = 0$ for at least t different e_j's by assumption.
- The degree of $\hat{Q}(x)$ is less than $D_0 + D_1 \cdot k \leq 2\lceil\sqrt{kn}\rceil$.
- Therefore, if $t \geq 2\lceil\sqrt{kn}\rceil$, $\hat{Q}(x) = 0$.
- If $\hat{Q}(x) = Q(x, P_m(x)) = 0$, then polynomial $y - P_m(x)$ must divide polynomial $Q(x, y)$.
- Therefore, $y - P_m(x)$ divides $Q(x, y)$ whenever $t \geq 2\lceil\sqrt{kn}\rceil$.

Reed-Soloman Codes: Decoding

- Factor polynomial $Q(x, y)$ and list all the factors of the form $y - P(x)$. [Requires polynomial factoring]
- Select the polynomial $P(x)$ from these that agrees with the sequence $(d_0, \ldots, d_{n-1})$ on maximum number of elements.
- This is likely to be the polynomial $P_m(x)$.
- This algorithm decodes up to $n - 2\lceil\sqrt{kn}\rceil$ errors.
- Given by Madhu Sudan (1994).

Reed-Soloman Codes: Decoding

- Factor polynomial $Q(x, y)$ and list all the factors of the form $y - P(x)$. [Requires polynomial factoring]
- Select the polynomial $P(x)$ from these that agrees with the sequence $(d_0, \ldots, d_{n-1})$ on maximum number of elements.
- This is likely to be the polynomial $P_m(x)$.
- This algorithm decodes up to $n - 2\lceil\sqrt{kn}\rceil$ errors.
- Given by Madhu Sudan (1994).

Reed-Soloman Codes: Decoding

- Factor polynomial $Q(x, y)$ and list all the factors of the form $y - P(x)$. [Requires polynomial factoring]
- Select the polynomial $P(x)$ from these that agrees with the sequence $(d_0, \ldots, d_{n-1})$ on maximum number of elements.
- This is likely to be the polynomial $P_m(x)$.
- This algorithm decodes up to $n - 2\lceil\sqrt{kn}\rceil$ errors.
- Given by Madhu Sudan (1994).

Reed-Soloman Codes: Decoding

- Factor polynomial $Q(x, y)$ and list all the factors of the form $y - P(x)$. [Requires polynomial factoring]
- Select the polynomial $P(x)$ from these that agrees with the sequence $(d_0, \ldots, d_{n-1})$ on maximum number of elements.
- This is likely to be the polynomial $P_m(x)$.
- This algorithm decodes up to $n - 2\lceil\sqrt{kn}\rceil$ errors.
- Given by Madhu Sudan (1994).

Outline

Introduction

Two Applications

Coding Theory Application: Reed-Solomon Codes

Cryptography Application: RSA Cryptosystem

Complexity of Basic Operations

Tools for Designing Algorithms for Basic Operations

Overview of the Tools

RSA Cryptosystem

- The first and most popular public-key cryptosystem.
- Used in secure communication everywhere.
- Uses modular arithmetic for encryption and decryption.
- Uses primality testing for generating keys.
- Integer factoring dominates cryptanalysis, with modular equation solving also playing a role.

RSA Cryptosystem

- The first and most popular public-key cryptosystem.
- Used in secure communication everywhere.
- Uses modular arithmetic for encryption and decryption.
- Uses primality testing for generating keys.
- Integer factoring dominates cryptanalysis, with modular equation solving also playing a role.

RSA Cryptosystem

- The first and most popular public-key cryptosystem.
- Used in secure communication everywhere.
- Uses modular arithmetic for encryption and decryption.
- Uses primality testing for generating keys.
- Integer factoring dominates cryptanalysis, with modular equation solving also playing a role.

RSA: Key Generation

- Fix a key length, say, 2^r bits.
- Randomly select two primes p and q each of 2^{r-1} bits. [Requires primality testing]
- Randomly select an e, $3 \leq e < (p-1)(q-1)$ and $\gcd(e, (p-1)(q-1)) = 1$.
- Find the smallest d such that $d \cdot e = 1$ $(mod\ (p-1)(q-1))$. [Requires modular inverse computation]
- Let $n = pq$.
- The encryption key is the pair (n, e).
- The decryption key is d.

RSA: Key Generation

- Fix a key length, say, 2^r bits.
- Randomly select two primes p and q each of 2^{r-1} bits. [Requires primality testing]
- Randomly select an e, $3 \leq e < (p-1)(q-1)$ and $\gcd(e, (p-1)(q-1)) = 1$.
- Find the smallest d such that $d \cdot e = 1 \ (mod \ (p-1)(q-1))$. [Requires modular inverse computation]
- Let $n = pq$.
- The encryption key is the pair (n, e).
- The decryption key is d.

RSA: Key Generation

- Fix a key length, say, 2^r bits.
- Randomly select two primes p and q each of 2^{r-1} bits. [Requires primality testing]
- Randomly select an e, $3 \leq e < (p-1)(q-1)$ and $\gcd(e, (p-1)(q-1)) = 1$.
- Find the smallest d such that $d \cdot e = 1 \ (mod\ (p-1)(q-1))$. [Requires modular inverse computation]
- Let $n = pq$.
- The encryption key is the pair (n, e).
- The decryption key is d.

RSA: Key Generation

- Fix a key length, say, 2^r bits.
- Randomly select two primes p and q each of 2^{r-1} bits. [Requires primality testing]
- Randomly select an e, $3 \leq e < (p-1)(q-1)$ and $\gcd(e, (p-1)(q-1)) = 1$.
- Find the smallest d such that $d \cdot e = 1 \ (mod\ (p-1)(q-1))$. [Requires modular inverse computation]
- Let $n = pq$.
- The encryption key is the pair (n, e).
- The decryption key is d.

RSA: Key Generation

- Fix a key length, say, 2^r bits.
- Randomly select two primes p and q each of 2^{r-1} bits. [Requires primality testing]
- Randomly select an e, $3 \leq e < (p-1)(q-1)$ and $\gcd(e, (p-1)(q-1)) = 1$.
- Find the smallest d such that $d \cdot e = 1 \ (mod \ (p-1)(q-1))$. [Requires modular inverse computation]
- Let $n = pq$.
- The encryption key is the pair (n, e).
- The decryption key is d.

RSA: Encryption and Decryption

- Let m be the message to be encrypted.
- Treat m as a number less than n.
- Compute $c = m^e \ (mod\ n)$. [Requires modular exponentiation]
- c is the encrypted message.
- Note that $c^d \ (mod\ n) = m^{ed} \ (mod\ n) = m$.
- Thus c can be decrypted using key d.

RSA: Encryption and Decryption

- Let m be the message to be encrypted.
- Treat m as a number less than n.
- Compute $c = m^e \pmod{n}$. [Requires modular exponentiation]
- c is the encrypted message.
- Note that $c^d \pmod{n} = m^{ed} \pmod{n} = m$.
- Thus c can be decrypted using key d.

RSA: Encryption and Decryption

- Let m be the message to be encrypted.
- Treat m as a number less than n.
- Compute $c = m^e \ (mod\ n)$. [Requires modular exponentiation]
- c is the encrypted message.
- Note that $c^d \ (mod\ n) = m^{ed} \ (mod\ n) = m$.
- Thus c can be decrypted using key d.

RSA: Cryptanalysis

- If n can be factored, then d can be easily computed using e: $d = e^{-1} \ (mod \ (p-1)(q-1))$.
- So efficiency of factoring algorithms determines how safe RSA is.
- It is not the only way to break RSA though.
- We will see a different attack later that works for a special case.

RSA: Cryptanalysis

- If n can be factored, then d can be easily computed using e: $d = e^{-1} \; (mod\; (p-1)(q-1))$.
- So efficiency of factoring algorithms determines how safe RSA is.
- It is not the only way to break RSA though.
- We will see a different attack later that works for a special case.

RSA: Cryptanalysis

- If n can be factored, then d can be easily computed using e: $d = e^{-1} \ (mod \ (p-1)(q-1))$.
- So efficiency of factoring algorithms determines how safe RSA is.
- It is not the only way to break RSA though.
- We will see a different attack later that works for a special case.

Outline

Introduction

Two Applications
Coding Theory Application: Reed-Solomon Codes
Cryptography Application: RSA Cryptosystem

Complexity of Basic Operations

Tools for Designing Algorithms for Basic Operations

Overview of the Tools

Basic Operations: Polynomial Algebra

- Efficient algorithms are known for most of the operations.
  - Degree n Polynomial addition: $O(n)$ arithmetic operations.
  - Degree n Polynomial multiplication: $M_P(n) = O(n \log n)$ arithmetic operations.
- Several other operations reduce to polynomial multiplication:
  - Polynomial division: $O(M_P(n))$,
  - Polynomial gcd: $O(M_P(n) \log n)$.
  - Polynomial evaluation and interpolation: $O(M_P(n) \log n)$.

Basic Operations: Polynomial Algebra

- Efficient algorithms are known for most of the operations.
  - Degree n Polynomial addition: $O(n)$ arithmetic operations.
  - Degree n Polynomial multiplication: $M_P(n) = O(n \log n)$ arithmetic operations.
- Several other operations reduce to polynomial multiplication:
  - Polynomial division: $O(M_P(n))$,
  - Polynomial gcd: $O(M_P(n) \log n)$.
  - Polynomial evaluation and interpolation: $O(M_P(n) \log n)$.

Basic Operations: Polynomial Algebra

- Polynomial factorization over finite field F_p: $\tilde{O}(n^2 \log p)$ randomized.
  - $\tilde{O}(t(n)) = O(t(n) \cdot (\log t(n))^c)$ for some constant $c \geq 0$.
- Polynomial factorization over rationals: $\tilde{O}(n^{10} + n^8 \log^2 \|f\|_2)$, $\|f\|_2$ square-root of the sum of square of coefficients of f.

Basic Operations: Polynomial Algebra

- Polynomial factorization over finite field F_p: $\tilde{O}(n^2 \log p)$ randomized.
  - $\tilde{O}(t(n)) = O(t(n) \cdot (\log t(n))^c)$ for some constant $c \geq 0$.
- Polynomial factorization over rationals: $\tilde{O}(n^{10} + n^8 \log^2 \|f\|_2)$, $\|f\|_2$ square-root of the sum of square of coefficients of f.

Basic Operations: Arithmetic

- Very similar to polynomial algebra.
  - Addition: $O(n)$,
  - Multiplication: $M_I(n) = O(n \log n \log \log n)$,
  - Gcd: $O(n^2)$.
- A number of operations can be transformed to multiplication:
  - Division, Modular arithmetic, computing integer roots: $O(M_I(n))$.

Basic Operations: Arithmetic

- Very similar to polynomial algebra.
  - Addition: $O(n)$,
  - Multiplication: $M_I(n) = O(n \log n \log \log n)$,
  - Gcd: $O(n^2)$.
- A number of operations can be transformed to multiplication:
  - Division, Modular arithmetic, computing integer roots: $O(M_I(n))$.

Basic Operations: Arithmetic

- Primality testing: $\tilde{O}(n^6)$ deterministic, $\tilde{O}(n^2)$ randomized.
- Integer factoring:
  - $e^{O((\log n)^{1/2}(\log \log n)^{1/2})}$ randomized.
  - $e^{O((\log n)^{1/3}(\log \log n)^{2/3})}$ heuristic.

Basic Operations: Arithmetic

- Primality testing: $\tilde{O}(n^6)$ deterministic, $\tilde{O}(n^2)$ randomized.
- Integer factoring:
  - $e^{O((\log n)^{1/2}(\log \log n)^{1/2})}$ randomized.
  - $e^{O((\log n)^{1/3}(\log \log n)^{2/3})}$ heuristic.

Basic Operations: Linear Algebra

- The central problem is matrix multiplication.
- Coppersmith and Winograd (1986) showed that time complexity of multiplying two $n \times n$ matrices is $M_M(n) = O(n^{2.376})$ arithmetic operations.
- Several problems reduce to matrix multiplication:
  - Matrix inverse: $O(M_M(n))$,
  - Determinant, Characteristic polynomial: $O(M_M(n))$,
  - Solving a system of linear equations in n variables: $O(M_M(n))$.

Basic Operations: Linear Algebra

- The central problem is matrix multiplication.
- Coppersmith and Winograd (1986) showed that time complexity of multiplying two $n \times n$ matrices is $M_M(n) = O(n^{2.376})$ arithmetic operations.
- Several problems reduce to matrix multiplication:
  - Matrix inverse: $O(M_M(n))$,
  - Determinant, Characteristic polynomial: $O(M_M(n))$,
  - Solving a system of linear equations in n variables: $O(M_M(n))$.

Basic Operations: Abstract Algebra

- Computing order of an element in finite group G:
  - Complexity depends on the group.
  - Trivial for some groups, e.g., $(Z_n, +)$.
  - As hard as integer factoring for some groups, e.g., Z_n^*.
- Computing discrete log of an element in finite cyclic group G: given generator g for G, and element e, find m such that $e = g^m$.
  - Easy for some groups, e.g., $(Z_n, +)$. [requires modular inverse and multiplication]
  - Similar in hardness to integer factoring for groups, e.g., Z_p^*.
  - Very hard (time $= 2^{O(n)}$) for some groups, e.g., groups of points on elliptic curve E_p.

Basic Operations: Abstract Algebra

- Computing order of an element in finite group G:
  - Complexity depends on the group.
  - Trivial for some groups, e.g., $(Z_n, +)$.
  - As hard as integer factoring for some groups, e.g., Z_n^*.
- Computing discrete log of an element in finite cyclic group G: given generator g for G, and element e, find m such that $e = g^m$.
  - Easy for some groups, e.g., $(Z_n, +)$. [requires modular inverse and multiplication]
  - Similar in hardness to integer factoring for groups, e.g., Z_p^*.
  - Very hard (time $= 2^{O(n)}$) for some groups, e.g., groups of points on elliptic curve E_p.

Basic Operations: Abstract Algebra

- Computing order of an element in finite group G:
  - Complexity depends on the group.
  - Trivial for some groups, e.g., $(Z_n, +)$.
  - As hard as integer factoring for some groups, e.g., Z_n^*.
- Computing discrete log of an element in finite cyclic group G: given generator g for G, and element e, find m such that $e = g^m$.
  - Easy for some groups, e.g., $(Z_n, +)$. [requires modular inverse and multiplication]
  - Similar in hardness to integer factoring for groups, e.g., Z_p^*.
  - Very hard (time $= 2^{O(n)}$) for some groups, e.g., groups of points on elliptic curve E_p.

Outline

Introduction

Two Applications
Coding Theory Application: Reed-Solomon Codes
Cryptography Application: RSA Cryptosystem

Complexity of Basic Operations

Tools for Designing Algorithms for Basic Operations

Overview of the Tools

Tools for Designing Algorithms

1. Chinese Remaindering: Used in speeding integer and algebraic computations.
2. Discrete Fourier Transform: Used in polynomial and integer multiplication.
3. Automorphisms: Used in polynomial and integer factorization and irreducibility testing.
4. Hensel Lifting: Used in polynomial factorization and division.
5. Short Vectors in a Lattice: Used in polynomial factorization (over fields and rings) and breaking cryptosystems.
6. Smooth Numbers: Used in integer factorization and discrete log problem.

Tools for Designing Algorithms

1. Chinese Remaindering: Used in speeding integer and algebraic computations.
2. Discrete Fourier Transform: Used in polynomial and integer multiplication.
3. Automorphisms: Used in polynomial and integer factorization and irreducibility testing.
4. Hensel Lifting: Used in polynomial factorization and division.
5. Short Vectors in a Lattice: Used in polynomial factorization (over fields and rings) and breaking cryptosystems.
6. Smooth Numbers: Used in integer factorization and discrete log problem.

Tools for Designing Algorithms

1. Chinese Remaindering: Used in speeding integer and algebraic computations.
2. Discrete Fourier Transform: Used in polynomial and integer multiplication.
3. Automorphisms: Used in polynomial and integer factorization and irreducibility testing.
4. Hensel Lifting: Used in polynomial factorization and division.
5. Short Vectors in a Lattice: Used in polynomial factorization (over fields and rings) and breaking cryptosystems.
6. Smooth Numbers: Used in integer factorization and discrete log problem.

Tools for Designing Algorithms

1. Chinese Remaindering: Used in speeding integer and algebraic computations.
2. Discrete Fourier Transform: Used in polynomial and integer multiplication.
3. Automorphisms: Used in polynomial and integer factorization and irreducibility testing.
4. Hensel Lifting: Used in polynomial factorization and division.
5. Short Vectors in a Lattice: Used in polynomial factorization (over fields and rings) and breaking cryptosystems.
6. Smooth Numbers: Used in integer factorization and discrete log problem.

Tools for Designing Algorithms

1. Chinese Remaindering: Used in speeding integer and algebraic computations.
2. Discrete Fourier Transform: Used in polynomial and integer multiplication.
3. Automorphisms: Used in polynomial and integer factorization and irreducibility testing.
4. Hensel Lifting: Used in polynomial factorization and division.
5. Short Vectors in a Lattice: Used in polynomial factorization (over fields and rings) and breaking cryptosystems.
6. Smooth Numbers: Used in integer factorization and discrete log problem.

Tools for Designing Algorithms

1. Chinese Remaindering: Used in speeding integer and algebraic computations.
2. Discrete Fourier Transform: Used in polynomial and integer multiplication.
3. Automorphisms: Used in polynomial and integer factorization and irreducibility testing.
4. Hensel Lifting: Used in polynomial factorization and division.
5. Short Vectors in a Lattice: Used in polynomial factorization (over fields and rings) and breaking cryptosystems.
6. Smooth Numbers: Used in integer factorization and discrete log problem.

Outline

Introduction

Two Applications
- Coding Theory Application: Reed-Solomon Codes
- Cryptography Application: RSA Cryptosystem

Complexity of Basic Operations

Tools for Designing Algorithms for Basic Operations

Overview of the Tools

Chinese Remaindering

Definition

Example: Determinant Computation

Discrete Fourier Transform

Definition

Fast Fourier Transform

Example: Polynomial Multiplication

Automorphisms

Definition

Example: Polynomial Factoring over Finite Fields

Example: Primality Testing

Example: Integer Factoring

Hensel Lifting

Definition

Example: Polynomial Division

Short Vectors in a Lattice

Lattices and LLL Algorithm

Example: Solving Modular Equations

Example: Polynomial Factoring Over Rationals

Smooth Numbers

Definition

Example: Integer Factoring via Quadratic Sieve

Example: Discrete Log Computation via Index Calculus

Tool 1: Chinese Remaindering

Outline

Definition

Example: Determinant Computation

Chinese Remaindering Theorem

Theorem

Let $R = \mathbb{Z}$ or $F[x]$, and m_0, m_1, ..., $m_{r-1} \in R$ be pairwise coprime. Let $m = \prod_{i=0}^{r-1} m_i$. Then,

$$R/(m) \cong R/(m_0) \oplus R/(m_1) \oplus \cdots \oplus R/(m_{r-1}).$$

- An element of ring $R/(m)$ can be uniquely written as an r-tuple with ith component belonging to ring $R/(m_i)$.
- Addition and multiplication operations act component-wise.

CHINESE REMAINDERING THEOREM

THEOREM

Let $R = \mathbb{Z}$ *or* $F[x]$, *and* $m_0, m_1, \ldots, m_{r-1} \in R$ *be pairwise coprime. Let* $m = \prod_{i=0}^{r-1} m_i$. *Then,*

$$R/(m) \cong R/(m_0) \oplus R/(m_1) \oplus \cdots \oplus R/(m_{r-1}).$$

- An element of ring $R/(m)$ can be uniquely written as an r-tuple with ith component belonging to ring $R/(m_i)$.
- Addition and multiplication operations act component-wise.

Chinese Remaindering Applications

- Fundamental theorem used in arguing about rings everywhere.
- Used for speeding up computations over integers and polynomials.
- Based on the fact that it is much faster to compute modulo a small number (or small degree polynomial) than over integers (or polynomial ring):
  - Given a bound, say A, on the output of a computation, choose small $m_0, \ldots, m_{r-1}$ such that $\prod_{i=0}^{r-1} m_i > A$ and do the computations modulo each of m_i's.
  - At the end, combine the results of computations to get the desired result.
- Also lends itself to parallelization.

Chinese Remaindering Applications

- Fundamental theorem used in arguing about rings everywhere.
- Used for speeding up computations over integers and polynomials.
- Based on the fact that it is much faster to compute modulo a small number (or small degree polynomial) than over integers (or polynomial ring):
  - Given a bound, say A, on the output of a computation, choose small $m_0, \ldots, m_{r-1}$ such that $\prod_{i=0}^{r-1} m_i > A$ and do the computations modulo each of m_i's.
  - At the end, combine the results of computations to get the desired result.
- Also lends itself to parallelization.

Chinese Remaindering Applications

- Fundamental theorem used in arguing about rings everywhere.
- Used for speeding up computations over integers and polynomials.
- Based on the fact that it is much faster to compute modulo a small number (or small degree polynomial) than over integers (or polynomial ring):
    - Given a bound, say A, on the output of a computation, choose small m_0, …, m_{r-1} such that $\prod_{i=0}^{r-1} m_i > A$ and do the computations modulo each of m_i's.
    - At the end, combine the results of computations to get the desired result.
- Also lends itself to parallelization.

Chinese Remaindering Applications

- Fundamental theorem used in arguing about rings everywhere.
- Used for speeding up computations over integers and polynomials.
- Based on the fact that it is much faster to compute modulo a small number (or small degree polynomial) than over integers (or polynomial ring):
  - Given a bound, say A, on the output of a computation, choose small m_0, ..., m_{r-1} such that $\prod_{i=0}^{r-1} m_i > A$ and do the computations modulo each of m_i's.
  - At the end, combine the results of computations to get the desired result.
- Also lends itself to parallelization.

Outline

Definition

Example: Determinant Computation

Computing Determinant via CRT

- Let M be a $n \times n$ matrix over integers with A bounding the largest absolute value of its elements.
- Hadamard's inequality implies that $|\det M| \leq n^{n/2}A^n$.
- Let $B = n^{n/2}A^n$ and $r = \lceil \log(2B+1) \rceil$.
- Let $m_0, \ldots, m_{r-1}$ be first r primes and $m = \prod_{i=0}^{r-1} m_i$.
- Compute $v_i = \det M \ (mod\ m_i)$ for each i.
- Compute α_i such that $\alpha_i \cdot \frac{m}{m_i} = 1 \ (mod\ m_i)$ for each i.
- Output $\sum_{i=0}^{r-1} \alpha_i \cdot \frac{m}{m_i} \cdot v_i \ (mod\ m)$.

Computing Determinant via CRT

- Let M be a $n \times n$ matrix over integers with A bounding the largest absolute value of its elements.
- Hadamard's inequality implies that $|\det M| \leq n^{n/2}A^n$.
- Let $B = n^{n/2}A^n$ and $r = \lceil \log(2B+1) \rceil$.
- Let $m_0, \ldots, m_{r-1}$ be first r primes and $m = \prod_{i=0}^{r-1} m_i$.
- Compute $v_i = \det M \ (mod\ m_i)$ for each i.
- Compute α_i such that $\alpha_i \cdot \frac{m}{m_i} = 1 \ (mod\ m_i)$ for each i.
- Output $\sum_{i=0}^{r-1} \alpha_i \cdot \frac{m}{m_i} \cdot v_i \ (mod\ m)$.

Computing Determinant via CRT

- Let M be a $n \times n$ matrix over integers with A bounding the largest absolute value of its elements.
- Hadamard's inequality implies that $|\det M| \leq n^{n/2}A^n$.
- Let $B = n^{n/2}A^n$ and $r = \lceil \log(2B+1) \rceil$.
- Let $m_0, \ldots, m_{r-1}$ be first r primes and $m = \prod_{i=0}^{r-1} m_i$.
- Compute $v_i = \det M \ (mod\ m_i)$ for each i.
- Compute α_i such that $\alpha_i \cdot \frac{m}{m_i} = 1 \ (mod\ m_i)$ for each i.
- Output $\sum_{i=0}^{r-1} \alpha_i \cdot \frac{m}{m_i} \cdot v_i \ (mod\ m)$.

Computing Determinant via CRT

- Let M be a $n \times n$ matrix over integers with A bounding the largest absolute value of its elements.
- Hadamard's inequality implies that $|\det M| \leq n^{n/2} A^n$.
- Let $B = n^{n/2} A^n$ and $r = \lceil \log(2B+1) \rceil$.
- Let $m_0, \ldots, m_{r-1}$ be first r primes and $m = \prod_{i=0}^{r-1} m_i$.
- Compute $v_i = \det M \ (mod\ m_i)$ for each i.
- Compute α_i such that $\alpha_i \cdot \frac{m}{m_i} = 1 \ (mod\ m_i)$ for each i.
- Output $\sum_{i=0}^{r-1} \alpha_i \cdot \frac{m}{m_i} \cdot v_i \ (mod\ m)$.

Computing Determinant via CRT

- Let M be a $n \times n$ matrix over integers with A bounding the largest absolute value of its elements.
- Hadamard's inequality implies that $|\det M| \leq n^{n/2}A^n$.
- Let $B = n^{n/2}A^n$ and $r = \lceil \log(2B+1) \rceil$.
- Let $m_0, \ldots, m_{r-1}$ be first r primes and $m = \prod_{i=0}^{r-1} m_i$.
- Compute $v_i = \det M \ (mod\ m_i)$ for each i.
- Compute α_i such that $\alpha_i \cdot \frac{m}{m_i} = 1 \ (mod\ m_i)$ for each i.
- Output $\sum_{i=0}^{r-1} \alpha_i \cdot \frac{m}{m_i} \cdot v_i \ (mod\ m)$.

Tool 2: Discrete Fourier Transform

Outline

Definition

Fast Fourier Transform

Example: Polynomial Multiplication

Discrete Fourier Transform

- Discrete Fourier Transform is the discrete variant of Fourier transform.
- It is used in polynomial multiplication, integer multiplication, image compression, and many other applications.

Discrete Fourier Transform

- Discrete Fourier Transform is the discrete variant of Fourier transform.
- It is used in polynomial multiplication, integer multiplication, image compression, and many other applications.

Discrete Fourier Transform

- Let $f : [0, n-1] \mapsto F$ be a function 'selecting' n elements of field F.
- Let ω be a principle nth root of unity, i.e., $\omega^n = 1$, and $\omega^t \neq 1$ for $0 < t < n$.
- The DFT of f is $\mathcal{F}_f : [0, n-1] \mapsto F[\omega]$:

$$\mathcal{F}_f(j) = \sum_{i=0}^{n-1} f(i)\omega^{ij}.$$

Discrete Fourier Transform

- Let $f : [0, n-1] \mapsto F$ be a function ‘selecting’ n elements of field F.
- Let ω be a principle nth root of unity, i.e., $\omega^n = 1$, and $\omega^t \neq 1$ for $0 < t < n$.
- The DFT of f is $\mathcal{F}_f : [0, n-1] \mapsto F[\omega]$:

$$\mathcal{F}_f(j) = \sum_{i=0}^{n-1} f(i)\omega^{ij}.$$

Discrete Fourier Transform

- Let $f : [0, n-1] \mapsto F$ be a function 'selecting' n elements of field F.
- Let ω be a principle nth root of unity, i.e., $\omega^n = 1$, and $\omega^t \neq 1$ for $0 < t < n$.
- The DFT of f is $\mathcal{F}_f : [0, n-1] \mapsto F[\omega]$:

$$\mathcal{F}_f(j) = \sum_{i=0}^{n-1} f(i)\omega^{ij}.$$

Outline

Definition

Fast Fourier Transform

Example: Polynomial Multiplication

Fast Fourier Transform: An Algorithm for Computing DFT

- A straightforward algorithm takes $O(n^2)$ arithmetic operations.
- An $O(n \log n)$ time algorithm for DFT was (re)discovered by Cooley and Tukey (1965).
- It was first found by Gauss (1805).
- The algorithm is called Fast Fourier Transform and uses divide-and-conquer technique to recursively compute DFT.

Fast Fourier Transform: An Algorithm for Computing DFT

- A straightforward algorithm takes $O(n^2)$ arithmetic operations.
- An $O(n \log n)$ time algorithm for DFT was (re)discovered by Cooley and Tukey (1965).
- It was first found by Gauss (1805).
- The algorithm is called Fast Fourier Transform and uses divide-and-conquer technique to recursively compute DFT.

FFT

- Let f, $f : [0, n-1] \mapsto F$ for field field F, and assume $n = 2^k$.
- Note that for $0 \leq j < n/2$,

$$\mathcal{F}_f(2j) = \sum_{i=0}^{n-1} f(i)\omega^{2ij} = \sum_{i=0}^{n/2-1} (f(i) + f(n/2+i))(\omega^2)^{ij}.$$

- Similarly,

$$\mathcal{F}_f(2j+1) = \sum_{i=0}^{n-1} f(i)\omega^{i(2j+1)} = \sum_{i=0}^{n/2-1} (f(i)\omega^i - f(n/2+i)\omega^i)(\omega^2)^{ij}.$$

- Thus the problem reduces to computing DFT of two functions with $\frac{n}{2}$ domain size.

FFT

- Let f, $f : [0, n-1] \mapsto F$ for field field F, and assume $n = 2^k$.
- Note that for $0 \leq j < n/2$,

$$\mathcal{F}_f(2j) = \sum_{i=0}^{n-1} f(i)\omega^{2ij} = \sum_{i=0}^{n/2-1} (f(i) + f(n/2+i))(\omega^2)^{ij}.$$

- Similarly,

$$\mathcal{F}_f(2j+1) = \sum_{i=0}^{n-1} f(i)\omega^{i(2j+1)} = \sum_{i=0}^{n/2-1} (f(i)\omega^i - f(n/2+i)\omega^i)(\omega^2)^{ij}.$$

- Thus the problem reduces to computing DFT of two functions with $\frac{n}{2}$ domain size.

FFT

- Let f, $f : [0, n-1] \mapsto F$ for field field F, and assume $n = 2^k$.
- Note that for $0 \leq j < n/2$,

$$\mathcal{F}_f(2j) = \sum_{i=0}^{n-1} f(i)\omega^{2ij} = \sum_{i=0}^{n/2-1} (f(i) + f(n/2 + i))(\omega^2)^{ij}.$$

- Similarly,

$$\mathcal{F}_f(2j{+}1) = \sum_{i=0}^{n-1} f(i)\omega^{i(2j+1)} = \sum_{i=0}^{n/2-1} (f(i)\omega^i - f(n/2{+}i)\omega^i)(\omega^2)^{ij}.$$

- Thus the problem reduces to computing DFT of two functions with $\frac{n}{2}$ domain size.

FFT

- The functions are: $f_0(i) = f(i) + f(n/2 + i)$ and $f_1(i) = (f(i) - f(n/2 + i))\omega^i$ for $0 \leq i < n/2$.
- These functions can be computed using $O(n)$ operations from f.
- Setting the recurrence and solving, we get the time to compute DFT is $O(n \log n)$.

FFT

- The functions are: $f_0(i) = f(i) + f(n/2 + i)$ and $f_1(i) = (f(i) - f(n/2 + i))\omega^i$ for $0 \leq i < n/2$.
- These functions can be computed using $O(n)$ operations from f.
- Setting the recurrence and solving, we get the time to compute DFT is $O(n \log n)$.

FFT

- The functions are: $f_0(i) = f(i) + f(n/2 + i)$ and $f_1(i) = (f(i) - f(n/2 + i))\omega^i$ for $0 \leq i < n/2$.
- These functions can be computed using $O(n)$ operations from f.
- Setting the recurrence and solving, we get the time to compute DFT is $O(n \log n)$.

Outline

Definition

Fast Fourier Transform

Example: Polynomial Multiplication

Polynomial Multiplication via FFT

- Let P be a polynomial over field F of degree $< n$:

$$P(x) = \sum_{i=0}^{n-1} c_i x^i.$$

- Associate function $\hat{P}$ with P, $\hat{P} : [0, n-1] \mapsto F$, $\hat{P}(i) = c_i$.
- DFT of P is defined to be

$$\mathcal{F}_P(j) = \mathcal{F}_{\hat{P}}(j) = \sum_{i=0}^{n-1} c_i \omega^{ij} = P(\omega^j).$$

Polynomial Multiplication via FFT

- Let P be a polynomial over field F of degree $< n$:

$$P(x) = \sum_{i=0}^{n-1} c_i x^i.$$

- Associate function $\hat{P}$ with P, $\hat{P} : [0, n-1] \mapsto F$, $\hat{P}(i) = c_i$.
- DFT of P is defined to be

$$\mathcal{F}_P(j) = \mathcal{F}_{\hat{P}}(j) = \sum_{i=0}^{n-1} c_i \omega^{ij} = P(\omega^j).$$

Polynomial Multiplication via FFT

- Let P be a polynomial over field F of degree $< n$:

$$P(x) = \sum_{i=0}^{n-1} c_i x^i.$$

- Associate function $\hat{P}$ with P, $\hat{P} : [0, n-1] \mapsto F$, $\hat{P}(i) = c_i$.
- DFT of P is defined to be

$$\mathcal{F}_P(j) = \mathcal{F}_{\hat{P}}(j) = \sum_{i=0}^{n-1} c_i \omega^{ij} = P(\omega^j).$$

Polynomial Multiplication via FFT

Let P and Q be two polynomials of degree $< n = 2^k$.

1. Treat both P and Q as polynomials of degree $2n-1$ and compute their DFT, $\mathcal{F}_P$ and $\mathcal{F}_Q$.
2. Multiply $\mathcal{F}_P$ and $\mathcal{F}_Q$ component-wise.
3. Compute the inverse-DFT of resulting function by using the root ω^{-1} instead of ω.
4. The resulting polynomial is $P \cdot Q$.

The time complexity of each step is bounded by $O(n \log n)$.

Polynomial Multiplication via FFT

Let P and Q be two polynomials of degree $< n = 2^k$.

1. Treat both P and Q as polynomials of degree $2n - 1$ and compute their DFT, $\mathcal{F}_P$ and $\mathcal{F}_Q$.
2. Multiply $\mathcal{F}_P$ and $\mathcal{F}_Q$ component-wise.
3. Compute the inverse-DFT of resulting function by using the root ω^{-1} instead of ω.
4. The resulting polynomial is $P \cdot Q$.

The time complexity of each step is bounded by $O(n \log n)$.

Polynomial Multiplication via FFT

Let P and Q be two polynomials of degree $< n = 2^k$.

1. Treat both P and Q as polynomials of degree $2n - 1$ and compute their DFT, $\mathcal{F}_P$ and $\mathcal{F}_Q$.
2. Multiply $\mathcal{F}_P$ and $\mathcal{F}_Q$ component-wise.
3. Compute the inverse-DFT of resulting function by using the root ω^{-1} instead of ω.
4. The resulting polynomial is $P \cdot Q$.

The time complexity of each step is bounded by $O(n \log n)$.

Polynomial Multiplication via FFT

Let P and Q be two polynomials of degree $< n = 2^k$.

1. Treat both P and Q as polynomials of degree $2n-1$ and compute their DFT, $\mathcal{F}_P$ and $\mathcal{F}_Q$.
2. Multiply $\mathcal{F}_P$ and $\mathcal{F}_Q$ component-wise.
3. Compute the inverse-DFT of resulting function by using the root ω^{-1} instead of ω.
4. The resulting polynomial is $P \cdot Q$.

The time complexity of each step is bounded by $O(n \log n)$.

Polynomial Multiplication via FFT

Let P and Q be two polynomials of degree $< n = 2^k$.

1. Treat both P and Q as polynomials of degree $2n - 1$ and compute their DFT, $\mathcal{F}_P$ and $\mathcal{F}_Q$.
2. Multiply $\mathcal{F}_P$ and $\mathcal{F}_Q$ component-wise.
3. Compute the inverse-DFT of resulting function by using the root ω^{-1} instead of ω.
4. The resulting polynomial is $P \cdot Q$.

The time complexity of each step is bounded by $O(n \log n)$.

Tool 3: Automorphisms

Outline

Definition

Example: Polynomial Factoring over Finite Fields

Example: Primality Testing

Example: Integer Factoring

Definition

- Automorphism of an algebraic structure is a mapping of the structure to itself that preserves all the operations.
- Automorphisms of finite rings and fields play a crucial role in polynomial factoring and primality testing.

Definition

- Let $R = Z_n[X]/(f(X))$ be a finite ring, f a polynomial of degree d.
- An automorphism ϕ of R preserves both addition and multiplication in the ring.
- It is easy to see that ϕ is completely specified by its action on X: for any element $e(X) \in R$, $\phi(e(X)) = e(\phi(X))$.
- In addition, $\phi(f(X)) = f(\phi(X)) = 0$ in the ring.

Definition

- Let $R = Z_n[X]/(f(X))$ be a finite ring, f a polynomial of degree d.
- An automorphism ϕ of R preserves both addition and multiplication in the ring.
- It is easy to see that ϕ is completely specified by its action on X: for any element $e(X) \in R$, $\phi(e(X)) = e(\phi(X))$.
- In addition, $\phi(f(X)) = f(\phi(X)) = 0$ in the ring.

Definition

- Let $R = Z_n[X]/(f(X))$ be a finite ring, f a polynomial of degree d.
- An automorphism ϕ of R preserves both addition and multiplication in the ring.
- It is easy to see that ϕ is completely specified by its action on X: for any element $e(X) \in R$, $\phi(e(X)) = e(\phi(X))$.
- In addition, $\phi(f(X)) = f(\phi(X)) = 0$ in the ring.

Definition

- If R is a field, i.e., n is prime and f is irreducible over F_p, then the automorphisms of R are precisely ψ, ψ^2, ..., $\psi^d = id$ where $\psi(X) = X^p$.
- In general, R is a direct sum of fields (by CRT) and its automorphisms are compositions of automorphisms of fields in the sum.

Definition

- If R is a field, i.e., n is prime and f is irreducible over F_p, then the automorphisms of R are precisely ψ, ψ^2, . . ., $\psi^d = id$ where $\psi(X) = X^p$.
- In general, R is a direct sum of fields (by CRT) and its automorphisms are compositions of automorphisms of fields in the sum.

Outline

Definition

Example: Polynomial Factoring over Finite Fields

Example: Primality Testing

Example: Integer Factoring

Polynomial Factoring Over Finite Fields

- The algorithms developed by Berlekemp and others (1980s).
- Let f be a degree n monic polynomial over finite field F_p.
- We wish to compute all irreducible factors of f.
- If f is not square-free, i.e., g^2 divides f for some g, then f can be factored easily:
  - Compute $\gcd(f, \frac{df}{dx})$.
  - Since g divides both f and $\frac{df}{dx}$, the gcd will be non-trivial.

Polynomial Factoring Over Finite Fields

- The algorithms developed by Berlekemp and others (1980s).
- Let f be a degree n monic polynomial over finite field F_p.
- We wish to compute all irreducible factors of f.
- If f is not square-free, i.e., g^2 divides f for some g, then f can be factored easily:
    - Compute $\gcd(f, \frac{df}{dx})$.
    - Since g divides both f and $\frac{df}{dx}$, the gcd will be non-trivial.

Polynomial Factoring Over Finite Fields

- The algorithms developed by Berlekemp and others (1980s).
- Let f be a degree n monic polynomial over finite field F_p.
- We wish to compute all irreducible factors of f.
- If f is not square-free, i.e., g^2 divides f for some g, then f can be factored easily:
  - Compute $\gcd(f, \frac{df}{dx})$.
  - Since g divides both f and $\frac{df}{dx}$, the gcd will be non-trivial.

Polynomial Factoring Over Finite Fields

- The algorithms developed by Berlekemp and others (1980s).
- Let f be a degree n monic polynomial over finite field F_p.
- We wish to compute all irreducible factors of f.
- If f is not square-free, i.e., g^2 divides f for some g, then f can be factored easily:
    - Compute $\gcd(f, \frac{df}{dx})$.
    - Since g divides both f and $\frac{df}{dx}$, the gcd will be non-trivial.

Polynomial Factoring Over Finite Fields

- We now assume that f is square-free.
- Let $f = \prod_{i=1}^{t} f_i$, each f_i is irreducible and has degree d_i.
- Let $d_1 \leq d_2 \leq \cdots \leq d_t$.
- Consider ring $R = F_p[X]/(f) = \oplus_{i=1}^{t} F_p[X]/(f_i)$. [by CRT]
- Clearly, ψ^{d_1} is trivial in $F_p[X]/(f_1)$ but not in $F_p[X]/(f_j)$ when $d_j > d_1$.

Polynomial Factoring Over Finite Fields

- We now assume that f is square-free.
- Let $f = \prod_{i=1}^{t} f_i$, each f_i is irreducible and has degree d_i.
- Let $d_1 \leq d_2 \leq \cdots \leq d_t$.
- Consider ring $R = F_p[X]/(f) = \oplus_{i=1}^{t} F_p[X]/(f_i)$. [by CRT]
- Clearly, ψ^{d_1} is trivial in $F_p[X]/(f_1)$ but not in $F_p[X]/(f_j)$ when $d_j > d_1$.

Polynomial Factoring Over Finite Fields

- Therefore, $X^{p^{d_1}} = X$ in $F_p[X]/(f_1)$ but not in $F_p[X]/(f_j)$.
- So f_1 divides $\gcd(X^{p^{d_1}} - X, f(X))$ but not f_j.
- Computing $\gcd(X^{p^d} - X, f(X))$ starting from $d = 1$ to $d = n/2$ will factor f into equal degree factors.
- That is, each factor we get is a product of all the f_j's of the same degree.
- This also allows us to test if f is irreducible: all the gcds are 1 iff f is irreducible.

Polynomial Factoring Over Finite Fields

- Therefore, $X^{p^{d_1}} = X$ in $F_p[X]/(f_1)$ but not in $F_p[X]/(f_j)$.
- So f_1 divides $\gcd(X^{p^{d_1}} - X, f(X))$ but not f_j.
- Computing $\gcd(X^{p^d} - X, f(X))$ starting from $d = 1$ to $d = n/2$ will factor f into equal degree factors.
- That is, each factor we get is a product of all the f_j's of the same degree.
- This also allows us to test if f is irreducible: all the gcds are 1 iff f is irreducible.

Polynomial Factoring Over Finite Fields

- Therefore, $X^{p^{d_1}} = X$ in $F_p[X]/(f_1)$ but not in $F_p[X]/(f_j)$.
- So f_1 divides $\gcd(X^{p^{d_1}} - X, f(X))$ but not f_j.
- Computing $\gcd(X^{p^d} - X, f(X))$ starting from $d = 1$ to $d = n/2$ will factor f into equal degree factors.
- That is, each factor we get is a product of all the f_j's of the same degree.
- This also allows us to test if f is irreducible: all the gcds are 1 iff f is irreducible.

Polynomial Factoring Over Finite Fields

- Now suppose f is such that $d_1 = d_2 = \cdots = d_t$.
- Then the above method does not give any factor of f.
- To handle this, we convert the problem to finding roots of a polynomial in F_p.
- Let
$$S = \{e(X) \in R \mid \psi(e(X)) = e(X^p) = e(X)\}.$$
- S is a subring of R, $S = \oplus_{i=1}^{t} F_p$.
- S can be computed using linear algebra.

Polynomial Factoring Over Finite Fields

- Now suppose f is such that $d_1 = d_2 = \cdots = d_t$.
- Then the above method does not give any factor of f.
- To handle this, we convert the problem to finding roots of a polynomial in F_p.
- Let

$$S = \{e(X) \in R \mid \psi(e(X)) = e(X^p) = e(X)\}.$$

- S is a subring of R, $S = \oplus_{i=1}^{t} F_p$.
- S can be computed using linear algebra.

Polynomial Factoring Over Finite Fields

- Choose $e(X) \in S - F_p$.
- We must have $e(X)\ (mod\ f_i(X)) = c_i \in F_p$ for each i.
- Since $e(X) \notin F_p$, there exists i and j such that $c_i \neq c_j$.
- Therefore, $\gcd(e(X) - c_i, f(X))$ is divisible by f_i but not by f_j.
- Thus we get a factor of f.
- How do we compute a c_i?

Polynomial Factoring Over Finite Fields

- Choose $e(X) \in S - F_p$.
- We must have $e(X)\ (mod\ f_i(X)) = c_i \in F_p$ for each i.
- Since $e(X) \notin F_p$, there exists i and j such that $c_i \neq c_j$.
- Therefore, $\gcd(e(X) - c_i, f(X))$ is divisible by f_i but not by f_j.
- Thus we get a factor of f.
- How do we compute a c_i?

Polynomial Factoring Over Finite Fields

- Choose $e(X) \in S - F_p$.
- We must have $e(X)\ (mod\ f_i(X)) = c_i \in F_p$ for each i.
- Since $e(X) \notin F_p$, there exists i and j such that $c_i \neq c_j$.
- Therefore, $\gcd(e(X) - c_i, f(X))$ is divisible by f_i but not by f_j.
- Thus we get a factor of f.
- How do we compute a c_i?

Polynomial Factoring Over Finite Fields

- Let $g(y) = \mathrm{Res}(e(X) - y, f(X))$.
- Res is the resultant of two polynomials.
- For any $c \in F_p$, we have $g(c) = 0$ iff $\gcd(e(X) - c, f(X))$ is non-trivial giving a factor of f.
- So, if we can find roots of g in F_p, we can factor f!

Polynomial Factoring Over Finite Fields

- Let $g(y) = \text{Res}(e(X) - y, f(X))$.
- Res is the resultant of two polynomials.
- For any $c \in F_p$, we have $g(c) = 0$ iff $\gcd(e(X) - c, f(X))$ is non-trivial giving a factor of f.
- So, if we can find roots of g in F_p, we can factor f!

Polynomial Factoring Over Finite Fields

- Let $g(y) = \mathrm{Res}(e(X) - y, f(X))$.
- Res is the resultant of two polynomials.
- For any $c \in F_p$, we have $g(c) = 0$ iff $\gcd(e(X) - c, f(X))$ is non-trivial giving a factor of f.
- So, if we can find roots of g in F_p, we can factor f!

Polynomial Factoring Over Finite Fields

- Compute $\hat{g}(y) = \gcd(g(y), \psi(y) - y)$.
  - $\hat{g}$ factors completely in F_p and its roots are roots of g in F_p.
- Let $\hat{g}(y) = \prod_{i=0}^{k}(y - c_i)$.
- Compute $h(y) = \hat{g}(y^2 - r)$ for a randomly chosen $r \in F_p$.
- So, $h(y) = \prod_{i=0}^{k}(y^2 - (c_i + r))$.
- $y^2 - (c_i + r)$ factors over F_p iff $c_i + r$ is a quadratic residue.

Polynomial Factoring Over Finite Fields

- Compute $\hat{g}(y) = \gcd(g(y), \psi(y) - y)$.
  - $\hat{g}$ factors completely in F_p and its roots are roots of g in F_p.
- Let $\hat{g}(y) = \prod_{i=0}^{k}(y - c_i)$.
- Compute $h(y) = \hat{g}(y^2 - r)$ for a randomly chosen $r \in F_p$.
- So, $h(y) = \prod_{i=0}^{k}(y^2 - (c_i + r))$.
- $y^2 - (c_i + r)$ factors over F_p iff $c_i + r$ is a quadratic residue.

Polynomial Factoring Over Finite Fields

- Compute $\hat{g}(y) = \gcd(g(y), \psi(y) - y)$.
  - $\hat{g}$ factors completely in F_p and its roots are roots of g in F_p.
- Let $\hat{g}(y) = \prod_{i=0}^{k}(y - c_i)$.
- Compute $h(y) = \hat{g}(y^2 - r)$ for a randomly chosen $r \in F_p$.
- So, $h(y) = \prod_{i=0}^{k}(y^2 - (c_i + r))$.
- $y^2 - (c_i + r)$ factors over F_p iff $c_i + r$ is a quadratic residue.

Polynomial Factoring Over Finite Fields

- For any i and j, $i \neq j$, the probability that exactly one of $c_i + r$ and $c_j + r$ is a quadratic residue in F_p, is at least $\frac{1}{2}$.
- Therefore, using the equal degree factorization algorithm above factors $h(y)$ with probability at least $\frac{1}{2}$.
- Let $h(y) = h_1(y) \cdot h_2(y)$.
- Both h_1 and h_2 will have only even powers of y.
- Then, $g(y) = h(\sqrt{y} + r) = h_1(\sqrt{y} + r) \cdot h_2(\sqrt{y} + r)$.
- Iterate this to completely factor g.

Polynomial Factoring Over Finite Fields

- For any i and j, $i \neq j$, the probability that exactly one of $c_i + r$ and $c_j + r$ is a quadratic residue in F_p, is at least $\frac{1}{2}$.
- Therefore, using the equal degree factorization algorithm above factors $h(y)$ with probability at least $\frac{1}{2}$.
- Let $h(y) = h_1(y) \cdot h_2(y)$.
- Both h_1 and h_2 will have only even powers of y.
- Then, $g(y) = h(\sqrt{y} + r) = h_1(\sqrt{y} + r) \cdot h_2(\sqrt{y} + r)$.
- Iterate this to completely factor g.

Polynomial Factoring Over Finite Fields

- For any i and j, $i \neq j$, the probability that exactly one of $c_i + r$ and $c_j + r$ is a quadratic residue in F_p, is at least $\frac{1}{2}$.
- Therefore, using the equal degree factorization algorithm above factors $h(y)$ with probability at least $\frac{1}{2}$.
- Let $h(y) = h_1(y) \cdot h_2(y)$.
- Both h_1 and h_2 will have only even powers of y.
- Then, $g(y) = h(\sqrt{y} + r) = h_1(\sqrt{y} + r) \cdot h_2(\sqrt{y} + r)$.
- Iterate this to completely factor g.

Outline

Definition

Example: Polynomial Factoring over Finite Fields

Example: Primality Testing

Example: Integer Factoring

Primality Testing

- Fermat's Little Theorem states that if n is prime then for every a: $a^n = a \ (mod\ n)$.
- In other words: mapping $\phi(x) = x^n$ is the trivial automorphism of the ring Z_n.
- The converse of the statement is not true: there are composite n such that ϕ is the trivial automorphism of Z_n.
- Even if it were true, checking if ϕ is the trivial automorphism requires $\Omega(n)$ steps.
- So the theorem cannot be used for testing primality efficiently.

Primality Testing

- Fermat's Little Theorem states that if n is prime then for every a: $a^n = a \ (mod\ n)$.
- In other words: mapping $\phi(x) = x^n$ is the trivial automorphism of the ring Z_n.
- The converse of the statement is not true: there are composite n such that ϕ is the trivial automorphism of Z_n.
- Even if it were true, checking if ϕ is the trivial automorphism requires $\Omega(n)$ steps.
- So the theorem cannot be used for testing primality efficiently.

Primality Testing

- Fermat's Little Theorem states that if n is prime then for every a: $a^n = a \ (mod\ n)$.
- In other words: mapping $\phi(x) = x^n$ is the trivial automorphism of the ring Z_n.
- The converse of the statement is not true: there are composite n such that ϕ is the trivial automorphism of Z_n.
- Even if it were true, checking if ϕ is the trivial automorphism requires $\Omega(n)$ steps.
- So the theorem cannot be used for testing primality efficiently.

Primality Testing

- Both the problems can be eliminated using a generalization of the theorem.
- This was shown by A, Kayal and Saxena (2004) who obtained a deterministic $\tilde{O}(n^{15/2})$ algorithm for primality testing.
- Earlier, there were algorithms known for primality testing but they were either randomized or not polynomial-time.

Primality Testing

- Both the problems can be eliminated using a generalization of the theorem.
- This was shown by A, Kayal and Saxena (2004) who obtained a deterministic $\tilde{O}(n^{15/2})$ algorithm for primality testing.
- Earlier, there were algorithms known for primality testing but they were either randomized or not polynomial-time.

Primality Testing

- Fix $r > 0$ such that $O_r(n) > 4\log^2 n$ ($O_r(n)$ is order of n modulo r).
  - It is easy to see that such an r exists in $[4\log^2 n, 16\log^5 n]$.
- Let ring $R = Z_n[X]/(X^{2r} - X^r)$.
- Clearly we have:

Theorem (Generalized FLT)

If n is prime then ϕ is an automorphism of R.

Primality Testing

- Fix $r > 0$ such that $O_r(n) > 4\log^2 n$ ($O_r(n)$ is order of n modulo r).
    - It is easy to see that such an r exists in $[4\log^2 n, 16\log^5 n]$.
- Let ring $R = Z_n[X]/(X^{2r} - X^r)$.
- Clearly we have:

Theorem (Generalized FLT)

If n is prime then ϕ is an automorphism of R.

Primality Testing

- Does the converse also hold?
- Yes, it does!

Theorem (AKS, 2004)

If ϕ is an automorphism of R then n is prime.

Primality Testing

- Does the converse also hold?
- Yes, it does!

Theorem (AKS, 2004)

If ϕ is an automorphism of R then n is prime.

Primality Testing

- What about efficiency?
- Testing that ϕ is an automorphism naively requires exponential time.
- This can be eliminated too:

Theorem (AKS, 2004)

ϕ is an automorphism of R iff $\phi(X + a) = \phi(X) + a$ in R for $1 \leq a \leq 2\sqrt{r}\log n$.

Primality Testing

- What about efficiency?
- Testing that ϕ is an automorphism naively requires exponential time.
- This can be eliminated too:

Theorem (AKS, 2004)

ϕ is an automorphism of R iff $\phi(X+a) = \phi(X) + a$ in R for $1 \leq a \leq 2\sqrt{r}\log n$.

Primality Testing

- What about efficiency?
- Testing that ϕ is an automorphism naively requires exponential time.
- This can be eliminated too:

Theorem (AKS, 2004)

ϕ is an automorphism of R iff $\phi(X + a) = \phi(X) + a$ in R for $1 \leq a \leq 2\sqrt{r}\log n$.

Primality Testing

- Since $r = O(\log^5 n)$, testing if $\phi(X + a) = \phi(X) + a$ takes time $\tilde{O}(\log^7 n)$.
- So total time taken is $\tilde{O}(\log^7 n \cdot \log^{7/2} n) = \tilde{O}(\log^{21/2} n)$.
- Using an analytic number theory result by Fouvry (1985), it can be shown that $r = O(\log^3 n)$.
- This brings down time complexity to $\tilde{O}(\log^{15/2} n)$.
- Lenstra and Pomerance (2003) further bring it down to $\tilde{O}(\log^6 n)$.

Primality Testing

- Since $r = O(\log^5 n)$, testing if $\phi(X + a) = \phi(X) + a$ takes time $\tilde{O}(\log^7 n)$.
- So total time taken is $\tilde{O}(\log^7 n \cdot \log^{7/2} n) = \tilde{O}(\log^{21/2} n)$.
- Using an analytic number theory result by Fouvry (1985), it can be shown that $r = O(\log^3 n)$.
- This brings down time complexity to $\tilde{O}(\log^{15/2} n)$.
- Lenstra and Pomerance (2003) further bring it down to $\tilde{O}(\log^6 n)$.

Primality Testing

- Since $r = O(\log^5 n)$, testing if $\phi(X + a) = \phi(X) + a$ takes time $\tilde{O}(\log^7 n)$.
- So total time taken is $\tilde{O}(\log^7 n \cdot \log^{7/2} n) = \tilde{O}(\log^{21/2} n)$.
- Using an analytic number theory result by Fouvry (1985), it can be shown that $r = O(\log^3 n)$.
- This brings down time complexity to $\tilde{O}(\log^{15/2} n)$.
- Lenstra and Pomerance (2003) further bring it down to $\tilde{O}(\log^6 n)$.

Primality Testing

- Since $r = O(\log^5 n)$, testing if $\phi(X + a) = \phi(X) + a$ takes time $\tilde{O}(\log^7 n)$.
- So total time taken is $\tilde{O}(\log^7 n \cdot \log^{7/2} n) = \tilde{O}(\log^{21/2} n)$.
- Using an analytic number theory result by Fouvry (1985), it can be shown that $r = O(\log^3 n)$.
- This brings down time complexity to $\tilde{O}(\log^{15/2} n)$.
- Lenstra and Pomerance (2003) further bring it down to $\tilde{O}(\log^6 n)$.

Outline

Definition

Example: Polynomial Factoring over Finite Fields

Example: Primality Testing

Example: Integer Factoring

Integer Factoring

- Kayal and Saxena (2004) show that integer factoring reduces to several questions about automorphisms of rings.
- They show n can be factored if
  - A non-trivial automorphism of ring $Z_n[X]/(X^2-1)$ can be computed.
  - The number of automorphisms of ring $Z_n[X]/(X^2)$ can be computed.

Integer Factoring

- Kayal and Saxena (2004) show that integer factoring reduces to several questions about automorphisms of rings.
- They show n can be factored if
  - A non-trivial automorphism of ring $Z_n[X]/(X^2 - 1)$ can be computed.
  - The number of automorphisms of ring $Z_n[X]/(X^2)$ can be computed.

Integer Factoring

- Kayal and Saxena (2004) show that integer factoring reduces to several questions about automorphisms of rings.
- They show n can be factored if
  - A non-trivial automorphism of ring $Z_n[X]/(X^2 - 1)$ can be computed.
  - The number of automorphisms of ring $Z_n[X]/(X^2)$ can be computed.

Integer Factoring

Theorem (Kayal and Saxena, 2004)

An odd number n can be factored efficiently iff a non-trivial automorphism of ring $Z_n[X]/(X^2 - 1)$ can be computed efficiently.

Integer Factoring

Proof.

- First observe that n can be factored iff a non-trivial solution of $y^2 - 1 \ (mod\ n)$ can be found in Z_n:
  - If $y_0 \neq \pm 1 \ (mod\ n)$ is a non-trivial solution, then $\gcd(y_0 + 1, n)$ gives a factor.
  - If $n = n_1 n_2$, then a $y_0 < n$ with $y_0 = 1 \ (mod\ n_1)$ and $y_0 = -1 \ (mod\ n_2)$ exists (by CRT) and is therefore a non-trivial solution.

Integer Factoring

Proof.

- First observe that n can be factored iff a non-trivial solution of $y^2 - 1 \ (mod\ n)$ can be found in Z_n:
  - If $y_0 \neq \pm 1 \ (mod\ n)$ is a non-trivial solution, then $\gcd(y_0 + 1, n)$ gives a factor.
  - If $n = n_1 n_2$, then a $y_0 < n$ with $y_0 = 1 \ (mod\ n_1)$ and $y_0 = -1 \ (mod\ n_2)$ exists (by CRT) and is therefore a non-trivial solution.

Integer Factoring

Proof.

- First observe that n can be factored iff a non-trivial solution of $y^2 - 1 \ (mod\ n)$ can be found in Z_n:
  - If $y_0 \neq \pm 1 \ (mod\ n)$ is a non-trivial solution, then $\gcd(y_0 + 1, n)$ gives a factor.
  - If $n = n_1 n_2$, then a $y_0 < n$ with $y_0 = 1 \ (mod\ n_1)$ and $y_0 = -1 \ (mod\ n_2)$ exists (by CRT) and is therefore a non-trivial solution.

Integer Factoring

- Let $\phi(X) = a \cdot X + b$ be a non-trivial automorphism of $R = Z_n[X]/(X^2 - 1)$.
- Let $d = \gcd(a, n)$.
- Consider $\phi(\frac{n}{d}X) = \frac{n}{d} \cdot a \cdot X + \frac{n}{d} \cdot b = \frac{n}{d} \cdot b$.
- Since ϕ is a 1-1 map, this is only possible when $d = \gcd(a, n) = 1$.

Integer Factoring

- Let $\phi(X) = a \cdot X + b$ be a non-trivial automorphism of $R = Z_n[X]/(X^2 - 1)$.
- Let $d = \gcd(a, n)$.
- Consider $\phi(\frac{n}{d}X) = \frac{n}{d} \cdot a \cdot X + \frac{n}{d} \cdot b = \frac{n}{d} \cdot b$.
- Since ϕ is a 1-1 map, this is only possible when $d = \gcd(a, n) = 1$.

Integer Factoring

- Let $\phi(X) = a \cdot X + b$ be a non-trivial automorphism of $R = Z_n[X]/(X^2 - 1)$.
- Let $d = \gcd(a, n)$.
- Consider $\phi(\frac{n}{d}X) = \frac{n}{d} \cdot a \cdot X + \frac{n}{d} \cdot b = \frac{n}{d} \cdot b$.
- Since ϕ is a 1-1 map, this is only possible when $d = \gcd(a, n) = 1$.

INTEGER FACTORING

- We have:

$$0 = \phi(X^2 - 1) = (aX + b)^2 - 1 = 2abX + a^2 + b^2 - 1$$

  in the ring.
- This gives $2ab = 0 = a^2 + b^2 - 1 \ (mod\ n)$.
- Since n is odd and $\gcd(a, n) = 1$, we get $b = 0 \ (mod\ n)$ and $a^2 = 1 \ (mod\ n)$.
- Therefore, $\phi(X) = a \cdot X$ with $a^2 = 1 \ (mod\ n)$.
- As ϕ is non-trivial, $a \neq \pm 1 \ (mod\ n)$.
- So, given ϕ, we can use a to factor n.

Integer Factoring

- We have:

$$0 = \phi(X^2 - 1) = (aX + b)^2 - 1 = 2abX + a^2 + b^2 - 1$$

  in the ring.
- This gives $2ab = 0 = a^2 + b^2 - 1 \ (mod\ n)$.
- Since n is odd and $\gcd(a, n) = 1$, we get $b = 0 \ (mod\ n)$ and $a^2 = 1 \ (mod\ n)$.
- Therefore, $\phi(X) = a \cdot X$ with $a^2 = 1 \ (mod\ n)$.
- As ϕ is non-trivial, $a \neq \pm 1 \ (mod\ n)$.
- So, given ϕ, we can use a to factor n.

INTEGER FACTORING

- We have:

$$0 = \phi(X^2 - 1) = (aX + b)^2 - 1 = 2abX + a^2 + b^2 - 1$$

  in the ring.
- This gives $2ab = 0 = a^2 + b^2 - 1 \ (mod\ n)$.
- Since n is odd and $\gcd(a, n) = 1$, we get $b = 0 \ (mod\ n)$ and $a^2 = 1 \ (mod\ n)$.
- Therefore, $\phi(X) = a \cdot X$ with $a^2 = 1 \ (mod\ n)$.
- As ϕ is non-trivial, $a \neq \pm 1 \ (mod\ n)$.
- So, given ϕ, we can use a to factor n.

Integer Factoring

- We have:

$$0 = \phi(X^2 - 1) = (aX + b)^2 - 1 = 2abX + a^2 + b^2 - 1$$

  in the ring.
- This gives $2ab = 0 = a^2 + b^2 - 1 \ (mod\ n)$.
- Since n is odd and $\gcd(a, n) = 1$, we get $b = 0 \ (mod\ n)$ and $a^2 = 1 \ (mod\ n)$.
- Therefore, $\phi(X) = a \cdot X$ with $a^2 = 1 \ (mod\ n)$.
- As ϕ is non-trivial, $a \neq \pm 1 \ (mod\ n)$.
- So, given ϕ, we can use a to factor n.

Integer Factoring

- Conversely, assume that we know a number a such that $a \neq \pm 1 \ (mod\ n)$ and $a^2 = 1 \ (mod\ n)$.
- This a defines a non-trivial automorphism of R. □

Integer Factoring

- Conversely, assume that we know a number a such that $a \neq \pm 1 \ (mod\ n)$ and $a^2 = 1 \ (mod\ n)$.
- This a defines a non-trivial automorphism of R. □

Tool 4: Hensel Lifting

Outline

Definition

Example: Polynomial Division

Hensel Lifting

- Let $R = \mathbb{Z}$ or $F[x]$, and $m \in R$.
- Hensel (1918) designed a method to compute factorization of any element of R modulo m^ℓ given its factorization modulo m.
- The method is called Hensel Lifting.
- It is used in several places: polynomial division, polynomial factorization etc.

Hensel Lifting

- Suppose we are given $f, g, h, s, t \in R$ such that $f = g \cdot h \ (mod\ m)$, $\gcd(g, h) = 1 \ (mod\ m)$, and $sg + th = 1 \ (mod\ m)$.
- Compute $e = f - gh \ (mod\ m^2)$, $g' = g + te \ (mod\ m^2)$, $h' = h + se \ (mod\ m^2)$.
- Then we get:

$$\begin{aligned} g'h' \ (mod\ m^2) &= gh + sge + the + ste^2 \ (mod\ m^2) \\ &= gh + (sg + th)(f - gh) \ (mod\ m^2) \\ &= f \ (mod\ m^2). \end{aligned}$$

Hensel Lifting

- Suppose we are given $f, g, h, s, t \in R$ such that $f = g \cdot h \ (mod\ m)$, $\gcd(g, h) = 1 \ (mod\ m)$, and $sg + th = 1 \ (mod\ m)$.
- Compute $e = f - gh \ (mod\ m^2)$, $g' = g + te \ (mod\ m^2)$, $h' = h + se \ (mod\ m^2)$.
- Then we get:

$$\begin{aligned} g'h' \ (mod\ m^2) &= gh + sge + the + ste^2 \ (mod\ m^2) \\ &= gh + (sg + th)(f - gh) \ (mod\ m^2) \\ &= f \ (mod\ m^2). \end{aligned}$$

Hensel Lifting

- Suppose we are given $f, g, h, s, t \in R$ such that $f = g \cdot h \ (mod\ m)$, $\gcd(g, h) = 1 \ (mod\ m)$, and $sg + th = 1 \ (mod\ m)$.
- Compute $e = f - gh \ (mod\ m^2)$, $g' = g + te \ (mod\ m^2)$, $h' = h + se \ (mod\ m^2)$.
- Then we get:

$$\begin{aligned} g'h' \ (mod\ m^2) &= gh + sge + the + ste^2 \ (mod\ m^2) \\ &= gh + (sg + th)(f - gh) \ (mod\ m^2) \\ &= f \ (mod\ m^2). \end{aligned}$$

Hensel Lifting

- Also compute $d = sg' + th' - 1 \ (mod\ m^2)$, $s' = s(1-d)\ (mod\ m^2)$, $t' = t(1-d)\ (mod\ m^2)$.
- Then:

$$\begin{aligned} s'g' + t'h'\ (mod\ m^2) &= sg'(1-d) + th'(1-d)\ (mod\ m^2) \\ &= (1+d)(1-d)\ (mod\ m^2) \\ &= 1\ (mod\ m^2). \end{aligned}$$

- Thus we can 'lift' the factorization to modulo m^2.
- Iterating this $\log \ell$ times gives factorization modulo m^ℓ.

Hensel Lifting

- Also compute $d = sg' + th' - 1 \ (mod\ m^2)$, $s' = s(1-d) \ (mod\ m^2)$, $t' = t(1-d) \ (mod\ m^2)$.
- Then:

$$\begin{aligned} s'g' + t'h' \ (mod\ m^2) &= sg'(1-d) + th'(1-d) \ (mod\ m^2) \\ &= (1+d)(1-d) \ (mod\ m^2) \\ &= 1 \ (mod\ m^2). \end{aligned}$$

- Thus we can 'lift' the factorization to modulo m^2.
- Iterating this $\log \ell$ times gives factorization modulo m^ℓ.

Hensel Lifting

- Also compute $d = sg' + th' - 1 \ (mod\ m^2)$, $s' = s(1-d) \ (mod\ m^2)$, $t' = t(1-d) \ (mod\ m^2)$.
- Then:

$$
\begin{aligned}
s'g' + t'h' \ (mod\ m^2) &= sg'(1-d) + th'(1-d) \ (mod\ m^2) \\
&= (1+d)(1-d) \ (mod\ m^2) \\
&= 1 \ (mod\ m^2).
\end{aligned}
$$

- Thus we can 'lift' the factorization to modulo m^2.
- Iterating this $\log \ell$ times gives factorization modulo m^ℓ.

Outline

Definition

Example: Polynomial Division

Polynomial Division via Hensel Lifting

- Let $f(x)$ and $g(x)$ be two monic polynomials over field F, $\deg f = n$, $\deg g = m < n$.
- We wish to compute $d(x)$ and $r(x)$ such that $f = dg + r$ and $\deg r < m$.
- A naive algorithm takes $O(n^2)$ field operations.
- Using Hensel Lifting, we can do it in $O(n \log n)$ operations.

Polynomial Division via Hensel Lifting

- Let $f(x)$ and $g(x)$ be two monic polynomials over field F, $\deg f = n$, $\deg g = m < n$.
- We wish to compute $d(x)$ and $r(x)$ such that $f = dg + r$ and $\deg r < m$.
- A naive algorithm takes $O(n^2)$ field operations.
- Using Hensel Lifting, we can do it in $O(n \log n)$ operations.

Polynomial Division via Hensel Lifting

- For any polynomial $p(x)$ of degree d, define $\widetilde{p}(x) = x^d p(\frac{1}{x})$.
- The coefficients of $\widetilde{p}$ are 'reversed'.
- If $f(x) = d(x)g(x) + r(x)$, then

$$\widetilde{f}(x) = \widetilde{d}(x)\widetilde{g}(x) + x^{n-m+1}\widetilde{r}(x).$$

- Therefore,

$$\widetilde{f}(x) = \widetilde{d}(x)\widetilde{g}(x) \ (mod\ x^{n-m+1}).$$

Polynomial Division via Hensel Lifting

- For any polynomial $p(x)$ of degree d, define $\widetilde{p}(x) = x^d p(\frac{1}{x})$.
- The coefficients of $\widetilde{p}$ are 'reversed'.
- If $f(x) = d(x)g(x) + r(x)$, then

$$\widetilde{f}(x) = \widetilde{d}(x)\widetilde{g}(x) + x^{n-m+1}\widetilde{r}(x).$$

- Therefore,

$$\widetilde{f}(x) = \widetilde{d}(x)\widetilde{g}(x) \ (mod\ x^{n-m+1}).$$

POLYNOMIAL DIVISION VIA HENSEL LIFTING

- Since $\widetilde{g}(x)$ has degree zero coefficient 1, it is invertible modulo x^{n-m+1}.
- So, $\widetilde{d}(x) = \widetilde{f}(x) \cdot \widetilde{g}^{-1}(x) \ (mod\ x^{n-m+1})$.
- So if we can compute $\widetilde{g}^{-1}(x) \ (mod\ x^{n-m+1})$, then one multiplication would give $\widetilde{d}(x)$ from which $d(x)$ and then $r(x) = f(x) - d(x)g(x)$ can be easily recovered.
- We use Hensel Lifting to compute $\widetilde{g}^{-1}(x) \ (mod\ x^{n-m+1})$.

Polynomial Division via Hensel Lifting

- Since $\widetilde{g}(x)$ has degree zero coefficient 1, it is invertible modulo x^{n-m+1}.
- So, $\widetilde{d}(x) = \widetilde{f}(x) \cdot \widetilde{g}^{-1}(x)\ (mod\ x^{n-m+1})$.
- So if we can compute $\widetilde{g}^{-1}(x)\ (mod\ x^{n-m+1})$, then one multiplication would give $\widetilde{d}(x)$ from which $d(x)$ and then $r(x) = f(x) - d(x)g(x)$ can be easily recovered.
- We use Hensel Lifting to compute $\widetilde{g}^{-1}(x)\ (mod\ x^{n-m+1})$.

Polynomial Division via Hensel Lifting

- Since $\widetilde{g}(x)$ has degree zero coefficient 1, it is invertible modulo x^{n-m+1}.
- So, $\widetilde{d}(x) = \widetilde{f}(x) \cdot \widetilde{g}^{-1}(x)\ (mod\ x^{n-m+1})$.
- So if we can compute $\widetilde{g}^{-1}(x)\ (mod\ x^{n-m+1})$, then one multiplication would give $\widetilde{d}(x)$ from which $d(x)$ and then $r(x) = f(x) - d(x)g(x)$ can be easily recovered.
- We use Hensel Lifting to compute $\widetilde{g}^{-1}(x)\ (mod\ x^{n-m+1})$.

Polynomial Division via Hensel Lifting

- Let $h(x) = \widetilde{g}^{-1}(x)\ (mod\ x^{n-m+1})$.
- So, $h(x) \cdot \widetilde{g}(x) = 1\ (mod\ x^{n-m+1})$.
- Notice that $\widetilde{g}(x)\ (mod\ x) = 1$ and so $h(x)\ (mod\ x) = 1$.
- Let $s(x) = 1$ and $t(x) = 0$ so $s \cdot h + t \cdot \widetilde{g} = 1\ (mod\ x)$.
- Use Hensel Lifting iteratively $\ell = \lceil \log(n-m+1) \rceil$ times to compute $h(x)\ (mod\ x^{2^\ell})$ such that $h(x) \cdot \widetilde{g}(x) = 1\ (mod\ x^{2^\ell})$.
  - As we start with $t = 0$, t will remain zero through all the iterations.
  - Therefore, function $\widetilde{g}$ will also not change, as required.

Polynomial Division via Hensel Lifting

- Let $h(x) = \widetilde{g}^{-1}(x)\ (mod\ x^{n-m+1})$.
- So, $h(x) \cdot \widetilde{g}(x) = 1\ (mod\ x^{n-m+1})$.
- Notice that $\widetilde{g}(x)\ (mod\ x) = 1$ and so $h(x)\ (mod\ x) = 1$.
- Let $s(x) = 1$ and $t(x) = 0$ so $s \cdot h + t \cdot \widetilde{g} = 1\ (mod\ x)$.
- Use Hensel Lifting iteratively $\ell = \lceil \log(n - m + 1) \rceil$ times to compute $h(x)\ (mod\ x^{2^\ell})$ such that $h(x) \cdot \widetilde{g}(x) = 1\ (mod\ x^{2^\ell})$.
    - As we start with $t = 0$, t will remain zero through all the iterations.
    - Therefore, function $\widetilde{g}$ will also not change, as required.

Polynomial Division via Hensel Lifting

- This gives the inverse of $\widetilde{g}(x)$ $(mod\ x^{n-m+1})$.
- The algorithm uses only multiplication and addition.
- The kth iteration uses a constant number of multiplication and addition of polynomials of degree 2^k.
- Therefore, the whole algorithm requires $O(\sum_{k=1}^{\ell} M_P(2^k)) = O(M_P(2^\ell) = O(M_P(n)) = O(n \log n)$ operations.

Polynomial Division via Hensel Lifting

- This gives the inverse of $\widetilde{g}(x)$ $(mod\ x^{n-m+1})$.
- The algorithm uses only multiplication and addition.
- The kth iteration uses a constant number of multiplication and addition of polynomials of degree 2^k.
- Therefore, the whole algorithm requires $O(\sum_{k=1}^{\ell} M_P(2^k)) = O(M_P(2^\ell) = O(M_P(n)) = O(n \log n)$ operations.

Polynomial Division via Hensel Lifting

- This gives the inverse of $\widetilde{g}(x)$ $(mod\ x^{n-m+1})$.
- The algorithm uses only multiplication and addition.
- The kth iteration uses a constant number of multiplication and addition of polynomials of degree 2^k.
- Therefore, the whole algorithm requires $O(\sum_{k=1}^{\ell} M_P(2^k)) = O(M_P(2^\ell) = O(M_P(n)) = O(n \log n)$ operations.

Tool 5: Short Vectors in a Lattice

Outline

Lattices and LLL Algorithm

Example: Solving Modular Equations

Example: Polynomial Factoring Over Rationals

Lattices

- Let $\hat{v}_1, \ldots, \hat{v}_n \in \mathbb{R}^n$ be linearly independent vectors.
- Then,

$$\mathcal{L} = \{\sum_{i=1}^{n} \alpha_i \hat{v}_i \mid \alpha_1, \ldots, \alpha_n \in \mathbb{Z}\}$$

  is lattice generated by $\hat{v}_1, \ldots, \hat{v}_n$.
- Vector $\hat{v}$ is shortest vector in lattice $\mathcal{L}$ if $\|\hat{v}\|_2$ is minimum.

Lattices

- Let $\hat{v}_1, \ldots, \hat{v}_n \in \mathbb{R}^n$ be linearly independent vectors.
- Then,

$$\mathcal{L} = \{\sum_{i=1}^{n} \alpha_i \hat{v}_i \mid \alpha_1, \ldots, \alpha_n \in \mathbb{Z}\}$$

  is lattice generated by $\hat{v}_1, \ldots, \hat{v}_n$.
- Vector $\hat{v}$ is shortest vector in lattice $\mathcal{L}$ if $\|\hat{v}\|_2$ is minimum.

Lattices

- Let $\hat{v}_1, \ldots, \hat{v}_n \in \mathbb{R}^n$ be linearly independent vectors.
- Then,

$$\mathcal{L} = \{\sum_{i=1}^{n} \alpha_i \hat{v}_i \mid \alpha_1, \ldots, \alpha_n \in \mathbb{Z}\}$$

  is lattice generated by $\hat{v}_1, \ldots, \hat{v}_n$.
- Vector $\hat{v}$ is shortest vector in lattice $\mathcal{L}$ if $\|\hat{v}\|_2$ is minimum.

Lattices

- For lattice $\mathcal{L}$, its norm $|\mathcal{L}|$ is defined to be $\det(\hat{v}_1\ \hat{v}_2\ \ldots\ \hat{v}_n)$.
- $|\mathcal{L}|$ is independent of the choice of basis of $\mathcal{L}$.

Theorem (Minkowski, 1896)

The length of shortest vector of $\mathcal{L}$ is at most $\sqrt{n} \cdot |\mathcal{L}|^{1/n}$.

Lattices

- For lattice $\mathcal{L}$, its norm $|\mathcal{L}|$ is defined to be $\det(\hat{v}_1 \; \hat{v}_2 \; \dots \; \hat{v}_n)$.
- $|\mathcal{L}|$ is independent of the choice of basis of $\mathcal{L}$.

Theorem (Minkowski, 1896)

The length of shortest vector of $\mathcal{L}$ is at most $\sqrt{n} \cdot |\mathcal{L}|^{1/n}$.

LLL Algorithm

- Lenstra, Lenstra and Lovasz (1982) designed a polynomial-time algorithm for computing a short vector in any lattice.
- The algorithm computes a vector whose length is at most $2^{\frac{n-1}{2}}$ times the length of shortest vector in the lattice.
- It is now known that finding a vector within a $\sqrt{2}$ factor of shortest vector length is NP-hard.

LLL Algorithm

- Lenstra, Lenstra and Lovasz (1982) designed a polynomial-time algorithm for computing a short vector in any lattice.
- The algorithm computes a vector whose length is at most $2^{\frac{n-1}{2}}$ times the length of shortest vector in the lattice.
- It is now known that finding a vector within a $\sqrt{2}$ factor of shortest vector length is NP-hard.

LLL Algorithm

- Lenstra, Lenstra and Lovasz (1982) designed a polynomial-time algorithm for computing a short vector in any lattice.
- The algorithm computes a vector whose length is at most $2^{\frac{n-1}{2}}$ times the length of shortest vector in the lattice.
- It is now known that finding a vector within a $\sqrt{2}$ factor of shortest vector length is NP-hard.

Outline

Lattices and LLL Algorithm

Example: Solving Modular Equations

Example: Polynomial Factoring Over Rationals

Finding Small Solutions of Modular Equations

- Modular equations for prime moduli can be solved using polynomial factorization.
- But this does not work for composite moduli.
- For this, short lattice vectors can be used to find small solutions.
    - Small = solutions much smaller than the moduli in absolute value
- An example is breaking low-exponent RSA when part of the message is known.

Finding Small Solutions of Modular Equations

- Modular equations for prime moduli can be solved using polynomial factorization.
- But this does not work for composite moduli.
- For this, short lattice vectors can be used to find small solutions.
  - Small = solutions much smaller than the moduli in absolute value
- An example is breaking low-exponent RSA when part of the message is known.

Finding Small Solutions of Modular Equations

- Modular equations for prime moduli can be solved using polynomial factorization.
- But this does not work for composite moduli.
- For this, short lattice vectors can be used to find small solutions.
  - Small = solutions much smaller than the moduli in absolute value
- An example is breaking low-exponent RSA when part of the message is known.

Breaking Low Exponent RSA

- Let $(n, 3)$ be the public-key of an RSA cryptosystem.
- Notice that the exponent of encryption is set to 3.
- Let $c = m^3 \ (mod\ n)$ be a ciphertext.
- Suppose that leading $\frac{11}{12}|n|$ bits of m are known.
- This is possible in certain situations, e.g., when there is a fixed $\frac{11}{12}|n|$-bit header appended to each message.
- Let $m = h \cdot 2^{|n|/12} + x$ where h is known.

Breaking Low Exponent RSA

- Let $(n, 3)$ be the public-key of an RSA cryptosystem.
- Notice that the exponent of encryption is set to 3.
- Let $c = m^3 \ (mod\ n)$ be a ciphertext.
- Suppose that leading $\frac{11}{12}|n|$ bits of m are known.
- This is possible in certain situations, e.g., when there is a fixed $\frac{11}{12}|n|$-bit header appended to each message.
- Let $m = h \cdot 2^{|n|/12} + x$ where h is known.

Breaking Low Exponent RSA

- Let $(n, 3)$ be the public-key of an RSA cryptosystem.
- Notice that the exponent of encryption is set to 3.
- Let $c = m^3 \ (mod\ n)$ be a ciphertext.
- Suppose that leading $\frac{11}{12}|n|$ bits of m are known.
- This is possible in certain situations, e.g., when there is a fixed $\frac{11}{12}|n|$-bit header appended to each message.
- Let $m = h \cdot 2^{|n|/12} + x$ where h is known.

Breaking Low Exponent RSA

- Let $(n, 3)$ be the public-key of an RSA cryptosystem.
- Notice that the exponent of encryption is set to 3.
- Let $c = m^3 \ (mod\ n)$ be a ciphertext.
- Suppose that leading $\frac{11}{12}|n|$ bits of m are known.
- This is possible in certain situations, e.g., when there is a fixed $\frac{11}{12}|n|$-bit header appended to each message.
- Let $m = h \cdot 2^{|n|/12} + x$ where h is known.

Breaking Low Exponent RSA

- Therefore, $c = (h \cdot 2^{|n|/12} + x)^3 \ (mod\ n) = x^3 + a_2x^2 + a_1x + a_0 \ (mod\ n)$.
- So if we can find all the roots of the above polynomial that are less than $2^{|n|/12} = n^{1/12}$ then m can be recovered.
- For a vector $\hat{v} \in \mathbb{Z}^d$, $\hat{v} = [v_{d-1}\ v_{d-2}\ \cdots\ v_0]$, let $v(x) = \sum_{i=0}^{d-1} v_i x^i$ and vice-versa.
- Let $p_3(x) = x^3 + a_2x^2 + a_1x + (a_0 - c)$.
- Then $\hat{p}_3 = [0\ 0\ 1\ a_2\ a_1\ a_0 - c] \in \mathbb{Z}^6$.

Breaking Low Exponent RSA

- Therefore,
  $c = (h \cdot 2^{|n|/12} + x)^3 \ (mod\ n) = x^3 + a_2 x^2 + a_1 x + a_0 \ (mod\ n)$.
- So if we can find all the roots of the above polynomial that are less than $2^{|n|/12} = n^{1/12}$ then m can be recovered.
- For a vector $\hat{v} \in \mathbb{Z}^d$, $\hat{v} = [v_{d-1}\ v_{d-2}\ \cdots\ v_0]$, let $v(x) = \sum_{i=0}^{d-1} v_i x^i$ and vice-versa.
- Let $p_3(x) = x^3 + a_2 x^2 + a_1 x + (a_0 - c)$.
- Then $\hat{p}_3 = [0\ 0\ 1\ a_2\ a_1\ a_0 - c] \in \mathbb{Z}^6$.

Breaking Low Exponent RSA

- Therefore, $c = (h \cdot 2^{|n|/12} + x)^3 \ (mod\ n) = x^3 + a_2x^2 + a_1x + a_0 \ (mod\ n)$.
- So if we can find all the roots of the above polynomial that are less than $2^{|n|/12} = n^{1/12}$ then m can be recovered.
- For a vector $\hat{v} \in \mathbb{Z}^d$, $\hat{v} = [v_{d-1}\ v_{d-2}\ \cdots\ v_0]$, let $v(x) = \sum_{i=0}^{d-1} v_i x^i$ and vice-versa.
- Let $p_3(x) = x^3 + a_2x^2 + a_1x + (a_0 - c)$.
- Then $\hat{p}_3 = [0\ 0\ 1\ a_2\ a_1\ a_0 - c] \in \mathbb{Z}^6$.

Breaking Low Exponent RSA

- Let $p_4(x) = x \cdot p_3(x)$, $p_5(x) = x^2 \cdot p_3(x)$, $p_0(x) = n$, $p_1(x) = n \cdot x$, and $p_2(x) = n \cdot x^2$.
- Let $\mathcal{L}$ be the lattice generated by vectors $\hat{p}_0, \ldots, \hat{p}_5$.
- Let vector $\hat{v} \in \mathcal{L}$, $\hat{v} = \sum_{i=0}^{5} \alpha_i \hat{p}_i$.
- Notice that polynomial $v(x) = \sum_{i=0}^{5} \alpha_i p_i(x) = p_3(x) \cdot q(x)\ (mod\ n)$ for some $q(x)$ of degree two.
- Hence, every root of $p_3(x)\ (mod\ n)$ is also a root of $v(x)\ (mod\ n)$.

Breaking Low Exponent RSA

- Let $p_4(x) = x \cdot p_3(x)$, $p_5(x) = x^2 \cdot p_3(x)$, $p_0(x) = n$, $p_1(x) = n \cdot x$, and $p_2(x) = n \cdot x^2$.
- Let $\mathcal{L}$ be the lattice generated by vectors $\hat{p}_0, \ldots, \hat{p}_5$.
- Let vector $\hat{v} \in \mathcal{L}$, $\hat{v} = \sum_{i=0}^{5} \alpha_i \hat{p}_i$.
- Notice that polynomial $v(x) = \sum_{i=0}^{5} \alpha_i p_i(x) = p_3(x) \cdot q(x) \ (mod\ n)$ for some $q(x)$ of degree two.
- Hence, every root of $p_3(x) \ (mod\ n)$ is also a root of $v(x) \ (mod\ n)$.

BREAKING LOW EXPONENT RSA

- Let $p_4(x) = x \cdot p_3(x)$, $p_5(x) = x^2 \cdot p_3(x)$, $p_0(x) = n$, $p_1(x) = n \cdot x$, and $p_2(x) = n \cdot x^2$.
- Let $\mathcal{L}$ be the lattice generated by vectors $\hat{p}_0, \ldots, \hat{p}_5$.
- Let vector $\hat{v} \in \mathcal{L}$, $\hat{v} = \sum_{i=0}^{5} \alpha_i \hat{p}_i$.
- Notice that polynomial $v(x) = \sum_{i=0}^{5} \alpha_i p_i(x) = p_3(x) \cdot q(x) \ (mod\ n)$ for some $q(x)$ of degree two.
- Hence, every root of $p_3(x) \ (mod\ n)$ is also a root of $v(x) \ (mod\ n)$.

BREAKING LOW EXPONENT RSA

- We have $|\mathcal{L}| = n^3$.
- By the property of lattices, $\mathcal{L}$ has a shortest vector of length at most $\sqrt{6}n^{3/6} = \sqrt{6n}$.
- Run LLL algorithm to find a short vector $\hat{u}$ in $\mathcal{L}$.
- The length of $\hat{u}$ is at most $2^{5/2}\sqrt{6n} = 4\sqrt{12n}$.
- Let $u(x) = \sum_{i=0}^{5} \beta_i x^i$.
- We have $|\beta_i| \leq 4\sqrt{12n}$.

Breaking Low Exponent RSA

- We have $|\mathcal{L}| = n^3$.
- By the property of lattices, $\mathcal{L}$ has a shortest vector of length at most $\sqrt{6}n^{3/6} = \sqrt{6n}$.
- Run LLL algorithm to find a short vector $\hat{u}$ in $\mathcal{L}$.
- The length of $\hat{u}$ is at most $2^{5/2}\sqrt{6n} = 4\sqrt{12n}$.
- Let $u(x) = \sum_{i=0}^{5} \beta_i x^i$.
- We have $|\beta_i| \leq 4\sqrt{12n}$.

Breaking Low Exponent RSA

- We have $|\mathcal{L}| = n^3$.
- By the property of lattices, $\mathcal{L}$ has a shortest vector of length at most $\sqrt{6}n^{3/6} = \sqrt{6n}$.
- Run LLL algorithm to find a short vector $\hat{u}$ in $\mathcal{L}$.
- The length of $\hat{u}$ is at most $2^{5/2}\sqrt{6n} = 4\sqrt{12n}$.
- Let $u(x) = \sum_{i=0}^{5} \beta_i x^i$.
- We have $|\beta_i| \leq 4\sqrt{12n}$.

Breaking Low Exponent RSA

- Consider a root γ of $p_3(x)$ $(mod\ n)$ with $\gamma \leq n^{1/12}$.
- As argued above, γ is a root of $u(x)$ $(mod\ n)$ too.
- Now, $|u(\gamma)| \leq 24\sqrt{12n} \cdot \gamma^5 < n$ for $n > (24\sqrt{12})^{12}$.
- Therefore, $u(\gamma) = 0$ over rationals!
- Factor $u(x)$ over rationals to compute all its roots.
- Identify the root that yields the ciphertext.

Breaking Low Exponent RSA

- Consider a root γ of $p_3(x)$ $(mod\ n)$ with $\gamma \leq n^{1/12}$.
- As argued above, γ is a root of $u(x)$ $(mod\ n)$ too.
- Now, $|u(\gamma)| \leq 24\sqrt{12n} \cdot \gamma^5 < n$ for $n > (24\sqrt{12})^{12}$.
- Therefore, $u(\gamma) = 0$ over rationals!
- Factor $u(x)$ over rationals to compute all its roots.
- Identify the root that yields the ciphertext.

Breaking Low Exponent RSA

- Consider a root γ of $p_3(x)$ $(mod\ n)$ with $\gamma \leq n^{1/12}$.
- As argued above, γ is a root of $u(x)$ $(mod\ n)$ too.
- Now, $|u(\gamma)| \leq 24\sqrt{12n} \cdot \gamma^5 < n$ for $n > (24\sqrt{12})^{12}$.
- Therefore, $u(\gamma) = 0$ over rationals!
- Factor $u(x)$ over rationals to compute all its roots.
- Identify the root that yields the ciphertext.

Breaking Low Exponent RSA

- This breaks exponent-3 RSA when first $\frac{11}{12}$-fraction of bits of plaintext are known.
- This can be improved to first $\frac{1}{2}$-fraction.
- Also generalizes to any small exponent.

Breaking Low Exponent RSA

- This breaks exponent-3 RSA when first $\frac{11}{12}$-fraction of bits of plaintext are known.
- This can be improved to first $\frac{1}{2}$-fraction.
- Also generalizes to any small exponent.

Breaking Low Exponent RSA

- This breaks exponent-3 RSA when first $\frac{11}{12}$-fraction of bits of plaintext are known.
- This can be improved to first $\frac{1}{2}$-fraction.
- Also generalizes to any small exponent.

Outline

Lattices and LLL Algorithm

Example: Solving Modular Equations

Example: Polynomial Factoring Over Rationals

The Problem

- Given a monic polynomial $f(x)$ of degree n, factor f over rationals.
- A deterministic polynomial time algorithm for this was given by Lenstra, Lenstra, Lovasz (1982).
- The algorithm uses Hensel Lifting and short vectors in lattices.

The Problem

- Given a monic polynomial $f(x)$ of degree n, factor f over rationals.
- A deterministic polynomial time algorithm for this was given by Lenstra, Lenstra, Lovasz (1982).
- The algorithm uses Hensel Lifting and short vectors in lattices.

Factoring Polynomials Over Rationals

- Choose a small prime p, and factor f over F_p.
- Let $f = g_1 \cdot g_2 \ (mod\ p)$ with g_1 being irreducible.
- Let ℓ be the smallest integer greater than $\frac{3}{2}(n^2 - 1) + (2n + 1) \log \|f\|_2$.
- Use Hensel Lifting to compute factors of f modulo p^ℓ.
- Let $f = g_1' \cdot g_2' \ (mod\ p^\ell)$.
- Note that g_1' remains irreducible modulo p^ℓ.

Factoring Polynomials Over Rationals

- Choose a small prime p, and factor f over F_p.
- Let $f = g_1 \cdot g_2 \ (mod\ p)$ with g_1 being irreducible.
- Let ℓ be the smallest integer greater than $\frac{3}{2}(n^2 - 1) + (2n+1)\log \|f\|_2$.
- Use Hensel Lifting to compute factors of f modulo p^ℓ.
- Let $f = g_1' \cdot g_2' \ (mod\ p^\ell)$.
- Note that g_1' remains irreducible modulo p^ℓ.

Factoring Polynomials Over Rationals

- Without loss of generality, assume g_1' is monic and $\deg(g_1') = d$.
- Define polynomials $h_i(x) = p^\ell x^i$ for $0 \leq i < d$.
- Define polynomials $h_{d+i}(x) = x^i \cdot g_1'(x)$ for $0 \leq i < n - d$.
- As before, let $\mathcal{L}$ be the n-dimensional lattice generated by vectors $\hat{h}_0, \ldots, \hat{h}_{n-1}$.
- The lattice contains precisely degree $n-1$ polynomials that are multiples of g_1' modulo p^ℓ.
- This lattice has a shortest vector of length at most $\sqrt{n}p^{d\ell/n}$.
- So, LLL algorithm produces a vector of length at most $2^{\frac{n-1}{2}}\sqrt{n}p^{d\ell/n}$.

FACTORING POLYNOMIALS OVER RATIONALS

- Without loss of generality, assume g_1' is monic and $\deg(g_1') = d$.
- Define polynomials $h_i(x) = p^\ell x^i$ for $0 \le i < d$.
- Define polynomials $h_{d+i}(x) = x^i \cdot g_1'(x)$ for $0 \le i < n - d$.
- As before, let $\mathcal{L}$ be the n-dimensional lattice generated by vectors $\hat{h}_0, \ldots, \hat{h}_{n-1}$.
- The lattice contains precisely degree $n-1$ polynomials that are multiples of g_1' modulo p^ℓ.
- This lattice has a shortest vector of length at most $\sqrt{n}p^{d\ell/n}$.
- So, LLL algorithm produces a vector of length at most $2^{\frac{n-1}{2}}\sqrt{n}p^{d\ell/n}$.

Factoring Polynomials Over Rationals

- Without loss of generality, assume g_1' is monic and $\deg(g_1') = d$.
- Define polynomials $h_i(x) = p^\ell x^i$ for $0 \leq i < d$.
- Define polynomials $h_{d+i}(x) = x^i \cdot g_1'(x)$ for $0 \leq i < n - d$.
- As before, let $\mathcal{L}$ be the n-dimensional lattice generated by vectors $\hat{h}_0, \ldots, \hat{h}_{n-1}$.
- The lattice contains precisely degree $n - 1$ polynomials that are multiples of g_1' modulo p^ℓ.
- This lattice has a shortest vector of length at most $\sqrt{n}p^{d\ell/n}$.
- So, LLL algorithm produces a vector of length at most $2^{\frac{n-1}{2}}\sqrt{n}p^{d\ell/n}$.

Factoring Polynomials Over Rationals

- But we can do better!
- Suppose $f = f_1 \cdot f_2$ over rationals.
- Since $f = g_1' \cdot g_2' \ (mod\ p^\ell)$, g_1' is irreducible and $Z_{p^\ell}[x]$ is a UFD, g_1' divides either f_1 or f_2 modulo p^ℓ.
- Without loss of generality, assume that $f_1 = f_1' \cdot g_1' \ (mod\ p^\ell)$.
- Then the vector $\hat{f}_1$ is in the lattice $\mathcal{L}$.
- What is the length of $\hat{f}_1$?

FACTORING POLYNOMIALS OVER RATIONALS

- But we can do better!
- Suppose $f = f_1 \cdot f_2$ over rationals.
- Since $f = g_1' \cdot g_2' \ (mod\ p^\ell)$, g_1' is irreducible and $Z_{p^\ell}[x]$ is a UFD, g_1' divides either f_1 or f_2 modulo p^ℓ.
- Without loss of generality, assume that $f_1 = f_1' \cdot g_1' \ (mod\ p^\ell)$.
- Then the vector $\hat{f}_1$ is in the lattice $\mathcal{L}$.
- What is the length of $\hat{f}_1$?

Factoring Polynomials Over Rationals

- But we can do better!
- Suppose $f = f_1 \cdot f_2$ over rationals.
- Since $f = g_1' \cdot g_2' \ (mod\ p^\ell)$, g_1' is irreducible and $Z_{p^\ell}[x]$ is a UFD, g_1' divides either f_1 or f_2 modulo p^ℓ.
- Without loss of generality, assume that $f_1 = f_1' \cdot g_1' \ (mod\ p^\ell)$.
- Then the vector $\hat{f}_1$ is in the lattice $\mathcal{L}$.
- What is the length of $\hat{f}_1$?

FACTORING POLYNOMIALS OVER RATIONALS

- Mignotte's bound shows that $\|f_1\|_2 \leq 2^{n-1}\|f\|_2$.
- Therefore, length of $\hat{f}_1 = \|f_1\|_2 \leq 2^{n-1}\|f\|_2$.
- So, the LLL algorithm will produce a vector $\hat{v}$ of length at most $2^{\frac{3(n-1)}{2}}\|f\|_2$.
- Consider polynomial $v(x)$.
- Since $\hat{v} \in \mathcal{L}$, $g_1'(x)$ divides $v(x)$ modulo p^ℓ.

Factoring Polynomials Over Rationals

- Mignotte's bound shows that $\|f_1\|_2 \leq 2^{n-1}\|f\|_2$.
- Therefore, length of $\hat{f}_1 = \|f_1\|_2 \leq 2^{n-1}\|f\|_2$.
- So, the LLL algorithm will produce a vector $\hat{v}$ of length at most $2^{\frac{3(n-1)}{2}}\|f\|_2$.
- Consider polynomial $v(x)$.
- Since $\hat{v} \in \mathcal{L}$, $g_1'(x)$ divides $v(x)$ modulo p^ℓ.

Factoring Polynomials Over Rationals

- Mignotte's bound shows that $\|f_1\|_2 \leq 2^{n-1}\|f\|_2$.
- Therefore, length of $\hat{f}_1 = \|f_1\|_2 \leq 2^{n-1}\|f\|_2$.
- So, the LLL algorithm will produce a vector $\hat{v}$ of length at most $2^{\frac{3(n-1)}{2}}\|f\|_2$.
- Consider polynomial $v(x)$.
- Since $\hat{v} \in \mathcal{L}$, $g_1'(x)$ divides $v(x)$ modulo p^ℓ.

Factoring Polynomials Over Rationals

- Therefore, $\gcd(v(x), f(x)) > 1 \ (mod\ p^{\ell})$.
- Using the resultant, we can say $\text{Res}(v(x), f(x)) = 0 \ (mod\ p^{\ell})$.
- Resultant of $v(x)$ and $f(x)$ is an $(2n+1) \times (2n+1)$ matrix whose columns are essentially vectors $\hat{v}$ and $\hat{f}$.
- From Hadamard's Inequality it follows that

$$\text{Res}(v(x), f(x)) \leq \|v\|_2^{n+1} \|f\|_2^n \leq 2^{\frac{3(n^2-1)}{2}} \|f\|_2^{2n+1}.$$

Factoring Polynomials Over Rationals

- Therefore, $\gcd(v(x), f(x)) > 1\ (mod\ p^\ell)$.
- Using the resultant, we can say $\mathrm{Res}(v(x), f(x)) = 0\ (mod\ p^\ell)$.
- Resultant of $v(x)$ and $f(x)$ is an $(2n+1) \times (2n+1)$ matrix whose columns are essentially vectors $\hat{v}$ and $\hat{f}$.
- From Hadamard's Inequality it follows that

$$\mathrm{Res}(v(x), f(x)) \leq \|v\|_2^{n+1} \|f\|_2^{n} \leq 2^{\frac{3(n^2-1)}{2}} \|f\|_2^{2n+1}.$$

Factoring Polynomials Over Rationals

- By the choice of ℓ, $\ell > \frac{3}{2}(n^2 - 1) + (2n + 1) \log \|f\|_2$, it follows that

$$\text{Res}(v(x), f(x)) < p^{\ell}.$$

- Coupled with the fact that $\text{Res}(v(x), f(x)) = 0 \ (mod\ p^{\ell})$, we get

$$\text{Res}(v(x), f(x)) = 0$$

over rationals.

- In other words, $\gcd(v(x), f(x)) > 1$ over rationals and thus we get a factor of f.

Factoring Polynomials Over Rationals

- By the choice of ℓ, $\ell > \frac{3}{2}(n^2 - 1) + (2n + 1)\log \|f\|_2$, it follows that
$$\text{Res}(v(x), f(x)) < p^\ell.$$
- Coupled with the fact that $\text{Res}(v(x), f(x)) = 0 \ (mod\ p^\ell)$, we get
$$\text{Res}(v(x), f(x)) = 0$$
over rationals.
- In other words, $\gcd(v(x), f(x)) > 1$ over rationals and thus we get a factor of f.

Factoring Polynomials Over Rationals

- By the choice of ℓ, $\ell > \frac{3}{2}(n^2 - 1) + (2n+1)\log\|f\|_2$, it follows that
$$\text{Res}(v(x), f(x)) < p^\ell.$$

- Coupled with the fact that $\text{Res}(v(x), f(x)) = 0 \ (mod\ p^\ell)$, we get
$$\text{Res}(v(x), f(x)) = 0$$
over rationals.
- In other words, $\gcd(v(x), f(x)) > 1$ over rationals and thus we get a factor of f.

Tool 6: Smooth Numbers

Outline

Definition

Example: Integer Factoring via Quadratic Sieve

Example: Discrete Log Computation via Index Calculus

Smooth Numbers

- Number $n > 0$ is *m-smooth* if all prime divisors of n are $\leq m$.
- Let $\Psi(x, y)$ denote the size of the set of numbers $\leq x$ that are y-smooth.

Theorem (Density of Smooth Numbers)
$\Psi(x, y) = x \cdot r^{-r(1+o(1))}$ *where* $r = \frac{\ln x}{\ln y}$, *and* $y = \Omega(\ln^2 x)$.

SMOOTH NUMBERS

- Number $n > 0$ is m-smooth if all prime divisors of n are $\leq m$.
- Let $\Psi(x, y)$ denote the size of the set of numbers $\leq x$ that are y-smooth.

THEOREM (DENSITY OF SMOOTH NUMBERS)

$\Psi(x, y) = x \cdot r^{-r(1+o(1))}$ *where* $r = \frac{\ln x}{\ln y}$, *and* $y = \Omega(\ln^2 x)$.

Smooth Numbers

- Number $n > 0$ is m-smooth if all prime divisors of n are $\leq m$.
- Let $\Psi(x, y)$ denote the size of the set of numbers $\leq x$ that are y-smooth.

Theorem (Density of Smooth Numbers)

$\Psi(x, y) = x \cdot r^{-r(1+o(1))}$ *where* $r = \frac{\ln x}{\ln y}$, *and* $y = \Omega(\ln^2 x)$.

Smooth Numbers

- Smooth numbers are used in Elliptic Curve Factoring, Quadratic Sieve and Number Field Sieve, the three most popular integer factoring algorithms.
- They are also used in index calculus method for discrete log problem.

Outline

Definition

Example: Integer Factoring via Quadratic Sieve

Example: Discrete Log Computation via Index Calculus

Quadratic Sieve

- Designed by Carl Pomerance (1983).
- Let n be an odd number with at least two distinct prime factors.
- n can be factored if non-trivial solution of the equation $x^2 = y^2 \ (mod\ n)$ can be computed.
  - A non-trivial solution is (x_0, y_0) such that $x_0^2 = y_0^2 \ (mod\ n)$ and $x_0 \neq \pm y_0 \ (mod\ n)$.
  - Given such a solution, $\gcd(x_0 + y_0, n)$ gives a factor of n.
- We will use this approach for factoring n.

Quadratic Sieve

- Designed by Carl Pomerance (1983).
- Let n be an odd number with at least two distinct prime factors.
- n can be factored if non-trivial solution of the equation $x^2 = y^2 \ (mod\ n)$ can be computed.
  - A non-trivial solution is (x_0, y_0) such that $x_0^2 = y_0^2 \ (mod\ n)$ and $x_0 \neq \pm y_0 \ (mod\ n)$.
  - Given such a solution, $\gcd(x_0 + y_0, n)$ gives a factor of n.
- We will use this approach for factoring n.

Quadratic Sieve

- Designed by Carl Pomerance (1983).
- Let n be an odd number with at least two distinct prime factors.
- n can be factored if non-trivial solution of the equation $x^2 = y^2 \ (mod\ n)$ can be computed.
  - A non-trivial solution is (x_0, y_0) such that $x_0^2 = y_0^2 \ (mod\ n)$ and $x_0 \neq \pm y_0 \ (mod\ n)$.
  - Given such a solution, $\gcd(x_0 + y_0, n)$ gives a factor of n.
- We will use this approach for factoring n.

Quadratic Sieve

1. Let $m = \lceil \sqrt{n} \rceil$, $B = e^{\frac{1}{2}\sqrt{\ln n \ln \ln n}}$, and p_1, ..., p_t the set of all primes $\leq B$.
2. For $k = 1, 2, 3, \ldots$ do the following:
   2.1 Let $v = m + k$.
   2.2 Let $u = v^2 \ (mod\ n)$, $0 < u < n$.
   2.3 Check if u is B-smooth.
   2.4 If yes, compute complete factorization of $u = \prod_{i=1}^{t} p_i^{e[i]}$.
   2.5 Store the triple $(u, v, \hat{e})$ where $\hat{e} = (e[1]\ e[2]\ \cdots\ e[t])$.

Quadratic Sieve

1. Let $m = \lceil \sqrt{n} \rceil$, $B = e^{\frac{1}{2}\sqrt{\ln n \ln \ln n}}$, and $p_1, \ldots, p_t$ the set of all primes $\leq B$.
2. For $k = 1, 2, 3, \ldots$ do the following:
   2.1 Let $v = m + k$.
   2.2 Let $u = v^2 \ (mod\ n)$, $0 < u < n$.
   2.3 Check if u is B-smooth.
   2.4 If yes, compute complete factorization of $u = \prod_{i=1}^{t} p_i^{e[i]}$.
   2.5 Store the triple $(u, v, \hat{e})$ where $\hat{e} = (e[1]\ e[2]\ \cdots\ e[t])$.

Quadratic Sieve

1. Let $m = \lceil \sqrt{n} \rceil$, $B = e^{\frac{1}{2}\sqrt{\ln n \ln \ln n}}$, and $p_1, \ldots, p_t$ the set of all primes $\leq B$.
2. For $k = 1, 2, 3, \ldots$ do the following:
   2.1 Let $v = m + k$.
   2.2 Let $u = v^2 \ (mod\ n)$, $0 < u < n$.
   2.3 Check if u is B-smooth.
   2.4 If yes, compute complete factorization of $u = \prod_{i=1}^{t} p_i^{e[i]}$.
   2.5 Store the triple $(u, v, \hat{e})$ where $\hat{e} = (e[1]\ e[2]\ \cdots\ e[t])$.

Quadratic Sieve

3. Exit the previous step after $t+1$ triples are stored.
4. Let these be $\{u_j, v_j, \hat{e}_j\}_{1 \le j \le t+1}$.
5. Find $\alpha_j \in \{0,1\}$ for $1 \le j \le t+1$ such that $\sum_{j=1}^{t+1} \alpha_j \hat{e}_j = 0 \ (mod\ 2)$ and not all α_j's are zero. [always possible]
6. Let

$$x = \prod_{j=1}^{t+1} v_j^{\alpha_j}$$

and

$$y = \prod_{i=1}^{t} p_i^{\frac{1}{2}\sum_{j=1}^{t+1} \alpha_j e_j[i]} = \prod_{j=1}^{t+1} \prod_{i=1}^{t} p_i^{\frac{1}{2}\alpha_j e_j[i]} = \prod_{j=1}^{t+1} u_j^{\frac{1}{2}\alpha_j}.$$

Quadratic Sieve

3. Exit the previous step after $t+1$ triples are stored.
4. Let these be $\{u_j, v_j, \hat{e}_j\}_{1 \le j \le t+1}$.
5. Find $\alpha_j \in \{0,1\}$ for $1 \le j \le t+1$ such that $\sum_{j=1}^{t+1} \alpha_j \hat{e}_j = 0\ (mod\ 2)$ and not all α_j's are zero. [always possible]
6. Let

$$x = \prod_{j=1}^{t+1} v_j^{\alpha_j}$$

and

$$y = \prod_{i=1}^{t} p_i^{\frac{1}{2}\sum_{j=1}^{t+1} \alpha_j e_j[i]} = \prod_{j=1}^{t+1}\prod_{i=1}^{t} p_i^{\frac{1}{2}\alpha_j e_j[i]} = \prod_{j=1}^{t+1} u_j^{\frac{1}{2}\alpha_j}.$$

Quadratic Sieve

3. Exit the previous step after $t+1$ triples are stored.
4. Let these be $\{u_j, v_j, \hat{e}_j\}_{1 \le j \le t+1}$.
5. Find $\alpha_j \in \{0,1\}$ for $1 \le j \le t+1$ such that $\sum_{j=1}^{t+1} \alpha_j \hat{e}_j = 0 \ (mod\ 2)$ and not all α_j's are zero. [always possible]
6. Let

$$x = \prod_{j=1}^{t+1} v_j^{\alpha_j}$$

and

$$y = \prod_{i=1}^{t} p_i^{\frac{1}{2}\sum_{j=1}^{t+1} \alpha_j e_j[i]} = \prod_{j=1}^{t+1} \prod_{i=1}^{t} p_i^{\frac{1}{2}\alpha_j e_j[i]} = \prod_{j=1}^{t+1} u_j^{\frac{1}{2}\alpha_j}.$$

Quadratic Sieve

7. Compute $\gcd(x + y, n)$ and check if a proper factor of n is obtained.
8. If not, generate more triples and repeat.

Quadratic Sieve Analysis

- First note that for each j, $\sum_{j=1}^{t+1} \alpha_j e_j[i]$ is divisible by two and so y is an integer.
- We have $x^2 = \prod_{j=1}^{t+1} \{v_j^2\}^{\alpha_j} = \prod_{j=1}^{t+1} u_j^{\alpha_j} \pmod{n} = y^2 \pmod{n}$.
- Since x and y are computed using very different numbers (x is a product of numbers of the form $m + k$ and y is a product of powers of p_i's), it is likely that $x \neq \pm y \pmod{n}$.
- This results in a factor of n.

Quadratic Sieve Analysis

- First note that for each j, $\sum_{j=1}^{t+1} \alpha_j e_j[i]$ is divisible by two and so y is an integer.
- We have $x^2 = \prod_{j=1}^{t+1} \{v_j^2\}^{\alpha_j} = \prod_{j=1}^{t+1} u_j^{\alpha_j} \ (mod\ n) = y^2 \ (mod\ n)$.
- Since x and y are computed using very different numbers (x is a product of numbers of the form $m + k$ and y is a product of powers of p_i's), it is likely that $x \neq \pm y \ (mod\ n)$.
- This results in a factor of n.

Quadratic Sieve Analysis

- First note that for each j, $\sum_{j=1}^{t+1} \alpha_j e_j[i]$ is divisible by two and so y is an integer.
- We have $x^2 = \prod_{j=1}^{t+1} \{v_j^2\}^{\alpha_j} = \prod_{j=1}^{t+1} u_j^{\alpha_j} \ (mod\ n) = y^2 \ (mod\ n)$.
- Since x and y are computed using very different numbers (x is a product of numbers of the form $m + k$ and y is a product of powers of p_i's), it is likely that $x \neq \pm y \ (mod\ n)$.
- This results in a factor of n.

Quadratic Sieve Analysis

- So how many k's are required to generate $t+1$ triples?
- Number $u = (m+k)^2 \ (mod\ n) \approx 2\sqrt{n}k + k^2 \approx 2\sqrt{n}k$ when k is small compared to $\sqrt{n}$.
- Assume that u is uniformly distributed over $[1, 2\sqrt{n}k]$ as k varies.
- Then the probability that u is B-smooth is around $(\frac{\ln n}{2 \ln B})^{-\frac{\ln n}{2 \ln B}} \sim e^{-\frac{1}{2}\sqrt{\ln n \ln \ln n}} = \frac{1}{B}$.
- So we need $B^{2+o(1)}$ k's to generate required triples.

Quadratic Sieve Analysis

- So how many k's are required to generate $t+1$ triples?
- Number $u = (m+k)^2 \ (mod\ n) \approx 2\sqrt{n}k + k^2 \approx 2\sqrt{n}k$ when k is small compared to $\sqrt{n}$.
- Assume that u is uniformly distributed over $[1, 2\sqrt{n}k]$ as k varies.
- Then the probability that u is B-smooth is around $(\frac{\ln n}{2 \ln B})^{-\frac{\ln n}{2 \ln B}} \sim e^{-\frac{1}{2}\sqrt{\ln n \ln \ln n}} = \frac{1}{B}$.
- So we need $B^{2+o(1)}$ k's to generate required triples.

Quadratic Sieve Analysis

- So how many k's are required to generate $t+1$ triples?
- Number $u = (m+k)^2 \ (mod\ n) \approx 2\sqrt{n}k + k^2 \approx 2\sqrt{n}k$ when k is small compared to $\sqrt{n}$.
- Assume that u is uniformly distributed over $[1, 2\sqrt{n}k]$ as k varies.
- Then the probability that u is B-smooth is around $(\frac{\ln n}{2 \ln B})^{-\frac{\ln n}{2 \ln B}} \sim e^{-\frac{1}{2}\sqrt{\ln n \ln \ln n}} = \frac{1}{B}$.
- So we need $B^{2+o(1)}$ k's to generate required triples.

Quadratic Sieve Analysis

- Using a clever sieving trick, it can be shown that time taken to compute all the triples remains $B^{2+o(1)}$.
- α_j's can be computed by solving a system of $t+1$ linear equations.
- Time taken to compute these can be shown to be $O(t^2) = O(B^2)$.
- Therefore, the time complexity of the whole algorithm is $B^{2+o(1)} = e^{(1+o(1))\sqrt{\ln n \ln \ln n}}$.

Quadratic Sieve Analysis

- Using a clever sieving trick, it can be shown that time taken to compute all the triples remains $B^{2+o(1)}$.
- α_j's can be computed by solving a system of $t+1$ linear equations.
- Time taken to compute these can be shown to be $O(t^2) = O(B^2)$.
- Therefore, the time complexity of the whole algorithm is $B^{2+o(1)} = e^{(1+o(1))\sqrt{\ln n \ln \ln n}}$.

Quadratic Sieve Analysis

- Using a clever sieving trick, it can be shown that time taken to compute all the triples remains $B^{2+o(1)}$.
- α_j's can be computed by solving a system of $t+1$ linear equations.
- Time taken to compute these can be shown to be $O(t^2) = O(B^2)$.
- Therefore, the time complexity of the whole algorithm is $B^{2+o(1)} = e^{(1+o(1))\sqrt{\ln n \ln \ln n}}$.

Number Field Sieve

- Designed by Pollard, Pomerance, Lenstra, … (1990s).
- Uses arithmetic in a number field instead of $\mathbb{Q}$.
- This allows one to reduce the size of u's thus increasing the chances of finding a smooth number.
- The time complexity comes down to $e^{c(\ln n)^{1/3}(\ln \ln n)^{2/3}}$, $c \approx 1.903$.

Number Field Sieve

- Designed by Pollard, Pomerance, Lenstra, ... (1990s).
- Uses arithmetic in a number field instead of $\mathbb{Q}$.
- This allows one to reduce the size of u's thus increasing the chances of finding a smooth number.
- The time complexity comes down to $e^{c(\ln n)^{1/3}(\ln \ln n)^{2/3}}$, $c \approx 1.903$.

Outline

Definition

Example: Integer Factoring via Quadratic Sieve

Example: Discrete Log Computation via Index Calculus

Discrete Log Problem Over Finite Fields

- Let p be a large prime.
- Let $g \in F_p$ be a generator of F_p^* and $\gamma \in F_p^*$.
- The discrete log problem over finite fields is: given p, g, and γ, compute m such that $g^m = \gamma \ (mod\ p)$.
- The hardness of this problem is the basis for security of El Gamal type encryption algorithms over finite fields and Diffie-Hellman key exchange scheme.

Discrete Log Problem Over Finite Fields

- Let p be a large prime.
- Let $g \in F_p$ be a generator of F_p^* and $\gamma \in F_p^*$.
- The discrete log problem over finite fields is: given p, g, and γ, compute m such that $g^m = \gamma \ (mod\ p)$.
- The hardness of this problem is the basis for security of El Gamal type encryption algorithms over finite fields and Diffie-Hellman key exchange scheme.

Index Calculus Method

- Compute r and s such that $g^r \gamma^s = 1 \ (mod\ p)$ and $\gcd(s, p-1) = 1$.
- Then $g^{r+ms} = 1 \ (mod\ p)$ giving $m = -rs^{-1} \ (mod\ p-1)$.
- How does one quickly find such r and s?
- We use a method similar to one used for integer factoring.

Index Calculus Method

- Compute r and s such that $g^r\gamma^s = 1\ (mod\ p)$ and $\gcd(s, p-1) = 1$.
- Then $g^{r+ms} = 1\ (mod\ p)$ giving $m = -rs^{-1}\ (mod\ p-1)$.
- How does one quickly find such r and s?
- We use a method similar to one used for integer factoring.

Index Calculus Method

- Compute r and s such that $g^r \gamma^s = 1 \ (mod\ p)$ and $\gcd(s, p-1) = 1$.
- Then $g^{r+ms} = 1 \ (mod\ p)$ giving $m = -rs^{-1} \ (mod\ p-1)$.
- How does one quickly find such r and s?
- We use a method similar to one used for integer factoring.

Index Calculus Method

1. Let $B = e^{\frac{1}{2}\sqrt{\ln p \ln \ln p}}$ and $p_1, \ldots, p_t$ be all primes $\leq B$.
2. Randomly select r and s, $0 < r, s < p - 1$.
3. Compute $u = g^r \gamma^s \ (mod\ p)$.
4. Check if u is B-smooth.
5. If yes, compute complete factorization of $u = \prod_{i=1}^{t} p_i^{e[i]}$.
6. Store the 4-tuple $(r, s, u, \hat{e})$ where $\hat{e} = (e[1]\ e[2]\ \cdots\ e[t])$.

Index Calculus Method

1. Let $B = e^{\frac{1}{2}\sqrt{\ln p \ln \ln p}}$ and $p_1, \ldots, p_t$ be all primes $\leq B$.
2. Randomly select r and s, $0 < r, s < p - 1$.
3. Compute $u = g^r \gamma^s \ (mod\ p)$.
4. Check if u is B-smooth.
5. If yes, compute complete factorization of $u = \prod_{i=1}^{t} p_i^{e[i]}$.
6. Store the 4-tuple $(r, s, u, \hat{e})$ where $\hat{e} = (e[1]\ e[2]\ \cdots\ e[t])$.

Index Calculus Method

1. Let $B = e^{\frac{1}{2}\sqrt{\ln p \ln \ln p}}$ and $p_1, \ldots, p_t$ be all primes $\leq B$.
2. Randomly select r and s, $0 < r, s < p - 1$.
3. Compute $u = g^r \gamma^s \ (mod\ p)$.
4. Check if u is B-smooth.
5. If yes, compute complete factorization of $u = \prod_{i=1}^{t} p_i^{e[i]}$.
6. Store the 4-tuple $(r, s, u, \hat{e})$ where $\hat{e} = (e[1]\ e[2]\ \cdots\ e[t])$.

INDEX CALCULUS METHOD

7. Exit the previous step after $t+1$ 4-tuples are stored.
8. Let these be $\{r_j, s_j, u_j, \hat{e}_j\}_{1\leq j\leq t+1}$.
9. Find $\alpha_j \in Z_{p-1}$ for $1 \leq j \leq t+1$ such that $\sum_{j=1}^{t+1} \alpha_j \hat{e}_j = 0 \ (mod\ p-1)$ and not all α_j's are zero.
10. Let

$$r = \sum_{j=1}^{t+1} \alpha_j r_j \ (mod\ p-1)$$

and

$$s = \sum_{j=1}^{t+1} \alpha_j s_j \ (mod\ p-1).$$

Index Calculus Method

7. Exit the previous step after $t+1$ 4-tuples are stored.
8. Let these be $\{r_j, s_j, u_j, \hat{e}_j\}_{1 \le j \le t+1}$.
9. Find $\alpha_j \in Z_{p-1}$ for $1 \le j \le t+1$ such that $\sum_{j=1}^{t+1} \alpha_j \hat{e}_j = 0 \ (mod\ p-1)$ and not all α_j's are zero.
10. Let

$$r = \sum_{j=1}^{t+1} \alpha_j r_j \ (mod\ p-1)$$

and

$$s = \sum_{j=1}^{t+1} \alpha_j s_j \ (mod\ p-1).$$

Index Calculus Method

7. Exit the previous step after $t+1$ 4-tuples are stored.
8. Let these be $\{r_j, s_j, u_j, \hat{e}_j\}_{1 \le j \le t+1}$.
9. Find $\alpha_j \in Z_{p-1}$ for $1 \le j \le t+1$ such that $\sum_{j=1}^{t+1} \alpha_j \hat{e}_j = 0 \ (mod\ p-1)$ and not all α_j's are zero.
10. Let

$$r = \sum_{j=1}^{t+1} \alpha_j r_j \ (mod\ p-1)$$

and

$$s = \sum_{j=1}^{t+1} \alpha_j s_j \ (mod\ p-1).$$

Index Calculus Method

11. Check if $\gcd(s, p-1) = 1$.
12. If yes, $m = -rs^{-1}\ (mod\ p-1)$ is the answer.

Analysis of Index Calculus Method

- Note that

$$
\begin{aligned}
g^r\gamma^s &= \prod_{j=1}^{t+1}(g^{r_j}\gamma^{s_j})^{\alpha_j} \ (mod\ p) \\
&= \prod_{j=1}^{t+1} u_j^{\alpha_j} \ (mod\ p) \\
&= \prod_{j=1}^{t+1}\prod_{i=1}^{t} p_i^{\alpha_j e_j[i]} \ (mod\ p) \\
&= \prod_{i=1}^{t} p_i^{\sum_{j=1}^{t+1}\alpha_j e_j[i]} \ (mod\ p) \\
&= 1 \ (mod\ p).
\end{aligned}
$$

Analysis of Index Calculus Method

- In addition, the probability that $\gcd(s, p-1) = 1$ is high since s_j's are randomly chosen.
- Therefore, the algorithm computes discrete log with high probability.
- For time complexity we proceed exactly as before.
- The probability that u is B-smooth is $\frac{\Psi(p-1,B)}{p-1} \sim \left(\frac{\ln p}{\ln B}\right)^{-\frac{\ln p}{\ln B}} \sim e^{-\ln p \ln\ln p} = \frac{1}{B^2}$.

Analysis of Index Calculus Method

- In addition, the probability that $\gcd(s, p-1) = 1$ is high since s_j's are randomly chosen.
- Therefore, the algorithm computes discrete log with high probability.
- For time complexity we proceed exactly as before.
- The probability that u is B-smooth is $\frac{\Psi(p-1,B)}{p-1} \sim \left(\frac{\ln p}{\ln B}\right)^{-\frac{\ln p}{\ln B}} \sim e^{-\ln p \ln \ln p} = \frac{1}{B^2}$.

Analysis of Index Calculus Method

- Therefore, we need to generate $B^{3+o(1)}$ u's.
- Testing each u for smoothness takes $B^{1+o(1)}$ steps (no savings here!).
- Also, solving the system of linear equation takes $O(B^3)$ steps.
- This gives the total complexity of $B^{4+o(1)} = e^{(2+o(1))\sqrt{\ln p \ln \ln p}}$.

Analysis of Index Calculus Method

- Therefore, we need to generate $B^{3+o(1)}$ u's.
- Testing each u for smoothness takes $B^{1+o(1)}$ steps (no savings here!).
- Also, solving the system of linear equation takes $O(B^3)$ steps.
- This gives the total complexity of $B^{4+o(1)} = e^{(2+o(1))\sqrt{\ln p \ln \ln p}}$.

Analysis of Index Calculus Method

- Therefore, we need to generate $B^{3+o(1)}$ u's.
- Testing each u for smoothness takes $B^{1+o(1)}$ steps (no savings here!).
- Also, solving the system of linear equation takes $O(B^3)$ steps.
- This gives the total complexity of $B^{4+o(1)} = e^{(2+o(1))\sqrt{\ln p \ln \ln p}}$.

Comments

- As in case of factoring, number fields can be used to bring the time complexity down to $e^{c(\ln n)^{1/3}(\ln\ln n)^{2/3}}$.
- The index calculus method can be generalized to work for any finite commutative group.
- However, it does not work well in groups with no good notion of 'smoothness'.
- For example, in group of points on an elliptic curve E_p.

Comments

- As in case of factoring, number fields can be used to bring the time complexity down to $e^{c(\ln n)^{1/3}(\ln\ln n)^{2/3}}$.
- The index calculus method can be generalized to work for any finite commutative group.
- However, it does not work well in groups with no good notion of 'smoothness'.
- For example, in group of points on an elliptic curve E_p.

Comments

- As in case of factoring, number fields can be used to bring the time complexity down to $e^{c(\ln n)^{1/3}(\ln\ln n)^{2/3}}$.
- The index calculus method can be generalized to work for any finite commutative group.
- However, it does not work well in groups with no good notion of 'smoothness'.
- For example, in group of points on an elliptic curve E_p.

Thank You!

Resultants

- Let f and v be two polynomials over field F of degree n and m respectively.
- We have $\gcd(f(x), v(x)) > 1$ iff there exist $r(x)$ and $s(x)$, of degrees $< m$ and $< n$ respectively, such that $r(x)f(x) + s(x)v(x) = 0$.
- Define map $T(r(x), s(x)) = r(x)f(x) + s(x)v(x)$ for $\deg(r) < m$ and $\deg(s) < n$.
- T is a bilinear map and so can be represented by a $(n+m) \times (n+m)$ matrix, $M_{f,v}$.
- Further, T is invertible iff $\gcd(f(x), v(x)) = 1$.
- Let $\text{Res}(f, v) = \det M_{f,v}$.

Resultants

- Let f and v be two polynomials over field F of degree n and m respectively.
- We have $\gcd(f(x), v(x)) > 1$ iff there exist $r(x)$ and $s(x)$, of degrees $< m$ and $< n$ respectively, such that $r(x)f(x) + s(x)v(x) = 0$.
- Define map $T(r(x), s(x)) = r(x)f(x) + s(x)v(x)$ for $\deg(r) < m$ and $\deg(s) < n$.
- T is a bilinear map and so can be represented by a $(n+m) \times (n+m)$ matrix, $M_{f,v}$.
- Further, T is invertible iff $\gcd(f(x), v(x)) = 1$.
- Let $\mathrm{Res}(f, v) = \det M_{f,v}$.

Resultants

- Let f and v be two polynomials over field F of degree n and m respectively.
- We have $\gcd(f(x), v(x)) > 1$ iff there exist $r(x)$ and $s(x)$, of degrees $< m$ and $< n$ respectively, such that $r(x)f(x) + s(x)v(x) = 0$.
- Define map $T(r(x), s(x)) = r(x)f(x) + s(x)v(x)$ for $\deg(r) < m$ and $\deg(s) < n$.
- T is a bilinear map and so can be represented by a $(n+m) \times (n+m)$ matrix, $M_{f,v}$.
- Further, T is invertible iff $\gcd(f(x), v(x)) = 1$.
- Let $\text{Res}(f, v) = \det M_{f,v}$.

Resultants

- Let f and v be two polynomials over field F of degree n and m respectively.
- We have $\gcd(f(x), v(x)) > 1$ iff there exist $r(x)$ and $s(x)$, of degrees $< m$ and $< n$ respectively, such that $r(x)f(x) + s(x)v(x) = 0$.
- Define map $T(r(x), s(x)) = r(x)f(x) + s(x)v(x)$ for $\deg(r) < m$ and $\deg(s) < n$.
- T is a bilinear map and so can be represented by a $(n+m) \times (n+m)$ matrix, $M_{f,v}$.
- Further, T is invertible iff $\gcd(f(x), v(x)) = 1$.
- Let $\text{Res}(f, v) = \det M_{f,v}$.