A Survey of Techniques Used in Algebraic and Number Theoretic Algorithms

Manindra Agrawal and Chandan Saha
Department of Computer Science and Engineering
Indian Institute of Technology Kanpur

Abstract

We survey some of the important tools and techniques used in designing algorithms for problems in algebra and number theory. We focus on the computational efficiency of these tools and show how they are applied to design algorithms for several basic algebraic and number theoretic operations.

1 Introduction

The past few decades have witnessed a growing interest among computer scientists and mathematicians, in the field of computational number theory and algebra. This field is primarily focussed on the task of designing fast algorithms for several algebraic and number theoretic problems. Owing to the fundamental nature of these problems, they have always been a subject of intense theoretical study. But interest in them has escalated recently because of their important applications in key areas like cryptography and coding theory.

The security of cryptographic protocols such as the RSA cryptosystem and the Diffie-Hellman key exchange protocol, relies on the hardness of problems like integer factoring and discrete logarithm. On the other hand, the efficiency of decoding algorithms for error correcting codes like Reed-Solomon and BCH codes hinges on fast algorithms for solving a system of linear equations and factoring polynomials over finite fields.

Another important application of algorithmic algebra is the development of computer algebra systems. These systems are indispensable tools for research in many computation intensive fields of physics, chemistry, biology and mathematics.

In this survey we briefly review some of the fundamental and widely used techniques for designing algebraic and number theoretic algorithms. Algebraic algorithms are used for performing operations like:

- **Polynomial operations:** Polynomial addition, multiplication, gcd computation, factoring, interpolation, multipoint evaluation, etc.
- **Matrix operations:** Matrix addition, multiplication, inverse and determinant computation, solving a system of linear equations, etc.
- **Abstract algebra operations:** Finding the order of a group element, computing discrete logarithm, etc.

Whereas number theoretic algorithms are used for performing operations like:

- **Operations on integers and rationals:** Addition, multiplication, gcd computation, square root finding, primality testing, integer factoring, etc.

We discuss two major applications namely, the RSA cryptosystem and the Reed-Solomon error correcting codes, where some of these algorithms are used. Following these applications, we give a brief account of the computational complexity of the basic operations mentioned above. Several important techniques are used to design algorithms for these operations. We describe some of these techniques here. These include:

1. **Chinese Remaindering:** Used in determinant computation and polynomial factoring.
2. **Discrete Fourier Transform:** Used in polynomial and integer multiplication.
3. **Automorphisms:** Used in polynomial and integer factoring, and primality testing.
4. **Hensel Lifting:** Used in polynomial factorization and division.
5. **Short Vectors in a Lattice:** Used in polynomial factoring and breaking cryptosystems.
6. **Smooth Numbers:** Used in integer factoring and computing discrete logarithm.

Our motive - Before we proceed to the main discussion let us at first clarify our motive behind writing this survey. An attempt to describe a field as vast as computational number theory through a relatively short article as this is perhaps a bit ambitious. But giving a thorough account of this subject is not our intention. Instead, we want to present a quick, yet fairly comprehensive, exposition to some of the salient features of this area, so that the reader finds it helpful to pursue further details on topics that interest him or her. To assist the reader in this regard, references to relevant details are provided along with every topic that we discuss.

We do assume that the reader is familiar with basic algebraic structures like groups, rings, fields and vector spaces, and their properties.

Notations and conventions - The set of integers, rationals, reals and complex numbers are denoted by $\mathbb{Z}, \mathbb{Q}, \mathbb{R}$ and $\mathbb{C}$, respectively. $\mathbb{Z}^+$ is the set of positive integers and $\mathbb{Z}_N$ is the ring of integers modulo $N \in \mathbb{Z}^+$. The multiplicative subgroup of $\mathbb{Z}_N$, consisting of all $m \in \mathbb{Z}^+$ with $\gcd(m, N) = 1$, is denoted by $\mathbb{Z}_N^\times$. For a positive real a, $\lfloor a \rfloor$ is the largest integer less than a, and $\lceil a \rceil$ is the smallest integer greater than a. $\mathbb{F}$ represents any arbitrary field, and $\mathbb{F}_q$ is the finite field with q elements. $\mathbb{F}_q^\times = \mathbb{F}_q \backslash \{0\}$ is the multiplicative subgroup of $\mathbb{F}_q$. The determinant of a square matrix M is denoted by $\det(M)$.

Given two functions $t_1(n)$ and $t_2(n)$ from $\mathbb{Z}^+ \to \mathbb{Z}^+$, we say $t_1(n) = O(t_2(n))$ if there exist positive constants c and n_0 such that $t_1(n) \leq c \cdot t_2(n)$ for every $n \geq n_0$. We write $t_1(n) = o(t_2(n))$ if for every constant c, there is an $n_0 > 0$ such that $t_1(n) < c \cdot t_2(n)$ for all $n \geq n_0$. We use the notation $\mathsf{poly}(n_1, \ldots, n_k)$ to mean some arbitrary but fixed polynomial in the parameters $n_1, \ldots, n_k$. Throughout this article, log refers to logarithm base 2 and ln is the natural logarithm. Sometimes, for brevity we use the notation $\tilde{O}(t(n))$ to mean $O(t(n) \cdot \mathsf{poly}(\log t(n)))$.

In this article, a ring $\mathcal{R}$ always means a commutative ring with unity 1 and by convention, any vector is a row vector. Given two polynomials $f, g \in \mathcal{R}[x]$, where $\mathcal{R}$ is a unique factorization domain, $\gcd(f, g)$ refers to the largest common divisor of f and g over $\mathcal{R}$ which is unique up to multiplication by units in $\mathcal{R}$. When $\mathcal{R}$ is a field, we assume that $\gcd(f, g)$ is monic.

Given two polynomials $f, g \in \mathcal{R}[x]$, where $\mathcal{R}$ is an integral domain, $S(f, g)$ denotes the Sylvester matrix of f and g over $\mathcal{R}$. The resultant of f and g over $\mathcal{R}$ is $\mathsf{Res}_x(f, g) = \det(S(f, g))$. Refer to Appendix A, for a discussion on resultant theory and Gram-Schmidt orthogonalization.

2 Two major applications in cryptography and coding theory

In this section, we take up the examples of the RSA cryptosystem and the Reed-Solomon codes, and show how there are intimately connected to problems like integer and polynomial factoring. As another motivating application, we give a brief account of the Diffie-Hellman key exchange protocol where the hardness of the discrete log problem comes into play.

2.1 The RSA cryptosystem

The RSA, named after its inventors Rivest, Shamir, and Adleman [RSA78], is the first and the most popular public-key cryptosystem. A cryptosystem is an encryption-cum-decryption scheme for communication between a sender and a receiver. Such a system is *secure* if it is infeasible for a (potentially malicious) third party to eavesdrop on the encrypted message and decrypt it efficiently. In a public-key cryptosystem, the receiver publishes a common key (also known as the public key) using which anyone can encrypt a message and send it to the receiver. On the other hand, only the receiver knows a secret private key using which the message can be decrypted efficiently. The RSA key generation procedure is as follows.

1 RSA: Key generation

1. Fix a key length, say, 2^r bits.
2. Randomly select two distinct primes p and q each of 2^{r-1} bits.
3. Let $n = pq$ and $\varphi(n) = (p-1)(q-1)$. /* φ is the totient function.*/
4. Randomly select an e such that $3 \leq e < \varphi(n)$ and $\gcd(e, \varphi(n)) = 1$.
5. Find the smallest d such that $d \cdot e = 1 \mod \varphi(n)$.
6. The encryption key is the pair (n, e).
7. The decryption key is d.

In practice, the primes p and q chosen in step 2 are fairly large in size, say about 512 bits. Evidently, a fast primality testing algorithm is required to pick such large primes. Other basic operations like multiplication (in step 3), gcd computation (in step 4) and modular inverse computation (in step 5) also play a significant role in determining the efficiency of the process, as the numbers involved are extremely large. The encryption key (n, e) is known to all, whereas the decryption key d is known only to the receiver. The RSA encryption scheme is as follows.

2 RSA: Encryption

1. Let m be the message to be encrypted.
2. Treat m as a number less than n.
3. Compute $c = m^e \mod n$.
4. c is the encrypted message.

Note the usage of the modular exponentiation operation in step 3. At the receiver's end the message is decrypted using d as follows.

3 RSA: Decryption

1. Compute $c^d \mod n = m^{ed} \mod n = m$.

Since n is given as part of the public key, if we can factor n efficiently we can compute the private key $d = e^{-1} \mod \varphi(n)$ using extended Euclidean algorithm (see Algorithm 8 in section

3.1). However, as yet no randomized polynomial time algorithm is known for integer factoring. The current best factoring algorithms ([JP92], [LJMP90]) have subexponential time complexity. Although it is not known if breaking RSA is equivalent to integer factoring, a recent result by Aggarwal and Maurer [AM09] shows that under a reasonable restriction this is indeed the case.

Readers interested in further details on potential attacks on RSA cryptosystem may consult the survey articles by Koblitz and Menezes [KM04], Boneh [Bon99], or the book by Katzenbeisser [Kat01]. In section 4.5 we will discuss a lattice based attack on the RSA, which was first proposed by Coppersmith [Cop97].

Another problem which is historically closely related to integer factoring is the discrete logarithm problem. In the next section, we show how the *hardness* of this problem is used to design a secure key exchange protocol.

2.2 Diffie-Hellman key exchange protocol

Unlike a public-key cryptosystem where the encryption key is known to all, in a symmetric cryptosystem the two communicating parties use the same secret key for encryption as well as decryption. However, they are only allowed to exchange this key by communicating through a public channel that is accessible to all. Diffie and Hellman [DH76] presented a scheme that makes this possible without compromising the security of the communication.

Let $\mathbb{F}_q$ be a finite field, where q is a large prime power. Let g be the generator of the multiplicative group $\mathbb{F}_q^\times$. Assume that the numbers q and g are known to all. The key-exchange protocol is as follows.

4 Diffie-Hellman key exchange

1. X chooses $a \in \mathbb{Z}_{q-1}$ uniformly at random, computes $x = g^a \in \mathbb{F}_q^\times$.
2. X sends x to Y.
3. Y chooses $b \in \mathbb{Z}_{q-1}$ uniformly at random, computes $y = g^b \in \mathbb{F}_q^\times$.
4. Y sends y to X.
5. X computes y^a, Y computes x^b.
6. X and Y use $y^a = x^b = g^{ab}$ as the secret key.

Since the communication uses a public channel, the numbers g^a and g^b are visible to all. If one can efficiently compute a from g and g^a, similarly b from g and g^b, one can also get the private key g^{ab}. The problem of computing z from g and g^z in $\mathbb{F}_q$ is the discrete logarithm problem. But just like integer factoring the current best algorithm [Gor93] for computing discrete logarithm has subexponential time complexity. It is not known though if breaking the Diffie-Hellman protocol is equivalent to computing discrete logarithm. For further details, the reader may refer to the article by Maurer and Wolf [MW99].

So far we have seen how the absence of efficient algorithms for certain number theoretic problems is exploited to design secure cryptographic protocols. Now we will see how fast algorithms for certain algebraic problems are used for designing efficient error-correcting codes.

2.3 The Reed-Solomon codes

The family of Reed-Solomon codes [RS60] is one of the most important and popular class of error-correcting codes which is used in many commercial applications, like encoding data on

CD and DVD, and in data transmission and broadcast systems. Error-correcting codes enable us to encode a message into a codeword that is tolerant to errors during transmission. At the receiver's end the possibly corrupted codeword is decoded back to the original message.

Fix a finite field $\mathbb{F}_q$ with $q \geq n$, the length of a codeword. The Reed-Solomon encoding procedure is as follows.

5 Reed-Solomon codes: Encoding

1. Let m be the message string that is to be encoded.
2. Split m as a sequence of $k < n$ elements of $\mathbb{F}_q$: $(m_0, \ldots, m_{k-1})$.
3. Let polynomial $P_m(x) = \sum_{i=0}^{k-1} m_i x^i$.
4. Let $c_j = P_m(e_j)$ for $0 \leq j < n$ where $e_0, \ldots, e_{n-1}$ are distinct elements of $\mathbb{F}_q$.
5. The sequence $(c_0, \ldots, c_{n-1})$ is the codeword corresponding to m.

Note the use of multipoint evaluation of the polynomial $P_m(x)$ in step 4 of the procedure.

Decoding of Reed-Solomon codes can be of two types - unique and list decoding. In list decoding, the codeword is decoded to a small set of strings so that the original message is one among these strings. Whereas in unique decoding the decoder outputs only the message string. We first discuss a version of the unique decoding algorithm due to Welch and Berlekamp [WB86].

Let $(d_0, \ldots, d_{n-1})$ be the received word which is possibly different from the codeword $(c_0, \ldots, c_{n-1})$. The following decoding algorithm finds the message polynomial $P_m(x)$ if the number of errors in $(d_0, \ldots, d_{n-1})$ is not too high.

6 Reed-Solomon codes: Unique decoding

1. Let $t = \lceil \frac{n-k+1}{2} - 1 \rceil$.
2. Find polynomials $M(x)$ and $F(x)$ such that,
   $\deg F(x) \leq t$, $F(x) \neq 0$;
   $\deg M(x) \leq k + t - 1$ and
   $M(e_i) = d_i \cdot F(e_i)$ for every $0 \leq i < n$.
3. Output $\frac{M(x)}{F(x)}$.

The correctness of the algorithm can be argued as follows. Let $\mathcal{E} \subset \{0, \ldots, n-1\}$ be the set of all positions in the received word $(d_0, \ldots, d_{n-1})$ which are in error i.e. $d_i \neq c_i$ for $i \in \mathcal{E}$. Assume that $|\mathcal{E}| \leq t$. Define the polynomials $E(x) \triangleq \prod_{i \in \mathcal{E}} (x - e_i)$ and $N(x) \triangleq P_m(x)E(x)$. It is easy to verify that $N(e_i) = d_i E(e_i)$ for all $0 \leq i < n$. Therefore, solutions for the polynomials $M(x)$ and $F(x)$ exist in step 2. Now notice that the polynomial $R(x) \triangleq N(x)F(x) - M(x)E(x)$ has degree at most $k + 2t - 1$ and $R(e_i) = 0$ for all $0 \leq i < n$. This implies that $R(x) = 0$ as $t < \frac{n-k+1}{2}$, and hence $\frac{M(x)}{F(x)} = \frac{N(x)}{E(x)} = P_m(x)$.

Step 2 can be implemented by solving a system of linear equations with the coefficients of $F(x)$ and $M(x)$ as unknowns. A preferable way of implementing this step is through rational function interpolation (for details see the online notes of Sudan [Sud] and Rosenblum [Ros]). In step 3 we can use polynomial division. Although very simple in nature, this algorithm does not achieve the best possible running time. The current fastest unique decoding algorithm is due to Justesen [Jus76].

The above decoding procedure can correct up to $\frac{n-k+1}{2}$ errors in the received word, and this error bound is the optimum as far as unique decoding is concerned. However, it turns out that

a lot more errors can be corrected if we allow the decoding procedure to output a small list of candidate message strings with the guarantee that the original message is one among them. The following is a simpler version of the list decoding algorithm given by Sudan [Sud97].

7 Reed-Solomon codes: List decoding

1. `Let` $u_0 = \lceil \sqrt{nk} \rceil$ `and` $u_1 = \lfloor \sqrt{\frac{n}{k}} \rfloor$.
2. `Find a nonzero bivariate polynomial` $Q(x, y)$ `with` x`-degree` u_0 `and` y`-degree` u_1 `such that` $Q(e_j, d_j) = 0$ `for every` $0 \leq j < n$.
3. `Factor` $Q(x, y)$.
4. `Output all polynomials` $P(x)$ `such that` $(y - P(x))$ `is a factor of` $Q(x, y)$.

We now show that the message polynomial $P_m(x)$ is in the output list of polynomials if $|\mathcal{E}| < n - 2\lceil \sqrt{nk} \rceil$ ($\mathcal{E}$ is the set of 'error' indices). In step 2 we can find $Q(x, y)$ by solving a system of linear equations with the coefficients of $Q(x, y)$ as variables. Since there are $(1+u_0)(1+u_1) > n$ unknowns with n equations to satisfy, there always exist a solution for $Q(x, y)$. Consider the polynomial $S(x) \triangleq Q(x, P_m(x))$. Degree of $S(x)$ is strictly less than $u_0 + u_1 k \leq 2\lceil \sqrt{nk} \rceil$. If $|\mathcal{E}| < n - 2\lceil \sqrt{nk} \rceil$ then from step 2 it follows that $S(e_j) = 0$ for at least $2\lceil \sqrt{nk} \rceil$ many distinct e_j's. This implies that $S(x) = 0$ and hence $(y - P_m(x))$ is a factor of $Q(x, y)$.

Another fundamental algebraic operation, namely polynomial factoring, is used in step 3 of the list decoding algorithm. In this case, it is bivariate polynomial factoring.

In practice, the value of k is much smaller than n and hence $n - 2\lceil \sqrt{nk} \rceil$ is much larger than $\frac{n-k+1}{2}$. The current best parameter is due to Guruswami and Sudan [GS99] that can correct up to $n - \sqrt{nk}$ errors, which is perhaps the best possible (see [BSKR06]). For an excellent exposition to the theory of error-correcting codes the reader may refer to the PhD thesis of Guruswami [Gur04] or the book by MacWilliams and Sloane [MS81].

At this stage we hope to have given the reader a glimpse of how algebraic and number theoretic operations are used in practice. In the following section we give a brief account of the time complexity of algorithms for some of these operations.

3 Complexity of basic operations

For the ease of presentation, we classify the basic algebraic and number theoretic operations into four broad categories:

1. **Polynomial arithmetic:** This includes polynomial addition, multiplication, division, computing gcd of polynomials, multipoint evaluation of a polynomial, polynomial interpolation, modular composition and polynomial factoring.
2. **Integer arithmetic:** This includes integer addition, multiplication, division, computing gcd, computing integer roots, primality testing and integer factoring.
3. **Linear algebra operations:** This includes matrix multiplication, computing determinant, inverse and characteristic polynomial of a matrix, and solving a system of linear equations.
4. **Abstract algebra operations:** This includes finding order of an element and computing discrete logarithm in a finite group.

We start with the operations under polynomial arithmetic.

3.1 Polynomial arithmetic

Let f and g be two degree n univariate polynomials over a field $\mathbb{F}$. In what follows, the time complexity of an algorithm measures the number of $\mathbb{F}$-operations performed by the algorithm.

Polynomial addition

This is simple addition of the coefficients of f and g, and hence it takes $O(n)$ operations in $\mathbb{F}$.

Polynomial multiplication

The naive approach of multiplying every coefficient of f with all coefficients of g takes $O(n^2)$ field operations. However, there is a trickier way of multiplying polynomials using a tool, known as Discrete Fourier Transform (DFT), that reduces the time complexity to $O(n \log n)$ $\mathbb{F}$-operations, if $\mathbb{F}$ contains a primitive n^{th} root of unity. We will discuss this algorithm in details in section 4.2. In case no such n^{th} root is present in $\mathbb{F}$, the time taken is only slightly worse which is $O(n \log n \log \log n)$ operations in $\mathbb{F}$ (see chapter 8 of [GG03]). We denote the time taken to multiply two polynomials of degree n by $\mathsf{M}(n)$. Note that, the function $\mathsf{M}(n)$ satisfies the property $\mathsf{M}(n_1 + n_2) \geq \mathsf{M}(n_1) + \mathsf{M}(n_2)$ for any integers $n_1, n_2 > 0$.

Polynomial division

The simple long division approach takes $O(n^2)$ time. But just like multiplication, there is a better way of dividing polynomials. Using a tool called Hensel lifting we can design a division algorithm that takes only $O(\mathsf{M}(n))$ operations. This algorithm will be discussed in section 4.4.

GCD of polynomials

The classical Euclidean algorithm computes $r = f \bmod g$ and then recursively computes the gcd of r and g. The extended Euclidean algorithm, given below, computes two polynomials s and t such that $sf + tg = gcd(f, g) = d$, with $\deg(s) < \deg(g)$ and $\deg(t) < \deg(f)$. We can use this algorithm to compute modular inverse. For instance, if f and g are relatively coprime then $s = f^{-1} \bmod g$.

8 Extended Euclidean algorithm

1. `Let` $(r, r') \leftarrow (f, g)$, $(s, s') \leftarrow (1, 0)$ `and` $(t, t') \leftarrow (0, 1)$.
2. `while` $r' \neq 0$ `do`
   $q \leftarrow \lfloor \frac{r}{r'} \rfloor$, $r'' \leftarrow r \mod r'$
   $(r, s, t, r', s', t') \leftarrow (r', s', t', r'', s - s'q, t - t'q)$
3. `Let` $d \leftarrow r$.
4. `Output` d, s, t.

It is not difficult to analyse this algorithm and show that the time complexity is bounded by $O(n^2)$ (refer to chapter 4 of [Sho09]). But then, there is also a faster version of the extended Euclidean algorithm that runs in only $O(\mathsf{M}(n) \log n)$ time (refer to chapter 11 of [GG03]).

Multipoint evaluation and interpolation

Let $f(x)$ be a given polynomial of degree less than $n = 2^k$. The multipoint evaluation problem is the task of evaluating f at n distinct points $u_0, \ldots, u_{n-1}$ of $\mathbb{F}$. Define the polynomials,

$p_j \triangleq (x - u_j)$ and $P_{i,j} \triangleq \prod_{\ell=0}^{2^i-1} p_{j \cdot 2^i + \ell}$ for $0 \leq i \leq k = \log n$ and $0 \leq j < 2^{k-i}$. At first compute all the $P_{i,j}$'s using the recursive equations,

$$P_{0,j} = p_j \text{ and } P_{i+1,j} = P_{i,2j} \cdot P_{i,2j+1}$$

Since $\deg(P_{i,j}) = 2^i$, it is easy to see that the complexity of the above computation is,

$$\begin{aligned} T(2^k) &= 2 \cdot T(2^{k-1}) + \mathsf{M}(2^{k-1}) \\ &= O(\mathsf{M}(n) \log n) \end{aligned}$$

The following procedure uses a divide and conquer strategy for multipoint evaluation.

9 Multipoint evaluation

1. `If` $n = 1$ `return` f.
2. `Let` $r_0 = f \mod P_{k-1,0}$ `and` $r_1 = f \mod P_{k-1,1}$.
3. `Recursively, evaluate` r_0 `at` $u_0, \ldots, u_{n/2-1}$.
4. `Recursively, evaluate` r_1 `at` $u_{n/2}, \ldots, u_{n-1}$.
5. `Output` $r_0(u_0), \ldots, r_0(u_{n/2-1}), r_1(u_{n/2}), \ldots, r_1(u_{n-1})$.

At deeper levels of the recursion, the procedure requires the values of the polynomials $P_{i,j}$'s, for $0 \leq i \leq k = \log n$ and $0 \leq j < 2^{k-i}$. Since all these polynomials are computed a priori, the time complexity of the procedure is,

$$\begin{aligned} T'(2^k) &= 2 \cdot T'(2^{k-1}) + O(\mathsf{M}(2^k)) \\ &= O(\mathsf{M}(n) \log n) \end{aligned}$$

considering the fact that division of degree n polynomials takes $O(\mathsf{M}(n))$ time.

Polynomial interpolation is the opposite of the multipoint evaluation problem. Given a set of n tuples $(u_0, v_0), \ldots, (u_{n-1}, v_{n-1})$ the task is to find a polynomial f of degree less than n such that $f(u_i) = v_i$ for all $0 \leq i < n$. Just like multipoint evaluation, there is a divide and conquer strategy for interpolating f using the Lagrange polynomial,

$$f = \sum_{i=0}^{n-1} \prod_{j=0, j \neq i}^{n-1} \frac{x - u_j}{u_i - u_j}$$

The reader might find it an interesting exercise to show that the time complexity of this problem is also $O(\mathsf{M}(n) \log n)$ (for details refer to chapter 10 of [GG03]).

Modular composition

Given three polynomials f, g and h with $\deg(g), \deg(h) < \deg(f) = n$, the task of computing $g(h) \mod f$ is known as modular composition. The following algorithm is due to Brent and Kung [BK78] and this particular exposition is present in chapter 12 of [GG03]. For simplicity, we assume that n is a perfect square with $m = \sqrt{n}$.

The correctness of the algorithm follows from the fact that $r_i = g_i(h) \bmod f$. Computing $h^i \bmod f$ for all $0 \leq i \leq m$ takes $O(m\mathsf{M}(n))$ time. Step 4 can be implemented as $\frac{n}{m}$, $m \times m$ matrix multiplications. This takes $O(m^{1+\omega})$ time, assuming ω to be the exponent for matrix multiplication (see section 3.3). Step 6 can also be implemented in $O(m\mathsf{M}(n))$ time by first

10 Modular composition

1. Let $g = \sum_{i=0}^{m-1} g_i x^{mi}$ where g_i is a polynomial with $\deg(g_i) < m$.
2. Let A be a $m \times n$ matrix with the coefficients of $h^i \mod f$ as the i^{th} row.
3. Let B be a $m \times m$ matrix with the coefficients of g_i as the i^{th} row.
4. Compute $C = B \cdot A$.
5. Let r_i be the polynomial with the i^{th} row of C as its coefficients.
6. Compute $r = \sum_{i=0}^{m-1} r_i (h^m)^i \mod f$ and output r.

computing $r'_{m-2} = (r_{m-1}h^m + r_{m-2}) \bmod f$ followed by $r'_{m-3} = (r'_{m-2}h^m + r_{m-3}) \bmod f$ and so on, until $r'_0 = r$ is computed. Therefore, the time complexity of the algorithm is $O(n^{\frac{1+\omega}{2}} + \sqrt{n} \cdot \mathsf{M}(n))$.

Very recently, Kedlaya and Umans [KU08] have improved this time complexity substantially. Their modular composition algorithm over a finite field uses merely $O(n^{1+o(1)})$ field operations, and this is optimum up to lower order terms.

Polynomial factoring

The problem of computing the irreducible factors of a given polynomial is one of the most fundamental and well-studied problem in algebraic computation. In case of polynomials over finite fields, the first randomized algorithm dates back to the work of Berlekamp [Ber70]. Several improvements in the running time came up subsequently due to Kaltofen and Shoup [KS98], von zur Gathen and Shoup [vzGS92], and others (refer to the survey by von zur Gathen and Panario [vzGP01]). The current best randomized algorithm was given very recently by Kedlaya and Umans [KU08]. Using a fast modular composition algorithm along with ideas from Kaltofen and Shoup [KS98], they achieved a running time of $\tilde{O}(n^{1.5} + n \log q)$ field operations, where $\mathbb{F}_q$ is the underlying finite field. Note that, this time complexity is indeed polynomial in the input size (which is about $n \log q$ bits), since a field operation takes $(\log q)^{O(1)}$ bit operations. In section 4.1 we will discuss a randomized polynomial time algorithm for factoring polynomials over finite fields.

The best deterministic algorithm so far, which is due to Evdokimov [Evd94], has quasi-polynomial time complexity under the assumption of the Extended Riemann Hypothesis (ERH). More precisely, it takes $(n^{\log n} \log q)^{O(1)}$ field operations. In section 4.3 we give an overview of this algorithm. Finding a deterministic polynomial time algorithm for factoring polynomials over finite fields is one of the major open problems in algebraic computation.

Unlike the case over finite fields, a deterministic polynomial time algorithm is known for factoring polynomials over rationals. Let $f \in \mathbb{Z}[x]$ be a polynomial with coefficients f_i for $0 \leq i \leq n$. The classical LLL factoring algorithm, given by Lenstra, Lenstra and Lovász [LJL82], has a time complexity of $\tilde{O}(n^{10} + n^8 \log^2 A)$ word operations, where $A = \max_{0 \leq i \leq n} \parallel f_i \parallel$. We will discuss this algorithm in section 4.5.

Algorithms for factoring multivariate polynomials have also been studied. In this regard, the reader may consult the celebrated result by Kaltofen [Kal85], who showed that multivariate polynomial factoring *reduces* to univariate polynomial factoring over any field. In another famous work, Kaltofen [Kal89] gave a randomized algorithm for black-box polynomial factoring. In section 4.4, we will show how bivariate polynomial factoring reduces to univariate factoring. The reduction technique generalizes to the case of multivariate polynomials.

The following table summarizes the results stated in this section.

Operations	Time complexity
Addition	$O(n)$
Multiplication	$\mathsf{M}(n) = O(n \log n \log\log n)$
Division	$O(\mathsf{M}(n))$
Gcd	$O(\mathsf{M}(n) \log n)$
Multipoint evaluation	$O(\mathsf{M}(n) \log n)$
Interpolation	$O(\mathsf{M}(n) \log n)$
Modular composition	$\tilde{O}(n)$ (over finite fields)
Factoring over finite fields	$\tilde{O}(n^{1.5} + n \log q)$ (randomized) $(n^{\log n} \log q)^{O(1)}$ (deterministic, under ERH)
Factoring over rationals	$\tilde{O}(n^{10} + n^8 \log^2 A)$

3.2 Integer arithmetic

Arithmetic operations with integers, like addition, multiplication, division and gcd are closely related to similar operations with polynomials. Often, the algorithms for the corresponding polynomial operations can be adapted to work for the case of integers as well. The cost of integer arithmetic is measured in terms of the number of bit operations.

Integer multiplication

The naive way of multiplying two n-bit integers, a and b, takes $O(n^2)$ bit operations. An alternative way is the following. Split the bit representation of each of a and b into m equal sized blocks. Consider a polynomial $A(x)$ of degree $m-1$ whose coefficients are the numbers formed by the blocks of a. Similarly consider polynomial $B(x)$ for b. Multiply $A(x)$ and $B(x)$ and evaluate the product at $2^{n/m}$ to get $a \cdot b$. Since polynomial multiplication can be done efficiently using Discrete Fourier Transform (DFT) (see section 4.2), one can hope to get a better algorithm for integer multiplication through this approach. Indeed, this idea was put to work first by Schönhage and Strassen [SS71], who gave an $O(n \log n \log\log n)$ algorithm for multiplying two n-bit integers. The main technical hurdle comes from the fact that, in order to use DFT we need a primitive root of unity in the base ring over which the polynomials are multiplied. Recently, Fürer [Für07] improved this algorithm further, using the novel idea of nested DFT's, to obtain a time complexity of $O(n \log n \cdot 2^{O(\log^* n)})$ bit operations. For a simpler exposition to Fürer's algorithm, the reader may refer to the work by De, Kurur, Saha and Saptharishi [DKSS08]. In the rest of this section, we will refer to the multiplication time of two n-bit integers by $\mathsf{M}_\mathsf{I}(n)$.

Other arithmetic operations

Algorithms for division and gcd computation over integers are very similar to that over polynomials, and it is not surprising that they have similar time complexity measures. For instance, division of two n-bit integers takes $O(\mathsf{M}_\mathsf{I}(n))$ bit operations, whereas gcd computation takes $O(\mathsf{M}_\mathsf{I}(n) \log n)$ bit operations. Another interesting problem is the computation of integer roots. Given an n-bit integer a and a number $m < n$, find an integer b (if it exists) such that $b^m = a$. It turns out, just as in polynomial division, Hensel lifting applies to this problem as well and yields a time complexity of $O(\mathsf{M}_\mathsf{I}(n))$ bit operations. For details refer to chapter 9 of [GG03].

Primality testing

Distinguishing primes from composite numbers is perhaps the most fundamental problem in algorithmic number theory. Till date, multiple randomized algorithms have been discovered to solve primality testing. In this section, we will discuss one such algorithm that is widely used in practice and is popularly known as the Miller-Rabin test [Mil76, Rab80]. It is a classic example of how randomization is used to design efficient algorithms in number theory.

Let $N > 0$ be an n-bit odd integer and $N - 1 = 2^t w$, where w is odd.

11 Miller-Rabin primality test

1. `Choose` a `randomly from the range` $[1, N-1]$.
2. `If` $\gcd(a, N) \neq 1$ `return 'composite'.`
3. `If` $a^{N-1} \neq 1 \mod N$ `return 'composite'.`
4. `If` $a^w = 1 \mod N$ `return 'probably prime'.`
5. `Else, let` $1 \leq r \leq t$ `be the smallest possible such that` $a^{2^r w} = 1 \mod N$.
6. `If` $a^{2^{r-1} w} \neq -1 \mod N$ `return 'composite'. Else return 'probably prime'.`

Correctness and success probability - First, it is easy to see that the algorithm always returns `'probably prime'` if N is a prime. The reason being, in step 3 $\gcd(a, N) = 1$ and hence from Fermat's little theorem $a^{N-1} = 1 \mod N$. Also in step 6, since $a^{2^r w} = 1 \mod N$ for the smallest possible $r \geq 1$, hence $a^{2^{r-1} w} = -1 \mod N$, $\mathbb{Z}_N$ being a field.

Let N be a composite. If the algorithm reaches step 3, we can assume that a has been chosen uniformly from $\mathbb{Z}_N^\times$, the set of positive integers coprime to and less than N. Now, if N is not a Carmichael number then the set of integers a such that $a^{N-1} = 1 \mod N$ is a strict subgroup of $\mathbb{Z}_N^\times$ under multiplication modulo N. Therefore, using Lagrange's theorem, the chance that $a^{N-1} \neq 1 \mod N$ is at least $\frac{1}{2}$.

Suppose N is a Carmichael number, which also means that N is square-free. Without loss of generality assume that $N = pq$, where p and q are distinct primes. By Chinese remaindering theorem (see section 4.1), $\mathbb{Z}_N^\times \cong \mathbb{F}_p^\times \oplus \mathbb{F}_q^\times$. Let $p - 1 = 2^k w_1$ and $q - 1 = 2^\ell w_2$, where w_1 and w_2 are odd. And suppose $a = \beta^{s_1} \mod p = \gamma^{s_2} \mod q$, where β and γ are generators of $\mathbb{F}_p^\times$ and $\mathbb{F}_q^\times$ respectively. In step 4, if $a^w = 1 \mod N$ then $\beta^{s_1 w} = 1 \mod p$ implying that $2^k | s_1$ as w is odd. Similarly, $2^\ell | s_2$ if $a^w = 1 \mod N$. Since a is randomly chosen from $\mathbb{Z}_N^\times$, equivalently s_1 and s_2 are chosen uniformly independently from the ranges $[1, p-1]$ and $[1, q-1]$ respectively. Therefore,

$$\begin{aligned} \Pr_a\{a^w = 1 \mod N\} &\leq \Pr_{s_1, s_2}\{2^k | s_1 \text{ and } 2^\ell | s_2\} \\ &= \Pr_{s_1}\{2^k | s_1\} \cdot \Pr_{s_2}\{2^\ell | s_2\} = \frac{1}{2^{k+\ell}}. \end{aligned}$$

Suppose in step 6, $a^{2^{r-1} w} = -1 \mod N$. Then $\beta^{s_1 2^{r-1} w} = -1 \mod p$ implying that $2^{k-r} \| s_1$ i.e. $k - r$ is the highest power of 2 that divides s_1. Similarly, $2^{\ell - r} \| s_2$. For a fixed $r \leq \min\{k, \ell\}$,

$$\Pr_{s_1, s_2}\{2^{k-r} \| s_1 \text{ and } 2^{\ell - r} \| s_2\} = \frac{1}{2^{k+\ell-2(r-1)}}.$$

By union bound, over all $1 \leq r \leq \min\{k, \ell\} = k$ (say),

$$\Pr_a\{\exists r, 1 \leq r \leq t \text{ such that } a^{2^{r-1} w} = -1 \mod N\} \leq \sum_{r=1}^{k} \frac{1}{2^{k+\ell-2(r-1)}}$$

Summing the error probabilities from step 4 and 6 we conclude that Miller-Rabin test succeeds with probability at least $1 - \frac{1}{2^{k+\ell}}(\frac{4^k+2}{3}) \geq \frac{1}{2}$.

Time complexity - Gcd computation in step 2 takes $O(\mathsf{M_I}(n) \log n)$ time. In step 3 we can use repeated squaring and compute $a_1 = a^2 \bmod N$, $a_2 = a_1^2 = a^4 \bmod N$, $a_3 = a_2^2 = a^8 \bmod N$ and so on till $a_{\lfloor \log(N-1) \rfloor}$. Then, we can multiply all those a_i's modulo N for which the i^{th} bit of $N-1$ in binary is 1. This process takes $O(\mathsf{M_I}(n) \log N) = O(n\mathsf{M_I}(n))$ time. The complexity of steps $4, 5$ and 6 are similarly bounded by $O(n\mathsf{M_I}(n))$ as $r \leq t \leq \log N \leq n$. Therefore, the overall time complexity of the Miller-Rabin test is $O(n\mathsf{M_I}(n)) = \tilde{O}(n^2)$ bit operations.

Deterministic primality test - In a major breakthrough, the first deterministic primality testing algorithm was given by Agrawal, Kayal and Saxena [AKS04] in 2002. It is famously known as the AKS primality test. In section 4.3 we will present this algorithm. The current best deterministic complexity is due to a version of the AKS-primality test given by Lenstra and Pomerance [JP05]. Their algorithm has a running time of $\tilde{O}(n^6)$ bit operations.

Integer factoring

Factoring an integer N is closely related to primality testing. However, as mentioned in section 2.1, this is a long standing open problem and as yet no randomized polynomial time algorithm is known. The current best randomized algorithm, due to Lenstra and Pomerance [JP92], has an expected running time of $e^{(1+o(1))\sqrt{\ln N \ln\ln N}}$ bit operations. This bound is an improvement on the complexity of *Dixon's algorithm* [Dix81], which also has an expected subexponential running time of the form $e^{O(\sqrt{\ln N \ln\ln N})}$. We will take up Dixon's algorithm in section 4.6. The best deterministic factoring algorithm, known as Pollard-Strassen method, runs in $\tilde{O}(N^{\frac{1}{4}})$ time.

Two important techniques are used to factor integers much more efficiently in practice than the algorithms mentioned above. They are the *Quadratic Sieve* [Pom84] and the *Number Field Sieve* [LJMP90]. A heuristic analysis of the Quadratic Sieve (QS) yields a time bound of $e^{(1+o(1))\sqrt{\ln N \ln\ln N}}$ operations. We include a discussion on this analysis in section 4.6. The Number Field Sieve (NFS) has a heuristic time complexity of $e^{(c+o(1))((\ln N)^{\frac{1}{3}}(\ln\ln N)^{\frac{2}{3}})}$ bit operations, for a constant $c > 0$. The best known value for c is 1.902, due to Coppersmith [Cop93]. For an exposition to NFS and some other related factoring algorithms refer to the survey by Lenstra [Len00] and the article by Stevenhagen [Ste08] in [BS08]. Although, the NFS is asymptotically faster than the QS, it has been noticed that the latter performs better in practice for numbers up to 100 digits (see [Pom96]).

The following table summarizes the complexity of different integer arithmetic operations.

Operations	**Time complexity**
Addition	$O(n)$
Multiplication	$\mathsf{M_I}(n) = O(n \log n \cdot 2^{O(\log^* n)})$
Division	$O(\mathsf{M_I}(n))$
Gcd	$O(\mathsf{M_I}(n) \log n)$
Finding integer roots	$O(\mathsf{M_I}(n))$
Primality testing	$\tilde{O}(n^2)$ (randomized) $\tilde{O}(n^6)$ (deterministic)
Integer factoring	$e^{(1+o(1))\sqrt{\ln N \ln\ln N}}$ (randomized) $e^{O((\ln N)^{\frac{1}{3}}(\ln\ln N)^{\frac{2}{3}})}$ (heuristic)

3.3 Linear algebra operations

Like polynomial and integer arithmetic, matrix operations are also of fundamental nature and are frequently used in designing algorithms. In section 3.1, we saw an application of matrix multiplication in the modular composition algorithm (Algorithm 10). In this section, we will note the time complexity of some of the basic linear algebra operations.

Matrix multiplication

The naive approach of multiplying the rows of an $n \times n$ matrix with the columns of another $n \times n$ matrix takes $O(n^3)$ time. Improving upon the exponent $\omega = 3$, Strassen [Str69] gave an algorithm that runs in $O(n^{\log 7}) \approx O(n^{2.807})$ time. The current best exponent is due to Coppersmith and Winograd [CW87]. Their algorithm has $O(n^{2.376})$ time complexity. It remains open as to whether this can be improved further to $O(n^2)$.

For a detailed account of the complexity of matrix multiplication, refer to chapter 15 of the book by Bürgisser, Clausen and Shokrollahi [BCS97].

Other linear algebra operations

One reason as to why matrix multiplication is so fundamental is that, a whole bunch of other problems like LUP decomposition, computing determinant, finding inverse and characteristic polynomial of an $n \times n$ matrix, and solving a system of n linear equations in n variables reduce to matrix multiplication. For details of these reductions the reader may refer to chapter 6 of the book by Aho, Hopcroft and Ullman [AHU74] or chapter 16 of the book by Bürgisser, Clausen and Shokrollahi [BCS97].

On some occasions the matrix under consideration is fairly sparse, meaning there are very few non-zero entries in the matrix. Solving a system of linear equations with a sparse coefficient matrix is one such example. Faster algorithms exist for solving sparse linear systems. For details the reader may refer to the work of Wiedemann [Wie86] and Kaltofen and Saunders [KS91].

The following table gives a summary of the basic linear algebra operations. We denote the time taken to multiply two $n \times n$ matrices by $\mathsf{M_m}(n)$.

Operations	**Time complexity**
Matrix multiplication	$\mathsf{M_m}(n) = O(n^{2.376})$
Inverse	$O(\mathsf{M_m}(n))$
Determinant	$O(\mathsf{M_m}(n))$
Finding the characteristic polynomial	$O(\mathsf{M_m}(n))$
Solving a linear system	$O(\mathsf{M_m}(n))$

3.4 Abstract algebra operations

Finding order of an element

Given an element a in a finite group, how hard is it to find its order? For groups like $(\mathbb{Z}_n, +)$ the answer is pretty easy. Find $b = \gcd(a, n)$, the order of a is exactly $\frac{n}{b}$. However, the answer need not be so simple for any arbitrary group. For instance, take the group $G = (\mathbb{Z}_n^\times, \cdot)$, where $n = pq$, p and q are primes. Choose a randomly from G and let m be the order of a. Using an argument similar to that in the analysis of the Miller-Rabin primality test (see Algorithm 11) it can be shown that m is even and $a^{\frac{m}{2}} + 1 \neq 0 \bmod n$ with probability at least $\frac{1}{2}$. Since $a^m - 1 = 0$, hence $(a^{\frac{m}{2}} - 1)(a^{\frac{m}{2}} + 1) = 0 \bmod n$ implying that $\gcd(a^{\frac{m}{2}} + 1, n)$ is non-trivial.

Thus order finding over groups of the form $(\mathbb{Z}_n^\times, \cdot)$, where n is composite, is at least as hard as factoring n. In fact, the two problems are randomly polynomial time equivalent. The reader is encouraged to argue the other direction.

An interesting open problem, mentioned by Adleman and McCurley [AM94], is the following - Is the problem of completely factoring numbers of the form $p-1$, p is a prime, polynomial time reducible to finding order of an element in $\mathbb{F}_p^\times$?

Computing discrete logarithm

The discrete logarithm problem is the following - Given a finite group $(G, \cdot)$ and two elements a and b in G find an $x \in \mathbb{N}$ such that $a^x = b$, if such an x exists. We have already seen an application of the conjectured hardness of this problem in section 2.2. Like the order finding problem, the hardness of computing discrete log depends on the choice of the underlying group. For instance, if the underlying group is $(\mathbb{Z}_n, +)$ then the value of x is precisely $ba^{-1} \bmod n$, assuming $\gcd(a, n) = 1$. On the other hand, a polynomial time algorithm for solving discrete log over $(\mathbb{Z}_n^\times, \cdot)$ for a composite n, yields a randomized polynomial time algorithm for factoring n. Once again, as an exercise the reader is encouraged to show this implication.

In the cryptanalysis of the Diffie-Hellman protocol, we are interested in the hardness of the discrete log problem for the group $\mathbb{F}_p^\times$. For certain choices of p, this problem turns out to be easy. Pohlig and Hellman [PH78] showed that if all the factors of $p-1$ have values bounded by $\log^c p$ for a constant c, then the problem can be solved in polynomial time. Therefore, such choices of p should be avoided for the security of the protocol.

The current best randomized algorithm for the discrete log problem over $\mathbb{F}_p^\times$ is due to Pomerance [Pom87]. It has an expected running time of $e^{(\sqrt{2}+o(1))\sqrt{\ln p \ln\ln p}}$ bit operations. The technique used in [Pom87] is more generally known as the *index calculus method* and was used earlier by Adleman [Adl79] to obtain a subexponential discrete log algorithm having similar complexity. In section 4.6 we include a discussion on index calculus method. Using an adaptation of the Number Field Sieve, Gordon [Gor93] proposed a heuristic algorithm showing a running time of $e^{(c+o(1))(\ln p)^{\frac{1}{3}}(\ln\ln p)^{\frac{2}{3}}}$ operations, where $c > 0$ is a constant. The best known value of c is 1.92. So far the best deterministic algorithms have $\tilde{O}(\sqrt{p})$ complexity (see [Pol78]). Despite the resemblance between the time complexity of integer factorization and discrete logarithm over $\mathbb{F}_p^\times$, so far no reduction is known from one problem to the other. In fact, it was asked by Bach [Bac84] whether factoring $p-1$ reduces to finding discrete logarithm in $\mathbb{F}_p^\times$.

There are certain groups for which computing discrete logarithm is even harder, in the sense that no randomized subexponential time algorithm is known. Such an example is the group of points on an elliptic curve. In this case the precise question is the following - Given an elliptic curve $E_{p,a,b} = \{(x, y) \mid x, y \in \mathbb{F}_p \text{ and } y^2 = x^3 + ax + b \pmod p\} \cup \{0\}$, with $\gcd(p, 4a^3 + 27b^2) = 1$ and two points P and Q on $E_{p,a,b}$, find an n such that $P = nQ$. Here, $nQ \triangleq Q + Q + \ldots$ n times, where $+$ is the operation that endows $E_{p,a,b}$ with a group structure. The hardness of this problem has also been exploited in designing public key cryptosystems and digital signature schemes (refer to [Kob87]).

For further details on discrete logarithm refer to the surveys by Odlyzko [Odl00] and McCurley [McC90], or the articles by Pomerance [Pom08a] and Schirokauer [Sch08] in [BS08].

This brings us to the end of this section. We hope to have given the reader an overview of the algorithmic complexity of some of the basic algebraic and number theoretic operations. In the following section, we will focus on some specific mathematical tools that are used to design algorithms for these operations.

4 Tools for designing algorithms for basic operations

Often, the design and analysis of algorithms for the basic operations involve other fundamental mathematical results or *tools*. This section is devoted to a brief study of some such tools, namely - Chinese remaindering theorem, discrete Fourier transform, automorphisms, Hensel lifting, short vectors in lattices and smooth numbers. After describing each tool, we will show its application by taking some examples.

4.1 Chinese remaindering theorem

This theorem is a structural result about rings which is used for speeding up computation over integers and polynomials, and also for arguing over rings like in the analysis of the Miller-Rabin primality test (Algorithm 11). For convenience, we present the theorem in a general form and then apply it to the rings of integers and polynomials.

Two ideals $\mathcal{I}$ and $\mathcal{J}$ of a ring $\mathcal{R}$ are *coprime* if there are elements $a \in \mathcal{I}$ and $b \in \mathcal{J}$ such that $a + b = 1$. The product of two ideals $\mathcal{I}$ and $\mathcal{J}$, denoted by $\mathcal{I}\mathcal{J}$, is the ideal generated by all elements of the form $a \cdot b$ where $a \in \mathcal{I}$ and $b \in \mathcal{J}$. The theorem states the following.

Theorem 4.1 (Chinese Remaindering Theorem) *Let $\mathcal{I}_1, \ldots, \mathcal{I}_r$ be pairwise coprime ideals of $\mathcal{R}$ and $\mathcal{I} = \mathcal{I}_1 \ldots \mathcal{I}_r$ be their product. Then,*

$$\frac{\mathcal{R}}{\mathcal{I}} \cong \frac{\mathcal{R}}{\mathcal{I}_1} \oplus \ldots \oplus \frac{\mathcal{R}}{\mathcal{I}_r}$$

Moreover, this isomorphism map is given by,

$$a \bmod \mathcal{I} \longrightarrow (a \bmod \mathcal{I}_1, \ldots, a \bmod \mathcal{I}_r)$$

for all $a \in \mathcal{R}$.

Proof: The proof uses induction on the number of coprime ideals. Let $\mathcal{J} = \mathcal{I}_2 \ldots \mathcal{I}_r$. Since $\mathcal{I}_1$ is coprime to $\mathcal{I}_j$ for every j, $2 \leq j \leq r$, there are elements $y_j \in \mathcal{I}_j$ and $x_j \in \mathcal{I}_1$ such that $x_j + y_j = 1$. Therefore, $\prod_{j=2}^{r} (x_j + y_j) = x + y' = 1$ where $x \in \mathcal{I}_1$ and $y' \in \mathcal{J}$, implying that $\mathcal{I}$ and $\mathcal{J}$ are coprime.

We claim that $\mathcal{I} = \mathcal{I}_1 \cap \mathcal{J}$. By definition, $\mathcal{I} = \mathcal{I}_1\mathcal{J}$ and it is easy to see that $\mathcal{I}_1\mathcal{J} \subseteq \mathcal{I}_1 \cap \mathcal{J}$. If $z \in \mathcal{I}_1 \cap \mathcal{J}$ then, from $x + y' = 1$ we have $zx + zy' = z$. The left hand side of the last equation is an element of $\mathcal{I}_1\mathcal{J}$ as both $zx, zy' \in \mathcal{I}_1\mathcal{J}$. Therefore, $\mathcal{I}_1 \cap \mathcal{J} = \mathcal{I}_1\mathcal{J} = \mathcal{I}$.

Consider the map $\phi : \frac{\mathcal{R}}{\mathcal{I}} \rightarrow \frac{\mathcal{R}}{\mathcal{I}_1} \oplus \frac{\mathcal{R}}{\mathcal{J}}$ defined as $\phi(a \bmod \mathcal{I}) = (a \bmod \mathcal{I}_1, a \bmod \mathcal{J})$. It is easy to check that ϕ is well-defined and is in fact a homomorphism. Let $a_1 = a \bmod \mathcal{I}_1$ and $a' = a \bmod \mathcal{J}$. We will abuse notation slightly and write $\phi(a) = (a_1, a')$.

If $\phi(a) = \phi(b) = (a_1, a')$ then $a_1 = a \bmod \mathcal{I}_1 = b \bmod \mathcal{I}_1$, implying that $a - b \in \mathcal{I}_1$. Similarly, $a - b \in \mathcal{J}$. This means $a - b \in \mathcal{I} \cap \mathcal{J} = \mathcal{I}$ and hence ϕ is a one-one map. Also, since $x + y' = 1$ for $x \in \mathcal{I}_1$ and $y' \in \mathcal{J}$, we have $\phi(a_1y' + a'x) = (a_1, a')$ implying that ϕ is onto. Therefore, ϕ is an isomorphism i.e. $\frac{\mathcal{R}}{\mathcal{I}} \cong \frac{\mathcal{R}}{\mathcal{I}_1} \oplus \frac{\mathcal{R}}{\mathcal{J}}$. Inductively, we can show that $\frac{\mathcal{R}}{\mathcal{J}} \cong \frac{\mathcal{R}}{\mathcal{I}_2} \oplus \ldots \oplus \frac{\mathcal{R}}{\mathcal{I}_r}$ and hence, $\frac{\mathcal{R}}{\mathcal{I}} \cong \frac{\mathcal{R}}{\mathcal{I}_1} \oplus \ldots \oplus \frac{\mathcal{R}}{\mathcal{I}_r}$. ∎

In $\mathbb{Z}$ (or $\mathbb{F}[x]$), two elements m_1 and m_2 are coprime integers (or polynomials) if and only if the ideals (m_1) and (m_2) are coprime. Applying the above theorem to the ring of integers (or polynomials) we get the following result.

Corollary 4.1 *Let $m \in \mathcal{R} = \mathbb{Z}$ (or $\mathbb{F}[x]$) be such that $m = \prod_{j=1}^{r} m_j$ where $m_1, \ldots, m_r$ are pairwise coprime integers (or polynomials). Then $\frac{\mathcal{R}}{(m)} \cong \frac{\mathcal{R}}{(m_1)} \oplus \ldots \oplus \frac{\mathcal{R}}{(m_r)}$.*

Thus every element of the ring $\frac{\mathcal{R}}{(m)}$ can be uniquely written as an r-tuple with the i^{th} component belonging to the ring $\frac{\mathcal{R}}{(m_i)}$. Addition and multiplication in $\frac{\mathcal{R}}{(m)}$ amounts to component-wise addition and multiplication in the rings $\frac{\mathcal{R}}{(m_i)}$. This suggests a strategy to speed up computation based on the fact that it is faster to compute modulo a small integer (or a small degree polynomial) than over integers (or polynomial ring).

- Given a bound, say A, on the output of a computation, choose small $m_1, \ldots, m_r$ such that $\prod_{i=1}^{r} m_i > A$ and do the computations modulo each m_i.
- At the end, combine the results of computations to get the desired result.

The following application shows this idea at work.

Application 1: Determinant computation

Suppose we want to compute the determinant of an $n \times n$ matrix M over integers. We can use Gaussian elimination, but it takes some effort to show that the sizes of the numerators and denominators of the rational numbers appearing at intermediate stages of the elimination are polynomially bounded in the input size. Alternatively, we can use a faster way for computing the determinant using Chinese remaindering. We say it is faster because Chinese remaindering is inherently parallelizable and each computation happens over a small modulus.

Let A be the bound on the largest absolute value of the elements of M. Hadamard's inequality gives a bound on the absolute value of $\det(M)$ in terms of n and A.

Lemma 4.1 (Hadamard's Inequality) $|\det(M)| \leq n^{\frac{n}{2}} A^n$.

Proof: Let $v_1, \ldots, v_n$ be the row vectors of M. Assuming that $v_1, \ldots, v_n$ are linearly independent we can find an orthogonal basis $v_1^*, \ldots, v_n^*$, using Gram-Schimdt orthogonalization (see Appendix A.2), such that $\|v_i^*\| \leq \|v_i\|$ for all i, $1 \leq i \leq n$. Here, $\|v\|$ denotes the 2-norm of vector v. It follows from the properties of this orthogonal basis that,

$$\det(M)^2 = \det\left(M \cdot M^T\right) = \prod_{i=1}^{n} \|v_i^*\|^2 \leq \prod_{i=1}^{n} \|v_i\|^2 \leq n^n A^{2n}.$$

■

We use this bound in the following algorithm.

12 Computing determinant using Chinese remaindering

1. `Let` $B = n^{\frac{n}{2}} A^n$ `and` $r = \lceil \log(2B+1) \rceil$.
2. `Let` $m_1, \ldots, m_r$ `be the first` r `primes and` $m = \prod_{i=1}^{r} m_i$.
3. `Compute` $a_i = \det(M) \mod m_i$ `for each` i.
4. `Compute` α_i `such that` $\alpha_i \cdot \frac{m}{m_i} = 1 \mod m_i$ `for each` i.
5. `Compute` $d = \sum_{i=1}^{r} \alpha_i \cdot \frac{m}{m_i} \cdot a_i \mod m$.
6. `If` $d \leq B$ `return` d, `else return` $d - m$.

Correctness and time complexity - First note that $m > 2^r > 2B$. Step 4 succeeds in finding an α_i as $\gcd(\frac{m}{m_i}, m_i) = 1$. From the choice of α_i it is clear that $d = a_i \bmod m_i$, for all i. By Chinese remaindering theorem, $d = \det(M) \bmod m$. We know that $|\det(M)| \leq B < m$. Therefore, if $d \leq B$ then $\det(M) = d$, otherwise $\det(M) = d - m$. Since the r^{th} prime has value about $r \log r$, the entire computation is polynomial time bounded.

Application 2: Polynomial factoring

We use the Chinese remaindering theorem to show that polynomial factoring reduces to the problem of root finding over a finite field.

Let $f \in \mathbb{F}_q[x]$ be a polynomial of degree n that factors as $f = f_1 \dots f_k$, where f_i's are irreducible polynomials over $\mathbb{F}_q$. We can assume that f is square-free or else we can take gcd of f and $\frac{df}{dx}$, the formal derivative of f with respect to x, and find nontrivial factors of f. The process can be continued until we are left with only square-free polynomials to factor. It is easy to see that this step, also known as *square-free factorization*, takes polynomial in n field operations.

Suppose that the irreducible factors of f are not of the same degree. Then, there are two factors, say, f_1 and f_2 such that $\deg(f_1) = d_1$ is minimum and $\deg(f_2) = d_2 > d_1$. Since f_1 and f_2 are irreducible, f_1 divides the polynomial $x^{q^{d_1}} - x$ whereas f_2 does not. Therefore, $\gcd(x^{q^{d_1}} - x, f)$ yields a non-trivial factor of f. As d_1 is unknown, we iteratively compute $x^{q^t} - x$ starting from $t = 1$ till we hit $t = d_1$. This can be done by using the repeated squaring method (like the one discussed in the time complexity analysis of the Miller-Rabin test) to compute $x^{q^t} \bmod f$. This step, known as *distinct-degree factorization*, takes $(n \log q)^{O(1)}$ field operations.

We are now left with the task of factoring a square-free polynomial $f = f_1 \dots f_k$ such that all the irreducible factors f_i's have the same degree, say, d. This step is called *equal-degree factorization*. By Chinese remaindering theorem,

$$\mathcal{R} = \frac{\mathbb{F}_q[x]}{(f)} \cong \oplus_{i=1}^{k} \frac{\mathbb{F}_q[x]}{(f_i)} \cong \oplus_{i=1}^{k} \mathbb{F}_{q^d}.$$

Let $g \in \mathcal{R} \backslash \mathbb{F}_q$ be such that $g^q = g$ in $\mathcal{R}$. First, let us show that such a g exists in $\mathcal{R}$. By the above isomorphism, any g whose direct-sum representation belongs to $\oplus_{i=1}^{k} \mathbb{F}_q$ satisfies the condition $g^q = g \bmod f$. Also, if f is not irreducible then such a $g \notin \mathbb{F}_q$ exists. This means, there exists $c_i, c_j \in \mathbb{F}_q$ $(i \neq j)$ such that $c_i = g \bmod f_i$, $c_j = g \bmod f_j$ and $c_i \neq c_j$. This also implies that there is a $c \in \mathbb{F}_q$ such that $\gcd(g - c, f)$ yields a non-trivial factor of f. For instance, for $c = c_i$, f_i divides $\gcd(g - c, f)$ but f_j does not.

To compute g, start with a generic element $\sum_{i=0}^{n-1} a_i x^i \in \mathcal{R}$, where $n = \deg(f)$ and a_i's are variables, and solve for $a_i \in \mathbb{F}_q$ such that $\sum_{i=0}^{n-1} a_i x^{qi} = \sum_{i=0}^{n-1} a_i x^i \bmod f$. Solving this equation reduces to solving a system of linear equations in the a_i's. This reduction follows once we compute x^{qi} rem f for all i and equate the coefficients of x^j, for $0 \leq j \leq n-1$, from both sides of the equation. Now all we need to do, while solving the linear equations, is to choose a solution for the a_i's such that $\sum_{i=0}^{n-1} a_i x^i \notin \mathbb{F}_q$. Take $g = \sum_{i=0}^{n-1} a_i x^i$ for that choice of a_i's. Taking into account that x^{qi} rem f can be computed using repeated squaring, we conclude that g can be found in polynomial time.

The only task that remains is to find a $c \in \mathbb{F}_q$ such that $\gcd(g - c, f)$ gives a nontrivial factor of f. This is where the problem gets reduced to root finding. The fact that $\gcd(g - c, f) \neq 1$ means resultant of the polynomials $g - c = \sum_{i=1}^{n-1} a_i x^i + (a_0 - c)$ and f is zero. This means, we need to solve for a $y \in \mathbb{F}_q$ such that $h(y) = \mathsf{Res}_x(\sum_{i=1}^{n-1} a_i x^i + (a_0 - y), f) = 0$, treating $a_0 - y$ as the constant term of the polynomial $g - y$. We can find h by computing the determinant of the Sylvester matrix, $S(\sum_{i=1}^{n-1} a_i x^i + (a_0 - y), f)$, of $g - y$ and f. Although there are entries in S containing variable y, we can find $\det(S)$ in polynomial time using interpolation. In this way, factoring polynomial $f(x)$ reduces to finding a root of the polynomial $h(y)$.

It follows from our discussion that, apart from the root finding part in step 3 of Algorithm 13 all the other steps run in polynomial time.

13 Equal-degree factoring to root finding

1. Using linear algebra, find a $g \in \mathbb{F}_q[x]$ such that $g^q = g \mod f$ and $g \notin \mathbb{F}_q$.
2. If no such g exists then declare 'f is irreducible'.
3. Compute polynomial $h(y) = \mathsf{Res}_x(g - y, f)$ and find a root c of $h(y)$.
4. Find a nontrivial factor $f' = \gcd(g - c, f)$ of f.
5. Repeat the above process to factor f' and $\frac{f}{f'}$.

We have already mentioned in section 3.1 that no deterministic polynomial time algorithm is known for factoring polynomials over finite fields. This means, by the above reduction, no polynomial time algorithm is known for root finding as well. However, there is a relatively simple randomized root finding algorithm that runs in polynomial time. For the sake of completion, we present the algorithm here although this does not involve Chinese remaindering as such.

Suppose, we want to find a root of the polynomial $h(y)$. At first, compute the polynomial $\tilde{h} = \gcd(y^q - y, h)$ which splits completely over $\mathbb{F}_q$ into linear factors. Any root of $\tilde{h}$ is also a root of h and vice versa, and so the problem reduces to finding a root of $\tilde{h}$. Once again, we would like to note that $\gcd(y^q - y, h)$ is computed by first computing $y^q \bmod h$ using repeated squaring.

Let $\tilde{h} = \prod_{i=1}^{m} (y - c_i)$ with $c_i \in \mathbb{F}_q$ for all i. If $\tilde{h}$ is linear then return the root. Otherwise, there are two distinct roots, say, c_1 and c_2 of $\tilde{h}$. Choose r randomly from $\mathbb{F}_q^\times$ and compute $\tilde{h}(y^2 - r) = \prod_{i=1}^{m} (y^2 - (c_i + r))$. Since the polynomial $(c_1 + x)^{\frac{q-1}{2}} - (c_2 + x)^{\frac{q-1}{2}}$ can have at most $\frac{q-1}{2}$ roots, the probability that $c_1 + r$ and $c_2 + r$ are both quadratic residues or both quadratic non-residues in $\mathbb{F}_q$ is at most $\frac{1}{2}$. Suppose $c_1 + r$ is a quadratic residue whereas $c_2 + r$ is a quadratic non-residue and let $\hat{h} = \gcd(y^q - y, \tilde{h}(y^2 - r))$. Then, $(y^2 - (c_1 + r)) = (y - \sqrt{c_1 + r})(y + \sqrt{c_1 + r})$ divides $\hat{h}$ but $(y^2 - (c_2 + r))$ does not. In fact, $\hat{h} = \prod_{c'} (y^2 - (c' + r))$ for all roots c' of $\tilde{h}$ such that $c' + r$ is a quadratic residue. Hence, by letting $h'(y) = \hat{h}(\sqrt{y}) = \prod_{c'} (y - (c' + r))$ we get $h'(y + r)$ as a proper factor of $\tilde{h}$.

14 Root finding

1. Given a polynomial $h(y)$ compute $\tilde{h}(y) = \gcd(y^q - y, h)$.
2. If $\deg(\tilde{h}) = 1$ return the root of $\tilde{h}$. /* Assuming $\deg(\tilde{h}) \neq 0$. */
3. Else, choose r randomly from $\mathbb{F}_q$.
4. Compute polynomials $\tilde{h}(y^2 - r)$ and $\hat{h}(y) = \gcd(y^q - y, \tilde{h}(y^2 - r))$.
5. Find $h'(y) = \hat{h}(\sqrt{y})$ and obtain $h'(y + r)$.
6. Repeat the above process from step 2 to factor $h'(y + r)$ and $\frac{\tilde{h}}{h'(y+r)}$.

Verify that the algorithm runs in $(n \log q)^{O(1)}$ time and $h'(y + r)$ is a proper factor of $\tilde{h}$ with probability at least $\frac{1}{2}$.

In fact, root finding over $\mathbb{F}_q$ reduces to root finding over $\mathbb{F}_p$, where p is the characteristic of the field $\mathbb{F}_q$. This was shown by Berlekamp [Ber67, Ber70]. For details, refer to chapter 4 of the book by Lidl and Neiderreiter [LN94].

4.2 Discrete Fourier Transform

This tool has been extensively used to design fast algorithms for polynomial and integer multiplication. In this section, we will see a polynomial multiplication algorithm based on Discrete

Fourier Transform (DFT). Multiplication of two polynomials is essentially a convolution of their coefficient vectors. Roughly speaking, a DFT transforms each of the coefficient vectors into another vector so that the convolution of the two original coefficient vectors is equivalent to pointwise multiplication of the two transformed vectors followed by an inverse-DFT of the resulting product vector. The following discussion makes this point clear.

Suppose $f : [0, n-1] \to \mathbb{F}$ be a function 'selecting' n elements of the field $\mathbb{F}$. And let ω be a primitive n^{th} root of unity in $\mathbb{F}$ i.e. $\omega^n = 1$ and $\omega^t \neq 1$ for $0 < t < n$. Then the Discrete Fourier Transform (DFT) of f is defined to be the map,

$$\begin{aligned} \mathcal{F}_f : [0, n-1] &\to \mathbb{F} \quad \text{given by} \\ \mathcal{F}_f(j) &= \sum_{i=0}^{n-1} f(i)\omega^{ij}. \end{aligned}$$

Computing the DFT of f is the task of finding the vector $(\mathcal{F}_f(0), \ldots, \mathcal{F}_f(n-1))$. This task can be performed efficiently using an algorithm called the Fast Fourier Transform (FFT), which was first found by Gauss in 1805 and later (re)discovered by Cooley and Tukey [CT65]. The algorithm uses a divide and conquer technique to compute DFT of a function f, with domain size n, using $O(n \log n)$ field operations. For simplicity, we will assume that n is a power of 2.

15 Fast Fourier Transform

1. `If` $n = 1$ `return` $f(0)$.
2. `Define` $f_0 : [0, \frac{n}{2} - 1] \to \mathbb{F}$ `as` $f_0(i) = f(i) + f(\frac{n}{2} + i)$.
3. `Define` $f_1 : [0, \frac{n}{2} - 1] \to \mathbb{F}$ `as` $f_1(i) = (f(i) - f(\frac{n}{2} + i))\omega^i$.
4. `Recursively, compute DFT of` f_0 `i.e.` $\mathcal{F}_{f_0}$ `using` ω^2 `as the root of unity.`
5. `Recursively, compute DFT of` f_1 `i.e.` $\mathcal{F}_{f_1}$ `using` ω^2 `as the root of unity.`
6. `Return` $\mathcal{F}_f(2j) = \mathcal{F}_{f_0}(j)$ `and` $\mathcal{F}_f(2j+1) = \mathcal{F}_{f_1}(j)$ `for all` $0 \leq j < \frac{n}{2}$.

Correctness of the algorithm - This is immediate from the following two observations,

$$\begin{aligned} \mathcal{F}_f(2j) &= \sum_{i=0}^{n-1} f(i)\omega^{2ij} = \sum_{i=0}^{\frac{n}{2}-1} (f(i) + f(n/2+i)) \cdot (\omega^2)^{ij} \quad \text{and} \\ \mathcal{F}_f(2j+1) &= \sum_{i=0}^{n-1} f(i)\omega^{i(2j+1)} = \sum_{i=0}^{\frac{n}{2}-1} (f(i) - f(n/2+i))\omega^i \cdot (\omega^2)^{ij} \end{aligned}$$

Thus, the problem of computing DFT of f reduces to computing the DFT of two functions f_0 and f_1 (as defined in the algorithm) with $n/2$ as the domain size.

Time complexity - Computing f_0 and f_1 from f takes $O(n)$ operations over $\mathbb{F}$. Each of step 4 and 5 computes the DFT of a function with $n/2$ as the domain size. By solving the recurrence, $T(n) = 2T(n/2) + O(n)$, we get the time complexity of FFT to be $O(n \log n)$.

Let us use this tool to design a fast algorithm for polynomial multiplication.

Application: Polynomial multiplication

Suppose $f, g \in \mathbb{F}[x]$ be two polynomials of degree less than $n/2$. We will assume that $\mathbb{F}$ contains a primitive n^{th} root of unity ω and the element $n \cdot 1 = 1 + 1 + \ldots$ n times, is not zero. Since ω is a primitive root, it satisfies the property $\sum_{j=0}^{n-1} \omega^{ij} = 0$ for every $1 \leq i \leq n-1$.

Let $f = \sum_{i=0}^{n-1} c_i x^i$ where $c_{n/2}, \ldots, c_{n-1}$ are all zeroes, as $\deg(f) < n/2$. Associate a function $\hat{f} : [0, n-1] \rightarrow \mathbb{F}$ with f given by $\hat{f}(i) = c_i$. Define DFT of f to be the DFT of $\hat{f}$ i.e. $\mathcal{F}_f(j) \triangleq \mathcal{F}_{\hat{f}}(j) = \sum_{i=0}^{n-1} c_i \omega^{ij} = f(\omega^j)$, for all $0 \leq j \leq n-1$. Similarly define DFT of g as $\mathcal{F}_g(j) = g(\omega^j)$, for all j. The product polynomial $h = fg$ has degree less than n and hence we can also define DFT of h as $\mathcal{F}_h(j) = h(\omega^j)$ for all j. Let $h = \sum_{i=0}^{n-1} r_i x^i$ and $\mathsf{D}(\omega)$ be the following matrix.

$$\mathsf{D}(\omega) = \begin{pmatrix} 1 & 1 & 1 & ... & 1 \\ 1 & \omega & \omega^2 & ... & \omega^{n-1} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & ... & \omega^{(n-1)^2} \end{pmatrix}$$

Define two vectors $\mathbf{r} = (r_0, r_1, \ldots, r_{n-1})$ and $\mathbf{h} = (h(1), h(\omega), \ldots, h(\omega^{n-1}))$. Then, $\mathbf{r} \cdot \mathsf{D}(\omega) = \mathbf{h}$, implying that $n \cdot \mathbf{r} = \mathbf{h} \cdot \mathsf{D}(\omega^{-1})$. This is because $\mathsf{D}(\omega) \cdot \mathsf{D}(\omega^{-1}) = n \cdot I$, which follows from the property $\sum_{j=0}^{n-1} \omega^{ij} = 0$ for every $1 \leq i \leq n-1$. Here I is the $n \times n$ identity matrix. Now observe that, computing the expression $\mathbf{h} \cdot \mathsf{D}(\omega^{-1})$ is equivalent to computing the DFT of the polynomial $\tilde{h} = \sum_{i=0}^{n-1} h(\omega^i) x^i$ using ω^{-1} as the primitive n^{th} root of unity. We call this DFT of $\tilde{h}$ the *inverse*-DFT of h. This observation suggests the following polynomial multiplication algorithm.

16 Polynomial multiplication using FFT

1. Compute DFT of f to find the vector $(f(1), f(\omega), \ldots, f(\omega^{n-1}))$.
2. Compute DFT of g to find the vector $(g(1), g(\omega), \ldots, g(\omega^{n-1}))$.
3. Multiply the two vectors component-wise and obtain $(h(1), h(\omega), \ldots, h(\omega^{n-1}))$.
4. Compute inverse-DFT of h to get the vector $n \cdot \mathbf{r}$.
5. Divide $n \cdot \mathbf{r}$ by n to get $\mathbf{r} = (r_0, \ldots, r_{n-1})$.
6. Return $h = \sum_{i=0}^{n-1} r_i x^i$.

Time complexity - In steps 1, 2 and 4 the algorithm computes three DFTs, each with domain size n. The component-wise multiplication in step 3 and the division in step 5 take $O(n)$ time. Therefore, the overall time complexity of the algorithm is $O(n \log n)$ field operations.

The assumptions we made for the algorithm to work are - the field has a primitive n^{th} root of unity and $n \cdot 1$ is non-zero. The second assumption is not binding at all, as we can work with $n+1$ instead of n if the characteristic of the field divides n. However, we need a more careful argument to get rid of the first assumption. The idea is to work with an extension ring that contains a *principal* n^{th} root of unity. In this case, the complexity of the multiplication algorithm goes slightly up to $O(n \log n \log \log n)$ operations. The details of this algorithm, which is due to Schönhage and Strassen [SS71], can be found in chapter 8 of [GG03].

4.3 Automorphisms

An automorphism is a one to one correspondence from an algebraic structure, such as a ring or a field, to itself that preserves all the operations. In the language of abstract algebra, it is an isomorphism from an algebraic structure onto itself. We will focus on automorphisms over rings. A bijection ϕ from a ring $(\mathcal{R}, +, \cdot)$ to itself is called an automorphism if for all $a, b \in \mathcal{R}$, $\phi(a+b) = \phi(a) + \phi(b)$ and $\phi(a \cdot b) = \phi(a) \cdot \phi(b)$. The properties of automorphisms over rings play a crucial role in developing efficient deterministic algorithms for polynomial factoring and primality testing. It also provides an interesting insight into the integer factoring problem. These three applications and others (see [AS05]) demonstrate the versatility of this tool.

Application 1: Deterministic polynomial factoring

In section 4.1, we have shown that factoring a polynomial over $\mathbb{F}_q$ reduces to root finding of a polynomial that splits completely over $\mathbb{F}_q$. Suppose, we want to find a root of the polynomial $f = \prod_{i=1}^{n}(x - a_i)$, where $a_i \in \mathbb{F}_q$ for $1 \leq i \leq n$. By Chinese remaindering theorem,

$$\mathcal{R} = \frac{\mathbb{F}_q[x]}{(f)} \cong \oplus_{i=1}^{n} \frac{F_q[x]}{(x - a_i)} \cong \oplus_{i=1}^{n} \mathbb{F}_q$$

Let ϕ be an automorphism of $\mathcal{R}$ that fixes $\mathbb{F}_q$. Since any element in $\mathcal{R}$ can be expressed as a polynomial in x, the map ϕ is completely specified by its action on x. For any element $g \in \mathcal{R}$, we use the Chinese remaindering isomorphism and write $g = (c_1, \ldots, c_n)$ to mean $g = c_i \bmod (x - a_i)$ for every $1 \leq i \leq n$. For example, in this direct sum notation the element $x \in \mathcal{R}$ is $(a_1, \ldots, a_n)$. Let $\phi(x) = (b_1, \ldots, b_n)$ for some b_i's in $\mathbb{F}_q$.

Since $\phi(f(x)) = f(\phi(x)) = 0$ in $\mathcal{R}$, each $b_i = a_j$ for some j. Now suppose that there exist k and ℓ, $1 \leq k < \ell \leq n$, such that $b_k = b_\ell$. Define $g = \prod_{i=1, i\neq\ell}^{n}(x - b_i)$. Clearly, $g \neq 0$ in $\mathcal{R}$ as its degree is less than n. However, $\phi(g(x)) = g(\phi(x)) = 0$ in $\mathcal{R}$ which means kernel of ϕ is nontrivial and so ϕ can not be an isomorphism, which is a contradiction. Therefore, $b_k \neq b_\ell$ for every pair k and ℓ with $k \neq \ell$. In other words, ϕ induces a permutation σ among the n coordinates of the direct sum representation, i.e. $b_i = a_{\sigma(i)}$ for all $1 \leq i \leq n$, where $\sigma \in S_n$, the symmetric group of degree n. It is also easy to verify that any such permutation defines an automorphism, implying that there are a total of $n!$ automorphisms of $\mathcal{R}$ fixing $\mathbb{F}_q$.

Among these automorphisms suppose we could compute $n + 1$ distinct automorphisms $\phi_1, \ldots, \phi_{n+1}$. Here computing these automorphisms means finding the elements $\phi_s(x) \in \mathcal{R}$, for $1 \leq s \leq n + 1$. By Pigeonhole principle, there is a pair of automorphisms ϕ_s and ϕ_t with $s \neq t$, such that the first coordinates of the direct sum representations of $\phi_s(x)$ and $\phi_t(s)$ are the same. Since ϕ_s and ϕ_t are distinct, $\phi_s(x) - \phi_t(x) \neq 0$ in $\mathcal{R}$, and hence $\gcd(\phi_s(x) - \phi_t(x), f)$ gives a nontrivial factor of f.

The current best deterministic algorithm, due to Evdokimov [Evd94], uses this key idea of computing automorphisms of $\mathcal{R}$ to factor f. Evdokimov showed that, under the Extended Riemann Hypothesis, finding even one nontrivial automorphism of $\mathcal{R}$ (instead of n of them) is sufficient to factor f. Since any automorphism $\phi(x)$ is a root of the polynomial $f(y) \in \mathcal{R}[y]$ (as $f(\phi(x)) = 0$ in $\mathcal{R}$), the problem of factoring f reduces to the problem of finding a root of $h(y) = \frac{f(y)}{(y-x)} \in \mathcal{R}[y]$. This reduction can be argued as follows. First, note that the element $x \in \mathcal{R}$ is a root of $f(y) \in \mathcal{R}[y]$ and hence $(y - x)$ divides $f(y)$ over $\mathcal{R}$. It can be shown, using Chinese remaindering, that any root $\psi(x) \in \mathcal{R}$ of $h(y)$ defines a homomorphism of $\mathcal{R}$ that takes x to $\psi(x)$. It is also relatively easy to show that any homomorphism ψ that is not an automorphism, readily gives a proper factor of f. And as mentioned above, the case when ψ is an automorphism is handled in Evdokimov's work.

By the above argument, the problem of finding a root of $f(x)$ in $\mathbb{F}_q$ gets reduced to the problem of finding a root of $h(y)$ in $\mathcal{R}$. The gain in this reduction is that the degree of $h(y)$ is less than the degree of $f(x)$. Extending this argument it can be shown that the problem of finding a root of $h(y)$ in $\mathcal{R}$ reduces to the problem of finding a root of another polynomial of lower degree in the ring $T = \frac{\mathcal{R}[y]}{(h)}$ and so on. Eventually, when the degree reduces to one we have a root in a much larger ring, whose dimension over $\mathbb{F}_q$ can be bounded by $n^{O(\log n)}$. Using this root we descend down the hierarchy of rings and obtain a root of $h(y)$ in $\mathcal{R}$ which in turn factors $f(x)$.

Although, the above discussion is an overtly simplified exposition to Evdokimov's algorithm, it still carries the essence. The interested reader is encouraged to see the details of Evdokimov's work [Evd94].

Application 2: Deterministic primality testing

Until recently, no efficient deterministic primality testing algorithm was known. In 2002, a breakthrough was finally given by Agrawal, Kayal and Saxena [AKS04] in the form of the first deterministic polynomial time algorithm for checking primality. The following discussion is based upon their remarkable work.

Let n be the input integer. If n is a prime then, by Fermat's Little Theorem, $a^n = a \bmod n$ for every a. In other words, the map $\phi(x) = x^n$ is the trivial automorphism of the ring $\mathbb{Z}_n$. However, the converse of this statement is not true. There are infinitely many composite numbers m, known as Carmichael numbers [Car10, AGP94], for which $a^m = a \bmod m$ for all $0 \leq a < m$. This means Fermat's condition is a necessary but not a sufficient condition for primality. Even if it were sufficient it is not clear how to design an efficient deterministic test based on this condition. The AKS primality test overcomes both these shortcomings of the Fermat's test by considering automorphisms of an extension ring of $\mathbb{Z}_n$ namely, $\mathcal{R} = \frac{\mathbb{Z}_n[x]}{(x^r-1)}$ for a suitable choice of r.

Fix an $r \geq 0$ such that the order of n modulo r, denoted by $o_r(n)$, is greater than $\log^2 n$. Using the fact that the *lcm* of the first m numbers ($m \geq 7$) is at least 2^m, it is easy to show that such an $r \leq \lceil \log^5 n \rceil$ exists. Assume that n has no prime factor less than or equal to r. Otherwise, by trial division we can determine if n is prime or composite in polynomial time. Also, assume that n is not a perfect power or else we can fix each b in the range $[2, \lfloor \log n \rfloor]$ and use binary search to determine if there is an a such that $a^b = n$.

Consider the map $\phi : \mathcal{R} \to \mathcal{R}$ defined as $\phi(f(x)) = [f(x)]^n$ in $\mathcal{R}$. If n is a prime then it is easy to see that ϕ defines an automorphism of $\mathcal{R}$. Agrawal, Kayal and Saxena showed the converse of this statement. That is, if ϕ is an automorphism of $\mathcal{R}$ then n is a prime. Therefore, testing primality of n reduces to the task of checking if ϕ is an automorphism of $\mathcal{R}$. They further showed that, ϕ is an automorphism if and only if $\phi(x + a) = \phi(x) + a$ in $\mathcal{R}$ for every $1 \leq a \leq \sqrt{r} \log n$. Since $r \leq \lceil \log^5 n \rceil$, this check takes only polynomial time.

17 AKS primality test

1. `For` $a = 1$ `to` $\lfloor \sqrt{r} \log n \rfloor$ `do`
   `if` $(x + a)^n \neq x^n + a \mod (n, x^r - 1)$, `output COMPOSITE.`
2. `Else output PRIME.`

Correctness of the algorithm: Since we have assumed that n is not a perfect power and no prime factor of n is less than or equal to r, to show the correctness we only need to prove the following lemma for such an n.

Lemma 4.2 [AKS04] *If $\phi(x + a) = \phi(x) + a$ in $\mathcal{R}$ for all $1 \leq a \leq \sqrt{r} \log n$ then n is a prime.*

Proof: Let $\ell = \lfloor \sqrt{r} \log n \rfloor$ and $p > r$ be a prime factor of n. For every, $0 \leq a \leq \ell$,

$$\begin{aligned} (x + a)^n &= x^n + a \mod (n, x^r - 1) \\ \Rightarrow (x + a)^n &= x^n + a \mod (p, x^r - 1) \end{aligned}$$

Also, $(x + a)^{\frac{n}{p}} = x^{\frac{n}{p}} + a \mod (p, x^r - 1)$, as $x^r - 1$ is square-free over $\mathbb{Z}_p$. We say that m is *introspective* for a polynomial $f(x) \in \mathbb{Z}[x]$, if $[f(x)]^m = f(x^m) \mod (p, x^r - 1)$. It is easy to verify that, if m_1 and m_2 are introspective for $f(x)$ then so is $m_1 m_2$. Also, if m is introspective for $f(x)$ and $g(x)$ then it is introspective for $f(x)g(x)$. Surely, p is introspective for any polynomial. Therefore, every element of the set $I = \{(\frac{n}{p})^i \cdot p^j \mid i, j \geq 0\}$ is introspective for every polynomial of the set $P = \{\prod_{a=0}^{\ell} (x + a)^{e_a} \mid e_a \geq 0\}$.

Define G to be the set of all residues of numbers in I modulo r. Since $o_r(n) > \log^2 n$, the size of G say, $t > \log^2 n$. Now, let $h(x)$ be an irreducible factor of the r^{th} cyclotomic polynomial over $\mathbb{Z}_p$ and $\mathcal{G}$ be the set of all residues of elements in P modulo p and $h(x)$. Surely, $h(x)$ divides $x^r - 1$.

First, let us lower bound the size of $\mathcal{G}$. Denote the field $\frac{\mathbb{Z}_p[x]}{(h(x))}$ by $\mathbb{F}$. The claim is, if $f(x)$ and $g(x)$ are distinct polynomials of P with degree less than t then they map to distinct elements in $\mathbb{F}$. That is, $f(x)$ and $g(x)$ are distinct modulo p and $h(x)$. Suppose, on the contrary, $f(x) = g(x) \mod (p, h(x))$. Without any loss of clarity, we will say that $f(x)$ is an element of $\mathbb{F}$ to mean the element $f(x) \mod (p, h(x))$. For any $m \in I$, $[f(x)]^m = f(x^m)$ in $\mathbb{F}$ and so $f(x^m) = g(x^m)$ in $\mathbb{F}$, further implying that $f(x^{m'}) = g(x^{m'})$ in $\mathbb{F}$, where $m' = m \mod r$. Also note that the elements $\{x^{m'} | m' \in G\}$ are distinct elements of $\mathbb{F}$ as $h(x)$ can not divide any $x^{r'} - 1$ for $r' < r$. Therefore, the polynomial $Q(y) = f(y) - g(y)$ has at least t distinct roots namely, $\{x^{m'} | m' \in G\}$, in $\mathbb{F}$. Since $\deg_y Q < t$ this can only mean that Q is identically zero in $\mathbb{F}[y]$. However, since $f(x)$ and $g(x)$ are distinct elements of P and $p > \ell$ (as $r > \log^2 n$), $Q(y)$ can not be zero in $\mathbb{F}[y]$. Therefore, distinct elements of P with degree less than t must map to distinct elements of $\mathbb{F}$. The number of such elements in P is exactly $\binom{t+\ell}{t-1}$ and hence $|\mathcal{G}| \geq \binom{t+\ell}{t-1}$.

Next, we upper bound the size of $\mathcal{G}$. Consider the set $S = \{(\frac{n}{p})^i \cdot p^j \mid 0 \leq i, j \leq \lfloor\sqrt{t}\rfloor\}$. If n has a prime factor other that p then all the elements of S are distinct integers. Since $(\lfloor\sqrt{t}\rfloor + 1)^2 > t = |G|$, there are distinct $m_1, m_2 \in S$ such that $x^{m_1} = x^{m_2} \mod (x^r - 1)$. Therefore, for any $f(x) \in P$,

$$[f(x)]^{m_1} = f(x^{m_1}) = f(x^{m_2}) = [f(x)]^{m_2} \mod (p, h(x))$$

In other words, every element of $\mathcal{G}$ is a root of the nonzero polynomial $y^{m_1} - y^{m_2} \in \mathbb{F}[y]$. Hence $|\mathcal{G}| \leq \max\{m_1, m_2\} \leq n^{\sqrt{t}}$.

Using the relations $t > \log^2 n$ and $\ell = \lfloor\sqrt{r}\log n\rfloor$ it can be shown that $\binom{t+\ell}{t-1} > n^{\sqrt{t}}$, which leads us to a contradiction. Therefore, our assumption that n has a prime factor other than p is incorrect. But then, n has to be a prime since it is not a perfect power. This also implies that ϕ is an automorphism of $\mathcal{R}$. ∎

Time complexity: Time taken to perform a ring operation in $\mathcal{R}$ is $\tilde{O}(r \log n)$ bit operations. In step 1 we can check if $\phi(x + a) = \phi(x) + a$ using repeated squaring. Hence the total time complexity of the algorithm is $\tilde{O}(r \log^2 n \cdot \sqrt{r} \log n) = \tilde{O}(\log^{\frac{21}{2}} n)$ bit operations. Using an analytic number theory result by Fouvry [Fou85] it can be shown that $r = O(\log^3 n)$ and this brings down the complexity to $\tilde{O}(\log^{\frac{15}{2}} n)$.

Application 3: Integer factoring

Kayal and Saxena [KS05] showed some interesting connections between the integer factoring problem and some ring automorphism problems. The following lemma states their results.

Lemma 4.3 (Kayal-Saxena, 2005) *Given an integer n,*

1. *n can be factored in deterministic polynomial time if a nontrivial automorphism of the ring $\frac{\mathbb{Z}_n[x]}{(x^2-1)}$ can be computed in deterministic polynomial time.*

2. *n can be factored in expected polynomial time if the number of automorphisms of the ring $\frac{\mathbb{Z}_n[x]}{(x^2)}$ can be counted in polynomial time.*

Proof: Assume that n is odd. Let ϕ be an automorphism of $\mathcal{R} = \frac{\mathbb{Z}_n[x]}{(x^2-1)}$ that maps x to $ax+b$ for $a, b \in \mathbb{Z}_n$. If $d = \gcd(a, n)$ then $\phi(\frac{n}{d}x) = \frac{n}{d}ax + \frac{n}{d}b = \frac{n}{d}b$ in $\mathcal{R}$, which means d has to be 1. Also, $\phi(x^2) = (ax+b)^2 = a^2 + b^2 + 2abx = 1$, as $x^2 = 1$ in $\mathcal{R}$. Hence, $a^2 + b^2 = 1 \bmod n$ and $ab = 0 \bmod n$, which further implies that $b = 0 \bmod n$ (as $\gcd(a, n) = 1$) and $a^2 = 1 \bmod n$. Therefore, ϕ is an automorphism of $\mathcal{R}$ only if $\phi(x) = ax$ with $a^2 = 1 \bmod n$. The converse of this also holds and hence ϕ is a nontrivial automorphism of $\mathcal{R}$ if and only if $\phi(x) = ax$ where $a^2 = 1 \bmod n$ and $a \neq \pm 1 \bmod n$. If such an a exists then $\gcd(a+1, n)$ yields a nontrivial factor of n. It follows from Chinese remaindering that such an a exists when n is composite. If $n = n_1 n_2$ with $\gcd(n_1, n_2) = 1$ then an a satisfying $a = 1 \bmod n_1$ and $a = -1 \bmod n_2$ also satisfies the conditions $a^2 = 1 \bmod n$ and $a \neq \pm 1 \bmod n$.

We now prove the second statement. Let ψ be an automorphism of $\mathcal{S} = \frac{\mathbb{Z}_n[x]}{(x^2)}$ that maps x to $ax + b$. Once again, it follows by a similar argument that $\gcd(a, n) = 1$ and $b = 0 \bmod n$. Therefore, ψ is an automorphism of $\mathcal{S}$ only if $\psi(x) = ax$ where $\gcd(a, n) = 1$. The converse is also true i.e. any map ψ that sends x to ax for an a coprime to n, defines an automorphism of $\mathcal{S}$. Hence, the number of automorphisms of $\mathcal{S}$ is exactly $\varphi(n)$, where φ is the Euler's totient function. Using an argument very similar to the analysis of the Miller-Rabin test it can be shown that n can be factored in expected polynomial time if we know $\varphi(n)$. ∎

4.4 Hensel lifting

Given a ring $\mathcal{R}$ and an element $m \in \mathcal{R}$, Hensel [Hen18] designed a method to compute factorization of any element of $\mathcal{R}$ modulo m^ℓ (for an integer $\ell > 0$), given its factorization modulo m. This method, known as Hensel lifting, is used in many algorithms including multivariate polynomial factoring and polynomial division. We discuss these two applications in this section.

Lemma 4.4 (Hensel lifting) *Let $\mathcal{I}$ be an ideal of ring $\mathcal{R}$. Given elements $f, g, h, s, t \in \mathcal{R}$ with $f = gh \bmod \mathcal{I}$ and $sg + th = 1 \bmod \mathcal{I}$ there exist $g', h' \in \mathcal{R}$ such that,*

$$\begin{aligned} f &= g'h' \mod \mathcal{I}^2 \\ g' &= g \mod \mathcal{I} \\ h' &= h \mod \mathcal{I}. \end{aligned}$$

Further, any g' and h' satisfying the above conditions also satisfy:

1. *$s'g' + t'h' = 1 \bmod \mathcal{I}^2$ for some $s' = s \bmod \mathcal{I}$ and $t' = t \bmod \mathcal{I}$.*
2. *g' and h' are unique in the sense that any other solution g^* and h^* satisfying the above conditions also satisfy, $g^* = (1+u)g' \bmod \mathcal{I}^2$ and $h^* = (1-u)h' \bmod \mathcal{I}^2$, for some $u \in \mathcal{I}$.*

Proof: Let $f - gh = e \bmod \mathcal{I}^2$. Verify that $g' = g + te \bmod \mathcal{I}^2$ and $h' = h + se \bmod \mathcal{I}^2$ satisfy the conditions $f = g'h' \bmod \mathcal{I}^2, g' = g \bmod \mathcal{I}$ and $h' = h \bmod \mathcal{I}$. We refer to these three conditions together by C.

For any g', h' satisfying C, let $d = sg' + th' - 1 \bmod \mathcal{I}^2$. Verify that $s' = (1-d)s \bmod \mathcal{I}^2$ and $t' = (1-d)t \bmod \mathcal{I}^2$ satisfy the conditions $s'g' + t'h' = 1 \bmod \mathcal{I}^2$, $s' = s \bmod \mathcal{I}$ and $t' = t \bmod \mathcal{I}$.

Suppose g^*, h^* be another solution satisfying C. Let $v = g^* - g'$ and $w = h^* - h'$. The relation $g^*h^* = g'h' \bmod \mathcal{I}^2$ implies that $g'w + h'v = 0 \bmod \mathcal{I}^2$, as $v, w \in \mathcal{I}$. Since $s'g' + t'h' = 1 \bmod \mathcal{I}^2$, multiplying both sides by v we get $(s'v - t'w)g' = v \bmod \mathcal{I}^2$. By taking $u = s'v - t'w \in \mathcal{I}$, $g^* = (1+u)g' \bmod \mathcal{I}^2$. Similarly, $h^* = (1-u)h' \bmod \mathcal{I}^2$. ∎

When applied to the ring of polynomials, Hensel lifting always generates a 'unique factorization' modulo an ideal. The following lemma clarifies this point.

Lemma 4.5 *Let $f, g, h \in \mathcal{R}[x]$ be monic polynomials and $\mathcal{I}$ be an ideal of $\mathcal{R}[x]$ generated by some set $\mathcal{S} \subseteq \mathcal{R}$. If $f = gh \bmod \mathcal{I}$ and $sg + th = 1 \bmod \mathcal{I}$ for some $s, t \in \mathcal{R}[x]$ then,*

1. *there exist monic $g', h' \in \mathcal{R}[x]$ such that $f = g'h' \bmod \mathcal{I}^2$, $g' = g \bmod \mathcal{I}$ and $h' = h \bmod \mathcal{I}$.*
2. *if g^* is any other monic polynomial with $f = g^*h^* \bmod \mathcal{I}^2$, for some $h^* \in \mathcal{R}[x]$, and $g^* = g \bmod \mathcal{I}$ then $g^* = g' \bmod \mathcal{I}^2$ and $h^* = h' \bmod \mathcal{I}^2$.*

Proof: Applying Hensel's lemma, we can get $\tilde{g}, \tilde{h} \in \mathcal{R}[x]$ such that $f = \tilde{g}\tilde{h} \bmod \mathcal{I}^2$, $\tilde{g} = g \bmod \mathcal{I}$ and $\tilde{h} = h \bmod \mathcal{I}$. But $\tilde{g}$ and $\tilde{h}$ need not be monic. Let $v = \tilde{g} - g \in \mathcal{I}$. Since g is monic, we can divide v by g to obtain $q, r \in \mathcal{R}[x]$ such that $v = qg + r$ and $\deg_x(r) < \deg_x(g)$. Note that $q, r \in \mathcal{I}$. Define $g' = g + r$ and $h' = \tilde{h} + qh$ and verify that $f = g'h' \bmod \mathcal{I}^2$. Since $\deg_x(r) < \deg_x(g)$, g' is monic which also implies that h' is monic as f is monic.

Let g^* be any other monic polynomial with $f = g^*h^* \bmod \mathcal{I}^2$, for some h^* and $g^* = g \bmod \mathcal{I}$. This means, $gh = g^*h^* \bmod \mathcal{I}$ implying that $g(h - h^*) = 0 \bmod \mathcal{I}$. Since g is monic, $h^* = h \bmod \mathcal{I}$. Therefore, by Hensel's lemma, $g^* = (1 + u)g' \bmod \mathcal{I}^2$ for some $u \in \mathcal{I}$. Since g^* is monic this can only mean $g^* = g' \bmod \mathcal{I}^2$ and also $h^* = h' \bmod \mathcal{I}^2$. ∎

Application 1: Multivariate polynomial factoring

For simplicity, we present a factoring algorithm for the case of bivariate polynomials. This algorithm shows that bivariate factoring reduces to univariate factoring over any field. All the ideas required for a general reduction from multivariate factoring to univariate factoring are present in this special case.

Let $f \in \mathbb{F}[x, y]$ be a polynomial of total degree n which factors into irreducible polynomials $f_1, \ldots, f_k$. Assume that f is squarefree and monic in x. (Later we comment on how to drop these two assumptions.) Then $r(y) = \mathsf{Res}_x(f, \frac{\partial f}{\partial x}) \neq 0$ and $\deg_y(r) \leq 2n^2$. If $|\mathbb{F}| > 2n^2$ we can find an $a \in \mathbb{F}$ such that $r(a) \neq 0$, in which case f is squarefree modulo $(y - a)$. If the field size is small, we can work with a sufficiently large field extension of $\mathbb{F}$.

Polynomial $f(x, y) \bmod (y - a) = f(x, y + a) \bmod y$. Since the factors of $f(x, y)$ can be retrieved from the factors of $f(x, y + a)$, we can as well factor $f(x, y + a)$ instead of $f(x, y)$. In other words, we can assume (after a suitable transformation) that $f, f_1, \ldots, f_2$ are squarefree modulo y.

Suppose $f = gh \bmod y$ where $g, h \in \mathbb{F}[x]$ are monic, $\gcd(g, h) = 1$ and g is irreducible. Because of unique factorization in $\frac{\mathbb{F}[x,y]}{(y)} = \mathbb{F}[x]$, g must divide one f_i modulo y. Without loss of generality, let $f_1 = gh_1 \bmod y$ for some $h_1 \in \mathbb{F}[x]$ with $\gcd(g, h_1) = 1$. Using Lemma 4.5 we can lift the factorization of f to $f = g'h' \bmod y^{2^\ell}$, where g', h' are monic in x and $\ell > 0$. We claim that g' divides f_1 modulo y^{2^ℓ}. To argue this, suppose after applying Lemma 4.5 the factorization of $f_1 = gh_1 \bmod y$ gets lifted to $f_1 = \tilde{g}\tilde{h}_1 \bmod y^{2^\ell}$, where $\tilde{g}$ is monic in x. Since $f = f_1 \ldots f_k$, substituting the factorization of f_1 we get, $f = \tilde{g}\hat{h} \bmod y^{2^\ell}$ for some $\hat{h} \in \mathbb{F}[x, y]$. Inductively, assume that $\tilde{g} = g' \bmod y^{2^{\ell-1}}$. (The base case is $\tilde{g} = g' = g \bmod y$.) Then, by Lemma 4.5, since $f = g'h' = \tilde{g}\hat{h} \bmod y^{2^\ell}$ and $\tilde{g} = g' \bmod y^{2^{\ell-1}}$, it has to be that $\tilde{g} = g' \bmod y^{2^\ell}$. This proves our claim that g' divides f_1 modulo y^{2^ℓ}.

Consider the task of finding two polynomials $f_1', h_1'' \in \mathbb{F}[x, y]$ using the equation $f_1' = g'h_1'' \bmod y^{2^\ell}$ such that $\deg_x(f_1') < n$, $\deg_x(h_1'') < n - \deg_x(g')$ and total degree of f_1' is bounded by n. The coefficients of f_1' and h_1'' are treated as variables. Since $f_1 = g'\tilde{h}_1 \bmod y^{2^\ell}$, f_1 and $\tilde{h}_1$ are nonzero solutions for f_1' and h_1'', respectively. By equating the coefficients of the monomials from both sides of the equation $f_1' = g'h_1'' \bmod y^{2^\ell}$ we get a system of linear equations with the coefficients of f_1' and h_1'' as unknowns. Solving this system, we fix a nonzero solution for f_1' and h_1'' that minimizes $\deg_x(f_1')$. The claim is, for an appropriate choice of ℓ, $f_1' = q(y)f_1$ for some $q(y) \in \mathbb{F}[y]\backslash\{0\}$.

To prove this claim we use the fact that there are polynomials $s, t \in \mathbb{F}[x, y]$ such that $sf_1 + tf_1' = \mathsf{Res}_x(f_1, f_1') = r(y)$ (say). Since g' divides both f_1 and f_1' modulo y^{2^ℓ}, g' also divides $r(y)$ modulo y^{2^ℓ}. But g' is monic in x, which means $r(y) = 0 \bmod y^{2^\ell}$. As $\deg_y(r) < 2n^2$, if we choose $\ell = \lceil \log(2n^2 + 1) \rceil$ then $r(y) = 0$, implying that $\gcd(f_1, f_1')$ is nontrivial. Since f_1 is irreducible and $\deg_x(f_1')$ is minimum among all possible solutions for f_1', it must be that $f_1' = q(y)f_1$ for some $q(y) \in \mathbb{F}[y]$. In hindsight, we could have solved for f_1' and h_1'' that along with minimizing $\deg_x(f_1')$ also ensures that f_1' is monic in x. In that case, f_1' is simply f_1.

The following algorithm is based on the above discussion. It assumes that f is monic in x and $f \bmod y$ is squarefree.

18 Bivariate polynomial factoring

1. `Factor` $f = gh \bmod y$, `where` $g, h \in \mathbb{F}[x]$ `are monic and` g `is irreducible.`
2. `Let` $\ell = \lceil \log(2n^2 + 1) \rceil$.
3. `Use Hensel lifting to obtain` $f = g'h' \bmod y^{2^\ell}$, `where` g', h' `are monic in` x.
4. `Solve for` $f_1', h_1'' \in \mathbb{F}[x, y]$ `such that` $f_1' = g'h_1'' \bmod y^{2^\ell}$, $\deg_x(f_1') < n$, $\deg_x(h_1'') < n - \deg_x(g')$ `and total degree of` f_1' `bounded by` n.
5. `If no nonzero solution for` f_1' `and` h_1'' `exists, declare '`f `is irreducible'.`
6. `Find a nonzero solution that minimizes` $\deg_x(f_1')$ `and makes` f_1' `monic in` x.
7. `Return` f_1' `as an irreducible factor of` f.

Time complexity - Using Lemma 4.4 and 4.5 it is not hard to show that Hensel lifting in step 3 takes time polynomial in n field operations. Solving the system of linear equations in step 4 and 6 also takes polynomial time as the number of equations and variables are both bounded by some $\mathsf{poly}(n)$. Therefore, bivariate factoring reduces in polynomial time to univariate factoring in step 1.

If f is not squarefree to begin with, take $\gcd(f, \frac{\partial f}{\partial x})$ to filter out all multiple factors. Gcd of bivariate polynomials can be computed using Hensel lifting in a similar way. If f is not monic then there is a b among any $(n + 1)$ distinct elements of $\mathbb{F}$ such that $f(x, y + bx)$ is monic (up to scaling by a field element).

The argument presented for the bivariate case generalize to any multivariate polynomial. If $f \in \mathbb{F}[x, y_1, \ldots, y_m]$, we can use the ideal $\mathcal{I} = (y_1, \ldots, y_m)$ and lift the factorization of f modulo powers of $\mathcal{I}$. Some other minor changes are required but the factoring algorithm remains the same in essence. Therefore, multivariate factoring (up to constant number of variables) reduces to univariate factoring over a field.

Application 2: Polynomial division

As another application of Hensel lifting, we show that polynomial division has the same complexity as polynomial multiplication.

Let $f, g \in \mathcal{R}[x]$ be two polynomials with $\deg(f) = n$ and $\deg(g) = m \leq n$, and g be monic. We wish to compute $q, r \in \mathcal{R}[x]$ such that $f = qg + r$ and $\deg(r) < m$. For any polynomial p of degree d, define $\tilde{p} = x^d p(\frac{1}{x})$. In other words, the coefficients of p are reversed in $\tilde{p}$. If $f = qg + r$ then $\tilde{f} = \tilde{q}\tilde{g} + x^{n-m+1} \cdot \tilde{r}$. Technically, $\deg(r)$ can be less than $m-1$, in which case $\tilde{r}$ simply means $x^{m-1} r(\frac{1}{x})$. Therefore, $\tilde{f} = \tilde{q}\tilde{g} \bmod x^{n-m+1}$.

Since g is monic, the constant term of $\tilde{g}$ is 1 and hence $\tilde{g}$ is invertible modulo x^{n-m+1}. Let $\tilde{g}h = 1 \bmod x^{n-m+1}$. Then, $\tilde{q} = \tilde{f}h \bmod x^{n-m+1}$. So if we can compute $h \bmod x^{n-m+1}$ then one multiplication would give $\tilde{q}$ (since $\deg(\tilde{q}) = n - m$) from which $q = x^{n-m}\tilde{q}(\frac{1}{x})$ and then $r = f - qg$ can be easily retrieved. We use Hensel lifting to compute $h \bmod x^{n-m+1}$.

Notice that $\tilde{g} = 1 \bmod x$ and so also $h = 1 \bmod x$. Let $s = 1$ and $t = 0$, implying $s\tilde{g} + th = 1 \bmod x$. Use Hensel lifting iteratively for $\ell = \lceil \log(n - m + 1) \rceil$ times to compute $h \bmod x^{2^\ell}$ such that $\tilde{g}h = 1 \bmod x^{2^\ell}$. We need to clarify one point here. Recall, from Lemma 4.4, that the parameters in Hensel lifting evolve as : $g' = g + te$ and $t' = t(1-d)$. In our case $t = 0$ to begin with, hence $g' = g$ and $t' = 0$. This explains why the polynomial $\tilde{g}$ remains unchanged (as required) throughout the lifting process, till we reach the factorization $\tilde{g}h = 1 \bmod x^{n-m+1}$.

19 Polynomial division

1. Compute $\tilde{f} = x^n f(\frac{1}{x})$ and $\tilde{g} = x^m g(\frac{1}{x})$.
2. Let $\ell = \lceil \log(n - m + 1) \rceil$.
3. Use Hensel lifting to compute h, such that $\tilde{g}h = 1 \bmod x^{2^\ell}$.
4. Compute $\tilde{q} = \tilde{f}h \bmod x^{n-m+1}$ and $q = x^{n-m}\tilde{q}(\frac{1}{x})$.
5. Return q as the quotient and $r = f - qg$ as the remainder.

Time complexity - Computation of $\tilde{f}$ and $\tilde{g}$ takes $O(n)$ time. From Lemma 4.4, each Hensel lifting step involves only constantly many polynomial multiplications modulo x^{2^i}, for some $i \leq \ell$. So, total time taken in step 3 is $\sum_{i=1}^{\ell} O(\mathsf{M}(2^i)) = O(\mathsf{M}(n))$. Also step 4 takes $O(\mathsf{M}(n))$ time. Therefore, the division algorithm has $O(\mathsf{M}(n))$ time complexity.

4.5 Short vectors in lattices

Hermann Minkowski introduced the theory of *geometry of numbers* [Min10], which studies certain mathematical objects known as *lattices*. In this section, we show how these objects are used to factor polynomials over rationals. We also discuss an application of lattice in the *low exponent attack* on the RSA cryptosystems.

A lattice $\mathcal{L} \subseteq \mathbb{R}^m$ is generated by integer linear combinations of vectors in $\mathbb{R}^m$. Formally,

Definition 4.1 (Lattice) *Let $v_1, \ldots, v_n$ be linearly independent vectors of $\mathbb{R}^m$. A lattice $\mathcal{L}$, generated by $v_1, \ldots, v_n$, is defined as $\mathcal{L} = \{\sum_{i=1}^{n} a_i v_i \mid a_1, \ldots, a_n \in \mathbb{Z}\}$.*

The set $\{v_1, \ldots, v_n\}$ is called a basis of the lattice and n is the dimension of the lattice. Let V be the $n \times m$ matrix with v_i as the i^{th} row. When $m = n$, volume of $\mathcal{L}$ is given by $\mathsf{vol}(\mathcal{L}) = |\det(V)|$, which is independent of the choice of the basis vectors. This can be argued as follows. Let $\{u_1, \ldots, u_n\}$ be some other basis of $\mathcal{L}$. Denote by U the $n \times m$ matrix with u_i as the i^{th} row. Then there are two $n \times n$ integer matrices A and B such that $V = A \cdot U$ and $U = B \cdot V$, implying that A is an invertible integer matrix. Therefore, $\det(A) = \pm 1$ and if $m = n$ then $|\det(V)| = |\det(U)|$.

We denote the 2-norm of a vector v by $\|v\|$. The following theorem gives an upper bound on the length of a shortest vector of a lattice (when $m = n$) with respect to the 2-norm.

Theorem 4.2 (Minkowski, 1896) *The length of a shortest vector of a lattice $\mathcal{L} \subseteq \mathbb{R}^n$ of dimension n, is at most $\sqrt{n} \cdot \mathsf{vol}(\mathcal{L})^{\frac{1}{n}}$.*

Proof: Let λ be the length of any shortest vector of $\mathcal{L}$. Consider the parallelepiped defined by the vectors $v_1, \ldots, v_n$. The volume of this parallelepiped is $\det(V) = \mathsf{vol}(\mathcal{L})$. Place spheres of radii $\frac{\lambda}{2}$ centered at the corners of this parallelepiped. Each sphere intersects with a part of this parallelepiped. The sum of the volumes of these intersecting spaces for all the spheres must equal the volume of a single sphere in $\mathbb{R}^n$ of radius $\frac{\lambda}{2}$. Since the spheres do not intersect among themselves, this volume must be less than the volume of the parallelepiped. Volume of a sphere in $\mathbb{R}^n$ of radius r is $\frac{\pi^{n/2} r^n}{\Gamma(n/2+1)}$, where Γ is the Gamma function. Therefore, $\frac{\pi^{n/2}(\lambda/2)^n}{\Gamma(n/2+1)} \leq \mathsf{vol}(\mathcal{L})$. Simplifying using Stirling's theorem, $\lambda \leq \sqrt{n} \cdot \mathsf{vol}(\mathcal{L})^{\frac{1}{n}}$. ∎

The task of computing a short vector in a given lattice is a key step in many algorithms. However, it is known that finding a vector of length very close to the shortest vector length is computationally hard. In a series of important developments, Ajtai [Ajt98] showed that it is NP-hard to compute a shortest vector in a lattice. Following this, Micciancio [Mic00] showed that finding a vector within $\sqrt{2}$ factor of the shortest vector length is also NP-hard. Recently, Haviv and Regev [HR07], building on an earlier work by Khot [Kho05], showed that under a reasonable complexity theory assumption ($\mathsf{NP} \not\subseteq \mathsf{RTIME}(2^{\mathsf{poly}(\log n)})$) there is no polynomial time algorithm that can find a vector to within $2^{(\log n)^{1-\epsilon}}$ factor of the shortest vector length, for any arbitrarily small $\epsilon > 0$.

Nevertheless, it is also known that approximating the shortest vector to within a polynomial factor is probably not a hard problem. Goldreich and Goldwasser [GG98] showed that obtaining a $\sqrt{\frac{n}{\log n}}$ factor approximation is in the complexity class AM[2] and hence unlikely to be NP-hard. Furthermore, it was shown by Aharonov and Regev [AR04] that finding a $\sqrt{n}$ factor approximate vector is in the class $\mathsf{NP} \cap \mathsf{coNP}$.

Fortunately, for many algorithms finding a short vector even within an exponential factor of the shortest vector length is useful enough. In a major breakthrough, Lenstra, Lenstra and Lovász [LJL82] gave a polynomial time algorithm (LLL algorithm, for short) to find a vector of length no more than $2^{\frac{n-1}{2}}$ times the length of a shortest vector. All the hardness results, mentioned above, followed after this work in an attempt to bridge the gap between the approximation factors for which either an efficient algorithm or a hardness result is known.

The main idea behind the LLL algorithm is to compute a *reduced* basis, from a given basis of the lattice, by closely following the Gram-Schmidt orthogonal basis computation. This reduced basis is guaranteed to contain a short vector. The following lemma explains why Gram-Schmidt orthogonalization (GSO) plays a role in short vector computation. Given a basis $\{v_1, \ldots, v_n\}$ of $\mathcal{L}$, let $\{v_1^*, \ldots, v_n^*\}$ be the orthogonal basis computed by the GSO. Then,

Lemma 4.6 *For any nonzero $v \in \mathcal{L}$, $\|v\| \geq \min\{\|v_1^*\|, \ldots, \|v_n^*\|\}$.*

Proof: Let $v = \sum_{i=1}^n a_i v_i$, where each $a_i \in \mathbb{Z}$, and k be the largest index for which $a_k \neq 0$. The GSO computes each v_i^* as $v_i^* = v_i - \sum_{j<i} \mu_{ij} v_j^*$, where $\mu_{ij} \in \mathbb{R}$. Therefore, $v = a_k v_k^* + \sum_{j<k} \mu_j' v_j^*$, for some $\mu_j' \in \mathbb{R}$. Now, $\|v\|^2 = a_k^2 \|v_k^*\|^2 + \sum_{j<k} {\mu_j'}^2 \|v_j^*\|^2 \geq \|v_k^*\|^2$, since a_k is an integer. Hence, $\|v\| \geq \|v_k^*\| \geq \min\{\|v_1^*\|, \ldots, \|v_n^*\|\}$. ∎

Thus, if the vectors $v_1^*, \ldots, v_n^*$ belong to the lattice $\mathcal{L}$ then a v_i^* with the minimum norm is a shortest vector of $\mathcal{L}$. But, the basis vectors computed by the GSO need not always belong to the lattice. This problem gives rise to the notion of a *reduced basis*.

Definition 4.2 (Reduced basis) *A basis $\{u_1, \ldots, u_n\}$ of $\mathcal{L}$ is called a reduced basis, if the orthogonal basis vectors $\{u_1^*, \ldots, u_n^*\}$, computed by the* GSO*, satisfy the property $\|u_i^*\|^2 \leq 2\|u_{i+1}^*\|^2$ for all $1 \leq i < n$.*

Suppose we succeed in efficiently computing a reduced basis $\{u_1, \ldots, u_n\}$ from a given basis of the lattice $\mathcal{L}$. From Lemma 4.6, any vector $u \in \mathcal{L}$ satisfies $\|u\| \geq \min\{\|u_1^*\|, \ldots, \|u_n^*\|\}$, and by the above definition any $\|u_i^*\| \geq 2^{-\frac{n-1}{2}}\|u_1^*\|$. Therefore, $\|u\| \geq 2^{-\frac{n-1}{2}}\|u_1\|$, as $\|u_1^*\| = \|u_1\|$. This means, the length of the vector u_1 is at most $2^{\frac{n-1}{2}}$ times the length of a shortest vector in $\mathcal{L}$.

The LLL algorithm computes a reduced basis $\{u_1, \ldots, u_n\}$ from a given basis $\{v_1, \ldots, v_n\}$, where $v_i \in \mathbb{Z}^m$ and $\|v_i\| \leq A \in \mathbb{Z}^+$ for every i, in time polynomial in m and $\log A$. Before we describe this algorithm, let us fix certain notations and conventions.

Matrices V and U are as defined before (see the paragraph following Definition 4.1). V^* and U^* are $n \times m$ matrices with v_i^* and u_i^* as the i^{th} rows, respectively. As defined in section A.2, the projection matrix M, in a GSO computation from U to U^*, is given by $M = (\mu_{ij})_{1 \leq i,j \leq n}$ where $\mu_{ii} = 1$ for all i, $\mu_{ij} = 0$ if $j > i$, and $u_i = u_i^* + \sum_{j<i} \mu_{ij} u_j^*$ with $\mu_{ij} = \frac{u_i \cdot u_j^*}{\|u_j^*\|^2}$ for $j < i$. Surely, $U = M \cdot U^*$. For brevity, we say 'GSO of U' to refer to the matrices M and U^*. The notation $\lceil \mu_{ij} \rfloor$ is used to mean the integer closest to μ_{ij}. The space generated by the vectors $u_1, \ldots, u_j$ over rationals is denoted by $\mathcal{U}_j$.

More details on the following exposition of the LLL algorithm and its analysis can be found in chapter 16 of [GG03].

20 LLL basis reduction

```
1.  Initialize U = V and compute GSO of U.  Set i = 2.
2.  while i ≤ n do
3.      for j = i − 1 to 1 do
4.          Set u_i = u_i − ⌈μ_ij⌋ u_j and update GSO of U.
5.      if i > 1 and ‖u*_{i−1}‖² > 2‖u*_i‖² then
6.          Swap u_i and u_{i−1} and update GSO of U.  Set i = i − 1.
7.      else, set i = i + 1.
8.  Return U.
```

Correctness and time complexity - Because of the check, $\|u_{i-1}^*\|^2 > 2\|u_i^*\|^2$, in step 5 of the algorithm, it is not hard to see that the algorithm outputs a reduced basis whenever it halts. One only has to use induction on the index i to show this. However, it is not clear as such, whether the algorithm halts in polynomial time. The following lemma proves this.

Lemma 4.7 *The* LLL *algorithm halts and finds a reduced basis in polynomial time.*

Proof: We need to show that the outer `while`-loop of the algorithm executes only polynomially many times. Moreover, we also need to show that the size of the numerator and denominator of any rational number involved in the computation is polynomially bounded.

The matrix U and its GSO are updated in step 4 and step 6. Right before an update, let U be the matrix with GSO data M and U^*. After an update, suppose U, M and U^* get altered to $\tilde{U}$, $\tilde{M}$ and $\tilde{U}^*$ respectively, with the corresponding entries as $\tilde{u}_k$, $\tilde{\mu}_{k\ell}$ and $\tilde{u}_k^*$ for $1 \leq k, \ell \leq n$. $\tilde{\mathcal{U}}_k$ be the space generated by $\tilde{u}_1, \ldots, \tilde{u}_k$.

First, let us focus on step 4. Indices i and j are as in step 4. Let N be the $n \times n$ matrix with ones on the diagonal and $-\lceil \mu_{ij} \rfloor$ as the $(i,j)^{th}$ entry. The remaining entries of N are

zeroes. Then, $\tilde{U} = N \cdot U$. Notice that, $\tilde{\mathcal{U}}_k = \mathcal{U}_k$ for all k. Since $\tilde{u}^*_{k+1}$ is the projection of $\tilde{u}_{k+1}$ on the orthogonal complement of $\tilde{\mathcal{U}}_k$, we can infer that $\tilde{U}^* = U^*$. It is also easy to verify that $\tilde{M} = N \cdot M$. Since $\tilde{\mu}_{ij} = \mu_{ij} - \lceil \mu_{ij} \rfloor$, $|\tilde{\mu}_{ij}| \leq \frac{1}{2}$. Also, $\tilde{\mu}_{i\ell} = \mu_{i\ell}$ for any $\ell > j$. Therefore, by induction on j, $|\tilde{\mu}_{i\ell}| \leq \frac{1}{2}$ for all $j \leq \ell < i$ and $|\tilde{\mu}_{k\ell}| \leq \frac{1}{2}$ for all $1 \leq \ell < k < i$. To summarize, after an update in step 4, the $(k.\ell)^{th}$ entry of the projection matrix has absolute value at most $\frac{1}{2}$ for all $1 \leq \ell < k \leq i$, and the orthogonal basis remains unaltered.

Let us see what happens after an update in step 6. Index i is as in step 6. This time $\tilde{U}$ is simply a permutation matrix times U. The permutation matrix has the effect of swapping the $(i-1)^{th}$ and the i^{th} rows of U. Since $\tilde{\mathcal{U}}_k = \mathcal{U}_k$ for all $k \neq i-1$, hence $\tilde{u}^*_k = u^*_k$ for all $k \notin \{i-1, i\}$. Now notice that $\tilde{u}^*_{i-1} = u^*_i + \mu_{ii-1} u^*_{i-1}$, implying $\|\tilde{u}^*_{i-1}\|^2 \leq \|u^*_i\|^2 + |\mu_{ii-1}|^2 \cdot \|u^*_{i-1}\|^2$. In step 6, $\|u^*_i\|^2 < \frac{1}{2}\|u^*_{i-1}\|^2$ and $\mu_{ii-1} \leq \frac{1}{2}$, as argued in the previous paragraph. Therefore, $\|\tilde{u}^*_{i-1}\|^2 < \frac{3}{4}\|u^*_{i-1}\|^2$. Also, since $\tilde{u}^*_i$ is the projection of u_{i-1} on the orthogonal complement of $\tilde{\mathcal{U}}_{i-1} \supseteq \mathcal{U}_{i-2}$, it is also the projection of u^*_{i-1} on the orthogonal complement of $\tilde{\mathcal{U}}_{i-1}$, implying that $\|\tilde{u}^*_i\| \leq \|u^*_{i-1}\|$. To summarize, after an update in step 6, $\tilde{u}^*_k = u^*_k$ for all $k \notin \{i-1, i\}$, $\|\tilde{u}^*_{i-1}\|^2 < \frac{3}{4}\|u^*_{i-1}\|^2$ and $\|\tilde{u}^*_i\| \leq \|u^*_{i-1}\|$, i.e. $\max_k\{\|\tilde{u}^*_k\|\} \leq \max_k\{\|u^*_k\|\}$.

Let U_k be a $k \times m$ matrix with rows $u_1, \ldots, u_k$. Define $d_k = \det(U_k \cdot U_k^T) \in \mathbb{Z}^+$. Surely, $U_k = M_k \cdot U^*_k$, where M_k is the Gram-Schmidt projection matrix and U^*_k is the orthogonal basis matrix with $u^*_1, \ldots, u^*_k$ as the rows. Since $\det(M_k) = 1$ and $u^*_1, \ldots, u^*_k$ are mutually orthogonal, $d_k = \prod_{\ell=1}^{k} \|u^*_\ell\|^2$. Thus, in step 4, d_k remains unchanged for every k. In step 6, d_k remains unchanged for every $k \neq i-1$ and d_{i-1} reduces by a factor of at least $\frac{3}{4}$. The reason d_k remains the same for $k \neq i-1$ is because in step 6, U_i only changes to $P \cdot U_i$ for some permutation matrix P. Define $D = \prod_{k=1}^{n} d_k$. Therefore, D remains unchanged in step 4 but decreases by a factor of at least $\frac{3}{4}$ each time step 6 is executed. At the start of the algorithm the value of D is at most $\prod_{k=1}^{n} A^{2k} \leq A^{n^2}$ (as $\|v^*_i\| \leq \|v_i\| \leq A$, for all i). Hence, step 6 can be executed at most $O(n^2 \log A)$ times. This proves that the algorithm halts after polynomially many executions of the `while`-loop as the index i can decrease for at most $O(n^2 \log A)$ times.

We are now left with the task of showing that all the rational numbers in U, M and U^* have small numerators and denominators at any stage of the algorithm. We begin with the row vectors in U. Since, every $u_k \in \mathbb{Z}^m$, it is sufficient to bound the value of $\|u_k\|$. To start with, $\|u_k\| \leq A$ for every k. Since $\|u^*_k\| \leq \|u_k\|$ and $\max_k\{\|u^*_k\|\}$ never increases after an update in step 4 or step 6, hence $\|u^*_k\| \leq A$ for every k at all times of the algorithm.

First, we claim that $\|u_k\| \leq \sqrt{n} \cdot A$ for every k at all times of the algorithm, except in step 4 when $k = i$. Since, step 6 does only a swap between u_{i-1} and u_i, we only need to show that the claim holds for $\|u_i\|$ at the end of the `for`-loop, just before step 5. As $u_i = u^*_i + \sum_{j<i} \mu_{ij} u^*_j$, $\|u_i\|^2 \leq n m_i^2 A^2$ where $m_i = \max_{1 \leq j \leq i}\{|\mu_{ij}|\}$. Notice that, at the end of the `for`-loop $|\mu_{ij}| \leq \frac{1}{2}$ for every $j < i$. Hence, by taking into account that $\mu_{ii} = 1$, we have $\|u_i\| \leq \sqrt{n} A$.

Let us see how $\|u_i\|$ changes within the `for`-loop. For this, we first need the following bound on the value of $\mu_{k\ell}$ for any $\ell < k$.

$$|\mu_{k\ell}| = \frac{|u_k \cdot u^*_\ell|}{\|u^*_\ell\|^2} \leq \frac{\|u_k\| \cdot \|u^*_\ell\|}{\|u^*_\ell\|^2} = \frac{\|u_k\|}{\|u^*_\ell\|} \leq \sqrt{d_{\ell-1}} \cdot \|u_k\|. \tag{1}$$

The last inequality holds because $\|u^*_\ell\|^2 = \frac{d_\ell}{d_{\ell-1}} \geq \frac{1}{d_{\ell-1}}$ as $d_\ell \in \mathbb{Z}^+$. Now notice that, after an update in step 4, $\mu_{i\ell}$ changes to $\mu_{i\ell} - \lceil \mu_{ij} \rfloor \mu_{j\ell}$ for every $\ell \leq j$. Since $|\mu_{j\ell}| \leq \frac{1}{2}$ for $\ell < j$,

$$|\mu_{i\ell} - \lceil \mu_{ij} \rfloor \mu_{j\ell}| \leq |\mu_{i\ell}| + |\lceil \mu_{ij} \rfloor| \cdot |\mu_{j\ell}| \leq m_i + (m_i + \frac{1}{2}) \cdot \frac{1}{2} \leq 2m_i.$$

The last inequality holds because $m_i \geq 1$. Also, for $\ell = j$, $\mu_{i\ell} - \lceil \mu_{ij} \rfloor \mu_{j\ell} \leq \frac{1}{2}$. Therefore, the value of m_i is at most doubled after an update in step 4. We have already shown that, at the start of the `for`-loop, $\|u_k\| \leq \sqrt{n}A$ for every k. Hence, from equation (1), $m_i \leq \sqrt{n}A^{n-1}$ at the start of the `for`-loop. Together with the facts that m_i can doubled for n times and $\|u_i\| \leq \sqrt{n}m_iA$, we get $\|u_i\| \leq n \cdot (2A)^n$. This shows that the size of any entry in u_k is at most $O(n \log A)$ bits at any time of the algorithm.

We now need to bound the numerator and denominator of $\mu_{k\ell}$ and the rational entries in u_k^*. We claim that, $d_{k-1}u_k^* \in \mathbb{Z}^m$ for every k. This can be argued as follows. It is easy to see that, every u_k^* can be expressed as $u_k^* = u_k - \sum_{\ell<k} \lambda_{k\ell} u_\ell$ for some $\lambda_{k\ell} \in \mathbb{Q}$. Since $u_j \,.\, u_k^* = 0$ for any $j < k$, we get $\sum_{\ell<k} \lambda_{k\ell}(u_j \,.\, u_\ell) = (u_j \,.\, u_k)$ for every $1 \leq j < k$. This gives a system of $k-1$ linear equations in the $\lambda_{k\ell}, 1 \leq \ell < k$. The determinant of the coefficient matrix $(u_j \,.\, u_\ell)_{1\leq j,\ell\leq k}$ is exactly d_{k-1}. Hence, by Cramer's rule, $d_{k-1}\lambda_{k\ell} \in \mathbb{Z}$. Therefore, $d_{k-1}u_k^* = d_{k-1}u_k - \sum_{\ell<k} d_{k-1}\lambda_{k\ell}u_\ell \in \mathbb{Z}^m$. This means, the denominator of every entry in u_k^* is at most $d_{k-1} \leq A^{2n}$ and the numerator is surely bounded by $d_{k-1}\|u_k^*\| \leq A^{2(n-1)} \cdot A \leq A^{2n}$.

Our last claim is that, $d_\ell \cdot \mu_{k\ell} \in \mathbb{Z}$. This is because, $d_\ell\mu_{k\ell} = d_\ell \cdot \frac{u_k \,.\, u_\ell^*}{\|u_\ell^*\|^2} = u_k \,.\, (d_{\ell-1}u_\ell^*)$. We have already shown that $d_{\ell-1}u_\ell^* \in \mathbb{Z}^m$ and so $d_\ell\mu_{k\ell} \in \mathbb{Z}$. Therefore, the denominator of $\mu_{k\ell}$ can be at most $d_\ell \leq A^{2n}$ and the numerator is bounded by $d_\ell|\mu_{k\ell}|$. From equation (1),

$$d_\ell \cdot |\mu_{k\ell}| \leq d_\ell \cdot \sqrt{d_{\ell-1}} \cdot \|u_k\| \leq A^{2(n-1)} \cdot A^{n-2} \cdot n(2A)^n \leq n \cdot (2A^4)^n.$$

Thus, the size of the numerator and denominator of any rational number involved in the computation is at most $O(n \log A)$ bits. This completes our proof. ∎

The main result of this section is summarized in the following theorem.

Theorem 4.3 (Lenstra-Lenstra-Lovász, 1982) *Given n vectors $v_1, \ldots, v_n \in \mathbb{Z}^m$ that are linearly independent over $\mathbb{Q}$ and $\|v_i\| \leq A \in \mathbb{Z}^+$ for every i, a vector $v \in \mathbb{Z}^m$ can be computed in time $\mathsf{poly}(m, \log A)$ such that $\|v\|$ is at most $2^{\frac{n-1}{2}}$ times the length of a shortest vector in the lattice $\mathcal{L} = \sum_{i=1}^n \mathbb{Z}v_i$.*

The condition that $v_1, \ldots, v_n$ are linearly independent is a bit superfluous because one can easily compute a set of linearly independent vectors $v_1', \ldots, v_{n'}' \in \mathbb{Z}^m$ from $v_1, \ldots, v_n$ such that $\mathcal{L} = \sum_{i\leq n} \mathbb{Z}v_i = \sum_{j\leq n'} \mathbb{Z}v_j'$. For a survey on the theory of lattices, refer to the article by Hendrik W. Lenstra Jr. [Jr.08] in [BS08].

Application 1: Factoring polynomials over rationals

Let us now see how to use short vectors in a lattice to factor univariate polynomials over rationals. Much of this algorithm, due to Lenstra, Lenstra and Lovász [LJL82], resembles the bivariate factoring algorithm discussed in section 4.4, but they differ at one crucial step. This will be clear from the following discussion.

Given a polynomial $f \in \mathbb{Q}[x]$, we can multiply f with the *lcm* of the denominators of its coefficients to get a polynomial in $\mathbb{Z}[x]$. So, without loss of generality assume that $f \in \mathbb{Z}[x]$. Before we get down to the details of factoring f, we need to address one basic question. How large can a coefficient of a factor of f be? If a factor of f has very large coefficients, it might not be possible to even output the factor in polynomial time. Fortunately, this is not the case.

With every polynomial $f = \sum_{i=1}^n c_n x^i \in \mathbb{Z}[x]$ associate a coefficient vector $v_f = (c_n, \ldots, c_0) \in \mathbb{Z}^{n+1}$. Norm of f, denoted by $\|f\|$, is defined as $\|f\| = \|v_f\|$. Let $\|f\| \leq A \in \mathbb{Z}^+$ and α be

a root of f in $\mathbb{C}$. Since, $f(\alpha) = 0$, $|c_n\alpha^n| = |\sum_{i=0}^{n-1} c_i\alpha^i| \leq \sum_{i=0}^{n-1} |c_i||\alpha|^i$. If $|\alpha| > 1$ then $|c_n\alpha^n| \leq nA|\alpha|^{n-1}$, implying that $|\alpha| \leq nA$. Any factor $f_1 \in \mathbb{Z}[x]$ of f is a product of at most $n-1$ linear factors over $\mathbb{C}$ and an integer with absolute value at most $|c_n| \leq A$. Therefore, absolute value of any coefficient of f_1 is bounded by $A \cdot \binom{n}{n/2} \cdot (nA)^{n-1} \leq (2nA)^n$ and hence $\|f_1\| \leq \sqrt{n}(2nA)^n$. Although, this bound is sufficient for our purpose, a sharper bound on $\|f_1\|$ is provided by Mignotte [Mig74] (also known as the Landau-Mignotte bound).

Lemma 4.8 (Landau-Mignotte bound) *If $f_1 \in \mathbb{Z}[x]$ is a proper factor of a polynomial $f \in \mathbb{Z}[x]$ of degree n, then $\|f_1\| \leq 2^{n-1}\|f\|$.*

Therefore, $\|f_1\| \leq 2^{n-1}A$ and size of any coefficient of f_1 is at most $O(n + \log A)$ bits. We refer to this bound on $\|f_1\|$ as B.

For simplicity, assume that f is monic and squarefree. Since the absolute value of any entry in the Sylvester matrix $S(f, \frac{df}{dx})$ is at most nA, by Hadamard's inequality $|\mathsf{Res}_x(f, \frac{df}{dx})| \leq (2n)^n(nA)^{2n} = (2n^3A^2)^n$. Searching the first $O(n\log(nA))$ primes we can find a prime p such that $f \bmod p$ is monic and squarefree. Suppose $f = gh \bmod p$, where g is monic and irreducible in $\mathbb{F}_p[x]$. We can lift this factorization using Hensel lifting for ℓ steps to obtain $f = g'h' \bmod p^{2^\ell}$, where $g' = g \bmod p$ and $h' = h \bmod p$. Arguing along the same line as in the bivariate factoring case, we can show that there is an irreducible factor $f_1 \in \mathbb{Z}[x]$ of f such that g' divides f_1 modulo p^{2^ℓ}. In bivariate factoring we could find f_1 by solving a system of linear equations over the underlying field. But this approach does not apply here straightaway.

Suppose $f_1' \in \mathbb{Z}[x]\backslash\{0\}$ with $\deg(f_1') < n$ and $\|f_1'\| = C \in \mathbb{Z}$, such that $f_1' = g'h_1'' \bmod p^{2^\ell}$ for some $h_1'' \in \mathbb{Z}[x]$. As argued in the proof of the Hadamard inequality (Lemma 4.1), the absolute value of the determinant of an integer matrix with row vectors $r_1, \ldots, r_k$ is bounded by $\prod_{i=1}^k \|r_i\|$. Therefore, $|\mathsf{Res}_x(f, f_1')| \leq A^{n-1}C^n$. Surely, there exists s and t in $\mathbb{Z}[x]$ such that $sf + tf_1' = \mathsf{Res}_x(f, f_1')$. Since f and f_1' share a common factor g' modulo p^{2^ℓ}, if $|\mathsf{Res}_x(f, f_1')| < p^{2^\ell}$ then $\gcd(f, f_1')$ is nontrivial. Therefore, all we want is $p^{2^\ell} > A^{n-1}C^n$. This means, $\|f_1'\| = C$ has to be small in order to ensure that ℓ is small. This is the reason why just solving a f_1' with $f_1' = g'h_1'' \bmod p^{2^\ell}$ does not help. We also need to ensure that $\|f_1'\|$ is small. Putting the conditions together, we want a nonzero $f_1' \in \mathbb{Z}[x]$ of degree less than n, such that $\|f_1'\|$ is 'small' and $f_1' = g'h_1'' \bmod p^{2^\ell}$ for some $h_1'' \in \mathbb{Z}[x]$. It is this step which is solved by finding a short vector in a lattice.

Let $\deg(g') = k < n$. Consider the polynomials $x^{n-k-1}g', x^{n-k-2}g', \ldots, xg', g'$ and the polynomials $p^{2^\ell}x^{k-1}, p^{2^\ell}x^{k-2}, \ldots, p^{2^\ell}$. Let $v_1, \ldots, v_n \in \mathbb{Z}^n$ be the n coefficient vectors corresponding to these n polynomials. It is not hard to see that the coefficient vector $v_{f_1'}$ of any solution f_1', satisfying $f_1' = g'h_1'' \bmod p^{2^\ell}$ for some $h_1'' \in \mathbb{Z}[x]$, belongs to the lattice $\mathcal{L} = \sum_{i=1}^n \mathbb{Z}v_i$. Also, the vectors $v_1, \ldots, v_n$ are linearly independent. Since f_1 is a solution for f_1', $v_{f_1} \in \mathcal{L}$ and hence the length of the shortest vector in $\mathcal{L}$ is at most $\|v_{f_1}\| = \|f_1\| \leq B = 2^{n-1}A$. Applying Theorem 4.3, we can find a short vector v such that $\|v\| \leq 2^{\frac{3(n-1)}{2}} \cdot A$. Let f_1' be the polynomial corresponding to the coefficient vector v. Surely, $f_1' = g'h_1'' \bmod p^{2^\ell}$ for some $h_1'' \in \mathbb{Z}[x]$, $\deg(f_1') < n$ and $\|f_1'\| = \|v\| = C \leq 2^{\frac{3(n-1)}{2}} \cdot A$. Since, $|\mathsf{Res}_x(f, f_1')| \leq A^{n-1}C^n \leq 2^{\frac{3n(n-1)}{2}} \cdot A^{2n-1}$, we only need to choose an ℓ such that $p^{2^\ell} > 2^{\frac{3n(n-1)}{2}} \cdot A^{2n-1}$. Therefore, all the integers involved in Hensel lifting have size at most $O(n^2 + n\log A)$ bits, implying that the lifting takes only polynomial time.

The following algorithm summarizes the above discussion. It assumes that $f \in \mathbb{Z}[x]$ is a monic, squarefree polynomial of degree n, and $\|f\| \leq A$.

21 Polynomial factoring over rationals

1. Find a prime p such that $f \mod p$ is squarefree.
2. Let $D = 2^{\frac{3(n-1)}{2}} \cdot A$ and $E = 2^{\frac{3n(n-1)}{2}} \cdot A^{2n-1}$.
3. Factor $f = gh \mod p$ where g is monic and irreducible in $\mathbb{F}_p[x]$.
4. Let $\deg(g) = k$. If $k = n$, declare 'f is irreducible'.
5. Use Hensel lifting to compute $f = g'h' \mod p^{2^\ell}$ such that $E^2 \geq p^{2^\ell} > E$.
6. Let $v_1, \ldots, v_n$ be the coefficient vectors in $\mathbb{Z}^n$ corresponding to the polynomials $x^{n-k-1}g', x^{n-k-2}g', \ldots, xg', g'$ and $p^{2^\ell}x^{k-1}, p^{2^\ell}x^{k-2}, \ldots, p^{2^\ell}$.
7. Use LLL algorithm to find a short vector v in the lattice $\mathcal{L} = \sum_{i=1}^{n} \mathbb{Z}v_i$.
8. If $\|v\| > D$ declare 'f is irreducible'.
9. Else let f_1' be the polynomial with coefficient vector v.
10. Return $\gcd(f, f_1')$.

Correctness and time complexity - Correctness of the algorithm follows immediately from the prior discussion. In step 1, the value of prime p is $O(n \log(nA))$ and so using Algorithm 13 and 14, we can factor f over $\mathbb{F}_p$ (in step 3) in polynomial time. Since $p^{2^\ell} \leq E^2$, Hensel lifting in step 5 also takes polynomial time. Norm of g', $\|g'\| \leq \sqrt{n}E^2$. So the LLL algorithm in step 7 takes time $\mathsf{poly}(n, \log E) = \mathsf{poly}(n, \log A)$. Lastly, in step 10, gcd of two integer polynomials can be computed in polynomial time (using Chinese Remaindering). Therefore, the algorithm has an overall polynomial running time.

Application 2: Breaking Low Exponent RSA

In this section, we show an interesting application of the LLL algorithm in breaking the RSA cryptosystem. The method, due to Coppersmith [Cop97], shows the vulnerability of the RSA when the exponent of the public key is small and a significant fraction of the message is already known to all as a header. The key step is the use of lattice basis reduction in finding a root of a modular equation, when the root is much smaller than the modulus in absolute value.

Following the notation in section 2.1, let m be the message string and (n, e) be the public key, where e is the exponent and n is the modulus. The encrypted message is $c = m^e \mod n$. Suppose that the first ℓ bits of the message m is a secret string x, and the remaining bits of m form a header h that is known to all. In other words, $m = h \cdot 2^\ell + x$, where h is known and x in unknown. Since, $(h \cdot 2^\ell + x)^e = c \mod n$, we get an equation $g(x) = x^e + \sum_{i=0}^{e-1} a_i x^i = 0 \mod n$, where $a_0, \ldots, a_{e-1}$ are known and are all less than n. Decryption of c is equivalent to finding a root of g modulo n. We intend to solve this modular root finding problem using lattice basis reduction.

The idea is to transform the modular root finding problem into a root finding problem over integers. The latter problem can then be solved using the LLL algorithm. To achieve this, we are going to do something very similar to what has been done in Algorithm 21 for polynomial factorization. Solve for a nonzero $f \in \mathbb{Z}[x]$ of degree less than $k > e$, such that $f = gh \mod n$ for some $h \in \mathbb{Z}[x]$ and $\|f\|$ is 'small'. Treat k as a parameter.

Consider the polynomials $x^{k-e-1}g, \ldots, xg, g$ and $nx^{e-1}, \ldots, n$, and $v_1, \ldots, v_k$ be the coefficient vectors in $\mathbb{Z}^k$ associated with these k polynomials. Notice that, the vectors $v_1, \ldots, v_k$ are linearly independent and the volume of the lattice $\mathcal{L} = \sum_{i=1}^{k} \mathbb{Z}v_i$ is $\mathsf{vol}(\mathcal{L}) = n^e$. By Minkowski's theorem, the length of the shortest vector in $\mathcal{L}$ is at most $\sqrt{k}n^{\frac{e}{k}}$. It is easy to observe that the coefficient vector v_f of any solution f is in the lattice $\mathcal{L}$. Using LLL algorithm, we can find a short vector $v \in \mathbb{Z}^k$ in $\mathcal{L}$, in time $\mathsf{poly}(k, \log n)$, such that $\|v\| \leq 2^{\frac{k-1}{2}} \cdot \sqrt{k}n^{\frac{e}{k}}$. Let f be the

polynomial with v as its coefficient vector. Surely, $f = gh \bmod n$ and $\deg(f) < k$.

We know that the secret string x has absolute value at most 2^ℓ. Let $x = a < 2^\ell$ be a root of g modulo n. Therefore, a is also a root of f modulo n. The absolute value of $f(a)$ is bounded by,

$$|f(a)| \leq k \cdot \|v\| \cdot 2^{\ell(k-1)} \leq k \cdot 2^{\frac{k-1}{2}} \cdot \sqrt{k} n^{\frac{e}{k}} \cdot 2^{\ell(k-1)} = A \text{ (say)}.$$

Now, if it happens that $A < n$ then $f(a) = 0$ over $\mathbb{Z}$. This means, we can apply Algorithm 21 to factor f and thereby find the secret string x.

Let us fix some parameters to get a feel of how this argument works in practice. Let $e = 3$ and $k = 6$, then all we need to find x is,

$$6\sqrt{6} \cdot 2^{\frac{5}{2}} \cdot \sqrt{n} \cdot 2^{5\ell} < n \;\Rightarrow\; \ell < \frac{1}{10} \cdot \log n - \frac{1}{5} \cdot \log(48\sqrt{3}).$$

Therefore, if roughly $\frac{9}{10}$-th of the message is known then the remaining secret $\frac{1}{10}$-th part of the message can be recovered in polynomial time. Since the approach works in time polynomial in k and $\log n$, it is efficient only if k is small. This means, from the relation $A < n$, that e has to be small. This shows that a low exponent e along with a small fraction of the secret part of the message make RSA vulnerable to lattice based attacks.

4.6 Smooth numbers

Our final topic of discussion is about the intriguing subject of smooth numbers. Smooth numbers are integers with *small* prime factors. So, by definition, they are easy to factor. Study on the density of these numbers dates back to the work of Dickman [Dic30]. It is easy to see that most integers have at least one small prime factor. For example, every second integer has 2 as its factor, every third has 3 as factor and so on. After we divide an integer by its small factors we are left with another integer with large prime factors. Factoring an integer with large prime factors is potentially a difficult task. Quite intriguingly, many such hard factoring instances can be *reduced* to factoring few smooth numbers. We will see the details of such reductions, in a short while, in Dixon's *random squares* algorithm and the Quadratic Sieve algorithm.

Smooth numbers find important applications in other factoring algorithms, like the Number Field Sieve [LJMP90] and Lenstra's elliptic curve method [Jr.87]. As another application, we will discuss the *index-calculus method* for computing discrete logarithms []. Smooth numbers also play a crucial part in proving many analytic number theoretic results. To cite an example, the result on infinitude of Carmichael numbers [AGP94] depends on the density of *smooth primes*, that is prime p such that $p - 1$ is smooth.

We now state the result by Dickman on the density of smooth numbers. In fact, this particular estimate is due to Canfield, Erdös and Pomerance [CEP83]. An integer is called *y-smooth* if all its prime factors are less than y. Let $\psi(x, y)$ be the total number of y-smooth integers between 1 and x.

Lemma 4.9 *If $u = O(\frac{\log x}{\log \log x})$ then $\frac{\psi(x, x^{\frac{1}{u}})}{x} = u^{-u(1+o(1))}$, where $o(1) \to 0$ as $u \to \infty$.*

We would use this estimate in the Quadratic Sieve algorithm. But before we move on to the applications, let us get a feel of how to find $\psi(x, y)$. Let $y^u = x$ and $p_1, \ldots, p_k$ be the primes less than y. Then, by the Prime Number Theorem, $k \approx \frac{y}{\ln y}$. Since, $p_k^u \leq x$, all numbers of the form $p_1^{e_1} \ldots p_k^{e_k}$, with $\sum_{i=1}^k e_i \leq u$, are y-smooth and less than x. Hence,

$\psi(x,y) \geq \binom{k+u}{u} \geq (\frac{k}{u})^u \approx \frac{x}{(u \ln y)^u}$ (taking u to be an integer). Therefore, $\frac{\psi(x,x^{\frac{1}{u}})}{x} \geq (u \ln y)^{-u}$. The purpose of presenting this coarse estimate is not merely to show a similarity with the expression in Lemma 4.9. A sophisticated version of this argument would be used in Dixon's algorithm.

For further details on the density of smooth numbers, the reader may refer to the article by Andrew Granville [Gra08] in [BS08]. An exposition to smooth numbers and their various applications can also be found in the article by Carl Pomerance [Pom94].

Application 1: Integer factoring

The two factoring algorithms, we are going to present here, are based on one simple idea. Let N be the given odd integer that we want to factor and α, β be two other integers less than N such that $\alpha^2 = \beta^2 \bmod N$. If it happens that neither $\alpha = \beta \bmod N$ nor $\alpha = -\beta \bmod N$ then $\gcd(\alpha + \beta, N)$ (as well as, $\gcd(\alpha - \beta, N)$) yields a nontrivial factor of N. This basic idea is the cornerstone of many modern day factoring algorithms and was first introduced as a general scheme by Kraitchik [Kra26, Kra29]. To make the scheme work, we are faced with two immediate questions.

- How to find a square modulo N?
- How to find two 'distinct' roots of the square modulo N?

By 'distinct' we mean the condition, $\alpha \neq \pm\beta \bmod N$. The first question is rather easy to answer. A random integer is a square modulo N with high probability if N has few prime factors. Besides, one can always take an integer and simply square it modulo N. However, it is the second question that demands more attention. Much of the effort in both Dixon's algorithm and the Quadratic Sieve is centered around efficiently finding 'distinct' roots of a square and it is here that smooth numbers enter the scene. The idea of involving smooth numbers is based on an earlier work by Morrison and Brillhart [MB75].

Dixon's random square method

Let N be an odd composite with ℓ distinct prime factors. Choose an integer α randomly in the range $[1, N-1]$ and compute $\gamma = \alpha^2 \bmod N$. We will show later (in Lemma 4.10) that with reasonable probability γ is y-smooth for a suitably chosen y. Further, if we are lucky then the prime factorization of $\gamma = p_1^{e_1} \dots p_k^{e_k}$ could be such that each e_i is even. Here, $p_1, \dots, p_k$ are the primes less than y. This means, $\beta = p_1^{e_1/2} \dots p_k^{e_k/2}$ is also a root of γ. Notice that, the value of β depends solely on γ and is independent of which random root α is chosen initially. Since there are 2^ℓ roots of γ (assuming $\gcd(\gamma, N) = 1$), the probability that $\alpha = \pm\beta \bmod N$ is only $\frac{1}{2^{\ell-1}}$. Therefore, if all goes well, $\gcd(\alpha + \beta, N)$ is nontrivial. Unfortunately, the chance that all the e_i's are even is not quite sufficient to ensure good success rate of the algorithm. But there is a neat way around!

Instead of choosing a single α, suppose we choose $k+1$ random numbers in the range $[1, N-1]$ such that $\gamma_j = \alpha_j^2 \bmod N$ is y-smooth for every j, $1 \leq j \leq k+1$. Consider the vectors $v_j = (e_{j1} \bmod 2, \dots, e_{jk} \bmod 2)$ corresponding to the prime factorizations of $\gamma_j = p_1^{e_{j1}} \dots p_k^{e_{jk}}$ for $1 \leq j \leq k+1$. Since there are $k+1$ vectors in a k dimensional space over $\mathbb{F}_2$, there exists a collection of vectors, $v_1, \dots v_m$ (say) such that their sum (over $\mathbb{F}_2$) is zero. What this means is that the prime factorization of $\gamma = \gamma_1 \dots \gamma_m = p_1^{e_1} \dots p_k^{e_k}$ has every e_i even. Once again, it can be easily argued that $\beta = p_1^{e_1/2} \dots p_k^{e_k/2} \bmod N$ and $\alpha = \alpha_1 \dots \alpha_m \bmod N$ are 'distinct' roots

of γ mod N with high probability.

We need to fill in one missing piece to formalize the above approach. It is to show that, for a random α, $\gamma = \alpha^2 \bmod N$ is y-smooth with high probability. The following lemma proves this fact.

Lemma 4.10 *Let $p_1, \ldots, p_k$ be the primes less than y in increasing order, and $S(N, y)$ be the following set,*

$$S(N, y) = \{\alpha : 1 \leq \alpha \leq N - 1 \text{ and } \gamma = \alpha^2 \bmod N \text{ is } y\text{-smooth}\}.$$

If $\gcd(p_i, N) = 1$ for all $1 \leq i \leq k$ and $r \in \mathbb{Z}^+$ is such that $p_k^{2r} \leq N$ then $|S(N, y)| \geq \frac{k^{2r}}{(2r)!}$.

Proof: Let $N = q_1^{f_1} \ldots q_\ell^{f_\ell}$ be the prime factorization of N. Since N has ℓ prime factors, every element $\alpha \in \mathbb{Z}_N$ is naturally associated with an element v_α in $\{1, -1\}^\ell$ that represents its *quadratic character*. That is, the i^{th} index of v_α is 1 if α is a square modulo $q_i^{f_i}$ and -1 otherwise. Moreover, by Chinese Remaindering Theorem, if two elements in $\mathbb{Z}_N$ have the same quadratic character then their product is a square modulo N. Define a set T as,

$$T = \{\alpha : \alpha = p_1^{e_1} \ldots p_k^{e_k} \text{ where } \sum_{i=1}^{k} e_i = r\} \tag{2}$$

Surely, every element of T is less than or equal to $\sqrt{N}$ as $p_k^{2r} \leq N$. The 2^ℓ possible values of the quadratic character define a partition of the set T. Call the partition corresponding to the quadratic character $g \in \{1, -1\}^\ell$ as T_g, so that $T_g = \{\alpha \in T : v_\alpha = g\}$. Every pair of elements within the same partition can be multiplied to form a square modulo N that is also a y-smooth number less than N. But there could be repetitions as the same square can result from multiplying different pairs of elements. Using the fact that $\sum_{i=1}^{k} e_i = r$ (in equation 2), a simple counting argument shows that there can be at most $\binom{2r}{r}$ repetitions for each square thus obtained. Therefore,

$$|S(N, y)| \geq \frac{2^\ell}{\binom{2r}{r}} \sum_{g \in \{1,-1\}^\ell} |T_g|^2.$$

The factor 2^ℓ in the above expression is because every square in $\mathbb{Z}_N^\times$ has exactly 2^ℓ roots. Simplifying using Cauchy-Schwartz inequality, $|S(N, y)| \geq \frac{1}{\binom{2r}{r}} (\sum_{g \in \{1,-1\}^\ell} |T_g|)^2 = \frac{1}{\binom{2r}{r}} |T|^2$. Once again, using the fact that $\sum_{i=1}^{k} e_i = r$, size of the set T is $\binom{k+r-1}{r} \geq \frac{k^r}{r!}$ and hence, $|S(N, y)| \geq \frac{k^{2r}}{(2r)!}$. ∎

We are almost done. All we need to do now is fix a value for y. This we will do in the analysis of the following algorithm and show that the optimum performance is obtained when $y = e^{O(\sqrt{\ln N \ln \ln N})}$.

Time complexity - Using the Sieve of Eratosthenes, we can find the first k primes in step 1 in $O(k \log^2 k \log \log k)$ time. The divisions in step 2 take $O(k \cdot \mathsf{M}_\mathsf{I}(n))$ time, where $n = \ln N$. Each iteration of the `while`-loop in step 4 takes $O(\mathsf{M}_\mathsf{I}(n) \log n)$ operations for the gcd computation and modular operations in steps 5 and 6. Whereas, trial divisions in step 7 can be done in $O((k+n)\mathsf{M}_\mathsf{I}(n))$ time. In step 8, we can use Gaussian elimination to find the linearly dependent vectors in $O(k^3)$ time. Modular computations and gcd finding can be done in steps 9 and 10

22 Dixon's random square method

1. Find all the primes, $p_1, \ldots, p_k$, less than $y = e^{(2^{-1/2}+o(1))\sqrt{\ln N \ln\ln N}}$.
2. If $p_i | N$ for any $1 \leq i \leq k$, return p_i.
3. Set $i = 1$.
4. while $i \leq k+1$ do
5. Choose integer α_i randomly from $[1, N-1]$. If $\gcd(\alpha_i, N) \neq 1$ return it.
6. Else, compute $\gamma_i = \alpha_i^2 \mod N$.
7. If γ_i is y-smooth, let $v_{\gamma_i} = (e_{i1} \mod 2, \ldots, e_{ik} \mod 2)$ where $\gamma_i = p_1^{e_{i1}} \ldots p_k^{e_{ik}}$. Set $i = i+1$.
8. Find $I \subset \{1, \ldots, k+1\}$ such that $\sum_{i \in I} v_{\gamma_i} = 0$ over $\mathbb{F}_2$.
9. Let $\gamma = \prod_{i \in I} \gamma_i = p_1^{e_1} \ldots p_k^{e_k}$, $\beta = p_1^{e_1/2} \ldots p_k^{e_k/2} \mod N$ and $\alpha = \prod_{i \in I} \alpha_i \mod N$.
10. If $\gcd(\alpha + \beta, N) \neq 1$ return the factor. Else, goto step 3.

using $O(\mathsf{M_I}(n) \log n)$ operations. Finally, we need to bound the number of iterations of the while-loop.

By Lemma 4.10, expected number of iterations to find a y-smooth square is $\frac{N \cdot (2r)!}{k^{2r}} \leq N \cdot (\frac{2r \ln y}{y})^{2r}$, taking $k \approx \frac{y}{\ln y}$. Now, if we choose y such that $y^{2r} = N$ then the expected number of executions of the loop is bounded by $(k+1) \cdot n^{2r}$. One final thing to notice is that the algorithm fails to return a proper factor at step 10 with only a constant probability, meaning the expected number of times we need to jump to step 3 and re-run the algorithm is also a constant. Therefore, the expected running time of the algorithm is $O(k^3 + k^2 n^{2r+2}) = O(e^{\frac{3n}{2r}} + e^{\frac{2n}{2r}} n^{2r+2})$, as $y^{2r} = N$. This expression is minimized if $2r \approx \sqrt{\frac{2n}{\ln n}}$. This means, $y = e^{(2^{-1/2}+o(1))\sqrt{\ln N \ln\ln N}}$ and the expected time taken by Dixon's random square method is $e^{(2\sqrt{2}+o(1))\sqrt{\ln N \ln\ln N}}$.

The best known running time for Dixon's algorithm is $e^{(\sqrt{\frac{4}{3}}+o(1))\sqrt{\ln N \ln\ln N}}$, a result due to Pomerance [Pom82] and Vallée [Val89]. But this is surpassed by the work of Lenstra and Pomerance [JP92] which has an expected time complexity of $e^{(1+o(1))\sqrt{\ln N \ln\ln N}}$.

Despite the theoretical progress, the performance of these randomized algorithms are not very encouraging in practice. With the widespread use of the RSA cryptosystems in various commercial applications, there is just too much at stake. One cannot help but wonder if there are variants of these ideas that are more efficient in practice, which in turn may deem some of the RSA public keys as unsafe. The Quadratic Sieve method is one such practical improvement.

The Quadratic Sieve method

The Quadratic Sieve (QS), which was first proposed by Pomerance [Pom82, Pom84], is also based on Kraitchik's scheme of finding 'distinct' roots of a square modulo N. But unlike Dixon's algorithm, this method is deterministic with a heuristic analysis showing a time bound of $e^{(1+o(1))\sqrt{\ln N \ln\ln N}}$ operations. Although it has the same asymptotic complexity as the best (rigorous) deterministic algorithm, the QS is much more efficient in practice. As for the assumptions made in the analysis of the QS, in Pomerance's [Pom08b] own words "...perhaps we should be more concerned with what is *true* rather than what is *provable*, at least for the design of a practical algorithm.".

The Quadratic Sieve method generates a sequence of squares modulo N using the polynomial $x^2 - N$, by varying integer x from $\sqrt{N}$ to $\sqrt{N} + N^{o(1)}$. We could say $x^2 \mod N$ instead

of $x^2 - N$, they being the same as $x \leq \sqrt{N} + N^{o(1)}$. This step of deterministically generating squares modulo N is in contrast to Dixon's algorithm where x is chosen randomly and then $x^2 \bmod N$ is computed. As before, we are interested in those numbers in the sequence that are y-smooth (for some fixed y). Understanding the distribution of smooth numbers in this sequence is a difficult number theoretic problem. Instead, to make the analysis go through we assume that the sequence generates y-smooth numbers in the same frequency as numbers picked randomly from the range $[0, 2N^{\frac{1}{2}+o(1)}]$. Since $\sqrt{N} \leq x \leq \sqrt{N} + N^{o(1)}$, $x^2 - N$ is between 0 and roughly $X = 2N^{\frac{1}{2}+o(1)}$. In other words, we assume that a y-smooth number is encountered after about $\frac{X}{\psi(X,y)}$ numbers of the sequence $\{x^2 - N\}$ as x is varied from $\sqrt{N}$ to $\sqrt{N} + N^{o(1)}$. This is where the Quadratic Sieve becomes heuristic in nature. Although there is no rigorous theory to back this assumption, years of experimental analysis have strongly supported it. So let us proceed with it and see the step to which the QS owes both its name and its efficiency.

Recall that, in Dixon's algorithm, we test for y-smoothness of $x^2 \bmod N$ by trial division with all the k primes less than y. This takes $O((k+n)\mathsf{M}_{\mathsf{I}}(n))$ time per number, where $n = \ln N$. We cannot do better than this if we test for smoothness, one number at a time. But in our case the numbers originate from a well-defined sequence. One can hope to spend much less overall time by testing smoothness of a collection of numbers together. To draw the analogy, let us briefly visit the Sieve of Eratosthenes method.

Suppose we want to find all the primes less than B. If we sequentially test for primality of all the B numbers, we end up spending $\tilde{O}(B \log^2 B)$ time (using Miller-Rabin primality test). Instead, the Sieve of Eratosthenes starts with an array of size B with all entries initialized as 'unmarked'. At each step, the process finds the next 'unmarked' index $p > 1$ and 'marks' those entries that are higher multiples of p. In the end, only the prime indices remain 'unmarked'. The total time spent is $\sum_{\text{prime } p < B} \frac{B}{p} \log B$, where the $\log B$ factor is due to the increment of the counter by p for each subsequent 'marking'. Using the analytic bound $\sum_{\text{prime } p < B} \frac{1}{p} = O(\log \log B)$, the time complexity comes out to be $O(B \log B \log \log B)$. By finding the primes together, the Sieve spends $O(\log B \log \log B)$ operations per number on average, as opposed to $\tilde{O}(\log^2 B)$ operations for primality testing. Pomerance made the observation that this idea of sieving primes can be adapted to sieve smooth numbers from the sequence $\{x^2 - N\}$.

Let p be a prime less than y and $p \nmid N$. Then p divides $x^2 - N$ if x is $\pm c$ modulo p where c is a square root of N modulo p. Since $x^2 - N$ is also an element of the sequence $\{x^2 - N\}$, $x = \lfloor \sqrt{N} \rfloor + i$ for some index $0 \leq i \leq N^{o(1)}$. Hence, there are two fixed values $a, b < p$ such that $x^2 - N$ is divisible by p if and only if $i = a$ or $b \bmod p$. This means, if the sequence $\{x^2 - N\}$ is presented as an array with the i^{th} index containing $(\lfloor \sqrt{N} \rfloor + i)^2 - N$, then the numbers that are divisible by p are exactly placed at the indices $\{a + kp\}_{k \geq 0}$ and $\{b + kp\}_{k \geq 0}$.

A sieving process can start from index a (similarly, b) and increment the counter by p, iteratively, to get to the numbers in the sequence that are divisible by p. A number that is divisible by p is then replaced by the quotient obtained by dividing it by the highest power of p. After this sieving is done for every prime $p < y$, all those numbers that have changed to 1 are y-smooth. If B is the size of the array then the total sieving time is, $O(\sum_{\text{prime } p < y} \frac{B}{p} \log y) = O(B \log y \log \log y)$. How large should B be? Since there are k primes below y, just like Dixon's algorithm, we need $k+1$ y-smooth numbers. We have assumed before that one y-smooth number is present in (roughly) every $\frac{X}{\psi(X,y)}$ numbers of the sequence, where $X = 2N^{\frac{1}{2}+o(1)}$. Therefore, it is sufficient if we take $B \approx (k+1) \cdot \frac{X}{\psi(X,y)}$ for the analysis. The value of y is fixed in the analysis of the following algorithm.

23 Quadratic Sieve method

1. Find all primes, $p_1, \ldots, p_k \leq y$. If $p_i | N$ for any $1 \leq i \leq k$, return p_i.
2. Sieve the sequence $(\lceil N \rceil + j)^2 - N$, $0 \leq j \leq N^{o(1)}$, for $(k+1)$ y-smooth numbers, $\gamma_i = \alpha_i^2 - N = p_1^{e_{i1}} \ldots p_k^{e_{ik}}$, $1 \leq i \leq k+1$. Let $v_{\gamma_i} = (e_{i1} \mod 2, \ldots, e_{ik} \mod 2)$.
3. Find $I \subset \{1, \ldots, k+1\}$ such that $\sum_{i \in I} v_{\gamma_i} = 0$ over $\mathbb{F}_2$.
4. Let $\gamma = \prod_{i \in I} \gamma_i = p_1^{e_1} \ldots p_k^{e_k}$, $\beta = p_1^{e_1/2} \ldots p_k^{e_k/2} \mod N$ and $\alpha = \prod_{i \in I} \alpha_i \mod N$.
5. If $\gcd(\alpha + \beta, N) \neq 1$ return the factor.

Heuristic time complexity - As with Dixon's algorithm, the time complexity of the Quadratic Sieve is mainly contributed by step 2 and step 3. If $y = X^{\frac{1}{u}}$ then by Lemma 4.9, $\frac{X}{\psi(X,y)} \approx u^u$. So the total time spent in step 2 is heuristically, $O(k \cdot u^u) = O(X^{\frac{1}{u}} u^u)$, ignoring lower order terms. This expression is minimized when $u \approx \sqrt{\frac{2 \ln X}{\ln \ln X}}$, implying that $y = e^{(2^{-1/2} + o(1))\sqrt{\ln X \ln \ln X}}$. Taking $X = 2N^{\frac{1}{2}+o(1)}$, the time complexity of step 2 comes out to be $O(X^{\frac{1}{u}} u^u) = e^{(1+o(1))\sqrt{\ln N \ln \ln N}}$. Notice that the vector v_{γ_i} (in step 2) is fairly sparse, it has at most $\log N$ nonzero entries. So the matrix formed by the $k+1$ vectors, $v_{\gamma_1}, \ldots, v_{\gamma_{k+1}}$, contains at most $(k+1) \log N$ nonzero entries. Using Wiedemann's sparse matrix method [Wie86], step 3 can be implemented in $O((k+1) \cdot (k+1) \log N)) = e^{(1+o(1))\sqrt{\ln N \ln \ln N}}$ time.

For an excellent survey on the Quadratic Sieve and how it evolved from earlier factoring methods, refer to the articles [Pom08b], [Pom96] and [Pom84] by Carl Pomerance. Several enhancements of the QS are described in chapter 6 of [CP05].

Application 2: Index calculus method for discrete logarithms

The index calculus is a probabilistic method for computing discrete logarithms in groups that are endowed with a notion of 'smoothness'. An example of such a group is $\mathbb{F}_p^\times$, which can be naturally identified with the numbers less than p. An element of $\mathbb{F}_p^\times$ is 'smooth' if as an integer it is a smooth number. The basic idea of the index calculus method is simple, and apparently this too can be traced back to the work of Kraitchik [Kra26, Kra29]. It was later rediscovered and analyzed by several other mathematicians including Adleman [Adl79].

Recall that, the discrete logarithm problem over $\mathbb{F}_p^\times$ is the task of finding an integer $x \in [0, p-2]$ such that $a^x = b \mod p$, given a generator a and an arbitrary element b of $\mathbb{F}_p^\times$. Going by the conventional notation, we write $x = \log_a b \mod (p-1)$. As before, let y be the smoothness parameter, to be fixed later, and $p_1, \ldots, p_k$ be the primes less than y. The index calculus starts by picking a random integer $\alpha \in [0, p-2]$ and computing $a^\alpha \mod p$. If $a^\alpha \mod p$ is y-smooth, we factor it completely as $a^\alpha = p_1^{e_1} \ldots p_k^{e_k} \mod p$. This gives us the following linear equation in the *indices*,

$$\alpha = e_1 \log_a p_1 + \ldots + e_k \log_a p_k \mod (p-1). \tag{3}$$

Here, $\log_a p_1, \ldots, \log_a p_k$ are the unknowns. If we collect k such linear equations then 'quite likely' the equations are linearly independent and we can solve for the $\log_a p_i$'s modulo $p-1$. Now pick another random integer $\beta \in [0, p-2]$ and compute $a^\beta b \mod p$. If $a^\beta b \mod p$ is also y-smooth then we get another linear relation of the form,

$$\beta + \log_a b = f_1 \log_a p_1 + \ldots + f_k \log_a p_k \mod (p-1), \tag{4}$$

by factoring as, $a^\beta b = p_1^{f_1} \ldots p_k^{f_k} \mod p$. But this time $\log_a b$ is the only unknown and we can

simply solve it from the above equation. This is the basic idea of the index calculus method. To formalize it, we need to address the following three questions:

- How likely is it that $a^\alpha \bmod p$ and $a^\beta b \bmod p$ are y-smooth ?
- What is the chance that the k equations are 'linearly independent' modulo $p-1$?
- How do we solve linear equations modulo $p-1$?

We have already seen the answer to the first question. Since α and β are randomly chosen and a is a generator of $\mathbb{F}_p^\times$, both a^α and $a^\beta b$ modulo p are uniformly distributed among the integers in $[1, p-1]$. Therefore, the probability that each of them is y-smooth is $\frac{\psi(p-1,y)}{p-1}$. The second question is also not difficult to answer. Using a counting argument it can be shown that the determinant of the coefficient matrix formed by the k linear equations is invertible modulo $p-1$ with high probabilty. Also, solving a set of linear equations modulo $p-1$ (in fact any integer) is just like solving modulo a prime. In Gaussian elimination if we fail to invert an element then using that element we can easily factor $p-1$ and continue solving modulo the factors. In the end we can compose the different modular solutions using Chinese Remaindering. All these can be done in $\tilde{O}(k^3)$ time, hiding some polylog factors in p.

We are now ready to present the algorithm. We will show in the analysis that the optimum choice of y is $e^{(2^{-1}+o(1))\sqrt{\ln p \ln\ln p}}$.

24 Index calculus method

1. Find all the primes, $p_1, \ldots, p_k$, less than $y = e^{(2^{-1}+o(1))\sqrt{\ln p \ln\ln p}}$.
2. Set $i = 1$.
3. while $i \le k$ do
4. Choose integer α_i randomly from $[0, p-2]$. Compute $\gamma_i = a^{\alpha_i} \bmod p$.
5. If γ_i is y-smooth, let $v_i = (e_{i1}, \ldots, e_{ik})$ where $\gamma_i = p_1^{e_{i1}} \ldots p_k^{e_{ik}}$. Set $i = i+1$.
6. If $i = k+1$, check if $v_1, \ldots, v_k$ span $\mathbb{Z}_{p-1}^k$. If not, goto step 2.
7. Solve for $\log_a p_i$, $1 \le i \le k$ modulo $p-1$ using equation 3 by Gaussian elimination on $v_1, \ldots, v_k$, and Chinese remaindering to compose solutions.
8. Keep choosing integer β randomly from $[0, p-2]$ till $a^\beta b \bmod p$ is y-smooth.
9. Solve for $\log_a b$ using equation 4.

Time complexity - The total time spent by the algorithm is dominated by the time spent in the while-loop and the time spent in steps 7 and 8. In step 4, the probability that γ_i is a y-smooth number is about $\frac{\psi(p,y)}{p} \approx u^{-u}$, where $u = \frac{\ln p}{\ln y}$ (by Lemma 4.9). Checking if γ_i is y-smooth in step 5 takes roughly $O(k \log p)$ time. Since, $v_1, \ldots, v_k$ span $\mathbb{Z}_{p-1}^k$ with high probability, the expected time spent in the loop is $O(k^2 u^u \log p)$. Time taken for Gaussian elimination in step 7 is about $O(k^3)$, and the expected time spent in step 8 is $O(k u^u \log p)$. So, the expected total time spent by the algorithm is $O(k^3 + k^2 u^u \log p)$. This expression is minimized when $u \approx 2\sqrt{\frac{\ln p}{\ln\ln p}}$, implying that $y = e^{(2^{-1}+o(1))\sqrt{\ln p \ln\ln p}}$. Therefore, the expected time taken by the algorithm is $e^{(2+o(1))\sqrt{\ln p \ln\ln p}}$.

Pomerance [Pom87] showed that using an elliptic curve method for fast smoothness test, the complexity of the index calculus method can be brought down to $e^{(\sqrt{2}+o(1))\sqrt{\ln p \ln\ln p}}$. For further details on modifications of the index calculus method, refer to the survey by Schirokauer, Weber and Denny [SWD96].

5 Concluding remarks

In writing this article, our main motive has been to introduce the reader to some of the basic computational aspects of number theory and algebra. We do hope that the reader has gained some familiarity and insight into the subject from the material presented here. Almost inevitably though, we could not fit in many other interesting topics in this relatively short article. Elliptic curves and Number Field Sieve being two of them. Also, for the sake of brevity and simplicity of presentation, we have skipped finer details on some occassions and sometimes the algorithms we have presented are not the best possible. The reader would find the references more helpful in this regard to fill in the missing parts. For further reading, we recommend the excellent books by von zur Gathen and Gerhard [GG03], Shoup [Sho09] and Crandall and Pomerance [CP05], and the survey article by Buhler and Wagon [BW08].

A Appendix

A.1 The resultant

Let $\mathcal{R}$ be an integral domain and $\mathbb{F}$ be its field of fractions. Let f and g be two polynomials in $\mathcal{R}[x]$ of degree n and m, respectively. Assume that $\gcd(f,g)$ is the unique, monic largest common divisor of f and g over $\mathbb{F}$.

Lemma A.1 *The* $\gcd(f,g)$ *is nontrivial if and only if there exists polynomials* $s,t \in \mathbb{F}[x]$*, with* $\deg(s) < m$ *and* $\deg(t) < n$*, such that* $sf + tg = 0$*.*

Proof: Suppose $h = \gcd(f,g)$. If $\gcd(f,g) \neq 1$ then $\deg(h) > 1$. Now, if we take $s = \frac{g}{h}$ and $t = -\frac{f}{h}$ then $\deg(s) < m, \deg(t) < n$ and $sf + tg = 0$.

To show the other direction, suppose there exist s and t with $\deg(s) < m, \deg(t) < n$ and $sf + tg = 0$. If $\gcd(f,g) = 1$ then by unique factorization over $\mathbb{F}$, g should divide s. But, this is not possible as $\deg(s) < \deg(g)$. Hence $\gcd(f,g)$ is nontrivial. ∎

Let $f = \sum_{i=0}^{n} f_i x^i$ and $g = \sum_{j=0}^{m} g_j x^j$, where $f_i, g_j \in \mathbb{F}$ for $0 \leq i \leq n$ and $0 \leq j \leq m$. Suppose $s = \sum_{k=0}^{m-1} \alpha_k x^k$ and $t = \sum_{\ell=0}^{n-1} \beta_\ell x^\ell$. Treat the coefficients $\alpha_0, \ldots, \alpha_{m-1}$ and $\beta_0, \ldots, \beta_{n-1}$ as variables. Now, consider the relation $sf + tg = 0$. By multiplying s, f and t, g, and then equating the coefficients of x^i to zero for all $0 \leq i \leq n + m - 1$, we get a system of $n+m$ homogeneous linear equations in the variables $\alpha_0, \ldots, \alpha_{m-1}, \beta_0, \ldots, \beta_{n-1}$. The coefficient matrix of this linear system is called the Sylvester matrix of f and g, and is denoted by $S(f,g)$. It is easy to verify that $S(f,g)$ is the following $(n+m) \times (n+m)$ matrix.

$$S(f,g) = \begin{pmatrix}
f_n & & & & g_m & & & & \\
f_{n-1} & f_n & & & g_{m-1} & g_m & & & \\
\vdots & \vdots & \ddots & & \vdots & \vdots & \ddots & & \\
\vdots & \vdots & & f_n & g_1 & \vdots & & \ddots & \\
\vdots & \vdots & & f_{n-1} & g_0 & \vdots & & & \ddots \\
f_1 & \vdots & & \vdots & & g_0 & & & g_m \\
f_0 & \vdots & & \vdots & & & \ddots & & \vdots \\
 & f_0 & & \vdots & & & & \ddots & \vdots \\
 & & \ddots & \vdots & & & & & \ddots \\
 & & & f_0 & & & & & g_0
\end{pmatrix}.$$

The resultant of f and g is defined as, $\mathsf{Res}_x(f,g) = \det(S(f,g))$. By Lemma A.1, the above linear system has a nonzero solution if and only if $\gcd(f,g)$ is nontrivial. This has the following implication.

Lemma A.2 *The* $\gcd(f,g)$ *is nontrivial if and only if* $\mathsf{Res}_x(f,g) = \det(S(f,g)) = 0$.

Another useful fact about the resultant is the following.

Lemma A.3 *There exist* $s,t \in \mathcal{R}[x]$, *with* $\deg(s) < m$ *and* $\deg(t) < n$, *such that* $sf + tg = \mathsf{Res}_x(f,g)$.

Proof: If $\gcd(f,g) \neq 1$, then from Lemma A.1 and A.2 it follows that, there exist $s',t' \in \mathbb{F}[x]$, with $\deg(s') < m$ and $\deg(t') < n$, such that $s'f + t'g = 0 = \mathsf{Res}_x(f,g)$. Since a coefficient of s' or t' is of the form $\frac{a}{b}$, where $a,b \in \mathcal{R}$ and $b \neq 0$, by clearing out the denominators of the coefficients of s' and t' we get $s,t \in \mathcal{R}[x]$ such that $sf + tg = 0$. Clearly, $\deg(s) = \deg(s') < m$ and $\deg(t) = \deg(t') < n$.

Suppose $\gcd(f,g) = 1$. By extended Euclidean algorithm, there exist $s',t' \in \mathbb{F}[x]$, with $\deg(s') < m$ and $\deg(t') < n$, such that $s'f + t'g = 1$. Let $s' = \sum_{k=0}^{m-1} \alpha_k x^k$ and $t' = \sum_{\ell=0}^{n-1} \beta_\ell x^\ell$. Once again, by multiplying s', f and t', g, and then equating the coefficients of x^i for $0 \leq i \leq n+m-1$, we get a linear system in $\alpha_0, \ldots, \alpha_{m-1}, \beta_0, \ldots, \beta_{n-1}$ with $S(f,g)$ as the coefficient matrix. By Cramer's rule, the polynomials $s = \mathsf{Res}_x(f,g) \cdot s'$ and $t = \mathsf{Res}_x(f,g) \cdot t'$ both belong to $\mathcal{R}[x]$. Therefore, $sf + tg = \mathsf{Res}_x(f,g)$. ∎

A.2 Gram-Schmidt orthogonalization

Let $v_1, \ldots, v_n$ be linearly independent vectors in $\mathbb{R}^m$ and $\mathcal{V}$ be the space spanned by them. Gram-Schmidt orthogonalization is a technique to find orthogonal vectors $v_1^*, \ldots, v_n^*$ such that the space spanned by them is $\mathcal{V}$. We denote the dot product of two vectors u and w by $u \cdot w$ and $\|u\| = \sqrt{u \cdot u}$ is the 2-norm of u. The construction of the orthogonal vectors proceeds as follows,

$$\begin{aligned} v_1^* &= v_1 \quad \text{and} \\ v_i^* &= v_i - \sum_{j<i} \mu_{ij} v_j^* \quad \text{for } 2 \leq i \leq n \text{ where, } \mu_{ij} = \frac{v_i \cdot v_j^*}{v_j^* \cdot v_j^*} \text{ for } 1 \leq j < i. \end{aligned}$$

Define the projection matrix as $M = (\mu_{ij})_{1 \leq i,j \leq n}$ where $\mu_{ii} = 1$ for all i, $\mu_{ij} = 0$ for $j > i$ and $\mu_{ij} = \frac{v_i \cdot v_j^*}{v_j^* \cdot v_j^*}$ for $j < i$. Let V be the $n \times m$ matrix with $v_1, \ldots, v_n$ as the rows and V^* be the matrix with $v_1^*, \ldots, v_n^*$ as the rows. The following facts are easy to verify and are left as exercise.

Lemma A.4 *1. The vectors* $v_1^*, \ldots, v_n^*$ *are mutually orthogonal and the space spanned by them is* $\mathcal{V}$.

2. v_i^* *is the projection of* v_i *on the orthogonal complement of* $\mathcal{V}_{i-1}$, *the space spanned by* $v_1, \ldots, v_{i-1}$ *which is also the space spanned by* $v_1^*, \ldots, v_{i-1}^*$. *Hence* $\|v_i^*\| \leq \|v_i\|$ *for all* i.

3. $V = M \cdot V^*$.

4. $\det(M) = 1$ *and so if* $m = n$ *then* $\det(V) = \det(V^*)$.

References

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

[AGP94] W. R. Alford, A. Granville, and C. Pomerance. There are Infinitely Many Carmichael Numbers. *Annals of Mathematics*, 139:703–722, 1994.

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

[Ajt98] Miklós Ajtai. The Shortest Vector Problem in L_2 is NP-hard for Randomized Reductions (Extended Abstract). In *STOC*, pages 10–19, 1998.

[AKS04] Manindra Agrawal, Neeraj Kayal, and Nitin Saxena. Primes is in P. *Annals of Mathematics*, 160(2):781–793, 2004.

[AM94] Leonard M. Adleman and Kevin S. McCurley. Open problems in number theoretic complexity, II. In *ANTS*, pages 291–322, 1994.

[AM09] Divesh Aggarwal and Ueli Maurer. Breaking RSA Generically is Equivalent to Factoring. In *Advances in Cryptology - EUROCRYPT 2009*, volume 5479 of *Lecture Notes in Computer Science*, pages 36–53. Springer-Verlag, 2009.

[AR04] Dorit Aharonov and Oded Regev. Lattice Problems in NP $\cap$ coNP. In *FOCS*, pages 362–371, 2004.

[AS05] Manindra Agrawal and Nitin Saxena. Automorphisms of Finite Rings and Applications to Complexity of Problems. In *STACS*, pages 1–17, 2005.

[Bac84] Eric Bach. Discrete Logarithms and Factoring. Technical report, Berkeley, CA, USA, 1984.

[BCS97] Peter Bürgisser, Michael Clausen, and Mohammad A. Shokrollahi. *Algebraic Complexity Theory*, volume 315 of *Grundlehren der mathematischen Wissenschaften*. Springer-Verlag, 1997.

[Ber67] Elwyn Berlekamp. Factoring Polynomials Over Finite Fields. *Bell System Technical Journal*, 46:1853–1859, 1967.

[Ber70] E. R. Berlekamp. Factoring Polynomials Over Large Finite Fields. *Mathematics of Computation*, 24(111):713–735, 1970.

[BK78] R. P. Brent and H. T. Kung. Fast Algorithms for Manipulating Formal Power Series. *J. ACM*, 25(4):581–595, 1978.

[Bon99] Dan Boneh. Twenty Years of Attacks on the RSA Cryptosystem. *Notices of the AMS*, 46:203–213, 1999.

[BS08] Joseph P. Buhler and Peter Stevenhagen, editors. *Algorithmic Number Theory: Lattices, Number Fields, Curves and Cryptography*, volume 44 of *Mathematical Sciences Research Institute Publications*. Cambridge University Press, 2008.

[BSKR06] Eli Ben-Sasson, Swastik Kopparty, and Jaikumar Radhakrishnan. Subspace Polynomials and List Decoding of Reed-Solomon Codes. In *FOCS*, pages 207–216, 2006.

[BW08] Joe Buhler and Stan Wagon. Basic algorithms in number theory. *Algorithmic Number Theory: Lattices, Number Fields, Curves and Cryptography, MSRI Publications*, 44:25–68, 2008.

[Car10] R. D. Carmichael. Note on a New Number Theory Function. *Bull. Amer. Math. Soc.*, 16:232–238, 1910.

[CEP83] Earl Rodney Canfield, Paul Erdös, and Carl Pomerance. On a problem of Oppenheim concerning "factorisatio numerorum". *Journal of Number Theory*, 17:1–28, 1983.

[Cop93] Don Coppersmith. Modifications to the number field sieve. *Journal of Cryptology*, 6:169–180, 1993.

[Cop97] Don Coppersmith. Small Solutions to Polynomial Equations, and Low Exponent RSA Vulnerabilities. *J. Cryptology*, 10(4):233–260, 1997.

[CP05] Richard Crandall and Carl Pomerance. *Prime Numbers: A Computational Perspective*. Springer, USA, 2005.

[CT65] James W. Cooley and John W. Tukey. An Algorithm for the Machine Calculation of Complex Fourier Series. *Mathematics of Computation*, 19(90):297–301, 1965.

[CW87] Don Coppersmith and Shmuel Winograd. Matrix Multiplication via Arithmetic Progressions. In *STOC*, pages 1–6, 1987.

[DH76] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, 1976.

[Dic30] Karl Dickman. On the frequency of numbers containing prime factors of a certain relative magnitude. *Arkiv för Mathematik, Fysik*, 22:1–14, 1930.

[Dix81] John D. Dixon. Asymptotically Fast Factorization of Integers. *Mathematics of Computation*, 36(153):255–260, 1981.

[DKSS08] Anindya De, Piyush P. Kurur, Chandan Saha, and Ramprasad Saptharishi. Fast integer multiplication using modular arithmetic. In *STOC*, pages 499–506, 2008.

[Evd94] Sergei Evdokimov. Factorization of polynominals over finite fields in subexponential time under GRH. In *ANTS*, pages 209–219, 1994.

[Fou85] E Fouvry. Theoreme de Brun-Titchmarsh; application au theoreme de Fermat. *Invent. Math.*, 79:383–407, 1985.

[Für07] Martin Fürer. Faster integer multiplication. In *STOC*, pages 57–66, 2007.

[GG98] Oded Goldreich and Shafi Goldwasser. On the Limits of Non-Approximability of Lattice Problems. In *STOC*, pages 1–9, 1998.

[GG03] Joachim Von Zur Gathen and Jurgen Gerhard. *Modern Computer Algebra*. Cambridge University Press, New York, NY, USA, 2003.

[Gor93] Daniel M. Gordon. Discrete Logarithms in *GF(p)* Using the Number Field Sieve. *SIAM J. Discrete Math.*, 6(1):124–138, 1993.

[Gra08] Andrew Granville. Smooth numbers: computational number theory and beyond. *Algorithmic Number Theory: Lattices, Number Fields, Curves and Cryptography, MSRI Publications*, 44:267–323, 2008.

[GS99] Venkatesan Guruswami and Madhu Sudan. Improved decoding of Reed-Solomon and algebraic-geometry codes. *IEEE Transactions on Information Theory*, 45(6):1757–1767, 1999.

[Gur04] Venkatesan Guruswami. *List Decoding of Error-Correcting Codes (Winning Thesis of the 2002 ACM Doctoral Dissertation Competition)*, volume 3282 of *Lecture Notes in Computer Science*. Springer, 2004.

[Hen18] Kurt Hensel. Eine neue Theorie der algebraischen Zahlen. *Mathematische Zeitschift*, 2:433–452, 1918.

[HR07] Ishay Haviv and Oded Regev. Tensor-based hardness of the shortest vector problem to within almost polynomial factors. In *STOC*, pages 469–477, 2007.

[JP92] Hendrik W. Lenstra Jr. and Carl Pomerance. A Rigorous Time Bound for Factoring Integers. *Journal of the American Mathematical Society*, 5:483–516, 1992.

[JP05] Hendrik W. Lenstra Jr. and Carl Pomerance. Primality testing with Gaussian periods, July 2005. Available from `http://www.math.dartmouth.edu/ carlp/PDF/complexity12.pdf`.

[Jr.87] Hendrik W. Lenstra Jr. Factoring integers with elliptic curves. *Annals of Mathematics*, 126:649–673, 1987.

[Jr.08] Hendrik W. Lenstra Jr. Lattices. *Algorithmic Number Theory: Lattices, Number Fields, Curves and Cryptography, MSRI Publications*, 44:127–181, 2008.

[Jus76] Jørn Justesen. On the complexity of decoding Reed-Solomon codes. *IEEE Transactions on Information Theory*, 22(2):237–238, 1976.

[Kal85] Erich Kaltofen. Polynomial-time reductions from multivariate to bi- and univariate integral polynomial factorization. *SIAM J. Comput.*, 14(2):469–489, 1985.

[Kal89] Erich Kaltofen. Factorization of Polynomials Given by Straight-Line Programs. In *Randomness and Computation*, pages 375–412. JAI Press, 1989.

[Kat01] Stefan Katzenbeisser. *Recent Advances in RSA Cryptography*. Kluwer Academic Publishers, Norwell, MA, USA, 2001.

[Kho05] Subhash Khot. Hardness of approximating the shortest vector problem in lattices. *J. ACM*, 52(5):789–808, 2005.

[KM04] Neal Koblitz and Alfred J. Menezes. A Survey of Public-Key Cryptosystems. *SIAM Rev.*, 46(4):599–634, 2004.

[Kob87] Neal Koblitz. Elliptic Curve Cryptosystems. *Mathematics of Computation*, 48(177):203–209, 1987.

[Kra26] M. Kraitchik. Théorie des Nombres, Tome II. 1926.

[Kra29] M. Kraitchik. Recherches sur la Théorie des Nombres, Tome II. 1929.

[KS91] Erich Kaltofen and B. David Saunders. On Wiedemann's Method of Solving Sparse Linear Systems. In *AAECC*, pages 29–38, 1991.

[KS98] Erich Kaltofen and Victor Shoup. Subquadratic-time factoring of polynomials over finite fields. *Math. Comput.*, 67(223):1179–1197, 1998.

[KS05] Neeraj Kayal and Nitin Saxena. On the Ring Isomorphism and Automorphism Problems. In *IEEE Conference on Computational Complexity*, pages 2–12, 2005.

[KU08] Kiran S. Kedlaya and Christopher Umans. Fast Modular Composition in any Characteristic. In *FOCS*, pages 146–155, 2008.

[Len00] Arjen K. Lenstra. Integer factoring. *Des. Codes Cryptography*, 19(2/3):101–128, 2000.

[LJL82] Arjen K. Lenstra, Hendrik W. Lenstra Jr., and László Lovász. Factoring polynomials with rational coefficients. *Mathematische Annalen*, 261(4):515–534, 1982.

[LJMP90] Arjen K. Lenstra, Hendrik W. Lenstra Jr., Mark S. Manasse, and John M. Pollard. The Number Field Sieve. In *STOC*, pages 564–572, 1990.

[LN94] Rudolf Lidl and Harald Neiderreiter. *Introduction to finite fields and their applications*. Cambridge University Press, 1994.

[MB75] M. Morrison and J. Brillhart. A method of factoring and the factorization of F_7. *Mathematics of Computation*, 29:183–205, 1975.

[McC90] Kevin S. McCurley. The Discrete Logarithm Problem. In *Proceedings of Symposia in Applied Mathematics*, volume 42, pages 49–74, 1990.

[Mic00] Daniele Micciancio. The Shortest Vector Problem is NP-hard to approximate to within some constant. *SIAM Journal on Computing*, 30(6):2008–2035, 2000.

[Mig74] Maurice Mignotte. An Inequality About Factors of Polynomials. *Mathematics of Computation*, 28(128):1153–1157, 1974.

[Mil76] Gary L. Miller. Riemann's hypothesis and tests for primality. *J. Comput. Syst. Sci.*, 13(3):300–317, 1976.

[Min10] Hermann Minkowski. *Geometrie der Zahlen*. B. G. Teubner, Leipzig, 1910.

[MS81] Florence Jessie MacWilliams and Neil J. A. Sloane. *The Theory of Error-Correcting Codes*. North Holland, Amsterdam, 1981.

[MW99] Ueli M. Maurer and Stefan Wolf. The Relationship Between Breaking the Diffie–Hellman Protocol and Computing Discrete Logarithms. *SIAM J. Comput.*, 28(5):1689–1721, 1999.

[Odl00] Andrew M. Odlyzko. Discrete Logarithms: The Past and the Future. *Des. Codes Cryptography*, 19(2/3):129–145, 2000.

[PH78] S. Pohlig and M. Hellman. An improved algorithm for computing logarithms over GF(p) and its cryptographic significance. *IEEE Transactions on Information Theory*, 24:106–110, 1978.

[Pol78] J. M. Pollard. Monte Carlo Methods for Index Computation mod p. *Mathematics of Computation*, 32(143):918–924, 1978.

[Pom82] Carl Pomerance. Analysis and comparison of some integer factoring algorithms. In H.W. Lenstra Jr. and R. Tijdeman, editors, *Computational Methods in Number Theory*, volume 154 of *Math. Centrum Tract*, pages 89–139, 1982.

[Pom84] Carl Pomerance. The Quadratic Sieve Factoring Algorithm. In *EUROCRYPT*, pages 169–182, 1984.

[Pom87] Carl Pomerance. Fast, rigorous factorization and discrete logarithm algorithms. In *Discrete Algorithms and Complexity*, pages 119–143. Academic Press, 1987.

[Pom94] Carl Pomerance. The Role of Smooth Numbers in Number Theoretic Algorithms. In *Proceedings of the International Congress of Mathematicians*, 1994.

[Pom96] Carl Pomerance. A Tale of Two Sieves. *Notices of the American Mathematical Society*, 43:1473–1485, 1996.

[Pom08a] Carl Pomerance. Elementary thoughts on discrete logarithms. *Algorithmic Number Theory: Lattices, Number Fields, Curves and Cryptography, MSRI Publications*, 44:385–396, 2008.

[Pom08b] Carl Pomerance. Smooth numbers and the quadratic sieve. *Algorithmic Number Theory: Lattices, Number Fields, Curves and Cryptography, MSRI Publications*, 44:69–81, 2008.

[Rab80] Michael O. Rabin. Probabilistic algorithm for testing primality. *J. Number Theory*, 12(1):128–138, 1980.

[Ros] Michael Rosenblum. A fast algorithm for rational function approximations. Available from `http://people.csail.mit.edu/madhu/FT01/notes/rosenblum.ps`.

[RS60] I. S. Reed and G. Solomon. Polynomial Codes Over Certain Finite Fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960.

[RSA78] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Commun. ACM*, 21(2):120–126, 1978.

[Sch08] Oliver Schirokauer. The impact of the number field sieve on the discrete logarithm problem in finite fields. *Algorithmic Number Theory: Lattices, Number Fields, Curves and Cryptography, MSRI Publications*, 44:397–420, 2008.

[Sho09] Victor Shoup. *A Computational Introduction to Number Theory and Algebra*. Cambridge University Press, New York, 2009. Available from `http://shoup.net/ntb/`.

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

[Ste08] Peter Stevenhagen. The Number Field Sieve. *Algorithmic Number Theory: Lattices, Number Fields, Curves and Cryptography, MSRI Publications*, 44:83–100, 2008.

[Str69] V. Strassen. Gaussian Elimination is not Optimal. *Numerische Mathematik*, 13(3):354–356, 1969.

[Sud] Madhu Sudan. Notes on an efficient solution to the rational function interpolation problem. Available from `http://people.csail.mit.edu/madhu/FT01/notes/rational.ps`.

[Sud97] Madhu Sudan. Decoding of Reed Solomon Codes beyond the Error-Correction Bound. *J. Complexity*, 13(1):180–193, 1997.

[SWD96] Oliver Schirokauer, Damian Weber, and Thomas F. Denny. Discrete logarithms: The effectiveness of the index calculus method. In *ANTS*, pages 337–361, 1996.

[Val89] B. Vallée. Provably fast integer factoring with quasi-uniform small quadratic residues. In *STOC '89: Proceedings of the twenty-first annual ACM symposium on Theory of computing*, pages 98–106, 1989.

[vzGP01] Joachim von zur Gathen and Daniel Panario. Factoring Polynomials Over Finite Fields: A Survey. *J. Symb. Comput.*, 31(1/2):3–17, 2001.

[vzGS92] Joachim von zur Gathen and Victor Shoup. Computing Frobenius Maps and Factoring Polynomials. *Computational Complexity*, 2:187–224, 1992.

[WB86] Lloyd R. Welch and Elwyn R. Berlekamp. Error correction for algebraic block codes, December 1986. U.S. Patent Number 4,633,470.

[Wie86] Douglas H. Wiedemann. Solving sparse linear equations over finite fields. *IEEE Transactions on Information Theory*, 32(1):54–62, 1986.