Analysis and implementation of algorithms in number theory

Preliminary Arizona Winter School, 2025

Juanita Duque-Rosero

Department of Mathematics and Statistics, Boston University

juanita@bu.edu

Introduction

These lecture notes accompany the lectures for the Preliminary Arizona Winter School 2025: Algorithms in number theory. The main references I used are [Coh93, Har21]. Other useful references are [Coh00, Ste, Voi21, vzGG13].

The course consists of an exploration of the algorithms and computational ideas that power modern algebra and number theory. We will start with the basics: analyzing what makes an algorithm *efficient*, and working through classic methods in integer arithmetic and linear algebra. These techniques will come up again and again in the rest of the lectures.

From there, we will study algebraic numbers and number fields. We will see how to represent them and do arithmetic with them. We then move on to working with rings of integers, discriminants, and integral bases. Then, we go back to algorithmic linear algebra to look at the LLL Algorithm and its applications to the study of number fields. Finally, we study ideals, class groups, and units and we pull everything together with examples from imaginary quadratic fields.

Explicit computations

The ideal way to follow these notes is to try examples in your favorite computer algebra system. I personally use Magma, but you are welcome to use whatever you prefer (SageMath, PARI/GP, Oscar, etc.). You can find Magma examples here. Also, here is a list of random tricks that I have compiled and here is a scavenger hunt to get you started.

Prerequisites

This course will assume fluency with algebra at a beginning graduate level and familiarity with the basic objects of algebraic number theory (such as number fields and their rings of integers). Some good references are [Mil, Lan94].

A note from the author

Despite my best efforts, these notes will contain typos. If you spot any, please feel free to email me, I appreciate your help!

An (irrelevant) note from the author

When I think about foundational algorithms that have really made a difference in my own number theory research, *Gröbner bases* are on top of my list. They are very useful for arithmetic geometry, and they have "saved" my work more than once. Unfortunately, I could not find space in these lecture notes to cover them. If you are curious, the book [CLO15] is beautifully written, and has a lot of the relevant theory.

Acknowledgments

I thank John Voight for helpful discussions while I was preparing these notes. Also, thank you to David Harvey for his helpful feedback on an earlier draft. Thanks to Jennifer Balakrishnan, Jerson Caro, Aashraya Jha, and Padmavathi Srinivasan for their valuable input.

Lecture 1: Arithmetic and linear algebra

Even when you are doing advanced computation, you will end up using basic algorithms in arithmetic and linear algebra. In this first lecture, we explore some of those algorithms to compute basic arithmetic, greatest common divisors, and matrices normal forms. They will be useful in the rest of the course. This lecture follows [Har21, §2] and [vzGG13, Chapter 2]. Other useful references are [BZ11] and [Coh93].

1.1 Analysis of algorithms

When analyzing the effectiveness of an algorithm, we can consider many factors, such as the amount of memory used, or the number of operations required for completion. We start by introducing some useful notation.

1.1.1 Big-O notation

Definition 1.1. Let $f(n)$ and $g(n)$ be functions defined on the natural numbers. We say that $f(n)$ is big-O of $g(n)$, and write

$$f(n) = O(g(n)),$$

if there exist constants $C > 0$ and $n_0 \in \mathbb{N}$ such that

$$|f(n)| \leq C\,|g(n)|$$

for all $n \geq n_0$.

You can think of $O(g(n))$ as describing a multiplicative bound on the growth of $f(n)$, for large n. That is, $f(n)$ does not grow faster than a constant multiple of $g(n)$ for sufficiently large inputs.

Lemma 1.2. *Let $f(n)$, $g(n)$, $a(n)$, and $b(n)$ be functions satisfying $f(n) = O(g(n))$ and $a(n) = O(b(n))$. Then*

$$f(n) + a(n) = O\big(\max(|g(n)|, |b(n)|)\big)$$

and

$$f(n)a(n) = O\big(g(n)b(n)\big).$$

Exercise 1.3. Prove Lemma 1.2.

In this lecture, we will use the term *input size* for an algorithm, which depends on how the data is represented. Precise analysis sometimes requires care in defining what counts as a single operation (for example, adding two numbers versus multiplying two numbers), but for most algorithms in arithmetic and linear algebra we will focus on basic operations such as integer addition, multiplication, and comparisons.

Lemma 1.4. *Let $f(n)$ be a function defined over $\mathbb{N}$. Then $f(n) = O(\log(n))$ if and only if $f(n) = O(\log_k(n))$ for any $k > 1$. In this case, we write $O(\log(n)) = O(\log_k(n))$.*

Proof. We recall the relation between logarithms: $\log(n) = \log_k(n)/\log_k(10)$. This shows that $f(n) = O(\log(n))$ if and only if there is a constant C such that for all large enough n,

$$|f(n)| \leq C|\log(n)| = C\frac{|\log_k(n)|}{|\log_k(10)|} = \frac{C}{\log_k(10)}|\log_k(n)| = C'|\log_k(n)|.$$

This shows that $f(n) = O(\log(n))$ if and only if $f(n) = O(\log_k(n))$. $\square$

1.1.2 Computational models

When deciding on the complexity of an algorithm, it is of vital importance to decide which computational model we are considering. That is, setting a formal framework for what it means to "compute". We have many choices available, for example, Turing machine, Random Access Machine, or even quantum computer. Each choice makes some analysis cleaner or more awkward. One good reference to learn about this is [Pap94]. In this course, we follow the choice of [Har21] and use the deterministic multitape Turing model. Even though this will not be the focus of this course, we give an intuition of what this model does.

(Deterministic) Turing machines

We can think of a (deterministic) Turing machine as an infinite tape together with a head. The head moves along the tape and can read and modify the contents of each position. For a given Turing machine, you need to pick the alphabet (possible symbols in the tape), a set of states (what is the machine doing at a given time), and a function describing the behavior of the machine given the state and symbol.

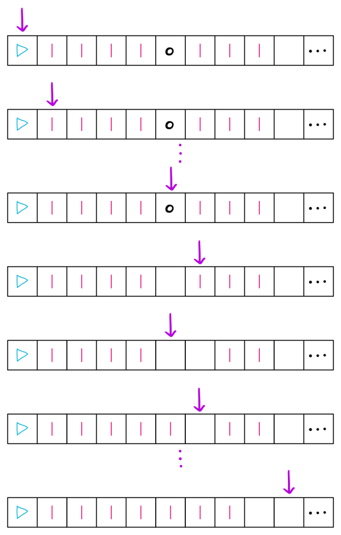

Figure 1.1: A Turing machine to add $4 + 3$.

Example 1.5. Just for fun, we can consider a very simple Turing machine that adds $4 + 3$. We represent the numbers $4 = ||||$ and $3 = |||$. The head always starts at the start symbol $\triangleright$. It moves right until it encounters the symbol $\circ$. Then, it deletes $\circ$, changes it state to "adding", and moves to the right.

While in this state, whenever we encounter a symbol $|$, we delete it, move to the left, add a symbol $|$, and move twice to the right. When the head reads a blank at this state, the new state is “halt” since we are done with the addition. You can see a picture of this in Figure 1.1. You can check that the total number of steps this machine takes to get to “halt” is 15.

Exercise 1.6. Can you describe a Turing machine that adds $4+3$ using less steps than in Example 1.5 but using the same alphabet $\{\triangleright, \square, \circ, |\}$?

Exercise 1.7. Can you change the alphabet and describe a Turing machine that adds $4+3$ using less steps than in Example 1.5?

Deterministic multitape Turing machines

Instead of working with one tape in a Turing machine, we can consider the case when we have finitely many tapes and one head that reads one position on each tape. It turns out that multitape Turing machines are as capable as Turing machines, but faster. For instance, one can describe a multitape Turing machine that performs the operation of Example 1.5 in 9 less steps!

Definition 1.8. The complexity of a multitape Turing machine model refers to the number of steps executed by the machine over the course of a computation. Each step that a Turing machine takes is called a bit operation.

Remark 1.9. The complexity of an algorithm depends on what the *alphabet* of the multitape Turing machine is. It it not the same to have a machine that reads any integer, to a machine that only reads $|$ and $\circ$, so we need to be specific about the alphabet.

In practice, we describe the complexity of turing Machines by writing the number of steps in using Big-O notation for functions on the size (number of bits) of the input. We also note that another useful thing to consider is the space complexity, i.e., the amount of memory used by a computation. We will only focus on the time complexity, i.e. the number of steps that it takes to terminate. For certain algorithms the space complexity becomes the main bottleneck in practice, so it is good to remember that this might be a problem.

Definition 1.10. Given an algorithm that takes an input of n-bits and requires at most $O(f(n))$ bit operations to complete, we say the algorithm runs in time $O(f(n))$ or has complexity $O(f(n))$.

Remark 1.11. Because we want to access the number theory and not the computational complexity, through these lectures, we will not be very specific about the particular construction of Turing machines for each algorithm.

1.1.3 An example: addition of integers.

The first question we need to answer is how to represent the objects that we want to input, or what is the alphabet of our Turing machine. In the next section, we discuss representing integers more generally; here, we assume integers are given in binary.

Example 1.12. The integer 431 is represented by the 9-digit binary number 110101111 since

$$431 = 1 \cdot 2^8 + 1 \cdot 2^7 + 0 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0.$$

Now we are ready to add two integers together. Without loss of generality, we can assume that the length of the expansions is n (we can always pick the maximum length, and then add zeros to the shortest one). Let's look at Algorithm 1.13.

Algorithm 1.13 (Integer addition [HvdH21, Algorithm 2.1]).

The input is two binary expansions $a = (a_{n-1}, \ldots, a_1, a_0)$ and $b = (b_{n-1}, \ldots, b_1, b_0)$ of length n. This algorithm outputs the binary expansion for the integer c that is the sum $a + b$.

```
1. Set $\gamma_0 := 0$.
2. For $i = 0, \ldots, n-1$ do
    (a) Set $c_i := a_i + b_i + \gamma_i$.
    (b) If $c_i \geq 2$, set $c_i := c_i - 2$ and $\gamma_{i+1} := 1$; otherwise, set $\gamma_{i+1} := 0$.
3. Set $c_n := \gamma_n$.
```

Return $c = (c_n, \ldots, c_1, c_0)$.

Remark 1.14. The description of the algorithm does not explicitly give a multitape Turing machine (§1.1.2), but you should convince yourself that it gives you all the information you need to rigorously define the machine.

Theorem 1.15. *The complexity of Algorithm 1.13 is $O(n)$.*

Proof. We analyze the complexity step by step. It is useful to recall arithmetic of big-O notation from Lemma 1.2. Step 1 requires one bit operation, so its time is $O(1)$. Step 2 iterates n times. Each iteration performs two bit addition operations, potentially subtracts 2, and sets the carry bit. Each of these operations is a constant-time word operation, so each iteration takes $O(1)$ time. In total, the complexity of Step 2 is $O(n)O(1) = O(n)$. Finally, Step 3 sets one bit, which is time $O(1)$. Altogether, the total complexity of the algorithm is

$$O(1) + O(n) + O(1) = O(n).$$

□

Remark 1.16. The constant n in Theorem 1.15 denotes the length of the inputs, so if a and b are the integers we want to add, then

$$n = \log_2(\max\{a, b\}) = O(\log(\max\{a, b\})),$$

where the last equality follows from Lemma 1.4.

1.2 Integer arithmetic

With basic notation established, we now explore algorithms for integer arithmetic. First, how do we represent integers? Then we move on to addition, multiplication, division, and greatest common divisors.

1.2.1 Representing integers

We will represent numbers in binary, with another bit to represent the sign.

That could be the only line of this subsection. The following is a small detour relates to expressing integers in actual computers.

Modern laptops use a 64-bit processor. That means that each integer can be up to $2^{64}-1$ in value (unsigned), or from -2^{63} to $2^{63}-1$ (signed). The CPU can process (add, multiply, etc.) two of these 64-bit numbers in one operation.

We can represent larger integers by an array of 64-bit words as follows

$$a = (-1)^s \sum_{i=0}^{n} a_i 2^{64i}, \tag{1.17}$$

where $s \in \{0, 1\}$, $0 \le n + 1 < 2^{63}$, and $a_i \in \{0, \ldots, 2^{64} - 1\}$. The numbers a_i are the *digits in base* 2^{64} of a.

Definition 1.18. For an integer a, its standard representation is given by the array

$$(s \cdot 2^{63} + n + 1, a_0, \ldots, a_n),$$

where s and a_i are as in (1.17) and a_n is nonzero if $a \neq 0$. If the standard representation of a has length n we call a an n-bit integer.

1.2.2 Addition

To add integers using their binary representation, we can just use Algorithm 1.13 that has complexity $O(n)$, where n is the number of bits of the integers (in binary). We usually just pick the largest number of bits and set that as n. By Remark 1.16, the complexity of addition is the same as $O(\log(\max(a, b)))$, where a and b are the integers we want to add.

If you went on the detour about the standard representation, you can modify Step 2 (b) of Algorithm 1.13 by

$$\text{If } c_i \ge 2^{64}, \text{ then set } c_i = c_i - 2^{64} \text{ and } \gamma_{i+1} = 1.$$

1.2.3 Multiplication

With addition, we noted that the naive algorithm to add integers (Algorithm 1.13) has complexity $O(n)$, where n is the number of bits. The naive algorithm that we use to multiply integers runs in time $O(n^2)$. This is usually fine for smaller integers, but more efficiency is needed for larger integers.

Exercise 1.19. You can check the time that it takes Magma to run one line by writing `time` at the beginning of the line:

```
> time 2^115032204*3^473444585;
Time: 312.990
```

Can you find two large integers (but maybe not as large as above) for which multiplication takes longer than 0 seconds? How big are your integers? How long does it take to add them?

Exercise 1.20. Write and analyze an algorithm that implements naive multiplication for integers. Your algorithm should use $O(mn)$ word operations, where m and n are the number of bits of the integers you are multiplying.

The theoretical state of the art is the Algorithm presented in [HvdH21]. This algorithm has complexity $O(n \log n)$ and is believed to be optimal but maybe not practical.

In practice, many computer algebra systems use the GMP library, which is a "free library for arbitrary precision arithmetic". GMP implements an algorithm whose theoretical complexity comes very close to $O(n \log n)$ [Har21, Remark 2.12].

1.2.4 Division

Division of large integers is typically accomplished using algorithms based on repeated subtraction, long division, or more advanced methods such as Newton-Raphson iteration for reciprocal approximation. For most practical purposes, the classical long division algorithm suffices.

In general, since $\mathbb{Z}$ is an Euclidean domain, given $a, b \in \mathbb{Z}$, the division algorithm to divide a by b should return the unique integers q and r with $0 \leq r < b$ such that $a = qb + r$. We call r the remainder of dividing a by b, or a modulo b, and denote it as $\mathrm{rem}(a, b)$.

Exercise 1.21. Describe the classical long division algorithm for binary integers. Prove that the algorithm takes time $O((n - m)m)$, where the integers have m and n bits, respectively.

Just like with multiplication, we can improve this bound. A division algorithm that combines fast multiplication with Newton's method gives the same complexity as multiplication: $O(n \log n)$.

1.2.5 Greatest common divisors

The greatest common divisor (gcd) of two integers a and b, denoted $\gcd(a, b)$, is the largest integer that divides both. The standard method to compute gcd's is the Euclidean Algorithm (Algorithm 1.22).

Algorithm 1.22 (Euclidean Algorithm for gcd).

Given integers $a \geq b > 0$, compute $g := \gcd(a, b)$.

```
1. Set $r_0 := a$ and $r_1 := b$.
2. Set $i := 1$ and while $r_i \neq 0$, do the following
    (a) Set $r_{i+1} := \text{rem}(r_{i-1}, r_i)$ and $i := i + 1$.
Return $r_{i-1}$.
```

Exercise 1.23. Explain why Algorithm 1.22 terminates and correctly computes the greatest common divisor.

Theorem 1.24 ([vzGG13, Theorem 3.13])**.** *Algorithm 1.22 for positive n-bit and m-bit integers has complexity $O(mn)$.*

Exercise 1.25. Use Exercises 1.20 and 1.21 to prove Theorem 1.24.

Remark 1.26. As you might know, an essential property of the greatest common divisor is that it corresponds to the smallest positive linear combination of a and b. That is, there exist $x, y \in \mathbb{Z}$ such that $\gcd(a, b) = ax + by$. Running the Euclidean Algorithm for gcd (Algorithm 1.22) is almost enough for computing x and y, we just need to "undo" the operations. An optimized version of this algorithm runs in time $O(n \log^2 n)$, where n is again the number of bits of a and b.

Remark 1.27. In particular, Remark 1.26 implies that we can find the inverse of an n-bit integer in $\mathbb{Z}/M\mathbb{Z}$ for $M \geq 2$ in time $O(n \log^2 n)$.

1.2.6 Large powers

We will look at a basic (but useful) algorithm for computing powers g^k, in the general case when g is an element of any group G and k is an integer.

Algorithm 1.28 (Exponentiation algorithm [Coh93, Algorithm 1.2.1]).

The input is an element g of a multiplicative group G and an integer k. This algorithm computes g^k in G.

```
1. Set $y := 1_G$. If $k = 0$, output $y$ and terminate. If $k < 0$, let $K := -k$ and $z := g^{-1}$.
   Otherwise, set $K := k$ and $z := g$.
2. If $K$ is odd set $y := z \cdot y$.
3. Set $K := \lfloor K/2 \rfloor$. If $K = 0$, output $y$ as the answer and terminate. Otherwise, set
   $z = z \cdot z$ and go to Step 2.
```

Exercise 1.29. Prove that Algorithm 1.28 computes g^k using $O(\log |k|)$ group multiplications. In particular, when g is an integer, prove that the algorithm runs in $O(n \log n)$ time, where n is the number of bits of the input g.

1.2.7 Summary

We end this section with a summary of results on integer arithmetic in Table 1.1.

Operation (of n-bit integers)	**Naive Algorithm**	**Optimized Algorithm**
Addition	$O(n)$	$O(n)$
Multiplication	$O(n^2)$	$O(n \log n)$
Division	$O(n^2)$	$O(n \log n)$
GCD	$O(n^2)$	$O(n \log^2 n)$
Exponentiation (to k-bit integer)	$O(n^2)$	$O(n \log n)$

Table 1.1: Summary of complexity for basic arithmetic algorithms.

1.3 More arithmetic

We will use what we learned about integer arithmetic to study the complexity of modular and polynomial arithmetic.

1.3.1 Modular arithmetic

Let $M \geq 2$. Elements of $\mathbb{Z}/M\mathbb{Z}$ can be represented as integers $x \in \{0, \ldots, M-1\}$. In particular, elements of $\mathbb{Z}/M\mathbb{Z}$ occupy at most $\log(M)$ bits of space. We describe the basic arithmetic operations in this ring and record a summary in Table 1.2.

- To add two elements of $\mathbb{Z}/M\mathbb{Z}$, we can add their representatives and subtract M if the result is $\geq M$. This algorithm has complexity $O(\log(M))$.
- To multiply, we multiply the representatives and then take the reminder modulo M, so the complexity of multiplication is $O(\log M \log\log M)$.
- Division is achieved by running the Euclidean Algorithm as in Remark 1.27 to invert the denominator (complexity $O(\log M(\log\log M)^2)$), and then multiplying by the numerator (complexity $O(\log M \log\log M)$). In total, the complexity of division is $O(\log M(\log\log M)^2)$.
- For exponentiation, one can use Algorithm 1.28 in time $O(\log M(\log\log M)\log k)$.

Remark 1.30. One can estimate the time of performing any arithmetic operation over $\mathbb{Z}/M\mathbb{Z}$ by $O(\log^{1+\epsilon} M)$. This is sometimes enough information for complexity computations.

Operation over $\mathbb{Z}/M\mathbb{Z}$	**Running Time**
Addition	$O(\log M)$
Multiplication	$O(\log M \log\log M)$
Division	$O(\log M \log^2(\log M))$
Exponentiation (x^k)	$O(\log M \log\log M \log k)$

Table 1.2: Summary of running times for modular arithmetic.

1.3.2 Polynomial arithmetic

We can use what we have studied about integer arithmetic to determine the complexity of arithmetic in $\mathbb{Z}[x]$ or $\mathbb{Q}[x]$. In general, polynomials in $\mathbb{Z}[x]$ can be represented as polynomials over finite rings $\mathbb{Z}/M\mathbb{Z}$ for large enough M. Polynomials in $\mathbb{Z}/M\mathbb{Z}[x]$ of degree $< n$ can be represented as a sequence of n coefficients. Also, note that each coefficient can be represented using $O(\log M)$ bits, so to encode a polynomial in $\mathbb{Z}/M\mathbb{Z}[x]$ of degree $< n$, we need space $O(n \log M)$. The running times for algorithms for polynomial multiplication are related to the ones for integer multiplication. Let $\mathsf{M}_{\mathsf{int}}(n)$ denote the cost of multiplying n-bit integers. Then, the cost of polynomial arithmetic is summarized in Table 1.3. The interested reader can look at [Har21, §2].

Operation over $\mathbb{Z}/M\mathbb{Z}[x]$ of deg n	**Running Time**
Addition	$O(n \log M)$
Multiplication	$O(\mathsf{M}_{\mathsf{int}}(n \log(nM)))$
Division	$O(\mathsf{M}_{\mathsf{int}}(n \log(nM)))$

Table 1.3: Summary of running times for polynomial arithmetic.

1.4 Linear algebra

Now that we have looked at basic arithmetic, we move on to linear algebra, an area that also allows us to make (fast) explicit computations. The complexity of arithmetic operations on matrices depends on the complexity of arithmetic over the base field or ring. As we saw in §1.2 and §1.3, this complexity varies and that is why we focus on the number of ring operations (multiplications/divisions) needed for each algorithm. For basic arithmetic of matrices, one can easily find (upper bounds) for the complexity of adding and multiplying matrices with $\mathbb{Z}$ coefficients, so we leave this as an exercise. We will skip the basic arithmetic and move on to more interesting matrix manipulations.

Exercise 1.31. Compute a function $f(n)$ such that $O(f(n))$ represents the number of field multiplications needed to compute the product of two $n \times n$ matrices. Why can you ignore the number of addition operations?

Remark 1.32. The best known bound for the number of operations (over a field) needed to multiply two $n \times n$ matrices is $O(n^{2.3728596})$ [Har21, Remark 2.5.1].

1.4.1 Gaussian elimination

This is perhaps one of the most useful and widely used algorithms in linear algebra. It gives us a way of solving linear systems, compute determinants, and find inverses and pseudo-inverses, etc. Because of the applications, we focus on square matrices.

The number of multiplications/divisions needed in Algorithm 1.33 is $O(n^3/3)$. Now we can look at a couple applications of the ideas from this algorithm.

Algorithm 1.33 (Gaussian Elimination for square matrices [Coh93, Algorithm 2.2.1]).
The input is an $n \times n$ matrix M with entries in a field and a vector B of length n. This algorithm returns a vector X such that $MX = B$ if possible and `false` if such vector does not exist.

1. Set $j := 0$.
2. Let $j := j + 1$. If $j > n$, then go to Step 6.
3. If $m_{i,j} = 0$ for all $i \geq j$, then return `false` and terminate. Otherwise, let $i \geq j$ be some index such that $m_{i,j} \neq 0$.
4. If $i > j$, for $l = j, \dots, n$ exchange $m_{i,l}$ and $m_{j,l}$ and then exchange b_i and b_j.
5. Note that $m_{j,j} \neq 0$. Set $d := m_{j,j}^{-1}$ and for all $k > j$ set $c_k := dm_{k,j}$. For all $k > j$ and $l > j$ set $m_{k,l} := m_{k,l} - c_k m_{j,l}$. Finally, for $k > j$ set $b_k := b_k - c_k b_j$ and go to Step 2.
6. Note that M is now upper-triangular! For $i = n, n-1, \dots, 1$ set
$$x_i := \left(b_i - \sum_{i<j\leq n} m_{i,j} x_j \right) / m_{i,i},$$
output $X = (x_i)_{1\leq i\leq n}$, and terminate.

Exercise 1.34. Modify Algorithm 1.33 to compute the inverse of a square matrix. Check how many multiplications/divisions does it take to run your algorithm. Can you get the number of operations to be asymptotic to $4n^3/3$?

Exercise 1.35. Modify Algorithm 1.33 to compute the determinant of a square matrix. Check how many multiplications/divisions does it take to run your algorithm. Can you get the number of operations to be asymptotic to $n^3/3$?

Remark 1.36. Algorithm 1.33 hinges on being able to invert $m_{j,j}$ in Step 5. This is an obstacle for computing determinants of matrices with coefficients in integral domains but not fields (which will be essential in the following lectures). The reader can check [Coh93, Algorithm 2.2.6] for an example of an algorithm to solve this. The algorithm takes $O(n^3)$ operations.

1.4.2 Normal forms and picking a basis

Gaussian elimination allows us to represent a matrix by a similar matrix that is simpler. This is definitely not the only normal form for a matrix. In §5.1.1, we will describe and use the Hermite normal form, which works over the integers $\mathbb{Z}$, and allows us to represent ideals in orders.

Other normal forms that we will not focus on here, but which are very useful, include the Smith normal form (which helps with module and abelian group structure computations) and the Jordan normal form.

Finding normal forms is just finding a new basis for your space, in which the linear operator represented by the matrix can be written in a simpler way. The last special basis that we will explore in Lecture 4 is the LLL-reduced basis, for which there is a highly efficient algorithm to compute.

Bibliography

[BZ11] Richard P. Brent and Paul Zimmermann. *Modern computer arithmetic*, volume 18 of *Cambridge Monographs on Applied and Computational Mathematics*. Cambridge University Press, Cambridge, 2011. ↑1.

[CLO15] David A. Cox, John Little, and Donal O'Shea. *Ideals, varieties, and algorithms*. Undergraduate Texts in Mathematics. Springer, Cham, fourth edition, 2015. An introduction to computational algebraic geometry and commutative algebra. ↑ii.

[Coh93] Henri Cohen. *A course in computational algebraic number theory*, volume 138 of *Graduate Texts in Mathematics*. Springer-Verlag, Berlin, 1993. ↑ii, 1, 7, 10, 12, 16, 19, 22, 25, 27, 30, 32, 33, 35, 36.

[Coh00] Henri Cohen. *Advanced topics in computational number theory*, volume 193 of *Graduate Texts in Mathematics*. Springer-Verlag, New York, 2000. ↑ii.

[Har21] David Harvey. Counting points on hyperelliptic curves over finite fields, 2021. IAS/Park City Mathematics Series. ↑ii, 1, 2, 6, 9.

[HvdH21] David Harvey and Joris van der Hoeven. Integer multiplication in time $O(n \log n)$. *Ann. of Math. (2)*, 193(2):563–617, 2021. ↑4, 6.

[Lan94] Serge Lang. *Algebraic number theory*, volume 110 of *Graduate Texts in Mathematics*. Springer-Verlag, New York, second edition, 1994. ↑ii, 12.

[LLL82] A. K. Lenstra, H. W. Lenstra, Jr., and L. Lovász. Factoring polynomials with rational coefficients. *Math. Ann.*, 261(4):515–534, 1982. ↑29.

[Mil] J. S. Milne. Algebraic number theory. `https://www.jmilne.org/math/CourseNotes/ant.html`. ↑ii, 12.

[Pap94] Christos H. Papadimitriou. *Computational complexity*. Addison-Wesley Publishing Company, Reading, MA, 1994. ↑2.

[Sta67] H. M. Stark. A complete determination of the complex quadratic fields of class-number one. *Michigan Math. J.*, 14:1–27, 1967. ↑35.

[Ste] William Stein. Algebraic number theory, a computational approach. `https://wstein.org/books/ant/`. ↑ii, 12, 27.

[Voi21] John Voight. *Quaternion algebras*, volume 288 of *Graduate Texts in Mathematics*. Springer, Cham, [2021] ©2021. ↑ii.

[vzGG13] Joachim von zur Gathen and Jürgen Gerhard. *Modern computer algebra*. Cambridge University Press, Cambridge, third edition, 2013. ↑ii, 1, 7.