Lecture 0
INTRODUCTION

This lecture is an orientation on the central problems that concern us. Specifically, we identify three families of "Fundamental Problems" in algorithmic algebra (§1 – §3). In the rest of the lecture (§4–§9), we briefly discuss the complexity-theoretic background. §10 collects some common mathematical terminology while §11 introduces computer algebra systems. The reader may prefer to skip §4-11 on a first reading, and only use them as a reference.

> All our rings will contain *unity* which is denoted 1 (and distinct from 0). They are commutative except in the case of matrix rings.

The main algebraic structures of interest are:

$\mathbb{N}$	=	natural numbers $0, 1, 2, \ldots$
$\mathbb{Z}$	=	integers
$\mathbb{Q}$	=	rational numbers
$\mathbb{R}$	=	reals
$\mathbb{C}$	=	complex numbers
$R[\mathbf{X}]$	=	polynomial ring in $d \geq 1$ variables $\mathbf{X} = (X_1, \ldots, X_n)$ with coefficients from a ring R.

Let R be any ring. For a univariate polynomial $P \in R[X]$, we let $\deg(P)$ and $\texttt{lead}(P)$ denote its *degree* and *leading coefficient* (or leading coefficient). If $P = 0$ then by definition, $\deg(P) = -\infty$ and $\texttt{lead}(P) = 0$; otherwise $\deg(P) \geq 0$ and $\texttt{lead}(P) \neq 0$. We say P is a (respectively) integer, rational, real or complex polynomial, depending on whether R is $\mathbb{Z}, \mathbb{Q}, \mathbb{R}$ or $\mathbb{C}$.

In the course of this book, we will encounter other rings: (e.g., §I.1). With the exception of matrix rings, all our rings are commutative. The basic algebra we assume can be obtained from classics such as van der Waerden [22] or Zariski-Samuel [27, 28].

§1. Fundamental Problem of Algebra

Consider an integer polynomial

$$P(X) = \sum_{i=0}^{n} a_i X^i \qquad (a_i \in \mathbb{Z}, a_n \neq 0). \tag{1}$$

Many of the oldest problems in mathematics stem from attempts to solve the equation

$$P(X) = 0, \tag{2}$$

i.e., to find numbers α such that $P(\alpha) = 0$. We call such an α a *solution* of equation (2); alternatively, α is a *root* or *zero* of the polynomial $P(X)$. By definition, an *algebraic number* is a zero of some polynomial $P \in \mathbb{Z}[X]$. The *Fundamental Theorem of Algebra* states that every non-constant polynomial $P(X) \in \mathbb{C}[X]$ has a root $\alpha \in \mathbb{C}$. Put another way, $\mathbb{C}$ is algebraically closed. d'Alembert first formulated this theorem in 1746 but Gauss gave the first complete proof in his 1799 doctoral thesis

at Helmstedt. It follows that there are n (not necessarily distinct) complex numbers $\alpha_1, \ldots, \alpha_n \in \mathbb{C}$ such that the polynomial in (1) is equal to

$$P(X) \equiv a_n \prod_{i=1}^{n} (X - \alpha_i). \tag{3}$$

To see this, suppose α_1 is a root of $P(X)$ as guaranteed by the Fundamental Theorem. Using the *synthetic division algorithm* to divide $P(X)$ by $X - \alpha_1$, we get

$$P(X) = Q_1(X) \cdot (X - \alpha_1) + \beta_1$$

where $Q_1(X)$ is a polynomial of degree $n-1$ with coefficients in $\mathbb{C}$ and $\beta_1 \in \mathbb{C}$. On substituting $X = \alpha_1$, the left-hand side vanishes and the right-hand side becomes β_1. Hence $\beta_1 = 0$. If $n = 1$, then $Q_1(X) = a_n$ and we are done. Otherwise, this argument can be repeated on $Q_1(X)$ to yield equation (3).

The computational version of the Fundamental Theorem of Algebra is the problem of finding roots of a univariate polynomial. We may dub this the *Fundamental Problem of Computational Algebra* (or *Fundamental Computational Problem of Algebra*). The Fundamental Theorem is about complex numbers. For our purposes, we slightly extend the context as follows. If $R_0 \subseteq R_1$ are rings, the *Fundamental Problem* for the pair (R_0, R_1) is this:

Given $P(X) \in R_0[X]$, solve the equation $P(X) = 0$ in R_1.

We are mainly interested in cases where $\mathbb{Z} \subseteq R_0 \subseteq R_1 \subseteq \mathbb{C}$. The three main versions are where (R_0, R_1) equals $(\mathbb{Z}, \mathbb{Z}), (\mathbb{Z}, \mathbb{R})$ and $(\mathbb{Z}, \mathbb{C})$, respectively. We call them the *Diophantine*, *real* and *complex versions* (respectively) of the Fundamental Problem.

What does it mean "to solve $P(X) = 0$ in R_1"? The most natural interpretation is that we want to enumerate all the roots of P that lie in R_1. Besides this *enumeration interpretation*, we consider two other possibilities: the *existential interpretation* simply wants to know if P has a root in R_1, and the *counting interpretation* wants to know the number of such roots. To enumerate[1] roots, we must address the representation of these roots. For instance, we will study a representation via "isolating intervals".

Recall another classical version of the Fundamental Problem. Let $R_0 = \mathbb{Z}$ and R_1 denote the complex subring comprising all those elements that can be obtained by applying a finite number of field operations (ring operations plus division by non-zero) and taking nth roots ($n \geq 2$), starting from $\mathbb{Z}$. This is the famous *solution by radicals version* of the Fundamental Problem. It is well known that when $\deg P = 2$, there is always a solution in R_1. What if $\deg P > 2$? This was a major question of the 16th century, challenging the best mathematicians of its day. We now know that solution by radicals exists for $\deg P = 3$ (Tartaglia, 1499-1557) and $\deg P = 4$ (variously ascribed to Ferrari (1522-1565) or Bombelli (1579)). These methods were widely discussed, especially after they were published by Cardan (1501-1576) in his classic Ars magna, "The Great Art", (1545). This was *the* algebra book until Descartes' (1637) and Euler's *Algebra* (1770). Abel (1824) (also Wantzel) show that there is no solution by radicals for a general polynomial of degree 5. Ruffini had a prior though incomplete proof. This kills the hope for a single formula which solves *all* quintic polynomials. This still leaves open the possibility that for *each* quintic polynomial, there is a formula to extract its roots. But it is not hard to dismiss this possibility: for example, an explicit quintic polynomial that

[1] There is possible confusion here: the word "enumerate" means to "count" as well as to "list by name". Since we are interested in both meanings here, we have to appropriate the word "enumerate" for only one of these two senses. In this book, we try to use it only in the latter sense.

does not admit solution by radicals is $P(X) = X^5 - 16X + 2$ (see [3, p.574]). Miller and Landau [12] (also [26]) revisits these question from a complexity viewpoint. The above historical comments may be pursued more fully in, for example, Struik's volume [21].

Remarks:. The Fundamental Problem of algebra used to come under the rubric "theory of equations", which nowadays is absorbed into other areas of mathematics. In these lectures, we are interested in general and effective methods, and we are mainly interested in real solutions.

§2. Fundamental Problem of Classical Algebraic Geometry

To generalize the Fundamental Problem of algebra, we continue to fix two rings, $\mathbb{Z} \subseteq R_0 \subseteq R_1 \subseteq \mathbb{C}$. First consider a bivariate polynomial

$$P(X, Y) \in R_0[X, Y]. \tag{4}$$

Let $\text{Zero}(P)$ denote the set of R_1-solutions of the equation $P = 0$, *i.e.*, $(\alpha, \beta) \in R_1^2$ such that $P(\alpha, \beta) = 0$. The *zero set* $\text{Zero}(P)$ of P is generally an infinite set. In case $R_1 = \mathbb{R}$, the set $\text{Zero}(P)$ is a planar curve that can be plotted and visualized. Just as solutions to equation (2) are called algebraic numbers, the zero sets of bivariate integer polynomials are called *algebraic curves*. But there is no reason to stop at two variables. For $d \geq 3$ variables, the zero set of an integer polynomial in d variables is called an *algebraic hypersurface*: we reserve the term *surface* for the special case $d = 3$.

Given two surfaces defined by the equations $P(X, Y, Z) = 0$ and $Q(X, Y, Z) = 0$, their intersection is generally a curvilinear set of triples $(\alpha, \beta, \gamma) \in R_1^3$, consisting of all simultaneous solutions to the pair of simultaneous equations $P = 0,\ Q = 0$. We may extend our previous notation and write $\text{Zero}(P, Q)$ for this intersection. More generally, we want the simultaneous solutions to a *system of* $m \geq 1$ *polynomial equations* in $d \geq 1$ variables:

$$\left.\begin{array}{c} P_1 = 0 \\ P_2 = 0 \\ \vdots \\ P_m = 0 \end{array}\right\} \qquad (\text{where } \ P_i \in R_0[X_1, \ldots, X_d]) \tag{5}$$

A point $(\alpha_1, \ldots, \alpha_d) \in R_1^d$ is called a *solution* of the system of equations (5) or a *zero* of the set $\{P_1, \ldots, P_m\}$ provided $P_i(\alpha_1, \ldots, \alpha_d) = 0$ for $i = 1, \ldots, m$. In general, for any subset $J \subseteq R_0[\mathbf{X}]$, let $\text{Zero}(J) \subseteq R_1^d$ denote the *zero set* of J. To denote the dependence on R_1, we may also write $\text{Zero}_{R_1}(J)$. If R_1 is a field, we also call a zero set an *algebraic set*. Since the primary objects of study in classical algebraic geometry are algebraic sets, we may call the problem of solving the system (5) the *Fundamental (Computational) Problem of classical algebraic geometry*. If each P_i is linear in (5), we are looking at a system of linear equations. One might call this is the *Fundamental (Computational) Problem of linear algebra*. Of course, linear systems are well understood, and their solution technique will form the basis for solving nonlinear systems.

Again, we have three natural meanings to the expression "solving the system of equations (5) in R_1": (i) The existential interpretation asks if $\text{Zero}(P_1, \ldots, P_m)$ is empty. (ii) The counting interpretation asks for the cardinality of the zero set. In case the cardinality is "infinity", we could refine the question by asking for the *dimension* of the zero set. (iii) Finally, the enumeration interpretation poses no problems when there are only finitely many solutions. This is because the coordinates of these solutions turn out to be algebraic numbers and so they could be explicitly enumerated. It becomes problematic when the zero set is infinite. Luckily, when $R_1 = \mathbb{R}$ or $\mathbb{C}$, such zero sets are well-behaved topologically, and each zero set consists of a finite number of connected components.

(For that matter, the counting interpretation can be re-interpreted to mean counting the number of components of each dimension.) A typical interpretation of "enumeration" is "give at least one sample point from each connected component". For real planar curves, this interpretation is useful for plotting the curve since the usual method is to "trace" each component by starting from any point in the component.

Note that we have moved from algebra (numbers) to geometry (curves and surfaces). In recognition of this, we adopt the geometric language of "points and space". The set R_1^d (d-fold Cartesian product of R_1) is called the *d-dimensional affine space* of R_1, denoted $\mathbb{A}^d(R_1)$. Elements of $\mathbb{A}^d(R_1)$ are called *d-points* or simply *points*. Our zero sets are subsets of this affine space $\mathbb{A}^d(R_1)$. In fact, $\mathbb{A}^d(R_1)$ can be given a topology (the Zariski topology) in which zero sets are the closed sets.

There are classical techniques via elimination theory for solving these Fundamental Problems. The recent years has seen a revival of these techniques as well as major advances. In one line of work, Wu Wen-tsun exploited Ritt's idea of characteristic sets to give new methods for solving (5) rather efficiently in the complex case, $R_1 = \mathbb{C}$. These methods turn out to be useful for proving theorems in elementary geometry as well [25]. But many applications are confined to the real case ($R_1 = \mathbb{R}$). Unfortunately, it is a general phenomenon that real algebraic sets do not behave as regularly as the corresponding complex ones. This is already evident in the univariate case: the Fundamental Theorem of Algebra fails for real solutions. In view of this, most mathematical literature treats the complex case. More generally, they apply to any algebraically closed field. There is now a growing body of results for real algebraic sets.

Another step traditionally taken to "regularize" algebraic sets is to consider projective sets, which abolish the distinction between finite and infinite points. A *projective d-dimensional point* is simply an equivalence class of the set $\mathbb{A}^{d+1}(R_1)\backslash\{(0,\ldots,0)\}$, where two non-zero $(d+1)$-points are *equivalent* if one is a constant multiple of the other. We use $\mathbb{P}^d(R_1)$ to denote the *d-dimensional projective space* of R_1.

Semialgebraic sets. The real case admits a generalization of the system (5). We can view (5) as a *conjunction* of basic predicates of the form "$P_i = 0$":

$$(P_1 = 0) \ \wedge \ (P_2 = 0) \wedge \cdots \wedge (P_m = 0).$$

We generalize this to an arbitrary Boolean combination of basic predicates, where a basic predicate now has the form $(P = 0)$ or $(P > 0)$ or $(P \geq 0)$. For instance,

$$((P = 0) \ \wedge \ (Q > 0)) \ \vee \ \neg(R \geq 0)$$

is a Boolean combination of three basic predicates where P, Q, R are polynomials. The set of real solutions to such a predicate is called a *semi-algebraic set* (or a *Tarski set*). We have effective methods of computing semi-algebraic sets, thanks to the pioneering work of Tarski and Collins [7]. Recent work by various researchers have reduced the complexity of these algorithms from double exponential time to single exponential space [15]. This survey also describes to applications of semi-algebraic in algorithmic robotics, solid modeling and geometric theorem proving. Recent books on real algebraic sets include [4, 2, 10].

§3. Fundamental Problem of Ideal Theory

Algebraic sets are basically geometric objects: witness the language of "space, points, curves, surfaces". Now we switch from the geometric viewpoint (back!) to an algebraic one. One of the beauties of this subject is this interplay between geometry and algebra.

Fix $\mathbb{Z} \subseteq R_0 \subseteq R_1 \subseteq \mathbb{C}$ as before. A polynomial $P(\mathbf{X}) \in R_0[\mathbf{X}]$ is said to *vanish* on a subset $U \subseteq \mathbb{A}^d(R_1)$ if for all $\mathbf{a} \in U$, $P(\mathbf{a}) = 0$. Define

$$\textsc{Ideal}(U) \subseteq R_0[\mathbf{X}]$$

to comprise all polynomials $P \in R_0[\mathbf{X}]$ that vanish on U. The set $\textsc{Ideal}(U)$ is an ideal. Recall that a non-empty subset $J \subseteq R$ of a ring R is an *ideal* if it satisfies the properties

1. $a, b \in J \Rightarrow a - b \in J$
2. $c \in R$, $a \in J \Rightarrow ca \in J$.

For any $a_1, \ldots, a_m \in R$ and $R' \supseteq R$, the set $(a_1, \ldots, a_m)_{R'}$ defined by

$$(a_1, \ldots, a_m)_{R'} := \{\sum_{i=1}^{m} a_i b_i : b_1, \ldots, b_m \in R'\}$$

is an ideal, the ideal *generated by* $a_1, \ldots, a_m$ in R'. We usually omit the subscript R' if this is understood.

The Fundamental Problem of classical algebraic geometry (see Equation (5)) can be viewed as computing (some characteristic property of) the zero set defined by the input polynomials $P_1, \ldots, P_m$. But note that

$$\textsc{Zero}(P_1, \ldots, P_m) = \textsc{Zero}(I)$$

where I is the ideal generated by $P_1, \ldots, P_m$. Hence we might as well assume that the input to the Fundamental Problem is the ideal I (represented by a set of generators). *This suggests that we view ideals to be the algebraic analogue of zero sets.* We may then ask for the algebraic analogue of the Fundamental Problem of classical algebraic geometry. A naive answer is that, "given $P_1, \ldots, P_m$, to enumerate the set $(P_1, \ldots, P_m)$". Of course, this is impossible. But we effectively "know" a set S if, for any purported member x, we can decisively say whether or not x is a member of S. Thus we reformulate the enumerative problem as the *Ideal Membership Problem*:

$$\textit{Given } P_0, P_1, \ldots, P_m \in R_0[\mathbf{X}], \textit{ is } P_0 \textit{ in } (P_1, \ldots, P_m)?$$

Where does R_1 come in? Well, the ideal $(P_1, \ldots, P_m)$ is assumed to be generated in $R_1[\mathbf{X}]$. We shall introduce effective methods to solve this problem. The technique of Gröbner bases (as popularized by Buchberger) is notable. There is strong historical basis for our claim that the ideal membership problem is fundamental: van der Waerden [22, vol. 2, p. 159] calls it the "main problem of ideal theory in polynomial rings". Macaulay in the introduction to his 1916 monograph [14] states that the "object of the algebraic theory [of ideals] is to discover those general properties of [an ideal] which will afford a means of answering the question whether a given polynomial is a member of a given [ideal] or not".

How general are the ideals of the form $(P_1, \ldots, P_m)$? The only ideals that might not be of this form are those that cannot be generated by a finite number of polynomials. The answer is provided by what is perhaps the starting point of modern algebraic geometry: the *Hilbert!Basis Theore*. A ring R is called *Noetherian* if all its ideals are finitely generated. For example, if R is a field, then it is Noetherian since its only ideals are (0) and (1). The Hilbert Basis Theorem says that $R[\mathbf{X}]$ is Noetherian if R is Noetherian. This theorem is crucial[2] from a constructive viewpoint: it assures us that although ideals are potentially infinite sets, they are finitely describable.

[2]The paradox is, many view the original proof of this theorem as initiating the modern tendencies toward non-constructive proof methods.

We now have a mapping

$$U \mapsto \textsc{Ideal}(U) \tag{6}$$

from subsets of $\mathbb{A}^d(R_1)$ to the ideals of $R_0[\mathbf{X}]$, and conversely a mapping

$$J \mapsto \textsc{Zero}(J) \tag{7}$$

from subsets of $R_0[\mathbf{X}]$ to algebraic sets of $\mathbb{A}^d(R_1)$. It is not hard to see that

$$J \subseteq \textsc{Ideal}(\textsc{Zero}(J)), \qquad U \subseteq \textsc{Zero}(\textsc{Ideal}(U)) \tag{8}$$

for all subsets $J \subseteq R_0[\mathbf{X}]$ and $U \subseteq \mathbb{A}^d(R_1)$. Two other basic identities are:

$$\begin{aligned} \textsc{Zero}(\textsc{Ideal}(\textsc{Zero}(J))) &= \textsc{Zero}(J), \qquad J \subseteq R_0[\mathbf{X}], \\ \textsc{Ideal}(\textsc{Zero}(\textsc{Ideal}(U))) &= \textsc{Ideal}(U), \qquad U \subseteq \mathbb{A}^d(R_1), \end{aligned} \tag{9}$$

We prove the first equality: If $\mathbf{a} \in \textsc{Zero}(J)$ then for all $P \in \textsc{Ideal}(\textsc{Zero}(J))$, $P(\mathbf{a}) = 0$. Hence $\mathbf{a} \in \textsc{Zero}(\textsc{Ideal}(\textsc{Zero}(J))$. Conversely, if $\mathbf{a} \in \textsc{Zero}(\textsc{Ideal}(\textsc{Zero}(J))$ then $P(\mathbf{a}) = 0$ for all $P \in \textsc{Ideal}(\textsc{Zero}(J))$. But since $J \subseteq \textsc{Ideal}(\textsc{Zero}(J))$, this means that $P(\mathbf{a}) = 0$ for all $P \in J$. Hence $\mathbf{a} \in \textsc{Zero}(J)$. The second equality (9) is left as an exercise.

If we restrict the domain of the map in (6) to algebraic sets and the domain of the map in (7) to ideals, would these two maps be inverses of each other? The answer is no, based on a simple observation: An ideal I is called *radical* if for all integers $n \geq 1$, $P^n \in I$ implies $P \in I$. It is not hard to check that $\textsc{Ideal}(U)$ is radical. On the other hand, the ideal $(X^2) \in \mathbb{Z}[X]$ is clearly non-radical.

It turns out that if we restrict the ideals to radical ideals, then $\textsc{Ideal}(\cdot)$ and $\textsc{Zero}(\cdot)$ would be inverses of each other. This is captured in the *Hilbert Nullstellensatz* (or, Hilbert's Zero Theorem in English). After the Basis Theorem, this is perhaps the next fundamental theorem of algebraic geometry. It states that if P vanishes on the zero set of an ideal I then some power P^n of P belongs to I. As a consequence,

$$I = \textsc{Ideal}(\textsc{Zero}(I)) \Leftrightarrow I \text{ is radical.}$$

In proof: Clearly the left-hand side implies I is radical. Conversely, if I is radical, it suffices to show that $\textsc{Ideal}(\textsc{Zero}(I)) \subseteq I$. Say $P \in \textsc{Ideal}(\textsc{Zero}(I))$. Then the Nullstellensatz implies $P^n \in I$ for some n. Hence $P \in I$ since I is radical, completing our proof.

We now have a bijective correspondence between algebraic sets and radical ideals. This implies that ideals in general carry more information than algebraic sets. For instance, the ideals (X) and (X^2) have the same zero set, *viz.*, $X = 0$. But the unique zero of (X^2) has multiplicity 2.

The ideal-theoretic approach (often attached to the name of E. Noether) characterizes the transition from classical to "modern" algebraic geometry. "Post-modern" algebraic geometry has gone on to more abstract objects such as schemes. Not much constructive questions are raised at this level, perhaps because the abstract questions are hard enough. The reader interested in the profound transformation that algebraic geometry has undergone over the centuries may consult Dieudonné [9] who described the subject in "seven epochs". The current challenge for constructive algebraic geometry appears to be at the levels of classical algebraic geometry and at the ideal-theoretic level. For instance, Brownawell [6]and others have recently given us effective versions of classical results such as the Hilbert Nullstellensatz. Such results yields complexity bounds that are necessary for efficient algorithms (see Exercise).

This concludes our orientation to the central problems that motivates this book. This exercise is pedagogically useful for simplifying the algebraic-geometric landscape for students. However, the richness of this subject and its complex historical development ensures that, in the opinion of some

experts, we have made gross oversimplifications. Perhaps an account similar to what we presented is too much to hope for – we have to leave this to the professional historians to tell us the full story. In any case, having *selected* our core material, the rest of the book will attempt to treat and view it through the lens of computational complexity theory. The remaining sections of this lecture addresses this.

Exercises

Exercise 3.1: Show relation (8), and relation (9). □

Exercise 3.2: Show that the ideal membership problem is polynomial-time equivalent to the problem of checking if two sets of elements generate the same ideal: Is $(a_1, \ldots, a_m) = (b_1, \ldots, b_n)$? [Two problems are polynomial-time equivalent if one can be reduced to the other in polynomial-time and vice-versa.] □

Exercise 3.3*: a) Given $P_0, P_1, \ldots, P_m \in \mathbb{Q}[X_1, \ldots, X_d]$, where these polynomials have degree at most n, there is a known double exponential bound $B(d, n)$ such that if $P_0 \in (P_1, \ldots, P_m)$ there there exists polynomials $Q_1, \ldots, Q_m$ of degree at most $B(d, n)$ such that

$$P_0 = P_1 Q_1 + \cdots + P_m Q_m.$$

Note that $B(d, n)$ does not depend on m. Use this fact to construct a double exponential time algorithm for ideal membership.
b) Does the bound $B(d, n)$ translate into a corresponding bound for $\mathbb{Z}[X_1, \ldots, X_d]$? □

§4. Representation and Size

We switch from mathematics to computer science. To investigate the computational complexity of the Fundamental Problems, we need tools from complexity theory. The complexity of a problem is a function of some size measure on its input instances. The size of a problem instance depends on its representation.

Here we describe the representation of some basic objects that we compute with. For each class of objects, we choose a notion of "size".

Integers: Each integer $n \in \mathbb{Z}$ is given the binary notation and has *(bit-)size*

$$\texttt{size}(n) = 1 + \lceil \log(|n| + 1) \rceil$$

where logarithms are always base 2 unless otherwise stated. The term "$1 + \ldots$" takes care of the sign-bit.

Rationals: Each rational number $p/q \in \mathbb{Q}$ is represented as a pair of integers with $q > 0$. We do not assume the reduced form of a rational number. The *(bit-)size* is given by

$$\texttt{size}\left(\frac{p}{q}\right) = \texttt{size}(p) + \texttt{size}(q) + \log(\texttt{size}(p))$$

where the "$+ \log(\texttt{size}(p))$" term indicates the separation between the two integers.

Matrices: The default is the *dense representation* of matrices so that zero entries must be explicitly represented. An $m \times n$ matrix $M = (a_{ij})$ has *(bit-)size*

$$\texttt{size}(M) = \sum_{i=1}^{m}\sum_{j=1}^{n}(\texttt{size}(a_{ij}) + \log(\texttt{size}(a_{ij})))$$

where the "$+\log(size(a_{ij}))$" term allows each entry of M to indicate its own bits (this is sometimes called the "self-limiting" encoding). Alternatively, a simpler but less efficient encoding is to essentially double the number of bits

$$\texttt{size}(M) = \sum_{i=1}^{m}\sum_{j=1}^{n}(2 + 2\texttt{size}(a_{ij}))\,.$$

This encoding replaces each 0 by "00" and each 1 by "11", and introduces a separator sequence "01" between consecutive entries.

Polynomials: The default is the *dense representation* of polynomials. So a degree-n univariate polynomial is represented as a $(n+1)$-tuple of its coefficients – and the size of the $(n+1)$-tuple is already covered by the above size consideration for matrices. *(bit-)size*

Other representations (especially of multivariate polynomials) can be more involved. In contrast to dense representations, *sparse representations* refer to *sparse representation* those whose sizes grow linearly with the number of non-zero terms of a polynomial. In general, such compact representations greatly increase (not decrease!) the computational complexity of problems. For instance, Plaisted [16, 17] has shown that deciding if two sparse univariate integer polynomials are relatively prime is NP-hard. In contrast, this problem is polynomial-time solvable in in the dense representation (Lecture II).

Ideals: Usually, 'ideals' refer to polynomial ideals. An ideal I is represented by any finite set $\{P_1, \ldots, P_n\}$ of elements that generate it: $I = (P_1, \ldots, P_n)$. The size of this representation just the sum of the sizes of the generators. Clearly, the representation of an ideal is far from unique.

The representations and sizes of other algebraic objects (such as algebraic numbers) will be discussed as they arise.

§5. Computational Models

We briefly review four models of computation: *Turing machines*, *Boolean circuits*, *algebraic programs* and *random access machines*. With each model, we will note some natural complexity measures (time, space, size, etc), including their correspondences across models. We will be quite informal since many of our assertions about these models will be (with some coaching) self-evident. A reference for machine models is Aho, Hopcroft and Ullman [1]. For a more comprehensive treatment of the algebraic model, see Borodin and Munro [5]; for the Boolean model, see Wegener [24].

I. Turing machine model. The Turing (machine) model is embodied in the *multitape Turing machine*, in which inputs are represented by a binary string. Our representation of objects and definition of sizes in the last section are especially appropriate for this model of computation. The machine is essentially a finite state automaton (called its *finite state control*) equipped with a finite set of doubly-infinite tapes, including a distinguished *input tape* Each tape is divided into cells indexed by the integers. Each cell contains a symbol from a finite alphabet. Each tape has a head

which scans some cell at any moment. A Turing machine may operate in a variety of *computational modes* such as *deterministic*, *nondeterministic* or *randomized*; and in addition, the machine can be generalized from sequential to parallel modes in many ways. We mostly assume the deterministic-sequential mode in this book. In this case, a Turing machine operates according to the specification of its finite state control: in each step, depending on the current state and the symbols being scanned under each tape head, the transition table specifies the next state, modifies the symbols under each head and moves each head to a neighboring cell. The main complexity measures in the Turing model are *time* (the number of steps in a computation), *space* (the number of cells used during a computation) and *reversal* (the number of times a tape head reverses its direction).

II. Boolean circuit model. This model is based on *Boolean circuits*. A Boolean circuit is a directed acyclic finite graph whose nodes are classified as either *input nodes* or *gates*. The input nodes have in-degree 0 and are labeled by an input variable; gates are labeled by Boolean functions with in-degree equal to the arity of the label. The set of Boolean functions which can be used as gate labels is called the *basis!of computational models* of the model. In this book, we may take the basis to be the set of Boolean functions of at most two inputs. We also assume no á priori bound on the out-degree of a gate. The three main complexity measures here are *circuit size* (the number of gates), *circuit depth* (the longest path) and *circuit width* (roughly, the largest antichain).

A circuit can only compute a function on a fixed number of Boolean inputs. Hence to compare the Boolean circuit model to the Turing machine model, we need to consider a *circuit family*, which is an infinite sequence $(C_0, C_1, C_2, \ldots)$ of circuits, one for each input size. Because there is no *a priori* connection between the circuits in a circuit family, we call such a family *non-uniform*. *non-uniform*. For this reason, we call Boolean circuits a "non-uniform model" as opposed to Turing machines which is "uniform". Circuit size can be identified with time on the Turing machine. Circuit depth is more subtle, but it can (following Jia-wei Hong be identified with "reversals" on Turing machines.

It turns out that the Boolean complexity of *any* problem is at most $2^n/n$ (see [24]). Clearly this is a severe restriction on the generality of the model. But it is possible to make Boolean circuit families "uniform" in several ways and the actual choice is usually not critical. For instance, we may require that there is a Turing machine using logarithmic space that, on input n in binary, constructs the (encoded) nth circuit of the circuit family. The resulting *uniform Boolean complexity* is now polynomially related to Turing complexity. Still, the non-uniform model suffices for many applications (see §8), and that is what we will use in this book.

Encodings and bit models. The previous two models are called *bit models* because mathematical objects must first be encoded as binary strings before they can be used on these two models. The issue of encoding may be quite significant. But we may get around this by assuming standard conventions such as binary encoding of numbers, list representation of sets, etc. In algorithmic algebra, it is sometimes useful to avoid encodings by incorporating the relevant algebraic structures directly into the computational model. This leads us to our next model.

III. Algebraic program models. In *algebraic programs*, we must fix some algebraic structures (such as $\mathbb{Z}$, polynomials or matrices over a ring R) and specify a set of primitive algebraic operations called the *basis!of computational models* of the model. Usually the basis includes the ring operations $(+, -, \times)$, possibly supplemented by other operations appropriate to the underlying algebraic structure. A common supplement is some form of root finding (e.g., multiplicative inverse, radical extraction or general root extraction), and GCD. The algebraic program model is thus a class of models based on different algebraic structures and different bases.

An *algebraic program* is defined to be a rooted ordered tree T where each node represents either an *assignment step* of the form

$$V \leftarrow F(V_1, \ldots, V_k),$$

or a *branch step* of the form

$$F(V_1, \ldots, V_k) : 0.$$

Here, F is a k-ary operation in the basis and each V_i is either an input variable, a constant or a variable that has been assigned a value further up the tree. The out-degree of an assignment node is 1; the out-degree of a branch node is 2, corresponding to the outcomes $F(V_1, \ldots, V_k) = 0$ and $F(V_1, \ldots, V_k) \neq 0$, respectively. If the underlying algebraic structure is real, the branch steps can be extended to a 3-way branch, corresponding to $F(V_1, \ldots, V_k) < 0, = 0$ or > 0. At the leaves of T, we fix some convention for specifying the output.

The *input size* is just the number of input variables. The main complexity measure studied with this model is *time*, the length of the longest path in T. Note that we charge a unit cost to each basic operation. This could easily be generalized. For instance, a multiplication step in which one of the operands is a constant (*i.e.*, does not depend on the input parameters) may be charged nothing. This originated with Ostrowski who wrote one of the first papers in algebraic complexity.

Like Boolean circuits, this model is non-uniform because each algebraic program solves problems of a fixed size. Again, we introduce the *algebraic program family* which is an infinite set of algebraic programs, one for each input size.

When an algebraic program has no branch steps, it is called a *straight-line program*. To see that in general we need branching, consider algebraic programs to compute the GCD (see Exercise below).

IV. RAM model. Finally, consider the *random access machine* model of computation. Each RAM is defined by a finite set of instructions, rather as in assembly languages. These instructions make reference to operands called *registers* Each register can hold an arbitrarily large integer and is *indexed* by a natural number. If n is a natural number, we can denote its contents by $\langle n \rangle$. Thus $\langle\langle n \rangle\rangle$ refers to the contents of the register whose index is $\langle n \rangle$. In addition to the usual registers, there is an unindexed register called the *accumulator* in which all computations are done (so to speak). The RAM instruction sets can be defined variously and have the simple format

INSTRUCTION OPERAND

where OPERAND is either n or $\langle n \rangle$ and n is the index of a register. We call the operand *direct* or *indirect* depending on whether we have n or $\langle n \rangle$. We have five RAM instructions: a STORE and LOAD instruction (to put the contents of the accumulator to register n and vice-versa), a TEST instruction (to skip the next instruction if $\langle n \rangle$ is zero) and a SUCC operation (to add one to the content of the accumulator). For example, 'LOAD 5' instructs the RAM to put $\langle 5 \rangle$ into the accumulator; but 'LOAD $\langle 5 \rangle$' puts $\langle\langle 5 \rangle\rangle$ into the accumulator; 'TEST 3' causes the next instruction to be skipped if $\langle 3 \rangle = 0$; 'SUCC' will increment the accumulator content by one. There are two main models of time-complexity for RAM models: in the *unit cost model*, each executed instruction is charged 1 unit of time. In contrast, the *logarithmic cost model*, charges $\lceil \lg(|n| + |\langle n \rangle|) \rceil$ whenever a register n is accessed. Note that an instruction accesses one or two registers, depending on whether the operand is direct or indirect. It is known that the logarithmic cost RAM is within a quadratic factor of the Turing time complexity. The above RAM model is called the *successor RAM* to distinguish it from other variants, which we now briefly note. More powerful arithmetic operations (ADDITION, SUBTRACTION and even MULTIPLICATION) are sometimes included in the instruction set. Schönhage describes an even simpler RAM model than the above model,

essentially by making the operand of each of the above instructions implicit. He shows that this simple model is real-time equivalent to the above one.

Exercises

Exercise 5.1:

(a) Describe an algebraic program for computing the GCD of two integers. (Hint: implement the Euclidean algorithm. Note that the input size is 2 and this computation tree must be infinite although it halts for all inputs.)

(b) Show that the integer GCD cannot be computed by a straight-line program.

(c) Describe an algebraic program for computing the GCD of two rational polynomials $P(X) = \sum_{i=0}^{n} a_i X^i$ and $Q(X) = \sum_{i=0}^{m} b_i X^i$. The input variables are $a_0, a_1, \ldots, a_n, b_0, \ldots, b_m$, so the input size is $n + m + 2$. The output is the set of coefficients of $\texttt{GCD}(P, Q)$. □

§6. Asymptotic Notations

Once a computational model is chosen, there are additional decisions to make before we get a "complexity model". This book emphasizes mainly the *worst case time measure* in each of our computational models. To each machine or program A in our computational model, this associates a function $T_A(n)$ that specifies the worst case number of time steps used by A, over all inputs of size n. Call $T_A(n)$ the *complexity of* A. Abstractly, we may define a *complexity model* to comprise a computational model together with an associated complexity function $T_A(n)$ for each A. The complexity models in this book are: *Turing complexity model*, *Boolean complexity model*, *algebraic complexity model*, and *RAM complexity model*. For instance, the Turing complexity model refers to the worst-case time complexity of Turing machines. "Algebraic complexity model" is a generic term that, in any specific instance, must be instantiated by some choice of algebraic structure and basis operations.

We intend to distinguish complexity functions up to constant multiplicative factors and up to their eventual behavior. To facilitate this, we introduce some important concepts.

Definition 1 *A* complexity function *is a real partial function* $f : \mathbb{R} \to \mathbb{R} \cup \{\infty\}$ *such that* $f(x)$ *is defined for all sufficiently large natural numbers* $x \in \mathbb{N}$. *Moreover, for sufficiently large* x, $f(x) \geq 0$ *whenever* x *is defined.*

If $f(x)$ is undefined, we write $f(x) \uparrow$, and this is to be distinguished from the case $f(x) = \infty$. *Note that we require that* $f(x)$ *be eventually non-negative.* We often use familiar partial functions such as $\log x$ and 2^x as complexity functions, even though we are mainly interested in their values at $\mathbb{N}$. Note that if f, g are complexity functions then so are

$$f + g, \quad fg, \quad f^g, \quad f \circ g$$

where in the last case, we need to assume that $(f \circ g)(x) = f(g(x))$ is defined for sufficiently large $x \in \mathbb{N}$.

The big-Oh notation. Let f, g be complexity functions. We say f *dominates* g if $f(x) \geq g(x)$ for all sufficiently large x, and provided $f(x), g(x)$ are both defined. By "sufficiently large x" or "large enough x" we mean "for all $x \geq x_0$" where x_0 is some unspecified constant.

The *big-Oh notation*asymptotic notation!big-Oh is the most famous member of a family of asymptotic notations. The prototypical use of this notation goes as follows. We say f *is big-Oh of* g (or, f *is order of* g) and write

$$f = O(g) \tag{10}$$

if there is a constant $C > 0$ such that $C \cdot g(x)$ dominates $f(x)$. As examples of usage, $f(x) = O(1)$ (respectively, $f(x) = x^{O(1)}$) means that $f(x)$ is eventually bounded by some constant (respectively, by some polynomial). Or again, $n \log n = O(n^2)$ and $1/n = O(1)$ are both true.

Our definition in Equation (10) gives a very specific formula for using the big-Oh notation. We now describe an extension. Recursively define *O-expressions* as follows. Basis: If g is a symbol for a complexity function, then g is an O-expression. Induction: If E_i $(i = 1, 2)$ are O-expressions, then so are

$$O(E_1), \quad E_1 \pm E_2, \quad E_1 E_2, \quad E_1^{E_2}, \quad E_1 \circ E_2.$$

Each O-expression *denotes* a set of complexity functions. Basis: The O-expression g denotes the singleton set $\{\overline{g}\}$ where $\overline{g}$ is the function denoted by g. Induction: If E_i denotes the set of complexity functions $\overline{E}_i$ then the O-expression $O(E_1)$ denotes the set of complexity functions $\overline{f}$ such that there is some $\overline{g} \in \overline{E}_1$ and $C > 0$ and $\overline{f}$ is dominated by $C\overline{g}$. The expression $E_1 + E_2$ denotes the set of functions of the form $f_1 + f_2$ where $f_i \in \overline{E}_i$. Similarly for $E_1 E_2$ (product), $E_1^{E_2}$ (exponentiation) and $E_1 \circ E_2$ (function composition). Finally, we use these O-expressions to assert the containment relationship: we write

$$E_1 = E_2,$$

to mean $\overline{E}_1 \subseteq \overline{E}_2$. Clearly, the equality symbol in this context is asymmetric. In actual usage, we take the usual license of confusing a function symbol g with the function $\overline{g}$ that it denotes. Likewise, we confuse the concept of an O-expression with the set of functions it denotes. By convention, the expressions 'c' ($c \in \mathbb{R}$) and 'n' denote (respectively) the constant function c and the identity function. Then 'n^2' and '$\log n$' are O-expressions denoting the (singleton set containing the) square function and logarithm function. Other examples of O-expressions: $2^{n+O(\log n)}$, $O(O(n)^{\log n} + n^{O(n)} \log\log n)$, $f(n) \circ O(n \log n)$. Of course, all these conventions depends on fixing 'n' as the distinguished variable. Note that $1 + O(1/n)$ and $1 - O(1/n)$ are different O-expressions because of our insistence that complexity functions are eventually non-negative.

The subscripting convention. There is another useful way to extend the basic formulation of Equation (10): instead of viewing its right-hand side "$O(g)$" as denoting a set of functions (and hence the equality sign as set membership '$\in$' or set inclusion '$\subseteq$'), we can view it as denoting some *particular* function $C \cdot g$ that dominates f. The big-Oh notation in this view is just a convenient way of hiding the constant 'C' (it saves us the trouble of inventing a symbol for this constant). In this case, the equality sign is interpreted as the "dominated by" relation, which explains the tendency of some to write '$\leq$' instead of the equality sign. Usually, the need for this interpretation arises because we want to obliquely refer to the implicit constant. For instance, we may want to indicate that the implicit constants in two occurrences of the same O-expression are really the same. To achieve this cross reference, we use a subscripting convention: *we can attach a subscript or subscripts to the O, and this particularizes that O-expression to refer to some fixed function.* Two identical O-expressions with identical subscripts refer to the same implicit constants. By choosing the subscripts judiciously, this notation can be quite effective. For instance, instead of inventing a function symbol $T_A(n) = O(n)$ to denote the running time of a linear-time algorithm A, we may simply use the subscripted expression "$O_A(n)$"; subsequent use of this expression will refer to the same function. Another simple illustration is "$O_3(n) = O_1(n) + O_2(n)$": the sum of two linear functions is linear, with different implicit constant for each subscript.

Related asymptotic notations. We say f is *big-Omega of* g and write

$$f(n) = \Omega(g(n))$$

if there exists a real $C > 0$ such that $f(x)$ dominates $C \cdot g(x)$. We say f is *Theta of* g and write

$$f(n) = \Theta(g(n))$$

if $f = O(g)$ and $f = \Omega(g)$. We normally distinguish complexity functions up to Theta-order. We say f is *small-oh of* g and write

$$f(n) = o(g(n))$$

if $f(n)/g(n) \to 0$ as $n \to \infty$. We say f is *small-omega of* g and write

$$f(n) = \omega(g(n))$$

if $f(n)/g(n) \to \infty$ as $n \to \infty$. We write

$$f \sim g$$

if $f = g[1 \pm o(1)]$. For instance, $n + \log n \sim n$ but not $n + \log n \sim 2n$.

These notations can be extended as in the case of the big-Oh notation. The semantics of mixing these notations are less obvious and is, in any case, not needed.

§7. Complexity of Multiplication

We introduce three "intrinsic" complexity functions,

$$\mathrm{M}_B(n), \quad \mathrm{M}_A(n), \quad \mathrm{MM}(n)$$

related to multiplication in various domains under various complexity models. These functions are useful in bounding other complexity functions. This leads to a discussion of intrinsic complexity.

Complexity of multiplication. Let us first fix the model of computation to be the multitape Turing machine. We are interested in the *intrinsic Turing complexity* T_P of a computational problem P, namely the intrinsic (time) cost of solving P on the Turing machine model. Intuitively, we expect $T_P = T_P(n)$ to be a complexity function, corresponding to the "optimal" Turing machine for P. If there is no optimal Turing machine, this is problematic – – see below for a proper treatment of this. If P is the problem of multiplying two binary integers, then the fundamental quantity $T_P(n)$ appears in the complexity bounds of many other problems, and is given the special notation

$$\mathrm{M}_B(n)$$

in this book. For now, we will assume that $\mathrm{M}_B(n)$ is a complexity function. The best upper bound for $\mathrm{M}_B(n)$ is

$$\mathrm{M}_B(n) = O(n \log n \log\log n), \tag{11}$$

from a celebrated result [20] of Schönhage and Strassen (1971). To simplify our display of such bounds (cf. [18, 13]), we write $\mathcal{L}^k(n)$ $(k \geq 1)$ to denote some fixed but non-specific function $f(n)$ that satisfies

$$\frac{f(n)}{\log^k n} = o(\log n).$$

If $k = 1$, the superscript in $\mathcal{L}^1(n)$ is omitted. In this notation, equation (11) simplifies to

$$\mathrm{M}_B(n) = n\mathcal{L}(n).$$

Note that we need not explicitly write the big-Oh here since this is implied by the $\mathcal{L}(n)$ notation. Schönhage [19] (cf. [11, p. 295]) has shown that the complexity of integer multiplication takes a simpler form with alternative computational models (see §6): *A successor RAM can multiply two n-bit integers in $O(n)$ time under the unit cost model, and in $O(n \log n)$ time in the logarithmic cost model.*

Next we introduce the *algebraic complexity of multiplying two degree n polynomials*, denoted

$$\mathrm{M}_A(n).$$

The basis (§6) for our algebraic programs is comprised of the ring operations of R, where the polynomials are from $R[X]$. Trivially, $\mathrm{M}_A(n) = O(n^2)$ but Lecture I will show that

$$\mathrm{M}_A(n) = O(n \log n).$$

Finally, we introduce the *algebraic complexity of multiplying two $n \times n$ matrices.* We assume the basis is comprised of the ring operations of a ring R, where the matrix entries come from R. This is another fundamental quantity which will be denoted by

$$\mathrm{MM}(n)$$

in this book. Clearly $\mathrm{MM}(n) = O(n^3)$ but a celebrated result of Strassen (1968) shows that this is suboptimal. The current record (see Lecture I) is

$$\mathrm{MM}(n) = O(n^{2.376}). \tag{12}$$

On Intrinsic Complexity.

The notation "$\mathrm{M}_B(n)$" is not rigorous when naively interpreted as a complexity function. Let us see why. More generally, let us fix a *complexity model M*: this means we fix a computational model (Turing machines, RAM, etc) and associate a complexity function $T_A(n)$ to each program A in M as in §7. But complexity theory really begins when we associate an *intrinsic complexity function $T_P(n)$* with each computational problem P. Thus, $\mathrm{M}_B(n)$ is the intrinsic complexity function for the problem of multiplying two binary integers in the standard (worst-case time) Turing complexity model. But how shall we define $T_P(n)$?

First of all, we need to clarify the concept of a "computational problem". One way is to introduce a logical language for specifying problems. But for our purposes, *we will simply identify a computational problem P with a set of programs in model M*. The set P comprises those programs in M that is said to "solve" the problem. For instance, the integer multiplication problem is identified with the set P_{mult} of all Turing machines that, started with $\overline{m}\#\overline{n}$ on the input tape, eventually halts with the product $\overline{mn}$ on the output tape (where $\overline{n}$ is the binary representation of $n \in \mathbb{N}$). If P is a problem and $A \in P$, we say *A solves P* or *A is an algorithm for P*. A complexity function $f(n)$ is an *upper bound*intrinsic complexity!upper bound on the problem P if there is an algorithm A for P such that $f(n)$ dominates $T_A(n)$. If, for every algorithm A for P, $T_A(n)$ dominates $f(n)$, then we call $f(n)$ a *lower bound*intrinsic complexity!lower bound on the problem P.

Let U_P be the set of upper bounds on P. Notice that there exists a unique complexity function $\ell_P(n)$ such that $\ell_P(n)$ is a lower bound on P and for any other lower bound $f(n)$ on P, $\ell_P(n)$ dominates $f(n)$. To see this, define for each n, $\ell_P(n) := \inf\{f(n) : f \in U_P\}$. On the other hand, there may not exist $T(n)$ in U_P that is dominated by all other functions in U_P; if $T(n)$ exists,

it would (up to co-domination) be equal to $\ell_P(n)$. In this case, we may call $\ell_P(n) = T(n)$ the *intrinsic complexity* $T_P(n)$ *of* P. To resolve the case of the "missing intrinsic complexity", we generalize our concept of a function: An *intrinsic (complexity) function* is *intrinsic (complexity) function* any non-empty family U of complexity functions that is closed under domination, *i.e.*, if $f \in U$ and g dominates f then $g \in U$. The set U_P of upper bounds of P is an intrinsic function: we identify this as the *intrinsic complexity* T_P of P. A subset $V \subseteq U$ is called a *generating set* of U if every $f \in U$ dominates some $g \in V$. We say U is *principal* if U has a generating set consisting of one function f_0; in this case, we call f_0 a *generator* of U. If f is a complexity function, we will identify f with the principal intrinsic function with f as a generator. Note that in non-uniform computational models, the intrinsic complexity of any problem is principal.

Let U, T be intrinsic functions. We extend the standard terminology for ordinary complexity functions to intrinsic functions. Thus

$$U + T, \quad UT, \quad U^T, \quad U \circ T \tag{13}$$

denote intrinsic functions in the natural way. For instance, $U + T$ denotes the intrinsic function generated by the set of functions of the form $u + t$ where $u \in U$ and $t \in T$. We say U *is big-Oh of* T, written

$$U = O(T),$$

if there exists $u \in U$ such that for all $t \in T$, we have $u = O(t)$ in the usual sense. The reader should test these definitions by interpreting $\mathrm{M}_B(n)$, etc, as intrinsic functions (e.g., see (14) in §9). Basically, these definitions allow us to continue to talk about intrinsic functions rather like ordinary complexity functions, provided we know how to interpret them. Similarly, we say U *is big-Omega of* T, written $U = \Omega(T)$, if for all $u \in U$, there exists $t \in T$ such that $u = \Omega(t)$. We say U *is Theta of* T, written $U = \Theta(T)$, if $U = O(T)$ and $U = \Omega(T)$.

Complexity Classes. Corresponding to each computational model, we have complexity classes of problems. Each complexity class is usually characterized by a complexity model (worst-case time, randomized space, etc) and a set of complexity bounds (polynomial, etc). The class of problems that can be solved in polynomial time on a Turing machine is usually denoted P: it is arguably the most important complexity class. This is because we identify this class with the "feasible problems". For instance, the the Fundamental Problem of Algebra (in its various forms) is in P but the Fundamental Problem of Classical Algebraic Geometry is not in P. Complexity theory can be characterized as the study of relationships among complexity classes. Keeping this fact in mind may help motivate much of our activities. Another important class is NC which comprises those problems that can be solved *simultaneously* in depth $\log^{O(1)} n$ and size $n^{O(1)}$, under the Boolean circuit model. Since circuit depth equals parallel time, this is an important class in parallel computation. Although we did not define the circuit analogue of algebraic programs, this is rather straightforward: they are like Boolean circuits except we perform algebraic operations at the nodes. Then we can define NC_A, the algebraic analogue of the class NC. Note that NC_A is defined relative to the underlying algebraic ring.

Exercises

Exercise 7.1: Prove the existence of a problem whose intrinsic complexity is not principal. (In Blum's axiomatic approach to complexity, such problems exist.) □

§8. On Bit versus Algebraic Complexity

We have omitted other important models such as pointer machines that have a minor role in algebraic complexity. But why such a proliferation of models? Researchers use different models depending on the problem at hand. We offer some guidelines for these choices.

1. There is a consensus in complexity theory that the Turing model is the most basic of all general-purpose computational models. To the extent that algebraic complexity seeks to be compatible to the rest of complexity theory, it is preferable to use the Turing model.

2. In practice, the RAM model is invariably used to describe algebraic algorithms because the Turing model is too cumbersome. Upper bounds (*i.e.*, algorithms) are more readily explained in the RAM model and we are happy to take advantage of this in order to make the result more accessible. Sometimes, we could further assert ("left to the reader") that the RAM result extends to the Turing model.

3. Complexity theory proper is regarded to be a theory of "uniform complexity". This means "naturally" uniform models such as Turing machines are preferred over "naturally non-uniform" models such as Boolean circuits. Nevertheless, non-uniform models have the advantage of being combinatorial and conceptually simpler. Historically, this was a key motivation for studying Boolean circuits, since it is hoped that powerful combinatorial arguments may yield super-quadratic lower bounds on the Boolean size of specific problems. Such a result would immediately imply non-linear lower bounds on Turing machine time for the same problem. (Unfortunately, neither kind of result has been realized.) Another advantage of non-uniform models is that the intrinsic complexity of problems is principal. Boolean circuits also seems more natural in the parallel computation domain, with circuit depth corresponding to parallel time.

4. The choice between bit complexity and the algebraic complexity is problem-dependent. For instance, the algebraic complexity of integer GCD would not make much sense (§6, Exercise). But bit complexity is meaningful for any problem (the encoding of the problem must be taken into account). This may suggest that algebraic complexity is a more specialized tool than bit complexity. But even in a situation where bit complexity is of primary interest, it may make sense to investigate the corresponding algebraic complexity. For instance, the algebraic complexity of multiplying integer matrices is $\mathrm{MM}(n) = O(n^{2.376})$ as noted above. Let[3] $\mathrm{MM}(n, N)$ denote the Turing complexity of integer matrix multiplication, where N is an additional bound on the bit size of each entry of the matrix. The best upper bound for $\mathrm{MM}(n, N)$ comes from the trivial remark,

$$\mathrm{MM}(n, N) = O(\mathrm{MM}(n)\mathrm{M}_B(N)). \tag{14}$$

That is, the known upper bound on $\mathrm{MM}(n, N)$ comes from the separate upper bounds on $\mathrm{MM}(n)$ and $\mathrm{M}_B(N)$.

Linear Programming. Equation (14) illustrates a common situation, where the best bit complexity of a problem is obtained as the best algebraic complexity multiplied by the best bit complexity on the underlying operations. We now show an example where this is not the case. Consider the linear programming problem. Let m, n, N be complexity parameters where the linear constraints are represented by $Ax \leq b$, A is an $m \times n$ matrix, and all the numbers in A, b have at most N bits. The linear programming problem can be reduced to checking for the feasibility of the inequality $Ax \leq b$, on input A, b. The Turing complexity $T_B(m, n, N)$ of this problem is known to be polynomial in m, n, N. This result was a breakthrough, due to Khacian in 1979. On the other hand, it is a major open problem whether the corresponding algebraic complexity $T_A(m, n)$ of linear programming is polynomial in m, n.

Euclidean shortest paths. In contrast to linear programming, we now show a problem for which the bit complexity is not known to be polynomial but whose algebraic complexity is polynomial.

[3]The bit complexity bound on any problem is usually formulated to have one more size parameter (N) than the corresponding algebraic complexity bound.

This is the problem of finding the shortest paths between two points on the plane. Let us formulate a version of the *Euclidean shortest path problem*: we are given a planar graph G that is linearly embedded in the plane, *i.e.*, each vertex v of G is mapped to a point $m(v)$ in the plane and each edge (u, v) between two vertices is represented by the corresponding line segment $[m(u), m(v)]$, where two segments may only intersect at their endpoints. We want to find the shortest (under the usual Euclidean metric) path between two specified vertices s, t. Assume that the points $m(v)$ have rational coordinates. Clearly this problem can be solved by Djikstra's algorithm in polynomial time, provided we can (i) take square-roots, (ii) add two sums of square-roots, and (iii) compare two sums of square-roots in constant time. Thus the algebraic complexity is polynomial time (where the basis operations include (i-iii)). However, the current best bound on the bit complexity of this problem is single exponential space. Note that the numbers that arise in this problem are the so-called *constructible reals* (Lecture VI) because they can be finitely constructed by a ruler and a compass.

The lesson of these two examples is that bit complexity and algebraic complexities do not generally have a simple relationship. Indeed, we cannot even expect a polynomial relationship between these two types of complexities: depending on the problem, either one could be exponentially worse than the other.

Exercises

Exercise 8.1*: Obtain an upper bound on the above Euclidean shortest path problem. □

Exercise 8.2: Show that a real number of the form

$$\alpha = n_0 \pm \sqrt{n_1} \pm \sqrt{n_2} \pm \cdots \pm \sqrt{n_k}$$

(where n_i are positive integers) is a zero of a polynomial $P(X)$ of degree at most 2^k, and that all zeros of $P(X)$ are real. □

§9. Miscellany

This section serves as a quick general reference.

Equality symbol. We introduce two new symbols to reduce[4] the semantic overload commonly placed on the equality symbol '='. We use the symbol '$\leftarrow$' for *programming variable assignments* , from right-hand side to the left. Thus, $V \leftarrow V + W$ is an assignment to V (and it could appear on the right-hand side, as in this example). We use the symbol ':=' to denote *definitional equality*, with the term being defined on the left-hand side and the defining terms on the right-hand side. Thus, "$f(n) := n \log n$" is a definition of the function f. Unlike some similar notations in the literature, we refrain from using the mirror images of the definition symbol (we will neither write "$V + W \rightarrow V$" nor "$n \log n =: f(n)$").

Sets and functions. The empty set is written $\emptyset$. Let A, B be sets. Subsets and proper subsets are respectively indicated by $A \subseteq B$ and $A \subset B$. Set difference is written $A \setminus B$. Set formation is usually written $\{x : \ldots x \ldots\}$ and sometimes written $\{x | \ldots x \ldots\}$ where $\ldots x \ldots$ specifies some

[4]Perhaps to atone for our introduction of the asymptotic notations.

properties on x. The A is the union of the sets A_i for $i \in I$, we write $A = \cup_{i\in I} A_i$. If the A_i's are pairwise disjoint, we indicate this by writing

$$A = \uplus_{i\in I} A_i.$$

Such a disjoint union is also called a *partition* of A. Sometimes we consider *multisets*. A multiset S can be regarded as sets whose elements can be repeated – the number of times a particular element is repeated is called its *multiplicity*. Alternatively, S can be regarded as a function $S : D \to \mathbb{N}$ where D is an ordinary set and $S(x) \geq 1$ gives the multiplicity of x. We write $f \circ g$ for the composition of functions $g : U \to V$, $f : V \to W$. So $(f \circ g)(x) = f(g(x))$. If a function f is undefined for a certain value x, we write $f(x) \uparrow$.

Numbers. Let $\mathbf{i}$ denote $\sqrt{-1}$, the square-root of -1. For a complex number $z = x + \mathbf{i}y$, let $\mathtt{Re}(z) := x$ and $\mathtt{Im}(z) := y$ denote its real and imaginary part, respectively. Its *modulus* $|z|$ is defined to be the positive square-root of $x^2 + y^2$. If z is real, $|z|$ is also called the *absolute value* . The *(complex) conjugate* of z is defined to be $\overline{z} := \mathtt{Re}(z) - \mathtt{Im}(z)$. Thus $|z|^2 = z\overline{z}$.

But if S is any set, $|S|$ will refer to the *cardinality* , *i.e.*, the number of elements in S. This notation should not cause a confusion with the notion of modulus of z.

For a real number r, we use Iverson's notation (as popularized by Knuth) $\lceil r \rceil$ and $\lfloor r \rfloor$ for the *ceiling* and *floor* functions. We have

$$\lfloor r \rfloor \leq \lceil r \rceil .$$

In this book, we introduce the *symmetric ceiling* and *symmetric floor* functions:

$$\lceil r \rceil_s := \begin{cases} \lceil r \rceil & \text{if } r \geq 0, \\ \lfloor r \rfloor & \text{if } r < 0. \end{cases}$$

$$\lfloor r \rfloor_s := \begin{cases} \lfloor r \rfloor & \text{if } r \geq 0, \\ \lceil r \rceil & \text{if } r < 0. \end{cases}$$

These functions satisfy the following inequalities, valid for all real numbers r:

$$|\, \lfloor r \rfloor_s \,| \leq |r| \leq |\, \lceil r \rceil_s \,|.$$

(The usual floor and ceiling functions fail this inequality when r is negative.) We also use $\lfloor r \rceil$ to denote the *rounding* function, $\lfloor r \rceil := \lceil r - 0.5 \rceil$. So

$$\lfloor r \rfloor \leq \lfloor r \rceil \leq \lceil r \rceil.$$

The base of the *logarithm function* $\log x$, is left unspecified if this is immaterial (as in the notation $O(\log x)$). On the other hand, we shall use

$$\lg x, \qquad \ln x$$

for logarithm to the base 2 and the natural logarithm, respectively.

Let a, b be integers. If $b > 0$, we define the *quotient* and *remainder functions* , $\mathtt{quo}(a, b)$ and $\mathtt{rem}(a, b)$ which satisfy the relation

$$a = \mathtt{quo}(a, b) \cdot b + \mathtt{rem}(a, b)$$

such that $b > \mathtt{rem}(a, b) \geq 0$. We also write these functions using an in-fix notation:

$$(a \,\mathbf{div}\, b) := \mathtt{quo}(a, b); \qquad (a \,\mathbf{mod}\, b) := \mathtt{rem}(a, b).$$

These functions can be generalized to Euclidean domains (lecture II, §2). We continue to use 'mod' in the standard notation "$a \equiv b (\mathrm{mod}\, m)$" for congruence modulo m. We say a *divides* b if $\mathtt{rem}(a, b) = 0$, and denote this by "$a \,|\, b$". If a does not divide b, we denote this by "$a \nmid b$".

Norms. For a complex polynomial $P \in \mathbb{C}[X]$ and for each positive real number k, let $\|P\|_k$ denote[5] the *k-norm* ,

$$\|P\|_k := \left(\sum_{i=0}^{n} |p_i|^k\right)^{1/k}$$

where $p_0, \ldots, p_n$ are the coefficients of P. We extend this definition to $k = \infty$, where

$$\|P\|_\infty := \max\{|p_i| : i = 0, \ldots, n\}. \tag{15}$$

There is a related L_k-norm defined on P where we view P as a complex function (in contrast to L_k-norms, it is usual to refer to our k-norms as "ℓ_k-norms"). The L_k-norms are less important for us. Depending on context, we may prefer to use a particular k-norm: in such cases, we may simply write "$\|P\|$" instead of "$\|P\|_k$". For $0 < r < s$, we have

$$\|P\|_\infty \le \|P\|_s < \|P\|_r \le (n+1)\|P\|_\infty \tag{16}$$

The second inequality (called Jensen's inequality) follows from:

$$\begin{aligned}\frac{(\sum_i |p_i|^s)^{1/s}}{(\sum_j |p_j|^r)^{1/r}} &= \left\{\sum_{i=0}^{n} \frac{|p_i|^s}{(\sum_j |p_j|^r)^{s/r}}\right\}^{\frac{1}{s}} = \left\{\sum_{i=0}^{n} \left(\frac{|p_i|^r}{\sum_j |p_j|^r}\right)^{\frac{s}{r}}\right\}^{\frac{1}{s}} \\ &< \left\{\sum_{i=0}^{n} \left(\frac{|p_i|^r}{\sum_j |p_j|^r}\right)\right\}^{\frac{1}{r}} = 1.\end{aligned}$$

The 1-, 2- and ∞-norms of P are also known as the *weight*, *length*, and *height* of P. If $\mathbf{u}$ is a vector of numbers, we define its *k-norm* $\|\mathbf{u}\|_k$ by viewing $\mathbf{u}$ as the coefficient vector of a polynomial. The following inequality will be useful:

$$\|P\|_1 \le \sqrt{n}\|P\|_2.$$

To see this, note that $n\sum_{i=1}^n a_i^2 \ge (\sum_{i=1}^n a_i)^2$ is equivalent to $(n-1)\sum_{i=1}^n a_i^2 \ge 2\sum_{1\le i<j\le n} a_i a_j$. But this amounts to $\sum_{1\le i<j\le n}(a_i - a_j)^2 \ge 0$.

Inequalities. Let $\mathbf{a} = (a_1, \ldots, a_n)$ and $\mathbf{b} = (b_1, \ldots, b_n)$ be real n-vectors. We write $\mathbf{a}\cdot\mathbf{b}$ or $\langle \mathbf{a}, \mathbf{b}\rangle$ for their scalar product $\sum_{i=1}^n a_i b_i$.

Hölder's Inequality: If $\frac{1}{p} + \frac{1}{q} = 1$ then

$$|\langle \mathbf{a}, \mathbf{b}\rangle| \le \|\mathbf{a}\|_p \|\mathbf{b}\|_q,$$

with equality iff there is some k such that $b_i^q = k a_i^p$ for all i. In particular, we have the Cauchy-Schwarz Inequality:

$$|\langle \mathbf{a}, \mathbf{b}\rangle| \le \|\mathbf{a}\|_2 \cdot \|\mathbf{b}\|_2.$$

Minkowski's Inequality: for $k > 1$,

$$\|\mathbf{a} + \mathbf{b}\|_k \le \|\mathbf{a}\|_k + \|\mathbf{b}\|_k.$$

This shows that the k-norms satisfy the triangular inequality.

A real function $f(x)$ defined on an interval $I = [a, b]$ is *convex* on I if for all $x, y \in I$ and $0 \le \alpha \le 1$, $f(\alpha x + (1-\alpha)y) \le \alpha f(x) + (1-\alpha)f(y)$. For instance, if $f''(x)$ is defined and $f''(x) \ge 0$ on I implies f is convex on I.

[5] In general, a *norm* of a real vector V is a real function $N : V \to \mathbb{R}$ such that for all $x \in V$, (i) $N(x) \ge 0$ with equality iff $x = \mathbf{0}$, (ii) $N(cx) = |c|N(x)$ for any $c \in \mathbb{R}$, and (iii) $N(x+y) \le N(x) + N(y)$. The k-norms may be verified to be a norm in this sense.

Polynomials. Let $A(X) = \sum_{i=0}^{n} a_i X^i$ be a univariate polynomial. Besides the notation $\deg(A)$ and $\texttt{lead}(A)$ of §1, we are sometimes interested in the largest power $j \geq 0$ such that X^j divides $A(X)$; this j is called the *tail degree* of A. The coefficient a_j is the *tail coefficient* of A, denoted $\texttt{tail}(A)$.

Let $\mathbf{X} = \{X_1, \ldots, X_n\}$ be $n \geq 1$ (commutative) variables, and consider multivariate polynomials in $R[\mathbf{X}]$. A *power product* over $\mathbf{X}$ is a polynomial of the form $T = \prod_{i=1}^{n} X_i^{e_i}$ where each $e_i \geq 0$ is an integer. In particular, if all the e_i's are 0, then $T = 1$. The *total degree* $\deg(T)$ of T is given by $\sum_{i=1}^{n} e_i$, and the *maximum degree* $\texttt{mdeg}(T)$ is given by $\max_{i=1}^{n} e_i$. Usually, we simply say "degree" for total degree. Let $\texttt{PP}(\mathbf{X}) = \texttt{PP}(X_1, \ldots, X_n)$ denote the set of power products over $\mathbf{X}$.

A *monomial* or *term* is a polynomial of the form cT where T is a power product and $c \in R \setminus \{0\}$. So a polynomial A can be written uniquely as a sum $A = \sum_{i=1}^{k} A_i$ of monomials with distinct power products; each such monomial A_i is said to *belong* to A. The *(term) length* of a polynomial A to be the number of monomials in A, not to be confused with its Euclidean length $\|A\|_2$ defined earlier. The total degree $\deg(A)$ (respectively, maximum degree $\texttt{mdeg}(A)$) of a polynomial A is the largest total (respectively, maximum) degree of a power product in A. Usually, we just say "degree" of A to mean total degree. A polynomial is *homogeneous* if each of its monomials has the same total degree. Again, any polynomial A can be written uniquely as a sum $A = \sum_i H_i$ of homogeneous polynomials H_i of distinct degrees; each H_i is said to be a *homogeneous component* of A.

The degree concepts above can be generalized. If $\mathbf{X}_1 \subseteq \mathbf{X}$ is a set of variables, we may speak of the "$\mathbf{X}_1$-degree" of a polynomial A, or say that a polynomial "homogeneous" in $\mathbf{X}_1$, simply by viewing A as a polynimial in $\mathbf{X}_1$. Or again, if $\mathbf{Y} = \{\mathbf{X}_1, \ldots, \mathbf{X}_k\}$ is a partition of the variables $\mathbf{X}$, the "$\mathbf{Y}$-maximum degree" of A is the maximum of the $\mathbf{X}_i$-degrees of A $(i = 1, \ldots, k)$.

Matrices. The set of $m \times n$ matrices with entries over a ring R is denoted $R^{m \times n}$. Let $M \in R^{m \times n}$. If the (i,j)th entry of M is x_{ij}, we may write $M = [x_{ij}]_{i,j=1}^{m,n}$ (or simply, $M = [x_{ij}]_{i,j}$). The (i,j)th entry of M is denoted $M(i;j)$. More generally, if $i_1, i_2, \ldots, i_k$ are indices of rows and $j_1, \ldots, j_\ell$ are indices of columns,

$$M(i_1, \ldots, i_k; j_1, \ldots, j_\ell) \tag{17}$$

denotes the *submatrix* obtained by intersecting the indicated rows and columns. In case $k = \ell = 1$, we often prefer to write $(M)_{i,j}$ or $(M)_{ij}$ instead of $M(i;j)$. If we delete the ith row and jth column of M, the resulting matrix is denoted $M[i;j]$. Again, this notation can be generalized to deleting more rows and columns. E.g., $M[i_1, i_2; j_1, j_2, j_3]$ or $[M]_{i_1,i_2;j_1,j_2,j_3}$. The *transpose* of M is the $n \times m$ matrix, denoted M^T, such that $M^T(i;j) = M(j;i)$.

A *minor* of M is the determinant of a square submatrix of M. The submatrix in (17) is *principal* if $k = \ell$ and

$$i_1 = j_1 < i_2 = j_2 < \cdots < i_k = j_k.$$

A minor is *principal* if it is the determinant of a principal submatrix. If the submatrix in (17) is principal with $i_1 = 1, i_2 = 2, \ldots, i_k = k$, then it is called the "kth principal submatrix" and its determinant is the "kth principal minor". (Note: the literature sometimes use the term "minor" to refer to a principal submatrix.)

Ideals. Let R be a ring and I, J be ideals of R. The ideal *generated* by elements $a_1, \ldots, a_m \in R$ is denoted $(a_1, \ldots, a_m)$ and is defined to be the smallest ideal of R containing these elements. Since

this well-known notation for ideals may be ambiguous, we sometimes write[6]

$$\texttt{Ideal}(a_1, \ldots, a_m).$$

Another source of ambiguity is the underlying ring R that generates the ideal; thus we may sometimes write

$$(a_1, \ldots, a_m)_R \quad \text{or} \quad \texttt{Ideal}_R(a_1, \ldots, a_m).$$

An ideal I is *principal* if it is generated by one element, $I = (a)$ for some $a \in R$; it is *finitely generated* if it is generated by some finite set of elements. For instance, the *zero ideal* is $(0) = \{0\}$ and the *unit ideal* is $(1) = R$. Writing $aR := \{ax : x \in R\}$, we have that $(a) = aR$, exploiting the presence of $1 \in R$. A *principal ideal ring* or *domain* is one in which every ideal is principal. An ideal is called *homogeneous* (resp., *monomial*) if it is generated by a set of homogeneous polynomials (resp., monomials).

The following are five basic operations defined on ideals:

Sum: $I + J$ is the ideal consisting of all $a + b$ where $a \in I, b \in J$.

Product: IJ is the ideal generated by all elements of the form ab where $a \in I, b \in J$.

Intersection: $I \cap J$ is just the set theoretic intersection of I and J.

Quotient: $I : J$ is defined to be the set $\{a | aJ \subseteq I\}$. If $J = (a)$, we simply write $I : a$ for $I : J$.

Radical: $\sqrt{I}$ is defined to be set $\{a | (\exists n \geq 1) a^n \in I\}$.

Some simple relationships include $IJ \subseteq I \cap J$, $I(J + J') = IJ + IJ'$, $(a_1, \ldots, a_m) + (b_1, \ldots, b_n) = (a_1, \ldots, a_m, b_1, \ldots, b_n)$. An element b is *nilpotent* if some power of b vanishes, $b^n = 0$. Thus $\sqrt{(0)}$ is the set of nilpotent elements. An ideal I is *maximal* if $I \neq R$ and it is not properly contained in an ideal $J \neq R$. An ideal I is *prime* if $ab \in I$ implies $a \in I$ or $b \in I$. An ideal I is *primary* if $ab \in I, a \notin I$ implies $b^n \in I$ for some positive integer n. A ring with unity is *Noetherian* if every ideal I is finitely generated. It turns out that for Noetherian rings, the basic building blocks are primary ideals (not prime ideals). We assume the reader is familiar with the construction of ideal quotient rings, R/I.

Exercises

Exercise 9.1: (i) Verify the rest of equation (16).
(ii) $\|A \pm B\|_1 \leq \|A\|_1 + \|B\|_1$ and $\|AB\|_1 \leq \|A\|_1 \|B\|_1$.
(iii) (Duncan) $\|A\|_2 \|B\|_2 \leq \|AB\|_2 \sqrt{\binom{2n}{n}\binom{2m}{m}}$ where $\deg(A) = m$, $\deg(B) = n$. □

Exercise 9.2: Show the inequalities of Hölder and Minkowski. □

Exercise 9.3: Let $I \neq R$ be an ideal in a ring R with unity.
a) I is maximal iff R/I is a field.
b) I is prime iff R/I is a domain.
c) I is primary iff every zero-divisor in R/I is nilpotent. □

[6] Cf. the notation $\textsc{Ideal}(U) \subseteq R_0[X_1, \ldots, X_d]$ where $U \in \mathbb{A}^d(R_1)$, introduced in §4. We capitalize the names of maps from an algebraic to a geometric setting or vice-versa. Thus Ideal, Zero.

§10. Computer Algebra Systems

In a book on algorithmic algebra, we would be remiss if we make no mention of *computer algebra systems*. These are computer programs that manipulate and compute on symbolic ("algebraic") quantities as opposed to just numerical ones. Indeed, there is an intimate connection between algorithmic algebra today and the construction of such programs. Such programs range from general purpose systems (e.g., `Maple`, `Mathematica`, `Reduce`, `Scratchpad`, `Macsyma`, etc.) to those that target specific domains (e.g., `Macaulay` (for Gröbner bases), `MatLab` (for numerical matrices), `Cayley` (for groups), `SAC-2` (polynomial algebra), `CM` (celestial mechanics), `QES` (quantum electrodynamics), etc.). It was estimated that about 60 systems exist around 1980 (see [23]). A computer algebra book that discuss systems issues is [8]. In this book, we choose to focus on the mathematical and algorithmic development, *independent of any computer algebra system.* Although it is possible to avoid using a computer algebra system in studying this book, we strongly suggest that the student learn at least one general-purpose computer algebra system and use it to work out examples. If any of our exercises make system-dependent assumptions, it may be assumed that `Maple` is meant.

Exercises

Exercise 10.1: It took J. Bernoulli (1654-1705) less than 1/8 of an hour to compute the sum of the 10th power of the first 1000 numbers: $91, 409, 924, 241, 424, 243, 424, 241, 924, 242, 500$.

(i) Write a procedure `bern(n,e)` in your favorite computer algebra system, so that the above number is computed by calling `bern(`$1000, 10$`)`.

(ii) Write a procedure `berns(m,n,e)` that runs `bern(n,e)` m times. Do simple profiling of the functions `bern, berns`, by calling `berns(`$100, 1000, 10$`)`. □

References

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

[2] S. Akbulut and H. King. *Topology of Real Algebraic Sets*. Mathematical Sciences Research Institute Publications. Springer-Verlag, Berlin, 1992.

[3] M. Artin. *Algebra*. Prentice Hall, Englewood Cliffs, NJ, 1991.

[4] R. Benedetti and J.-J. Risler. *Real Algebraic and Semi-Algebraic Sets*. Actualités Mathématiques. Hermann, Paris, 1990.

[5] A. Borodin and I. Munro. *The Computational Complexity of Algebraic and Numeric Problems*. American Elsevier Publishing Company, Inc., New York, 1975.

[6] W. D. Brownawell. Bounds for the degrees in Nullstellensatz. *Ann. of Math.*, 126:577–592, 1987.

[7] B. Buchberger, G. E. Collins, and R. L. (eds.). *Computer Algebra*. Springer-Verlag, Berlin, 2nd edition, 1983.

[8] J. H. Davenport, Y. Siret, and E. Tournier. *Computer Algebra: Systems and Algorithms for Algebraic Computation*. Academic Press, New York, 1988.

[9] J. Dieudonné. *History of Algebraic Geometry*. Wadsworth Advanced Books & Software, Monterey, CA, 1985. Trans. from French by Judith D. Sally.

[10] A. G. Khovanskiĭ. *Fewnomials*, volume 88 of *Translations of Mathematical Monographs*. American Mathematical Society, Providence, RI, 1991. tr. from Russian by Smilka Zdravkovska.

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

[12] S. Landau and G. L. Miller. Solvability by radicals in polynomial time. *J. of Computer and System Sciences*, 30:179–208, 1985.

[13] L. Langemyr. *Computing the GCD of two polynomials over an algebraic number field*. PhD thesis, The Royal Institute of Technology, Stockholm, Sweden, January 1989. Technical Report TRITA-NA-8804.

[14] F. S. Macaulay. *The Algebraic Theory of Modular Systems*. Cambridge University Press, Cambridge, 1916.

[15] B. Mishra. Computational real algebraic geometry. In J. O'Rourke and J. Goodman, editors, *CRC Handbook of Discrete and Comp. Geom.* CRC Press, Boca Raton, FL, 1997.

[16] D. A. Plaisted. New *NP*-hard and *NP*-complete polynomial and integer divisibility problems. *Theor. Computer Science*, 31:125–138, 1984.

[17] D. A. Plaisted. Complete divisibility problems for slowly utilized oracles. *Theor. Computer Science*, 35:245–260, 1985.

[18] M. O. Rabin. Probabilistic algorithms for finite fields. *SIAM J. Computing*, 9(2):273–280, 1980.

[19] A. Schönhage. Storage modification machines. *SIAM J. Computing*, 9:490–508, 1980.

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

[21] D. J. Struik, editor. *A Source Book in Mathematics, 1200-1800*. Princeton University Press, Princeton, NJ, 1986.

[22] B. L. van der Waerden. *Algebra*. Frederick Ungar Publishing Co., New York, 1970. Volumes 1 & 2.

[23] J. van Hulzen and J. Calmet. Computer algebra systems. In B. Buchberger, G. E. Collins, and R. Loos, editors, *Computer Algebra*, pages 221–244. Springer-Verlag, Berlin, 2nd edition, 1983.

[24] I. Wegener. *The Complexity of Boolean Functions*. B. G. Teubner, Stuttgart, and John Wiley, Chichester, 1987.

[25] W. T. Wu. *Mechanical Theorem Proving in Geometries: Basic Principles*. Springer-Verlag, Berlin, 1994. (Trans. from Chinese by X. Jin and D. Wang).

[26] K. Yokoyama, M. Noro, and T. Takeshima. On determining the solvability of polynomials. In *Proc. ISSAC'90*, pages 127–134. ACM Press, 1990.

[27] O. Zariski and P. Samuel. *Commutative Algebra*, volume 1. Springer-Verlag, New York, 1975.

[28] O. Zariski and P. Samuel. *Commutative Algebra*, volume 2. Springer-Verlag, New York, 1975.

Contents