# 3 Finite field arithmetic

In order to perform explicit computations with elliptic curves over finite fields, we first need to understand how to compute in finite fields. In many of the applications we will consider, the finite fields involved will be quite large, so it is important to understand the computational complexity of finite field operations. This is a huge topic, one to which an entire course could be devoted, but we will spend just one or two lectures on this topic, with the goal of understanding the most commonly used algorithms and analyzing their asymptotic complexity. This will force us to omit many details, but references to the relevant literature will be provided for those who want to learn more.

Our first step is to fix an explicit representation of finite field elements. This might seem like a technical detail, but it is actually quite crucial; questions of computational complexity are meaningless otherwise.

**Example 3.1.** By Theorem 3.12 below, the multiplicative group of a finite field $\mathbb{F}_q$ is cyclic. One way to represent the nonzero elements of a finite field is as explicit powers of a fixed generator, in which case it is enough to know the exponent, an integer in $[0, q-2]$. With this representation multiplication and division are easy, solving the discrete logarithm problem is trivial, but addition is costly (not known to be polynomial-time). We will instead choose a representation that makes addition (and subtraction) very easy, multiplication slightly harder but still easy, division slightly harder than multiplication but still easy (all these operations take quasi-linear time). But solving the discrete logarithm problem will be hard (no polynomial-time algorithm is known).

For the sake of brevity, we will focus primarily on finite fields of large characteristic, and prime fields in particular, although the algorithms we describe will work in any finite field of odd characteristic (most will also work in characteristic 2). Fields of characteristic 2 are quite important in many applications (coding theory in particular), and there are specialized algorithms that are optimized for such fields, but we will not address them here.[1]

## 3.1 Finite fields

We begin with a quick review of some basic facts about finite fields, all of which are straightforward but necessary for us to establish a choice of representation; we will also need them when we discuss algorithms for factoring polynomials over finite fields. Those already familiar with this material should feel free to skim this section.

**Definition 3.2.** For each prime $p$ we define $\mathbb{F}_p$ to be the quotient ring $\mathbb{Z}/p\mathbb{Z}$.

**Theorem 3.3.** *The ring $\mathbb{F}_p$ is a field, and every field of characteristic $p$ contains a canonical subfield isomorphic to $\mathbb{F}_p$. In particular, all fields of cardinality $p$ are isomorphic.*

*Proof.* To show that the ring $\mathbb{F}_p = \mathbb{Z}/p\mathbb{Z}$ is a field we just need to show that every nonzero element is invertible. If $[a] := a + p\mathbb{Z}$ is a nontrivial coset in $\mathbb{Z}/p\mathbb{Z}$ then $a$ and $p$ are coprime and $(a, p) = (1)$ is the unit ideal. Therefore $ua+vp = 1$ for some $u, v \in \mathbb{Z}$ with $ua \equiv 1 \bmod p$, so $[u][a] = [1]$ in $\mathbb{Z}/p\mathbb{Z}$ and $[a]$ is invertible. To justify the second claim, note that in any field of characteristic $p$ the subring generated by 1 is isomorphic to $\mathbb{Z}/p\mathbb{Z} = \mathbb{F}_p$, and this subring is clearly unique (any other must also contain 1), hence canonical. □

[1] The recent breakthrough in computing discrete logarithms in finite fields of small characteristic in quasi-polynomial time [1] has greatly diminished the enthusiasm for using such fields in cryptographic applications.

The most common way to represent $\mathbb{F}_p$ for computational purposes is to pick a set of unique coset representatives for $\mathbb{Z}/p\mathbb{Z}$, such as the integers in the interval $[0, p-1]$.

**Definition 3.4.** For each prime power $q = p^n$ we define $\mathbb{F}_q = \mathbb{F}_{p^n}$ to be the field extension of $\mathbb{F}_p$ generated by adjoining all roots of $x^q - x$ to $\mathbb{F}_p$ (the splitting field of $x^q - x$ over $\mathbb{F}_p$). Equivalently, $\mathbb{F}_q := \overline{\mathbb{F}_p}^{\sigma_q}$ is the subfield of the algebraic closure of $\mathbb{F}_p$ fixed by the $q$-power Frobenius automorphism $\sigma_q \colon x \mapsto x^q$.

**Remark 3.5.** We note that this definition makes sense for $n = 1$, with $q = p$: the polynomial $x^p - x$ splits completely over $\mathbb{F}_p$, and $\mathbb{F}_p$ is the subfield of $\overline{\mathbb{F}_p}$ fixed by $\sigma_p$.

**Theorem 3.6.** *Let $q = p^n$ be a prime power. The field $\mathbb{F}_q$ has cardinality $q$ and every field of cardinality $q$ is (non-canonically) isomorphic to $\mathbb{F}_q$.*

*Proof.* The map $x \mapsto x^q = x^{p^n}$ is an automorphism $\sigma_q$ of $\mathbb{F}_q$, since in characteristic $p$ we have

$$(a+b)^{p^n} = a^{p^n} + b^{p^n} \qquad \text{and} \qquad (ab)^{p^n} = a^{p^n} b^{p^n},$$

where the first identity follows from the binomial theorem: $\binom{p^n}{r} \equiv 0 \bmod p$ for $0 < r < p^n$. Let $k := \mathbb{F}_q^{\sigma_q}$ be the subfield of $\mathbb{F}_q$ fixed by $\sigma_q$. We have $\mathbb{F}_p \subseteq k$, since

$$(1 + \cdots + 1)^q = 1^q + \cdots + 1^q = 1 + \cdots + 1,$$

and it follows that $\mathbb{F}_q \subseteq k$, since $\sigma_q$ fixes $\mathbb{F}_p$ and every root of $x^q - x$, and therefore $k = \mathbb{F}_q$. The polynomial $x^q - x$ has no roots in common with its derivative $(x^q - x)' = qx^{q-1} - 1 = -1$, so it has $q$ distinct roots, which are precisely the elements of $\mathbb{F}_q$ (they lie in $\mathbb{F}_q$ by definition, and every element of $\mathbb{F}_q = \mathbb{F}_q^{\sigma_q}$ is fixed by $\sigma_q$ and therefore a root of $x^q - x$).

Now let $k$ be a field of cardinality $q = p^n$. Then $k$ must have characteristic $p$, since the set $\{1, 1+1, \ldots\}$ is a subgroup of the additive group of $k$, so the characteristic divides $\#k = p^n$, and in a finite ring with no zero divisors the characteristic must be prime. By Theorem 3.3, the field $k$ contains $\mathbb{F}_p$. The order of each $\alpha \in k^\times$ divides $\#k^\times = q - 1$; thus $\alpha^{q-1} = 1$ for all $\alpha \in k^\times$, so every $\alpha \in k$, including $\alpha = 0$, is a root of $x^q - x$. It follows that $k$ is isomorphic to a subfield of $\mathbb{F}_q$, and $\#k = \#\mathbb{F}_q$, so $k \simeq \mathbb{F}_q$ (this isomorphism is not canonical because when $q$ is not prime there are many ways to embed $k$ in $\mathbb{F}_q$). □

**Remark 3.7.** Now that we know all finite fields of cardinality $q$ are isomorphic, we will feel free to refer to any and all of them as *the* finite field $\mathbb{F}_q$, with the understanding that there are many ways to represent $\mathbb{F}_q$ and we will need to choose one of them.

**Theorem 3.8.** *The finite field $\mathbb{F}_{p^m}$ is a subfield of $\mathbb{F}_{p^n}$ if and only if $m$ divides $n$.*

*Proof.* If $\mathbb{F}_{p^m} \subseteq \mathbb{F}_{p^n}$ then $\mathbb{F}_{p^n}$ is an $\mathbb{F}_{p^m}$-vector space of (integral) dimension $n/m$, so $m|n$. If $m|n$ then $p^n - p^m = (p^m - 1)(p^{n-m} + p^{n-2m} + \cdots + p^{2m} + p^m)$ is divisible by $p^m - 1$ and

$$x^{p^n} - x = (x^{p^m} - x)(1 + x^{p^m - 1} + x^{2(p^m-1)} + \cdots + x^{p^n - p^m})$$

is divisible by $x^{p^m} - x$. Thus every root of $x^{p^m} - x$ is also a root of $x^{p^n} - x$, so $\mathbb{F}_{p^m} \subseteq \mathbb{F}_{p^n}$. □

**Theorem 3.9.** *For any irreducible $f \in \mathbb{F}_p[x]$ of degree $n > 0$ we have $\mathbb{F}_p[x]/(f) \simeq \mathbb{F}_{p^n}$.*

*Proof.* The ring $k := \mathbb{F}_p[x]/(f)$ is an $\mathbb{F}_p$-vector space with basis $1, x, \ldots, x^{n-1}$ and therefore has dimension $n$ and cardinality $p^n$. The ring $\mathbb{F}_p[x]$ is a principal ideal domain and $f$ is irreducible and not a unit, so $(f)$ is a maximal ideal and $\mathbb{F}_p[x]/(f)$ is a field with $p^n$ elements, hence isomorphic to $\mathbb{F}_{p^n}$ by Theorem 3.6. □

Theorem 3.9 allows us to explicitly represent $\mathbb{F}_{p^n}$ as $\mathbb{F}_p[x]/(f)$ using any irreducible polynomial $f \in \mathbb{F}_p[x]$ of degree $n$, and it does not matter which $f$ we pick; by Theorem 3.6 we always get the same field (up to isomorphism). We also note the following corollary.

**Corollary 3.10.** *Every irreducible $f \in \mathbb{F}_p[x]$ of degree $n$ splits completely in $\mathbb{F}_{p^n}$.*

*Proof.* We have $\mathbb{F}_p[x]/(f) \simeq \mathbb{F}_{p^n}$, so every root of $f$ is a root of $x^{p^n} - x$ and lies in $\mathbb{F}_{p^n}$. □

**Remark 3.11.** This corollary implies that $x^{p^n} - x$ is the product over the divisors $d|n$ of all monic irreducible polynomials of degree $d$ in $\mathbb{F}_p[x]$. This can be used to derive explicit formulas for the number of irreducible polynomials of degree $d$ in $\mathbb{F}_p[x]$ using Möbius inversion. It also implies that, even though we defined $\mathbb{F}_{p^n}$ as the splitting field of $x^{p^n} - x$, it is also the splitting field of every irreducible polynomial of degree $n$.

**Theorem 3.12.** *Every finite subgroup of the multiplicative group of a field is cyclic.*

*Proof.* Let $k$ be a field, let $G$ be a subgroup of $k^\times$ of order $n$, and let $m$ be the exponent of $G$ (the least common multiple of the orders of its elements), which necessarily divides $n$. Every element of $G$ is a root of $x^m - 1$, which has at most $m$ roots, so $m = n$. Every finite abelian group contains an element of order equal to its exponent, so $G$ contains an element of order $m = n = \#G$ and is therefore cyclic. □

**Corollary 3.13.** *The multiplicative group of a finite field is cyclic.*

If $\alpha$ is a generator for the multiplicative group $\mathbb{F}_q^\times$, then it generates $\mathbb{F}_q$ as an extension of $\mathbb{F}_p$, that is, $\mathbb{F}_q = \mathbb{F}_p(\alpha)$, and we have $\mathbb{F}_q \simeq \mathbb{F}_p[x]/(f)$, where $f \in \mathbb{F}_p[x]$ is the minimal polynomial of $\alpha$, but the converse need not hold. This motivates the following definition.

**Definition 3.14.** A monic irreducible polynomial $f \in \mathbb{F}_p[x]$ whose roots generate the multiplicative group of the finite field $\mathbb{F}_p[x]/(f)$ is called a *primitive polynomial*.

**Theorem 3.15.** *For every prime $p$ and positive integer $n$ there exist primitive polynomials of degree $n$ in $\mathbb{F}_p[x]$. Indeed, the number of such polynomials is $\phi(p^n - 1)/n$.*

Here $\phi(m)$ is the Euler function that counts the generators of a cyclic group of order $m$, equivalently, the number of integers in $[1, m-1]$ that are relatively prime to $m$.

*Proof.* Let $\alpha$ be a generator for $\mathbb{F}_{p^n}^\times$ with minimal polynomial $f_\alpha \in \mathbb{F}_p[x]$; then $f_\alpha$ is primitive. There are $\phi(p^n - 1)$ possible choices for $\alpha$. Conversely, if $f \in \mathbb{F}_p[x]$ is a primitive polynomial of degree $n$ then each of its $n$ roots is a generator for $\mathbb{F}_q^\times$. We thus have a surjective $n$-to-1 map $\alpha \to f_\alpha$ from the set of generators of $\mathbb{F}_{p^n}^\times$ to the set of primitive polynomials over $\mathbb{F}_p$ of degree $n$; the theorem follows. □

The preceding theorem implies that there are plenty of irreducible (and even primitive) polynomials $f \in \mathbb{F}_p[x]$ that we can use to represent $\mathbb{F}_q = \mathbb{F}_p[x]/(f)$ when $q$ is not prime. The choice of the polynomial $f$ has some impact on the cost of reducing polynomials in $\mathbb{F}_p[x]$ modulo $f$; ideally we would like $f$ to have as few nonzero coefficients as possible. We can choose $f$ to be a binomial whenever its degree divides $p - 1$, and we can usually (although not always) choose $f$ to be a trinomial; see [8]. Finite fields in cryptographic standards are often specified using an $f \in \mathbb{F}_p[x]$ that makes reduction modulo $f$ particularly efficient.

For mathematical purposes it is more useful to fix a universal choice of primitive polynomials once and for all; this simplifies the task of migrating data from one computer algebra

system to another, as well as the restoration of archived data. One way to do this is to take the lexicographically minimal primitive polynomial $f_{p,n} \in \mathbb{F}_p[x]$ of each degree $n$, where we represent monic $f_{p,n}(x) = \sum a_i x^{n-i}$ as a sequence of integers $(1, a_1, \ldots, a_n)$ with $0 \le a_i < p$.

There are two downsides to this simple-minded approach. First (and most significantly), we would like to be able to easily embed $\mathbb{F}_{p^m}$ in $\mathbb{F}_{p^n}$ when $m|n$, which means that if $\alpha$ is a root of $f_{p,n}(x)$ then we really want $\alpha^{(p^n-1)/(p^m-1)}$ to be a root of $f_{p,m}(x)$, including when $m = 1$. Secondly (and less significantly), we would like the root $r$ of $f_{p,1} = x - r$ to be the least primitive root modulo $p$, which will not be the case if we use the lexicographic ordering defined above, but will be the case if we tweak our sign convention and take $(1, a_1, \ldots, a_n)$ to represent the polynomial $x^n - a_1 x^{n-1} + \cdots + (-1)^n a_n$ with terms $(-1)^i a_i x^{n-i}$. This leads to the following recursive definition due to Richard Parker (named in honor of John Conway).

**Definition 3.16.** Order polynomials $f(x) = x^n - a_1 x^{n-1} + \cdots + (-1)^n a_n \in (\mathbb{Z}/p\mathbb{Z})[x]$ with $0 \le a_i < p$ according to the lexicographic order on integer sequences $(1, a_1, \ldots, a_n)$. For each prime $p$ and $n > 0$ the *Conway polynomial* $f_{p,n}(x)$ is defined by:

- For $n = 1$, let $f_{p,1}(x) := x - r$, where $r$ is the least positive integer generating $(\mathbb{Z}/p\mathbb{Z})^\times$;
- For $n > 1$, let $f_{p,n}(x)$ be the least primitive polynomial of degree $n$ such that for every $0 < m < n$ dividing $n$ and every root $\alpha$ of $f_{p,m}(x)$ we have $f_{p,n}(\alpha^{(p^n-1)/(p^m-1)}) = 0$.

That $f_{p,n}(x)$ exists is a straightforward proof by induction that we leave as an exercise.

Conway polynomials are now used by most computer algebra systems, including GAP, Magma, Macaulay2, and SageMath. One downside to their recursive definition is that it is quite time consuming to compute any particular Conway polynomial on demand; instead, each of these computer algebra systems includes a list of precomputed Conway polynomials. The key point is that, even in a post-apocalyptic scenario where all these tables are lost, they can all be readily reconstructed from the succinct definition above.

Having fixed a representation for $\mathbb{F}_q$, every finite field operation can ultimately be reduced to integer arithmetic: elements of $\mathbb{F}_p$ are represented as integers in $[0, p-1]$, and elements of $\mathbb{F}_q = \mathbb{F}_p[x]/(f)$ are represented as polynomials of degree less than $\deg f$ whose coefficients are integers in $[0, p-1]$.

Before leaving our review of finite fields, we want to recall one other key fact about finite fields, which is that every finite field $\mathbb{F}_q$ is a Galois extension of its prime field $\mathbb{F}_p$, and the Galois group $\mathrm{Gal}(\mathbb{F}_q/\mathbb{F}_p)$ is cyclic of order $[\mathbb{F}_q : \mathbb{F}_p]$, generated by the $p$-power Frobenius automorphism $\sigma_p \colon x \mapsto x^p$. This follows immediately from our definition of $\mathbb{F}_q$ as the splitting field of $x^q - x$ over $\mathbb{F}_p$, provided we know that $\mathbb{F}_q/\mathbb{F}_p$ is Galois. This follows from the fact that $x^q - x$ is a *separable polynomial*.

**Definition 3.17.** Let $k$ be a field and let $f = \sum f_i x^i \in k[x]$ be a polynomial. We say that $f$ is *separable* if any of the following equivalent conditions hold:

- $f$ has $\deg f$ distinct roots in any algebraic closure $\bar{k}$ of $k$;
- $f$ is squarefree over every extension of $k$;
- $\gcd(f, f') = 1$ in $k[x]$, where $f'_i := \sum i f_i x^{i-1}$ denotes the formal derivative of $f$.

A polynomial that is not separable is said to be *inseparable*.

**Remark 3.18.** We will typically write $\gcd(f, f') = 1$ to indicate that $\gcd(f, f')$ is a unit. The gcd of two elements in a ring is defined only up to units (if $a$ divides $b$ and $c$ then so

does $ua$ for any unit $u$), and in a principal ideal domain it is standard to take $\gcd(a,b)$ to be a unique representative of the ideal $(a,b)$. For the ring $\mathbb{Z}$ there is a unique positive representative (the only units are $\pm 1$), and in the ring $k[x]$ there is a unique monic representative (units are elements of $k^\times$).

**Remark 3.19.** Some older textbooks (notably including Bourbaki) define a polynomial to be separable if its irreducible factors are separable, which would make polynomials like $(x-1)^2$ separable, but for us this is not a separable polynomial. On the other hand, it is clear that if a polynomial $f$ is separable under our definition, then all its irreducible factors are separable, since if $f$ has distinct roots in $\bar{k}$ then so does every divisor of $f$.

**Lemma 3.20.** *An irreducible polynomial $f \in k[x]$ is inseparable if and only if $f' = 0$.*

*Proof.* Let $f \in k[x]$ be irreducible. Then $f$ is nonzero and not a unit. If $f' = 0$ then $\gcd(f, f') = f$ is not a unit and $f$ is inseparable. If $f$ is inseparable then $g = \gcd(f, f')$ is a nonconstant divisor of $f$ and $f'$, and if $f'$ is nonzero then $\deg g \le \deg f' < \deg f$, which is impossible because $f$ is irreducible. □

The polynomial $x^q - x$ is separable because

$$\gcd(x^q - x, (x^q - x)') = \gcd(x^q - x, -1) = 1,$$

and it follows that its splitting field over $\mathbb{F}_p$ is a Galois extension of $\mathbb{F}_p$ (this is the basic tenet of Galois theory: splitting fields of separable polynomials $f \in k[x]$ are finite Galois extensions of $k$, and every finite Galois extension of $k$ is the splitting field of some separable polynomial $f \in k[x]$). An important consequence of this fact is that finite fields are perfect.

**Definition 3.21.** A field $k$ is *perfect* if every irreducible polynomial in $k[x]$ is separable, equivalently, has a nonzero derivative.

Fields of characteristic zero are always perfect, since there is no way for the derivative of a nonconstant polynomial to be zero in such fields. Fields of positive characteristic $p$ need not be perfect (we will see many examples of this in later lectures), but finite fields are.

**Theorem 3.22.** *Finite fields are perfect.*

*Proof.* Let $f = \sum_i f_i x^i$ be an irreducible polynomial in $\mathbb{F}_q[x]$, and let

$$g \coloneqq \prod_{\sigma \in \mathrm{Gal}(\mathbb{F}_q/\mathbb{F}_p)} f^\sigma,$$

where $f^\sigma := \sum_i \sigma(f_i) x^i$. Let $\mathbb{F}_p$ be the prime field of $\mathbb{F}_q$. We have $g \in \mathbb{F}_p[x]$, since it is invariant under the action of $\mathrm{Gal}(\mathbb{F}_q/\mathbb{F}_p)$, and it is irreducible in $\mathbb{F}_p[x]$ since any non-trivial factor of $g$ in $\mathbb{F}_p[x]$ would also be a non-trivial factor in $\mathbb{F}_q[x]$, none of which are invariant under the action of $\mathrm{Gal}(\mathbb{F}_q/\mathbb{F}_p)$ (note that each $f^\sigma$ is irreducible in $\mathbb{F}_q[x]$). Now $f | g$, so if $g$ is separable then $f$ is separable, which means that if $\mathbb{F}_p$ is perfect then so is $\mathbb{F}_q$.

Let $g = \sum_i g_i x^i$. If $g$ is inseparable then $g' = \sum_i i g_i x^{i-1} = 0$, which implies that $g_i = 0$ for $i$ not divisible by $p$, meaning that $g = h(x^p)$ for some $h \in \mathbb{F}_p[x]$. But this cannot be the case because $h(x^p) = h(x)^p$ is not irreducible. □

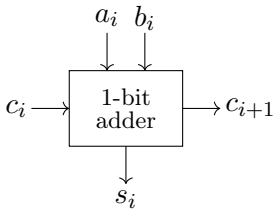
### 3.2 Integer addition

Every nonnegative integer $a$ has a unique *binary representation* $a = \sum_{i=0}^{n-1} a_i 2^i$ with $a_i \in \{0,1\}$ and $a_{n-1} \neq 0$. The binary digits $a_i$ are called *bits*, and we say that $a$ is an *$n$-bit integer*; we can represent negative integers by including an additional sign bit.

To add two integers in their binary representations we apply the "schoolbook" method, adding bits and carrying as needed. For example, we can compute 43+37=80 in binary as

$$\begin{array}{r} {\scriptstyle 101111} \\ 101011 \\ +100101 \\ \hline 1010000 \end{array}$$

The carry bits are shown in red. To see how this might implemented in a computer, consider a 1-bit adder that takes two bits $a_i$ and $b_i$ to be added, along with a carry bit $c_i$.




$$c_{i+1} = (a_i \wedge b_i) \vee (c_i \wedge a_i) \vee (c_i \wedge b_i)$$

$$s_i = a_i \otimes b_i \otimes c_i$$

The symbols $\wedge$, $\vee$, and $\otimes$ denote the boolean functions AND, OR, and XOR (exclusive-or) respectively, which we may regard as primitive components of a boolean circuit. By chaining $n+1$ of these 1-bit adders together, we can add two $n$-bit numbers using $7n+7 = O(n)$ boolean operations on individual bits.

**Remark 3.23.** Chaining adders is known as *ripple* addition and is no longer commonly used, since it forces a sequential computation. In practice more sophisticated methods such as *carry-lookahead* are used to facilitate parallelism. This allows most modern microprocessors to add two 64 (or even 128) bit integers in a single clock cycle, and with the SIMD (Single Instruction Multiple Data) instruction sets available on newer AMD and Intel processors, one may be able to perform four (or even eight) 64 bit additions in a single clock cycle.

We could instead represent the same integer $a$ as a sequence of words rather than bits. For example, write $a = \sum_{i=0}^{k-1} a_i 2^{64i}$, where $k = \left\lceil \frac{n}{64} \right\rceil$. We may then add two integers using a sequence of $O(k)$, equivalently, $O(n)$, operations on 64-bit words. Each word operation is ultimately implemented as a boolean circuit that involves operations on individual bits, but since the word-size is fixed, the number of bit operations required to implement any particular word operation is a constant. So the number of bit operations is again $O(n)$, and if we ignore constant factors it does not matter whether we count bit or word operations.

Subtraction is analogous to addition (now we need to borrow rather than carry), and has the same complexity, so we will not distinguish these operations when analyzing the complexity of algorithms. With addition and subtraction of integers, we have everything we need to perform addition and subtraction in a finite field. To add two elements of $\mathbb{F}_p \simeq \mathbb{Z}/p\mathbb{Z}$ that are uniquely represented as integers in the interval $[0, p-1]$ we simply add the integers and check whether the result is greater than or equal to $p$; if so we subtract $p$ to obtain a value in $[0, p-1]$. Similarly, after subtracting two integers we add $p$ if the result is negative.

The total work involved is still $O(n)$ bit operations, where $n = \lg p$ is the number of bits needed to represent a finite field element.

To add or subtract two elements of $\mathbb{F}_q \simeq (\mathbb{Z}/p\mathbb{Z})[x]/(f)$ we simply add or subtract the corresponding coefficients of the polynomials, for a total cost of $O(d \lg p)$ bit operations, where $d = \deg f$, which is again $O(n)$ bit operations, if we put $n = \lg q = d \lg p$.

**Theorem 3.24.** *The time to add or subtract two elements of $\mathbb{F}_q$ in our standard representation is $O(n)$, where $n = \lg q$ is the size of a finite field element.*

**Remark 3.25.** We will discuss the problem of reducing an integer modulo a prime $p$ using fast Euclidean division in the next lecture. But this operation is not needed to reduce the sum or difference of two integers in $[0, p-1]$ to a representative in $[0, p-1]$; it is faster (both in theory and practice) to simply subtract or add $p$ as required (at most once).

## 3.3 A quick refresher on asymptotic notation

Let $f$ and $g$ be two real-valued functions whose domains include the positive integers. The *big-O* notation "$f(n) = O(g(n))$" is shorthand for the statement:

*There exist constants $c$ and $N$ such that for all $n \geq N$ we have $|f(n)| \leq c|g(n)|$.*

This is equivalent to

$$\limsup_{n\to\infty} \frac{|f(n)|}{|g(n)|} < \infty.$$

**Warning 3.26.** "$f(n) = O(g(n))$" is an abuse of notation; in words we would say $f(n)$ is $O(g(n))$, where the word "is" does not imply equality (e.g., "Aristotle is a man"), and it is generally better to write this way. Symbolically, it would make more sense to write $f(n) \in O(g(n))$, regarding $O(g(n))$ as a set of functions. Some do, but the notation $f(n) = O(g(n))$ is far more common and we will occasionally use it in this course, with one caveat: we will never write a big-$O$ expression to the left of the equal sign. It may be true that $f(n) = O(n \log n)$ implies $f(n) = O(n^2)$, but we avoid writing $O(n \log n) = O(n^2)$ because $O(n^2) \neq O(n \log n)$.

We also have *big-*$\Omega$ notation "$f(n) = \Omega(g(n))$", which means $g(n) = O(f(n))$,[2] as well as *little-o* notation "$f(n) = o(g(n))$," which is shorthand for

$$\lim_{n\to\infty} \frac{|f(n)|}{|g(n)|} = 0.$$

An alternative notation that is sometimes used is $f \ll g$, but depending on the author this may mean $f(n) = o(g(n))$ or $f(n) = O(g(n))$ (computer scientists tend to mean the former, while number theorists usually mean the latter, so we will avoid this notation). There is also a little-omega notation, but the symbol $\omega$ already has so many uses in number theory that we will not burden it further (we can always use little-$o$ notation instead). The notation $f(n) = \Theta(g(n))$ means that $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$ both hold.

It is easy to see that the complexity of integer addition is $\Theta(n)$, since we have shown it is $O(n)$ and it is clearly $\Omega(n)$ because it takes this long to output $n$ bits (in a Turing machine model one can show that for most inputs the machine will have to write to $\Omega(n)$ cells on the Turing tape, no matter what algorithm it uses).

[2] The $\Omega$-notation originally defined by Hardy and Littlewood had a slightly weaker definition, but modern usage generally follows our convention, which is due to Knuth.

**Warning 3.27.** Don't confuse a big-$O$ statement with a big-$\Theta$ statement; the former implies only an upper bound. If Alice has an algorithm that is $O(2^n)$ this does not mean that Alice's algorithm requires exponential time, and it does not mean that Bob's $O(n^2)$ algorithm is better; Alice's algorithm could be $O(n)$ for all we know. But if Alice's algorithm is $\Omega(2^n)$ then we would definitely prefer to use Bob's algorithm for all sufficiently large $n$.

Big-$O$ notation can also be used for multi-variable functions: "$f(m,n) = O(g(m,n))$" is shorthand for the statement:

*There exist constants $c$ and $N$ such that for all $m, n \geq N$ we have $|f(m,n)| \leq c|g(m,n)|$.*

This statement is weaker than it appears. For example, it says nothing about the relationship between $f(m,n)$ and $g(m,n)$ if we fix one of the variables. However, in virtually all of the examples we will see it will actually be true that if we regard $f(m,n) = f_m(n)$ and $g(m,n) = g_m(n)$ as functions of $n$ with a fixed parameter $m$, we have $f_m(n) = O(g_m(n))$, and similarly, $f_n(m) = O(g_n(m))$. In this situation one says that $f(m,n) = O(g(m,n))$ holds *uniformly* (in $m$ and $n$).

So far we have spoken only of *time complexity*, but *space complexity* plays a crucial role in many algorithms that we will see in later lectures. Space complexity measures the amount of memory an algorithm requires; this can never be greater than its time complexity (it takes time to use space), but it may be smaller. When we speak of "the complexity" of an algorithm, we should really consider both time and space. An upper bound on the time complexity is also an upper bound on the space complexity but it is often possible (and desirable) to obtain a better bound for the space complexity.

For more information on asymptotic notation and algorithmic complexity, see [5].

**Warning 3.28.** In this class, unless explicitly stated otherwise, our asymptotic bounds always count bit operations (as opposed to finite field operations, or integer operations). When comparing complexity bounds found in the literature, one must be sure to understand exactly what is being counted. For example, a complexity bound that counts operations in finite fields may need to be converted to a bit complexity to get an accurate comparison, and this conversion is going to depend on exactly which finite field operations are being used and how the finite fields are represented. A lack of care in this regard has led to more than one erroneous claim in the literature.

## 3.4 Integer multiplication

We now consider the problem of integer multiplication. Unlike addition, this is (still) an open problem; it is widely believed that $O(n \log n)$ is the best possible, and this has even been proved conditionally under various conjectures, but it is not known unconditionally, and it is only very recently that $O(n \log n)$ was established as an upper bound.

Because we do not know the exact complexity of integer multiplication, it is common practice to use the notation $\mathsf{M}(n)$ to denote the time to multiply two $n$-bit integers; this allows us to state bounds for algorithms that depend on the complexity of integer multiplication in a way that does not depend on whatever the current state of the art is. This convention has proved useful over the past two decades during which upper bounds on $\mathsf{M}(n)$ have improved at least four times.

### 3.4.1 Schoolbook method

Let us compute $37 \times 43 = 1591$ with the "schoolbook" method, using a binary representation.

$$\begin{array}{r} 101011 \\ \times\ 100101 \\ \hline 101011 \\ 101011 \\ +101011 \\ \hline 11000110111 \end{array}$$

Multiplying individual bits is easy (just use an AND gate), but we need to do $n^2$ bit multiplications, followed by $n$ additions of $n$-bit numbers (suitably shifted). The complexity of this algorithm is thus $\Theta(n^2)$. This gives us the upper bound $\mathsf{M}(n) = O(n^2)$. The only lower bound known is the trivial one, $\mathsf{M}(n) = \Omega(n)$, so one might hope to do better than $O(n^2)$, and indeed we can.

### 3.4.2 Karatsuba's algorithm

Before presenting Karatsuba's algorithm, it is worth making a few remarks regarding its origin. In the first half of the twentieth century it was widely believed that $\mathsf{M}(n) = \Omega(n^2)$; indeed, no less a mathematician than Kolmogorov formally stated this conjecture in a 1956 meeting of the Moscow Mathematical Society [16, §5]. This conjecture was one of the topics at a 1960 seminar led by Kolmogorov, with Karatsuba in attendance. Within the first week of the seminar, Karatsuba was able to disprove the conjecture. Looking back on the event, Karatsuba writes [16, §6]

> *After the next seminar I told Kolmogorov about the new algorithm and about the disproof of the $n^2$ conjecture. Kolmogorov was very agitated because this contradicted his very plausible conjecture. At the next meeting of the seminar, Kolmogorov himself told the participants about my method and at this point the seminar was terminated.*

Karatsuba's algorithm is based on a divide-and-conquer approach. Rather than representing $n$-bit integers using $n$ digits in base 2, we may instead write them in base $2^{n/2}$ and may compute their product as follows

$$\begin{aligned} a &= a_0 + 2^{n/2} a_1, \\ b &= b_0 + 2^{n/2} b_1, \\ ab &= a_0 b_0 + 2^{n/2}(a_1 b_0 + b_1 a_0) + 2^n a_1 b_1, \end{aligned}$$

As written, this reduces an $n$-bit multiplication to four multiplications of $(n/2)$-bit integers and three additions of $O(n)$-bit integers (multiplying an intermediate result by a power of 2 can be achieved by simply writing the binary output "further to the left" and is effectively free). However, as observed by Karatsuba one can use the identity

$$a_0 b_1 + b_0 a_1 = (a_0 + a_1)(b_0 + b_1) - a_0 b_0 - a_1 b_1$$

to compute $a_0 b_1 + b_0 a_1$ using just one multiplication in addition to computing the products $a_0 b_0$ and $a_1 b_1$. By reusing the common subexpressions $a_0 b_0$ and $a_1 b_1$, we can compute $ab$

using three multiplications and six additions (we count subtractions as additions). We can use the same idea to recursively compute the three products $a_0b_0$, $a_1b_1$, and $(a_0+a_1)(b_0+b_1)$; this recursive approach yields Karatsuba's algorithm.

If we let $T(n)$ denote the running time of this algorithm, we have

$$\begin{aligned} T(n) &= 3T(n/2) + O(n) \\ &= O(n^{\lg 3}) \end{aligned}$$

It follows that $\mathsf{M}(n) = O(n^{\lg 3})$, where $\lg 3 := \log_2 3 \approx 1.59$.[3]

### 3.4.3 The Fast Fourier Transform (FFT)

The fast Fourier transform is widely regarded as one of the top ten algorithms of the twentieth century [6, 10], and has applications throughout applied mathematics. Here we focus on the discrete Fourier transform (DFT), and its application to multiplying integers and polynomials, following the presentation in [9, §8]. It is actually more natural to address the problem of polynomial multiplication first.

Let $R$ be a commutative ring containing a primitive $n$th root of unity $\omega$, by which we mean that $\omega^n = 1$ and $\omega^i - \omega^j$ is not a zero divisor for $0 \le i < j < n$ (when $R$ is a field this coincides with the usual definition). We shall identify the set of polynomials in $R[x]$ of degree less than $n$ with the set of all $n$-tuples with entries in $R$. Thus we represent the polynomial $f(x) = \sum_{i=0}^{n-1} f_i x^i$ by its coefficient vector $(f_0, \dots, f_{n-1}) \in R^n$ and may speak of the polynomial $f \in R[x]$ and the vector $f \in R^n$ interchangeably.

The discrete Fourier transform $\mathrm{DFT}_\omega : R^n \to R^n$ is the $R$-linear map

$$(f_0, \dots, f_{n-1}) \xrightarrow{\mathrm{DFT}_\omega} (f(\omega^0), \dots, f(\omega^{n-1})).$$

You should think of this map as a conversion between two types of polynomial representations: we take a polynomial of degree less than $n$ represented by $n$ coefficients (its *coefficient-representation*) and convert it to a representation that gives its values at $n$ known points (its *point-representation*).

One can use Lagrange interpolation to recover the coefficient representation from the point representation, but our decision to use values $\omega^0, \dots, \omega^{n-1}$ that are $n$th roots of unity allows us to do this more efficiently. If we define the Vandermonde matrix

$$V_\omega := \begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \cdots & \omega^{2n-2} \\ 1 & \omega^3 & \omega^6 & \cdots & \omega^{3n-3} \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ 1 & \omega^{n-1} & \omega^{2n-2} & \cdots & \omega^{(n-1)^2} \end{pmatrix},$$

then $\mathrm{DFT}_\omega(f) = V_\omega f^{\mathrm{tr}}$. Our assumption that none of the differences $\omega^i - \omega^j$ is a zero divisor in $R$ ensures that the matrix $V_\omega$ is invertible, and its inverse is simply $\frac{1}{n} V_{\omega^{-1}}$. It follows that

$$\mathrm{DFT}_\omega^{-1} = \frac{1}{n} \mathrm{DFT}_{\omega^{-1}}.$$

[3] In general we shall use $\lg n$ to denote $\log_2 n$.

Thus if we have an algorithm to compute $\mathrm{DFT}_\omega$ we can use it to compute $\mathrm{DFT}_\omega^{-1}$: just replace $\omega$ by $\omega^{-1} = \omega^{n-1}$ and multiply the result by $\frac{1}{n}$.

We now define the *cyclic convolution* $f * g$ of two polynomials $f, g \in R^n$:

$$f * g = fg \bmod (x^n - 1).$$

Reducing the product on the right modulo $x^n - 1$ ensures that $f * g$ is a polynomial of degree less than $n$, thus we may regard the cyclic convolution as a map $R^n \times R^n \to R^n$. If $h = f * g$, then $h_i = \sum f_j g_k$, where the sum is over $j + k \equiv i \bmod n$. If $f$ and $g$ both have degree less than $n/2$, then $f * g = fg$; thus the cyclic convolution of $f$ and $g$ can be used to compute their product, provided that we make $n$ big enough.

We also define the *pointwise product* $f \cdot g$ of two vectors in $f, g \in R^n$:

$$f \cdot g = (f_0 g_0,\ f_1 g_1,\ \ldots,\ f_{n-1} g_{n-1}).$$

We have now defined three operations on vectors in $R^n$: the binary operations of convolution and pointwise product, and the unary operation $\mathrm{DFT}_\omega$. The following theorem relates these three operations and is the key to the fast Fourier transform.

**Theorem 3.29.** $\mathrm{DFT}_\omega(f * g) = \mathrm{DFT}_\omega(f) \cdot \mathrm{DFT}_\omega(g)$.

*Proof.* Since $f * g = fg \bmod (x^n - 1)$, we have

$$f * g = fg + q \cdot (x^n - 1)$$

for some polynomial $q \in R[x]$. For every integer $i$ from 0 to $n - 1$ we then have

$$\begin{aligned}(f * g)(\omega^i) &= f(\omega^i) g(\omega^i) + q(\omega^i)(\omega^{in} - 1) \\ &= f(\omega^i) g(\omega^i),\end{aligned}$$

where we have used $(\omega^{in} - 1) = 0$, since $\omega$ is an $n$th root of unity. □

The theorem implies that if $f$ and $g$ are polynomials of degree less then $n/2$ then

$$fg = f * g = \mathrm{DFT}_\omega^{-1}\big(\mathrm{DFT}_\omega(f) \cdot \mathrm{DFT}_\omega(g)\big). \tag{1}$$

This identity allows us to multiply polynomials using the discrete Fourier transform. In order to put this into practice, we need an efficient way to compute $\mathrm{DFT}_\omega$. This is achieved by the following recursive algorithm.

**Algorithm**: Fast Fourier Transform (FFT)
**Input**: A positive integer $n = 2^k$, a vector $f \in R^n$, and the vector $(\omega^0, \ldots, \omega^{n-1}) \in R^n$.
**Output**: $\mathrm{DFT}_\omega(f) \in R^n$.

1. If $n = 1$ then return $(f_0)$ and terminate.
2. Write the polynomial $f(x)$ in the form $f(x) = g(x) + x^{\frac{n}{2}} h(x)$, where $g, h \in R^{\frac{n}{2}}$.
3. Compute the vectors $r = g + h$ and $s = (g - h) \cdot (\omega^0, \ldots, \omega^{\frac{n}{2}-1})$ in $R^{\frac{n}{2}}$.
4. Recursively compute $\mathrm{DFT}_{\omega^2}(r)$ and $\mathrm{DFT}_{\omega^2}(s)$ using $(\omega^0, \omega^2, \ldots, \omega^{n-2})$.
5. Return the vector $(r(\omega^0), s(\omega^0), r(\omega^2), s(\omega^2), \ldots, r(\omega^{n-2}), s(\omega^{n-2}))$

Let $T(n)$ be the number of operations in $R$ used by the FFT algorithm. Then

$$\begin{aligned} T(n) &= 2T(n/2) + O(n) \\ &= O(n \log n). \end{aligned}$$

This shows that the FFT is fast (justifying its name); let us now prove that it is correct.

**Theorem 3.30.** *The FFT algorithm outputs* $\mathrm{DFT}_\omega(f)$.

*Proof.* We must verify that the $k$th entry of the output vector is $f(\omega^k)$, for $0 \le k < n$. For even $k = 2i$ we have:

$$\begin{aligned} f(\omega^{2i}) &= g(\omega^{2i}) + (\omega^{2i})^{n/2} h(\omega^{2i}) \\ &= g(\omega^{2i}) + h(\omega^{2i}) \\ &= r(\omega^{2i}). \end{aligned}$$

For odd $k = 2i + 1$ we have:

$$\begin{aligned} f(\omega^{2i+1}) &= \sum_{0 \le j < n/2} f_j \omega^{(2i+1)j} + \sum_{0 \le j < n/2} f_{n/2+j} \omega^{(2i+1)(n/2+j)} \\ &= \sum_{0 \le j < n/2} g_j \omega^{2ij} \omega^j + \sum_{0 \le j < n/2} h_j \omega^{2ij} \omega^{in} \omega^{n/2} \omega^j \\ &= \sum_{0 \le j < n/2} (g_j - h_j) \omega^j \omega^{2ij} \\ &= \sum_{0 \le j < n/2} s_j \omega^{2ij} \\ &= s(\omega^{2i}), \end{aligned}$$

where we have used the fact that $\omega^{n/2} = -1$. □

**Corollary 3.31.** *Let $R$ be a commutative ring containing a primitive $n$th root of unity, with $n = 2^k$, and assume $2 \in R^\times$. We can multiply two polynomials in $R[x]$ of degree less than $n/2$ using $O(n \log n)$ operations in $R$.*

*Proof.* From (1) we have

$$fg = \mathrm{DFT}_\omega^{-1}(\mathrm{DFT}_\omega(f) \cdot \mathrm{DFT}_\omega(g)) = \frac{1}{n} \mathrm{DFT}_{\omega^{-1}}(\mathrm{DFT}_\omega(f) \cdot \mathrm{DFT}_\omega(g))$$

and we note that $n = 2^k \in R^\times$ is invertible. We can compute $\omega^0, \dots, \omega^{n-1}$ using $O(n)$ multiplications in $R$ (this also gives us $(\omega^{-1})^0, \dots, (\omega^{-1})^{n-1}$). Computing $\mathrm{DFT}_\omega$ and $\mathrm{DFT}_{\omega^{-1}}$ via the FFT algorithm uses $O(n \log n)$ operations in $R$, computing the pointwise product of $\mathrm{DFT}_\omega(f)$ and $\mathrm{DFT}_\omega(g)$ uses $O(n)$ operations in $R$, and computing $1/n$ and multiplying a polynomial of degree less than $n$ by this scalar uses $O(n)$ operations in $R$. □

What about rings that do not contain an $n$th root of unity? By extending $R$ to a new ring $R' := R[\omega]/(\omega^n - 1)$ we can obtain a formal $n$th root of unity $\omega$, and one can then generalize Corollary 3.31 to multiply polynomials in any ring $R$ in which 2 is invertible using $O(n \log n \log\log n)$ operations in $R$; see [9, §8.3] for details.

The need for 2 to be invertible can be overcome by considering a 3-adic version of the FFT algorithm that works in rings $R$ in which 3 is invertible. For rings in which neither 2 nor 3 is invertible we instead compute $2^k fg$ and $3^m fg$ (just leave out the multiplication by $1/n$ at the end). Once we know both $2^k fg$ and $3^m fg$ we can recover the coefficients of $fg$ by using the Euclidean algorithm to compute $u, v \in \mathbb{Z}$ such that $u2^k + v3^m = 1$ and applying $u2^k fg + v3^m fg = fg$.

### 3.5 Integer multiplication

To any positive integer $a = \sum_{i=0}^{n-1} a_i 2^i$ we may associate the polynomial $f_a(x) = \sum_{i=0}^{n} a_i x^i \in \mathbb{Z}[x]$, with $a_i \in \{0, 1\}$, so that $a = f_a(2)$. We can then multiply positive integers $a$ and $b$ via

$$ab = f_{ab}(2) = (f_a f_b)(2).$$

Note that the polynomials $f_a(x)f_b(x)$ and $f_{ab}(x)$ may differ (the former may have coefficients greater than 1), but they take the same value at $x = 2$; in practice one typically uses base $2^{64}$ rather than base 2 (the $a_i$ and $b_i$ are then integers in $[0, 2^{64} - 1]$).

Applying the generalization of Corollary 3.31 noted above to the ring $\mathbb{Z}$, Schönhage and Strassen [19] obtain an algorithm to multiply two $n$-bit integers in time $O(n \log n \log\log n)$, which gives us a new upper bound

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

In 2007 Fürer [7] showed that this bound can been improved to

$$\mathsf{M}(n) = O\left(n \log n \, 2^{O(\log^* n)}\right)$$

where $\log^* n$ denotes the iterated logarithm, which counts how many times the log function must be applied to $n$ before the result is less than or equal to 1. In 2016 Harvey, van der Hoeven and Lecerf [15] proved the sharper bound

$$\mathsf{M}(n) = O\left(n \log n \, 8^{\log^* n}\right),$$

and in 2018 Harvey and van der Hoeven [12] further improved this to

$$\mathsf{M}(n) = O\left(n \log n \, 4^{\log^* n}\right).$$

In 2019 Harvey and van der Hoeven [14] announced the spectacular and long awaited result

$$\mathsf{M}(n) = O\left(n \log n\right),$$

which as far as asymptotics go, is almost certainly the final word on the matter.

The algorithms that enabled these improvements and even the original Schönhage–Strassen algorithm are fairly intricate and purely of theoretical interest: in practice one uses the "three primes" algorithm sketched below, which for integers with $n \leq 2^{62}$ bits has a "practical complexity" of $O(n \log n)$; this statement is mathematically meaningless but gives a rough indication of how the running time increases as $n$ varies in this bounded range. But it is a great relief and convenience to know that the theoretical complexity now matches the practical complexity, and that we can dispense with the "$\log\log n$" term you will find in almost any literature that mentions the complexity of integer multiplication prior to 2020.

### 3.5.1 Three primes FFT for integer multiplication

As noted above, the details of the Schoönhage and Strassen algorithm and its subsequent improvements are rather involved. There is a much simpler approach that is used in practice to multiply integers less than $2^{2^{62}}$; this includes integers that would require 500 petabytes (500,000 terabytes) to write down and is more than enough for any practical application that is likely to arise in the near future. Let us briefly outline this approach.

Write the positive integers $a, b < 2^{2^{62}}$ that we wish to multiply in base $2^{64}$ as $a = \sum a_i 2^{64i}$ and $b = \sum b_i 2^{64i}$, with $0 \le a_i, b_i < 2^{64}$, and define the polynomials $f_a = \sum a_i x^i \in \mathbb{Z}[x]$ and $f_b = \sum b_i x^i \in \mathbb{Z}[x]$ as above. Our goal is to compute $f_{ab}(2^{64}) = (f_a f_b)(2^{64})$, and we note that the polynomial $f_a f_b \in \mathbb{Z}[x]$ has less than $2^{62}/64 = 2^{56}$ coefficients, each of which is bounded by $2^{56}2^{64}2^{64} < 2^{184}$.

Rather than working over a single ring $R$ we will use three finite fields $\mathbb{F}_p$ of odd characteristic, where $p$ is one of the primes

$$p_1 := 71 \cdot 2^{57} + 1, \qquad p_2 := 75 \cdot 2^{57} + 1, \qquad p_3 := 95 \cdot 2^{57} + 1.$$

Note that if $p$ is any of the primes $p_1, p_2, p_3$, then $\mathbb{F}_p^\times$ is a cyclic group whose order $p-1$ is divisible by $2^{57}$, which implies that $\mathbb{F}_p$ contains a primitive $2^{57}$th root of unity $\omega$; indeed, for $p = p_1, p_2, p_3$ we can use $\omega = \omega_1, \omega_2, \omega_3$, respectively, where $\omega_1 = 287, \omega_2 = 149, \omega_3 = 55$.

We can thus use the FFT Algorithm above with $R = \mathbb{F}_p$ to compute $f_a f_b \bmod p$ for each of the primes $p \in \{p_1, p_2, p_3\}$. This gives us the values of the coefficients of $f_a f_b \in \mathbb{Z}[x]$ modulo three primes whose product $p_1 p_2 p_3 > 2^{189}$ is more than large enough to uniquely recover the coefficients via the Chinese Remainder Theorem (CRT); the time to recover the integer coefficients of $f_a f_b$ from their values modulo $p_1, p_2, p_3$ is negligible compared to the time to apply the FFT algorithm over these three fields. If $a$ and $b$ are significantly smaller, say $a, b \le 2^{2^{44}}$, a "one prime" approach suffices.

## 3.6 Kronecker substitution

We now note an important converse to the idea of using polynomial multiplication to multiply integers: we can use integer multiplication to multiply polynomials. This is quite useful in practice, as it allows us take advantage of very fast implementations of FFT–based integer multiplication that are now widely available. If $f$ is a polynomial in $\mathbb{F}_p[x]$, we can lift $f$ to $\hat{f} \in \mathbb{Z}[x]$ by representing its coefficients as integers in $[0, p-1]$. If we then consider the integer $\hat{f}(2^m)$, where $m = \lceil 2 \lg p + \lg(\deg f + 1) \rceil$, the coefficients of $\hat{f}$ will appear in the binary representation of $\hat{f}(2^m)$ separated by blocks of $m - \lceil \lg p \rceil$ zeros. If $g$ is a polynomial of similar degree, we can easily recover the coefficients of $\hat{h} = \hat{f}\hat{g} \in \mathbb{Z}[x]$ in the integer product $N = \hat{f}(2^m)\hat{g}(2^m)$; we then reduce the coefficients of $\hat{h}$ modulo $p$ to get $h = fg$. The key is to make $m$ large enough so that the $k$th block of $m$ binary digits in $N$ contains the binary representation of the $k$th coefficient of $\hat{h}$.

This technique is known as *Kronecker substitution*, and it allows us to multiply two polynomials of degree $d$ in $\mathbb{F}_p[x]$ in time $O(\mathsf{M}(d(n + \log d)))$, where $n = \log p$. Typically we have $\log d = O(n)$, in which case this simplifies to $O(\mathsf{M}(dn))$ In particular, we can use Kronecker substitution to multiply elements of $\mathbb{F}_q \simeq \mathbb{F}_p[x]/(f)$ in time $O(\mathsf{M}(n))$, where $n = \log q$, provided $\log \deg f = O(\log p)$.

**Remark 3.32.** When $\log d = O(n)$, if we make the standard assumption that $\mathsf{M}(n)$ grows super-linearly then using Kronecker substitution is strictly faster (by more than any constant

factor) than a layered approach that uses the FFT to multiply polynomials and then recursively uses the FFT for the coefficient multiplications; this is because $\mathsf{M}(dn) = o(\mathsf{M}(d)\,\mathsf{M}(n))$.

### 3.7 Euclidean division

Given integers $a, b > 0$, we wish to compute the unique integers $q, r \geq 0$ for which

$$a = bq + r \qquad (0 \leq r < b).$$

We have $q = \lfloor a/b \rfloor$ and $r = a \bmod b$. It is enough to compute $q$, since we can then compute $r = a - bq$. To compute $q$, we determine a sufficiently precise approximation $c \approx 1/b$ and obtain $q$ by computing $ca$ and rounding down to the nearest integer.

We recall Newton's method for finding the root of a real-valued function $f(x)$. We start with an initial approximation $x_0$, and at each step, we refine the approximation $x_i$ by computing the $x$-coordinate $x_{i+1}$ of the point where the tangent line through $(x_i, f(x_i))$ intersects the $x$-axis, via

$$x_{i+1} := x_i - \frac{f(x_i)}{f'(x_i)}.$$

To compute $c \approx 1/b$, we apply this to $f(x) = 1/x - b$, using the Newton iteration

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} = x_i - \frac{\frac{1}{x_i} - b}{-\frac{1}{x_i^2}} = 2x_i - bx_i^2.$$

As an example, let us approximate $1/b = 1/123456789$. For the sake of illustration we work in base 10, but in an actual implementation would use base 2, or base $2^w$, where $w$ is the word size.

$$\begin{aligned}
x_0 &= 1 \times 10^{-8} \\
x_1 &= 2(1 \times 10^{-8}) - (1.2 \times 10^{8})(1 \times 10^{-8})^2 \\
&= 0.80 \times 10^{-8} \\
x_2 &= 2(0.80 \times 10^{-8}) - (1.234 \times 10^{8})(0.80 \times 10^{-8})^2 \\
&= 0.8102 \times 10^{-8} \\
x_3 &= 2(0.8102 \times 10^{-8}) - (1.2345678 \times 10^{8})(0.8102 \times 10^{-8})^2 \\
&= 0.81000002 \times 10^{-8}.
\end{aligned}$$

Note that we double the precision we are using at each step, and each $x_i$ is correct up to an error in its last decimal place. The value $x_3$ suffices to correctly compute $\lfloor a/b \rfloor$ for $a \leq 10^{15}$.

To analyze the complexity of this approach, let us assume that $b$ has $n$ bits and $a$ has at most $2n$ bits; this is precisely the situation we will encounter when we wish to reduce the product of two integers in $[0, p-1]$ modulo $p$. During the Newton iteration to compute $c \approx 1/b$, the size of the integers involved doubles with each step, and the cost of the arithmetic operations grows at least linearly. The total cost is thus at most twice the cost of the last step, which is $\mathsf{M}(n) + O(n)$; note that all operations can be performed using integers by shifting the operands appropriately. Thus we can compute $c \approx 1/b$ in time $2\,\mathsf{M}(n) + O(n)$. We can then compute $ca \approx a/b$, round to the nearest integer, and compute $r = a - bq$ using at most $4\,\mathsf{M}(n) + O(n)$ bit operations.

With a slightly more sophisticated version of this approach it is possible to compute $r$ in time $3\,\mathsf{M}(n) + O(n)$, and if we expect to repeatedly perform Euclidean division with the

same denominator we can further reduce this to $2\,\mathsf{M}(n) + O(n)$ by precomputing $c \approx 1/b$. This approach is exploited by two widely used approaches to modular arithmetic, *Barrett reduction* (see [4, Alg. 10.17]) and *Montgomery reduction* (see Problem Set 1). Regardless of the approach taken, we obtain the following bound for multiplication in $\mathbb{F}_p$ using our standard representation as integers in $[0, p-1]$.

**Theorem 3.33.** *The time to multiply two elements of $\mathbb{F}_p$ is $O(\mathsf{M}(n))$, where $n = \lg p$.*

There is an analogous version of this algorithm above for polynomials that uses the exact same Newton iteration $x_{i+1} = 2x_i - bx_i^2$, where $b$ and the $x_i$ are now polynomials. Rather than working with Laurent polynomials (the polynomial version of approximating a rational number with a truncated decimal expansion), it is simpler to reverse the polynomials and work modulo a sufficiently large power of $x$, doubling the power of $x$ with each Newton iteration. More precisely, we have the following algorithm, which combines Algorithms 9.3 and 9.5 from [9]. For any polynomial $f(x)$ we write $\operatorname{rev} f$ for the polynomial $x^{\deg f} f(\frac{1}{x})$; this simply reverses the coefficients of $f$.

**Algorithm 3.34** (Fast Euclidean division of polynomials)**.** Given $a, b \in \mathbb{F}_p[x]$ with $b$ monic, compute $q, r \in \mathbb{F}_p[x]$ such that $a = qb + r$ with $\deg r < \deg b$ as follows:

1. If $\deg a < \deg b$ then return $q = 0$ and $r = a$.
2. Let $m = \deg a - \deg b$ and $k = \lceil \lg m + 1 \rceil$.
3. Let $f = \operatorname{rev}(b)$ (reverse the coefficients of $b$).
4. Compute $g_0 = 1, g_i = (2g_{i-1} - fg_{i-1}^2) \bmod x^{2^i}$ for $i$ from 1 to $k$.
   (this yields $fg_k \equiv 1 \bmod x^{m+1}$).
5. Set $s = \operatorname{rev}(a) g_k \bmod x^{m+1}$ (now $\operatorname{rev}(b)s \equiv \operatorname{rev}(a) \bmod x^{m+1}$).
6. Return $q = x^{m - \deg s} \operatorname{rev}(s)$ and $r = a - bq$.

As in the integer case, the work is dominated by the last iteration in step 4, which involves multiplying polynomials in $\mathbb{F}_p[x]$. To multiply elements of $\mathbb{F}_q \simeq \mathbb{F}_p[x]/(f)$ represented as polynomials of degree less than $d = \deg f$, we compute the product $a$ in $\mathbb{F}[x]$ and then reduce modulo $b = f$, and the degree of the polynomials involved are all $O(d)$. With Kronecker substitution, we can reduce these polynomial multiplications to integer multiplications, and obtain the following result.

**Theorem 3.35.** *Let $q = p^e$ be a prime power, and assume that $\log e = O(\log p)$. The time to multiply two elements of $\mathbb{F}_q$ is $O(\mathsf{M}(n)) = O(n \log n)$, where $n = \lg q$.*

**Remark 3.36.** The constraints on the relative growth rates of $p$ and $e$ in the theorem above are present only so that we can conveniently use Kronecker substitution to bound the complexity in terms of the bound $\mathsf{M}(n)$ for multiplying integers. In fact we fully expect that the $O(n \log n)$ bound implied by Theorem 3.35 holds uniformly. This is known under a widely believed conjecture about the least prime in arithmetic progressions, namely that the least prime in every arithmetic progression $m\mathbb{Z} + a$ with $a$ coprime to $m$ is bounded by $O(m^{1+\epsilon})$ for any $\epsilon > 0$ (in fact any $\epsilon < 2^{-1162}$ would do); see [13].

Before leaving the topic of Euclidean division, we should also mention the standard "schoolbook" algorithm of long division. The classical algorithm works with decimal digits (base 10), but for the sake of simplicity let us work in base 2; in practice one works in base $2^w$ for some fixed $w$.

**Algorithm 3.37** (Long division)**.** Given positive integers $a = \sum_{i=0}^{m} a_i 2^i$ and $b = \sum_{i=0}^{n} b_i 2^i$, compute $q, r \in \mathbb{Z}$ such that $a = qb + r$ with $0 \le r < b$ as follows:

1. If $b > a$ return $q = 0$ and $r = a$, and if $b = 1$ return $q = a$ and $r = 0$.
2. Set $q \leftarrow 0$, $r \leftarrow 0$, and $k \leftarrow m$.
3. While $k \ge 0$ and $r < b$ set $q \leftarrow 2q$, $r \leftarrow 2r + a_k$, and $k \leftarrow k - 1$.
4. If $r < b$ then return $q$ and $r$.
5. Set $q \leftarrow q + 1$, $r \leftarrow r - b$, and return to Step 3.

The net effect of all the executions of Step 3 is is to add $a$ to $qb+r$ using double-and-add bitwise addition. The quantity $qb + r$ is initially set to 0 in Step 2 and is unchanged by Step 5, so when the algorithm terminates in Step 4 we have $a = qb + r$ and $0 \le r < b$ as desired. If we are only interested in the remainder $r$ we can omit all operations involving $q$.

For the complexity analysis we can assume that multiplication by 2 is achieved by bit-shifting and costs $O(1)$ (consider a multi-tape Turing machine, or a bit-addressable RAM). Step 2 costs $O(1)$, the total cost of Step 3 over all iterations is $O(nm)$, as is the total cost of Step 5 (note that $q$ is a multiple of 2 at the start of Step 5, so computing $q \leftarrow q + 1$ is achieved by setting the least significant bit). This yields the following result.

**Theorem 3.38.** *The long division algorithm uses $O(mn)$ bit operations to perform Euclidean division of an $m$-bit integer by an $n$-bit integer.*

**Remark 3.39.** For $m = O(n)$ the $O(n^2)$ complexity of long division is worse than the $O(\mathsf{M}(n))$ cost of Euclidean division using Newton iteration. But when $m$ is much larger than $n$, say $n = O(\log m)$ or $n = O(1)$, long division is a better choice. In particular, for any fixed prime $p$ (so $O(1)$ bits) we can reduce $n$-bit integers modulo $p$ in linear time.

## 3.8 Extended Euclidean algorithm

We recall the Euclidean algorithm for computing the greatest common divisor of positive integers $a$ and $b$. For $a > b$ we repeatedly apply

$$\gcd(a, b) = \gcd(b, a \bmod b),$$

where we take $a \bmod b$ to be the unique integer $r \in [0, b-1]$ congruent to $a$ modulo $b$.

To compute the multiplicative inverse of an integer modulo a prime, we use the extended Euclidean algorithm, which expresses $\gcd(a, b)$ as a linear combination

$$\gcd(a, b) = as + bt,$$

with $|s| \le b/\gcd(a, b)$ and $|t| \le a/\gcd(a, b)$. If $a$ is prime, we obtain $as + bt = 1$, and $t$ is the inverse of $b$ modulo $a$. To compute the integers $s$ and $t$ we use the following algorithm. First, let

$$R_1 = \begin{bmatrix} a \\ b \end{bmatrix}, \quad S_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \quad T_1 = \begin{bmatrix} 0 \\ 1 \end{bmatrix},$$

and note that $R_1 = aS_1 + bT_1$. We then compute

$$Q_i = \begin{bmatrix} 0 & 1 \\ 1 & -q_i \end{bmatrix}, \quad R_{i+1} = Q_i R_i, \quad S_{i+1} = Q_i S_i, \quad T_{i+1} = Q_i T_i,$$

where $q_i$ is the quotient $\lfloor R_{i,1}/R_{i,2} \rfloor$ obtained via Euclidean division. Note that applying the linear transformation $Q_i$ to both sides of $R_i = aS_i + bT_i$ ensures $R_{i+1} = aS_{i+1} + bT_{i+1}$. The algorithm terminates in the $k$th step where $R_{k,2}$ becomes zero, at which point we have

$$R_k = \begin{bmatrix} d \\ 0 \end{bmatrix}, \quad S_k = \begin{bmatrix} s \\ \pm b \end{bmatrix}, \quad T_k = \begin{bmatrix} t \\ \mp a \end{bmatrix},$$

with $\gcd(a, b) = d = sa + tb$. As an example, with $a = 1009$ and $b = 789$ we have

| $r$ | $q$ | $s$ | $t$ |
|---|---|---|---|
| 1009 | | 1 | 0 |
| 789 | 1 | 0 | 1 |
| 220 | 3 | 1 | −1 |
| 129 | 1 | −3 | 4 |
| 91 | 1 | 4 | −5 |
| 38 | 2 | −7 | 9 |
| 15 | 2 | 18 | −23 |
| 8 | 1 | −43 | 55 |
| 7 | 1 | 61 | −78 |
| 1 | 7 | −104 | 133 |
| 0 | | 789 | −1009 |

From the second-to-last line with $s = -104$ and $t = 133$ we see that

$$1 = -104 \cdot 1009 + 133 \cdot 789,$$

and therefore 133 is the inverse of 789 modulo 1009 (and $-104 \equiv 685$ is the inverse of 1009 modulo 789).

It is clear that $r$ is reduced by a factor of at least 2 every two steps, thus the total number of iterations is $O(n)$, and each step involves Euclidean division, whose cost is bounded by $O(\mathsf{M}(n))$. This yields a complexity of $O(n\,\mathsf{M}(n))$, but a more careful analysis shows that it is actually $O(n^2)$, even if schoolbook multiplication is used (the key point is that the total size of all the $q_i$ is $O(n)$ bits).

This can be further improved using the *fast Euclidean algorithm*, which uses a divide-and-conquer approach to compute the product $Q = Q_{k-1} \cdots Q_1$ by splitting the product in half and recursively computing each half using what is known as a *half-gcd* algorithm. One can then compute $R_k = QR_1$, $S_k = QS_1$, and $T_k = QT_1$. The details are somewhat involved (care must be taken when determining how to split the product in a way that balances the work evenly), but this yields a recursive running time of

$$T(n) = 2T(n/2) + O(\mathsf{M}(n)) = O(\mathsf{M}(n) \log n);$$

see [9, §11] for details.

**Theorem 3.40.** *Let $p$ be a prime. The time to invert an element of $\mathbb{F}_p^\times$ is $O(\mathsf{M}(n) \log n)$, where $n = \lg p$.*

The extended Euclidean algorithm works in any Euclidean ring, that is, a ring with a norm function that allows us to use Euclidean division to write $a = qb + r$ with $r$ of norm strictly less than $b$ (for any nonzero $b$). This includes polynomial rings, in which the norm of

a polynomial is simply its degree. Thus we can compute the inverse of a polynomial modulo another polynomial, provided the two polynomials are relatively prime.

One issue that arises when working in Euclidean rings other than $\mathbb{Z}$ is that there may be units (invertible elements) other than $\pm 1$, and the gcd is only defined up to a unit. In the case of the polynomial ring $\mathbb{F}_p[x]$, every element of $\mathbb{F}_p^\times$ is a unit, and with the fast Euclidean algorithm in $\mathbb{F}_p[x]$ one typically normalizes the intermediate results by making the polynomials monic at each step; this involves computing the inverse of the leading coefficient in $\mathbb{F}_p$. If $\mathbb{F}_q = \mathbb{F}_p[x]/(f)$ with $\deg f = d$, one can then bound the time to compute an inverse in $\mathbb{F}_q$ by $O(\mathsf{M}(d)\log d)$, operations in $\mathbb{F}_p$, of which $O(d)$ are inversions; see [9, Thm. 11.10(i)]. This gives a bit complexity of

$$O(\mathsf{M}(d)\,\mathsf{M}(\log p)\log d + d\,\mathsf{M}(\log p)\log\log p),$$

but with Kronecker substitution we can sharpen this to

$$O(\mathsf{M}(d(\log p + \log d))\log d + d\,\mathsf{M}(\log p)\log\log p).$$

We will typically assume that either $\log d = O(\log p)$ (large characteristic) or $\log p = O(1)$ (small characteristic); in both cases we can simplify this bound to $O(\mathsf{M}(n)\log n)$, where $n = \lg q = d\lg p$ is the number of bits in $q$, the same result we obtained for the case where $q = p$ is prime.

**Theorem 3.41.** *Let $q = p^e$ be a prime power and assume $\log e = O(\log p)$. The time to invert an element of $\mathbb{F}_q^\times$ is $O(\mathsf{M}(n)\log n) = O(n\log^2 n)$, where $n = \lg q$.*

**Remark 3.42.** As with Theorem 3.35, the assumption $\log e = O(\log p)$ can be removed if one assumes the least prime in every arithmetic progression $m\mathbb{Z} + a$ with $a$ coprime to $m$ is bounded by $O(m^{1+\epsilon})$ for any $\epsilon > 0$.

## 3.9 Exponentiation (scalar multiplication)

Let $a$ be a positive integer. In a multiplicative group, the computation

$$g^a = \underbrace{gg\cdots g}_{a}$$

is known as *exponentiation*. In an additive group, this is equivalent to

$$ag = \underbrace{g + g + \cdots + g}_{a},$$

and is called *scalar multiplication*. The same algorithms are used in both cases, and most of these algorithms were first developed in a multiplicative setting (the multiplicative group of a finite field) and are called exponentiation algorithms. It is actually more convenient to describe the algorithms using additive notation (fewer superscripts), so we will do so.

The oldest and most commonly used exponentiation algorithm is the "double-and-add" method, also known as left-to-right binary exponentiation. Given an element $P$ of an additive group and a positive integer $a$ with binary representation $a = \sum 2^i a_i$, we compute the scalar multiple $Q = aP$ as follows:

```
def DoubleAndAdd (P,a):
    a=a.digits(2); n=len(a)       # represent a in binary using n bits
    Q=P;                          # start 1 bit below the high bit
    for i in range(n-2,-1,-1):    # for i from n-2 down to 0
        Q += Q                    # double
        if a[i]==1: Q += P        # add
    return Q
```

Alternatively, we may use the "add-and-double" method, also known as right-to-left binary exponentiation.

```
def AddAndDouble (P,a):
    a=a.digits(2); n=len(a)       # represent a in binary using n bits
    Q=0; R=P;                     # start with the low bit
    for i in range(n-1):
        if a[i]==1: Q += R        # add
        R += R                    # double
    Q += R                                                  # last add
    return Q
```

The number of group operations required is effectively the same for both algorithms. If we ignore the first addition in the `add_and_double` algorithm (which could be replaced by an assignment, since initially $Q = 0$), both algorithms use precisely

$$n + \mathrm{wt}(a) - 2 \leq 2n - 2 = O(n)$$

group operations, where $\mathrm{wt}(a) = \#\{a_i : a_i = 1\}$ is the *Hamming weight* of $a$, the number of 1's in its binary representation. Up to the constant factor 2, this is asymptotically optimal, and it implies that exponentiation in a finite field $\mathbb{F}_q$ has complexity $O(n\,\mathsf{M}(n))$ with $n = \lg q$; this assumes the exponent is less than $q$, but note that we can always reduce the exponent modulo $q - 1$, the order of the cyclic group $\mathbb{F}_q^\times$. Provided the bit-size of the exponent is $O(n^2)$, the $O(\mathsf{M}(n^2))$ time to reduce the exponent modulo $q - 1$ will be majorized by the $O(n\,\mathsf{M}(n))$ time to perform the exponentiation.

Notwithstanding the fact that the simple double-and-add algorithm is within a factor of 2 of the best possible, researchers have gone to great lengths to eliminate this factor of 2, and to take advantage of situations where either the base or the exponent is fixed, and there are a wide variety of optimizations that are used in practice; see [4, Ch. 9] and [11]. Here we give just one example, windowed exponentiation, which is able to reduce the constant factor from 2 to an essentially optimal $1 + o(1)$.

### 3.9.1 Fixed-window exponentiation

Let the positive integer $s$ be a *window size* and write $a$ as

$$a = \sum a_i 2^{si}, \quad (0 \leq a_i < 2^s).$$

This is equivalent to writing $a$ in base $2^s$. With fixed-window exponentiation, one first precomputes multiples $dP$ for each of the "digits" $d \in [0, 2^s - 1]$ that may appear in the base-$2^s$ expansion of $a$. One then uses a left-to-right approach as in the double-and-add algorithm, except now we double $s$ times and add the appropriate multiple $a_i P$.

```
def FixedWindow (P,a,s):
    a=a.digits(2^s); n=len(a)                    # write a in base 2^s
```

```
R = [0*P,P]
for i in range(2,2^s): R.append(R[-1]+P)    # precompute digits
Q = R[a[-1]]                                # copy the top digit
for i in range(n-2,-1,-1):
    for j in range(0,s): Q += Q             # double s times
    Q += R[a[i]]                            # add the next digit
return Q
```

In the algorithm above we precompute multiples of $P$ for every possible digit that might occur. As an optimization one could examine the base-$2^s$ representation of $a$ and only precompute the multiples of $P$ that are actually needed.

Let $n$ be the number of bits in $a$ and let $m = \lceil n/s \rceil$ be the number of base-$2^s$ digits $a_i$. The precomputation step uses $2^s - 2$ additions (we get $0P$ and $1P$ for free), there are $m - 1$ additions of multiples of $P$ corresponding to digits $a_i$ (when $a_i = 0$ these cost nothing), and there are a total of $(m-1)s$ doublings. This yields an upper bound of

$$2^s - 2 + m - 1 + (m-1)s \approx 2^s + n/s + n$$

group operations. If we choose $s = \lg n - \lg\lg n$, we obtain the bound

$$n/\lg n + n/(\lg n - \lg\lg n) + n = n + O(n/\log n),$$

which is $(1 + o(1))n$ group operations.

### 3.9.2 Sliding-window exponentiation

The sliding-window algorithm modifies the fixed-window algorithm by "sliding" over blocks of 0s in the binary representation of $a$. There is still a window size $s$, but $a$ is no longer treated as an integer written in a fixed base $2^s$. Instead, the algorithm scans the bits of the exponent from left to right, assembling "digits" of at most $s$ bits with both high and low bits set: with a sliding window of size 3 the bit-string 110011010101100 could be broken up as 11|00|11|0|101|0|11|00 with 4 nonzero digits, whereas a fixed window approach would use 110|011|010|101|100 with 5 nonzero digits. This improves the fixed-window approach in two ways: first, it is only necessarily to precompute odd digits, and second, depending on the pattern of bits in $a$, sliding over the zeros may reduce the number of digits used, as in the example above. In any case, the sliding-window approach is never worse than the fixed-window approach, and for $s > 2$ it is always better.

**Example 3.43.** Let $a = 26284$ corresponding to the bit-string 110011010101100 above. To compute $aP$ using a sliding window approach with $s = 3$ one would first compute $2P, 3P, 5P$ using 3 additions and then

$$aP = 2^2 \cdot (2^3 \cdot (2^4 \cdot (2^4 \cdot (3P) + 3P)) + 5P) + 3P)$$

using 3 additions and 13 doublings, for a total cost of 19 group operations. A fixed window approach with $s = 3$ would instead compute $2P, 3P, 4P, 5P, 6P$ using 5 additions and

$$aP = 2^3 \cdot (2^3 \cdot (2^3 \cdot (2^3 \cdot 6P + 3P) + 2P) + 5P) + 4P$$

using 4 additions and 12 doublings for a total cost of 21 group operations. Note that in both cases we avoided computing $7P$ since it was not needed.

## 3.10 Root-finding in finite fields

Let $f(x)$ be a polynomial in $\mathbb{F}_q[x]$ of degree $d$. We wish to find a solution to $f(x) = 0$ that lies in $\mathbb{F}_q$. As an important special case, this will allow us to compute square roots using $f(x) = x^2 - a$, and, more generally, $r$th roots.[4]

The algorithm we give here was originally proposed by Berlekamp for prime fields [2], and then refined and extended by Rabin [18], whose presentation we follow here. The algorithm is probabilistic, and is one of the best examples of how randomness can be exploited in a number-theoretic setting. As we will see, it is quite efficient, with an expected running time that is quasi-quadratic in the size of the input. By contrast, no deterministic polynomial-time algorithm for root-finding is known, not even for computing square roots.[5]

### 3.10.1 Randomized algorithms

Probabilistic algorithms are typically classified as one of two types: *Monte Carlo* or *Las Vegas*. Monte Carlo algorithms are randomized algorithms whose output may be incorrect, depending on random choices that are made, but whose running time is bounded by a function of its input size, independent of any random choices. The probability of error is required to be less than $1/2-\epsilon$, for some $\epsilon > 0$, and can be made arbitrarily small by running the algorithm repeatedly and using the output that occurs most often. In contrast, a Las Vegas algorithm always produces a correct output, but its running time may depend on random choices; we do require that its expected running time is finite. As a trivial example, consider an algorithm to compute $a + b$ that first flips a coin repeatedly until it gets a head and then computes $a + b$ and outputs the result. The running time of this algorithm may be arbitrarily long, even when computing $1 + 1 = 2$, but its *expected* running time is $O(n)$, where $n$ is the size of the inputs.

Las Vegas algorithms are generally preferred, particularly in mathematical applications. Note that any Monte Carlo algorithm whose output can be verified can always be converted to a Las Vegas algorithm (just run the algorithm repeatedly until you get an answer that is verifiably correct). The root-finding algorithm we present here is a Las Vegas algorithm.

### 3.10.2 Using GCDs to find roots

Recall from the previous lecture that we defined the finite field $\mathbb{F}_q$ to be the splitting field of $x^q - x$ over its prime field $\mathbb{F}_p$; this definition also applies when $q = p$ is prime (since $x^p - x$ splits completely in $\mathbb{F}_p$), and in every case, the elements of $\mathbb{F}_q$ are precisely the roots of $x^q - x$. The roots of $f$ that lie in $\mathbb{F}_q$ are the roots it has in common with the polynomial $x^q - x$. We thus have

$$g(x) := \gcd(f(x), x^q - x) = \prod_i (x - \alpha_i),$$

where the $\alpha_i$ range over all the distinct roots of $f$ that lie in $\mathbb{F}_q$. If $f$ has no roots in $\mathbb{F}_q$ then $g$ will have degree 0 (in which case $g = 1$). We have thus reduced our problem to finding a root of $g$, where $g$ has distinct roots that are known to lie in $\mathbb{F}_q$.

[4] An entirely different approach to computing $r$th roots using discrete logarithms is explored in Problem Set 2. It has better constant factors when the $r$-power torsion subgroup of $\mathbb{F}_q^*$ is small (which is usually the case), but is asymptotically slower than the algorithm presented here in the worst case.

[5] Deterministic polynomial-time bounds for root-finding can be proved in various special cases, including the computation of square-roots, if one assumes a generalization of the Riemann hypothesis.

In order to compute $g = \gcd(f, x^q - x)$ efficiently, we generally do *not* compute $x^q - x$ and then take the gcd with $f$; this would take time exponential in $n = \log q$.[6] Instead, we compute $x^q \bmod f$ by exponentiating the polynomial $x$ to the $q$th power in the ring $\mathbb{F}_q[x]/(f)$, whose elements are uniquely represented by polynomials of degree less than $d = \deg f$. Each multiplication in this ring involves the computation of a product in $\mathbb{F}_q[x]$ followed by a reduction modulo $f$; note that we do not assume $\mathbb{F}_q[x]/(f)$ is a field (indeed for $\deg f > 1$, if $f$ has a root in $\mathbb{F}_q$ then $\mathbb{F}_q[x]/(f)$ is definitely not a field). This reduction is achieved using Euclidean division, and can be accomplished using two polynomial multiplications once an approximation to $1/f$ has been precomputed, see §3.7, and is within a constant factor of the time to multiply two polynomials of degree $d$ in any case. The total cost of each multiplication in $\mathbb{F}_q[x]/(f)$ is thus $O(\mathsf{M}(d(n+\log d)))$, assuming that we use Kronecker substitution to multiply polynomials. The time to compute $x^q \bmod f$ using any of the exponentiation algorithms described in §3.9 is then $O(n\,\mathsf{M}(d(n+\log d)))$.

Once we have computed $x^q \bmod f$, we subtract $x$ and compute $g = \gcd(f, x^q - x)$. Using the fast Euclidean algorithm, this takes $O(\mathsf{M}(d(n+\log d))\log d)$ time. Thus the total time to compute $g$ is $O(\mathsf{M}(d(n+\log d))(n+\log d))$; and in the typical case where $\log d = O(n)$ (e.g. $d$ is fixed and only $n$ is growing) this simplifies to $O(n\,\mathsf{M}(dn))$.

So far we have not used randomness; we have a deterministic algorithm to compute the polynomial $g = (x - r_1)\cdots(x - r_k)$, where $r_1, \ldots, r_k$ are the distinct $\mathbb{F}_q$-rational roots of $f$. We can thus determine the number of distinct roots $f$ has (this is just the degree of $g$), and in particular, whether it has any roots, deterministically, but knowledge of $g$ does not imply knowledge of the roots $r_1, \ldots, r_k$ when $k > 1$; for example, if $f(x) = x^2 - a$ has a nonzero square root $r \in \mathbb{F}_q$, then $g(x) = (x - r)(x + r) = f(x)$ tells us nothing beyond the fact that $f(x)$ has a root.

## 3.11 Randomized GCD splitting

Having computed $g$, we seek to factor it into two polynomials of lower degree by again applying a gcd, with the goal of eventually obtaining a linear factor, which will yield a root.

Assuming that $q$ is odd (which we do), we may factor the polynomial $x^q - x$ as

$$x^q - x = x(x^s - 1)(x^s + 1),$$

where $s = (q-1)/2$. Ignoring the root 0 (which we can easily check separately), this factorization splits $\mathbb{F}_q^\times$ precisely in half: the roots of $x^s - 1$ are the elements of $\mathbb{F}_q^\times$ that are squares in $\mathbb{F}_q^\times$, and the roots of $x^s + 1$ are the elements of $\mathbb{F}_q^\times$ that are not. Recall that $\mathbb{F}_q^\times$ is a cyclic group of order $q - 1$, and for $\alpha \in \mathbb{F}_q^\times$ we have $\alpha^s = \pm 1$ with $\alpha^s = 1$ precisely when $\alpha$ is a square in $\mathbb{F}_q^\times$. If we compute

$$h(x) = \gcd(g(x), x^s - 1),$$

we obtain a divisor of $g$ whose roots are precisely the roots of $g$ that are squares in $\mathbb{F}_q^\times$. If we suppose that the roots of $g$ are as likely to be squares as not, we should expect the degree of $h$ to be approximately half the degree of $g$. And so long as the degree of $h$ is strictly between 0 and $\deg g$, one of $h$ or $g/h$ is a polynomial of degree at most half the degree of $g$, whose roots are all roots of our original polynomial $f$.

[6] The exception is when $d > q$, but in this case computing $\gcd(f(x), x^q - x)$ takes $O(\mathsf{M}(d(n+\log d)\log d))$ time, which turns out to be the same bound that we get for computing $x^q \bmod f(x)$ in any case.

To make further progress, and to obtain an algorithm that is guaranteed to work no matter how the roots of $g$ are distributed in $\mathbb{F}_q$, we take a probabilistic approach. Rather than using the fixed polynomial $x^s - 1$, we consider random polynomials of the form

$$(x + \delta)^s - 1,$$

where $\delta$ is uniformly distributed over $\mathbb{F}_q$.

We claim that if $\alpha$ and $\beta$ are any two nonzero roots of $g$, then with probability $1/2$, exactly one of these is a root $(x + \delta)^s - 1$. It follows from this claim that so long as $g$ has at least 2 distinct nonzero roots, the probability that the polynomial $h(x) = \gcd(g(x), (x + \delta)^s - 1)$ is a proper divisor of $g$ is at least $1/2$.

Let us say that two elements $\alpha, \beta \in \mathbb{F}_q$ are of *different type* if they are both nonzero and $\alpha^s \neq \beta^s$ (in which case $\alpha^s = \pm 1$ and $\beta^s = \mp 1$). Our claim is an immediate consequence of the following theorem from [18].

**Theorem 3.44** (Rabin 1980)**.** *For every pair of distinct $\alpha, \beta \in \mathbb{F}_q$ we have*

$$\#\{\delta \in \mathbb{F}_q : \alpha + \delta \text{ and } \beta + \delta \text{ are of different type}\} = \frac{q-1}{2}.$$

*Proof.* Consider the map $\phi(\delta) = \frac{\alpha+\delta}{\beta+\delta}$, defined for $\delta \neq -\beta$. We claim that $\phi$ is a bijection from the set $\mathbb{F}_q - \{-\beta\}$ to the set $\mathbb{F}_q - \{1\}$. The sets are the same size, so we just need to show surjectivity. Let $\gamma \in \mathbb{F}_q - \{1\}$, then we wish to find a solution $\sigma \neq -\beta$ to $\gamma = \frac{\alpha+\sigma}{\beta+\sigma}$. We have $\gamma(\beta + \sigma) = \alpha + \sigma$ which means $\sigma - \gamma\sigma = \gamma\beta - \alpha$. This yields $\sigma = \frac{\gamma\beta-\alpha}{1-\gamma}$; we have $\gamma \neq 1$, and $\sigma \neq -\beta$, because $\alpha \neq \beta$. Thus $\phi$ is surjective.

We now note that

$$\phi(\delta)^s = \frac{(\alpha + \delta)^s}{(\beta + \delta)^s}$$

is $-1$ if and only if $\alpha + \delta$ and $\beta + \delta$ are of different type. The elements $\gamma = \phi(\delta)$ for which $\gamma^s = -1$ are precisely the non-residues in $\mathbb{F}_q \backslash \{1\}$, of which there are exactly $(q - 1)/2$. □

We now give the algorithm, which assumes that its input $f \in \mathbb{F}_q[x]$ is monic (has leading coefficient 1). If $f$ is not monic we can make it so by dividing $f$ by its leading coefficient, which does not change its roots or the complexity of finding them. You can find an implementation of the algorithm below in this Jupyter notebook.

**Algorithm 3.45.** Given a monic polynomial $f \in \mathbb{F}_q[x]$, output an element $r \in \mathbb{F}_q$ such that $f(r) = 0$, or `null` if no such $r$ exists.

1. If $f(0) = 0$ then return 0.
2. Compute $g = \gcd(f, x^q - x)$.
3. If $\deg g = 0$ then return `null`.
4. While $\deg g > 1$:
   a. Pick a random $\delta \in \mathbb{F}_q$.
   b. Compute $h = \gcd(g, (x + \delta)^s - 1)$.
   c. If $0 < \deg h < \deg g$ then replace $g$ by $h$ or $g/h$, whichever has lower degree.
5. Return $r$, where $g(x) = x - r$.

It is clear that the output of the algorithm is always correct: either it outputs a root of $f$ in step 1, proves that $f$ has no roots in $\mathbb{F}_q$ and outputs `null` in step 3, or outputs a root of $g$ that is also a root of $f$ in step 5 (note that whenever $g$ is updated it replaced with a proper divisor). We now consider its complexity.

### 3.11.1 Complexity analysis

It follows from Theorem 3.44 that the polynomial $h$ computed in step 4b is a proper divisor of $g$ with probability at least $1/2$, since $g$ has at least two distinct nonzero roots $\alpha, \beta \in \mathbb{F}_q$. Thus the expected number of iterations needed to obtain a proper factor $h$ of $g$ is bounded by 2, and the expected cost of obtaining such an $h$ is $O(\mathsf{M}(e(n+\log e))(n+\log e))$, where $n = \log q$ and $e = \deg g$, and this dominates the cost of the division in step 4c.

Each time $g$ is updated in step 4c its degree is reduced by at least a factor of 2. It follows that the expected total cost of step 4 is within a constant factor of the expected time to compute the initial value of $g = \gcd(f, x^q - x)$, which is $O(\mathsf{M}(d(n+\log d))(n+\log d))$, where $d = \deg f$; this simplifies to $O(n\,\mathsf{M}(dn))$ in the typical case that $\log d = O(n)$, which holds in all the applications we shall be interested in.

### 3.11.2 Finding all roots

We modify our algorithm to find all the distinct roots of $f$, by modifying step 4c to recursively find the roots of both $h$ and $g/h$. In this case the amount of work done at each level of the recursion tree is bounded by $O(\mathsf{M}(d(n+\log d))(n+\log d))$. Bounding the depth of the recursion is somewhat more involved, but one can show that with very high probability the degrees of $h$ and $g/h$ are approximately equal and that the expected depth of the recursion is $O(\log d)$. Thus we can find all the distinct roots of $f$ in

$$O(\mathsf{M}(d(n+\log d))(n+\log d)\log d) \tag{2}$$

expected time. When $\log d = O(n)$ this simplifies to $O(n\,\mathsf{M}(dn)\log d)$.

Once we know the distinct roots of $f$ we can determine their multiplicity by repeated division, but this is not the most efficient approach. By taking GCDs with derivatives one can first compute the *squarefree factorization* of $f$, which for a monic nonconstant polynomial $f$ is defined as the unique sequence $g_1, \ldots, g_m \in \mathbb{F}_q[x]$ of monic squarefree coprime polynomials with $g_m \neq 1$ such that

$$f = g_1 g_2^2 \cdots g_m^m.$$

When the degree of $f$ is less than the characteristic $p$ of $\mathbb{F}_q$, this can be done directly via Yun's algorithm [21]; see Exercise 14.30 in [9] for the necessary modifications to handle $\deg f \geq p$, which simply involves taking $p$th roots of known $p$th powers at suitable points and does not change the complexity.

**Algorithm 3.46.** Given a monic polynomial $f \in \mathbb{F}_q[x]$ with $\deg f < \mathrm{char}(\mathbb{F}_q)$, compute squarefree coprime polynomials $g_1, \ldots, g_m \in \mathbb{F}_q[x]$ with $g_m \neq 1$ such that $f = g_1 g_2^2 \cdots g_m^m$.

1. Compute $u = \gcd(f, f')$, $v_1 = f/u$, $w_1 = f'/u$, and set $i = 1$.
2. Compute $g_1 = \gcd(v_1, w_1 - v_1')$
3. While $v_i \neq g_i$:
   a. Compute $v_{i+1}$ with $v_i/g_i$ and $w_{i+1} = (w_i - v_i')/g_i$.
   b. Increment $i$ and compute $g_i = \gcd(v_i, w_i - v_i')$
4. Set $m = i$ and return $g_1, \ldots, g_m$.

The key fact that Yun's algorithm exploits is that if $g \in \mathbb{F}_q[x]$ is irreducible then $g^2 | f$ if and only if $g | \gcd(f, f')$. This is true because $\mathbb{F}_q$ is a perfect field, (by Theorem 3.22): if

$f = gh$ then $f' = g'h + gh'$ is divisible by $g$ if and only if $g'h$ is divisible by $g$, which occurs if and only if $g|h$ (in which case $g^2|f$), since $g' \neq 0$ for any irreducible $g$ in a perfect field.

Yun's algorithm begins with $u = \gcd(f, f') = f/(g_1 \cdots g_m) = g_2 g_3^2 \cdots g_m^{m-1}$, which yields $v_1 = g_1 \cdots g_m$ and $w_1 = \sum_j j g_j' v_1/g_j$, since $f' = u \sum_{j=1}^m j g_j' v_1/g_j$. One can show by induction that we always have

$$v_i = g_i \cdots g_m \qquad \text{and} \qquad w_i = \sum_{j=i}^{m} (j - i + 1) g_j' v_i/g_j,$$

which implies $\gcd(v_i, w_i - v_i') = g_i$, since $w_i - v_i' = \sum_{j=i+1}^m (j-i) g_j' v_i/g_j$; see [9, Thm. 14.23].

Yun's algorithm uses $O(\mathsf{M}(d) \log d)$ ring operations in $\mathbb{F}_q$, which is $O(\mathsf{M}(n)\, \mathsf{M}(d) \log d)$ bit operations and strictly dominated by the complexity bound (2) for finding the distinct roots of $f$. The cost of finding the distinct roots of each $g_i$ separately is no greater than the cost of finding the distinct roots of $f$, since the complexity of root-finding is superlinear in the degree, and with this approach we know *a priori* the multiplicity of each root of $f$.

It follows that we can determine all the roots of $f$ and their multiplicities, with the same time complexity as finding the distinct roots of $f$ (with the same leading constant, the extra time to determine the multiplicities is not only asymptotically negligible, when $f$ is not squarefree it is actually faster to compute the squarefree factorization first).

### 3.12 Computing a complete factorization

Factoring a polynomial $f \in \mathbb{F}_q[x]$ into irreducibles can effectively be reduced to finding roots of $f$ in extensions of $\mathbb{F}_q$. Linear factors of $f$ correspond to the roots of $f$ in $\mathbb{F}_q$, irreducible quadratic factors of $f$ correspond to roots of $f$ that lie in $\mathbb{F}_{q^2}$ but do not lie in $\mathbb{F}_q$; recall from Corollary 3.10 that *every* quadratic polynomial $\mathbb{F}_q[x]$ splits completely in $\mathbb{F}_{q^2}[x]$. More generally, each irreducible degree $d$-factor $g$ of $f$ is the minimal polynomial of a root $\alpha$ of $f$ that lies in $\mathbb{F}_{q^d}$ but none of its proper subfields; note that if $\alpha$ is a root of $f \in \mathbb{F}_q[x]$, then so are all of its Galois conjugates, and these are precisely the roots of its minimal polynomial.

One can thus compute the complete factorization of $f$ by applying the root-finding algorithm of the previous section over extensions of $\mathbb{F}_q$. But note that this involves picking random $\delta \in \mathbb{F}_{q^n}$ and performing polynomial arithmetic in $\mathbb{F}_{q^n}[x]$. As observed by Cantor and Zassenhaus shortly after Rabin's probabilistic root-finding algorithm appeared, rather than using random linear polynomials $x + \delta \in \mathbb{F}_{q^n}[x]$, it is better to use random degree $n$ polynomials in $\mathbb{F}_q[x]$, and one can show that this works just as well.

To state this more precisely, let us first note that by computing the squarefree factorization of $f$ and successively computing gcds with $x^{q^j} - x$ we can deterministically compute a factorization of $f$ into squarefree polynomials each of which is a product of irreducible polynomials of the same known degree; this is called the *distinct-degree factorization* of $f$. It then only remains to consider the case where $f$ is a product of distinct irreducible polynomials $f_1, \ldots, f_r$ of degree $j$. If $r = 1$ then $f$ is irreducible and we are done, so let us assume $r > 1$. By the Chinese Remainder Theorem (CRT) we have a ring isomorphism

$$\frac{\mathbb{F}_q[x]}{(f)} \simeq \frac{\mathbb{F}_q[x]}{(f_1)} \times \cdots \times \frac{\mathbb{F}_q[x]}{(f_r)} \simeq \mathbb{F}_{q^j}^r$$

that sends $h \bmod f$ to $(h \bmod f_1, \ldots, h \bmod f_r)$. We can represent $\mathbb{F}_q[x]/(f)$ as the set of all polynomials $u \in \mathbb{F}_q[x]$ of degree strictly less than $\deg f = rj$. If we pick $u$ uniformly at random, the polynomials $u \bmod f_i$ will also be uniformly random, and independent, by the

CRT (because the $f_i$ are coprime). In other words, picking $u$ at random amounts to picking an element $(u_1,\dots,u_r)$ of $\mathbb{F}_{q^j}^r$ at random. Moreover, if we pick a random $u$ coprime to $f$ we get a random $(u_1,\dots,u_r) \in (\mathbb{F}_{q^j}^\times)^r$.

Now let $s = (q^j-1)/2$. The ring isomorphism $u \mapsto (u_1,\dots,u_r) \in \mathbb{F}_{q^j}^r$ sends $u^s$ to $(u_1^s,\dots,u_r^s) \in \{0,\pm 1\}^r$, and if we restrict to $u$ that are coprime to $f$ we will have $(u_1^s,\dots,u_r^s) \in \{\pm 1\}^r$ and $\gcd(f,u^s-1)$ will be non-trivial whenever we have $u_i^s = 1$ for at least one but not all of the $u_i$. Exactly half the elements of $\mathbb{F}_{q^j}^\times$ are roots of $x^s-1$ and half are not, so this probability is

$$1 - 2^{-r} - 2^{-r} = 1 - 2^{1-r} \geq 1/2.$$

We thus have at least a fifty-fifty chance of splitting $f$ with each random $u$ coprime to $f$. We now give the complete Cantor-Zassenhaus algorithm, as described in [9, §14]; you can find a basic implementation of the algorithm below in this Jupyter notebook.

**Algorithm 3.47.** Given a monic polynomial $f \in \mathbb{F}_q[x]$, compute its irreducible factorization as follows:

1. Compute the squarefree factorization of $f = g_1 g_2^2 \cdots g_m^m$ using Yun's algorithm.
2. By successively computing $g_{ij} = \gcd(g_i, x^{q^j} - x)$ and replacing $g_i$ with $g_i/g_{ij}$ for $j = 1,2,3,\dots,\deg g_i$, factor each $g_i$ into polynomials $g_{ij}$ that are each (possibly trivial) products of distinct irreducible polynomials of degree $j$; note that once $j > (\deg g_i)/2$ we know $g_i$ must be irreducible and can immediately determine all the remaining $g_{ij}$.
3. Factor each nontrivial $g_{ij}$ into irreducible polynomials $g_{ijk}$ of degree $j$ as follows: while $\deg g_{ij} > j$ generate random polynomials $u \in \mathbb{F}_q[x]$ with $\deg u < \deg g_{ij}$ until either $h = \gcd(g_{ij}, u)$ or $h := \gcd(g_{ij}, u^{(q^j-1)/2} - 1)$ properly divides $g_{ij}$, then recursively factor $h$ and $g_{ij}/h$ (note that $j | \deg h$ and $j | \deg(g_{ij}/h)$).
4. Output each $g_{ijk}$ with multiplicity $i$.

In step 3, for $j > 1$ one computes $h_j := x^{q^j} \bmod g_{ij}$ via $h_j = h_{j-1}^q \bmod g_{ij}$. The expected cost of computing the $g_{ij}$ for a given $g_i$ of degree $d$ is then bounded by

$$O(\mathsf{M}(d(n+\log d))d(n+\log d)),$$

which simplifies to $O(dn\,\mathsf{M}(dn))$ when $\log d = O(n)$ and is in any case quasi-quadratic in both $d$ and $n$. The cost of factoring a particular $g_{ij}$ satisfies the same bound with $d$ replaced by $j$; the fact that this bound is superlinear and $\deg g_i = \sum_j \deg g_{ij}$ implies that the cost of factoring all the $g_{ij}$ for a particular $g_i$ is bounded by the cost of computing them, and superlinearity also implies that simply putting $d = \deg f$ gives us a bound on the cost of computing the $g_{ij}$ for all the $g_i$, and this bound also dominates the $O(\mathsf{M}(d)(\log d)\,\mathsf{M}(n))$ complexity of step 1.

Notice that the first three steps of Algorithm 3.47, which compute the squarefree and distinct degree factorizations of $f$ without making any random choices, yield a deterministic algorithm for computing the factorization pattern of $f$ (the degrees and multiplicities of its irreducible factors), and in particular, can function as an irreducibility test.

There are faster algorithms for polynomial factorization that use linear algebra in $\mathbb{F}_q$; see [9, 14.8]. These are of interest primarily when the degree $d$ is large relative to $n = \log q$.

The asymptotically fastest algorithm due to Kedlaya and Umans [17] uses recursive modular composition to obtain an expected running time of

$$O(d^{1.5+o(1)}n^{1+o(1)} + d^{1+o(1)}n^{2+o(1)}),$$

but this algorithm is primarily of theoretical interest.

There are also algorithms for $d = 2, 3, 4$ that use specialized methods for computing square-roots and cube-roots and then solve by radicals; these achieve a significant constant factor improvement for for most values of $q$, but will be slower in the rare worst case (the worst-case is slower by a $\log n/\log\log n$ factor [20], but one can easily detect this and switch algorithms if the slowdown outweighs the constant factor improvement).

For general purpose factoring of polynomials over finite fields, the Cantor-Zassenhaus algorithm is the algorithm of choice; it is implemented in virtually every computer algebra system that supports finite fields.

**Remark 3.48.** We should emphasize that all provably efficient algorithms known for root-finding and factoring polynomials over finite fields are probabilistic algorithms (of Las Vegas type). Even for the simplest non-trivial case, computing square roots, no deterministic polynomial-time algorithm is known. There are deterministic algorithms that can be shown to run in polynomial-time under the generalized Riemann hypothesis, but even these have worst-case running times that are asymptotically worse than the expected running time of the fastest probabilistic algorithms by at least a linear factor in $n = \log q$.

## 3.13 Summary

The table below summarizes the bit-complexity of the various arithmetic operations we have considered, both in the integer ring $\mathbb{Z}$ and in a finite field $\mathbb{F}_q$ of characteristic $p$ with $q = p^e$; in both cases $n$ denotes the bit-size of elements of the base ring (so $n = \log q$ for $\mathbb{F}_q$). Here we use $\mathsf{M}_q(n)$ to denote the time to multiply elements of $\mathbb{F}_q$. As noted in Remarks 3.36 and 3.42, if one is willing to assume a widely believed conjecture about the least prime in arithmetic progressions, we can take $\mathsf{M}_q(n) = O(n\log n)$ in the bounds below, and for $\log e = O(\log p)$ then this applies unconditionally.

| | integers $\mathbb{Z}$ | finite field $\mathbb{F}_q$ |
|---|---|---|
| addition/subtraction | $O(n)$ | $O(n)$ |
| multiplication | $O(n\log n)$ | $\mathsf{M}_q(n)$ |
| Euclidean division (reduction) | $O(n\log n)$ | $O(\mathsf{M}_q(n))$ |
| extended gcd (inversion) | $O(n\log^2 n)$ | $O(\mathsf{M}_q(n)\log n)$ |
| exponentiation | | $O(n\,\mathsf{M}_q(n))$ |
| square-roots (probabilistic) | | $O(n\,\mathsf{M}_q(n))$ |
| root-finding (probabilistic) | | $O(\mathsf{M}_q(d(n+\log d))(n+\log d))$ |
| factoring (probabilistic) | | $O(\mathsf{M}_q(d(n+\log d))d(n+\log d))$ |
| irreducibility testing | | $O(\mathsf{M}_q(d(n+\log d))d(n+\log d))$ |

In the case of root-finding, factorization, and irreducibility testing, $d$ is the degree of the polynomial, and for probabilistic algorithms these are bounds on the expected running time of a Las Vegas algorithm. The bound for exponentiation assumes that the bit-length of the exponent is $O(n^2)$. Unless $d$ is very large (super-exponential in $n$) one can ignore the $\log d$ terms in the last three complexity bounds, and we note that for large $d$ there are faster approaches to factoring and irreducibility testing that are sub-quadratic in $d$.

## References


[1] Razvan Barbulescu, Pierrick Gaudry, Antoine Joux, Emmanuel Thomé, *A heuristic quasi-polynomial algorithm for discrete logarithm in finite fields of small characteristic*, in *Advances in Cryptology — EUROCRYPT 2014*, LNCS **8441** (2014), 1–16.

[2] Elwyn R. Berlekamp, *Factoring polynomials over large finite fields*, Mathematics of Computation **24** (1970), 713–735.

[3] David G. Cantor and Hans Zassenhaus, *A new algorithm for factoring polynomials over finite fields*, Math. Comp. **36** (1981), 587–592.

[4] Henri Cohen et al., *Handbook of elliptic and hyperelliptic curve cryptography*, CRC Press, 2006.

[5] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction to algorithms*, third edition, MIT Press, 2009.

[6] Jack Dongarra, Francis Sullivan, *Top ten algorithms of the century*, Computing in Science and Engineering **2** (2000), 22–23.

[7] Martin Fürer, *Faster integer multiplication*, Proceedings of the thirty-ninth annual ACM Symposium on the Theory of Computing (STOC), 2007.

[8] Joachim von zur Gathen, *Irreducible trinomials over finite fields*, Mathematics of Computation **72** (2003), 1787–2000.

[9] Joachim von zur Gathen and Jürgen Gerhard, *Modern computer algebra*, third edition, Cambridge University Press, 2013.

[10] Dan Givoli, *The top 10 computational methods of the 20th century*, IACM Expressions **11** (2001), 5–9.

[11] Daniel M. Gordon, *A survey of fast exponentiation methods*, Journal of Algorithms **27** (1998), 129–146.

[12] David Harvey and Joris van der Hoeven, *Faster integer multiplication using short lattice vectors*, Thirteenth Algorithmic Number Theory Symposium (ANTS XIII), Open Book Series **2** (2018), 293–310.

[13] David Harvey and Joris van der Hoeven, *Polynomial multiplication over finite fields in time $O(n \log n)$*, J. ACM **69** (2022), 1–40.

[14] David Harvey and Joris van der Hoeven, *Integer multiplication in time $O(n \log n)$*, Annals of Math. **193** (2021), 563–617.

[15] David Harvey, Joris van der Hoeven, and Grégoire Lecerf, *Even faster integer multiplication*, J. Complexity **36** (2016), 1–30.

[16] A. A. Karatsuba, *The complexity of computations*, Proceedings of the Steklov Institute of Mathematics **211** (1995), 169–193 (English translation of Russian article).

[17] Kiran S. Kedlaya and Christopher Umans, *Fast polynomial factorization and modular composition*, SIAM J. Comput. **40** (2011), 1767–1802.

[18] Michael O. Rabin, *Probabilistic algorithms in finite fields*, SIAM Journal of Computing **9** (1980), 273–280.

[19] Arnold Schönhage and Volker Strassen, *Schnelle Multiplikation großer Zahlen*, Computing, **7** (1971), 281–292.

[20] Andrew V. Sutherland, *Structure computation and discrete logarithms in finite abelian p-groups*, Math. Comp. **80** (2011), 815–831.

[21] David Y.Y. Yun, *On square-free decomposition algorithms*, in *Proceedings of the third ACM symposium on symbolic and algebraic computation (SYMSAC '76)*, R.D. Jenks (ed.), ACM Press, 1976, 26–35.