3 Arithmetic in finite fields

In order to perform explicit computations with elliptic curves over finite fields, we first need to understand how to do arithmetic 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 week on finite field arithmetic (this lecture and the next), with the goal of understanding the most commonly used algorithms and analyzing their asymptotic complexity. This will force us to omit many details.

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

Example 3.1. As we will prove shortly, the multiplicative group of a finite field is cyclic. So one way to represent nonzero elements of a finite field is as powers of a fixed generator. With this representation multiplication and division are easy, and solving the discrete logarithm problem is essentially trivial. But addition is hard. We will instead choose a representation that makes addition and subtraction very easy, multiplication slightly harder but still easy, and division slightly harder still, but still easy (all quasi-linear time). But solving the discrete logarithm problem will be very difficult (no polynomial-time algorithm known).

For they 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. Fields of characteristic 2 are quite important in practical applications (coding theory in particular), and there are many specialized algorithms that are optimized for such fields, but we will not consider 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 in the next lecture.[2]

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 unique subfield isomorphic to $\mathbb{F}_p$. In particular, all fields of cardinality p are isomorphic.*

Proof. For each nonzero $a \in \mathbb{Z}/p\mathbb{Z}$ the map $\mathbb{Z}/p\mathbb{Z} \xrightarrow{\times a} \mathbb{Z}/p\mathbb{Z}$ is onto because its image is not $\{0\}$ and $\mathbb{Z}/p\mathbb{Z}$ has prime order; thus $ab = 1$ for some $b \in \mathbb{Z}/p\mathbb{Z}$ and $\mathbb{Z}/p\mathbb{Z}$ is a field. In any field of characteristic p the set $\{0, 1, 1+1, \ldots\}$ forms a subring isomorphic to $\mathbb{Z}/p\mathbb{Z} = \mathbb{F}_p$. $\square$

[1] With the recent breakthrough in computing discrete logarithms in finite fields of small characteristic [1] there is slightly less enthusiasm for using these fields in elliptic curve cryptography, although in principle this should only impact curves with small embedding degree (so-called "pairing-friendly" curves).

[2] For students already familiar with this material, I recommend the following exercise: write down each of the theorems in this section on a separate piece of paper and prove them yourself (to make things more interesting, see if you can do it without appealing to Galois theory).

Andrew V. Sutherland

The most common way to represent $\mathbb{F}_p$ for computational purposes is to pick a set of coset representatives for $\mathbb{Z}/p\mathbb{Z}$, typically 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 the roots of $x^q - x$ (the splitting field of $x^q - x$ over $\mathbb{F}_p$).

Theorem 3.5. *Let $q = p^n$ be a prime power. The field $\mathbb{F}_q$ has cardinality q and every field of cardinality q is isomorphic to $\mathbb{F}_q$.*

Proof. The map $x \mapsto x^q = x^{p^n}$ is an automorphism 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. The subfield of $\mathbb{F}_q$ fixed by this automorphism is the set S of roots of $x^q - x$, which includes $\mathbb{F}_p$, since

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

Thus $\mathbb{F}_q$ is equal to S. 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. Therefore $\#\mathbb{F}_q = q$.

Every field k of cardinality $q = p^n$ has characteristic p and therefore contains a subfield isomorphic to $\mathbb{F}_p$ (by the previous theorem). The multiplicative order of each nonzero element of k must divide $\#k^\times = q - 1$. It follows that $x \mapsto x^q$ fixes each element of k, hence every element of k is a root of $x^q - x$; And k has q elements so it contains all the roots of $x^q - x$ and is isomorphic to $\mathbb{F}_q$. $\square$

Remark 3.6. 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 $\mathbb{F}_q$.

Theorem 3.7. *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 $p^n - p^m = (p^m - 1)(p^{n-2m} + p^{n-3m} + \cdots + p^{2m} + p^m)$ 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 $x^{p^n} - x$ and $\mathbb{F}_{p^m} \subseteq \mathbb{F}_{p^n}$. $\square$

Theorem 3.8. *If $f \in \mathbb{F}_p[x]$ is an irreducible polynomial of degree n then $\mathbb{F}_p[x]/(f) \simeq \mathbb{F}_{p^n}$.*

Proof. The ring $k := \mathbb{F}_p[x]/(f)$ is an $\mathbb{F}_p$-vector space with basis $1, \ldots, x^{n-1}$ and therefore has cardinality p^n. If $g, h \in \mathbb{F}_p[x]$ are not divisible by f, then neither is gh; one way to see this is to note that since f is irreducible we must have $\gcd(f, g) = \gcd(f, h) = 1$, so f has no roots in common with g or h (over any extension of $\mathbb{F}_p$), and therefore none in common with gh.[3] This implies that for each nonzero $g \in k$, the map $k \xrightarrow{\times g} k$ is a linear transformation with trivial kernel, hence onto, so $gh = 1$ for some $h \in k$ and k is a field. $\square$

This implies that we can 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. By Theorem 3.5, it does not matter which f we pick, we always get the same field (up to isomorphism). We also note the following corollary.

[3]Equivalently, $\mathbb{F}_p[x]$ is a Euclidean domain, so irreducible elements are prime.

Corollary 3.9. *Every irreducible polynomial $f \in \mathbb{F}_p[x]$ 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 must be a root of $x^{p^n} - x$, hence an element of $\mathbb{F}_{p^n}$. □

Remark 3.10. This corollary implies that $x^{p^n} - x$ is the product over the divisors d of 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.

Theorem 3.11. *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 $m = n$. For each prime power q dividing m, there must be an element of G of order q (otherwise m would be smaller). Since G is abelian, any product of elements of relatively prime orders a and b has order ab. It follows that G contains an element of order $m = n$ and is therefore cyclic. □

Corollary 3.12. *The multiplicative group of a finite field is cyclic.*

If α is a generator for the multiplicative group $\mathbb{F}_q^\times$, then it certainly 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 α, but the converse need not hold. This motivates the following definition.

Definition 3.13. 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 3.14. *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 α be a generator for $\mathbb{F}_{p^n}^\times$ with minimal polynomial $f \in \mathbb{F}_p[x]$; then f is primitive, and there are $\phi(p^n - 1)$ possible choices for α. 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 $\mathbb{F}_q^\times$. □

The theorem implies that there are plenty of irreducible (and 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 a 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 only when its degree divides $p - 1$, but we can usually (although not always) choose f to be a trinomial; see [6]. Finite fields in cryptographic standards are often specified using an $f \in \mathbb{F}_p[x]$ that makes reduction modulo f particularly efficient.

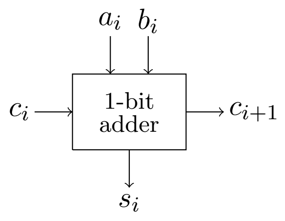
Having fixed a representation for $\mathbb{F}_q$, every finite field operation can ultimately be reduced to integer arithmetic (we will see exactly how this is done as we go along). We now consider the complexity of integer arithmetic.

3.2 Integer addition

Every positive 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. To add two integers, we write out their binary representations and apply the "schoolbook" method, adding bits and carrying as needed. As an example, let us compute 43+37=80 in binary.

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

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 3.15. Chaining adders is known as *ripple* addition and is no longer commonly used, since it forces a sequential computation (each carry bit depends on the one before it). In practice more sophisticated methods such as *carry-lookahead* are used to facilitate parallelism. This allows most modern microprocessors to add 64 (or even 128) bit integers 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 3.16. *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.*

3.3 A quick refresher on asymptotic notation

Let f and g be two real-valued functions whose domains include the positive integers. The 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 3.17. This notation is a horrible abuse of the symbol "=". When speaking 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 on the left of an "equality". 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 the big-Ω notation "$f(n) = \Omega(g(n))$", which means $g(n) = O(f(n))$.[4] Then there is the little-o notation "$f(n) = o(n)$," 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). There is also a little-omega notation, but the symbol ω 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 $f(n) = \Theta(n)$ means that both $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$ hold.

Warning 3.18. Don't confuse a big-O statement with a big-Θ statement. If Alice has an algorithm that is $O(2^n)$ this does not mean that Alice's algorithm runs in exponential time. Nor does it mean that Bob's $O(n^2)$ algorithm is asymptotically faster; Alice's algorithm could also 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

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

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

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 $f_n(m) = O(g_n(m))$).

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. The space complexity of an algorithm can never be greater than its time complexity (it takes time to use space), but it may be less. 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 [2].

Warning 3.19. 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 which finite field operations are being used.

3.4 Integer multiplication

We now consider the problem of multiplying two positive integers.

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 an upper bound on the time $\mathsf{M}(n)$ to multiply two n-bit integers, but we can do better.

3.4.2 Karatsuba's algorithm

Rather than representing n-bit integers using n digits in base 2, we may instead represent them using 2 digits in base $2^{n/2}$. We may then compute their product as follows

$$\begin{aligned}
a &= a_0 + 2^{n/2}a_1\\
b &= b_0 + 2^{n/2}b_1\\
ab &= a_0b_0 + 2^{n/2}(a_1b_0 + b_1a_0) + 2^n a_1b_1
\end{aligned}$$

Naively, this requires four multiplications of $(n/2)$-bit integers and three additions of $O(n)$-bit integers (note that 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, we can use the following simplification:

$$a_0b_1 + b_0a_1 \quad = \quad (a_0 + a_1)(b_0 + b_1) - a_0b_0 - a_1b_1.$$

By reusing the common subexpressions a_0b_0 and a_1b_1, we can multiply a and b 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 is known as Karatsuba's algorithm.

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

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

Thus $\mathsf{M}(n) = O(n^{\lg 3}) \approx O(n^{1.59})$.[5]

3.4.3 The Fast Fourier Transform (FFT)

The fast Fourier transform is generally considered one of the top ten algorithms of the twentieth century [3, 5], with 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 [7, §8]. It is actually more natural to address the problem of polynomial multiplication first.

Let R be a commutative ring containing a primitive nth root of unity ω, 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 $(f_0, \ldots, f_{n-1}) \in R^n$ 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

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

This map is invertible; if we consider the Vandermonde matrix

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

then $\mathrm{DFT}_\omega(f) = V_\omega f^t$. The inverse of the matrix V_ω is $\frac{1}{n} V_{\omega^{-1}}$, and it follows that

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

[5] We write $\lg n$ for $\log_2 n$.

Thus if we have an algorithm to compute DFT_ω we can use it to compute DFT_ω^{-1}: simply replace ω by ω^{-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 from 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 $f, g \in R^n$:

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

Theorem 3.20. $\mathrm{DFT}_\omega(f * g) = \mathrm{DFT}_\omega(f) \cdot \mathrm{DFT}_\omega(g)$.

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 ω is an nth root of unity. $\square$

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

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

which allows us to multiply polynomials using the discrete Fourier transform. To put this into practice, we need an efficient way to compute DFT_ω. This is achieved by the following 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 $(r(\omega^0), s(\omega^0), r(\omega^2), s(\omega^2), \ldots, r(\omega^{n-2}), s(\omega^{n-2}))$
```

Let $T(n)$ be the number of operations in R used by the FFT algorithm. Then $T(n)$ satisfies the recurrence $T(n) = 2T(\frac{n}{2}) + O(n)$, and therefore $T(n) = O(n \lg n)$.

Theorem 3.21. *The FFT algorithm outputs* $\mathrm{DFT}_\omega(f)$.

Proof. We must verify that the kth entry of the output vector is $f(\omega^k)$, for $0 \le k < n$. For the even values of $k = 2i$ 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 the odd values of $k = 2i+1$ 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 $\omega^{n/2} = -1$. $\square$

We can use the FFT to multiply polynomials over any ring R that contains a suitable nth root of unity using $O(n \log n)$ operations in R. But how does this help us to multiply integers?

To any integer $a = \sum_{i=0}^{n-1} a_i 2^i$ we may associate the polynomial $f_a(x) = \sum_{i=0}^{n} a_i x^i$, so that $a = f_a(2)$. We can then multiply integers a and b via $ab = f_{ab}(2) = (f_a f_b)(2)$; in practice one typically uses base 2^{64} rather than base 2.

In order to get the required nth root of unity ω, Schönhage and Strassen [9] adjoin a "virtual" root of unity to $\mathbb{Z}[x]$, and this yields an algorithm to multiply integers in time $O(n \log n \log\log n)$, which gives us a new upper bound:

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

Remark 3.22. As shown by Fürer [4], 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. Very recently the sharper bound

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

was proved in [8], and under a conjecture about the existence of Mersenne primes, the 8 can be replaced with a 4. But these improvements are primarily of theoretical interest; the smallest integers for which they are likely to be faster than an optimized implementation of the Schönhage–Strassen algorithm are likely to be many gigabytes in size.

3.5 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 $m = \lceil 2\lg p + \lg_2(\deg f + 1)\rceil$, the coefficients of $\hat{f}$ will appear in the binary representation of $\hat{f}(2^m)$ separated by blocks of $m - \lceil \lg p \rceil$ zeroes. 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 kth block of m binary digits in N contains the binary representation of the kth 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 $\log d = O(n)$ and this simplifies to $O(\mathsf{M}(dn)$ In particular, we can multiply elements of $\mathbb{F}_q \simeq \mathbb{F}_p[x]/(f)$ in time $O(\mathsf{M}(n))$, where $n = \log q$, provided that either $\log \deg f = O(n)$ or $\log p = O(1)$, which are the two most typical cases, corresponding to large characteristic and small characteristic fields, respectively.

Remark 3.23. 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 $\mathsf{M}(dn) = o(\mathsf{M}(d)\mathsf{M}(n))$.

3.6 Complexity of integer arithmetic

To sum up, we have the following complexity bounds for arithmetic on n-bit integers:

addition/subtraction	$O(n)$
multiplication (schoolbook)	$O(n^2)$
multiplication (Karatsuba)	$O(n^{\lg 3})$
multiplication (FFT)	$O(n \log n \log\log n)$

References

[1] Razvan Barbulescu, Pierrick Gaudry, Antoine Joux, Emmanuel Thomé, *A quasi-polynomial algorithm for discrete logarithm in finite fields of small characteristic*, 2013 preprint, arXiv:1306.4244.

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

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

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

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

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

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

[8] David Harvey, Joris van der Hoeven, and Grégoire Lecerf, *Even faster integer multiplication*, 2014 preprint, arXiv:1407.3360.

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