3.1 Arithmetic in finite fields

To make explicit computations with elliptic curves over finite fields, we need to know how to perform arithmetic operations in finite fields, and we would like to do so as efficiently as possible. In the applications we will consider, the finite fields involved may be very large, so it is important to understand the asymptotic complexity of finite field operations.

This is a huge topic, one to which an entire course could be devoted. However, 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.

For they sake of brevity, we will focus on fields of large characteristic (and primes fields in particular), although the algorithms we describe will generally work in any finite field of odd characteristic. Fields of characteristic 2 are quite important in practical applications and there are specialized algorithms that are optimized for such fields, but we will not consider them here.

We may represent elements of a prime field $\mathbb{F}_p \simeq \mathbb{Z}/p\mathbb{Z}$ as integers in the interval $[0, p-1]$. For finite fields $\mathbb{F}_q$ with $q = p^d$, we may pick any irreducible monic polynomial $f \in \mathbb{F}_p[x]$ of degree d and represent $\mathbb{F}_q$ as the quotient $\mathbb{F}_p[x]/(f) \simeq (\mathbb{Z}/p\mathbb{Z})[x]/(f)$, whose elements may be uniquely represented as polynomials of degree less than d with integer coefficients in $[0, p-1]$. The choice of the polynomial f impacts 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 if (and only if) d divides $p-1$: let $f = x^d - a$ where a is a generator of $\mathbb{F}_p^*$. We can often, but not always, choose f to be a trinomial; see [5] for necessary criteria. It is also useful (but not necessary) for f to be a *primitive* polynomial; this means that f is the minimal polynomial of a generator for $\mathbb{F}_q^*$, equivalently, the polynomial x generates the multiplicative group of $\mathbb{F}_p[x]/(f)$.

Having fixed a representation for $\mathbb{F}_q$ as described above, every finite field operation can be reduced to arithmetic operations on integers, which we now consider.

Remark 3.1. There are other ways to represent finite fields. For example, if we pick a generator α for the cyclic group $\mathbb{F}_q^*$ we can uniquely represent every nonzero element of $\mathbb{F}_q$ in the form α^k for some integer $k \in [0, q-2]$. This representation makes multiplication very easy (just add exponents modulo $q-1$), but addition then becomes quite complicated. We will not consider this representation here.

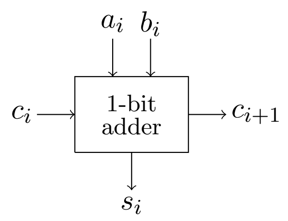
3.2 Addition

Every nonnegative integer a has a unique *binary representation* $a = \sum_{i=0}^{n-1} a_i 2^i$, where the binary digits $a_i \in \{0, 1\}$ 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 when needed. As an example, let us compute 43+37=80 in binary.

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

Andrew V. Sutherland

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.



$$c_{i+1} = (b_1 \wedge b_2) \vee (c \wedge b_1) \vee (c \wedge b_2)$$

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

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 3.2. 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 a parallel implementation. 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; is 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 $n = \log_2 p$ is the size of 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 $O(d \log p)$ bit operations, where $d = \deg f$, which is $O(n)$ bit operations, where $n = \log_2 q$ is again the size of a finite field element.

3.3 A quick refresher on asymptotic notation

Let us assume that f and g are functions from the positive integers to the positive real numbers. 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 $f(n) \leq cg(n)$.

This is equivalent to

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

Strictly speaking, 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"). It is often 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 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 is equivalent to writing $g(n) = O(f(n))$. 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, but number theorists usually mean the latter). There is also a little-omega notation, but the symbol ω already has many uses in number theory so 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.

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 $f(m, n) \leq cg(m, n)$.

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

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 [1].

3.4 Multiplication

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 $\mathsf{M}(n) = O(n^{\lg 3}) \approx O(n^{1.59})$.[1]

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 [2, 4], 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 [6, §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

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

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 $f(x)$ and the vector f interchangeably.

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

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

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 $\mathrm{DFT}_\omega^{-1} = \frac{1}{n}\mathrm{DFT}_{\omega^{-1}}$. Thus if we have an algorithm to compute DFT_ω we can also 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 vectors $f, g \in R^n$:

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

Reducing the product on the right modulo $x^n - 1$ ensure 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 $f, g \in R^n$:

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

Theorem 3.3. $\mathrm{DFT}_\omega(f * g) = \mathrm{DFT}_\omega(f) \cdot \mathrm{DFT}_\omega(g)$.

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

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

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. □

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

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

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 3.4. *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:

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

And 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 $a = f_a(2)$. Then we can 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 [7] 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

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

This bound has recently been improved to $O(n \log n\, 2^{O(\log^* n)})$ by Fürer [3], but this improvement is primarily of theoretical interest and is not currently used in practice.

3.5 Kronecker substitution

We also note an important converse of the above: 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 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 $m = 2\lg p + \lceil \lg_2 \deg f \rceil$, the coefficients of $\hat{f}$ will appear in the binary representation of $\hat{f}(2^m)$ separated by blocks of m zeroes. If g is a polynomial of similar degree, we can easily recover the coefficients of $h = \hat{f}\hat{g} \in \mathbb{Z}[x]$ in the integer product $N = \hat{f}(2^m)\hat{g}(2^m)$; the key point is that m is large enough to ensure that the kth block of m binary digits in N contains the binary representation of the kth coefficient of h. We then reduce the coefficients of h modulo p to recover $fg \in \mathbb{F}_p[x]$. 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.

Remark 3.5. Under the standard assumption that the cost $\mathsf{M}(n)$ of multiplication grows super-linearly, using Kronecker substitution is *strictly superior* to the more conventional approach of multiplying coefficients of the polynomials individually, which would yield the asymptotic bound $O(\mathsf{M}(d)\mathsf{M}(n))$ rather than $O(\mathsf{M}(dn))$.

3.6 Complexity of arithmetic operations

To sum up, we have determined the following complexities for integer arithmetic:

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

[2] Jack Dongarra, Francis Sullivan, *Top ten algorithms of the century*, Computing in Science and Engineering **2:1** (2000), 22–23.

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

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

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

[6] Joachim von zur Gathen and Jürgen Gerhard, *Modern Computer Algebra*, second edition, Cambridge University Press, 2003.

[7] A. Schönhage and V. Strassen, ‘Schnelle Multiplikation großer Zahlen’, *Computing*, **7** (1971), 281–292.