4 Finite field arithmetic

We saw in Lecture 3 how to efficiently multiply integers, and, using Kronecker substitution, how to efficiently multiply polynomials with integer coefficients. This gives us what we need to multiply elements in finite fields, provided we can efficiently reduce the results to our standard representations of $\mathbb{F}_p \simeq \mathbb{Z}/p\mathbb{Z}$ and $\mathbb{F}_q \simeq \mathbb{F}_p[x]/(f)$, using integers in $[0, p-1]$ and polynomials of degree less than $\deg f$, respectively. In both cases we use Euclidean division.

4.1 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 Netwon 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 $3\,\mathsf{M}(n) + O(n)$. If we expect to repeatedly perform Euclidean division with the same denominator, as when working in $\mathbb{F}_p$, the cost of all subsequent reductions can be reduced to $\mathsf{M}(n) + O(n)$ using what is known as *Barret Reduction*; see [2, Alg. 10.17].[1] In any case, we obtain the following bound for multiplication in $\mathbb{F}_p$ using our standard representation as integers in $[0, p-1]$.

Theorem 4.1. *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 [3]. 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 4.2 (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 4.3. *Let $q = p^d$ be a prime power, and assume that either $\log d = O(\log p)$ or $p = O(1)$. The time to multiply two elements of $\mathbb{F}_q$ is $O(\mathsf{M}(n))$, where $n = \lg q$.*

Remark 4.4. The constraints on the relative growth rate on p and d in the theorem above are present only so that we can easily express our bounds in terms of the bound $\mathsf{M}(n)$ for

[1] The algorithm given in [2] for the precomputation step of Barret reduction uses Newton iteration with a fixed precision, which is asymptotically suboptimal; it is better to use the varying precision approach described above. But in practice the precomputation cost is usually not a major concern.

multiplying integers. In fact, for all the bounds currently known for $\mathsf{M}(n)$, Theorem 4.3 holds uniformly, without any assumptions about the relative growth rate of p and d. More precisely, it follows from the recent result of [5] that for any prime power q we can multiply two elements in $\mathbb{F}_q$ in time $O(n \log n 8^{\log^* n})$, no matter how $q = p^d$ tends to infinity; but the proof of this requires more than just Kronecker substitution.

4.2 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

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

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 when the kth 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 the 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 [3, §11] for details.

Theorem 4.5. *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 ± 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 [3, 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 4.6. *Let $q = p^d$ be a prime power and assume that either $\log d = O(\log p)$ or $p = O(1)$. The time to invert an element of $\mathbb{F}_q^\times$ is $O(\mathsf{M}(n) \log n)$, where $n = \lg q$.*

4.3 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

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

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 [2, Ch. 9] and [4] Here we give just one example, windowed exponentiation, which is able to reduce the constant factor from 2 to an essentially optimal $1 + o(1)$.

4.3.1 Fixed-window exponentiation

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

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

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.

4.3.2 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 4.7. 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

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

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.

4.4 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, rth roots.[2]

The algorithm we give here was originally proposed by Berlekamp for prime fields [1], and then refined and extended by Rabin [6], 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.[3]

4.4.1 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 be 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

[2] An entirely different approach to computing rth 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 then the algorithm presented here in the worst case.

[3] 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.

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.

4.4.2 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 α_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 $\mathbb{F}_q$.

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 $f(x)$; this would take time exponential in $n = \log q$.[4] Instead, we compute $x^q \bmod f$ by exponentiating the polynomial x to the qth 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. 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 §4.1, 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 §4.3 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 which splits completely in $\mathbb{F}_q[x]$ and whose roots are precisely the distinct roots of $f(x)$. 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.

4.5 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.

[4] 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.

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 $q-1$, which is even (since q is odd), thus the squares in $\mathbb{F}_q^\times$ are the elements that are even powers of a generator, equivalently, elements whose order divides $(q-1)/2$. 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.

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 δ is uniformly distributed over $\mathbb{F}_q$. We claim that if α and β 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 $\alpha^s \neq \beta^s$. Our claim is an immediate consequence of the following theorem from [6].

Theorem 4.8 (Rabin)**.** *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 ϕ is a bijection form 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 ϕ 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 nor change the complexity of the algorithm below.

Algorithm 4.9. 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.

4.5.1 Complexity analysis

It follows from Theorem 4.8 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.

4.5.2 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

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

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

$$f = g_1 g_2^2 g_3^3 \cdots g_m^m.$$

This can be done using Yun's algorithm [7] (see Algorithm 14.21 and Exercise 14.30 in [3]) using $O(\mathsf{M}(d)\log d)$ operations in $\mathbb{F}_q$, which is dominated by the complexity bound for root-finding determined above. The cost of finding the roots of all the g_i is no greater than the cost of finding the roots of f (the complexity of root-finding is superlinear in the degree), and with this approach we know *a priori* the multiplicity of each root as a root of f. It follows that we can determine all the roots of f and their multiplicities, within the time bound given above for finding the distinct roots of f.

4.6 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 3.9 that *every* quadratic polynomial $\mathbb{F}_q[x]$ splits completely in $\mathbb{F}_{q^2}[x]$. Similarly, each irreducible degree d-factor of f corresponds to a root of f that lies in $\mathbb{F}_{q^d}$ but none of its proper subfields.

We now sketch the algorithm; see [3, §14] for further details on each step.

Algorithm 4.10. Given a monic polynomial $f \in \mathbb{F}_q[x]$, compute its irreducible factorization in $\mathbb{F}_q[x]$ as follows:

1. Determine the largest power x^e dividing f and replace f with f/x^e.
2. Compute the squarefree factorization $f = g_1 g_2^2 \cdots g_m^m$ of f using Yun's algorithm.
3. By successively computing $g_{ij} = \gcd(g_i, x^{q^j} - x)$ and replacing g_i with g_i/g_{ij} for $j = 1, 2, 3, \ldots . \deg g_i$, 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}.
4. Factor each nontrivial g_{ij} into irreducible polynomials h_{ijk} of degree j as follows: while $\deg g_{ij} > j$ generate random monic polynomials $u \in \mathbb{F}_q[x]$ of degree j until $h := \gcd(g_{ij}, u^{(q^j-1)/2} - 1)$ properly divides g_{ij}, then recursively factor h and g_{ij}/h.
5. Output x with multiplicity e and output each g_{ijk} with multiplicity i.

In step 3, for $j > 1$ one computes $h_j := x^{q^j} \bmod g_{ij}$ via $h_j = h_{j-1}^q \bmod q_{ij}$. 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.

As a special case, Algorithm 4.10 can be used as a deterministic algorithm for irreducibility testing; steps 1-3 do not involve any random choices and suffice to determine whether or not the input is irreducible (this holds if and only if g_{1d} is the only nontrivial g_{ij}).

There are faster algorithms for polynomial factorization (including algorithms that are sub-quadratic in the degree) that use linear algebra in $\mathbb{F}_q$; see [3, 14.8]. These are of interest primarily when the degree d is large relative to $n = \log q$ or the characteristic is small.

4.6.1 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 $q = p^e$, where we assume either $\log e = O(\log q)$ (large characteristic) or $p = O(1)$ (small characteristic); in both cases n is the bit-size of the inputs (so $n = \log q$ for $\mathbb{F}_q$).

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

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 $O(n^2)$.

References

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

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

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

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

[5] David Harvey, Joris van der Hoeven, and Grégoire Lecerf, *Faster polynomial multiplication over finite fields*, arXiv:1407.3361.

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

[7] 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.