Notes on finite fields

James Aspnes

December 13, 2010

Our goal here is to find computationally-useful structures that act enough like the rational numbers $\mathbb{Q}$ or the real numbers $\mathbb{R}$ that we can do arithmetic in them that are small enough that we can describe any element of the structure uniquely with a finite number of bits. Such structures are called **finite fields**.

An example of a finite field is $\mathbb{Z}_p$, the integers mod p (see ModularArithmetic). These finite fields are inconvenient for computers, which like to count in bits and prefer numbers that look like 2^n to horrible nasty primes. So we'd really like finite fields of size 2^n for various n, particularly if the operations of addition, multiplication, etc. have a cheap implementation in terms of sequences of bits. To get these, we will show how to construct a finite filed of size p^n for any prime p and positive integer n, and then let $p = 2$.

1 A magic trick

We will start with a magic trick. Suppose we want to generate a long sequence of bits that are hard to predict. One way to do this is using a mechanism known as a linear-feedback shift register (LFSR). There are many variants of LFSRs. Here is one that generates a sequence that repeats every 15 bits by keeping track of 4 bits of state, which we think of as a binary number $r_3r_2r_1r_0$.

To generate each new bit, we execute the following algorithm:

1. Rotate the bits of r left, to get a new number $r_2r_1r_0r_3$.
2. If the former leftmost bit was 1, flip the new leftmost bit.
3. Output the rightmost bit.

Here is the algorithm in action, starting with $r = 0001$:

r	rotated r	rotated r after flip	output
0001	0010	0010	0
0010	0100	0100	0
0100	1000	1000	0
1000	0001	1001	1
1001	0011	1011	1
1011	0111	1111	1
1111	1111	0111	1
0111	1110	1110	0
1110	1101	0101	1
0101	1010	1010	0
1010	0101	1101	1
1101	1011	0011	1
0011	0110	0110	0
0110	1100	1100	0
1100	1001	0001	1
0001	0010	0010	0

After 15 steps, we get back to 0001, having passed through all possible 4-bit values except 0000. The output sequence 000111101011001... has the property that every 4-bit sequence except 0000 appears starting at one of the 15 positions, meaning that after seeing any 3 bits (except 000), both bits are equally likely to be the next bit in the sequence. We thus get a sequence that is almost as long as possible given we have only 2^4 possible states, that is highly unpredictable, and that is cheap to generate. So unpredictable and cheap, in fact, that the governments of both the United States and Russia operate networks of orbital satellites that beam microwaves into our brains carrying signals generated by linear-feedback shift registers very much like this one. Similar devices are embedded at the heart of every modern computer, scrambling all communications between the motherboard and PCI cards to reduce the likelihood of accidental eavesdropping.

What horrifying deep voodoo makes this work?

2 Fields and rings

A **field** is a set F together with two operations $+$ and $\cdot$ that behave like addition and multiplication in the rationals or real numbers. Formally, this means that:

1. Addition is **associative**: $(x + y) + z = x + (y + z)$ for all x, y, z in F.
2. There is an **additive identity** 0 such that $0 + x = x + 0 = x$ for all x in F.
3. Every x in F has an **additive inverse** $-x$ such that $x+(-x) = (-x)+x = 0$.
4. Addition is **commutative**: $x + y = y + x$ for all x, y in F.

5. Multiplication **distributes** over addition: $x \cdot (y + z) = (x \cdot y + x \cdot z)$ and $(y + z) \cdot x = (y \cdot x + z \cdot x)$ for all x, y, z in F.
6. Multiplication is associative: $(x \cdot y) \cdot z = x \cdot (y \cdot z)$ for all x, y, z in F.
7. There is a **multiplicative identity** 1 such that $1 \cdot x = x \cdot 1 = x$ for all x in F.
8. Multiplication is commutative: $x \cdot y = y \cdot x$ for all x, y in F.
9. Every x in $F - \{0\}$ has a **multiplicative inverse** x^{-1} such that $x \cdot x^{-1} = x^{-1} \cdot x = 1$.

Some structures fail to satisfy all of these axioms but are still interesting enough to be given names. A structure that satisfies 1–3 is called a **group**; 1–4 is an **abelian group**; 1–7 is a **ring**; 1–8 is a **commutative ring**. In the case of groups and abelian groups there is only one operation $+$. There are also more exotic names for structures satisfying other subsets of the axioms; see AbstractAlgebra.

Some examples of fields: $\mathbb{R}, \mathbb{Q}, \mathbb{C}, \mathbb{Z}_p$ where p is prime. We will be particularly interested in $\mathbb{Z}_p$, since we are looking for *finite* fields that can fit inside a computer.

If $(F, +, \cdot)$ looks like a field except that multiplication isn't necessarily commutative and some nonzero elements might not have inverses, then it's a **ring** (or a **commutative ring** if multiplication is commutative). The integers $\mathbb{Z}$ are an example of a commutative ring, as is $\mathbb{Z}_m$ for $m > 1$. Square matrices of fixed dimension greater than 1 are an example of a non-commutative ring.

3 Polynomials over a field

Any field F generates a **polynomial ring** $F[x]$ consisting of all polynomials in the variable x with coefficients in F. For example, if $F = \mathbb{Q}$, some elements of $\mathbb{Q}[x]$ are $3/5, (22/7)x^2 + 12,9003x^{417} - (32/3)x^4 + x^2$, etc. Addition and multiplication are done exactly as you'd expect, by applying the distributive law and combining like terms: $(x+1) \cdot (x^2+3/5) = x \cdot x^2 + x \cdot (3/5) + x^2 + (3/5) = x^3 + x^2 + (3/5)x + (3/5)$.

The **degree** $\deg(p)$ of a polynomial p in $F[x]$ is the exponent on the **leading term**, the term with a nonzero coefficient that has the largest exponent. Examples: $\deg(x^2 + 1) = 2$, $\deg(17) = 0$. For 0, which doesn't have any terms with nonzero coefficients, the degree is taken to be $-\infty$. Degrees add when multiplying polynomials: $\deg((x^2 + 1)(x + 5)) = \deg(x^2 + 1) + deg(x + 5) = 2 + 1 = 3$; this is just a consequence of the leading terms in the polynomials we are multiplying producing the leading term of the new polynomial. For addition, we have $\deg(p+q) \leq \max(\deg(p), \deg(q))$, but we can't guarantee equality (maybe the leading terms cancel).

Because $F[x]$ is a ring, we can't do division the way we do it in a field like $\mathbb{R}$, but we *can* do division the way we do it in a ring like $\mathbb{Z}$, leaving a remainder. The equivalent of the integer **division algorithm** for $\mathbb{Z}$ is:

Division algorithm for polynomials Given a polynomial f and a nonzero polynomial g in $F[x]$, there are unique polynomials q and r such that $f = q \cdot g + r$ and $\deg(r) < \deg(g)$.

The essential idea is that we can find q and r using the same process of long division as we use for integers. For example, in $\mathbb{Q}[x]$:

```
            x -  1
     ---------------
x+2 ) x^2 +  x +  5
      x^2 + 2x
      --------
            -x +  5
            -x + -5
            =======
                 10
```

From this we get $x^2 + x + 5 = (x + 2)(x - 1) + 10$, with $\deg(10) = 0 < deg(x + 2) = 1$. We are going to use this later to define finite fields by taking $F[x]$ modulo some well-chosen polynomial, analogously to the way we can turn $\mathbb{Z}$ (a ring) into a field $\mathbb{Z}_p$ by taking quotients mod p.

4 Algebraic field extensions

Given a field F, we can make a bigger field by adding in extra elements that behave in a well-defined and consistent way. An example of this is the extension of the real numbers $\mathbb{R}$ to the complex numbers $\mathbb{C}$ by adding i.

The general name for this trick is **algebraic field extension** and it works by first constructing the ring of polynomials $F[x]$ and then smashing it down into a field by taking remainders modulo some fixed polynomial $p(x)$. For this to work, the polynomial has to to be **irreducible**, which mean that $p(x) = 0$ if and only if $x = 0$, or equivalently that p can't be factored as $(x + a)p'$ for some a and p' (which makes irreducibility sort of like being prime, and makes this construction sort of like the construction of $\mathbb{Z}_p$).

The fact that the resulting object is a field follows from inheriting all the commutative ring properties from $F[x]$, plus getting multiplicative inverses for essentially the same reason as in $\mathbb{Z}_p$: we can find them using the extended Euclidean algorithm applied to polynomials instead of integers (we won't prove this).

In the case of the complex numbers $\mathbb{C}$, the construction is $\mathbb{C} = \mathbb{R}[i]/(i^2 + 1)$. Because $i^2 + 1 = 0$ has no solution $i \in \mathbb{R}$, this makes $i^2 + 1$ an irreducible polynomial. An element of $\mathbb{C}$ is then a degree-1 or less polynomial in $\mathbb{R}[i]$, because these are the only polynomials that survive taking the remainder mod $i^2 + 1$ intact.

If you've used complex numbers before, you were probably taught to multiply them using the rule $i^2 = -1$, which is a rewriting of $i^2 + 1 = 0$. This is equivalent to taking remainders: $(i+1)(i+2) = (i^2+3i+2) = 1 \cdot (i^2+1)+(3i+1) = 3i+1$.

The same thing works for other fields and other irreducible polynomials. For example, in $\mathbb{Z}_2$, the polynomial x^2+x+1 is irreducible, because $x^2+x+1=0$ has no solution (try plugging in 0 and 1 to see). So we can construct a new finite field $\mathbb{Z}_2[x]/(x^2+x+1)$ whose elements are polynomials with coefficients in $\mathbb{Z}_2$ with all operations done modulo x^2+x+1.

Addition in $\mathbb{Z}_2[x]/(x^2+x+1)$ looks like vector addition:[1] $(x+1)+(x+1)=0\cdot x+0=0, (x+1)+x=1, (1)+(x)=(x+1)$. Multiplication in $\mathbb{Z}_2[x]/(x^2+x+1)$ works by first multiplying the polynomials and taking the remainder mod (x^2+x+1): $(x+1)\cdot(x+1)=x^2+1=1\cdot(x^2+x+1)+x=x$. If you don't want to take remainders, you can instead substitute $x+1$ for any occurrence of x^2 (just like substituting -1 for i^2 in $\mathbb{C}$), since $x^2+x+1=0$ implies $x^2=-x-1=x+1$ (since $-1=1$ in $\mathbb{Z}_2$).

The full multiplication table for this field looks like this:

	0	1	x	$x+1$
0	0	0	0	0
1	0	1	x	$x+1$
x	0	x	$x+1$	1
$x+1$	0	$x+1$	1	x

We can see that every nonzero element has an inverse by looking for ones in the table; e.g. $1\cdot 1=1$ means 1 is its own inverse and $x\cdot(x+1)=x^2+x=1$ means that x and $x+1$ are inverses of each other.

Here's the same thing for $\mathbb{Z}_2[x]/(x^3+x+1)$:

	0	1	x	$x+1$	x^2	x^2+1	x^2+x	x^2+x+1
0	0	0	0	0	0	0	0	0
1	0	1	x	$x+1$	x^2	x^2+1	x^2+x	x^2+x+1
x	0	x	x^2	x^2+x	$x+1$	1	x^2+x+1	x^2+1
$x+1$	0	$x+1$	x^2+x	x^2+1	x^2+x+1	x^2	1	x
x^2	0	x^2	$x+1$	x^2+x+1	x^2+x	x	x^2+1	1
x^2+1	0	x^2+1	1	x^2	x	x^2+x+1	$x+1$	x^2+x
x^2+x	0	x^2+x	x^2+x+1	1	x^2+1	$x+1$	x	x^2
x^2+x+1	0	x^2+x+1	x^2+1	x	1	x^2+x	x^2	$x+1$

Note that we now have $2^3=8$ elements. In general, if we take $\mathbb{Z}_p[x]$ modulo a degree-n polynomial, we will get a field with p^n elements. These turn out to be all the possible finite fields, with exactly one finite field for each number of the form p^n (up to isomorphism, which means that we consider two fields equivalent if there is a bijection between them that preserves $+$ and $\cdot$). We can refer to a finite field of size p^n abstractly as $GF(p^n)$, which is an abbreviation for the **Galois field of order** p^n.

[1] This is not an accident; it can be shown that that any extension field acts like a vector space over its base field.

5 Applications

So what are these things good for?

On the one hand, given an irreducible polynomial $p(x)$ of degree n over $\mathbb{Z}_2(x)$, it's easy to implement arithmetic in $\mathbb{Z}_2[x]/p(x)$ (and thus $GF(2^n)$) using standard-issue binary integers. The trick is to represent each polynomial $\sum a_i x^i$ by the integer value $a = \sum a_i 2^i$, so that each coefficient a_i is just the i-th bit of a. Adding two polynomials $a + b$ represented in this way corresponds to computing the bitwise exclusive or of a and b: `a^b` in programming languages that inherit their arithmetic syntax from C (i.e., almost everything except Scheme). Multiplying polynomials is more involved, although it's easy for some special cases like multiplying by x, which becomes a left-shift (`a<<1`) followed by XORing with the representation of our modulus if we get a 1 in the n-th place. (The general case is like this but involves doing XORs of a lot of left-shifted values, depending on the bits in the polynomial we are multiplying by.)

On the other hand, knowing that we can multiply $7 \equiv x^2+x+1$ by $5 \equiv x^2+1$ and get $6 \equiv x^2 + x$ quickly using C bit operations doesn't help us much if this product doesn't mean anything. For modular arithmetic, we at least have the consolation that $7 \cdot 5 = 6 \pmod{29}$ tells us something about remainders. In $GF(2^3)$, what this means is much more mysterious. This makes it useful—not in contexts where we want multiplication to make sense—but in contexts where we don't. These mostly come up in random number generation and cryptography.

5.1 Linear-feedback shift registers

Let's suppose we generate $x^0, x^1, x^2, \ldots$ in $\mathbb{Z}_2/(x^4 + x^3 + 1)$, which happens to be one of the finite fields isomorphic to $GF(2^4)$. Since there are only $2^4 - 1 = 15$ nonzero elements in $GF(2^4)$, we can predict that eventually this sequence will repeat, and in fact we can show that $p^{15} = 1$ for any nonzero p using essentially the same argument as for Fermat's Little Theorem. So we will have $x^0 = x^{15} = x^{30}$ etc. and thus will expect our sequence to repeat every 15 steps (or possibly some factor of 15, if we are unlucky).

To compute the actual sequence, we could write out full polynomials: $1, x, x^2, x^3, x^3 + 1, x^3 + x + 1, \ldots$, but this gets tiresome fast. So instead we'd like to exploit our representation of $\sum a_i x^i$ as $\sum a_i 2^i$.

Now multiplying by x is equivalent to shifting left (i.e. multiplying by 2) followed by XORing with 11001, the binary representation of $x^4 + x^3 + 1$, if we get a bit in the x^4 place that we need to get rid of. For example, we might do:

```
 1101 (initial value)
11010 (after shift)
 0011 (after XOR with 11001)
```

or

```
 0110 (initial value)
```

```
01100 (after shift)
 1100 (no XOR needed)
```

If we write our initial value as $r_3r_2r_1r_0$, the shift produces a new value $r_3r_2r_1r_00$. Then XORing with 11001 has three effects: (a) it removes a leading 1 if present; (b) it sets the rightmost bit to r_3; and (c) it flips the new leftmost bit if $r_3 = 1$. Steps (a) and (b) turn the shift into a rotation. Step (c) is the mysterious flip from our sequence generator. So in fact what our magic sequence generator was doing was just computing all the powers of x in a particular finite field.

As in $\mathbb{Z}_p$, these powers of an element bounce around unpredictably, which makes them a useful (though cryptographically very weak) pseudorandom number generator. Because high-speed linear-feedback shift registers are very cheap to implement in hardware, they are used in applications where a pre-programmed, statistically smooth sequence of bits is needed, as in the Global Positioning System and to scramble electrical signals in computers to reduce radio-frequency interference.

5.2 Checksums

Shifting an LFSR corresponds to multiplying by x. If we also add 1 from time to time, we can build any polynomial we like, and get the remainder mod m; for example, to compute the remainder of 100101 mod 11001 we do

```
 0000 (start with 0)
00001 (shift in 1)
 0001 (no XOR)
00010 (shift in 0)
 0010 (no XOR)
00100 (shift in 0)
 0100 (no XOR)
01001 (shift in 1)
 1001 (no XOR)
10010 (shift in 0)
 1011 (XOR with 11001)
10111 (shift in 1)
 1110 (XOR with 11001)
```

and we have computed that the remainder of $x^5 + x^3 + 1$ mod $x^4 + x^3 + 1$ is$x^3 + x^2 + x$.

This is the basis for cyclic redundancy check checksums, which are used to detect accidental corruption of data. The idea is that we feed our data stream into the LFSR as the coefficients of some gigantic polynomial, and the checksum summarizing the data is the state when we are done. Since it's unlikely that a random sequence of flipped or otherwise damaged bits would equal 0 mod m, most non-malicious changes to the data will be visible by producing an incorrect checksum.

5.3 Cryptography

$GF(2^n)$ can also substitute for $\mathbb{Z}_p$ in some cryptographic protocols. An example would be the function $f(s) = x^s \pmod{m}$, which is fairly easy to compute in $\mathbb{Z}_p$ and even easier to compute in $GF(2^n)$, but which seems to be hard to invert in both cases. Here we can take advantage of the fast remainder operation provided by LFSRs to avoid having to do expensive division in $\mathbb{Z}$.