WIKIPEDIA

Finite field arithmetic

In mathematics, **finite field arithmetic** is arithmetic in a finite field (a field containing a finite number of elements) as opposed to arithmetic in a field with an infinite number of elements, like the field of rational numbers.

While no finite field is infinite, there are infinitely many different finite fields. Their number of elements is necessarily of the form p^n where p is a prime number and n is a positive integer, and two finite fields of the same size are isomorphic. The prime p is called the characteristic of the field, and the positive integer n is called the dimension of the field over its prime field.

Finite fields are used in a variety of applications, including in classical coding theory in linear block codes such as BCH codes and Reed–Solomon error correction and in cryptography algorithms such as the Rijndael (AES) encryption algorithm.

Contents

Effective polynomial representation

The finite field with p^n elements is denoted GF(p^n) and is also called the **Galois Field**, in honor of the founder of finite field theory, Évariste Galois. GF(p), where p is a prime number, is simply the ring of integers modulo p. That is, one can perform operations (addition, subtraction, multiplication) using the usual operation on integers, followed by reduction modulo p. For instance, in GF(5), 4 + 3 = 7 is reduced to 2 modulo 5. Division is multiplication by the inverse modulo p, which may be computed using the extended Euclidean algorithm.

A particular case is GF(2), where addition is exclusive OR (XOR) and multiplication is AND. Since the only invertible element is 1, division is the identity function.

Elements of GF(p^n) may be represented as polynomials of degree strictly less than n over GF(p). Operations are then performed modulo R where R is an irreducible polynomial of degree n over GF(p), for instance using polynomial long division. The addition of two polynomials P and Q is done as usual; multiplication may be done as follows: compute $W = P \cdot Q$ as usual, then compute the remainder modulo R (there exist better ways to do this).

When the prime is 2, it is conventional to express elements of GF(p^n) as binary numbers, with each term in a polynomial represented by one bit in the corresponding element's binary expression. Braces ("{" and "}") or similar delimiters are commonly added to binary numbers, or to their hexadecimal equivalents, to indicate that the value is an element of a field. For example, the following are equivalent representations of the same value in a characteristic 2 finite field:

Polynomial: $x^6 + x^4 + x + 1$
Binary: {01010011}
Hexadecimal: {53}

Addition and subtraction

Addition and subtraction are performed by adding or subtracting two of these polynomials together, and reducing the result modulo the characteristic.

In a finite field with characteristic 2, addition modulo 2, subtraction modulo 2, and XOR are identical. Thus,

Polynomial: $(x^6 + x^4 + x + 1) + (x^7 + x^6 + x^3 + x) = x^7 + x^4 + x^3 + 1$
Binary: {01010011} + {11001010} = {10011001}
Hexadecimal: {53} + {CA} = {99}

Under regular addition of polynomials, the sum would contain a term $2x^6$. This term becomes $0x^6$ and is dropped when the answer is reduced modulo 2.

Here is a table with both the normal algebraic sum and the characteristic 2 finite field sum of a few polynomials:

p_1	p_2	$p_1 + p_2$ (normal algebra)	$p_1 + p_2$ in GF(2^n)
$x^3 + x + 1$	$x^3 + x^2$	$2x^3 + x^2 + x + 1$	$x^2 + x + 1$
$x^4 + x^2$	$x^6 + x^2$	$x^6 + x^4 + 2x^2$	$x^6 + x^4$
$x + 1$	$x^2 + 1$	$x^2 + x + 2$	$x^2 + x$
$x^3 + x$	$x^2 + 1$	$x^3 + x^2 + x + 1$	$x^3 + x^2 + x + 1$
$x^2 + x$	$x^2 + x$	$2x^2 + 2x$	0

In computer science applications, the operations are simplified for finite fields of characteristic 2, also called GF(2^n) Galois fields, making these fields especially popular choices for applications.

Multiplication

Multiplication in a finite field is multiplication modulo an irreducible reducing polynomial used to define the finite field. (I.e., it is multiplication followed by division using the reducing polynomial as the divisor—the remainder is the product.) The symbol "•" may be used to denote multiplication in a finite field.

Rijndael's finite field

Rijndael uses a characteristic 2 finite field with 256 elements, which can also be called the Galois field **GF**(2^8). It employs the following reducing polynomial for multiplication:

$$x^8 + x^4 + x^3 + x + 1.$$

For example, {53} • {CA} = {01} in Rijndael's field because

$$\begin{aligned}
&(x^6 + x^4 + x + 1)(x^7 + x^6 + x^3 + x) \\
&= (x^{13} + x^{12} + x^9 + \mathbf{x^7}) + (x^{11} + x^{10} + \mathbf{x^7} + x^5) + (x^8 + \mathbf{x^7} + x^4 + x^2) + (\mathbf{x^7} + x^6 + x^3 + x) \\
&= x^{13} + x^{12} + x^9 + x^{11} + x^{10} + x^5 + x^8 + x^4 + x^2 + x^6 + x^3 + x \\
&= x^{13} + x^{12} + x^{11} + x^{10} + x^9 + x^8 + x^6 + x^5 + x^4 + x^3 + x^2 + x
\end{aligned}$$

and

> $x^{13} + x^{12} + x^{11} + x^{10} + x^9 + x^8 + x^6 + x^5 + x^4 + x^3 + x^2 + x$ modulo $x^8 + x^4 + x^3 + x^1 + 1$ = (11111101111110 mod 100011011) = {3F7E mod 11B} = {01} = 1 (decimal), which can be demonstrated through long division (shown using binary notation, since it lends itself well to the task. Notice that exclusive OR is applied in the example and not arithmetic subtraction, as one might use in grade-school long division.):

```
         11111101111110 (mod) 100011011
        ^100011011     
          1110000011110
         ^100011011    
           110110101110
          ^100011011   
            10101110110
           ^100011011  
             0100011010
            ^00000000  
              100011010
             ^100011011 
               00000001
```

(The elements {53} and {CA} are multiplicative inverses of one another since their product is 1.)

Multiplication in this particular finite field can also be done using a modified version of the "peasant's algorithm". Each polynomial is represented using the same binary notation as above. Eight bits is sufficient because only degrees 0 to 7 are possible in the terms of each (reduced) polynomial.

This algorithm uses three variables (in the computer programming sense), each holding an eight-bit representation. **a** and **b** are initialized with the multiplicands; **p** accumulates the product and must be initialized to 0.

At the start and end of the algorithm, and the start and end of each iteration, this invariant is true: **a** **b** + **p** is the product. This is obviously true when the algorithm starts. When the algorithm terminates, **a** or **b** will be zero so **p** will contain the product.

- Run the following loop eight times (once per bit). It is OK to stop when **a** or **b** is zero before an iteration:
  1. If the rightmost bit of **b** is set, exclusive OR the product **p** by the value of **a**. This is polynomial addition.
  2. Shift **b** one bit to the right, discarding the rightmost bit, and making the leftmost bit have a value of zero. This divides the polynomial by **x**, discarding the x^0 term.
  3. Keep track of whether the leftmost bit of **a** is set to one and call this value **carry**.
  4. Shift **a** one bit to the left, discarding the leftmost bit, and making the new rightmost bit zero. This multiplies the polynomial by **x**, but we still need to take account of **carry** which represented the coefficient of x^7.
  5. If **carry** had a value of one, exclusive or **a** with the hexadecimal number `0x1b` (00011011 in binary). `0x1b` corresponds to the irreducible polynomial with the high term eliminated. Conceptually, the high term of the irreducible polynomial and **carry** add modulo 2 to 0.
- **p** now has the product

This algorithm generalizes easily to multiplication over other fields of characteristic 2, changing the lengths of **a**, **b**, and **p** and the value `0x1b` appropriately.

Multiplicative inverse

The multiplicative inverse for an element **a** of a finite field can be calculated a number of different ways:

- By multiplying **a** by every number in the field until the product is one. This is a Brute-force search.
- Since the nonzero elements of GF(p^n) form a finite group with respect to multiplication, $a^{p^n-1} = 1$ (for $a \neq 0$), thus the inverse of a is a^{p^n-2}.
- By using the extended Euclidean algorithm.
- By making a logarithm table of the finite field, and performing subtraction in the table. Subtraction of logarithms is the same as division.

Implementation tricks

When developing algorithms for Galois field computation on small Galois fields, a common performance optimization approach is to find a generator g and use the identity:

$$ab = g^{\log_g(ab)} = g^{\log_g(a)+\log_g(b)}$$

to implement multiplication as a sequence of table look ups for the $\log_g(x)$ and $g^{(x)}$ functions and an integer addition operation. This exploits the property that all finite fields contain generators. In the Rijndael field example, the polynomial x + 1 (or {03}) is one such generator. A necessary but not sufficient condition for a polynomial to be a generator is to be irreducible.

This same strategy can be used to determine the multiplicative inverse with the identity:

$$a^{-1} = g^{\log_g(a^{-1})} = g^{-\log_g(a)} = g^{|g|-\log_g(a)}$$

Here, the order of the generator, |g|, is the number of non-zero elements of the field. In the case of GF(2^8) this is $2^8-1 = 255$. That is to say, for the Rijndael example: $(x + 1)^{255} = 1$. So this can be performed with two look up tables and an integer subtract. Using this idea for exponentiation also derives benefit:

$$a^n = g^{\log_g(a^n)} = g^{n\log_g(a)} = g^{n\log_g(a)(mod|g|)}$$

This requires two table look ups, an integer multiplication and an integer modulo operation.

However, in cryptographic implementations, one has to be careful with such implementations since the cache architecture of many microprocessors leads to variable timing for memory access. This can lead to implementations that are vulnerable to a timing attack.

Program examples

C programming example

Here is some C code which will add, subtract, and multiply numbers in a finite field of characteristic 2^8, used for example by Rijndael algorithm or Reed–Solomon, using the Russian Peasant Multiplication algorithm:

```c
/* Add two numbers in a GF(2^8) finite field */
uint8_t gadd(uint8_t a, uint8_t b) {
    return a ^ b;
}

/* Subtract two numbers in a GF(2^8) finite field */
uint8_t gsub(uint8_t a, uint8_t b) {
    return a ^ b;
}

/* Multiply two numbers in the GF(2^8) finite field defined
 * by the polynomial x^8 + x^4 + x^3 + x + 1 = 0
 * using the Russian Peasant Multiplication algorithm
 * (the other way being to do carry-less multiplication followed by a modular reduction)
 */
uint8_t gmul(uint8_t a, uint8_t b) {
    uint8_t p = 0; /* the product of the multiplication */
    while (a && b) {
            if (b & 1) /* if b is odd, then add the corresponding a to p (final product = sum of all a's cor
                p ^= a; /* since we're in GF(2^m), addition is an XOR */

            if (a & 0x80) /* GF modulo: if a >= 128, then it will overflow when shifted left, so reduce */
                a = (a << 1) ^ 0x11b; /* XOR with the primitive polynomial x^8 + x^4 + x^3 + x + 1 (0b1_000
            else
                a <<= 1; /* equivalent to a*2 */
            b >>= 1; /* equivalent to b // 2 */
    }
    return p;
}
```

This example has cache, timing, and branch prediction side-channel leaks, and is not suitable for use in cryptography

D programming example

This D program will multiply numbers in Rijndael's finite field and generate a PGM image:

```
/**
Multiply two numbers in the GF(2^8) finite field defined
by the polynomial x^8 + x^4 + x^3 + x + 1.
*/
ubyte gMul(ubyte a, ubyte b) pure nothrow {
    ubyte p = 0;

    foreach (immutable ubyte counter; 0 .. 8) {
        p ^= -(b & 1) & a;
        auto mask = -((a >> 7) & 1);
        // 0b1_0001_1011 is x^8 + x^4 + x^3 + x + 1.
        a = (a << 1) ^ (0b1_0001_1011 & mask);
        b >>= 1;
    }

    return p;
}

void main() {
    import std.stdio, std.conv;
    enum width = ubyte.max + 1, height = width;

    auto f = File("rijndael_finite_field_multiplication.pgm" , "wb");
    f.writefln("P5\n%d %d\n255" , width, height);
    foreach (immutable y; 0 .. height)
        foreach (immutable x; 0 .. width) {
            immutable char c = gMul(x.to!ubyte, y.to!ubyte);
            f.write(c);
        }
}
```

This example does not use any branches or table lookups in order to avoid side channels and is therefore suitable for use in cryptography.

External links

- A description of Rijndael's finite field
- Fast Galois Field Arithmetic Library in C/C++
- Wikiversity: Reed–Solomon for Coders – Finite Field Arithmetic

Retrieved from "https://en.wikipedia.org/w/index.php?title=Finite_field_arithmetic&oldid=805795455"