Modular Arithmetic

Consider the following equivalence relation: let $A \equiv B (mod\, n)$ (pronounced "A is congruent to B modulo n") be true if and only if A and B have the same remainder when they are divided by n. In other words, (A-B) is divisible by n. This means that $A - B = kn$ for some integer k. For example, $32 \equiv 68 (mod\, 3)$, $5 \equiv -15 (mod\, 5)$ and $3 \equiv 7 (mod\, 1)$. Consider the *modulus*, n, fixed. Then we can simply write $A \equiv B$ if n is understood. The $\equiv$ relation is reflexive ($A \equiv A$ for all A), symmetric (if $A \equiv B$ then $B \equiv A$) and transitive (if $A \equiv B$ and $B \equiv C$ then $A \equiv C$). Therefore, it is an *equivalence relation* and it partitions the set of integers into a collection of equivalence classes. If you think of an infinite graph, where the vertices are all the possible integers and an edge exists between A and B whenever $A \equiv B$, then this graph will have exactly n connected components.

For example, let n=5. Then there are 5 equivalence classes. We will denote them [0], [1], [2], [3] and [4]. The set [0] is the set of all numbers divisible by 5. The set [1] contains all numbers that have remainder 1 when divided by 5, namely {1, -4, 6, -9, 11, -14, ...}. All 5 sets are disjoint – no two of them have any elements in common. The union of all 5 sets is the set of integers. For convenience, we will denote by [x] the equivalence class that contains x. In our example, [15]=[0], [51]=[1] and [-1] =[4].

What can we do with these classes? First of all, we can add and subtract them, just like normal integers. Take any member of [2] (for example, 12) and any member of [4] (for example, 4). Then 12+4=16, which belongs to [1], and 12-4=8, which belongs to [3]. In fact, [2]+[4]=[1] and [2]-[4] =[3]. This is true in general – addition and subtraction are always defined. Let us work modulo n. Now consider two (not necessarily different) equivalence classes [a] and [b] modulo n. Let $A \in [a]$ and $B \in [b]$. Then from the definition of the congruence relation, we know that $A - a = jn$ and $B - b = kn$ for some integers j and k. Adding the two equations gives $(A+B)-(a+b)=(j+k)n$ $j+k$ is an integer. Therefore, by the same definition, $A+B \equiv a+b$, so $A+B \in [a+b]$. Since A and B were arbitrary members of [a] and [b], addition of equivalence classes is well defined: $[a]+[b]=[a+b]$; similarly, $[a]-[b]=[a-b]$.

Multiplication works, too. If $A \in [a]$ and $B \in [b]$ then $A - a = jn$ and $B - b = kn$ for some integers j and k. Multiplying the two equations,

$$(A-a)\times(B-b) = jkn^2,$$
$$AB + ab - Ab - aB = jkn^2,$$
$$AB + (ab-ab) + ab - Ab - Ba = jkn^2,$$
$$AB - ab - a(B-b) - b(A-a) = jkn^2,$$
$$AB - ab = jkn^2 + a(kn) + b(jn),$$
$$AB - ab = (jkn + kn + jn)n.$$

(jkn+kn+jn) is an integer, so $AB \equiv ab (mod\, n)$, and we have shown that [a][b]=[ab].

From now on, instead of writing square brackets all the time, we will simply work with integers, but replace the equal sign (=) with the congruence sign $(\equiv)$.

In regular integers, division is a problem – an integer divided by another integer is not always an integer, and we can never divide by zero. In modular arithmetic, dividing by [0] is still forbidden, but a lot of other restrictions disappear, as long as we choose a "good" modulus n. In real numbers, in order to divide x by y, we need to multiply x by the inverse of y, which is 1/y. We will do the same in

modular arithmetic. Whenever we want to divide a by b, we will multiply a by the *modular inverse* of b.

A modular inverse of b is a number (in fact, an equivalence class) b^{-1} that obeys $b \times b^{-1} \equiv 1$, just like in real numbers. This means that we are searching for integers b^{-1} and k that solve the equation $bb^{-1} - 1 = kn$ or $bb^{-1} + (-n)k = 1$. From the Euclidean algorithm, we know that there exists a solution if and only if $gcd(b, n) = 1$. When this holds, the extended Euclidean algorithm gives us an infinite number of solutions, all of which differ by a multiple of n. Therefore, the inverse of b exists if and only if b is relatively prime to n, *i.e.* $gcd(b, n) = 1$. In particular, when n is prime, then every integer except 0 is relatively prime to n, so each equivalence class modulo n (except for [0]) has a corresponding inverse class.

For example, the inverse of 2 modulo 7 is 4, because $2 \times 4 = 8 \equiv 1 (mod\, 7)$. But, there is no inverse for 2 modulo 10 because no matter what we multiply 2 by, we can never get a number of the form 10k+1. Hence, if we are working modulo 10, we are not allowed to divide by 2.

Representations and the % operator

Each equivalence class [a] contains all the numbers that have the same remainder as a when divided by n. So, to represent each class with integers, we must use a canonical representation. This is usually done by taking a to be the smallest positive number within its class – *i.e.* we use the numbers 0, 1, ..., n-1 to represent the n classes. C++ and Java have the remainder operator (%) that does this, but it does not work as expected for negative numbers. Given b>0, b%n will give you the remainder in the range [0, n-1]. But, when b<0, b%n can be negative. For example, 5%4=1, but -5%4=-1, not 4. It is a good idea to always ensure that b is non-negative and n is positive when computing b%n.

Solving modular linear equations

Once we have division, we can solve equations of the form $Ax \equiv B (mod\, n)$. If A has an inverse modulo n, then we simply multiply both sides by the inverse and get the answer $x \equiv BA^{-1} (mod\, n)$. Otherwise, we know that A and n are not relatively prime; $gcd(A, n) = g \neq 1$. In this case, we have to write out the whole equation: $Ax - nk = B$ for some integers x and k. From last time, if g does not divide B, we have no solution. If g does divide B, then we can divide everything by g and get $(A/g)x - (n/g)k = (B/g)$, where A/g is relatively prime to n/g. This is another modular equation: $(A/g)x \equiv B/g (mod\, n/g)$, only this time we know that A/g has an inverse, so we can solve it as before. Therefore, when solving a general equation $Ax \equiv B (mod\, n)$, there is always a solution modulo n/g if $g = gcd(A, n)$ divides B, meaning that our solution is one of the equivalence classes [0], [1], ..., [n/g-1].

Modular linear equations are comletely equivalent to linear Diophantine equations (Ax+By=C) that we saw last time. There, we had two unknowns, x and y. Here we have one unknown, x, that we are trying to solve for, and one implicit unknown, k, which is completely determined once we know x. Hence we obtain a proof that when there is an infinite set of solutions to Ax+By=C, all the solutions differ by multiples of n/g, because each solution is within the same modular equivalence class with modulus n/g.

Powers modulo n

Implementing addition, subtraction and multiplication modulo n is easy. All we need is the long division algorithm to compute the remainder of a number after dividing by n. Division is a little more difficult – we need the extended Euclidean algorithm, and there are some situations in which we are not allowed to divide at all. The next step is taking powers. How do we compute the value (equivalence class) of $z^a (mod\, n)$?

First of all, let's recall something completely unrelated – Horner's rule for polynomial evaluation. When we have a polynomial of the form $a_0+a_1 x+a_2 x^2+...+a_n x^n$, where the $a_i's$ are known, we can evaluate it for any given x by proceeding righ-to-left. The trick is to rewrite the polynomial in the form $a_0+x(a_1+x(a_2+x(a_3+...+x(a_{n-1}+xa_n)...)))$, start with r=0 and proceed by executing the operations "times x," "plus a_n ", "times x," "plus a_{n-1} ", ..., "times x", "plus a_0 ". At the end, r is equal to the value of the polynomial at x. Here is a simple implementation.

```
int a[1024];     // contains the coefficients
int horner( int x, int n ) {
    int r = 0;
    for( int i = n; i >= 0; i-- ) {
        r *= x;
        r += a[i];
    }
    return r;
}
```

This way of evaluating has some advantages over the obvious method of taking powers of x, multiplying by the corresponding coefficient and adding. We are not interested in those right now. What we will do instead is replace all the additions with multiplications and all the multiplications with exponentiations. Also, we will set x=2 and all the a_i coefficients will be either 1 or z.

This gives us the following problem: compute the value of $a_0^{2^0} \times a_1^{2^1} \times a_2^{2^2} \times ... \times a_n^{2^n}$. Horner's rule still works, only now it is called **successive squaring** because instead of starting with r=0, we will start with r=1, and instead of repeating "times x, plus a_i ," we will do "raise to the power of x, times a_i ". Raising to the power of x is easy because x=2, so that is a simple squaring operation – hence, the name "successive squaring".

What does this algorithm compute? When given z, b, and n, let us first write $b=(b_0 b_1 ... b_n)_2$ as the binary representation of b (so the b_i are 0 or 1). Now, we see that $z^b = z^{(b_0 b_1 b_2 ... b_n)_2}$, which equals $z^{b_0} \times z^{b_1 2} \times z^{b_2 2^2} \times ... \times z^{b_n 2^n}$. Each of the z^{b_i} is 1 or z depending on whether b_i is 0 or 1, so setting $a_i = z^{b_i}$,we get back the equation in the above paragraph. So, this algorithm computes z^b . Unfortunately, z^b can be very big – certainly big enough to overflow any integer variable type that we might use to store it. Note, however, that we are interested in $z^b (mod\, n)$, and that we are only using a single operation throughout the algorithm – multiplication. This means we can work entirely in (mod n) and compute remainders of r as we go. Here is an implementation.

```
int powmod( int z, int b, int n ) {
    int r = 1;
    for( int i = (1<<30); i > 0; i >>= 1 ) {
        r *= r;
        r %= n;
        if( b & i ) {
            r *= z;
            r %= n;
        }
    }
    return r;
}
```

The function powmod(z,b,n) returns the remainder after dividing z^b by n. r starts off at 1 and gets successively squared and multiplied by z. The variable i ranges from 2^{30} to 2^0 and is used to check whether the corresponding bit in the exponent b is set. If it is, then we multiply r by z. Both lines where r is multiplied are followed by "`r %= n;`", which is meant to keep r between 0 and n and avoid overflow. The largest that r can ever get is $(n-1)^2$, assuming that z is also smaller than n.

We could also implement powmod() recursively, which looks a bit cleaner:

```
int powmod( int z, int b, int n ) {
    if( b == 0 ) return 1;
    int r = powmod( z, b / 2, n );
    r = r * r;
    return ( ( b % 2 == 1 ) ? ( r % n ) * z : r ) % n;
}
```

This version relies on the equations $z^a=(z^{a/2})^2$ if a is even and $z^a=(z^{a/2})^2 z$ if a is odd.

Modular Systems

We have explored how to add, subtract, multiply, divide, and even take powers with remainders. But, because the results of these operations are always within a finite set [0, n-1], there are many special properties that we don't get with rational or real numbers. With rationals, we can add, subtract, or multiply 2 numbers without problem, but we cannot divide by 0. Thus the rationals **excluding** 0 forms the **rational system**, where +,-,*,/ are well defined. In the realm of remainders, division poses problems not just for 0, as there are other numbers that we cannot divide by.

We saw above that we can divide by a number b (mod n) if and only if b^{-1} exists. But b^{-1} exists whenever gcd(b,n)=1. So, the **modular system** mod n, defined as the set of numbers that the operations +,-,*,/ are defined (mod n), is the set of remainders relatively prime to n. For example, when n=10, its modular system is {1,3,7,9}. When n=7, its modular system is {1,2,3,4,5,6}.

Each n has its unique modular system, and its size is fixed. The size of the modular system (mod n) is denoted $\phi(n)$, called Euler's *phi*-function (or totient function). Hence, $\phi(n)$ is also equal to the number of positive remainders less than n that are relatively prime to n. The phi function is an important one in Number Theory, and has many other applications. We explore how to compute it in a later section.

In a modular system, because division is well-defined, we obtain a useful operation that we take for granted in algebra – cancelling numbers. In real numbers, $ax=ay$ is equivalent to $x=y$ if

$a \neq 0$. The same does not hold in general for remainders, as $3 \times 2 \equiv 3 \times 5 (mod\, 9)$, but $2 \neq 5 (mod\, 9)$. In a modular system, because division is well defined, we do get cancellation – ie. if a, x, y are all in the modular system mod n, then $ax \equiv ay (mod\, n)$ if and only if $x \equiv y (mod\, n)$ because we can multiply both sides by the inverse of a. This has many useful ramifications, one of which is the following

Lemma 1. If $\{r_1, r_2, \ldots, r_{\phi(n)}\}$ is a modular system (mod n), and a is any number in the modulo system, then the list $\{ar_1, ar_2, \ldots, ar_{\phi(n)}\}$ is a permutation of $\{r_1, r_2, \ldots, r_{\phi(n)}\}$, and is also a modular system (mod n).

Proof. *With cancellation we can show that $ar_i \equiv ar_j$ if and only if $r_i \equiv r_j$ (mod n).* Q.E.D.

Multiplication is a way of *scrambling* numbers in a modular system, albeit a highly predictable one. Now what about more than one multiplications – ie. taking powers? Consider the number 7 given n=10, then repeatedly taking powers we get $7^1 \equiv 7, 7^2 \equiv 9, 7^3 \equiv 3, 7^4 \equiv 1$. So 7 has a period of 4. But we already know that the size of the modular system (mod 10) is 4, so the powers of 7 comprise of the entire modular system in some *scrambled* order. This doesn't always work out, as the period of 9 (mod 10) is 2 ($9^1 \equiv 9, 9^2 \equiv 1$). Knowing the period of any number is useful, but more important is to know which powers of a give us back a itself (i.e. for what q is $a^q \equiv a$).

Theorem 2. (Euler) For any number n , and any number a in the modular system mod n, we have

$$a^{\phi(n)} \equiv 1 (mod\, n) \quad .$$

Proof. *Take $\{r_1, r_2, \ldots, r_{\phi(n)}\}$ as a modular system (mod n). From Lemma 1, we have*

$$\prod a r_i \equiv \prod r_i \quad ,$$

which gives us

$$a^{\phi(n)} \prod r_i \equiv \prod r_i \quad ,$$
$$a^{\phi(n)} \equiv 1 \quad .$$

In this theorem, when n=p is a prime number, we see that $\phi(p) = p - 1$, and we obtain what is called **Femat's Little Theorem**, which states that $a^p \equiv a$ when p is prime.

This theorem shows that $a^{\phi(n)+1} \equiv a$ always holds, but as we've shown above this isn't always the smallest such power. On the other hand, $\phi(n)$ does not depend on the number a , so for **any number** a , its period **must divide** $\phi(n)$. We can use this to solve equations involving powers.

When we get equations like $ax \equiv b (mod\, n)$, the number a can vary with a period of n. So $3x \equiv b (mod\, 7)$ is equivalent to $24x \equiv b (mod\, 7)$, because both 3 and 24 are congruent mod 7. But when we take powers, it's a bit different. Suppose we are given the equation $a^x \equiv a^y (mod\, n)$, then what do we know about x and y? We cannot say $x \equiv y$ in this case, because for example $7^2 \equiv 9 (mod\, 10)$, but $7^{12} \equiv 1 (mod\, 10)$, and clearly $2 \equiv 12 (mod\, 10)$. The period of powers is not n anymore, but by Theorem 2 above, it is $\phi(n)$. Hence we obtain

Theorem 3. Suppose a is in the modular system (mod n). Then

$$a^x \equiv a^y (mod\, n) \Leftrightarrow x \equiv y (mod\, \phi(n)) \quad .$$