CS 70 Discrete Mathematics and Probability Theory
Fall 2018 Alistair Sinclair and Yun Song **Note 6**

1 Modular Arithmetic

In several settings, such as error-correcting codes and cryptography, we sometimes wish to work over a smaller range of numbers. Modular arithmetic is useful in these settings, since it limits numbers to a predefined range $\{0,1,\ldots,N-1\}$, and wraps around whenever you try to leave this range — like the hand of a clock (where $N = 12$) or the days of the week (where $N = 7$).

Example: Calculating the time When you calculate the time, you automatically use modular arithmetic. For example, if you are asked what time it will be 13 hours from 1 pm, you say 2 am rather than 14. Let's assume our clock displays 12 as 0. This is limiting numbers to a predefined range, $\{0,1,2,\ldots,11\}$. Whenever you add two numbers in this setting, you divide by 12 and provide the remainder as the answer.

If we wanted to know what the time would be 24 hours from 2 pm, the answer is easy. It would be 2 pm. This is true not just for 24 hours, but for any multiple of 12 hours (ignoring the detail of am/pm). What about 25 hours from 2 pm? Since the time 24 hours from 2 pm is still 2 pm, 25 hours later it would be 3 pm. Another way to say this is that we add 1 hour, which is the remainder when we divide 25 by 12.

This example shows that under certain circumstances it makes sense to do arithmetic within the confines of a particular number (12 in this example). That is, we only keep track of the remainder when we divide by 12, and when we need to add two numbers, instead we just add the remainders. This method is quite efficient in the sense of keeping intermediate values as small as possible, and we shall see in later lectures how useful it can be.

More generally, we can define $x \bmod m$ (in words: "x modulo m") to be the remainder r when we divide x by m. I.e., if $x \bmod m = r$, then $x = mq + r$ where $0 \leq r \leq m-1$ and q is an integer. Thus $29 \bmod 12 = 5$ and $13 \bmod 5 = 3$.

Computation

If we wish to calculate $x + y \bmod m$, we would first add $x + y$ and then calculate the remainder when we divide the result by m. For example, if $x = 14$ and $y = 25$ and $m = 12$, we would compute the remainder when we divide $x + y = 14 + 25 = 39$ by 12, to get the answer 3. Notice that we would get the same answer if we first computed $2 = x \bmod 12$ and $1 = y \bmod 12$ and added the results modulo 12 to get 3. The same holds for subtraction: $x - y \bmod 12$ is $-11 \bmod 12$, which is 1. Again, we could have directly obtained this by simplifying first, i.e., $(x \bmod 12) - (y \bmod 12) = 2 - 1 = 1$.

This is even more convenient if we are trying to multiply: to compute $xy \bmod 12$, we could first compute $xy = 14 \times 25 = 350$ and then compute the remainder when we divide by 12, which is 2. Notice that we get the same answer if we first compute $2 = x \bmod 12$ and $1 = y \bmod 12$ and simply multiply the results modulo 12.

More generally, while carrying out any sequence of additions, subtractions or multiplications $\bmod m$, we get the same answer if we reduce any intermediate results $\bmod m$. This can considerably simplify the calculations.

Set representation

There is an alternative view of modular arithmetic which helps understand all this better. For any integer m we say that x and y are *congruent modulo* m if they differ by a multiple of m, or in symbols,

$$x \equiv y \pmod{m} \;\Leftrightarrow\; m \text{ divides } (x-y).$$

For example, 29 and 5 are congruent modulo 12 because 12 divides $29-5$. We can also write $22 \equiv -2 \pmod{12}$. Notice that x and y are congruent modulo m iff they have the same remainder modulo m.

What is the set of numbers that are congruent to $0 \pmod{12}$? These are all the multiples of 12: $\{\ldots,-36,-24,-12,0,12,24,36,\ldots\}$. What about the set of numbers that are congruent to $1 \pmod{12}$? These are all the numbers that give a remainder 1 when divided by 12: $\{\ldots,-35,-23,-11,1,13,25,37,\ldots\}$. Similarly the set of numbers congruent to $2 \pmod{12}$ is $\{\ldots,-34,-22,-10,2,14,26,38,\ldots\}$. Notice in this way we get 12 such sets of integers, and every integer belongs to one and only one of these sets.

In general, if we work modulo m, then we get m such disjoint sets whose union is the set of all integers: these are often called *residue classes mod m*. We can think of each set as represented by the unique element it contains in the range $(0,\ldots,m-1)$. The set represented by element i would be all numbers z such that $z = mx+i$ for some integer x. Observe that all of these numbers have remainder i when divided by m; they are therefore congruent modulo m.

We can understand the operations of addition, subtraction and multiplication in terms of these sets. When we add two numbers, say $x \equiv 2 \pmod{12}$ and $y \equiv 1 \pmod{12}$, it does not matter which x and y we pick from the two sets, since the result is always an element of the set that contains 3. The same is true about subtraction and multiplication. It should now be clear that the elements of each set are interchangeable when computing modulo m, and this is why we can reduce any intermediate results modulo m.

Here is a more formal way of stating this observation:

Theorem 6.1. *If $a \equiv c \pmod{m}$ and $b \equiv d \pmod{m}$, then $a+b \equiv c+d \pmod{m}$ and $a \cdot b \equiv c \cdot d \pmod{m}$.*

Proof. We know that $c = a + k \cdot m$ and $d = b + \ell \cdot m$, so $c+d = a+k\cdot m+b+\ell\cdot m = a+b+(k+\ell)\cdot m$, which means that $a+b \equiv c+d \pmod{m}$. The proof for multiplication is similar and left as an exercise. $\square$

Exercise. Complete the proof of Theorem 6.1 for multiplication.

What this theorem tells us is that we can always reduce any arithmetic expression modulo m to a number in the range $\{0,1,\ldots,m-1\}$. As an example, consider the expresion $(13+11)\cdot 18 \bmod 7$. Using the above theorem several times we can write:

$$\begin{aligned}(13+11)\cdot 18 &\equiv (6+4)\cdot 4 \pmod{7}\\ &= 10\cdot 4 \pmod{7}\\ &\equiv 3\cdot 4 \pmod{7}\\ &= 12 \pmod{7}\\ &\equiv 5 \pmod{7}. \qquad (1)\end{aligned}$$

In summary, we can always do basic arithmetic (multiplication, addition, subtraction) calculations modulo m by reducing intermediate results modulo m. (Note that we haven't mentioned division: much more on that later!)

2 Exponentiation

Another standard operation in arithmetic algorithms (this is used heavily in primality testing and RSA) is raising one number to a power modulo another number. I.e., how do we compute $x^y \bmod m$, where x, y, m are natural numbers and $m > 0$? A naïve approach would be to compute the sequence $x \bmod m, x^2 \bmod m, x^3 \bmod m, \ldots$ up to y terms, but this requires time exponential in the number of bits in y. We can do much better using the trick of *repeated squaring*:

```
algorithm mod-exp(x, y, m)
  if y = 0 then return(1)
  else
    z = mod-exp(x, y div 2, m)
    if y mod 2 = 0 then return(z * z mod m)
    else return(x * z * z mod m)
```

This algorithm uses the fact that any $y > 0$ can be written as $y = 2a$ or $y = 2a + 1$, where $a = \lfloor \frac{y}{2} \rfloor$ (which we have written as `y div 2` in the above pseudo-code), plus the facts

$$\begin{aligned} x^{2a} &= (x^a)^2; \qquad \text{and} \\ x^{2a+1} &= x \cdot (x^a)^2. \end{aligned}$$

Exercise. Use the above facts to prove by induction that the algorithm always returns the correct value.

What is its running time? The main task here, as is usual for recursive algorithms, is to figure out how many recursive calls are made. But we can see that the second argument, y, is being (integer) divided by 2 in each call, so the number of recursive calls is exactly equal to the number of bits, n, in y. (The same is true, up to a small constant factor, if we let n be the number of decimal digits in y.) Thus, if we charge only constant time for each arithmetic operation (`div`, `mod` etc.) then the running time of `mod-exp` is $O(n)$. Note that this is *very* efficient: it means that we can handle exponents with (at least) thousands of bits!

In a more realistic model (where we count the cost of operations at the bit level), we would need to look more carefully at the cost of each recursive call. Note first that the test on y in the `if`-statement just involves looking at the least significant bit of y, and the computation of $\lfloor \frac{y}{2} \rfloor$ is just a shift in the bit representation. Hence each of these operations takes only constant time. The cost of each recursive call is therefore dominated by the mod operation[1] in the final result. A fuller analysis of such algorithms is performed in CS170.

3 Inverses

We have so far discussed addition, multiplication and exponentiation. Subtraction is the inverse of addition and just requires us to notice that subtracting b modulo m is the same as adding $-b \equiv m - b \pmod{m}$.

What about division? This is a bit harder. Over the reals dividing by a number x is the same as multiplying by $y = 1/x$. Here y is that number such that $x \cdot y = 1$. Of course we have to be careful when $x = 0$, since such a y does not exist. Similarly, when we wish to divide by $x \pmod{m}$, we need to find $y \pmod{m}$ such

[1] You can analyze grade-school long-division for binary numbers to understand how long a mod operation would take.

that $x \cdot y \equiv 1 \pmod{m}$; then dividing by x modulo m will be the same as multiplying by y modulo m. Such a y is called the *multiplicative inverse* of x modulo m. In our present setting of modular arithmetic, can we be sure that x has an inverse mod m, and if so, is it unique (modulo m) and can we compute it?

As a first example, take $x = 8$ and $m = 15$. Then $2x = 16 \equiv 1 \pmod{15}$, so 2 is a multiplicative inverse of 8 mod 15. As a second example, take $x = 12$ and $m = 15$. Then the sequence $\{ax \bmod m : a = 1,2,3,\ldots\}$ is periodic, and takes on the values $(12,9,6,3,0,12,9,6\ldots)$. [Exercise: check this!] Thus 12 *has no multiplicative inverse mod* 15 since the number 1 never appears in this sequence.

This is the first warning sign that working in modular arithmetic might actually be very different from grade-school arithmetic. Two weird things are happening. First, no multiplicative inverse seems to exist for a number that isn't zero. (In normal arithmetic, the only number that has no inverse is zero.) Second, the "times table" for a number that isn't zero has zero showing up in it! So, e.g., 12 times 5 is equal to zero when we are considering numbers modulo 15. (In normal arithmetic, zero never shows up in the multiplication table for any number other than zero.)

So, when *does* x have a multiplicative inverse modulo m? The answer is: if and only if the greatest common divisor of m and x is 1. Moreover, when the inverse exists it is unique. Recall that the *greatest common divisor* of two natural numbers x and y, denoted $\gcd(x,y)$, is the largest natural number that divides them both. For example, $\gcd(30,24) = 6$. If $\gcd(x,y)$ is 1, it means that x and y share no common factors (except 1). This is often expressed by saying that x and y are *relatively prime* or *coprime*.

Theorem 6.2. *Let m,x be positive integers such that $\gcd(m,x) = 1$. Then x has a multiplicative inverse modulo m, and it is unique (modulo m).*

Proof. Consider the sequence of m numbers $0,x,2x,\ldots(m-1)x$. We claim that these are all distinct modulo m. Since there are only m distinct values modulo m, it must then be the case that $ax \equiv 1 \bmod m$ for exactly one a (modulo m). This a is the unique multiplicative inverse.

To verify the above claim, suppose for contradiction that $ax \equiv bx \pmod{m}$ for two distinct values a,b in the range $0 \le b \le a \le m-1$. Then we would have $(a-b)x \equiv 0 \pmod{m}$, or equivalently, $(a-b)x = km$ for some integer k (possibly zero or negative).

However, x and m are relatively prime, so x cannot share any factors with m. This implies that $a-b$ must be an integer multiple of m. This is not possible, since $a-b$ ranges between 1 and $m-1$. □

Actually it turns out that $\gcd(m,x) = 1$ is also a *necessary* condition for the existence of an inverse: i.e., if $\gcd(m,x) > 1$ then x has no multiplicative inverse modulo m.

Exercise. Verify this claim, using a similar idea to that in the above proof. [Hint: Think about when zeros show up in multiplication tables.]

Since we know that multiplicative inverses are unique when $\gcd(m,x) = 1$, we shall write the inverse of x as $x^{-1} \pmod{m}$. Being able to compute the multiplicative inverse of a number is crucial to many applications, so ideally the algorithm used should be efficient. It turns out that we can use an extended version of Euclid's algorithm, which computes the gcd of two numbers, to compute the multiplicative inverse.

4 Computing Inverses: Euclid's Algorithm

Let us first discuss how computing the multiplicative inverse of x modulo m is related to finding $\gcd(x,m)$. For any pair of numbers x,y, suppose we could not only compute $\gcd(x,y)$, but also find integers a,b such that

$$d = \gcd(x,y) = ax + by. \tag{2}$$

(Note that this is not a modular equation; and the integers a,b could be zero or negative.) For example, we can write $1 = \gcd(35,12) = -1 \cdot 35 + 3 \cdot 12$, so here $a = -1$ and $b = 3$ are possible values for a,b.

If we could do this then we'd be able to compute inverses, as follows. We first find integers a and b such that

$$1 = \gcd(m,x) = am + bx.$$

But this means that $bx \equiv 1 \pmod{m}$, so b is a multiplicative inverse of x modulo m. Reducing b modulo m gives us the unique inverse we are looking for. In the above example, we see that 3 is the multiplicative inverse of 12 mod 35. So, we have reduced the problem of computing inverses to that of finding integers a,b that satisfy equation (2). Remarkably, Euclid's algorithm for computing gcd's also allows us to find the integers a and b described above. So computing the multiplicative inverse of x modulo m is as simple as running Euclid's gcd algorithm on input x and m!

Euclid's algorithm

If we wish to compute the gcd of two numbers x and y, how would we proceed? If x or y is 0, then computing the gcd is easy; it is simply the other number, since 0 is divisible by everything (although of course it divides nothing). The algorithm for other cases is ancient, and although associated with the name of Euclid, is almost certainly a folk algorithm invented by craftsmen (the engineers of their day) because of its intensely practical nature. This algorithm exists in cultures throughout the globe.

The algorithm for computing $\gcd(x,y)$ uses the following theorem to eventually reduce to the case where one of the numbers is 0:

Theorem 6.3. *Let $x \geq y > 0$. Then* $\gcd(x,y) = \gcd(y, x \bmod y)$.

Proof. The theorem follows immediately from the fact that a number d is a common divisor of x and y *if and only if* d is a common divisor of y and $x \bmod y$. To see this, if d divides divides both x and y, there exist integers z and z' such that $zd = x$ and $z'd = y$. Therefore $r = x - yq = zd - z'dq = (z - z'q)d$, and so d divides r. The other direction follows in exactly the same way. □

Given this theorem, let's see how to compute $\gcd(16,10)$:

$$\begin{aligned}
\gcd(16,10) &= \gcd(10,6) \\
&= \gcd(6,4) \\
&= \gcd(4,2) \\
&= \gcd(2,0) = 2
\end{aligned}$$

In each line, we replace the pair of arguments (x,y) with $(y, x \bmod y)$, until the second argument becomes 0. At this point the gcd is just the first argument. By the theorem, each of these substitions preserves the gcd.

This algorithm can be written recursively as follows. The algorithm assumes that the inputs are natural numbers x, y satisfying $x \geq y \geq 0$ and $x > 0$.

```
algorithm gcd(x,y)
  if y = 0 then return(x)
  else return(gcd(y,x mod y))
```

Theorem 6.4. *The algorithm above correctly computes the gcd of x and y.*

Proof. Correctness is proved by (strong) induction on y, the smaller of the two input numbers. For each $y \geq 0$, let $P(y)$ denote the proposition that the algorithm correctly computes $\gcd(x, y)$ for all values of x such that $x \geq y$ (and $x > 0$). Certainly $P(0)$ holds, since $\gcd(x, 0) = x$ and the algorithm correctly computes this in the `if`-clause. For the inductive step, we may assume that $P(z)$ holds for all $z < y$ (the inductive hypothesis); our task is to prove $P(y)$. The key observation here is that $\gcd(x,y) = \gcd(y, x \bmod y)$ — that is, replacing x by $x \bmod y$ does not change the gcd. This follows immediately from Theorem 6.3. Hence the `else`-clause of the algorithm will return the correct value provided the recursive call `gcd(y,x mod y)` correctly computes the value $\gcd(y, x \bmod y)$. But since $x \bmod y < y$, we know this is true by the inductive hypothesis! This completes our verification of $P(y)$, and hence the induction proof. □

What is the running time of this algorithm? We shall see that, in terms of arithmetic operations on integers, it takes time $O(n)$, where n is the total number of bits in the input (x, y). This is again very efficient. The argument for this fact will be similar to the one we used earlier for exponentiation, but slightly trickier: it is obvious that the arguments of the recursive calls become smaller and smaller (because $y \leq x$ and $x \bmod y < y$). The question is, how fast?

The key point we will prove is that, in the computation of $\gcd(x, y)$, after *two* recursive calls the first (larger) argument is smaller than x by at least a factor of two (assuming $x > 0$). (Note that we can't argue much about what happens in just one call.) There are two cases:

1. $y \leq \frac{x}{2}$. Then the first argument in the next recursive call, y, is already smaller than x by a factor of 2, and thus in the next recursive call it will be even smaller.
2. $x \geq y > \frac{x}{2}$. Then in two recursive calls the first argument will be $x \bmod y$, which is smaller than $\frac{x}{2}$.

So, in both cases the first argument decreases by a factor of at least two every two recursive calls. Thus after at most $2n$ recursive calls, where n is the number of bits in x, the recursion must stop. (Note that the first argument is always a natural number.)

Note that the above argument only shows that the *number of recursive calls* in the computation is $O(n)$. We can make the same claim for the running time if we assume that each call only requires constant time. Since each call involves one integer comparison and one mod operation, it is reasonable to claim that its running time is constant. In a more realistic model of computation, however, we should really make the time for these operations depend on the size of the numbers involved. This will be discussed in CS170.

Extended Euclid's algorithm

Recall that, in order to compute the multiplicative inverse, we need an algorithm which also returns integers a and b such that:

$$\gcd(x, y) = ax + by. \tag{3}$$

Then, in particular, when $\gcd(x,y) = 1$ we can deduce that b is an inverse of $y \bmod x$.

Now since this problem is a generalization of the basic gcd, it is perhaps not too surprising that we can solve it with a fairly straightforward extension of Euclid's algorithm.

The following recursive algorithm *extended-gcd* follows the same recursive structure as Euclid's original algorithm, but keeps track of the required coefficients a, b in equation (3) as the recursion unwinds. Specifically, the algorithm takes as input a pair of natural numbers $x \geq y$ as in Euclid's algorithm, and returns a triple of integers (d, a, b) such that $d = \gcd(x,y)$ and $d = ax + by$:

```
algorithm extended-gcd(x,y)
  if y = 0 then return(x, 1, 0)
  else
    (d, a, b) := extended-gcd(y, x mod y)
    return((d, b, a - (x div y) * b))
```

Exercise. Carefully hand-turn this algorithm on the input $(x,y) = (16,10)$ from our earlier example, and check that it delivers correct values for a, b. Does 10 have an inverse mod 16?

Exercise. Carefully hand-turn the algorithm on the input $(x,y) = (35,12)$, and verify that it returns the triple $(1,-1,3)$. Deduce the inverse of 12 mod 35, and check your answer.

Let's now look at why the algorithm works. In the base case ($y = 0$), the algorithm returns the gcd value $d = x$ as before, together with coefficients $a = 1$ and $b = 0$; clearly these satisfy $ax + by = d$, as required.

When $y > 0$, the algorithm first recursively computes values (d, a, b) such that $d = \gcd(y, x \bmod y)$ and

$$d = ay + b(x \bmod y). \tag{4}$$

It then returns the triple (d, A, B), where $A = b$ and $B = a - (a - \lfloor x/y \rfloor b)y)$. Just as in our earlier analysis of the vanilla gcd algorithm, we know that the value d computed recursively will be equal to $\gcd(x,y)$. So the first component of the triple returned by the algorithm is correct.

What about the other two components, A and B? From the specification of the algorithm, they should be integers that satisfy

$$d = Ax + By. \tag{5}$$

To figure out what A and B should be in terms of the previously returned values a and b, we can rearrange equation (4), as follows:

$$\begin{aligned} d &= ay + b(x \bmod y) \\ &= ay + b(x - \lfloor x/y \rfloor y) \\ &= bx + (a - \lfloor x/y \rfloor b)y. \end{aligned} \tag{6}$$

(In the second line here, we have used the fact that $x \bmod y = x - \lfloor x/y \rfloor y$ — check this!) Comparing this last equation with equation (5), we see that we need to take $A = b$ and $B = a - \lfloor x/y \rfloor b$. This is exactly what the algorithm does, and this is why it works!

Exercise. Turn the above argument into a formal proof by induction that the algorithm extended-gcd is correct.

Since the extended gcd algorithm has exactly the same recursive structure as the vanilla version, its running time will be the same up to constant factors (reflecting the increased time per recursive call). So once again the running time on n-bit numbers will be $O(n)$ arithmetic operations. This means that we can find multiplicative inverses very efficiently.

Division in modular arithmetic

Now that we know how to compute the inverse of x modulo m (assuming that x and m are coprime), how can we use it to do arithmetic? The simplest scenario is solving a modular equation such as the following:

$$8x \equiv 9 \pmod{15}. \tag{7}$$

Recall that the inverse of $8 \pmod{15}$ is 2 (since $2 \cdot 8 = 16 \equiv 1 \pmod{15}$). Hence we can multiply both sides of equation (7) by $8^{-1} \equiv 2$ to get

$$x \equiv 18 \equiv 3 \pmod{15}.$$

I.e., the solution to the modular equation (7) is $x = 3$, and this solution is unique modulo 15.