CS 70 Discrete Mathematics and Probability Theory
Summer 2017 Course Notes Note 6

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. 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 notes 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 $5 = 29 \bmod 12$ and $3 = 13 \bmod 5$.

Computation

If we wish to calculate $x+y \bmod m$, we would first add $x+y$ and the 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 as $2-1$ by first simplifying $x \bmod 12$ and $y \bmod 12$.

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 even if we reduce any intermediate results $\bmod m$. This can considerably simplify the calculations.

Set Representation

There is an alternate 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: $\{\dots,-36,-24,-12,0,12,24,36,\dots\}$. 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: $\{\dots,-35,-23,-11,1,13,25,37,\dots\}$. Similarly the set of numbers congruent to $2 \pmod{12}$ is $\{\dots,-34,-22,-10,2,14,26,38,\dots\}$. 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. We can think of each set as represented by the unique element it contains in the range $(0,\dots,m-1)$. 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. □

What this theorem tells us is that we can always reduce any arithmetic expression modulo m into a natural number smaller than m. As an example, consider the expresion $(13+11)\cdot 18 \bmod 7$. Using the above Theorem several times we can write:

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

In summary, we can always do basic arithmetic (multiplication, addition, subtraction, and division) calculations modulo m by reducing intermediate results modulo m.

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

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

As a useful exercise, you should use these facts to construct a formal inductive argument 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 $O(n)$.

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 170.

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[2]. 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 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

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

[2] Inverting exponentiation uses logarithms in the real numbers. The discrete logarithm is currently essentially impossible to compute efficiently. So we will not be talking about it.

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 $\{12,9,6,3,0,\}$ (check this!). Thus 12 *has no multiplicative inverse mod* 15 since the number 1 never appears in that sequence.

This is the first warning sign that working in modulo arithmetic might actually be a bit different than 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 thing you have to worry about is dividing by zero. Second, the "times table" for a number that isn't zero has zero showing up in it. So 12 times 5 is equal to zero when we are considering numbers modulo 15. For grade-school 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: iff 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 m 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 $ax = 1 \bmod m$ for exactly one a (modulo m). This a is the unique multiplicative inverse.

To verify the above claim, suppose that $ax \equiv bx \pmod{m}$ for two distinct values a,b in the range $0 \leq b \leq a \leq m-1$. 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. You might like to try to prove this 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 $x^{-1} \pmod{m}$, where the modulus is sometimes denoted as a subscript so $(x)_m^{-1}$ can also mean the same thing. 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.

Computing the Multiplicative Inverse

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

$$d = \gcd(x,y) = ax + by. \tag{1}$$

(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 the integers a and b such that

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

But this means that $bx \equiv 1 \pmod{m}$, so b is the 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 (1). 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 for computing the GCD

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[3]. 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 $x \geq y$ and let q,r be natural numbers such $x = yq + r$ and $r < y$. Then $\gcd(x,y) = \gcd(r,y)$.

Proof. This is because any common divisor of x and y is also a common divisor of y and r and vice versa. 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 $\gcd(16,10)$:

$16 = 10 \times 1 + 6$
$10 = 6 \times 1 + 4$
$6 = 4 \times 1 + 2$
$4 = 2 \times 2 + 0$
$2 = 0 \times 0 + 2$

In each line, we write the larger number x as $yq + r$, where y is the smaller number. The next line then replaces the larger number with y, and the smaller number with r. This preserves the gcd, as shown in the theorem above. Therefore, $\gcd(16,10) = \gcd(2,0) = 2$. Or if you wish you can stop a step earlier and say that the gcd is the last non-zero remainder: i.e. you can stop at the step $6 = 4 \times 1 + 2$, since at the next step the remainder is 0.

[3] This algorithm is used for figuring out a common unit of measurement for two lengths. You can imagine how this is extremely important for building something up from a scale model. Different lengths in a design can be expressed as integer multiples of a common length, and then a new measuring stick can be found for the scaled-up design. We will see how the algorithm itself can be executed without literacy or symbolic notation. It is fundamentally *physical* in its intuition and you should figure out how this can be executed using threads. In the homework, you will see how this algorithm reveals the secret hidden in plain sight within the Pentagram.

This algorithm can be written recursively as follows:

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

Note: This algorithm assumes that $x \geq y \geq 0$ and $x > 0$.

Let's go through a quick example of this recursive implementation of Euclid's algorithm. We wish to compute $\gcd(32,10)$:

$$\begin{aligned} \gcd(32,10) &= \gcd(10,2) \\ &= \gcd(2,0) \\ &= 2 \end{aligned}$$

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 is because a divisor d of y also divides x if and only if it divides $x \bmod y$ (divisibility by d is not affected by adding or subtracting multiples of d, and y *is* a multiple of d). 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. □

How long does this algorithm take? 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).

You should be able to see the intuitive connection to exponentiation-by-repeated-squaring. 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?

We shall show 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 $x > 0$). 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 will 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 170.

Extended Euclid's Algorithm

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

$$\gcd(x,y) = ax + by.$$

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.

Examples

Let's first see how we would compute such numbers for $x = 6$ and $y = 4$. We'll need the equations from our example above, copied here for reference:

$16 = 10 \times 1 + 6$
$10 = 6 \times 1 + 4$
$6 = 4 \times 1 + 2$
$4 = 2 \times 2 + 0$

From the last two equations it follows that $gcd(6,4) = 2$. But now the second last equation gives us the numbers a,b, since we just rearrange that equation to say $2 = 6 \times 1 - 4 \times 1$. So $a = 1$ and $b = -1$.

What if we started with $x = 10$ and $y = 6$? Now we would write the last three equations to determine that $gcd(10,6) = 2$. But how do we find a,b? Start as above and write $2 = 6 \times 1 - 4 \times 1$. But we want 10 and 6 on the right hand side, not 6 and 4. But notice that the third from the last equation allows us to write 4 as a linear combination of 6 and 10 and so we can just back substitute: we rewrite that equation as $4 = 10 \times 1 - 6 \times 1$ and substitute to get:
$2 = 6 \times 1 - 4 \times 1 = 6 \times 1 - (10 \times 1 - 6 \times 1) = 6 \times 2 - 10 \times 1.$

If we started with $x = 16$ and $y = 10$ we would back substitute again using the first equation rewritten as $6 = 16 - 10$ to get:
$2 = 6 \times 2 - 10 \times 1 = (16 - 10) \times 2 - 10 = 16 \times 2 - 10 \times 3$. So $a = 2$ and $b = -3$.

Algorithm

The following recursive algorithm *extended-gcd* implements the idea used in the examples above. It 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))
```

Note that this algorithm has the same form as the basic gcd algorithm we saw earlier; the only difference is that we now carry around in addition the required values a, b. You should hand-turn the algorithm on the input $(x, y) = (16, 10)$ from our earlier example, and check that it delivers correct values for a, b.

Let's now look at why the algorithm works. We just need to generalize the back substitution method we used in the example above.

In the base case ($y = 0$), we return the gcd value $d = x$ as before, together with values $a = 1$ and $b = 0$ which satisfy $ax + by = d$. If $y > 0$, we first recursively compute values (d, a, b) such that $d = \gcd(y, x \bmod y)$ and

$$d = ay + b(x \bmod y). \tag{2}$$

Just as in our analysis of the vanilla GCD algorithm, we know that this d 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? We need to update these values of a and b, say to A and B.

What should their values be? Well, from the specification of the algorithm, they must be integers that satisfy

$$d = Ax + By. \tag{3}$$

To figure out what A and B should be, we need to rearrange equation (2), as follows:

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

(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 (3), 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 the algorithm works. The ideas here can be made more formal to get a full proof of correctness.

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 efficiently.

Chinese Remainder Theorem

It is worth stepping back for a moment and looking at what the EGCD revealed to us. It said that the GCD could be expressed as $ax + by$ for two numbers x, y. To interpret this, we can imagine the number line, starting at zero and stretching out infinitely in both directions. Imagine that we are only allowed to take steps that are either x or y long. So, if $x = 5$ and $y = 7$, then we can either move to the right or left by 5 units or 7 units. Suppose we start at zero, and want to know everywhere we can reach by taking a sequence of such moves.

Intuitively, if we can reach a number z, we can reach any multiple of z by simply repeating the steps it took to get to z over and over again. The fact that we can execute the steps of the Euclid's GCD algorithm tells us that anything we can reach by taking steps of x and y must share all the common factors of x and y. This means that we can only reach any multiple of the GCD of x and y. The set of points that we can reach with such operations is called a “lattice” and this lattice-width interpretation of the GCD is interesting[4].

When the GCD is 1, it means that we can reach all points on the integer lattice in this manner. Those who have taken linear algebra will notice a very striking intellectual “rhyme” with the ideas of a basis and span. When their GCD is 1, it is as though the numbers x and y span all the integers[5]. The Chinese Remainder Theorem (CRT) can be interpreted as a way to make this interpretation even more striking.

Suppose we wanted to understand all the numbers mod pq where p and q are relatively prime to each other. If we had to arrange these numbers onto a sheet of paper, how would we do so? Going back to elementary school, it is natural to associate a product pq with a rectangle: p long on one side and q long on the other. So now, we know that we can place the pq numbers from 0 to $pq-1$ on this rectangle. But how? In what order? Given a number, how can you find its “x-coordinate” as something from $0,1,\ldots,p-1$ and its “y-coordinate” as something from $0,1,\ldots,q-1$? The natural first guess is to take a number z and just compute z mod p and z mod q to get two “coordinates” for z.

At this point, it is very useful to do a little exercise for yourself. Suppose $p=3$ and $q=5$ and just place all the numbers from 0 to 14 on this grid. You will see the coordinates as $0=(0,0),1=(1,1),2=(2,2),3=(0,3),4=(1,4),5=(2,0),6=(0,1),7=(1,2),8=(2,3),9=(0,4),10=(1,0),11=(2,1),12=(0,2),13=(1,3),14=(2,4)$. When writing them out, you will see that all the numbers lie on a diagonal line that wraps around the rectangle until it fills it. Notice that no two numbers from 0 to 14 have the same coordinates. Furthermore, notice that doing component-wise mod $(3,5)$ addition on the coordinates corresponds to doing mod 15 addition on the numbers themselves. Perhaps more interestingly, doing component-wise mod $(3,5)$ multiplication on the coordinates corresponds to doing mod 15 multiplication on the numbers themselves. (e.g. $3*4=12$ and $(0,3)*(1,4)\equiv(0,2)$). This means that operations can be equivalently performed component-wise in the tuple-representation.

Furthermore, we notice that there are two special tuples $(1,0)=10$ and $(0,1)=6$. The corresponding numbers act like “orthonormal basis elements” do in linear algebra. They provide an easy way to map from coordinates back to numbers. So (a,b) in coordinates represents the same number as $10a+6b$ mod 15. For example, $(2,1)\to 20+6=26\equiv 11 \pmod{15}$. So, not only can we easily move from numbers to coordinates (by just taking mods), we can also easily move from coordinates to numbers (by using these special basis elements). Before we state the general form of the Chinese Remainder Theorem, it is useful to observe that the basis element 10 corresponding the first coordinate (obtained by modding by 3) is a multiple of the other modulus 5. This has to be true because its representation in coordinates is designed to have a zero in that other coordinate. Similarly, 6 corresponds to the second coordinate (obtained by modding by 5) and is a multiple of 3.

With this example in hand, we are ready to generalize and to state the result more formally.

Chinese Remainder Theorem: Let $n_1,n_2,\ldots,n_k$ be positive integers that are coprime to each other. Then, for any sequence of integers a_i there is a unique integer x between 0 and $\prod_{i=1}^{k} n_i$ that satisfies the congru-

[4] This interpretation also makes short work of the classic family of puzzles of the form “you have a 5 oz cup and a 7 oz cup, an infinite reservoir of water, and a unlimited size mixing bowl. Can you manage to pour exactly z oz of water into a jar?” Do you see how such puzzles can be solved using EGCD?

[5] And when the GCD is 2, we can reach all even numbers. The even numbers behave in a way analogous to a subspace in linear algebra.

ences:

$$x \equiv a_1 \pmod{n_1} \tag{4}$$

$$\vdots \dots \tag{5}$$

$$x \equiv a_i \pmod{n_i} \tag{6}$$

$$\vdots \dots \tag{7}$$

$$x \equiv a_k \pmod{n_k} \tag{8}$$

Moreover this integer x can be found:

$$x = (\sum_{i=1}^{k} a_i b_i) \bmod N \tag{9}$$

where $N = \prod_{i=1}^{k} n_i$ and the "basis" numbers b_i are found using the formula $b_i = \frac{N}{n_i}(\frac{N}{n_i})_{n_i}^{-1}$ where $(\frac{N}{n_i})_{n_i}^{-1}$ denotes the multiplicative inverse (mod n_i) of the integer $\frac{N}{n_i}$.

Proof: The only question in being able to apply the formulas is to make sure that $(\frac{N}{n_i})_{n_i}^{-1}$ exists. To verify this, we first notice that $\frac{N}{n_i} = \prod_{j \neq i} n_j$ is a nonzero integer that is coprime to n_i since by construction, they can share no common factors. So the multiplicative inverse exists. This means that the formula is indeed computable and because it involves modding by N, it clearly gives rise to an x between 0 and $N-1$.

To see that this x solves the system of congruences, we need to take $x \bmod n_i$ and see what happens. First notice that $\frac{N}{n_r} = \prod_{j \neq r} n_j$ is congruent to 0 when we mod by $n_i \neq n_r$. This means that:

$$\begin{aligned}
x \bmod n_i &= ((\sum_{i=1}^{k} a_i b_i) \bmod N) \bmod n_i \\
&= (\sum_{i=1}^{k} a_i b_i) \bmod n_i \\
&= a_i b_i \bmod n_i \\
&= a_i(\frac{N}{n_i}(\frac{N}{n_i})_{n_i}^{-1}) \bmod n_i \\
&= a_i \bmod n_i
\end{aligned}$$

where the last quality used the definition of multiplicative inverse and the second equality used the fact that modding by a product and then by one of terms in that product is the same as just modding by that single term.

The above establishes that $x \equiv a_i \pmod{n_i}$ and so x does indeed solve the system of congruences. To see that it is unique, we have two arguments that we could use. The simplest argument is by counting. There are $N = \prod_{i=1}^{k} n_i$ possible values for the $(a_1, a_2, \ldots, a_k)$ tuples and the N numbers from 0 to $N-1$ each land in exactly one of these. If two landed in one bin, then that means that another bin must be empty. But we can construct an x corresponding to that bin and so it cannot be empty. This means that there must be a bijection from the coordinate tuples $(a_1, a_2, \ldots, a_k)$ and the N numbers from 0 to $N-1$.

Alternatively, suppose that some y also solves these congruences. Consider $z = y - x$. Clearly $z \bmod n_i$ is zero for all the n_i. This means that z is a multiple of n_i for each i and since they are all coprime, z is a multiple of N, their product. But the difference of two numbers ranging from 0 to $N-1$ must have an absolute value of at most $N-1$. This means that the only multiple of N that z can be is 0. This means that $y = x$ and so indeed, the given solution is unique. ♠

The Chinese Remainder Theorem (CRT) is a very powerful tool since it lets us move between numbers and their coordinates for the purpose of doing computations. Although stated for moduli that are all coprime, it can be extended to moduli n_i that are not coprime. However, in those cases, one has to be more careful. First, the range of numbers that we are interested in now is the Least-Common-Multiple (LCM) of the n_i values. Second, we must beware of inconsistent congruences. For example, we cannot simultaneously be congruent to $1 \pmod 2$ and be congruent to $2 \pmod 6$. In general, $a_i \equiv a_j \pmod{gcd(n_i, n_j)}$ must hold for a pair of congruences to be consistent[6]. You might be tempted to just use the formulas above with $N = LCM(n_1, n_2, \ldots, n_k)$, but that is not quite enough[7].

The homework has problems that will help you discover for yourself how the CRT can be very useful in solving problems.

[6] Since we can just mod both sides of both congruences by the GCD of n_i and n_j to get a congruence mod the GCD. If these two disagree, then the system of equations is clearly inconsistent.

[7] Instead, you can proceed by turning all congruences into statements about remainders mod prime powers. For every congruence that involves a composite modulus, just replace it with the equivalent system of congruences in terms of the prime-power factors of the modulus. By the regular CRT, these are equivalent to the original congruence. Once this has been applied to all the congruences, you simply have to discard redundant information. The rule is simple: keep only the congruence involving the largest power of any given prime. All the congruences for smaller powers are redundant. At this point, you have expressed the original congruences into a set of canonical congruences in terms of the prime factorization of the LCM of the original moduli.