CS 70 Discrete Mathematics and Probability Theory
Spring 2026 Course Notes

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. Assuming that our clock displays 12 as 0, this entails limiting numbers to the 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, after 25 hours 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 $x \equiv r \pmod{m}$, then $x = mq + r$ where $0 \leq r \leq m-1$ and q is an integer. Thus $29 \equiv 5 \pmod{12}$ and $13 \equiv 3 \pmod{5}$.

Computation

If we wish to calculate $x + y \pmod{m}$, 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, and get the answer 3. Notice that we would get the same answer if we first computed $2 \equiv x \pmod{12}$ and $1 \equiv y \pmod{12}$ and added the results modulo 12 to get 3. The same holds for subtraction: $x - y \pmod{12}$ is $-11 \pmod{12}$, which is 1. Again, we could have obtained this directly by simplifying first, i.e., $(x \pmod{12}) - (y \pmod{12}) \equiv 2 - 1 \equiv 1 \pmod{12}$.

This idea saves us even more effort with multiplication: to compute $xy \pmod{12}$, 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 much more easily if we first compute $2 \equiv x \pmod{12}$ and $1 \equiv y \pmod{12}$ and simply multiply the results modulo 12.

More generally, while carrying out any sequence of additions, subtractions or multiplications mod m, we get the same answer if we reduce any intermediate results mod m. This can considerably simplify the calculations. We will formally prove the correctness of this claim in Theorem 6.1 below.

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,

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

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; in other words,

$$x \equiv y \pmod{m} \iff x \pmod{m} = y \pmod{m}.$$

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 $z = mq + i$ for some integer q. 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 $d = b + \ell \cdot m$ for integers k, ℓ, 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. □

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 expression $(13+11) \cdot 18 \pmod{7}$. 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) 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 (used heavily, e.g., in primality testing and RSA) is raising one number to a power modulo another number. I.e., how do we compute $x^y \pmod m$, where x, y, m are natural numbers and $m > 1$? A naïve approach would be to compute the sequence $x \pmod m, x^2 \pmod m, x^3 \pmod m, \ldots$ 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

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

Exercise. Use the above facts to prove by induction on y 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 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 may want to analyze grade-school long-division for binary numbers to understand how long a mod operation would take. Since all arithmetic is being done mod m, the cost of this operation depends only on the number of bits in m and x (and not on y).

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 $\{ax \pmod{m} : a = 1, 2, 3, \ldots\}$ 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, there can be non-zero numbers that have no inverse. (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 $ax \equiv 1 \pmod{m}$ for exactly one $a \pmod{m}$[2] This a is the unique multiplicative inverse of x.

To verify the above claim, suppose for contradiction 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).

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. [Hint: Assume for contradiction that x does have an inverse, say a. Then write down a (non-modular) equation that x, m and a must satisfy. Finally, derive a contradiction from the fact that x and m have a common factor $d > 1$.]

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.

[2] Another way of expressing this property is that the operation of multiplication by x is a *bijection* between the set of integers mod m and itself. I.e., multiplication by x maps each integer mod m to a *distinct* integer mod m.

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

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

(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 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 (1). Remarkably, Euclid's algorithm for computing gcd's also allows us to find integers a and b as 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? As a base case, 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 folklore algorithm invented by craftsmen (the engineers of their day) because of its intensely practical nature

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

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

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 $x \pmod{y}$. To see this, write $x = qy + r$ where q is an integer and $r = x \pmod{y}$. Then, if d divides x and y then it also divides x and qy, and thus it also divides their difference $r = x - qy$ (as we proved in Note 1). Conversely, if d divides y and r then it also divides qy and r and thus also their sum $x = qy + r$. □

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 $(y, x \pmod{y})$, 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 $\gcd(x,y) = \gcd(y, x \pmod y)$ — that is, replacing x by $x \pmod y$ does not change the gcd. This was proved in 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 $\gcd(y, x \pmod y)$. But since $x \pmod y < y$, 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 $x \pmod y < y$). 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 $x \pmod y$, 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:

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

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 (2) 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))
```

In this algorithm, x div y denotes the usual truncated integer division, written mathematically as $\lfloor x/y \rfloor$.

Here is the sequence of recursive calls (top row) along with the sequence of triples that they return (bottom row) for the same input $(x, y) = (16, 10)$ as in our previous gcd example:

$$\begin{array}{ccccccccc} \text{egcd}(16,10) & \longrightarrow & \text{egcd}(10,6) & \longrightarrow & \text{egcd}(6,4) & \longrightarrow & \text{egcd}(4,2) & \longrightarrow & \text{egcd}(2,0) \\ (2,2,-3) & \longleftarrow & (2,-1,2) & \longleftarrow & (2,1,-1) & \longleftarrow & (2,0,1) & \longleftarrow & (2,1,0) \end{array}$$

Exercise. Check the above execution sequence.

The final triple returned is $(d, a, b) = (2, 2, -3)$, meaning that the gcd of $(x, y) = (16, 10)$ is $d = 2$, and that it can be expressed as $d = ax + by = 2 \cdot 16 - 3 \cdot 10$, which we can easily see is correct. (In fact, the sequence of triples (d, a, b) returned each expresses $d = 2$ as a linear combination of the corresponding inputs to that recursive call: so we can check that $d = 1 \cdot 2 + 0 \cdot 0 = 0 \cdot 4 + 1 \cdot 2 = 1 \cdot 6 - 1 \cdot 4 = -1 \cdot 10 + 2 \cdot 6 = 2 \cdot 16 - 3 \cdot 10$.) Since $\gcd(16, 10) \neq 1$, 10 doesn't have an inverse mod 16, so the coefficients a, b returned by the algorithm are not useful to us for computing inverses. However, the next exercise suggests an example where they allow us to read off the inverse, as described earlier.

Exercise. Hand-turn the algorithm yourself on the input $(x, y) = (35, 12)$ and verify that it returns the triple $(1, -1, 3)$. Deduce that the inverse of 12 mod 35 is 3.

Let's now reverse-engineer the algorithm and understand why it works and why it was designed this way. 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 $d = \gcd(y, x \pmod y)$ and

$$d = ay + b(x \pmod y). \tag{3}$$

It then returns the triple (d, A, B), where $A = b$ and $B = a - \lfloor x/y \rfloor b$. 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

$$d = Ax + By. \tag{4}$$

To figure out what A and B should be in terms of the previously returned values a and b, we can rearrange (3), as follows:

$$\begin{aligned} d &= ay + b(x \pmod y) \\ &= ay + b(x - \lfloor x/y \rfloor y) \\ &= bx + (a - \lfloor x/y \rfloor b)y. \end{aligned} \tag{5}$$

(In the second line here, we have used the fact that $x \pmod y = x - \lfloor x/y \rfloor y$ — check this!) Now compare (5) with (4): comparing coefficients of x and y, we see that we need to take $A = b$ and $B = a - \lfloor x/y \rfloor b$. But this is exactly what the last line of 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:

$$8x \equiv 9 \pmod{15}. \tag{6}$$

To solve the analogous equation $8x = 9$ over the rational numbers, we would multiply both sides by 8^{-1} to get $x = 9/8$. Let's do the same thing in arithmetic mod 15. 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 (6) by $8^{-1} \equiv 2$ to get

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

i.e., the solution to the modular equation in (6) is $x = 3$, and this solution is unique modulo 15.

5 Fundamental Theorem of Arithmetic

You may recall that any positive integer greater than 1 can be expressed uniquely as a product of primes, up to a reordering of the factors. This is known as the Fundamental Theorem of Arithmetic. For example, $12 = 2 \cdot 2 \cdot 3$. How do we know that this is true? It turns out that Extended Euclid's Algorithm is the key to proving this!

We'll start by proving an important fact about divisibility.

Claim: Let x, y, and z be positive integers such that $\gcd(x,y) = 1$. If $x \mid yz$, then $x \mid z$.

Proof: By Extended Euclid's algorithm, since $\gcd(x,y) = 1$, there exists integers a and b such that

$$1 = \gcd(x,y) = ax + by.$$

Multiplying both sides by z gives

$$z = axz + byz.$$

We know that $x \mid axz$, and $x \mid byz$ because $x \mid yz$. Thus, x divides their sum, or in other words, $x \mid axz + byz$. Since $axz + byz = z$, we conclude that $x \mid z$. □

Recall that a prime number p is a number whose only positive factors are 1 and p.

Fundamental Theorem of Arithmetic: Every positive integer $n > 1$ can be expressed uniquely in the form $p_1 p_2 \cdots p_k$, where each p_i is a (not necessarily unique) prime number, up to reordering of the prime factors.

For example, we can write $12 = 2 \cdot 2 \cdot 3$. Any other prime factorization of 12 is some reordering of $2, 2$, and 3.

Proof: We showed by induction in Note 3 that every positive integer n can be expressed in the form $p_1 p_2 \cdots p_k$, where each p_i is a (not necessarily different) prime number. Thus, it suffices to show that such a form is unique up to reordering the factors.

In other words, suppose we can express n with two prime factorizations

$$n = p_1 p_2 \cdots p_k = q_1 q_2 \cdots q_l,$$

where the p_i and q_j are prime numbers. We need to show that $k = l$, and the p_i's are some reordering of the q_j's.

Without loss of generality, assume that $k \leq l$. (If $k \geq l$, we can simply swap the factorizations.) Our goal is to show that each p_i is one of the q_j's.

First, consider p_1. Since $p_1 \mid n$, we must have $p_1 \mid q_1 q_2 \cdots q_l$. If p_1 is one of $q_1, q_2, \ldots, q_{l-1}$, then we are done. Otherwise, if $p_1 \neq q_j$, for $1 \leq j \leq l-1$, then $\gcd(p_1, q_j) = 1$ for $1 \leq j \leq l-1$, as the only factor they share in common is 1. Applying the previous claim, we conclude that $p_1 \mid q_l$. Since 1 is not prime, p_1 is not 1, so $p_1 = q_l$.

Thus, we conclude that $p_1 = q_j$ for some j. We can thus divide both sides by p_1, reducing the number of primes on both sides by 1.

We now repeat the process for p_2, p_3, all the way to p_k. In the end, the left hand side will be 1, and the right hand side will be a product of $l - k$ primes. Every prime number is at least 2, so for the two sides to be equal, we must have $l - k = 0$, or $k = l$. Moreover, in this process, we have matched each p_i with a unique q_j. Thus, the p_i's are some reordering of the q_j's, as desired. □

What were the keys to the above proof? Well, to prove that a prime factorization existed (something we showed in Note 3), we used strong induction. To prove that such a factorization is unique, we used a corollary of the Euclidean algorithm. The key to the Euclidean algorithm is the existence and uniqueness of division with remainder; in other words, for integers x and $m \neq 0$, there exists unique integers q and r such that $x = mq + r$ with $0 \leq r \leq m - 1$. If we have other arithmetic systems that emulate this type of division algorithm, we can follow the same type of proof to get a form of unique prime factorization. This is an important generalization in number theory.[3]

6 Chinese Remainder Theorem.

To conclude this Note, we present an interesting aspect of modular arithmetic which is embodied in the Chinese Remainder Theorem, so named because its earliest known statement was by the Chinese mathematician Sunzi in the 3rd century AD.

[3] If you are interested in seeing more details, take a look at https://en.wikipedia.org/wiki/Euclidean_domain.

The simplest case of the theorem is as follows:

Claim: For n_1, n_2 with $gcd(n_1, n_2) = 1$, there is exactly one $x \pmod{n_1 n_2}$ that satisfies the equations:

$$x \equiv a_1 \pmod{n_1} \quad \text{and} \quad x \equiv a_2 \pmod{n_2}. \tag{7}$$

Proof: Since $\gcd(n_1, n_2) = 1$, we know by Extended Euclid that there exist integers b_1, b_2 such that

$$b_1 n_1 + b_2 n_2 = 1. \tag{8}$$

Now we claim that the integer $x = a_1 b_2 n_2 + a_2 b_1 n_1$ satisfies the equations (7). To check this, we can write

$$\begin{aligned} x &= a_1 b_2 n_2 + a_2 b_1 n_1 \\ &= a_1(1 - b_1 n_1) + a_2 b_1 n_1 \\ &= a_1 + (a_2 - a_1) b_1 n_1. \end{aligned}$$

Hence we see that $x \equiv a_1 \pmod{n_1}$, as required.

By exactly the same argument, switching the subscripts 1 and 2, we see also that $x \equiv a_2 \pmod{n_2}$. Hence x does indeed satisfy both the equations (7).

To complete the proof, we need to verify that x is unique $\pmod{n_1 n_2}$. To see this, suppose that integers x and y both satisfy the equations (7). Thus in particular $x \equiv y \pmod{n_1}$ and $x \equiv y \pmod{n_2}$. This implies that $(x - y)$ is divisible by n_1 and also by n_2. But since $\gcd(n_1, n_2) = 1$, this means that $(x - y)$ must be divisible by $n_1 n_2$, and hence $x \equiv y \pmod{n_1 n_2}$, as required. □

Looking back at the above proof, note that the integers b_1, b_2 that show up in the definition of x actually have a specific meaning as inverses. Specifically, recall from our discussion of Extended Euclid that in equation (8), b_1 is the inverse of $n_1 \pmod{n_2}$. And similarly, b_2 is the inverse of $n_2 \pmod{n_1}$. (Both of these inverses exist because $\gcd(n_1, n_2) = 1$.) Thus the solution x that we constructed in the proof has the form

$$x = a_1 u_1 + a_2 u_2,$$

where $u_1 = n_2(n_2^{-1} \pmod{n_1})$ and $u_2 = n_1(n_1^{-1} \pmod{n_2})$. Note that this implies

$$u_1 \equiv \begin{cases} 1 \pmod{n_1} \\ 0 \pmod{n_2} \end{cases} \qquad u_2 \equiv \begin{cases} 0 \pmod{n_1} \\ 1 \pmod{n_2} \end{cases}$$

Hence we can view u_1, u_2 as "basis vectors", in the sense that u_1 has coordinate 1 in "direction" n_1 and coordinate 0 in "direction" n_2, and vice versa for u_2. Thus in order to satisfy the equations (7), our solution x should be a linear combination of u_1, u_2, with coefficients a_1 and a_2, respectively.

The above result can readily be extended to the full Chinese remainder theorem, which is stated as follows.

Chinese Remainder Theorem: Let $n_1, n_2, \ldots, n_k$ be positive integers that are coprime to each other, and let $N = \prod_{i=1}^{k} n_i$. Then for any sequence of integers $a_1, a_2, \ldots, a_k$, there is a unique integer $x \pmod{N}$ that satisfies all the congruences

$$\begin{cases} x \equiv a_1 \pmod{n_1} \\ \vdots \equiv \vdots \\ x \equiv a_i \pmod{n_i} \\ \vdots \equiv \vdots \\ x \equiv a_k \pmod{n_k} \end{cases}$$

We won't prove the full theorem in detail here, but we will sketch two approaches.

The first approach is to iterate the result of the $k = 2$ version of the theorem that we proved above. Namely, we first use that versoin to find an x that satisfies the first two congruences (for moduli n_1 and n_2). This x has a certain (unique) value $\pmod{n_1 n_2}$, which we will call a_{12}. Now we can replace the first two congruences by the single congruence $x \equiv a_{12} \pmod{n_1 n_2}$. Note that $n_1 n_2$ is also coprime with all the other n_i. We now have a smaller problem with $k-1$ rather than k congruences, and we can continue to iterate until we are finished.

The second approach is to extend the "coordinate" view of our earlier proof for $k = 2$. I.e., we construct basis vectors u_i for each i using inverses as before, and then derive the solution $x = \sum_i a_i u_i$ as a linear combination of these basis vectors. Following our earlier construction, the correct definition for each u_i should be $u_i = \frac{N}{n_i}((\frac{N}{n_i})^{-1} \pmod{n_i})$. If you are interested, you may like to check that this solution works as before.