LESSON: RA1: Modular Arithmetic

***

(Slide 1) Randomized Algorithms

RANDOMIZED ALGORITHMS

Now we'll dive into randomized algorithms. Hopefully at the end, you'll appreciate how randomness is a beautiful and powerful algorithmic tool.

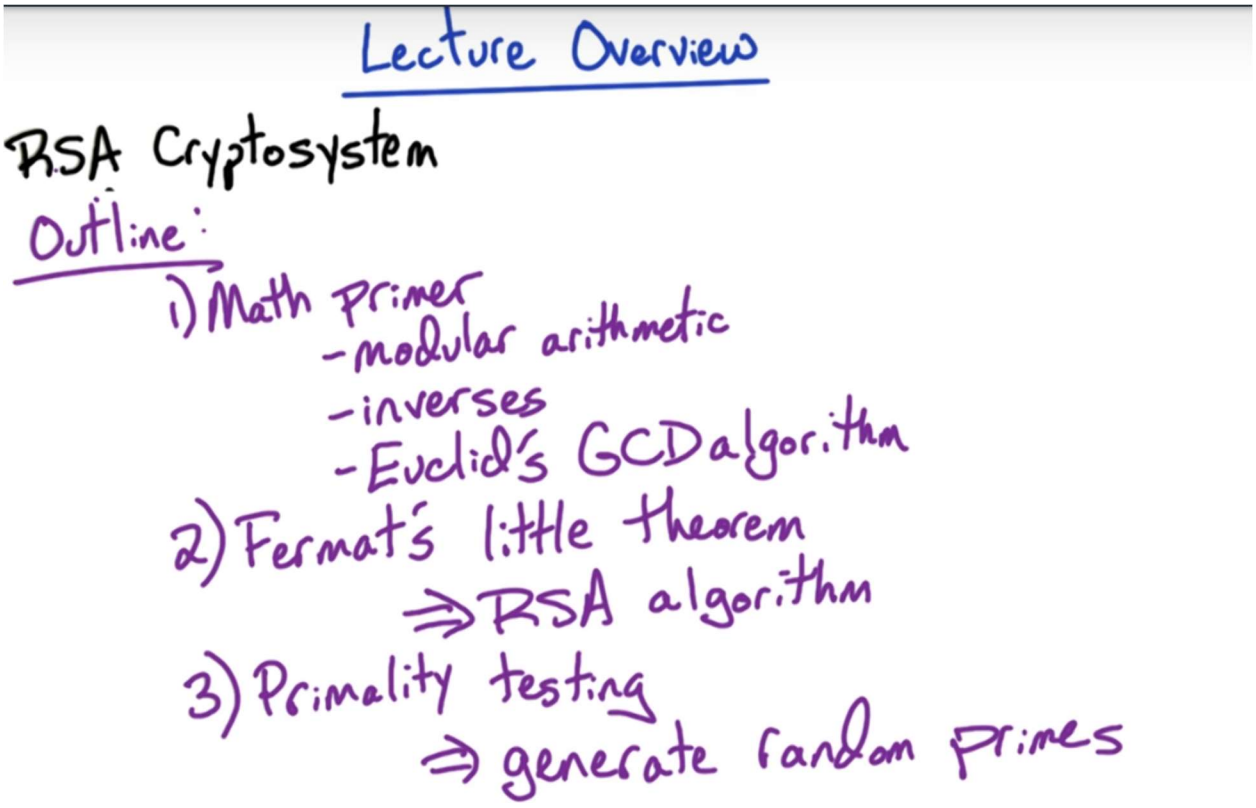
We'll start with cryptography. We'll look at the amazing RSA Cryptosystem. This is one of the most widely used Cryptosystems. It's extremely elegant. Once we learn some of the basic mathematics about modular arithmetic, you'll appreciate the ingenuity of the RSA protocol. Then you'll have a basic understanding of how these Cryptosystem that we use everyday, many times a day in fact, actually work.

Another beautiful and incredibly useful application of randomized algorithms that we'll study is for hashing. We're going to look at a hashing scheme known as Bloom Filters. This is a simple scheme that's quite popular in many different fields.

We'll look at the mathematics behind it. This will involve some basic probability analysis and then you'll do a programming project to implement and study Bloom Filters.

***

(Slide 2) RSA: Lecture Overview



Let me give you an overview of the topics that we're gonna study in this lecture.

Now the mathematics of the RSA cryptosystem are very beautiful and they're fairly simple once you have the right mathematical background. So we're going to start with a short primer on the mathematical topics that we need.

- The relevant topic that we need is Modular Arithmetic. Some of you may have seen this before, so it might be a review; for others, it's fairly simple to learn. One key concept that we need is multiplicative inverses, and what they mean in modular arithmetic.

- And then we're gonna look at Euclid's GCD algorithm, greatest common divisor, which is gonna be used in the RSA cryptosystem.

- Next, we're going to learn about Fermat's Little Theorem and it's the key tool in the design of the RSA algorithm.

- So. at this point we'll be able to detail the RSA algorithm.

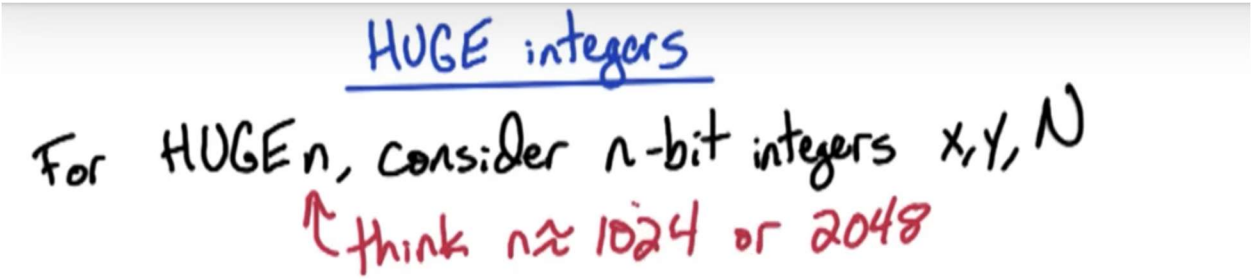
- Finally, we'll look at Primality testing, given a number, can we test whether the number is prime or not, or composite? Once we see how to do that using Fermat's Little Theorem, then we're going to be able to generate random primes.

Generating random primes is a key component in the RSA algorithm. So once we complete that, we'll have completed our discussion of the RSA algorithm.

Now let's go ahead and dive into the algorithms related to the RSA.

***

(Slide 3) Huge Integers



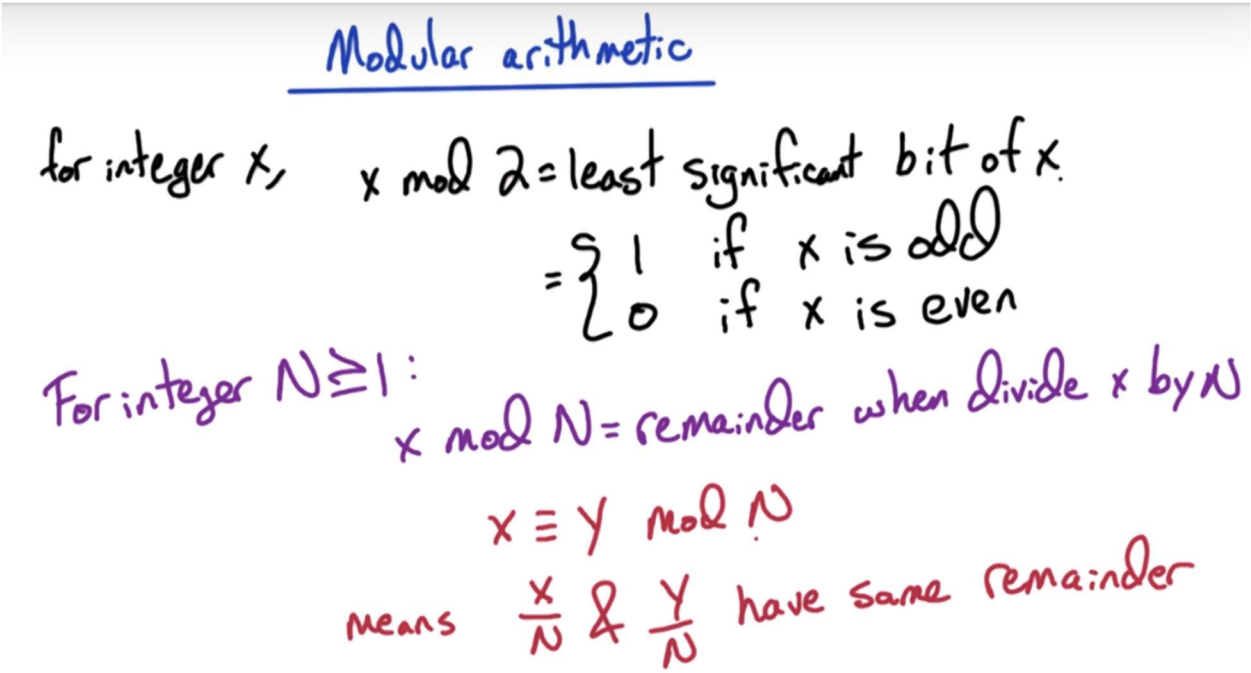
Let's look at the context of the RSA algorithm.

In cryptography, we're going to work with n-bit numbers x, y and N. For example, the number of bits in these numbers is going to be huge. Think of the number of bits as 1024 or 2048. It's a huge number of bits. So normally, we thought of our arithmetic operations as built into hardware, but that usually restricts the attention to 64 bits or so. But here we're talking about 1000 or 2000 bits.

So, we have to go back and review how long does it take for basic arithmetic operations.

***

(Slide 4) Modular Arithmetic



Now let's take a look at modular arithmetic. This is the basic mathematics that underlies the RSA algorithm and we're going to see a lot of beautiful mathematics along the way.

Let's look at a simple example: x mod 2, where x is an integer. What is x mod 2? - this is the least significant bit of x.

What does the least significant bit of x tell you? It tells you whether x is odd or even. In particular, it's 1, if x is odd, and 0, if x is even.

Now, what's another way of looking at this least significant bit of x? Well, we can take x/2. If it's even, then the remainder is going to be 0 because it's a multiple of two; and, if it's odd, then the remainder when x is divided by two is going to be 1.

So we looked at x mod 2. Now, let's look at x mod N where N is an arbitrary integer at least 1. Recall: x mod 2 is the remainder when we divide x by 2. So, x mod N is going to be the remainder when we divide x by n.

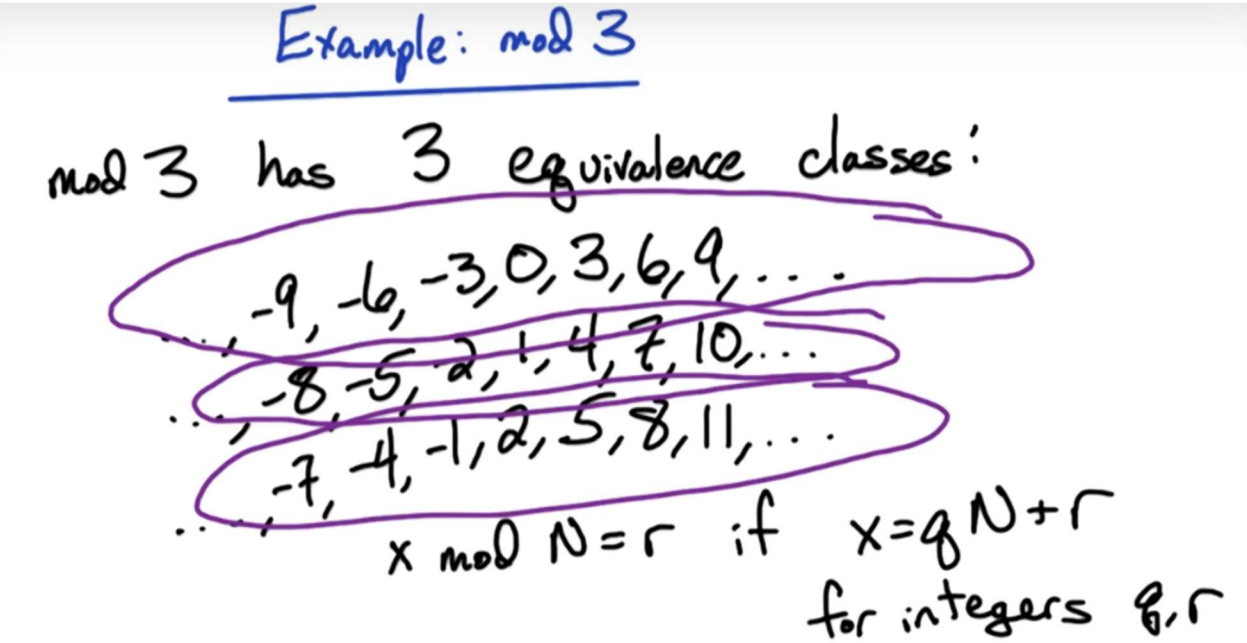
Finally, let's look at some important notation for modular arithmetic.

- Suppose we have two numbers x and y, and we look at these two numbers mod N, and suppose that they're same mod N, how do we denote that? - well, they're not equal - the two numbers are not equal - but they're congruent in this, or they're equivalent in, this wold modular N.

  So how do we denote that equivalence or congruence? - the standard indication is three lines instead of two lines: $x \equiv y \bmod N$. This notation means that x and y are congruent modular N, which means that when we look at x divided by N and y divided by N, they have the same remainder.

***

(Slide 5) Example Mod 3



Look at a simple example to make sure everybody understands the concept of modular arithmetic.

Let's look at numbers modulo 3. When you look at numbers mod 3, there are three possible values, 0, 1, or 2. Hence, there are three equivalence classes for the numbers modulo 3:

- 0: 3 mod 3 is also zero. Same with 6, 9, and so on. All the multiples of three.
- 1: Similarly, 4 mod 3 is one. Same with seven, and 10, and so on.
- 2: Finally, 5 mod 3 is two. Same for eight, and 11, and so on.
- But look, we can also go negative. What is -3 mod 3? Well, -3 divided by 3 equals -1, and the remainder is 0. So -3 mod 3 = 0. Same for -6, -9, and so on.
- Now if I look at -2 mod 3, then it is 1. Why is that?

  x mod N = r means that when I divide x by N, I get a remainder of r. What does that mean? - that means that there is some multiple (q) of N such that qN + r = x

  Now look at -2 mod 3 - r is 1. Why is that? So if I look at -2 divided by N = 3, what do I get? I get q = -1: -1 x 3 + 1 = -2 . So, the remainder is one. So -2 mod 3 is one.

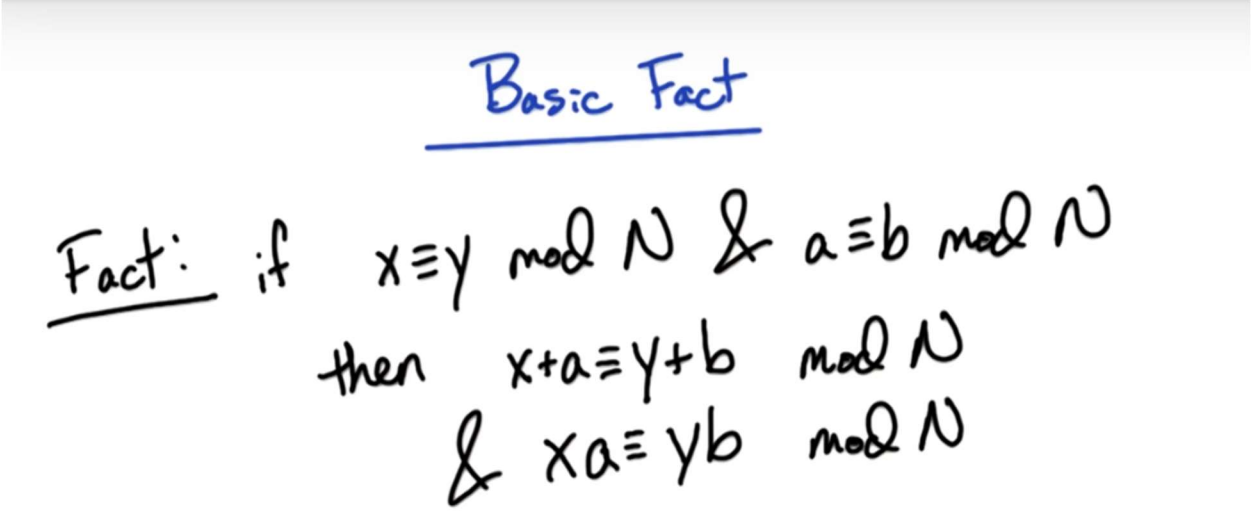
  Similarly, -5 mod 3 = 1; -8 mod 3 = 1 as well.

- Finally, -1 mod 3 = 2. Same with -4 mod 3, -7 mod 3, and so on.

So for these three equivalence classes … all of these numbers in an equivalence class – for example, -9, -6, -3, 0, 3,6, 9, - are all the same with respect to modulo three. They're all congruent mod 3.

***

(1st Slide 6) Basic Fact



Here's a basic fact that we're going to use repeatedly throughout our work.

> Let's say I have two numbers, x and y, which are congruent mod N:
> $x \equiv y \bmod N$
>
> And, I have two additional numbers, a and b, which are congruent mod N:
> $a \equiv b \bmod N$
>
> Now, the point is, that with respect to mod N, x and y are equivalent and a and b are equivalent. So, I can replace one by the other. More precisely, suppose I'm looking at x plus a mod N. Well, x is equivalent to y, so I can replace x by y, and a is equivalent to b, so, I can replace a by b.
>
> Therefore, $(x + a) \equiv (y + b) \bmod N$. I can replace x by y and a by b.

Similarly, if I look at x a mod N, then I can do the same replacement operation:
$x\,a \equiv y\,b \bmod N$.

Now let's look at an example to illustrate the usefulness of this basic operation.

***

RA1: Modular Arithmetic: Basic Fact

Fact: if $x \equiv y \bmod N$ & $a \equiv b \bmod N$
then $x + a \equiv y + b \bmod N$
& $xa \equiv yb \bmod N$

Question: Compute 321 x 17 mod 320

***

(2nd Slide 6) Basic Fact (Answer)

Here's how to compute it:

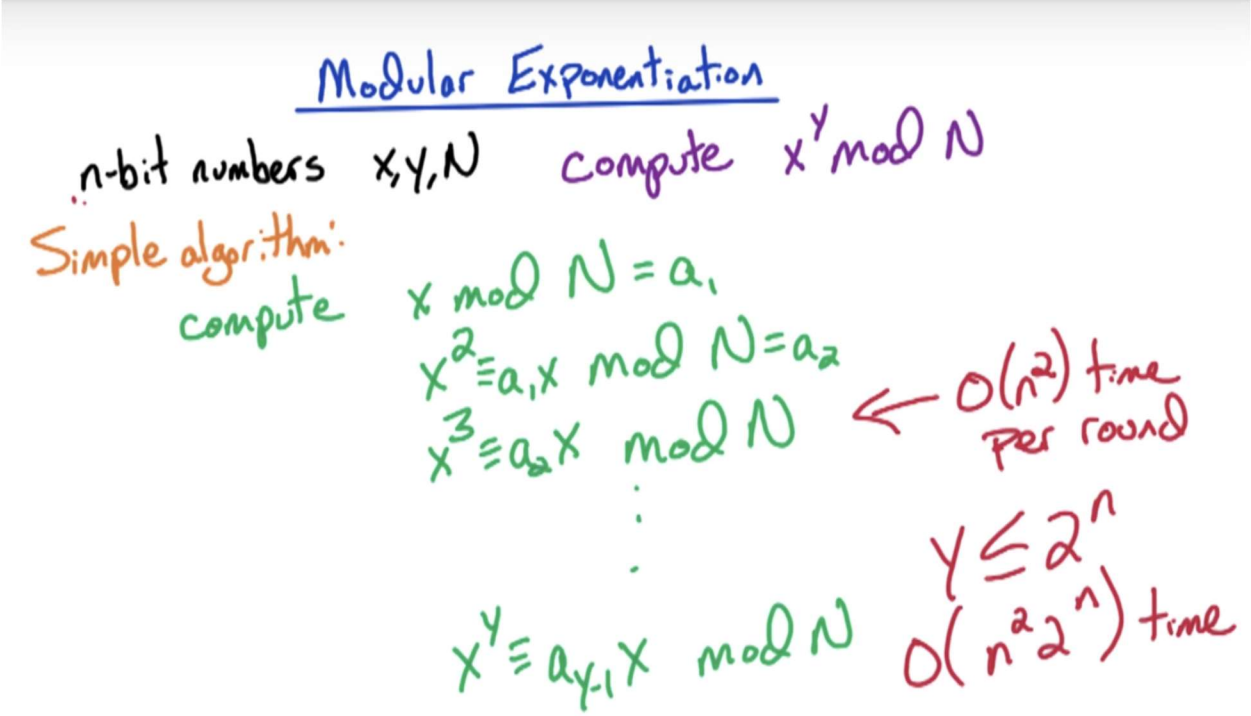
Note, $321 \equiv 1 \bmod 320$.

Hence, $321 \times 17 \equiv 1 \times 17 \equiv 17 \bmod 320$.

Answer: 17

***

(1st Slide 7) Modular Exp.: Naive



We've seen the definition of the modulo operation. The basic modular arithmetic operation that we're going to do repeatedly is modular exponentiation. So, the setting is that we have n-bit numbers x, y, and N, and our goal is to compute x^y mod N.

Now recall, little n, the number of bits, is going to be huge. For example, we might have little n, the number of bits, be about a thousand or 2000 in which case these numbers, x, y, and capital N, are on the order of about two to the thousand or two to the 2000. S o we want an efficient algorithm to compute x^y mod N.

By efficient, we want polynomial in little n, because little n is the size of the input. To represent these numbers, it takes little n bits. So, we want polynomial in the input size, which is polynomial in little n. We don't want polynomial in the numbers themselves because the magnitude of these numbers is exponential in little n. These are huge, if we had running time which was polynomial in 2^{1000} that would be enormous. There would no way we could run such an algorithm. So, let's look at the time it takes to compute x^y mod N.

So, let's start off with a simple algorithm for computing x^y mod N:

- We start off by computing x mod N, let's call that a1.

- Then, what do we do? We compute x^2 mod N. Now x squared - that's going to be a1, which is x mod N, and then multiply that by x.

  So, we take the previous solution, a1, multiply it by x and then take that mod N.

- Then for x^3 mod N, we take the previous solution, let's call that a2, and we take the a2 and we multiply by x and we take it mod N.

- And we keep repeating finally for x^y in the last round we take the previous solution, which is x^{y-1}, and we multiply by x and we take mod N.

Now, how long does this algorithm, this simple algorithm take? Let's look at one round. What are we doing?

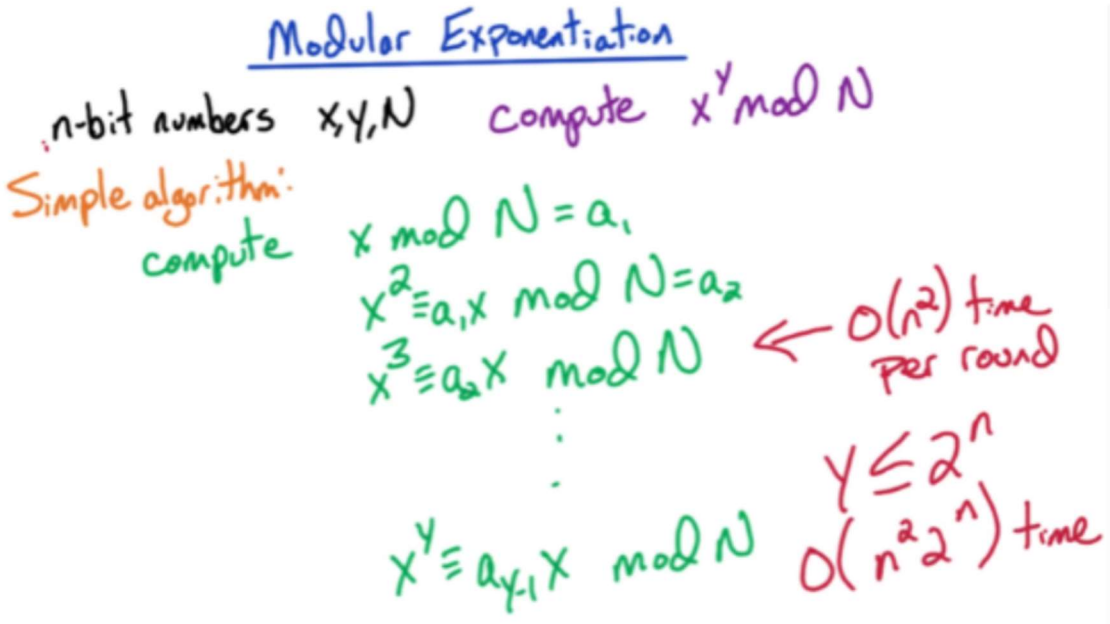
- Let's look at x^3. So, we're taking this number a2, which is an n bit number because it's at most N-1. Okay? So, this is n bit number, x is an n bit number. How long does it take to multiply two n bit numbers? That's just normal real arithmetic. That has nothing to do with modular arithmetic, okay? So, we're multiplying two n-bit numbers - that takes $O(n^2)$ time, and then we're taking it mod capital N.
- How do we do that? - we take this n-bit number and we divide it by this n-bit number and we take the remainder.
- How long does it take to divide two n-bit numbers? That's $O(n^2)$ time. So, this takes $O(n^2)$ time for this one operation.
- How many rounds, how many operations do we have? - how many ai's do we have? - we have y ai's. How large is y? - y is n-bits, little n-bits. So, y is at most 2^n.
- So then, the total runtime of our algorithm is $O(n^2)$ per round, and then the number of rounds is on the order of y, which is at most 2^n. So, that means the total runtime of our algorithm that we just described is $O(n^2 \times 2^n)$. It's an exponential time algorithm - exponential in the input size, little n.

So, this is a terrible algorithm. Even if we have a small n, let's say about 30, there's no way we could run such an algorithm. And we're going to be looking at a little n which is on the order of about 1000 or 2000. So, this is enormous.

So, what can we do better? - we used the basic idea of repeated squaring, which you've probably seen many times. So, instead of taking the previous answer and then multiplying by x, we're going to take the previous answer squared and that's going to give us the powers of two. Let's go ahead and and detail that.

***

RA1: Modular Arithmetic: Modular Exp.: Naïve



Question

Compute $7^5 \bmod 23$

***

(2nd Slide 7) Modular Exp.: Naïve (Answer)

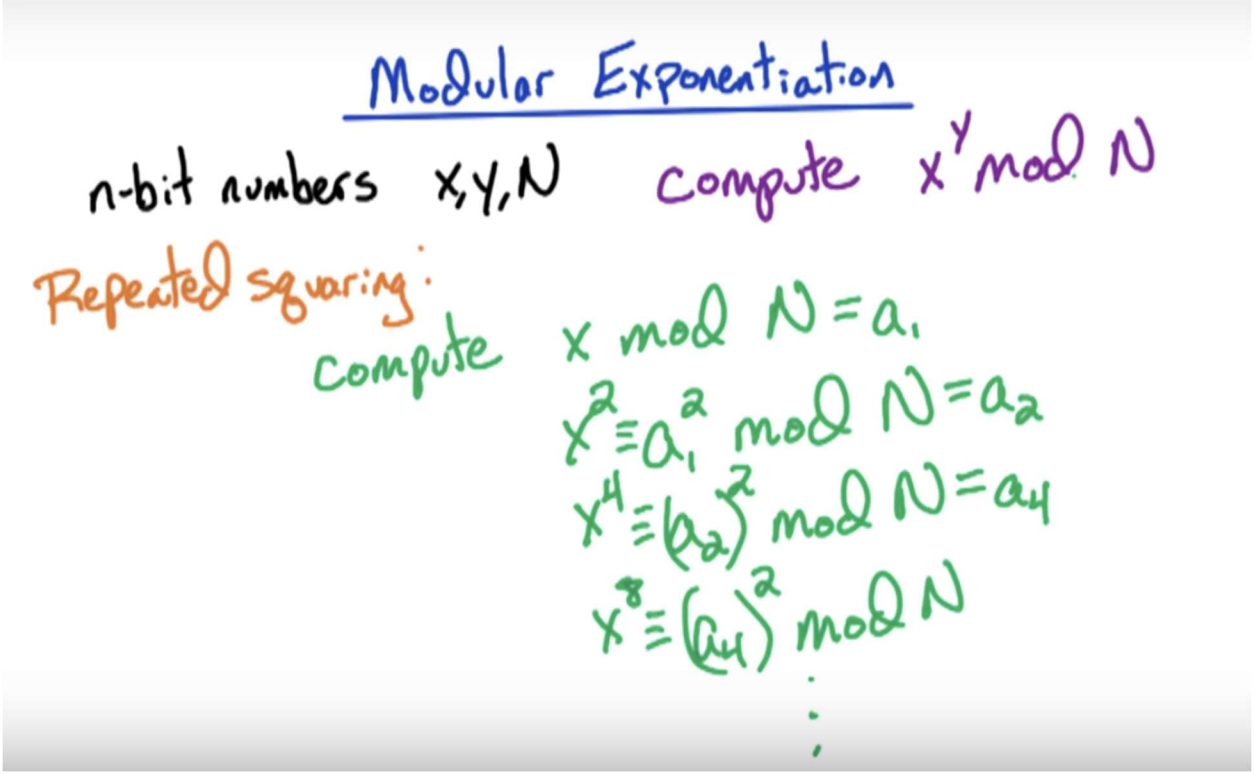
Here's how to compute it:

$$7^2 \equiv 3 \bmod 23$$
$$7^3 \equiv 3 \times 7 \equiv 21 \bmod 23$$
$$7^4 \equiv 21 \times 7 \equiv 147 \equiv 9 \bmod 23$$
$$7^5 \equiv 9 \times 7 \equiv 63 \equiv 17 \bmod 23$$

Answer: 17

***

(1st Slide 8) Modular Exp.: Fast



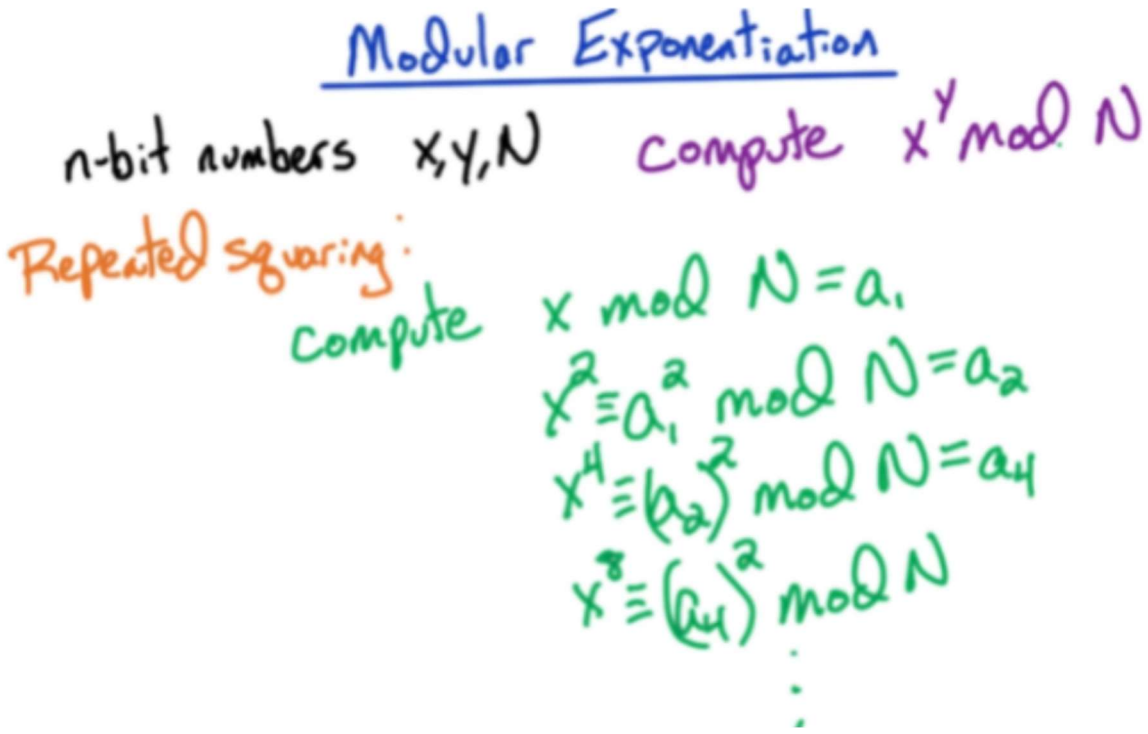
So in the repeated squaring algorithm, we start off the same.

- We compute x mod N and let's store the answer as a1.
- Then we compute x^2 mod N. How do we get x^2? - well, a1 is x mod N, so we take $a1^2$ mod N. So far it's the same as the other algorithm.
- Now we're going to skip x^3 and we're going to go to x^4. How do we get x^4? We take the previous solution - let's call it a2 and we square that. a2 ≡ x^2 mod N, so if we take x^2 and we square that, then we get x^4. Then we take that solution and let's call it a4.
- Now to compute x^8 mod N, you take the previous solution and square that. That gives us x4 squared which is x^8 mod N.
- And we repeat.

What do we end up with? We end up with x raised to the powers of two. Then what do we do? We look at the binary representation of y and then we can use the appropriate powers of two and we can get x^y mod N. Let's look at a specific example to illustrate the idea better.

***

RA1: Modular Arithmetic: Modular Exponentiation



Question: Use repeated squaring to compute $7^{25} \bmod 23$

***

(2nd Slide 8) Modular Exp. : Fast (Answer)

Here's how to compute it:

$7^2 \equiv 3 \bmod 23.$

$7^4 \equiv 3^2 \equiv 9 \bmod 23.$

$7^8 \equiv 9^2 \equiv 81 \equiv 12 \bmod 23.$

$7^{16} \equiv 12^2 \equiv 144 \equiv 6 \bmod 23.$

25 in binary is $11001.$

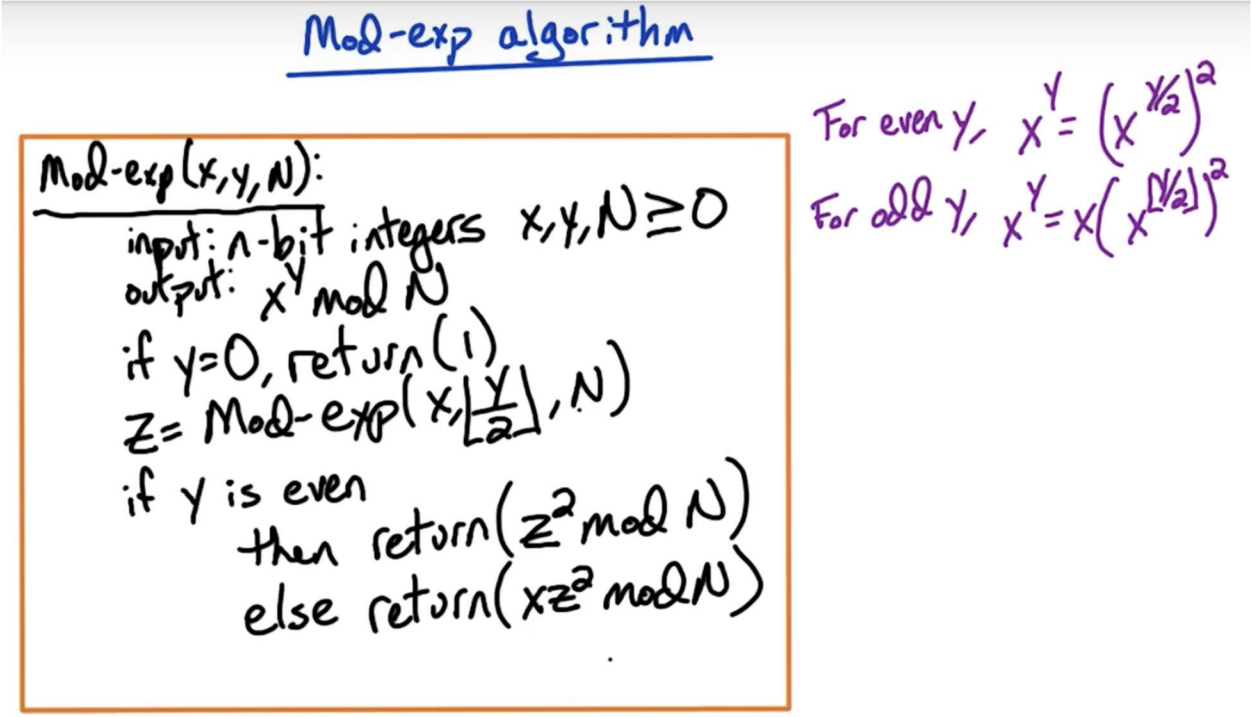
Therefore,

$$7^{25} \equiv 7^{16} \times 7^8 \times 7 \equiv 6 \times 12 \times 7 \equiv 72 \times 7 \equiv 3 \times 7 \equiv 21 \bmod 23.$$

Answer: 21

***

(Slide 9) Mod Exp Algorithm



Here's the key fact that we're going to use:

- For even y, when we divide y by 2, we get an integer. So, for even y, $x^y = (x^{y/2})^2$.
- What happens for odd y? Well, we need to be careful because when we divide y by 2, we get a fractional amount.

  So, for odd y, when we want to look at x^y. Let's take out an x, so that we look at x^{y-1}: $x^y = x * x^{y-1}$.

  Now, for x^{y-1}, we're going to take (y-1) / 2, which is the same as $\lfloor y/2 \rfloor$ and then we square it: $x^{y-1} = (x^{\lfloor y/2 \rfloor})^2$.

  And we can multiply by the extra factor of x, and we get $x^y = x * x^{y-1} = x (x^{\lfloor y/2 \rfloor})^2$.

Now, let's take this simple observation and detail a divide and conquer algorithm for modular exponentiation. And these three input parameters, x, y, and N are all little n-bit integers. And let's assume they are non negative integers. Now, our algorithm is computing x^y mod N. This can be a recursive algorithm. So, let's start with the base case.

The exponent is going to keep going down. The base case is going to correspond to y=0, in

which case we're taking x^0, which is 1. So, when we take x^0 mod N, that's also 1, assuming N is at least one.

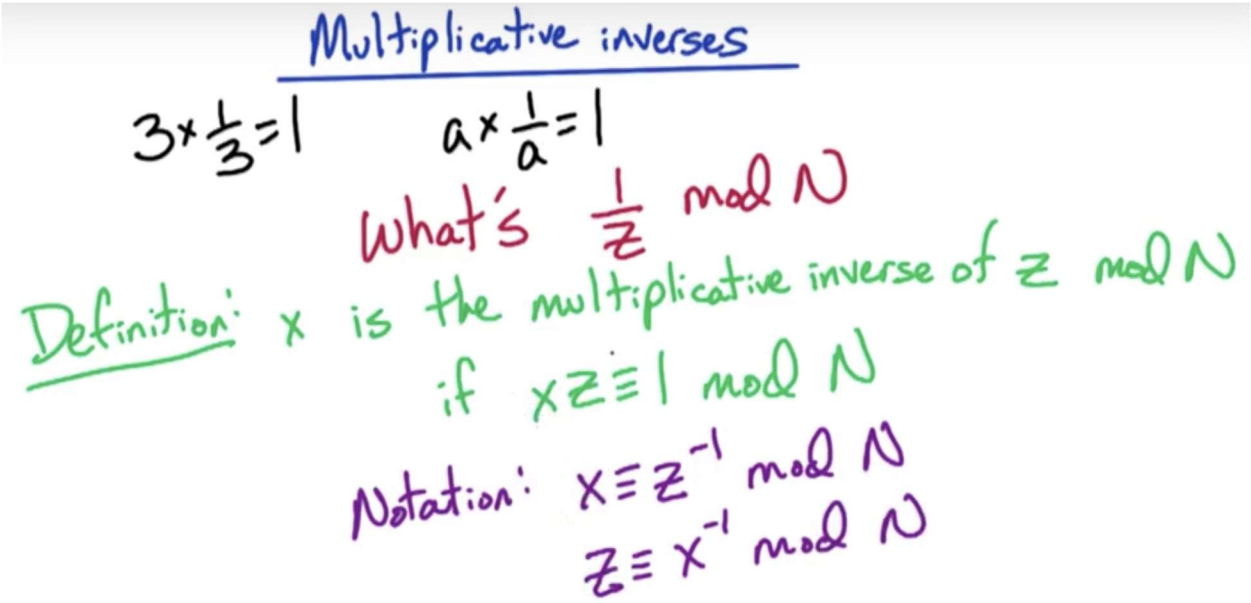
So, let's actually assume here that N is positive number. Now, let's look at the general case – Mod-exp(x,y,N) - what are we going to do? - we're going to compute this $x^{y/2}$:

- If y is odd, then we got to round it down. If y is even, we don't have to worry about rounding it down; but, if we round it down, it doesn't make any difference. So, we're going to recursively call this algorithm using Mod-exp(x, $\lfloor y/2 \rfloor$, N). We store that answer in z.
- Now, we check whether y is even and we follow the case for even numbers - $x^y = (x^{y/2})^2$ - or, if y is odd, we follow this case for odd numbers: $x^y = x(x^{\lfloor y/2 \rfloor})^2$
- In the case when y is even, then we want to take z (which is $x^{y/2}$) and we want to return z^2 mod N.
- In the other case, when y is odd, we have to take z^2, which is the inner term, and then multiply by x. So, in this case, we want to return x * z^2 mod N.

And that's our answer when we return.

***

(Slide 10) Multiplicative Inverse



A key concept that we're going to need for the RSA algorithm are multiplicative inverses. Now the concept is a bit subtle for a modular arithmetic. Let's go back and look at the normal, real numbers and see what multiplicative inverses mean there.

If we take a number 3, what's its multiplicative inverse? - it's 1/3. And so if we look at 3 times 1/3, what do we get? We got 1. And, in general, for a number a, it's multiplicative inverse is 1/a. And, when we multiply those two together, what do we get? 1.

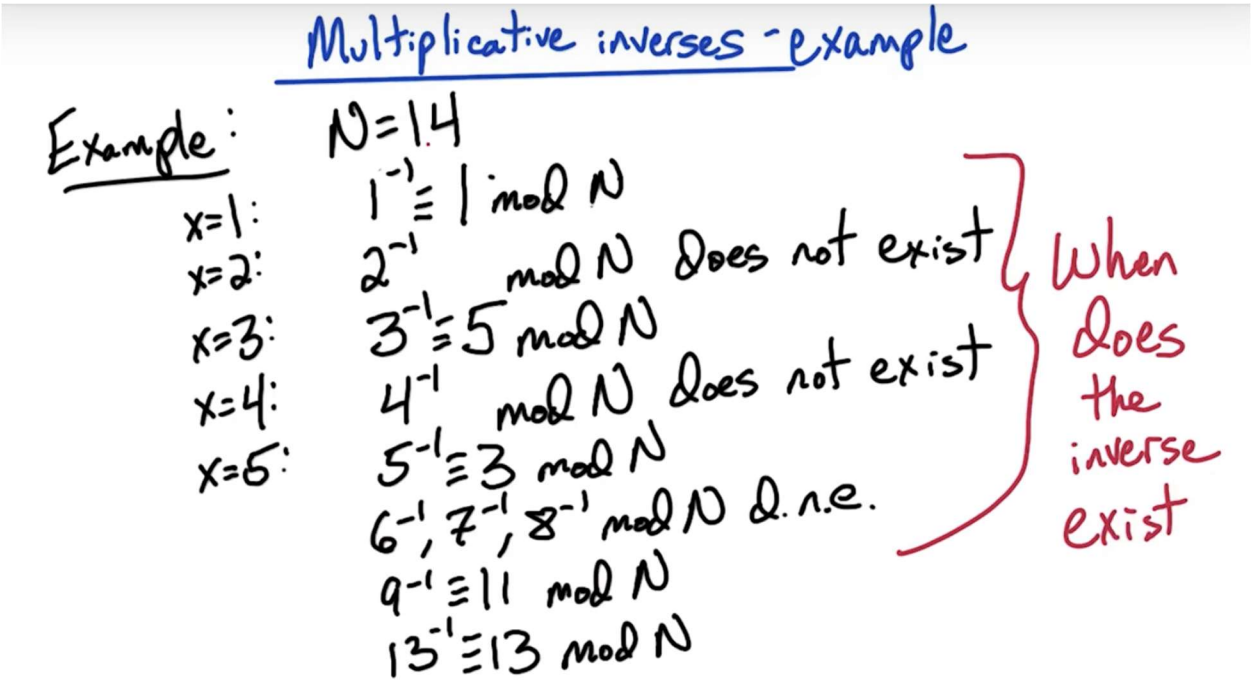
Now, we're going to have a number z and we want to look at its inverse mod N. So, we want to know what one over z mod N is. So we're going to define it as this number that when we do z times one over z, we get back 1. So our definition of the inverse of z mod N is the number x, so that if we do x * z, we get 1 mod N. In this case x is the inverse, it's 1/z mod N.

And the notation is that $x = z^{-1}$ (same as one over z); so, $x = z^{-1} \bmod N$. So x is the multiplicative inverse of z mod N.

Notice also, that if x is the inverse of z mod N, then z is the inverse of x mod N, because x times z is 1 mod N. So, x is the inverse of z and z is the inverse of x mod N. Now let's go ahead and look at a specific example to make sure this notion of multiplicative inverse modulo N makes sense.

***

(Slide 11) Inverse: Example



So, let's take the example where capital N is 14. And let's look at the inverses of the numbers starting with 1 thru 13 mod N, starting with x = 1.

- So, what is the inverse of 1 mod 14? What is the number so that 1 times that number is 1 mod 14? Well, 1 times itself is 1. So, the inverse of 1 is 1 itself. Its a self inverse. And that's always the case. 1 is always itself's inverse.
- Let's do a non-trivial case where x = 2. What's the inverse of 2 mod 14? What number times two is congruent to 1 mod 14? Well, actually there is no number where that's the case. So, the inverse of 2 mod 14, does not exist.
- Let's try x = 3. What's the inverse of 3 mod 14? Well, notice 3 * 5 = 15 which is 1 mod 14. So, the inverse of 3 is 5.
- And the inverse of 5 is 3 mod 14.
- What about x = 4? What's the inverse of 4 mod 14? Once again, just like the case of x = 2, that doesn't exist.
- What about x=5? We already solved that. Its inverse is 3.
- How about 6, 7 and 8? Actually, none of those have an inverse mod 14.
- What about 9? What's the inverse of 9 mod 14? This is 11 because 9 * 11 = 99 and 14 * 7 = 98. So, 9 * 11 is 1 mod 14.
- The last one is, what about 13? What's the inverse of 13 mod 14? It turns out that it is a self-inverse. It's 13 itself.

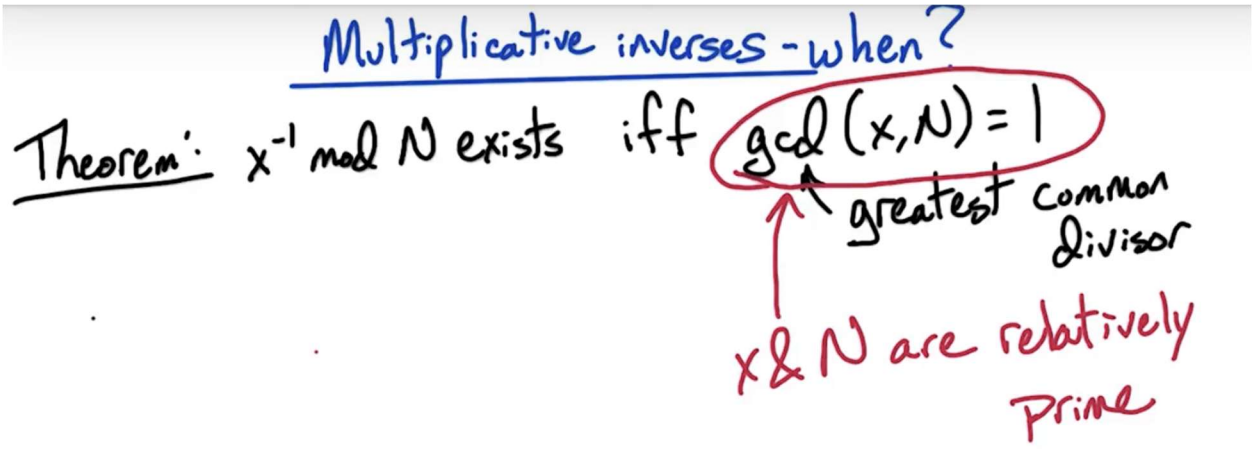
All the other cases - the inverse doesn't exist. When exactly does the inverse exist or not? What is the key fact about when the inverse exists or not?

- When 2, 4 and 6 - those are all even numbers. And 14 is an even number. So, they all have a common divisor too.
- Okay. What about 7? Well, 7 is not even but it shares a common divisor with 14: 7.

So, that's the key property: They share a common divisor (with the modulus, i.e., 14), then there is no inverse; but, if they have no common divisor (and in which sense, in which case, they're feel like primes relative to each other), then there is an inverse. So, 1, 3, 5, 9 and 13 have no common divisor with 14.

***

(Slide 12) Inverse: Existence



S, when exactly are there multiplicative inverses or not? The general theorem is that x^{-1} mod N exists if and only if x and N have no common divisor.

So, gcd(x.N) = 1. gcd stands for the greatest common divisor. So, if they're both even, then 2 is a divisor of both so the gcd(x,N) is at least 2 - it might be bigger – but it's at least 2.

Now just some terminology when the greatest gcd(x,N) = 1:

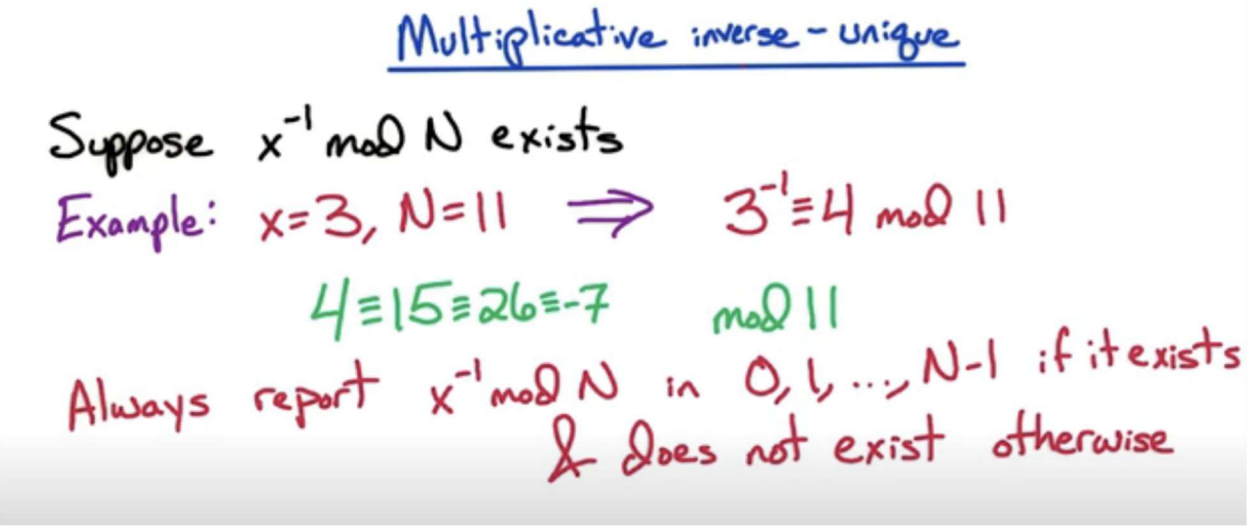
- so they have no common divisor, then we say that x and N are **relatively prime with respect to each other** - they feel like primes to each other.

  For instance, 9 and 14 neither one is prime; but, with respect to each other, they have no common divisors. So, they feel like primes to each other.

So the theorem says that if these numbers relatively prime to each other, then there is an inverse, but if they have a common divisor, then there is no inverse. Let's look at why this theorem holds, but before we get into it, let's look at some basic properties about inverses.

***

(Slide 13) Inverse: Terminology



Suppose that x has an inverse mod N. So, x^{-1} mod N exists. Let's prove that this inverse, if it exists, is unique.

What exactly do we mean by unique? Well, let's consider a specific example. Here's a simple example:

- x = 3 and N = 11. What's the inverse of 3 mod 11? Well, these numbers are quite small, so it's easy to figure out that the inverse of three mod 11 is 4, because 3 x 4 = 12, which is 1 mod 11.
- Now, if we look at 4 mod 11, there are an infinite number of equivalent numbers. For example, 15, 26, and so on, these are all congruent to 4 mod 11. We can also look at negative numbers, -7 is congruent to 4 mod 11.

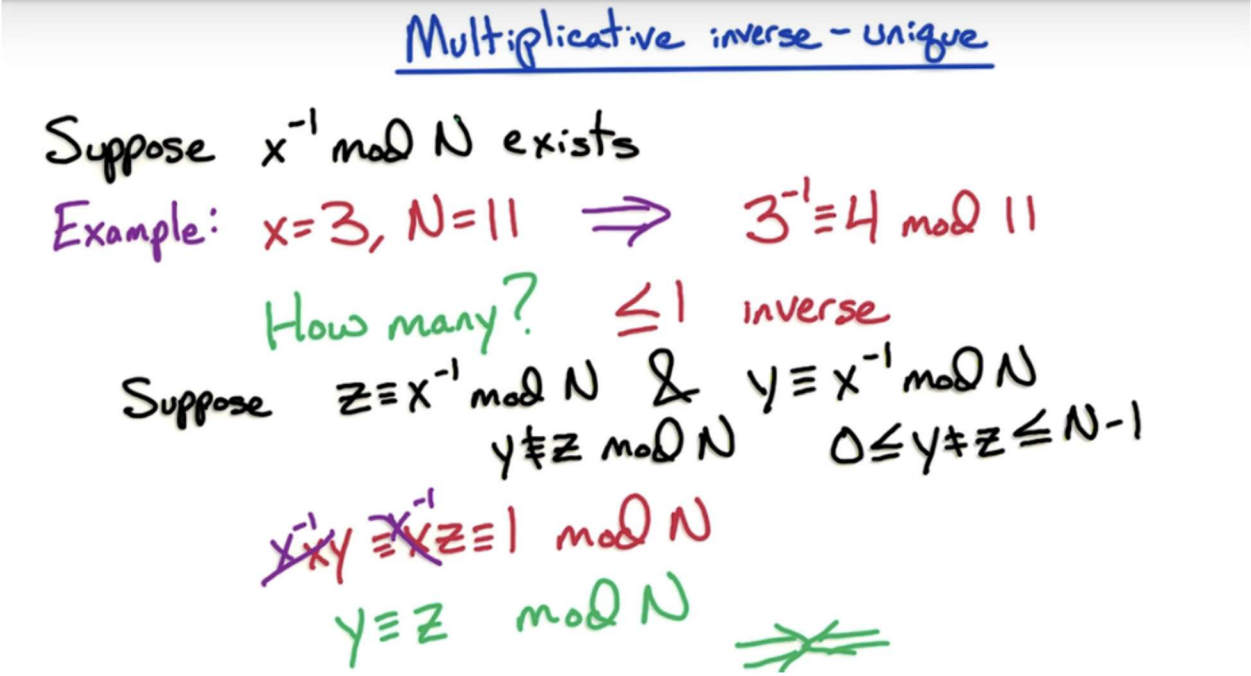
  And there are an infinite number of such numbers which are equivalent to 4 mod 11.

All of these numbers are inverses of x mod N, but for concreteness, we'll always report the inverse of x mod N as the smallest non-negative integer. So, we report it as the integer between 0 and N-1, if that inverse of X mod N exists. And, if the inverse of X mod N does not exist, we simply report does not exist.

Now, later in this lecture, you'll see how to find the multiplicative inverse if it exists. And to do that, we'll use the extended Euclid algorithm. In that case, often the algorithm will return the negative number and then we'll have to do a simple calculation in order to convert it to an integer between 0 and N-1.

***

(Slide 14) Inverse: Unique



Now, let's go back to our original question. Suppose x^{-1} mod N exists. How many such inverses can there be?

So, in this example, 3 mod 11, 4 is one such inverse. Are there any other inverses? When we're looking at integers between zero and 10, well, it's easy to check that 4 is the only integer between 0 and 10 which is an inverse of 3 mod 11.

And, what we want to prove now is that if x has an inverse mod N, then there is a unique such inverse. So, there is only one such inverse.

Let's prove now that if the inverse exists, it's a unique choice. So, there's a unique number between 0 and N-1, which is an inverse of X mod N. How are we going to prove this? Let's prove it by contradiction:

> Proof by contradiction
> So, let's suppose that x mod N has two inverses. And then, let's show a contradiction.
>
> So, suppose that z is an inverse of x mod N ($z \equiv x^{-1} \bmod N$), and suppose that y is also an inverse of x mod N ($y \equiv x^{-1} \bmod N$). And let's suppose that these numbers y and z are different. What do we mean different? We mean, they are different mod N: $y \not\equiv z \bmod N$. So, we're assuming that y and z are different mod N. This means that if we

think of the y and z as numbers between 0 and N-1, y and z are different.

For this small example ($3^{-1} \equiv 4 \bmod 11$) - for these small numbers, it's easy to verify that this number 3 can have multiple inverses mod 11.

We want to prove, in general, that if a number x has an inverse mod N, and that inverse is unique. So, we're assuming that there's multiple inverses of x mod N. And we're denoting those as y and z. And now we're going to derive a contradiction.

Now z is an inverse of x mod N, so what does that mean? That means that if we look at the product of x times z, $x * z \equiv 1 \bmod N$. That's the definition of an inverse. Similarly, if we look at y, we know that x * y is also congruent to 1 mod N. Therefore, $x * y \equiv x * z \bmod N$.

Now, we know that the inverse of x mod N exists. Actually, we're supposing that there's multiple such inverses … Anyways, take one of those inverses and multiply both sides of this equation by that inverse. So, multiply the left side and the right side by x^{-1}: $x^{-1} * x * y \equiv x^{-1} * x * z \bmod N$. What do you get? Well, $x^{-1} * x = 1$, so we can cancel these terms, and what are we left with? We are left with the y and z. So, the left-hand side simplifies to y, the right-hand side simplifies to z. And we get that $y \equiv z \bmod N$. But our assumption earlier was that $y \not\equiv z \bmod N$, that these are different numbers between zero and N-1. Therefore, we derived our contradiction, and thus, our assumption that there is multiple inverses of X mod N is not true.

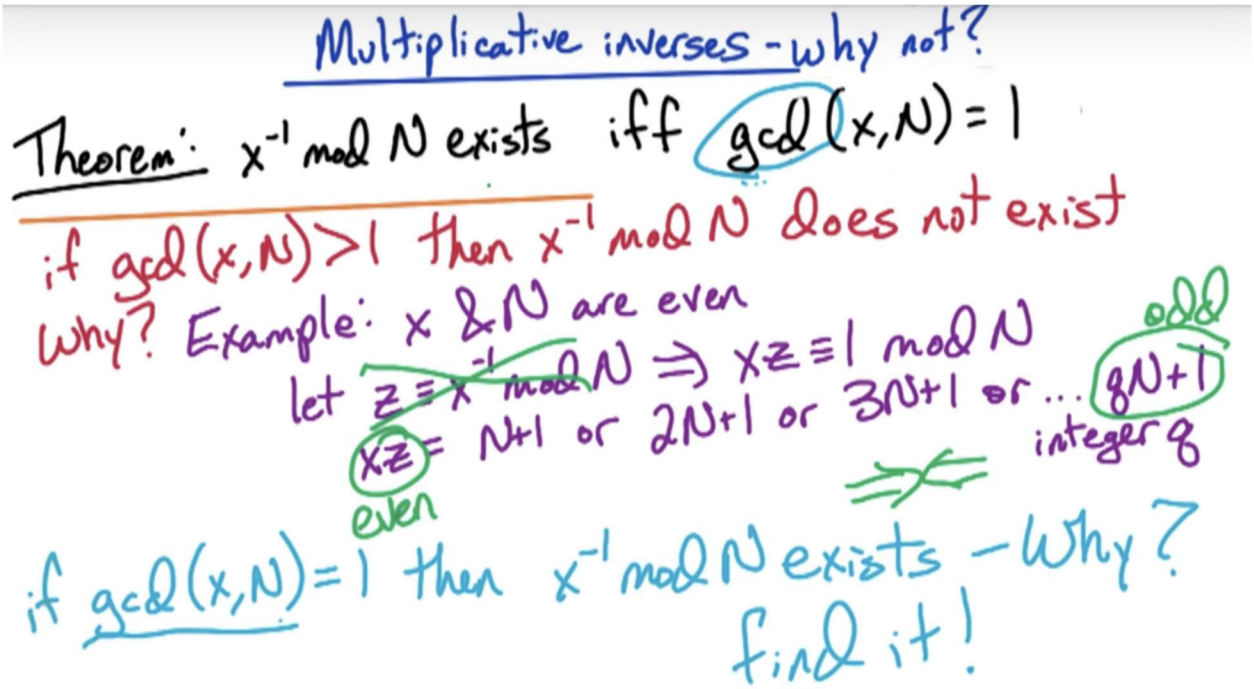
So, if the inverse of x mod N exists, it's a unique such inverse. It's unique when we look at these numbers between 0 and N-1.

Later, we're going to prove that if x and N are relatively primes (i.e., the GCD of x and N is one), then the inverse of x mod N exists. How are we going to prove that? We're going to prove that this inverse exists by finding it. We're going to show an algorithm extended Euclid algorithm to find the inverse, when the gcd(x,N) = 1.

But, on the next slide, we'll prove that if x and N have a common factor, so gcd(x,N) > 1, then the inverse of x mod N does not exist.

***

(Slide 15) Inverse: Non-existence



Let's look at the case when multiplicative inverses do not exist. Why don't they exist? So, in the case where x and N have a common divisor, their gcd is strictly greater than one. In which case, this theorem (Theorem: x^{-1} mod N exists iff gcd(x,N) = 1) says that the inverse of x mod N does not exist. Let's take a look at why that happens.

Let me give you the intuition for the proof. Let me do the simple case where x and N are both even. So, then they have a common divisor of 2 - If you understand this case then you can understand the general case.

- Suppose x did have an inverse mod N, let's call it z. So, what does that mean? That means that x * z ≡ 1 mod N. So, how can xz be congruent to 1 mod N? That means xz is either N+1 or it might be 2N+1 or 3N+1. In general, it's some multiple of N plus 1. So xz = qN+1 for an integer q.
- Now, x we assumed was even. So what is x times z? Even number times some other number is going to be an even number. Okay? Because it still has 2 as a divisor. Okay. so xz is even because X is even.
- Now N is even. So, what about q times N? That's going to be even as well just the same as before. So xz is even and qN is even.
- What about qN+1? We take an even plus one we got an odd number. So this is an odd number.
- xz is an even number. qN+1 is an odd number. How can they be equal? They can't be equal. So that's our contradiction. Therefore, there is no inverse of x mod N.

And similarly, if x and N just share a common divisor then you have a similar proof to explain why the inverse does not exist.

Now the interesting case is when they are relatively prime to each other, so their greatest common divisor is one. So they have no common divisors except for one. Then the inverse exists. Why is that the case? Can we prove why the inverse exists? We'll prove it exists by actually giving an algorithm for finding it. We'll actually find it and by finding it we prove its existence. Okay?

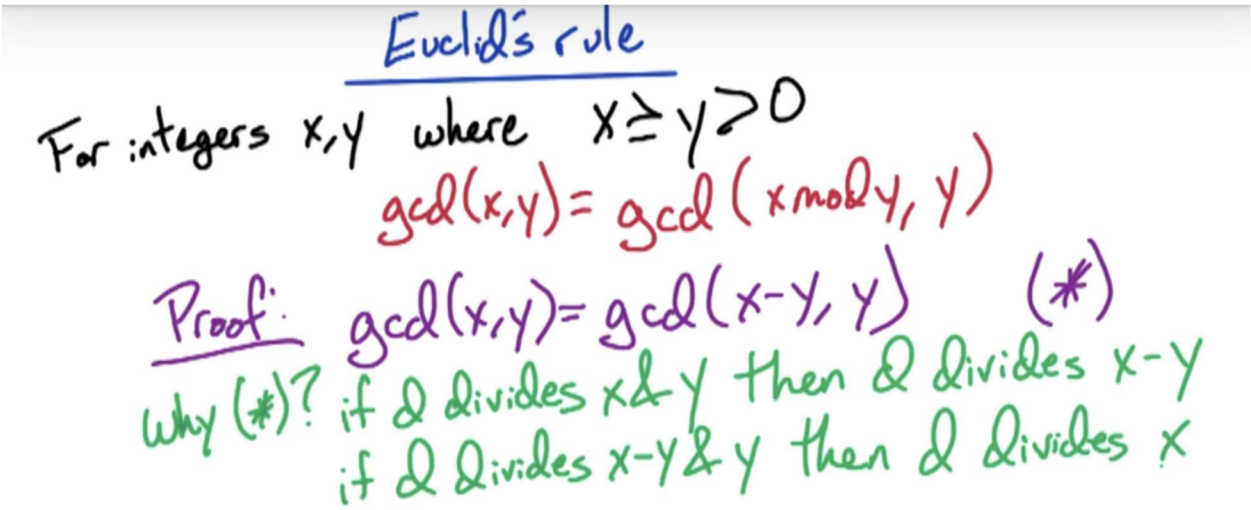
What is the algorithm we're going to use for finding it? Well, some of you might have seen Euclid's algorithm before for computing the greatest common divisor. So even if we want to check whether the inverse exists or not we have to compute gcd(x,N) – i.e., check whether they're relatively prime to each other. (I mean these are big numbers - so it's not easy to just eyeball and check – so, we got to run an algorithm to check the greatest common divisor and see whether it's 1 or not)

The algorithm for that is Euclid's algorithm which is very simple recursive algorithm and we're going to see it in a moment. And then a slight modification of Euclid's algorithm is the Extended Euclidean algorithm and that Extended Euclidean algorithm will give us the greatest common divisor of x and N and, if the greatest common divisor is 1, it will give us the inverse of X mod N. And therefore, that will prove the existence of the inverse of x mod N.

Okay? So let's go ahead and dive into Euclid's algorithm and the extended version of the algorithm to find the inverse.

***

(Slide 16) GCD: Euclid's Algorithm



Before diving into Euclid's algorithm, let's look at the basic fact which is called Euclid's rule which is the basis for Euclid's recursive algorithm.

Let's take integers x and y. And we want to compute gcd(x,y). And, let's assume for simplicity that x ≥ y (notice the order) and let's assume that y > 0. Then. the basic fact is that gcd(x,y) = gcd(x mod y, y). And notice these are now in reverse order because y is going to be at least x mod y. So, if we want to compute gcd(x,y), then, given this fact, we can compute gcd(x mod y, y). And that's going to be the basis of our recursive algorithm. But let's first see why this is true.

Here's a basic proof idea. The proof relies on the following simple fact: gcd(x,y) = gcd(x - y, y). Let's call this fact 'star' (shown as '*' on the slide). Notice, if we apply star repeatedly, we get the theorem statement: gcd(x,y) = gcd(x mod y, y).

> How do we get x mod y? We get x mod y by taking x and subtracting off y repeatedly. Let's say q times. So we take off as many y's as we can from x and then we take the remainder and that's x mod y. So it keeps subtracting y, and y, and y, and y repeatedly until we're left with a number smaller than y. So we keep applying this x-y. So we take x, x-y, and then that thing (x-y) minus y, minus y, minus y again, again…q times and then we end up with x mod y.

So, if we prove this fact star, then that implies this theorem statement. So it suffices to prove star and then that implies the theorem. Okay, and then, the star is a bit simpler statement than the theorem statement. So why's star true? Let's look at why that's true.

You can think of this as if and only if: (=>) If a number d divides x and y, then it divides x-y and u. (<=) if the number d divides x-y and y, then it divides x and y. So, any divisor of these two, it's a divisor of these two. Any divisor of these two, it's a divisor of these two. If we prove that, we're done. So we can think as a "if and only if". Given a divisor here of these two, it divides these two. Let's do that statement first.

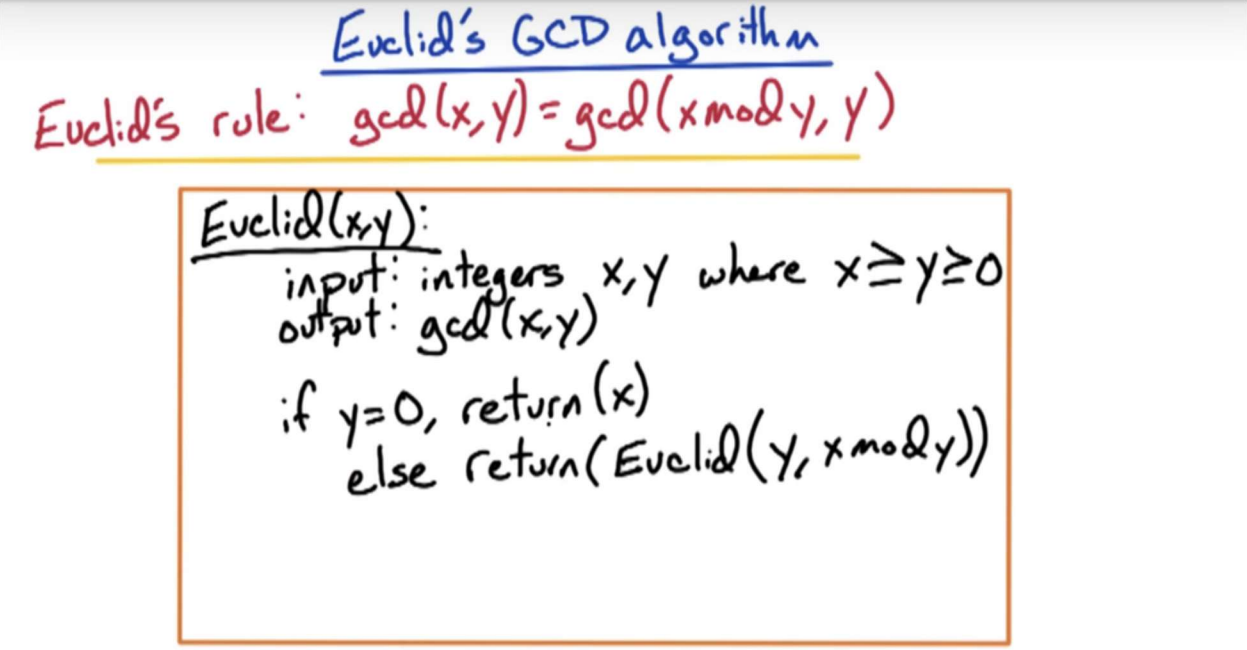
> (=>) If d divides x and y then clearly it divides y. And d divides x-y as well. If it divides x and y, then it obviously divides x-y as well. You can factor out d from x and from y, so you can factor it out from x-y. That gives this one direction, the forward direction.
>
> (<=) What about the reverse direction? So if d divides x-y and y, then d divides x as well because x is just the sum of x-y plus Y. So, it divides both of these then it divides their sum. So that gives us the reverse direction therefore they have all the divisors in common, so the greatest divisor is also in common.

That proves this statement star which implies this Euclid's rule.

Here is Euclid's rule that we saw in last time: gcd(x,y) = gcd(x mod y, y). Now, let's use that basic fact and design of recursive algorithm for computing the GCD. It's called the Euclid's algorithm.

***

(Slide 17) GCD: Euclid's Algorithm



What we just saw is the following fact known as Euclid's Rule:

gcd(x,y) = gcd(x mod y, y)

This leads to a natural, recursive algorithm which we'll now detail.

Here's Euclid's GCD algorithm. The input to the algorithm are two integers, x and y – Euclid(x,y) - and we'll assume these input parameters are ordered so that x >= y and that both of these integers are non-negative. And the output of the algorithm is the gcd of x and y. This is a recursive algorithm, so let's start with the base case.

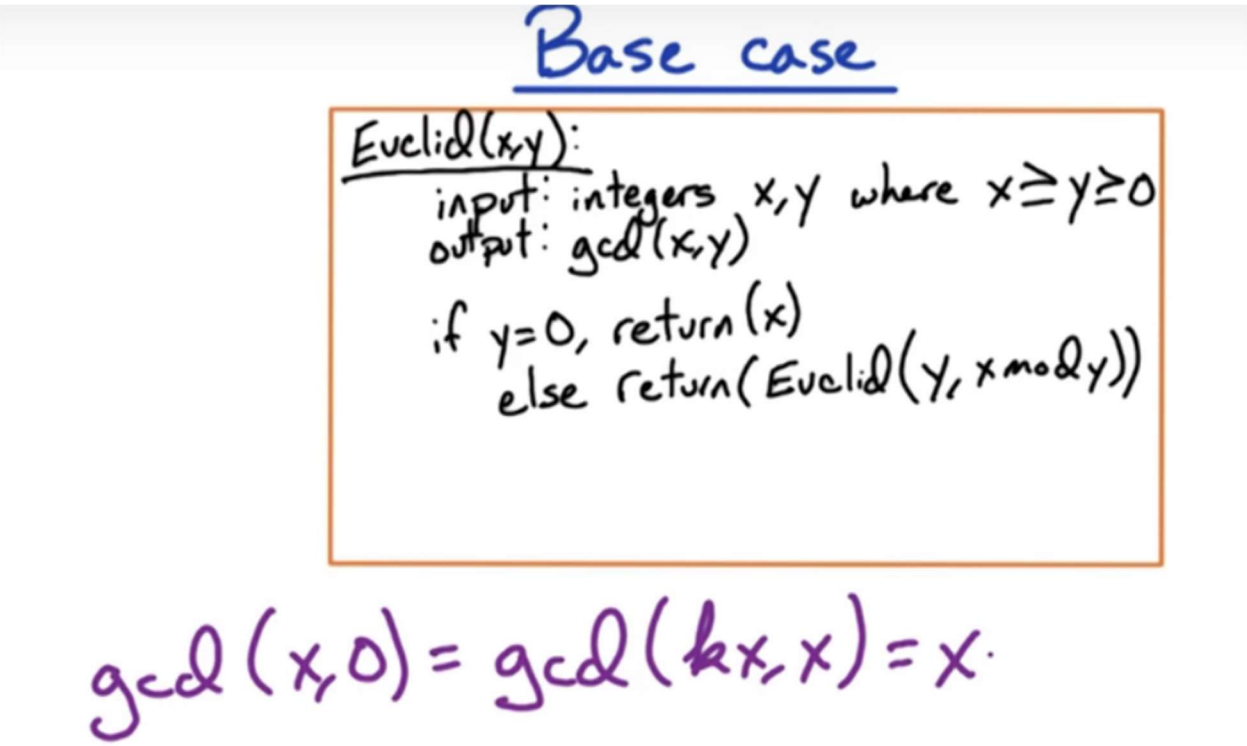
The base case will be when y, the smaller parameter, is equal to zero. So, when y = 0, we're looking at Euclid(x,0), so, we're going to return x as the gcd. Now, to be honest, this case actually is confusing to me. So, we'll go back and we'll discuss this more, in a little bit more detail, after.

Now, in the general case, we're just going to apply Euclid's rule. Euclid's rule tells us that the gcd of x and y is the same as the gcd of x mod y and y. So, we're going to return the recursive call of Euclid's algorithm on y and x mod y: return (Euclid(y, x mod y)). We flip the order of these two parameters in order to maintain the fact that the first parameter is at least the second

parameter. That completes the pseudocode of the algorithm. Let's go back and look at the base case.

***

(Slide 18) GCD: Base Case

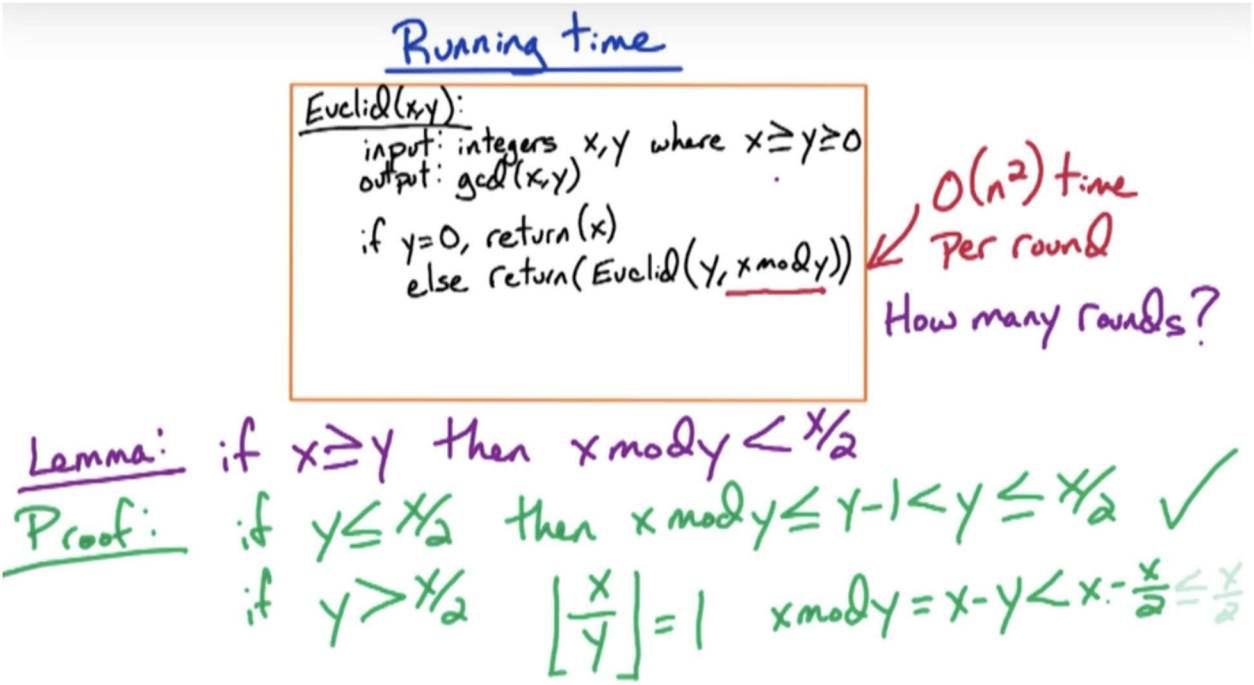


In the base case of our algorithm, y = 0. So, we're looking at the gcd of x and zero. What are the divisors of zero? This is a case where we should define it in an appropriate way. What is an appropriate way? What is a reasonable manner of defining the divisors of zero? Well, how do we get to this case? We got to this case by taking gcd(kx,x) – the gcd of some multiple of x with x. Because, look at what happens when we do the recursive call (Euclid(y,x mod y)) on gcd(kx,x) - we're going to take kx mod x, and that's going to be zero. So, the second parameter is going to be zero and the first parameter will be the second parameter (i.e., gcd(kx,x) = gcd(x,kx mod x) = gcd(x,0)).

Now, what is the gcd of kx and x? Well, this is clearly x, since the first parameter is a multiple of x. So, this is why we define the gcd(x,0) = 1.

***

(Slide 19) GCD: Running Time



Let's take a look at the running time of Euclid's algorithm.

The only non-trivial step in each round is computing X mod Y. How long does that take to compute? Well, that involves dividing x by y which takes $O(N^2)$ time, where N is the number of bits.

So the time is $O(N^2)$ time per round. How many rounds or recursive calls do we have in this algorithm though?

Here's, the key lemma for figuring out the number of rounds. If $x \geq y$ (which is the case for our algorithm - the first parameter is always at least the second parameter), then x mod y (which is the second parameter in our recursive call - that's the only parameter which is changing), is strictly less than x/2. So. the one parameter which is changing goes down by a factor of at least 2.

So, let's take a look at the algorithm. Let's say we have a call with x and y – Euclid(x,y) - then a recursive call is going to be Euclid(y, x mod y). What happens in our next recursive call? Well this second parameter is going to become the first parameter. So we get x mod y as the first parameter. What's the second parameter? The second parameter is y mod x mod y. Okay, it's a little bit hard to write. Let's skip it. It's not important. What does our lemma tell us? It tells

us x mod y is strictly less than x/2.

So, notice after two rounds of the algorithm, what happens to the first parameter? It went down by least a factor 2. So how many rounds does the algorithm have? It's going to have at most 2N rounds. Why? Because, every other round we've shown the first parameter goes down by a factor of at least 2. Since we have O(N) rounds and we have $O(N^2)$ time per round. The total running time is $O(N^3)$.

So we've established the running time of the algorithm modular this lemma.

Let's go ahead and prove the lemma which should be quite straightforward to prove. Once we break it up into the appropriate cases, the proof will be almost immediate. We're going to break it up based on the size of y - either y is small and then x mod y is immediately small, or y is big and then we'll figure out another reason why x mod y is small.

- So let's first take the case where y is small. Let's say $y \leq x/2$.

  What do we know about x mod y? How big can it be? The largest it can be is $y-1 < y$.

  What do we know about y? $y \leq x/2$. So what have we shown? We've shown that x mod $y < x/2$ which is what we're trying to prove.

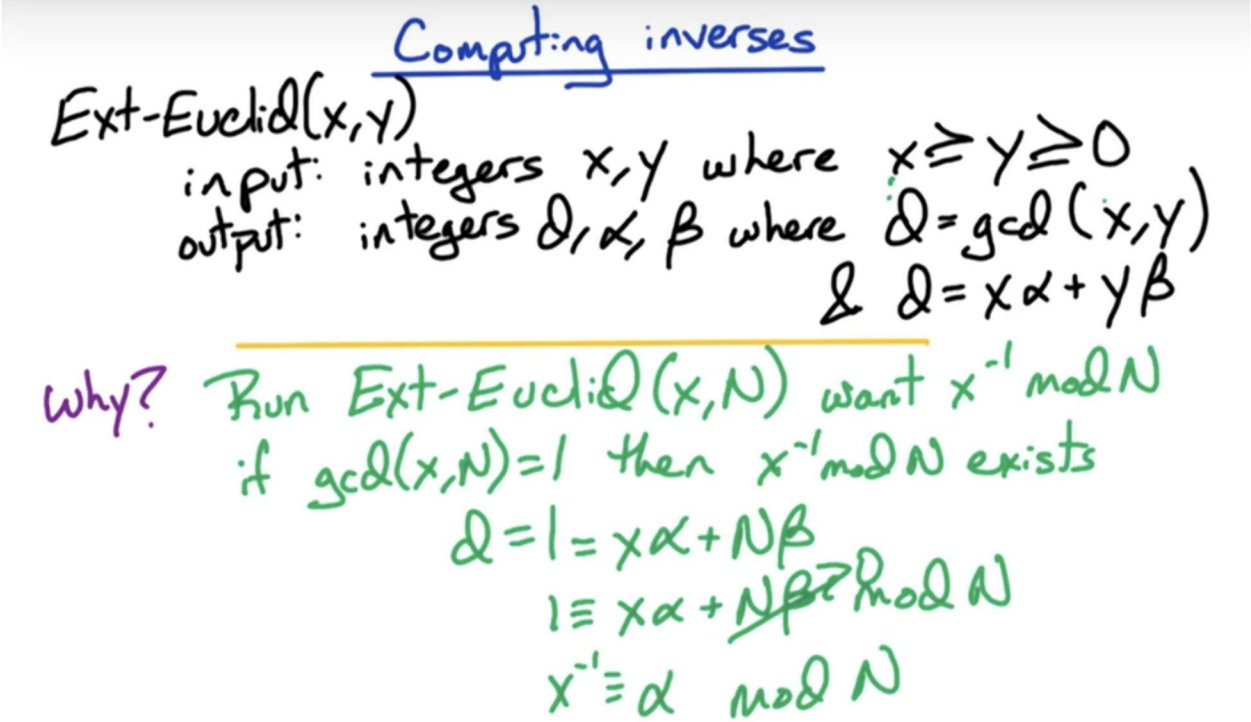
- Now let's take the case where y is big. $y > x/2$. What do we know about x divided by y? Well y goes into x at most one time. So $\lfloor x/y \rfloor = 1$.

  So what does that imply about x mod y? In order to get the remainder when we do x / y, we just have to subtract off y one time from x. So x mod y = x - y. Well, we know that $y > x/2$, so $x - y < x - x/2$ because $y > x/2$. And $x - x/2$ is clearly at most x/2. And that's what we're trying to prove.

We proved that x mod $y < x/2$, which is what we're trying to prove in the lemma. That completes the proof of the lemma, and therefore we've shown that the running time of Euclid's algorithm is $O(N^3)$ time.

***

(Slide 20) Computing Inverses



Now, let's look at how we compute inverses. To do this, we're going to use the extension of Euclid's algorithm called the Extended Euclidean Algorithm (Ex-Euclid).

Euclid's algorithm takes as input two parameters x and y: Euclid(x,y). And the same will be true for Extended Euclidean Algorithm: Ext-Euclid(x,y). These parameters x and y are integers, and we assume that these parameters are ordered, so that $x \geq y$ and both of these are non-negative.

Now, here's the difference: the output of the algorithm here is three parameters – d, α, and β. These are integers. Now $d = \gcd(x,y)$. So, if we just pay attention to the first parameter d, this will behave just as in Euclid's algorithm. Now, what are these additional parameters α and β?

Well, these parameters α and β satisfy the following relation: $d = x\alpha + y\beta$. Well, this is a bit mysterious -- why do I want an α and β which satisfy this relation? Let's look at why it's useful to have an α and β satisfying this relation.

Recall, our goal is to compute inverses. We're going to be in a setting where we're going to run Ext-Euclid(x,N). And, we're trying to compute the $x^{-1} \bmod N$. First off, do we know that this inverse exists? Well, we have to look at gcd(x,N) to see whether the inverse exists. We need

them to be relatively prime. We need that gcd(x,N) = 1 and then the inverse exists.

Well, when we run the Extended Euclid algorithm, d, the first parameter, gives us gcd(x,y). In this case, it's gcd(x,N), so if d = 1, then we know the inverse exists. So, where in this scenario, where d = 1, when we run Ext-Eculd(x,N) and we get back this alpha and beta which satisfy d = $x \alpha + N \beta$, we know that $1 = x \alpha + N \beta$.

Now from this α or this β, can we figure out $x^{-1} \bmod N$? Well, $1 = x \alpha + N \beta$ is True, so, if we take mod N of both sides, then the relation will still be True. So, now we have that

$$1 \equiv x \alpha + N \beta \bmod N$$

Now can we simplify this? Yeah, look at the second term here on the right hand side. This is N times beta. This is a multiple of N, therefore it's 0 mod N. So, we are going to replace $N \beta$ by zero. So, that goes away and what are we left with, we're left with

$$1 \equiv x \alpha \bmod N.$$

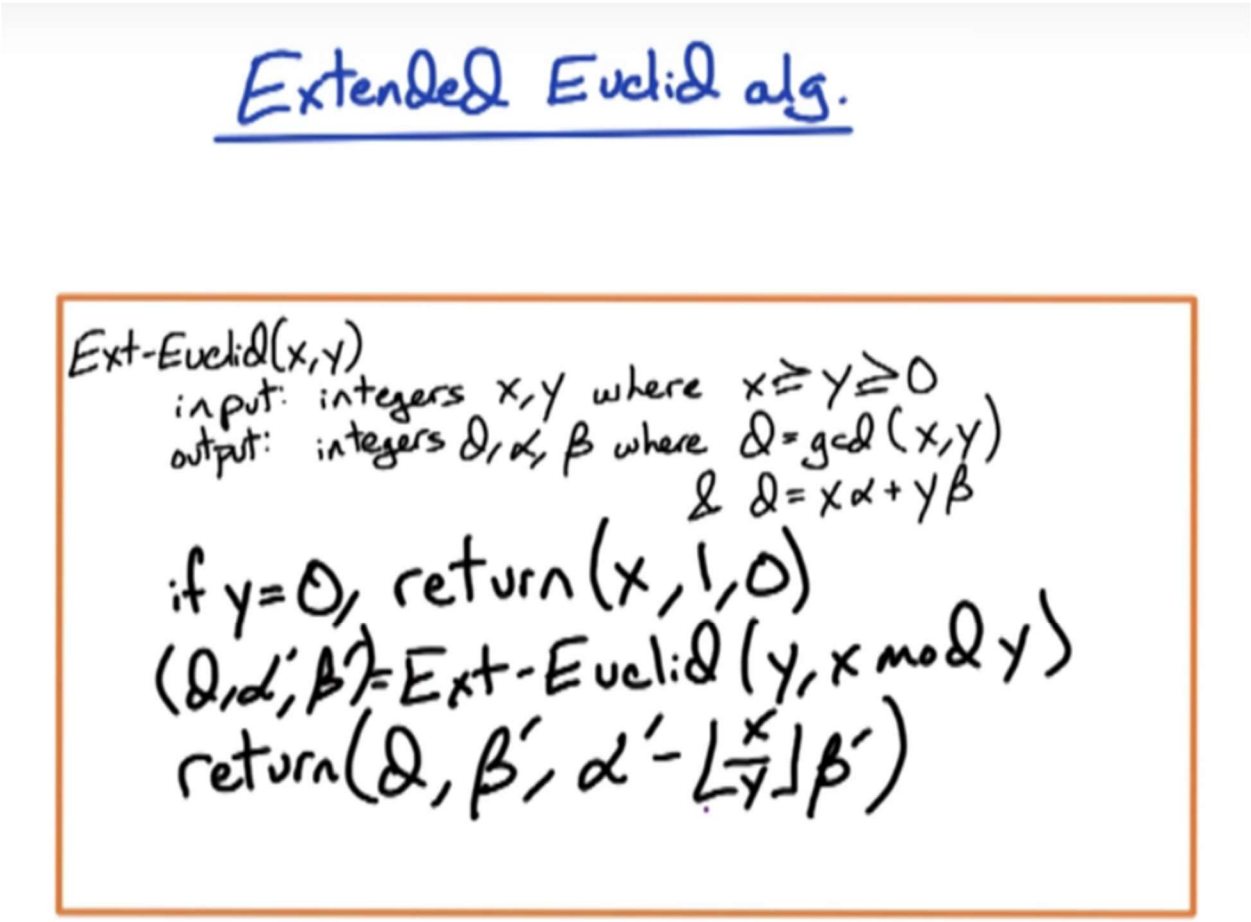
That means that α is the inverse of x mod N, because alpha is the integer where $x \alpha \equiv 1 \bmod N$ -- that's the definition of its inverse. So, we shown that $\alpha \equiv x^{-1} \bmod N$, in the case where d, the gcd of the two parameters, is equal to 1.

And, similarly $\beta = N^{-1} \bmod x$.

So, in the case where these two parameters x and y are relatively prime to each other, so gcd(x,y) = 1, so d = 1, then $\alpha = x^{-1} \bmod y$ and $\beta = y^{-1} \bmod x$.

***

(1st Slide 21) Inverses: Ext. Euclid Alg.



Let's go ahead and detail the Extended-Euclid algorithm:

- Ext-Euclid(x,y):

  Recall, it takes two parameters as input x and y where $x \geq y$
  and we're going to output three parameters: d, α, and β, where d = gcd(x,y).

- So, if we just look at the first parameter, d, it's going to behave like Euclid's algorithm and we're also going to output α and β, which satisfy the relation $d = x\alpha + y\beta$.

- Let's start again with the base case. So, when y=0, what do we return? Well, the first parameter d is the GCD. So, that should be equal to x.

  Now, we want to find α and β which satisfy the relation, where d in this case is equal to x. So, we want $x = x\,\alpha + y\,\beta$. Well, that's easier to satisfy. We can just set α equal to 1, and β beta equal to zero. That takes care of the base case.

- Now, for the general case, we're going to use Euclid's rule, which says that gcd(x, y) = gcd(y, x mod y). So, we recursively call this algorithm: Ext-Euclid(y, x mod y).

  That's going to return three numbers, let's call it d, α', β'. Now, we're going to have to do a little bit of manipulation of these three parameters in order to get our output for x and y.

  Now, we know the first parameter stays the same since gcd(y,x mod y) = gcd(x,y). Now, what do we know about α' and β'? Well, we know that $d = y\,\alpha' + (x \bmod y)\,\beta'$.

  Now, in order to find an α and β which satisfy $d = x\alpha + y\beta$, we have to do some algebra to manipulate this α' and β'. It turns out that it's sufficient to set $\alpha = \beta'$, and set $\beta = \alpha' - \lfloor x/y \rfloor\, \beta'$.

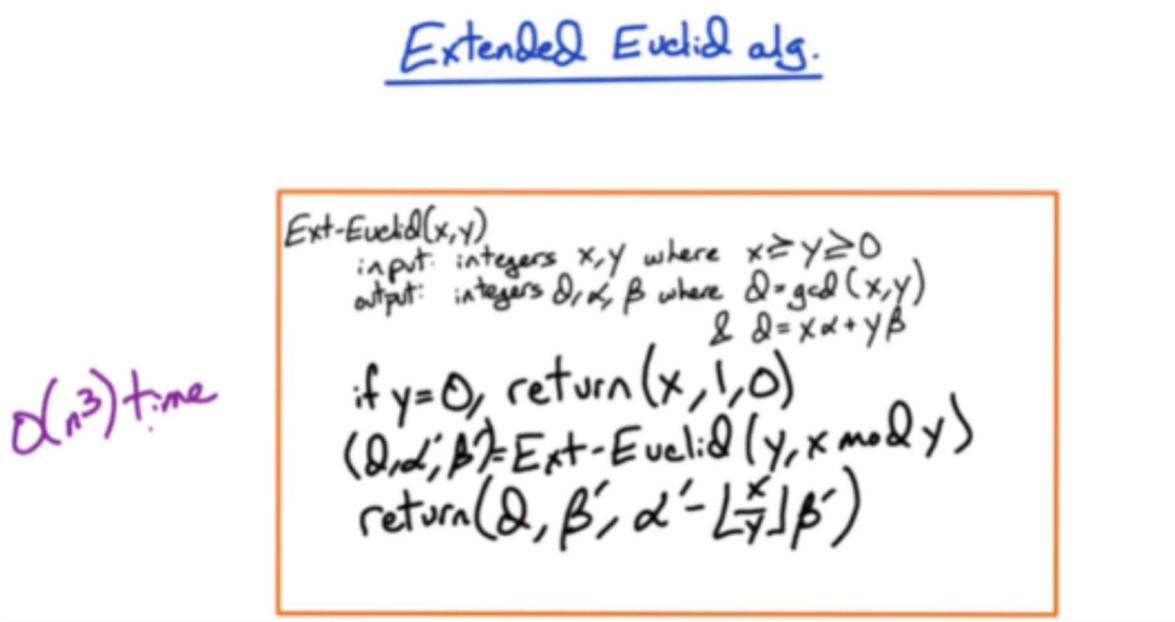
  In order to see where this quantity comes from, you want to look at the proof of correctness which is not too bad, but it's just a bit of algebra. I'll refer you to the textbook for the details on that proof of correctness.

That completes its pseudocode for the Extended-Euclid algorithm.

Finally, what's the running time with this algorithm? Well, notice, each step -- the non-trivial operations are x mod y and there's a little bit of algebra to calculate $\alpha' - \lfloor x/y \rfloor\, \beta'$. So, it takes $O(N^2)$ per round, and the number of rounds is O(N) just like Euclid's algorithm. So, the total runtime is $O(N^3)$.

***

RA1: Modular Arithmetic: Inverses: Ext. Euclid Alg.



Question

Use the Extended Euclid algorithm to compute $7^{-1} \mod 360$

***

(2nd Slide 21)

Here's how to compute it:

We're running $Ext - Euclid(360, 7)$.

Let's first look at (x, y) in the recursive subproblems.

They are $(360, 7) \to (7, 3) \to (3, 1) \to (1, 0)$.

(Note, these are the same sequence of subproblems as encountered in Euclid's GCD algorithm.)

Now let's look at the pair (α, β) returned for these inputs.

For the base case $(x, y) = (1, 0)$ we have: $(\alpha, \beta) = (1, 0)$.

Next, for $(x, y) = (3, 1)$ we have: $(\alpha, \beta) = (0, 1)$.

Then for $(x, y) = (7, 3)$ we have: $(\alpha, \beta) = (1, -2)$.

Finally, for $(x, y) = (360, 7)$ we have: $(\alpha, \beta) = (-2, 103)$.

Therefore,

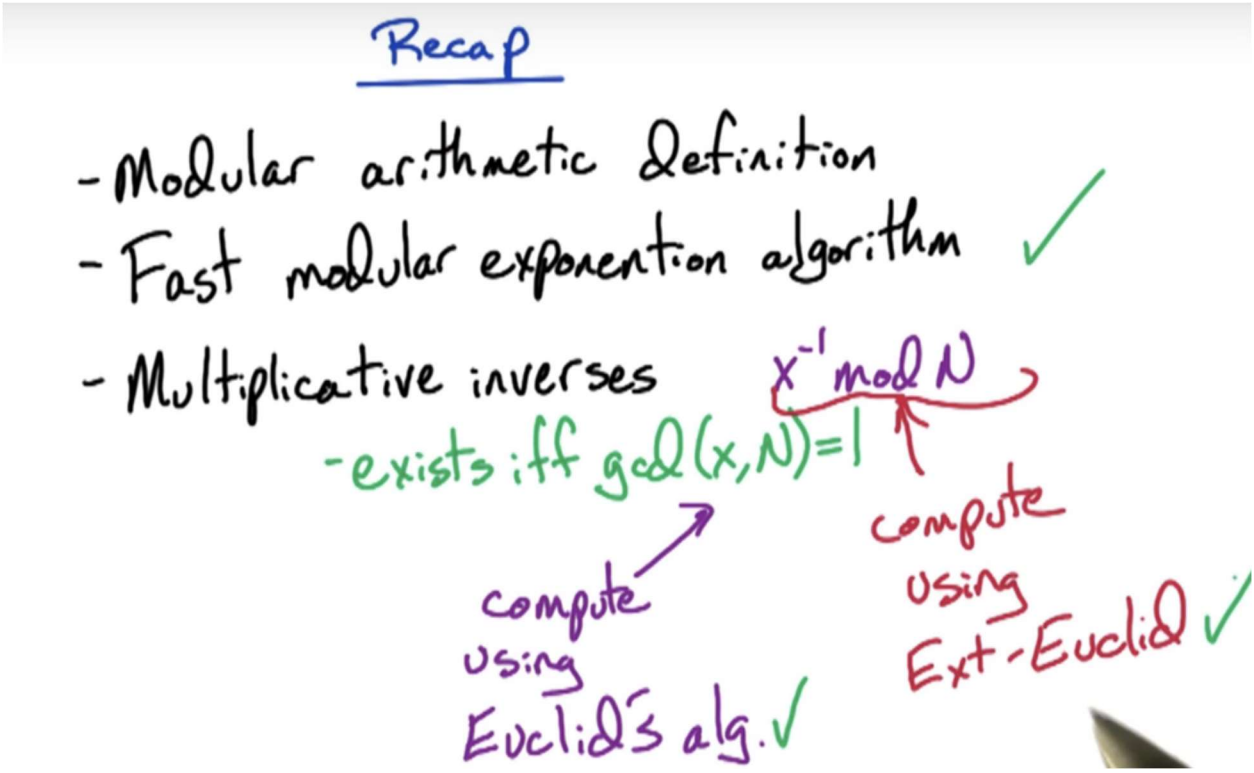
$7^{-1} \equiv 103 \bmod 360$.

Note, we also get that: $360^{-1} \equiv -2 \bmod 7$. Simplifying, we have: $360 \equiv 3 \bmod 7$ and $-2 \equiv 5 \bmod 7$. Therefore, $3^{-1} \equiv 5 \bmod 7$.

Answer: 103

***

(Slide 22)



Let's recap what we've seen so far:

- We've seen the **Modular arithmetic** definition, x mod y.

- The first non-trivial algorithm that we saw was computing **modular exponentiation** using the repeated squaring idea. This gave us an algorithm which was polynomial in the number of bits, little n.

  This modular exponentiation algorithm is gonna be a key component in the RSA algorithm. We're gonna have numbers x, y and capital N and we're gonna have to compute x^y mod N. And these numbers x, y and N are all going to be a huge number of bits. So we're going to have to compute this modular exponentiation, in time polynomial, in the number of bits, not in the size of the numbers.

  So it will be key that we use this **Fast algorithm** that we devised.

- The other key concept that we looked at was **multiplicative inverses**. For example, x^{-1} mod N. What we saw is that this inverse exists, if and only if, these two numbers are relatively prime.

- In other words, that their gcd is one. How do we check that they're relatively prime? How do we check their gcd? Well, we can compute their gcd using **Euclid's algorithm**.

- Now, if they are relatively prime, how do we compute their inverse? Well, we saw how to compute their inverse using the **Extended-Euclid algorithm**.

So, these are the key algorithms that we're gonna use in our RSA cryptosystem -- Euclid's algorithm, Extended-Euclid algorithm and this Fast modular exponentiation. Now, we can dive into their RSA cryptosystem.