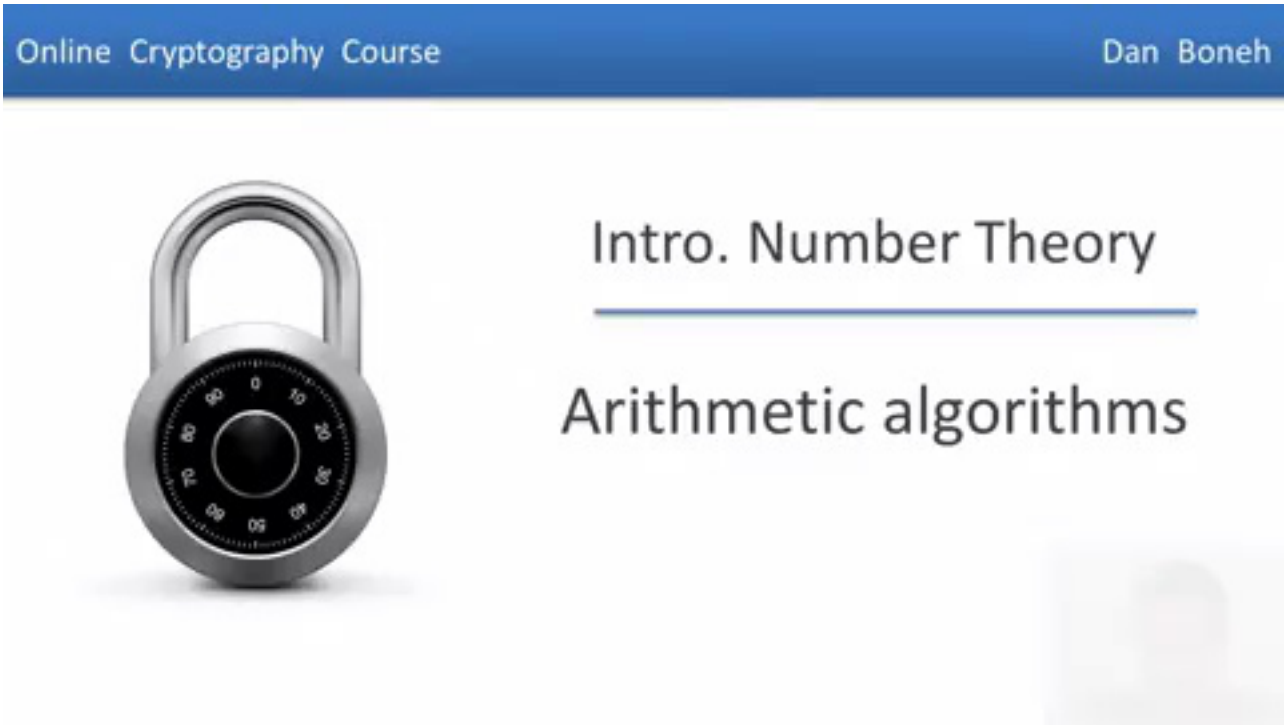
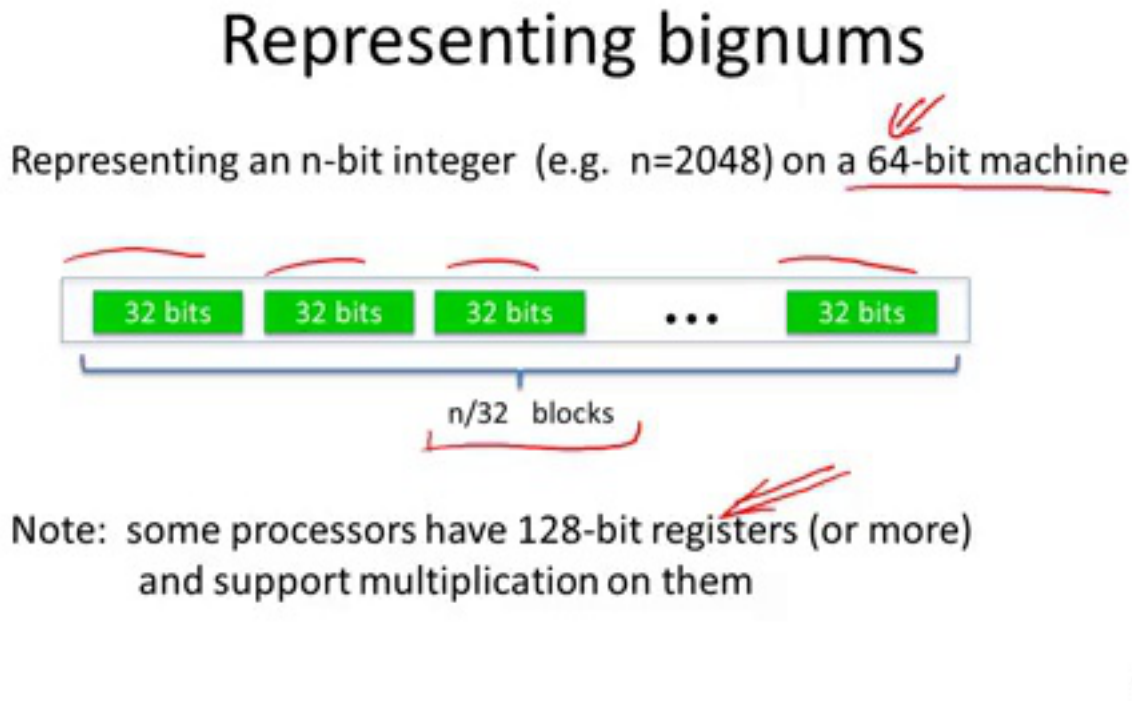
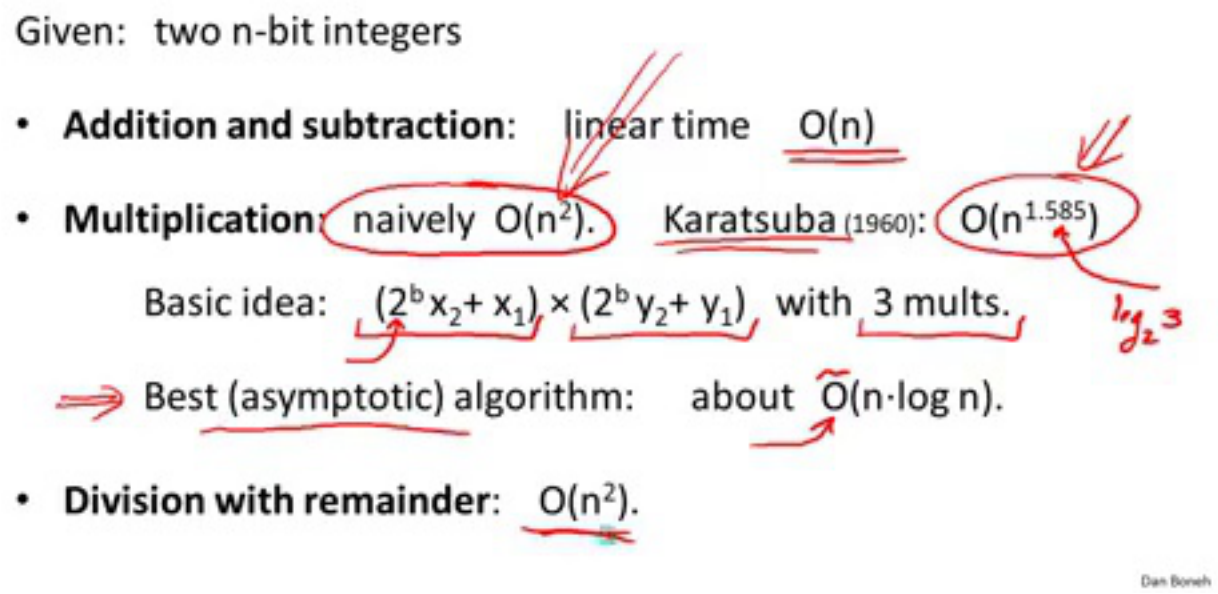



The next thing we're going to look at is how to compute modular large integers.

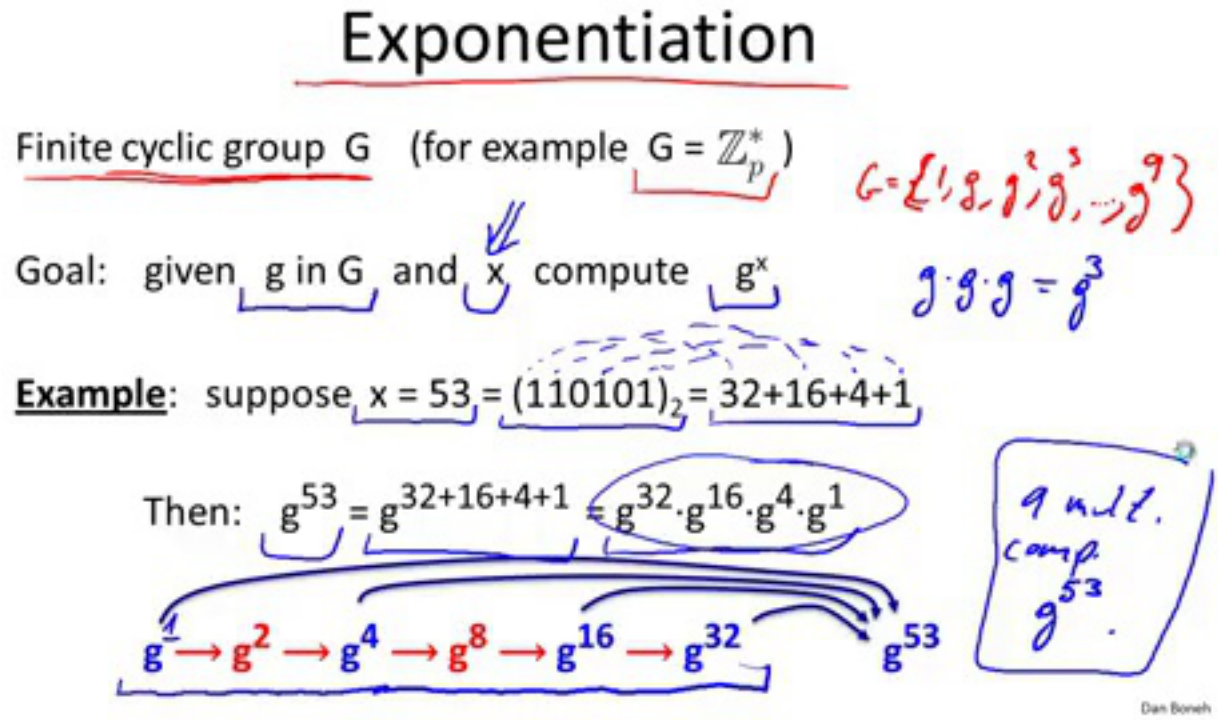



So the first question is how do we represent large integers in a computer? So that's actually fairly straightforward. So imagine we're on a 64 bit machine, what we would do is we would break the number we want to represent, into 32 bit buckets And then, we will basically have these n/32 bit buckets, and together they will represent the number that we want to store on the computer. Now, I should mention that I'm only giving 64 bit registers as an example. In fact, many modern processors have 128 bit registers or more, and you can even do multiplications on them. So normally you would actually use much larger blocks than just 32 bits. The reason, by the way, you want to limit yourself to 32 bits is so that you can multiply two blocks together, and the result will still be less than 64 bits, less than the word size on the machine.



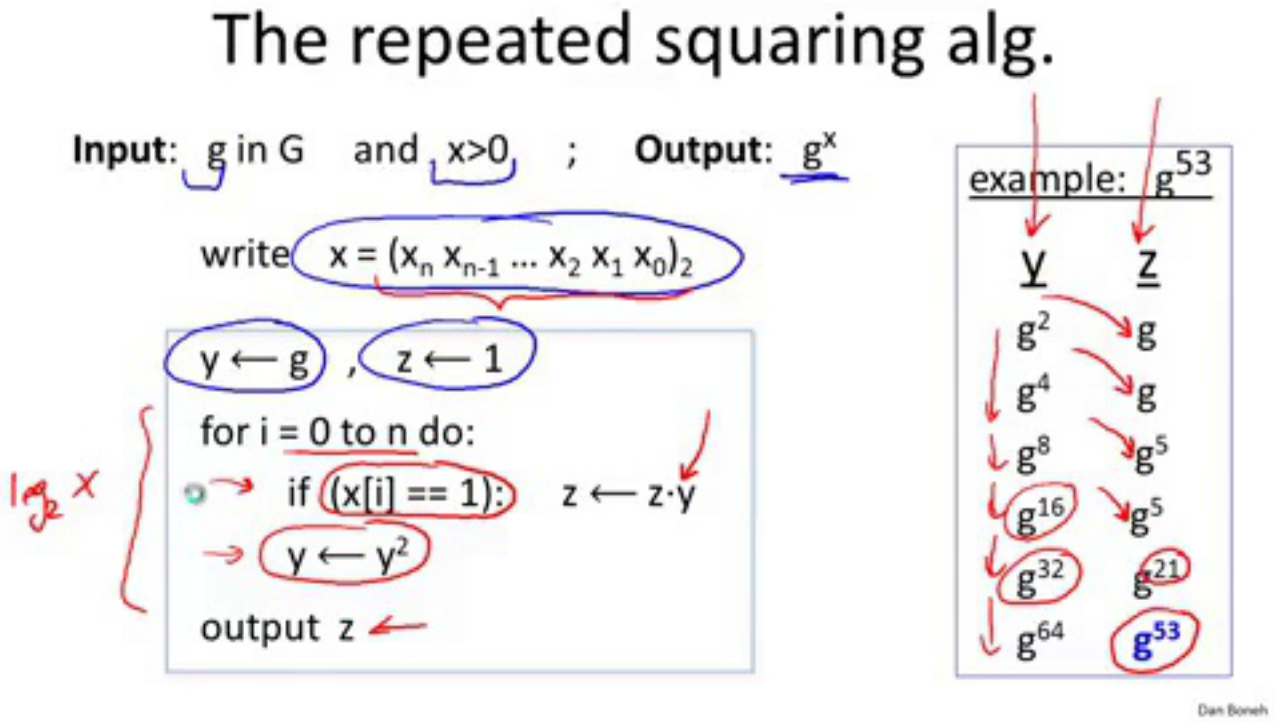

So now let's look at particular arithmetic operations and see how long each one takes. So addition and subtraction basically what you would do is that addition would carry or subtraction would borrow and those are basically linear time operations. In other words, if you want to add or subtract two n bit integers the running time is basically linear in n. Multiplication naively will take quadratic time. In fact, this is what's called the high school algorithm. This is what you kind of learned in school, where if you think about this for a minute you'll see that, that algorithm basically is quadratic in the length of the numbers that are being multiplied. So a big surprise in the 1960s was an algorithm due to Karatsuba that actually achieves much better than quadratic time in fact, it achieved a running time of n to the 1.585. And there's actually no point in me showing you how the algorithm actually worked, I'll just mention the main idea What Karatsuba realized, is that in fact when you want to multiply two numbers, you can write them as, you can take the first number x, write it as 2 to the b times x2 plus x1. Where x2 and x1 are roughly the size of the square root of x. Okay, so we can kind of break the number x into the left part of x and the right part of x. And basically, you're writing x as if it was written base 2 to the b. So it's got two digits base 2 to the b. And you do the same thing with, y. You write y base 2 to the b. Again, you would write it as, the sum of the left half plus the right half, And then, normally, if you try to do this multiplication, when you open up the

parentheses. You see that, this would require 4 multiplications, right? It would require x2 times y2, x2 times y1, x1 times y2, and x1 times y1. What Karatsuba realized is there's a way to do this multiplication of x by y using only three multiplications of x1 x2 y1 y2. So it's just a big multiplication of x times y only it takes three little multiplications. You can now recursively apply exactly the same procedure to multiplying x2 by y2, and x2 by y1, and so on and so forth. And you would get this recursive algorithm. And if you do the recursive analysis, you will see that basically, you get a running time of n to the 1.585. This number is basically, the 1.585 is basically, log of 3 base 2. Surprisingly, it turns out that Karatsuba isn't even the best multiplication algorithm out there. It turns out that, in fact, you can do multiplication in about nlog(n) time. So you can do multiplication in almost linear time. However, this is an extremely asymptotic results. The big O here hides very big constants. And as a result, this algorithm only becomes practical when the numbers are absolutely enormous. And so this algorithm is actually not used very often. But Karatsuba's algorithm is very practical. And in fact, most crypto-libraries implement Karatsuba's algorithm for multiplication. However, for simplicity here, I'm just gonna ignore Karatsuba's algorithm, and just for simplicity, I'm gonna assume that multiplication runs in quadratic time. But in your mind, you should always be thinking all multiplication really is a little bit faster than quadratic. And then the next question after multiplication is what about division with remainder and it turns out that's also a quadratic time algorithm.




So the main operation that remains, and one that we've used many times so far, and I've never, actually never, ever told you how to actually compute it, is this question of exponentiation. So let's solve this exponentiation problem a bit more abstractly. So imagine we have a finite cyclic group G. All this means is that this group is generated from the powers of some generator little g. So for example think of this group as simply ZP, and think of little g as some generator of big G. The reason I'm sitting in this way, is I'm, I want you to start getting used to this abstraction where we deal with a generic group G and ZP really is just one example of such a group. But, in fact, there are many other examples of finite cyclic groups. And again I want to emphasis basically that group G, all it is, it's simply this powers of this generator up to the order of the group. I'll write it as G to the Q. So our goal now is given this element g, and some exponent x, our goal is to compute the value of g to the x. Now normally what you would say is, you would think well, you know, if x is equal to 3 then I'm gonna compute you know, g cubed. Well, there's really nothing to do. All I do is I just do g times g times g and I get g cubed, which is what I wanted. So that's actually pretty easy. But in fact, that's not the case that we're interested in. In our case, our exponents are gonna be enormous. And so if you try, you know, think of like a 500-bit number and so if you try to compute g to the power of a 500-bit number simply by multiplying g by g by g by g this is

gonna take quite a while. In fact it will take exponential time which is not something that we want to do. So the question is whether even though x is enormous, can we still compute g to the x relatively fast and the answer is yes and the algorithm that does that is called a repeated squaring algorithm. And so let me show you how repeated squaring works. So let's take as an example, 53. Naively you would have to do 53 multiplications of g by g by g by g until you get to g by the 53 but I want to show you how you can do it very quickly. So what we'll do is we'll write 53 in binary. So here this is the binary representation of 53. And all that means is, you notice this one corresponds to 32, this one corresponds to 16, this one corresponds to 4, and this one corresponds to 1. So really 53 is 32 plus 16 plus 4 plus 1. But what that means is that g to the power of 53 is g to the power of 32+16+4+1. And we can break that up, using again, the rules of exponentiation. We can break that up as g to the 32 times g to the 16 times g to the 4 times g to the 1, Now that should start giving you an idea for how to compute g to the 53 very quickly. What we'll do is, simply, we'll take g and we'll start squaring it. So what square wants, g wants to get g squared. We square it again to get g to the 4, turn g to the 8. Turn g to the 16, g to the 32. So we've computed all these squares of g. And now, what we're gonna do is we're simply gonna multiply the appropriate powers to give us the g to the 53. So this is g to the one times g to the 4 times g to the 16 times g to the 32, is basically gonna give us the value that we want, which is g to the 53. So here you see that all we had to do was just compute, let's see, we had to do one, two, three, four, five squaring, plus four more multiplications so with 9 multiplications we computed g to the 53. Okay so that's pretty interesting. And it turns out this is a general phenomena that allows us to raise g to very, very high powers and do it very quickly.




So let me show you the algorithm, as I said this is called the repeated squaring algorithm. So the input to the algorithm is the element g and the exponent x. And the output is g to the x. So what we're gonna do is we're gonna write x in binary notation. So let's say that x has n bits. And this is the actual bit representation of x as a binary number. And then what we'll do is the following. We'll have these two registers. y is gonna be a register that's constantly squared. And then z is gonna be an accumulator that multiplies in the appropriate powers of g as needed. So all we do is the following we loop through the bits of x starting from the least significant bits, And then we do the following: in every iteration we're simply gonna square y. Okay, so y just keeps on squaring at every iteration. And then whenever the corresponding bit of the exponent x happens to be one, we simply accumulate the current value of y into this accumulator z and then at the end, we simply output z. That's it. That's the whole algorithm, and that's the repeated squaring algorithm. So, let's see an example with G to the 53. So, you can see the two columns. y is one column, as it evolves through the iterations, and z is another column, again as it evolves through the iterations. So, y is not very interesting. Basically, all that happens to y is that at every iteration, it simply gets squared. And so it just walks through the powers of two and the exponents and that's it. z is the more interesting register where what it does is it accumulates the appropriate powers of g

whenever the corresponding bit to the exponent is one. So for example the first bit of the exponent is one, therefore, the, at the end of the first iteration the value of z is simply equal to g. The second bit of the exponent is zero so the value of z doesn't change after the second iteration. And at the end of the third iteration well the third bit of the exponent is one so we accumulate g to the fourth into z. The next bit of the exponent is zero so z doesn't change. The next bit of the exponent is one and so now we're supposed to accumulate the previous value of y into the accumulator z so let me ask you so what's gonna be the value of z? Well, we simply accumulate g to the 16 into z and so we simply compute the sum of 16 and 5 we get g to the 21. Finally, the last bit is also set to one so we accumulate it into z, we do 32 plus 21 and we get the finally output g to the 53. Okay, so this gives you an idea of how this repeated squaring algorithm works. It's is quite an interesting algorithm and it allows us to compute enormous powers of g very, very, very quickly. So the number of iterations here, essentially, would be log base 2 of x. Okay. You notice the number of iterations simply depends on the number of digits of x, which is basically the log base 2 of x. So even if x is a 500 bit number in 500 multiplication, well, 500 iterations, really 1,000 multiplications because we have to square and we have to accumulate. So in 1,000 multiplications we'll be able to raise g to the power of a 500 bit exponent.

# Running times

Given n-bit int. N:

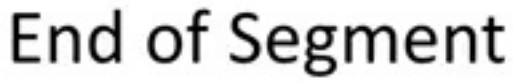
- **Addition and subtraction in $Z_N$:** linear time $T_+ = O(n)$
- **Modular multiplication in $Z_N$:** naively $T_\times = O(n^2)$
- **Modular exponentiation in $Z_N$ ($g^x$):**

$$O((\log x)\cdot T_\times) \le O((\log x)\cdot n^2) \le O(n^3)$$

$x < N$

Dan Boneh

Okay so now we can summarize kind of the running times so suppose we have an N bit modulus capital N as we said addition and subtraction in ZN takes linear time. Multiplication of just, you know, as I said, Karatsuba's actually makes this more efficient, but for simplicity we'll just say that it takes quadratic time. And then exponentiation, as I said, basically takes log of x iterations, and at each iteration we basically do two multiplications. So it's O(log (x)) times the time to multiply. And let's say that the time to multiply is quadratic. So the running time would be, really, N squared log x. And since x is always gonna be less than N, by Fermat's theorem there's no point in raising g to a power that's larger than the modulus. So x is gonna be less than N. Let's suppose that x is also an N-bit integer, then, really exponentiation is a cubic-time algorithm. Okay so that's what I wanted you to remember, that exponentiation is actually a relatively slow. These days, it actually takes a few microseconds on a modern computer. But still, microseconds for a, for a, say a four gigahertz processor is quite a bit of work. And so just keep in mind that all the exponentiation operations we talked about. For example, for determining if a number is a quadratic residue or not, All those, all those exponentiations basically take cubic time.




Okay, so that completes our discussion of arithmetic algorithms, and then in the next segment we'll start talking about hard problems, modulo, primes and composites.