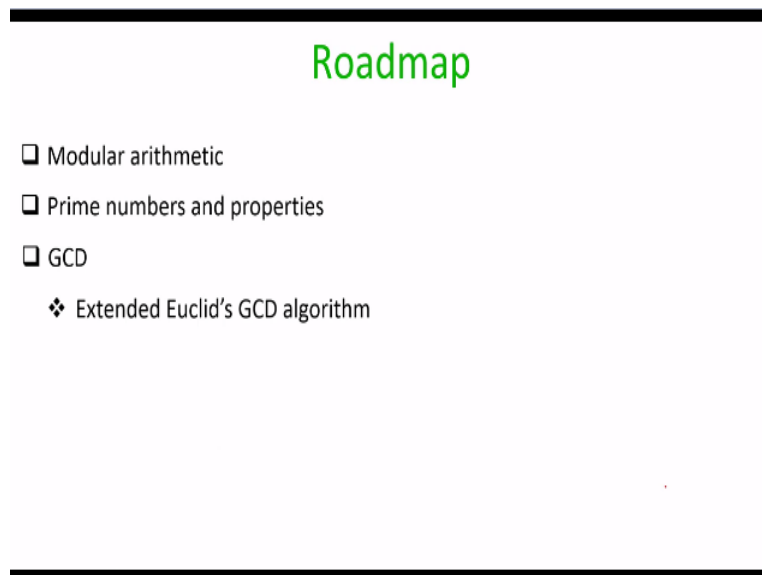


Foundations of Cryptography
Dr. Ashish Choudhury
Department of Computer Science
International Institute of Information Technology – Bengaluru

Lecture – 55
Number Theory

Hello everyone. Welcome to this lecture. Just a quick recap, in the last lecture we concluded our discussion on Public-key cryptography. So in this lecture the roadmap is as follows.

(Refer Slide Time: 00:29)



We will basically discuss some of the main results related to number theory which we have used during our extensive discussion on public-key cryptography. I stress that we will not be having a full-fledged discussion on number theory because I personally feel that as; in a course on foundations of cryptography we should basically just use the important facts from number theory. But I thought that probably we should have a very high-level discussion on some of the important results which we used extensively during our discussion on public-key cryptography.

So motivated by that we will have this discussion; today's lecture is based completely on the number theory where we will discuss about Modular arithmetic, Prime numbers and their properties and we will see the Extended Euclid's GCD algorithm.

(Refer Slide Time: 01:28)

Modular Arithmetic

□ Let $a, N \in \mathbb{Z}$, with $N > 1$. Then:

$[a \bmod N] \stackrel{\text{def}}{=} r$ such that $0 \leq r \leq N - 1$ and $a = qN + r$, for some integer q

❖ Ex: $[5 \bmod 4] = 1$

❖ Ex: $[-11 \bmod 3] = 1$, as $-11 = 3(-4) + 1$

➤ $[-11 \bmod 3] \neq -2$, even though $-11 = 3(-3) - 2$

□ If $[a \bmod N] = [b \bmod N]$, then a is said to be congruent to b modulo N

❖ Denoted as $a \equiv b \pmod{N}$ --- a and b are mapped to the same remainder

❖ $[a \equiv b \pmod{N}] \Leftrightarrow (a - b)$ is divisible by N

So let us start with Modular Arithmetic. So imagine you are given 2 numbers or 2 integers a, N belonging to the set of integers \mathbb{Z} . And here N is the modulus where $N > 1$. Then $a \bmod N$ is defined to be the remainder r , such that the remainder r is in the range 0 to $N - 1$, such that the relationship $a = q$ times $N + r$ holds for some integer q . If that is the case, then we say a modulo N is r .

So for instance, if I want to find out 5 modulo 4 then 5 modulo 4 is 1 , because if I divide $5/4$ I get the remainder 1 . In the same way, if I want to find out -11 modulo 3 then -11 modulo 3 is 1 because 1 is in the range 0 to 2 and -11 can be related to 3 by this relationship namely I can say that -11 is 3 times $-4 + 1$. It turns out that I can write -11 as a linear combination of my modulus 3 in another form as well, namely I can say that $-11 = (3 \text{ times } -3) - 2$.

And hence one might feel that -11 modulo 3 should be -2 but that is not that case because as per our definition the remainder r should be in the range of 0 to $N - 1$. That is why -11 modulo 3 is 1 and not -2 . Now we have the definition of $a \bmod N$; so if we have two integers a and b such that both a as well as b gives you the same remainder modulo, the modulus N then we say that a is congruent to b and we use the notation that $a \equiv b \pmod{N}$.

So this means that a and b are equivalent in the sense that they give you the same remainder when you divide them by the modulus N . So as per the definition of $a \bmod N$ it turns out that,

if a is congruent to b modulo N then this is possible if and only if the difference of a and b is completely divisible by N . Because if a gives the remainder r modulo N and since a is congruent to b modulo N that means b modulo N is also the same remainder r and if I subtract b from a then the remainder r and r cancels out and what I obtain is that $a - b$ is completely a multiple of N and hence it will be completely divisible by N .

(Refer Slide Time: 03:58)

Arithmetic Rules for Modular Arithmetic

❑ Let $[a \bmod N] = a'$ and $[b \bmod N] = b'$. Then:

- ❖ $[a + b \bmod N] = [a' + b' \bmod N]$
- ❖ $[a - b \bmod N] = [a' - b' \bmod N]$
- ❖ $[a \times b \bmod N] = [a' \times b' \bmod N]$

❑ Reduce and then add/subtract/multiply

➤ Instead of add/subtract/multiply and then reduce }

❑ Example: Compute $[1093028 * 190301 \bmod 100]$

➤ Option I : first compute $1093028 * 190301$ and then reduce mod 100

➤ Option II : first reduce 1093028 and 190301 mod 100 and get 28 and 1 respectively. Then compute $28 * 1$ and reduce mode 100

So we have some well-known arithmetic rules for modular arithmetics. So for instance we have two numbers a and b and if I know that a modulo N is a' and b modulo N is b' then I can say that $a + b$ modulo N will be the same value as you obtained by doing the operation $a' + b'$ modulo N . The same is true for subtraction as well as multiplication. So the way you can interpret these rules is that you can imagine as if you can first perform the modulars or you can first do the reduction modulo, the modulus N .

And then you can perform the addition, subtraction, multiplication operation instead of adding or subtracting or multiplying and then doing the reduction modulo N . To be more specific for instance imagine I want to compute the value of this number namely I want to perform the product of 1093028 and 190301 modulo 100. If I want to compute this value, then there are two ways to do that.

The first way will be that I first perform the product here or first compute the product here and then I do the reduction modulo 100. But this will be a complicated operation because multiplying these two large numbers will be really challenging. I cannot do it easily on a notebook. On the other hand, the same operation I can perform by first reducing the individual values here namely 1093028 and 190301 modulo 100. And reducing each of the modulo 100 is very simple.

Because I just have to focus on the last two digits here which I get as 28 and 1 and then I can multiply 28 and 1 and then again reduce it, reduce the answer modulo 100. And whatever obtained by option 1 and whatever result I obtained by option 2 they will be same as per the our rules of arithmetic rules of modular arithmetic. So this is a very powerful trick or powerful concept applicable in the context of modular arithmetic where you should; where; instead of applying the operation and then doing the modular or reduction modulo N and you can first reduce the operands modulo N and then you can perform the operation and if required you can again perform a reduction modulo N.

(Refer Slide Time: 06:13)

Modular Division

❑ Let $[a \bmod N] = a'$ and $[b \bmod N] = b'$. Then:

$$\left[\frac{a}{b} \bmod N \right] = \left[\frac{a'}{b'} \bmod N \right] ?$$

❖ **Not necessarily** --- in fact $\left[\frac{a}{b} \bmod N \right]$ is **not always well defined** !!

❑ Ex: $[3 \bmod 4] = 3$
 $[5 \bmod 4] = 1$

$\left[\frac{3}{5} \bmod 4 \right] = ?$

$\left[\frac{3}{1} \bmod 4 \right] = 3$

$ac = bc \Rightarrow a = b$

❑ In modular arithmetic:

$$[a \cancel{c} \bmod N] = [b \cancel{c} \bmod N] \not\Rightarrow [a \bmod N] = [b \bmod N]$$

So we have seen the addition, subtraction and multiplication modulo N and what about modular division. So the question is can we say the following, imagine a modulo N is a' and b modulo N is b' then can we say that a divided by b modulo N will be the same as a'/b' modulo N. Answer is no, because in fact in turns out that operation a divided by b modulo N is not at all well-defined.

So to demonstrate my point imagine my a and b are 3 and 5 and my modulus is 4 so I am given 3 modulo 4 is 3 and 5 modulo 4 is 1 and I know that 3 divided by 1 modulo 4 what have given me 3 but if I want to calculate 3 divided by 5 modulo 4 I do not have the answer, because this operation 3 divided by 5 modulo 4 is not at all well-defined. That means in modular arithmetic I cannot say that, if I am given that the product of ac modulo N and the product bc modulo N are same.

Then I cannot say that cancel out c from both the sides and it cannot get implication that a modulo $N = b$ modulo N , which would have been possible if I had performed the same operation in regular arithmetic and c would not; and c is not 0. That means in the regular arithmetic if I know that $ac = bc$ and c is not 0 then I can say that if I divide both the sides by c then I obtain $a = b$. But the same thing I cannot conclude in the modular arithmetic because modular division is not well - defined.

(Refer Slide Time: 07:50)

Algorithms for Modular Arithmetic

- ❑ Inputs: n -bit positive integers a, b and N
- ❑ Operations: $[(a + b) \bmod N]$, $[(a - b) \bmod N]$, $[(a \cdot b) \bmod N]$, $[a^b \bmod N]$
- ❑ Complexity measurement: Number of bit operations as a function of n


- ❑ Operations: $[(a + b) \bmod N]$, $[(a - b) \bmod N]$, $[(a \cdot b) \bmod N]$
- ❖ Computable using $\mathcal{O}(\text{poly}(n))$ number of bit operations

- ❑ **Modular exponentiation** : how to compute $[a^b \bmod N]$?

$$\underbrace{((a \cdot_N a) \cdot_N a) \dots \cdot_N a}_{(b-1) \text{ modular multiplications}}$$

$$\mathcal{O}(\underbrace{b}_{\approx 2^n}) \cdot \text{poly}(n) \text{ bit operations}$$

$$\approx \mathcal{O}(\underbrace{2^n}_{\text{poly}(n)}) \cdot \text{poly}(n) \text{ bit operations}$$



So now let us discuss some of the algorithms for modular arithmetic. So imagine we are given two operands a and b each of size n - bit and the modulus N whose size is also n -bit, and some of the common operations which we encountered in public-key cryptography are to perform the modular addition, multiplication, subtraction and the exponentiation. So remember modular

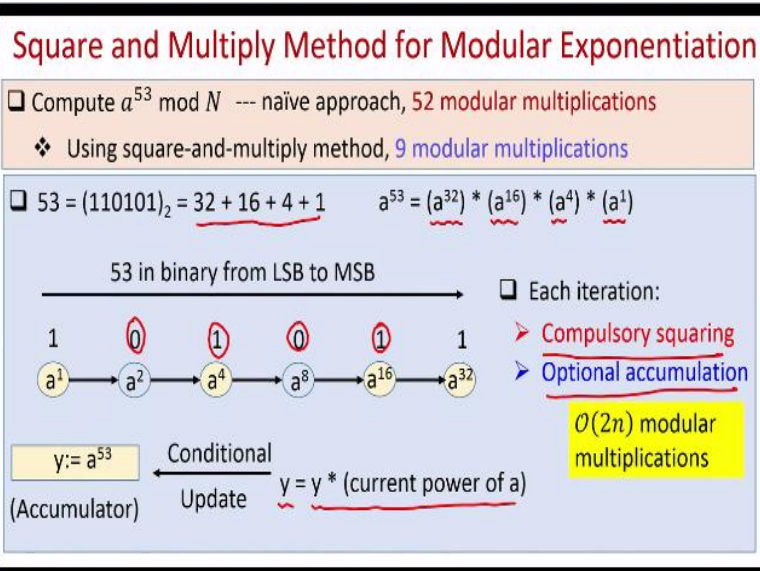
exponentiation is a very important operation in almost every instantiation of public-key cryptosystem, where we encountered modular exponentiation.

So we now want to have algorithms to perform this modular arithmetic operations, such that the resultant algorithms, their complexity, their time complexity should be some polynomial function in the number of bits namely n . So it turns out that if I want to perform modular addition, subtraction and multiplication I can perform them in $\text{poly}(n)$ number of bit operations.

But what about modular exponentiation, imagine I want to compute a^b modulo N , a naive algorithm will be that I perform a^2 modulo N whatever is the result I multiply it again with a and do a modulo N , and like that I have to perform basically $b - 1$ modular multiplications and since each modular multiplication requires a $\text{poly}(n)$ number of operations and I am performing b such order of b such operations one may say that, the overall time complexity of this naive modular exponentiation algorithm will be $O(b * \text{poly}(n))$ - bit operations and hence it is overall polynomial.

But that is not correct because what exactly is the value of b here. I have to express the value of b also as some function of n and it turns out that the magnitude of b here is some exponential function in number of bits n . Basically b can be as large as 2^n . That means, this naive modular exponentiation algorithm; its time complexity is exponential function and the number of bits that you need to represent your a , b and the modulus N and that is why it is the exponential time algorithm. We cannot use this naive exponentiation algorithm to perform; to compute a^b modulo N .

(Refer Slide Time: 10:16)



So now what we are going to discuss is we will see a poly time algorithm to compute modular exponentiations. So this method is called as the Square and Multiply method. And to demonstrate the algorithm I will take an example here. I want to compute a to the power 53 modulo N . Remember, the naïve approach will be to multiply a 52 times and each; with each multiplication you do a mod N operation.

So basically, the naïve approach will require you to perform 52 modular multiplications. But what we are going to see here is that using the square and multiply method we can perform the same computation with just 9 modular multiplications. So the idea behind this square and multiply method is as follows. So I express my exponent 53 in this case in binary. And depending upon the bit representation of 53 wherever I have the bit value 1 I have to take that power of 2.

Wherever the bit power is 0 I have to exclude that power of 2 and hence I can say that 53 depending upon its binary representation can be expressed as summation of these powers of 2. And hence I can say that a to the power 53 is nothing but the product of some selective powers of a namely a to the power 32, a to the power 16, a to the power 4 and a to the power 1. So the idea behind this square and multiply method as the name suggests is that in each iteration we will accumulate or we will compute the successive powers of a .

And that means, we will start with a power 1 and then we will go to a square then from a square we will go to a to the power 4 then from a to the power 4 we will go to a to the power 8 and so on and then selectively depending upon the current bit in the binary representation of the exponent namely 53 in this case we will decide whether to accumulate the current power of a or not, and that is why the name square and multiply.

In a more detailed way imagine I write 53 in this binary representation from LSB to MSB in this form. And as I go from LSB to MSB the different powers of a that I obtain as I go from 1 current bit to the next bit is just a square of the previous power, right and it is easy to see that my goal is to compute a to the power 53 and a to the power 53 basically can be computed by just multiplying some selective powers of a namely the powers of a corresponding to the bit positions where in the binary representation of 53, where I have the bit 1.

So based on this idea the algorithm is as follows. So we set an accumulator here which will have the final answer. The final answer that I want to compute is a to the power b or a to the power 53 in this case. So I initialize my accumulator to be 1 and once I perform the all the iterations my accumulator will have the final answer. So in each iteration what I am going to do is, I am going to do a conditional update, namely I will do the operation that $y = y$ into current power of a depending upon whether my current bit in the exponent b which I am exploring right now is 1 or not.

So for instance, currently I start with the LSB of 53 or the exponent b, right now the bit is 1 that means I have to take accumulate this power and that is why I will update my accumulator by existing y and the current power of a and then I go to the next power of a. And I check the bit position, bit position is 0; bit is 0 so I do not need to accumulate this power. Then I go to the next power of a which will be a to the power 4 and I check the current bit, it is one.

That means I have to accumulate this power, so that is why I will update my accumulated by multiplying the current content of the accumulated with the current power of a. Then I go to the next power of a which I do not need to include or accumulate because the bit position in the

exponent is 0 then I go to the next power of a and I check the bit position of the exponent in its binary representation it is 1, so I have to accumulate.

And hence I update my accumulator, and then I finally go to the MSB of the bit representation of my exponent, it is 1, that means I have to accumulate the current power of a and hence that is my final answer. So I am not writing the exact pseudo code here. You can understand; I hope you will be able to write down the pseudo code here. But idea here is that I will need to perform a sequence of iteration, the number of iterations is nothing but the number of bits that you need to represent your exponent b .

And in each iteration we have to do a compulsory squaring to compute the next power of a . And an optional accumulation depending upon whether the current bit in the binary representation of the exponent b is 0 or 1 and it turns out that in the worst case my exponent b could be such that all the bits in its binary representation is 1, that means in the worst case in each iteration this optional accumulation has to be performed compulsory.

But even in that case the total number of modular multiplication that we end up performing is two times n which is some polynomial function in the number of bits that you need to represent your n , your modulus and your numbers a and b and hence this is a poly time algorithm and this is the algorithm which we use to perform modular exponentiation in all our public-key cryptosystems.

(Refer Slide Time: 15:33)

Prime Numbers

□ An integer $p > 1$ is called prime if the only positive factors of p are 1 and p

❖ A positive integer that is greater than 1 and not prime is called composite

□ Fundamental theorem of arithmetic:

❖ Every integer greater than 1 can be written uniquely as a prime or as the product of two or more primes, where the prime factors are written in order of non-decreasing size

$$\forall n \geq 1: n = 2^{b_1} \cdot 3^{b_2} \cdot 5^{b_3} \dots, \text{ where each } b_i \geq 0$$

□ The infinitude of primes:

❖ There are infinitely many primes }

Now let us discuss about prime numbers. So the definition of prime numbers is that an integer $p > 1$ is called a prime number if its only positive factors are the number itself and 1. And a number positive integer greater than 1 which is not prime is called as composite number. And a well - known theorem of arithmetic which is also often called as the fundamental theorem of arithmetic is that any number $n \geq 1$ can always be expressed as product of different powers of prime and this is true for every $n \geq 1$.

This is a well-known fact which we can prove using induction and not going into the exact proof. And there is another well-known fact from number theory which says that, there are infinitely many primes. It is not the case that you have only finite number of primes. And again this theorem we can prove using contradiction or any proof method. So there are several well-known proofs to prove the fact that there are infinitely many primes. I am not going into the details of the proof here.

(Refer Slide Time: 16:39)

Primality Testing : Naïve Algorithm

❑ Theorem: If p is composite then it has a divisor less than or equal to \sqrt{p}

- ❖ Let p be composite
- ❖ p has a factor, say a , where $1 < a < p$
- ❖ Since a is a factor, we have $p = ab$, where $b > 1$
- ❖ Either $a \leq \sqrt{p}$ or $b \leq \sqrt{p}$
 - If not then $ab > p$, which is a contradiction

2002

Aks | Miller-Rabin

❑ Primality-testing algorithm (p):

- ❖ For $i = 2, \dots, \sqrt{p}$
 - If i divides p , then output p is composite
- ❖ Output p is prime

❑ Running time:

- ❖ $O(\sqrt{p})$ divisions
- ❖ $O(2^{\frac{n}{2}})$ divisions, where p is an n -bit number

So if you recall that in our RSA cryptosystem and also in the context of Elgamal encryption scheme, we basically require as a part of the setup description of a prime number to be available to all the parties and the prime number should be sufficiently large. So the question is how exactly we pick the prime numbers. That means, we need an algorithm to check whether a randomly chosen n - bit number is a prime number or not.

And there is a naive algorithm for doing that. And the naive algorithm is based on the fact that if a number p is composite then it has at least one of the divisors which is less than or equal to \sqrt{p} and the proof of this theorem is very simple because if you have all the divisors of a composite number p greater than \sqrt{p} . Say you have two such factors a and b and neither a nor b is less than or equal to \sqrt{p} and both a and b are the factors of p and p is composite then it turns out that product ab will be greater than p which is a contradiction.

So that means if at all you have a composite number p it is bound to have at least one of the factors in the range 1 to \sqrt{p} . And based on this observation we can have this following Primality-Testing algorithm. So imagine you are given a number p and you want to verify whether the number p is prime or not and say the number p is an n - bit number. So the naive algorithm is basically you check whether there exist any divisor i in the range 2 to \sqrt{p} for the number p .

Because if indeed p would have been composite its bound to have at least one of the divisors in the range 2 to \sqrt{p} that means it will have at least one divisor i in the range 2 to \sqrt{p} and that is what you are searching in this naive algorithm. It turns out that the running time of this algorithm is $O(\sqrt{p})$ divisions because you are performing \sqrt{p} number of divisions in this algorithm.

And since p is n - bit number the magnitude of p is 2^n so basically you are performing 2 to the power $n / 2$ divisions in this naive algorithm. And hence this is an exponential time algorithm which we cannot use in practice to test whether a given number p is prime or not, if my n is sufficiently large. So it was really a very challenging problem to verify whether a given number p is prime or not in polynomial amount of time.

In fact, people believed that it is not possible to come up with the poly time algorithm to check whether a given number p is prime or not. But history was made in year 2002 where a group of computer scientists from IIT-Kanpur namely Manindra Agrawal, Neeraj Kayal and Nitin Saxena. They came up with a poly time algorithm, polynomial in the number of bits that you need to represent with your prime p , or number p .

And they came up that this poly time algorithm which tells you whether a given number p is prime or not and that algorithm is now well - known as AKS algorithm. So you can use that algorithm to check whether a given number p is prime or not. But it turns out that we do not use the AKS algorithm because even though it is a poly time algorithm the underlying polynomials are sufficiently large and that prevents us from deploying this algorithm as it is in practice.

Instead what we do in reality to check whether a given number is prime or not is we use an algorithm which is called as Miller-Rabin primality testing algorithm. And it is a randomized algorithm in the sense that you cannot always trust the output of this algorithm. In the sense that, for 99.99% cases it will give you the right answer whereas there is a small error probability in the output of this algorithm.

That means even though your input may not be a prime number it may end up labeling the number as a prime number. So in that sense it is an error prone algorithm. And the reason we use

Miller - Rabin test to check whether a given number is prime or not is that its running time is super, super fast compared to your AKS algorithm. I stress that the AKS algorithm is completely error - free.

You can completely trust the outcome that is given by AKS algorithm because it is a deterministic algorithm. There is no randomization involved as part of the algorithm.

(Refer Slide Time: 21:08)

Greatest Common Divisor (GCD)

- ❑ $\text{GCD}(a, b)$: a and b non-zero integers
 - ❖ Greatest integer which divides both a and b
- ❑ Integers a and b are **relatively prime (co-prime)** if $\text{GCD}(a, b) = 1$
- ❑ Integers a_1, a_2, \dots, a_n are **pair-wise relatively prime** if $\text{GCD}(a_i, a_j) = 1$, for all $1 \leq i < j \leq n$
- ❑ Computing $\text{GCD}(a, b)$ using prime-factorization
 - ❖ Let $a = p_1^{a_1} p_2^{a_2} \dots p_n^{a_n}$ ❖ Let $b = p_1^{b_1} p_2^{b_2} \dots p_n^{b_n}$
 - $\text{GCD}(a, b) = p_1^{\min(a_1, b_1)} p_2^{\min(a_2, b_2)} \dots p_n^{\min(a_n, b_n)}$
 - ❖ How to find the prime-factorization of a and b ?

Now let us discuss about Greatest Common Divisor or GCD. So if a and b are two non - zero integer then the GCD of a and b is the greatest integer which is a common factor for both a as well as b . And if we have 2 numbers or 2 integers a and b whose GCD is 1 then we say that the integers a and b are co - prime. And if we have such several pairs of such numbers or several integers say a_1 to a_n we say that they are pair-wise co-prime or relatively prime if pair-wise the GCDs are 1 for every pair of integer a_i, a_j in the list. So if we want to out the GCD of 2 numbers a and b then the naive algorithm will be that I first find out the prime factorization of a because as per the fundamental theorem of arithmetic I can express a as the product of powers of prime. And in the same way I can express my number b as a product of powers of prime.

And then I can say that GCD of a and b is nothing but I take the minimum exponents from individual prime factorization of a and b and arrange them and that will give me the GCD of a

and b. But the problem with this algorithm is that how at the first place I find the prime factorization of a and b.

(Refer Slide Time: 22:31)

Euclid's GCD Algorithm

□ Let $a = bq + r$, where a, b, q and r are integers. Then $\text{GCD}(a, b) = \text{GCD}(b, r)$

Algorithm GCD (a, b)
// a, b : positive integers, $a > b$

```

x = a,    y = b
While y ≠ 0
    r = x mod y
    x = y
    y = r
Return x

```

□ **Lame's Theorem:** If $\text{GCD}(a, b)$ needs n divisions $\Rightarrow b \geq f_{n+1}$

□ (Lower bound): $\forall n \geq 3: f_n > \alpha^{n-2}$,
where $\alpha = \frac{1+\sqrt{5}}{2}$

$\text{GCD}(a, b)$ needs at most $5 \cdot \log_{10} b$ divisions

So it turns out that we can do something very interesting here. We can use an efficient algorithm to compute GCD algorithm and this algorithm is called as Euclid's GCD algorithm. And base for Euclid's algorithm is the following fact. So if you are given integers a and b such that a is (b times q) + r. And if your goal is to find GCD of a and b then GCD of a and b is nothing but GCD of b and r, if $a = (b \text{ times } q) + r$.

Again I am not giving the proof of this fact. You have to believe me. But based on this we can find out an algorithm to compute the GCD of a and b. And basically, what we do in this algorithm is we set my x value to a and y value to b and iteratively I find the value of x modulo y, I set my current y to be the next x and I take the remainder of my current x modulo y to be next y. And I perform this operation till my y becomes 0.

As soon as my y becomes 0 which will eventually happen, I obtain the x at that point to be the GCD of a and b. And a correctness of this algorithm follows from the fact that GCD of a and b will be same as GCD of b and r and GCD of b and r will be same as GCD of r and b modulo r and so on. That means at every step I am basically reducing the value of so called b and eventually it will become 0.

So if you are wondering that what is the time complexity or how many modular divisions I am performing here; how many mod operations I am performing inside this Euclid's algorithm. I can come up with an exact bound using well - known Lamé's theorem which says that if the GCD of a, b is computed using Euclid's algorithm and it needs n number of divisions then your b should be at least $n + 1$ th Fibonacci numbers namely b should be greater than or equal to $f_{(n+1)}$, so here $f_{(n+1)}$ denotes a Fibonacci $n + 1$ Fibonacci number.

If you are wondering what is the Fibonacci number, it is a sequence starting with 0 the next number is one. These are the first two Fibonacci numbers. And then if I want to compute the value of the i^{th} Fibonacci number, the i^{th} Fibonacci number is basically the summation of the previous Fibonacci number and then previous to previous Fibonacci number. So that is my Fibonacci sequence here.

So what Lamé's theorem basically says is that, if the Euclid's GCD algorithm would have performed n number of divisions then the magnitude of b will be at least as large as the $n + 1$ Fibonacci number. And there also exists well - known lower bounds on the value of the n^{th} Fibonacci number. It says, the lower bound says that for every end of the value of n^{th} Fibonacci number is at least α times $n - 2$ where α is also called as the golden ratio namely $\frac{1+\sqrt{5}}{2}$.

So based on these two facts I can say that if my GCD of a, b needs n number of divisions then it is utmost the n will be utmost five times $\log_{10}(b)$. Hence, asymptotically the number of divisions that we need to perform in this algorithm is proportional to the number of bits that we need to represent the value b and hence the overall algorithm is an efficient algorithm.

(Refer Slide Time: 25:57)

GCD as a Linear Combination

□ **Bezout's Theorem:** There exists integers s and t , such that $\text{GCD}(a, b) = sa + tb$

❖ How to explicitly find the Bezout's coefficients (s and t)?

❖ By doing some extra "book-keeping" in Euclid's algorithm

Extended Euclid's algorithm

➤ At each step, express the remainder in terms of a and b

□ Ex: Find the Bezout's coefficients for $a = 252$ and $b = 198$

$$252 = 1 \cdot 198 + 54$$

$$198 = 3 \cdot 54 + 36$$

$$54 = 1 \cdot 36 + 18$$

$$36 = 2 \cdot 18$$

$$54 = 252 - 1 \cdot 198$$

$$36 = 198 - 3 \cdot 54$$

$$18 = 4 \cdot 252 - 5 \cdot 198$$

Actual algorithm makes only a forward pass

It turns out that we can use the Euclid's GCD algorithm to do something more. So to understand that let me first go to here Bezout's theorem here. So what Bezout's theorem says is that, you can always express the GCD of two numbers a and b as the linear combination of the numbers a and b . That means you can always come up, you can always find out linear combiner s and t , such that you can express the inputs a and b as a linear combination which will give you the GCD of a and b .

That means the GCD of a and b can be always expressed as a linear combination of the inputs a and b where the combiners s and t always exists. And if you are wondering how exactly you can find out these Bezout's coefficients s and t or the combiners s and t where you can do that by performing some extra bookkeeping or maintaining some extra variables in Euclid's algorithm. And this extended algorithm where you perform the extra bookkeeping activity to find out the linear combiners s and t is also known as the Extended Euclid's algorithm.

So I will not go into the full details of how exactly this extra bookkeeping happens in the Euclid's algorithm. Let me demonstrate it. So suppose you want to find out the linear combiners or Bezout's coefficients for the numbers a to be 252 and b to be 198. So if you see the way the GCD algorithm will operate, the Euclid's GCD algorithm will operate for the input a to be 252 and b to be 198 is as follows.

In the first iteration your a will be 252, b will be 198 and you will obtain the value of 252 modulo 198. In the next iteration your a will become 198 and your b will become the current remainder 54 and so on. And you do this operation till your remainder become 0. And when your remainder becomes 0 you know that 18 is your GCD. So now your goal is to find out the linear combiners so that you can represent the GCD 18 as the linear combination of 252 and 198.

And for that what we can do is we can back track here and we can see that I can rewrite 18 to be; based on this equation, I can rewrite 18 to be 54 times 1 - 36. And if I go back once step above then I know that my 36 is nothing but $198 - 3 \text{ times } 54$. So if I come back here and I substitute the value of 36 here then I obtain that 18 can be rewritten as a linear combination of 54 and 198, but that is not my goal.

What I can do is again I can go back one step up and I can rewrite my 54 as a linear combination of 252 and 198. And if I substitute that linear combination in this last equation I end up getting that 18 is represented as a linear combination of 252 and 198, that means my Bezout's coefficients are 4 and - 5. So in this demonstration actually to find out the Bezout's coefficients I did a back tracking but in the actually Euclid's algorithm you need to only the forward pass, you do not need to do a back track into, do the bookkeeping and find out the Bezout's coefficients.

(Refer Slide Time: 29:21)

Multiplicative Inverse Modulo N

- Recall : $a \cdot_N b \stackrel{\text{def}}{=} [a \cdot b \bmod N]$
- Integer b is called **multiplicative inverse of a modulo N** , if $a \cdot_N b = 1$
 - ❖ We use the notation $b = a^{-1}$
- If b is the multiplicative inverse of a modulo N , then a is the multiplicative inverse of b modulo N
 - ❖ If $b = a^{-1} \Rightarrow a = b^{-1}$
- If $b = a^{-1}$, then so are $b \pm k \cdot N$, for all $k \in \mathbb{Z}$
 - ❖ $[a \cdot (b \pm k \cdot N)] \bmod N$ $= \{[a \cdot b \bmod N] \pm [a \cdot k \cdot N \bmod N]\} \bmod N$
 $= \underline{[a \cdot b \bmod N]} = 1$
 - ❖ If multiplicative inverse exists then they are **infinite in numbers**

So how exactly this Extended Euclid's GCD algorithm will be useful. So it will be useful to compute multiplicative inverse modulo N . So recall how did; the way we define multiplicative modulo N operation. So $a * b$ modulo N is defined to the remainder of a into b modulo N and recall that we define an integer b to be the multiplicative inverse of a modulo N if the product of a and b modulo N gives you 1.

And we use the notation $b = a^{-1}$ to denote the multiplicative inverse of a modulo N and the basic fact here is that if b is multiplicative inverse of a modulo N then a is also the multiplicative inverse of b modulo N and it turns out that if indeed we have one multiplicative inverse of a modulo N , namely if you have b such that a times b modulo N is 1 then so is $b + \text{any multiple of the modulus } N$.

That means it does not matter whether you add multiples of modulus N to b or you subtract multiples of modulus N from b all of them also will be the multiplicative inverse of a modulo N and this is because of the fact that a times $b + -k$ times N ends up finally giving you $a * b$ modulo N ; and since b is multiplicative inverse of a modulo N , a times b modulo N is nothing but 1. That means if at all if multiplicative inverse exists then they are infinite in number.

(Refer Slide Time: 30:53)

Multiplicative Inverse Modulo N

❑ Let $a \in \mathbb{Z}$ and N be a positive integer. Then a^{-1} exists iff $\text{GCD}(a, N) = 1$

❑ **Sufficiency proof** (Using Bezout's theorem)

- ❖ Let $\text{GCD}(a, N) = 1$
- ❖ Using Extended-Euclid algorithm, find **Bezout's coefficients** s, t , such that:
$$as + Nt = \text{GCD}(a, N) = 1$$
- ❖ Taking **mod N** on both the sides
$$[as + Nt] \bmod N = 1$$
- ❖ Since $[Nt \bmod N] = 0$, we get
$$[as \bmod N] = 1 \Rightarrow s = a^{-1}$$

Now the question is when exactly can we say that multiplicative inverse of number a modulo N exists. So there is a well - known result in number theory which says that, if you are given a

modulus N and an integer a then the multiplicative inverse of a modulo N exists if and only if a is co-prime to n namely if the $\text{GCD}(a, N)$ is 1 and we can prove the sufficiency part of this theorem. I will not go proof of the necessity of this condition here.

I will prove that indeed it suffice if a is co - prime to N to find out the multiplicative inverse of a modulo N and that can be computed using Bezout's theorem. So imagine a is relatively prime to N that means GCD of a and N is 1. That means since GCD of a and N is 1 that means if I run the extended Euclid's algorithm I can find out Bezout's coefficients namely the linear combiners s and t , such that the GCD of a and N namely 1 can be expressed as the linear combination of a and N using the combiners s and t .

And now if I take mod N on both the sides of this equation then on the right side I obtain one modulo N and 1 modulo N will give me 1. But my left - hand side namely a times s + N times t modulo N will give me a times s modulo N because N times t modulo N will give me 0 because it is the multiple of N . So what I obtain by taking mod N on both the sides is that I obtain the relationship that a times s modulo $N = 1$ which shows that s is actually the multiplicative inverse of a modulo N .

And that is how we can obtain the; or you can compute the multiplicative inverse of a number a if it is co-prime to N . So that brings me to the end of this lecture. Just to summarize and this lecture we just discussed on a very high-level some of the basic; some of the important facts from number theory which we used extensively during our discussion on public-key cryptography. Specifically, we saw; discussed about modular arithmetic.

We saw a poly time algorithm to compute modular exponentiation which is a key operation which we perform in any public - key cryptosystem or any encryption algorithm in the public-key cryptosystem. We also discussed about a Primality testing, properties of prime number. And we also discussed about Extended Euclid's Algorithm to compute the GCD of two numbers and to compute Bezout's coefficients will be useful to compute multiplicative inverse of numbers. Thank you.