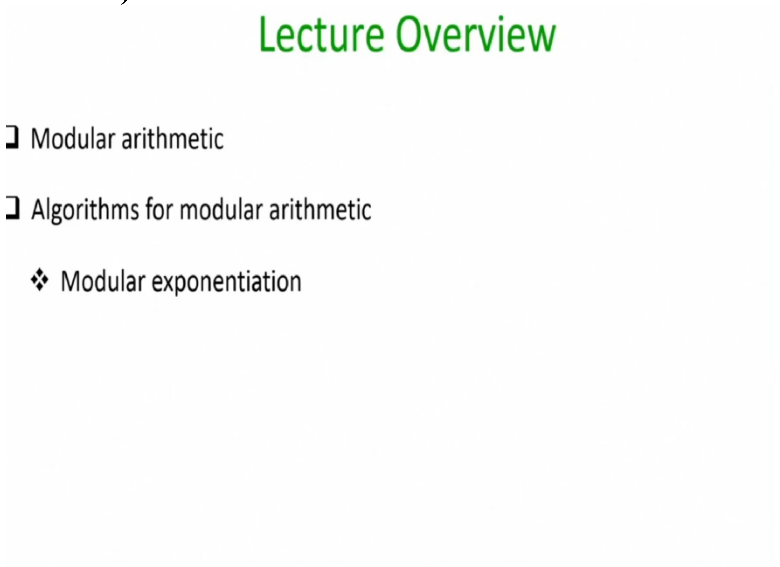
Discrete Mathematics

Prof. Ashish Choudhury
International Institute of Information Technology - Bangalore

Lecture - 55
Modular Arithmetic

Hello, everyone, welcome to this lecture. So, we will now shift our focus on number theory. And number theory in itself is a very fascinating topic we can have a full course on number theory. But we will be discussing only the relevant topics in number theory that is useful in the context of computer science namely modular arithmetic, properties of prime numbers, algorithm related to GCD and this will be very useful especially in cryptography.

(Refer Slide Time: 00:54)

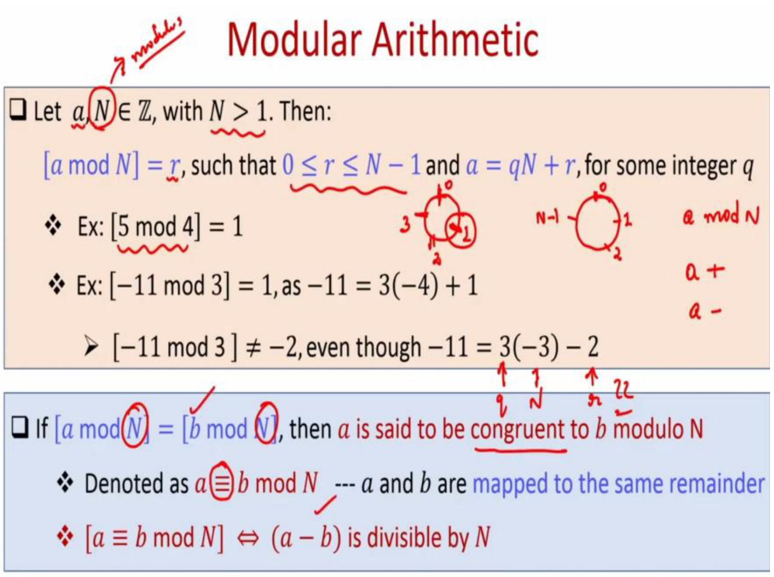


Lecture Overview

- Modular arithmetic
- Algorithms for modular arithmetic
  - Modular exponentiation

So, the plan for this lecture is as follows. We will discuss about modular arithmetic and various algorithms for doing modular arithmetic, especially modular exponentiation, which is a very central or important operation in cryptographic algorithms.

(Refer Slide Time: 01:07)



Modular Arithmetic

- Let $a, N \in \mathbb{Z}$, with $N > 1$. Then:
  - $[a \bmod N] = r$, such that $0 \le r \le N-1$ 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 $a \equiv b \bmod N$ --- a and b are mapped to the same remainder
  - $[a \equiv b \bmod N] \Leftrightarrow (a - b)$ is divisible by N

modulus
0
1
2
3
N-1
0
1
2
a mod N
a +
a -
q
N
r
Z

So, let us begin with modular arithmetic. So, imagine you are given a value a, which is an integer and you are given a modulus N. So, N will be called as the modulus where the modulus is positive, then a modulo N is r, where r will be a value in the range 0 to N – 1, such that a is equal to some quotient times the modulus plus your r. So basically r is the remainder which you obtain after dividing a by N where the remainder is in the range 0 to N - 1. So, for instance, 5 modulo 4 is 1 and -11 modulo 3 is 1.

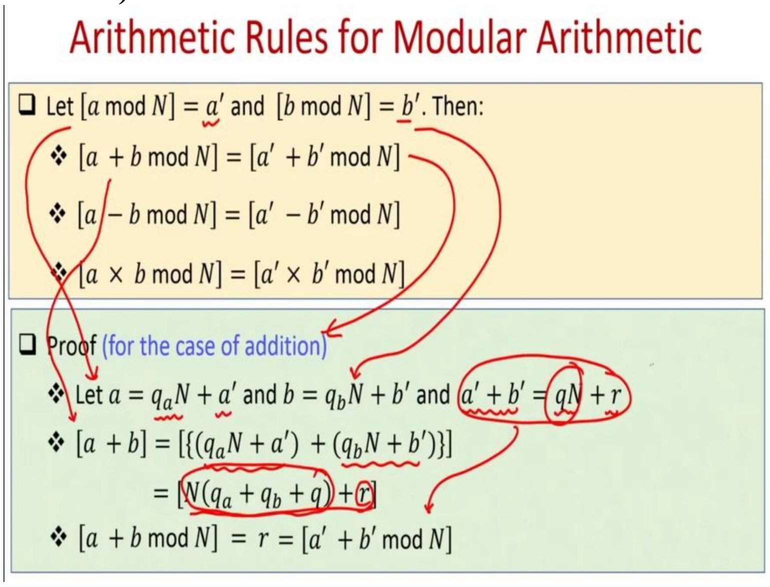
So, you might be wondering why not -11 modulo 3 is -2 because I can write -11 as q times my N and r, why cannot I do that? Well, the requirement here is that the remainder should be in the range 0 to N - 1. So, very easy way to visualize modular arithmetic is the following. So, you can imagine that you have a clock with the N marks. So, basically, the possible remainders which you can obtain after dividing any value by N.

And now, if you want to find out a modulo N. Where a is positive, then you start at 0 and you count N times along this clock, wherever you stop, that will be the value of your a modulo N. So, for instance, if you want to do it for 5 mod 4, and the possible remainders which you can obtain are 0, 1, 2 and 3, you want to find at 5 mod 4. So, you start at 0 so, this is 1, 2, 3, 4, 5 you stop at 1. So, that is why 5 modulo 4 is 1 whereas, if you are a is negative, then you have to do the same process but in an anti-clockwise direction.

Now, let us define what we call as congruence with respect to modulo. So if you have a modulus N, then a will be; then we say that a is congruent to b modulo N if the remainder that we obtained by dividing a by N is same as, the remainder that we obtained by dividing b by N, that means if a modulo and b modulo N are same then I will say that a is congruent to b modulo N and the notation that we use is the following : [$a \equiv b \bmod N$].

So, in some sense, you can imagine that I am trying to say that a and b are equivalent, in the sense that they give you the same remainder on getting divided by N. And it is very easy to verify that if a is congruent to b modulo N, then that is possible if and only if a - b is completely divisible by N. It is a very simple exercise to do that I am leaving that for you. So we will use multiple definitions of a congruent b modulo and we can use this definition or we can also use the fact that a congruent to b modulo N means a - b is completely divisible by N.

(Refer Slide Time: 04:44)



Arithmetic Rules for Modular Arithmetic

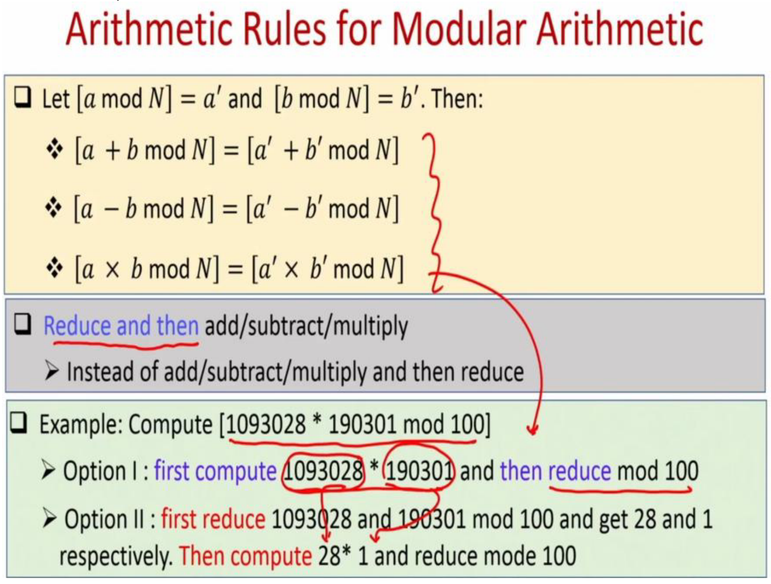
- Let $[a \bmod N] = a'$ and $[b \bmod N] = b'$. Then:
  - $[a + b \bmod N] = [a' + b' \bmod N]$
  - $[a - b \bmod N] = [a' - b' \bmod N]$
  - $[a \times b \bmod N] = [a' \times b' \bmod N]$
- Proof (for the case of addition)
  - Let $a = q_a N + a'$ and $b = q_b N + b'$ and $a' + b' = qN + r$
  - $[a + b] = [\{(q_a N + a') + (q_b N + b')\}]$
    $= [N(q_a + q_b + q) + r]$
  - $[a + b \bmod N] = r = [a' + b' \bmod N]$

So there are some interesting arithmetic rules which your modular arithmetic follows. So imagine that a modulo N is a' and b modulo N is b'. Then a + b modulo N will be the same as a' + b' modulo N, a - b modulo N will be same as a' - b' modulo N so on and similarly, a multiplied by b modulo N is same as a' into b' and then you take modulo N. So, let us prove these rules, we will just prove it for the case of addition, for the other operations you can prove in the similar way.

So, since a modulo N is a' that means, I can say that a is some quotient times N + a' and since b modulo N is b' I can say that b is some another quotient times N + b' and imagine that a' + b' is some quotient times N + r. Let us see the left hand side of this rule; addition rule it is a + b modulo and that means, you are adding a + b and then you are taking modulo. So, I am just expanding a and b here and I can take out N common and then I can substitute that a' + b' is some q times N + r.

So, this will be the overall value of a + b then what will be the remainder if I divide this value by N. So, this term will vanish because on dividing it will give the remainder 0 and you will get r. So, a + b modulo N will be r and same value you will obtain by dividing a' + b' and then dividing a' + b' by N. Because if you divide a' + b' by N this term will cancel out remember it will vanish and you will be left with the remainder.

(Refer Slide Time: 07:07)



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 mod 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, what will be the interpretation of these arithmetic rules the interpretation here will be the following that you can first reduce the operands namely a and b modulo N and then you can perform the plus operation, subtraction operation, multiplication operation and if again required and you can do a modulo instead of first adding and then taking a modulo. So, what I am saying is that you first reduce and then do the operation and that will have the same effect as if you are performing the operation and then you are reducing.

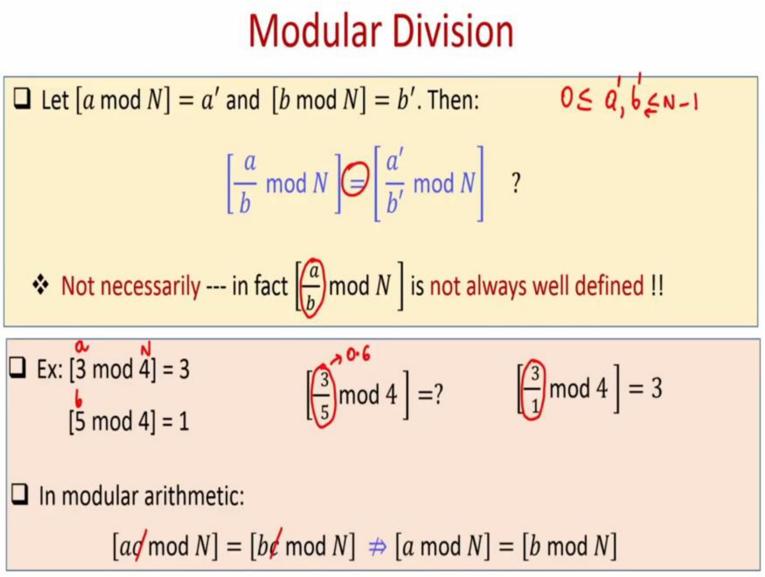
So, to make my point more clear, imagine that I want to ask you compute this. So, one approach will be that you first compute the product of these 2 large numbers and then take a modulo 100 that will give you an answer. But that will require some effort you cannot do it easily using your paper and pen. Option 2 will be as per this multiplication rule that I reduce the operand a modulo 100.

So, if I reduce a modulo 100 I get 28 and I reduce b modulo 100 and I will get 1 and then to find out a b modulo N it is sufficient to find out a' b' modulo N so, a' b' are now very small numbers which I can easily multiply and I will get 28 and I do not need to reduce it modulo

100 because 28 is already less than 100. So, my answer will be 28 which I will obtain very easily.

So, these are some interesting rules which we will be encountering again and again that means, if I want to do some modular arithmetic I can always reduce the operands first and then apply the operation and then again if required I can reduce instead of doing the operation and then applying the modulus.

(Refer Slide Time: 09:14)



Modular Division

❑ Let $[a \bmod N] = a'$ and $[b \bmod N] = b'$. Then: $0 \le a', b' \le N-1$

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

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

❑ Ex: $[\overset{a}{3} \bmod \overset{N}{4}] = 3$

$[\overset{b}{5} \bmod 4] = 1$

$\left[\frac{3}{5} \xrightarrow{} 0.6 \bmod 4\right] = ?$

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

❑ In modular arithmetic:

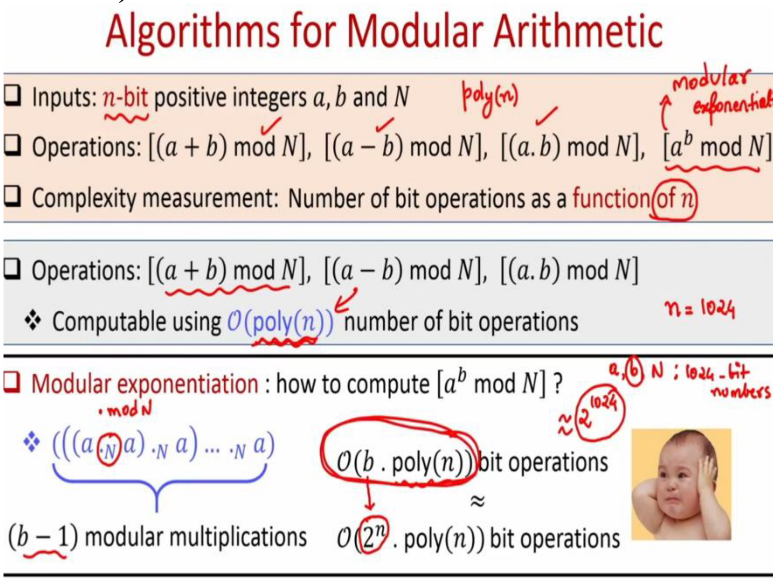
$$[a\cancel{c} \bmod N] = [b\cancel{c} \bmod N] \not\Rightarrow [a \bmod N] = [b \bmod N]$$

Now, we have seen the rules for addition, subtraction, multiplication, what about division? So, imagine a modulo N is a’ and b modulo N is b’ of course, a’ and b’ are in the range 0 to N - 1. Now, what can I say about a over b modulo N and a’ over b’ modulo N. Can I say that these 2 expressions will be same? Well the answer is no because at the first place the value a over b modulo N may not be well defined.

Because a over b might be a fraction and maybe a real number for instance, if a is less than b, or even if a is greater than b, a over b may not be an integer value. So how exactly you define a over b modulo N. So let us see an example here to make my point more clear, imagine my a is 3, b is 5, N is 4. Now, 3 over 1 modulo 4 is 3, because 3 over 1 will be 3 and 3 modulo 4 will be 3. But what over 3 over 5 modulo 4, 3 over 5 is 0.6 and 0.6 modulo 4 well, it is not at all well-defined.

That means in modular arithmetic, if ac and bc are congruent to each other, that means ac modulo N is same as bc modulo N that does not necessarily mean that a and b are also congruent modulo N or equivalently I cannot say that you can cancel out c from both the sides. No, that is not necessarily is the case, only in some cases certain conditions you can cancel out c from both the sides and conclude that a and b are congruent modulo N but that may not be always the case.

(Refer Slide Time: 11:22)



Algorithms for Modular Arithmetic

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

- ❑ Operations: $[(a+b) \bmod N]$, $[(a-b) \bmod N]$, $[(a.b) \bmod N]$
  - ❖ Computable using $O(\text{poly}(n))$ number of bit operations $n = 1024$

- ❑ Modular exponentiation : how to compute $[a^b \bmod N]$? a, b N ; 1024-bit numbers
  - ❖ $\underbrace{(((a \overset{\cdot \bmod N}{\cdot_N} a) \cdot_N a) \ldots \cdot_N a)}_{(b-1) \text{ modular multiplications}}$ $O(b \,.\, \text{poly}(n))$ bit operations $\approx O(2^n \,.\, \text{poly}(n))$ bit operations $\approx 2^{1024}$

So, now, let us discuss some algorithms for modular arithmetic. And we will be seeing algorithms for addition, multiplication, subtraction and modular exponentiation. So, this is addition, this is subtraction, this is multiplication, and this is called modular exponentiation. This is a very important operation in cryptography and our inputs a, b and modulus N are all some n bit integers.

Now, what will be the complexity measurement? How exactly we judge whether our given algorithm that we designed for performing this modular arithmetic operations are efficient or not. Our complexity measurement will be how many operations are we performing as a function of the number of bits that we need to represent as integer values a, b and N. And we will be requiring algorithms where the number of operations that we perform is a polynomial function in n.

Because, typically, we prefer algorithms whose running time is polynomial function of your parameter; the parameter here is the size of your integer a, size of your integer b and the size of your modulus N, which is the number of bits that you need to represent those values which is n. We do not prefer any algorithm which is exponential time or sub exponential time in the number of bits. So, it turns out that addition, subtraction and modular multiplication all of them can be performed in polynomial in n number of bit operations.

So, for instance if you want to perform a + b modulo N. So, imagine that the bit representation of a is a_{n-1}, a_{n-2} and up to a_0 and bit representation of b is b_{n-1}, b_{n-2} up to b_0 now to add a + b you can perform the bits of a and b bit by bit and taking care of carry and all those things and then you will obtain a bit representation of a + b. And then, as we are performing addition of two n bit numbers that will require polynomial in n number of operations.

And then you have to do a + b modulo N that means, you have now got two n bit numbers. So, this is n bit, this is n bit and you have to add one n bit number by another n bit number which can again can be done in polynomial in n number of operations. Same you can perform a - b and then take modulo N you can multiply a and b and then take modulo N all of them will need polynomial in n number of operations: $\mathcal{O}(\text{poly}(n))$.

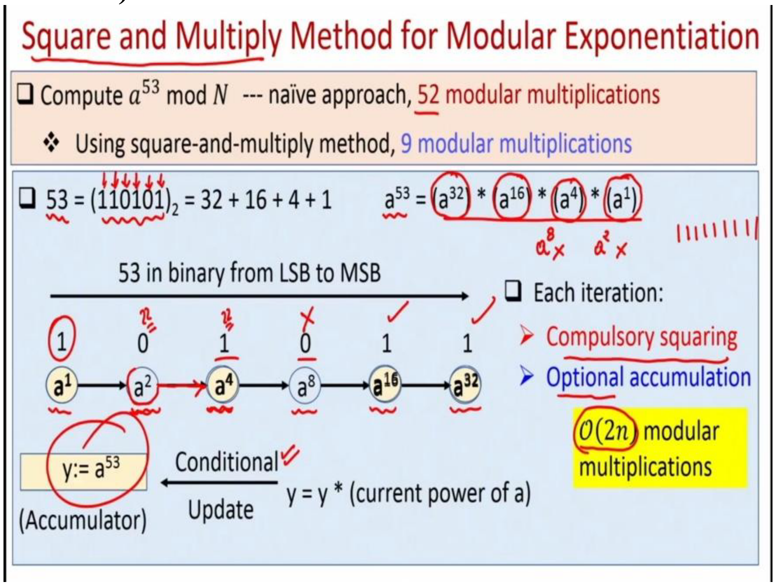
Now, what about modular exponentiation? How can we compute a^b modulo N? You might be wondering that why cannot I do the following multiply a with itself and then take mod and then again multiply with a and then take mod and so on. So, this operation DOT and then in subscript N, means I am doing multiplication modulo N ($\cdot_N$). So basically I am saying that you multiply a to itself b - 1 number of times and then keep on taking mod is that will be equivalent to saying that I am performing b - 1 number of modular multiplications.

And one instance of modular multiplication need these many number of bit operations, since I am doing it b number of times, this will be the overall complexity of my algorithm: $\mathcal{O}(b \cdot \text{poly}(n))$. And you might be saying this is a polynomial time algorithm. But if you see it closely, this is not a polynomial time algorithm. Because what exactly is b, b is an n bit number, that means magnitude wise, b could be as large as 2^n.

So, for instance, if you are a n is 1024 that means if your a is some 1024 bit number, your b is some 1024 bit number and your n is also some 1024 bit number. Then what I am basically saying is that this naive algorithm will require me to perform b times polynomial in n number of operations, but my b itself could be as large as 2 to the power 1024 bit number; those many, this is an enormously large quantity, you cannot even imagine how big this number is.

So, even though this might look like a polynomial time algorithm that is not the case, this is an exponential time algorithm.

(Refer Slide Time: 16:42)



Square and Multiply Method for Modular Exponentiation

- ❑ Compute a^{53} mod N --- naïve approach, 52 modular multiplications
  - ❖ Using square-and-multiply method, 9 modular multiplications

❑ $53 = (110101)_2 = 32 + 16 + 4 + 1$ $a^{53} = (a^{32}) * (a^{16}) * (a^{4}) * (a^{1})$

53 in binary from LSB to MSB

$$\begin{tikzcd} 1 & 0 & 1 & 0 & 1 & 1 \\ a^1 \arrow[r] & a^2 \arrow[r] & a^4 \arrow[r] & a^8 \arrow[r] & a^{16} \arrow[r] & a^{32} \end{tikzcd}$$

$y := a^{53}$ (Accumulator) ← Conditional Update $y = y *$ (current power of a)

❑ Each iteration:
- ➢ Compulsory squaring
- ➢ Optional accumulation

$O(2n)$ modular multiplications

So, now, we will see a very nice method, which is a polynomial time algorithm for performing modular exponentiation and this is called as the square and multiply approach why it is called square and multiply? It will be clear soon so let me demonstrate the algorithm. Suppose I want to compute a^{53} modulo N the naive approach will be you multiply a to itself and then take mod N.

And then again, you multiply the result with a and then again, take mod, that means you perform 52 modular multiplications that will be the naive approach. I am saying, do not do that, using square multiply method, we will be seeing how to compute a^{52} modulo N with just 9 modular multiplications. So now you can see the drop from 52 multiplications, I have brought down it to 9. So you can imagine the level by which it drops when N is an enormously large value.

So what I am saying is the naive algorithm will require you 2^{1024} bit operations if n would have been 1024, I would bring it down to only 1024 modular multiplications. So, you can see exponential drop in the number of modular multiplications that we require, if we follow this square and multiply approach. So the idea here will be the following, I will treat my exponent in binary form. So I will come up with its binary representation and the binary representation is this : 110101.

And now it is easy to see that a^{53} can be rewritten like this. So now, what it means is the following that I have to accumulate certain powers of a. I have to accumulate the first power of a. I have to accumulate the 4th power of a, I have to accumulate the 16th power of a, and I have to accumulate the 32nd power of a. Which powers of a I am not accumulating: a power 2 that I am not accumulating, I am not accumulating a power 8 and so on.

So, which powers of a I have to accumulate and which powers of a I have to leave that depends upon the binary representation of 53. So, you can see the positions at which the binary representation of my exponent was 0 the corresponding power of a I am excluding I am not accumulating and bit positions where it was 1, the corresponding powers of a I am accumulating that is the idea of square and multiply. So, I am writing my exponent from LSB to MSB. And my square and multiply approach will be an iterative algorithm.

What I will do is the following in each iteration, I will compute the next higher power of a by squaring the current power. So, I will start with a power 1 and if I square it, I will obtain a power 2, if I square it I will obtain a power 4, if I square it I will obtain a power 8, square it a power 16, square it a power 32. Of course all the things are performed modulo N. So when I am saying squaring that means I will be multiplying a power 1 with itself and then take modulo N that will give me a^2 modulo N.

I multiply a^2 with a^2 and do modulo N that will give me a^4 modulo and so on. And now what I will do is depending upon which bit of my exponent is 1 and which bit of my exponent is 0. I will determine whether to accumulate the corresponding power of a or not, that is the whole idea. So, what I will do is I will initialize an accumulator variable here which will have my final result stored.

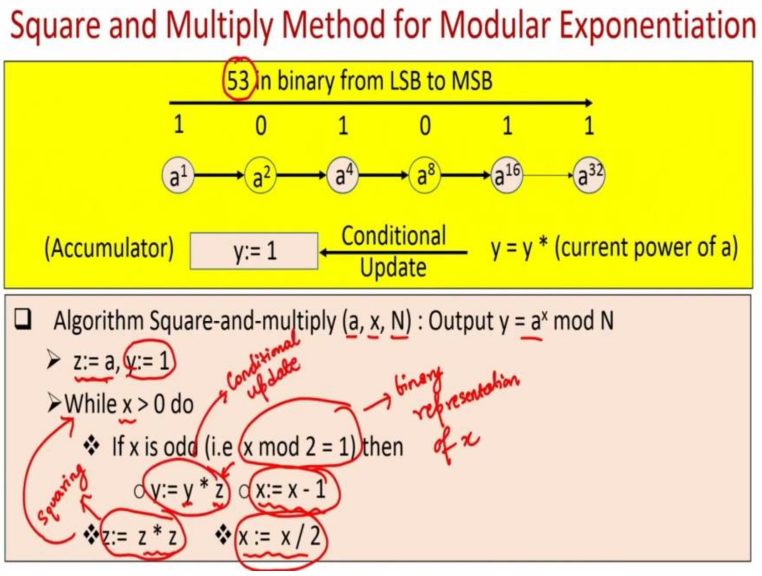
And as my algorithm proceeds the value of my accumulator will keep on changing depending upon how do I accumulate the different powers of a. And in each iteration there will be a conditional update depending upon whether I am accumulating the current power of a or not, and which depends upon my current bit in the exponent. So, for instance, I will start with a power 1 and I have to accumulate it because my current bit is 1.

So, I will accumulate it and my accumulator will become a power 1. Now, I will go to the next iteration and I will do a square to get the next power of a that will need one modular multiplication and I will obtain a^2 and I will check whether I need to accumulate this power or not, I do not need to accumulate. So, go to the next iteration obtain the next power of a, that requires doing 1 modular multiplication namely I have to multiply a^2 to itself and then take mod N I will obtain the next power of a namely a power 4 and I have to decide whether I have to accumulate this power or not, I have to so, I will do the conditional update, and my y will become a power 5, I go to the next power by doing modular multiplication, I have to exclude this. I go to the next power and I have to accumulate this. So, my conditional update will be triggered I go to the next power of a and then I have to accumulate this power of a.

So, my accumulator gets modified and then I am done with all the bits of my exponent and this will be my final answer. So, you can see that in each iteration I will be doing a compulsory squaring that is why the name square and why I have to do a compulsory squaring because I have to obtain the next higher power of a and there is an optional accumulation depending upon whether my current bit of the exponent is 0 or 1.

So, in the worst case, what can happen you will be definitely doing n number of compulsory squaring where n is the number of bits of your exponent. And it may so happen that your exponent has a bit representation all 1s in which case you will be doing accumulation n number of times. So in the worst case, you will be doing 2 times n number of modular multiplications to get your final answer.

(Refer Slide Time: 23:06)



Square and Multiply Method for Modular Exponentiation

53 in binary from LSB to MSB

$$\begin{array}{cccccc} 1 & 0 & 1 & 0 & 1 & 1 \\ a^1 \rightarrow & a^2 \rightarrow & a^4 \rightarrow & a^8 \rightarrow & a^{16} \rightarrow & a^{32} \end{array}$$

(Accumulator) $y := 1$ ← Conditional Update $y = y * (\text{current power of a})$

```
❑ Algorithm Square-and-multiply (a, x, N) : Output y = a^x mod N
  ➢ z:= a, y:= 1
  ➢ While x > 0 do
      ❖ If x is odd (i.e x mod 2 = 1) then
            o y:= y * z   o x:= x - 1
      ❖ z:= z * z   ❖ x := x / 2
```

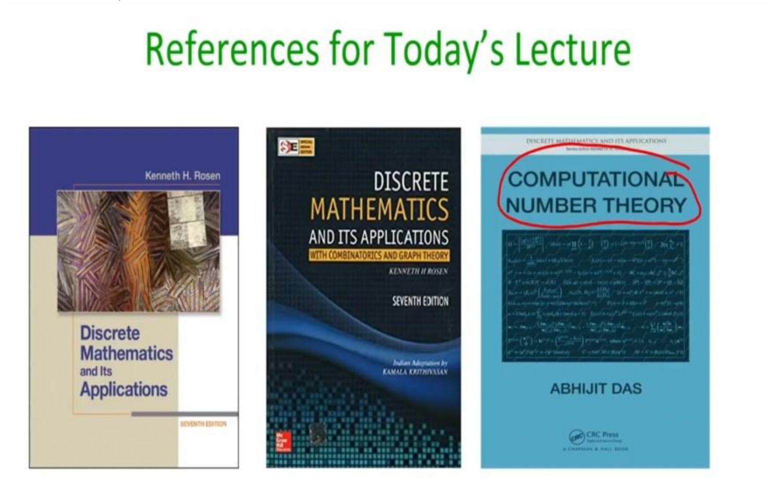
So that is the square and multiply approach and this is the pseudocode which is very simple. Let me go through the pseudocode 1 step at a time. So, your inputs will be our base exponent and the modulus and you need output a^x modulo N. So I do some initializations here so this y will be the accumulator and z will have the various powers of a as the algorithm proceeds.

Now what I have to do is, in the example I had the binary representation of my exponent available with me, but that you need to compute in the algorithm itself. So how do you compute the binary representation of the method that you are aware of namely, keep on dividing x and depending upon whether you get the remainder 0 or 1. You accumulate 0 or 1 in your binary representation, the same trick we will be doing here.

So what I do is the following till my exponent is non 0. I will do the following, I will check whether my x is odd or not. And that will determine, whether the next bit in the binary representation of my exponent is 0 or 1, if I go from LSB to MSB. So if it is 1, that means I have to modify my accumulators so this is the conditional update. So I update my accumulator by multiplying whatever is the current content of the accumulator with my current power of a.

And then I update my x to get my next bit in the binary representation of x. And I have to do a compulsory squaring to go to the next power of a and then I have to update my x. So that I can obtain the next bit in the binary representation of x and I have to do this process. So this step, coupled with this step, and checking whether x modulo 2 is 1 or not, this will give you the binary representation of x. This is the squaring step and this is the conditional update. So that is the square and multiply trick.

(Refer Slide Time: 25:44)



References for Today's Lecture

Kenneth H. Rosen
Discrete Mathematics and Its Applications
DISCRETE MATHEMATICS AND ITS APPLICATIONS
WITH COMBINATORICS AND GRAPH THEORY
KENNETH H ROSEN
SEVENTH EDITION
COMPUTATIONAL NUMBER THEORY
ABHIJIT DAS
CRC Press

So that brings me to the end of this lecture. So these are the references used for today's lecture. And for number theory, there are lots of nice texts available. I find his text very handy and very useful. It is very easy to understand and just to summarize, in this lecture we started our discussion on modular arithmetic. We saw the rules of addition, subtraction, multiplication we also saw that modular division is not well defined always.

We saw the square and multiply trick for doing modular exponentiation. The naive algorithm for doing the modular exponentiation will be an exponential time algorithm. So that is why we do not use that we instead use the square and multiply trick. Thank you.