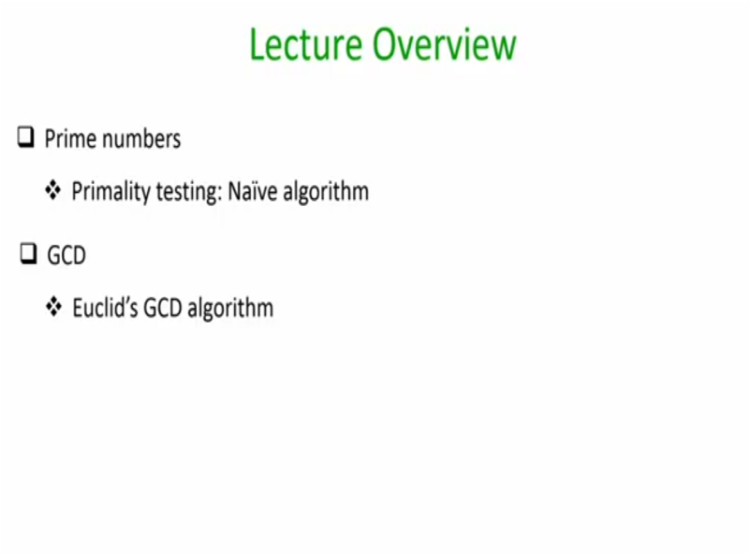
Discrete Mathematics
Prof. Ashish Choudhury
International Institute of Information Technology, Bangalore

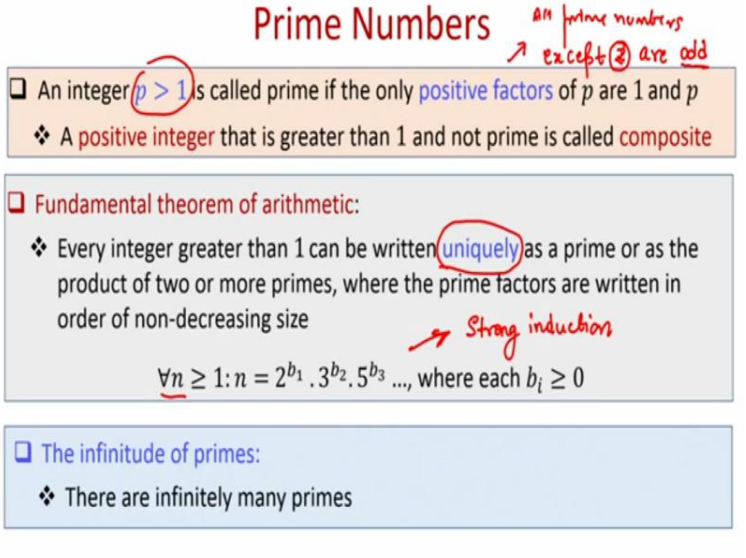
Lecture – 56
Prime Numbers and GCD

(Refer Slide Time: 00:22)



Hello everyone welcome to this lecture. The plan for this lecture is as follows. So, in this lecture, we will introduce prime numbers, we will see some properties of prime numbers, we will also discuss about the naive algorithm for primality testing, which is a very interesting computational problem. And we will discuss about GCD and Euclid's GCD algorithm.

(Refer Slide Time: 00:43)

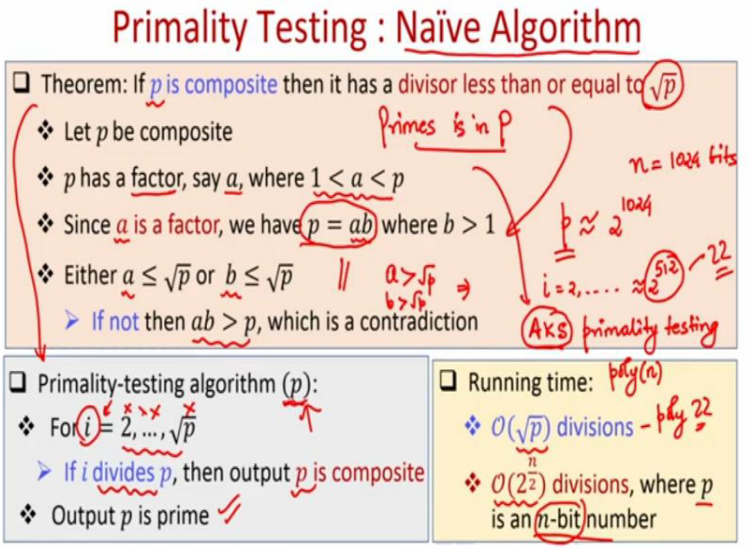


So, let us start with prime numbers. So, the definition which all of you are aware of, so, we will say an integer p which is greater than 1 is called a prime number, if the only positive factors of p are 1 and the number p itself. Whereas if the number p is not a prime number, then we call it as a composite number. It is easy to see that all prime numbers except 2 are odd. Because, if you consider an even number, 2 is always a factor of that even number.

But 2 is a prime number because the only factors of 2 are 1 and the number 2 itself. So, in some sense, except 2 all the prime numbers are odd, but we do have an even number which is also prime namely the number 2 so you can consider to be an odd prime. It is very odd that odd in the English sense, it is a very odd property that you have an even number, but it is still a prime number. So, we know some nice properties of prime numbers, which we had already discussed in this course.

So, we know the fundamental theorem of arithmetic, which says that you take any integer greater than equal to 1 it can be written down uniquely as product of prime powers. So how do we prove this? We proved it using strong induction, when we discussed proof by induction and we also know that there are infinitely many primes. So, we saw there are several proofs for this, but we discussed one of the proofs in this course earlier. So, these are 2 interesting properties of prime numbers, which you should keep in mind.

(Refer Slide Time: 03:12)



Now, the next question is how exactly we check whether a given number is a prime number or not. So, I will be discussing the naive algorithm, which many of you might be aware of. So, the algorithm uses the fact that if at all your given number p is a composite number then it will have a divisor which is less than equal to the square root of that number. And the proof is very simple. So, let p be a composite number and since p is a composite number it will have a factor say a.

And the factor will be definitely greater than 1 because the definition of composite number is that it should have at least 1 factor different from 1 and the number itself. So, the factor a will be greater than 1 and less than p. And then since a is a factor I can say that there is a b such that a times b = p that means that b is basically $\frac{p}{a}$. So, now, the claim of the theorem statement here is that either $a \leq \sqrt{p}$ or $b \leq \sqrt{p}$ and this can be easily proved using contradiction.

So, on contrary assume that $a > \sqrt{p}$ and $b > \sqrt{p}$. Then this gives me a contradiction that $ab > p$, which is not the case because my premise is that p is equal to the product of a and b. Based on this theorem, the naive algorithm is as follows. So, you are given an input number p you want to check whether it is prime or composite, you know that if at all it is composite, then it will have a factor within the range square root of that number.

So, that is what you try to do here you try to check whether it has at least whether you are given input number p has at least 1 factor i within the range 2 to $\sqrt{p}$. So, you range over all possible

values of i between 2 to square root of your number and check whether the number i divides p or not. If you encounter any i then you can declare that p is composite, but if none of the numbers 2, 3, 4, $\sqrt{p}$ divides your p then you can declare your p to be prime.

That is a naive primality testing algorithm that you are aware of. Now, what is the running time of this algorithm? So, how many operations you perform here, so, there are $\sqrt{p}$ iterations and in each iteration, you are performing a division. So, our measurement, our complexity measurement here will be how many divisions you are performing here to declare whether a given number p is prime or not. So, we will be needing $\sqrt{p}$ number of divisions.

So, you might be wondering, this is a polynomial time algorithm, but that is not the case. So, imagine p is represented by an n bit number then the magnitude of your $\sqrt{p}$ will be $2^{\frac{n}{2}}$, because your p could be as large as 2^n. So, $\sqrt{p}$ could be as large as $2^{\frac{n}{2}}$. So, it might look like a polynomial time algorithm, but it is not, it is an exponential time algorithm exponential in the number of bits that you need to represent your value p.

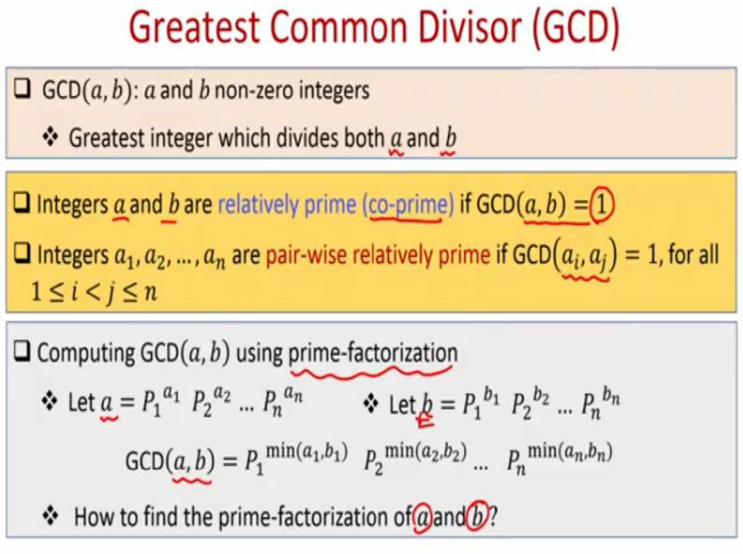
So again, take the case when n is equal to say 1024 bit, and say your p is as large as 2 to the power 1024 bit number, then basically your i is between 2 to something of order 512. That means you will be basically performing these many divisions: 2^{512} divisions to check whether a given number p of 1024 bit is a prime number or not. And this is an enormously large quantity; it is not a small quantity. So, this is not a polynomial time algorithm.

Now, you might be wondering that whether we have a polynomial time algorithm or not. And coming up with a polynomial time algorithm for checking whether a given number is prime or not had been a long standing open problem, people thought that we do not have any polynomial time algorithm. But in 2002 there was this algorithm proposed called as AKS primality testing, which is a polynomial time algorithm.

Polynomial in the number of bits that you need to represent your input number p and this is called Agarwal Kayal Saxena primality testing algorithm which in polynomial time can tell you

whether your given number p is a prime number or not. So, due to interest of time, I would not be discussing the AKS primary key testing algorithm but if you are interested you can see the original paper the paper title was "Primes is in P."

(Refer Slide Time: 08:58)



Now let us next define the greatest common divisor or GCD. So, imagine you are given 2 numbers a and b which are nonzero integers. And the GCD of a and b is the greatest integer which divides both a and b. So, we say integers a and b are relatively prime, we also use the term co-prime if their greatest common divisor is 1 that means so of course, 1 is a common trivial divisor of every a and b.

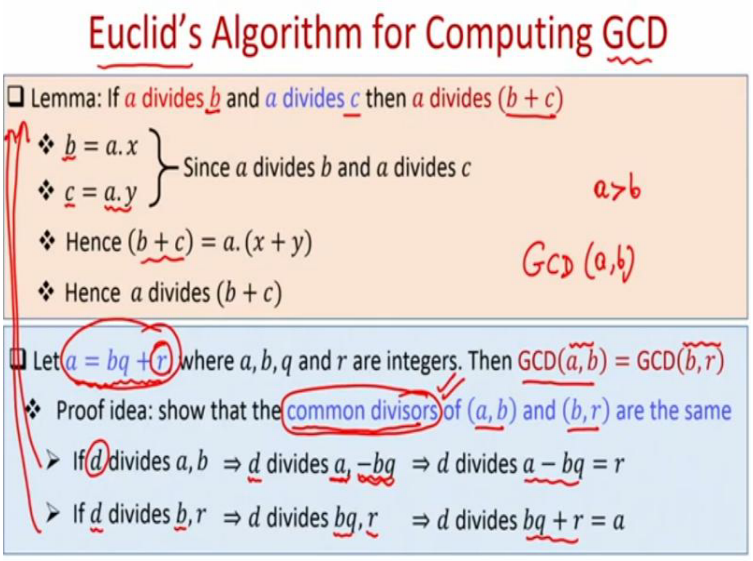
But if 1 happens to be the only common divisor, or if 1 happens to be the greatest common divisor of a and b, then we say that a and b are co-prime or relatively prime. Whereas if we have n values, a_1 to a_n, and they are pairwise, we call them as pairwise relatively prime. If you take any pair of values a_i and a_j in the set of n values which are given to you, they are co-prime to each other. Now an interesting question is if we are given 2 values a and b how exactly we find out the greatest common divisor?

One approach could be that you use the prime factorization of a and b, what do I mean by that. So, as per the fundamental theorem of arithmetic, a will have its unique prime factorization namely a can be expressed as product of prime powers. So, let the various powers of the primes

used in the representation of a are a_1, a_2, a_n and so on. And in the same way, the integer b will have its unique prime power factorization, then it is easy to see that the GCD of a, b will be this value : $(P_1^{\min(a_1,b_1)} P_2^{\min(a_2,b_2)} \ldots P_n^{\min(a_n,b_n)})$.

But to use this algorithm at the first place, you have to come up with a prime power factorization of a and b which in itself is a very computationally heavy task. So, we do not prefer to use this algorithm in general if your a and b are very large quantities.

(Refer Slide Time: 11:11)



Instead what we use is Euclid's GCD algorithm which is probably one of the oldest algorithms known. In fact, people believe that this is the first instance of an algorithm for any computational task, interesting computational tasks of course addition, subtraction, they are also computational task and you have algorithms for that. But this is probably a very interesting computation namely the computing GCD and Euclid gave a very simple algorithm, which we will be seeing soon.

But to understand the Euclid algorithm and why it is correct, let us first try to understand some properties of GCD. So, the first property here is that if your number a divides b, and if the same number a divides c, then a divides b + c. And it is very simple to prove. Since a / b, then I can write b as some a times x, where x is the quotient and the remainder is 0 and c is divisible by a I can say c is some y times a. Now what I can say about b + c, so I can say b + c is same as x + y times a and hence b + c is completely divisible by a.

Now, the crucial observation on which the Euclid's GCD algorithm is based upon is the following. Our goal is to find out GCD of a, b. And for simplicity, imagine a is greater than b. So, the idea that is used in the Euclid's GCD algorithm is that if a is some q times b + r, where r may be 0, if a is divisible by b, otherwise, r will be something in the range of 0 to b - 1. So, if a is b times q + r, then we can see that the GCD of a and b is same as the GCD of b and r.

So, you start with a and b, where a is greater than b, your goal is to find out the GCD of a and b. What the statement says is that it is equivalent to finding the GCD of b and r. So, let us prove this statement. So, the proof idea here will be the following. We want to prove that the greatest common divisor of a and b and b and r are same, we will instead prove that every divisor of a and b is a divisor of b and r; common divisor of b and r. And every common divisor of b and r is also a common divisor of a and b.

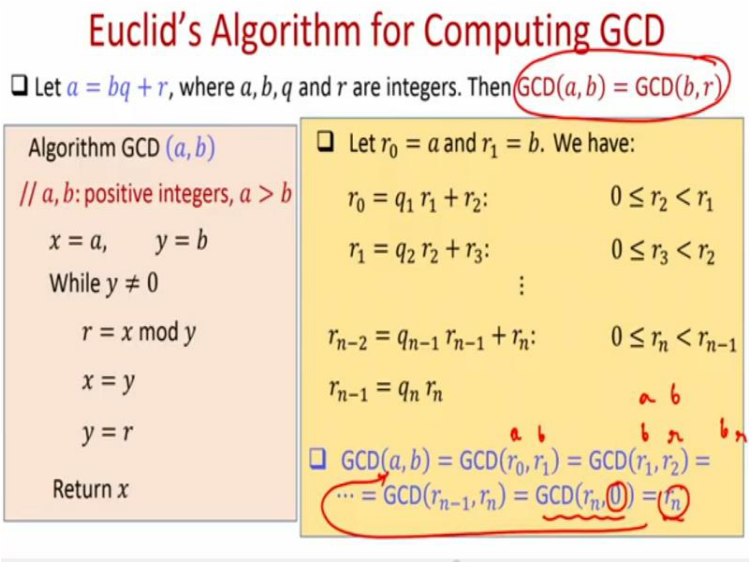
If you prove that, then that automatically shows that the greatest common divisor of a and b is same as the greatest common divisor of b and r. Because we are proving this, we are actually proving a stronger statement; we are proving the statement about all common divisors of a and b, whether it is greatest or not. So, consider an arbitrary divisor d which is a common divisor of a and b. So, since d divides b, it will divide -b times q as well. And any how d divides a say as per our assumption and now I can apply this Lemma.

So, you have a divisor which divides one number and another number. So, the same divisor will divide the addition of the 2 numbers and addition of the 2 numbers here will be a - bq and a - bq as per our definition is r. So, that means, if you have a common divisor of a and b then the same common divisor is also a common divisor of b and r. On the other hand, assume that you have a common divisor d of b and r we have to show that this common divisor d is also a common divisor of a and b.

So, again we use a similar idea here since d divides b it will divide any q times b as well and anyhow as per our assumption d divides r. So, again apply the same lemma I can say that d divides the addition of these 2 numbers and the addition of these 2 numbers is nothing but a, so,

that shows that every common divisor of a, b is also a common divisor of b, r and hence, the greatest common divisor of a, b will be the same as the greatest common divisor of b, r.

(Refer Slide Time: 16:07)



So, based on this observation, this is a very simple Euclid's algorithm, your input pair is a, b where a is greater than b and idea is that in each iteration, we will use this rule: to reduce the magnitude of our a and b till we reach a point where r becomes 0. So, what we do is we start with x equal to a and y is being b. So, x and y will be my placeholders and my value of x and y will keep on changing. So, the placeholder occupies the value a and b to begin with.

And i will iteratively keep on changing the value of x and y till I reach a stage where my y becomes 0. So, as I said that I will be using this lemma again and again. So, what I will do is I will find my r which is my current x modulo y and whatever is my current y that goes and becomes next x and whatever the r that I obtained that will go and become the next y and I keep on doing the process till my y becomes 0 when I reach a stage when y becomes 0, then I return my x and that will be my GCD.

So, you can imagine that what is happening here is the following: you start and compute a sequence of remainders and you stop when you obtain a remainder which is 0. So, I start with the given pair of values and treat them as the 0th remainder and the first remainder respectively. So,

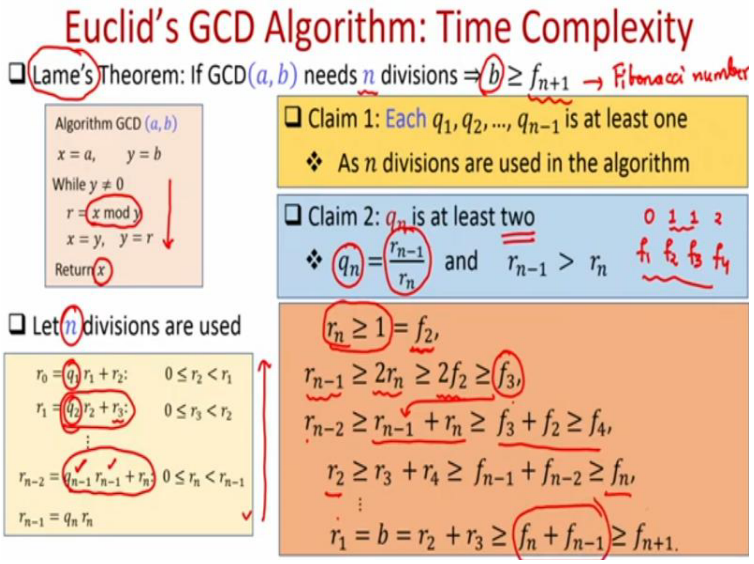
imagine that r_0 modulo r_1 is r_2, where r_2 is a value in the range 0 to r_1 - 1 and then in the next iteration, you will update your x and y.

So, this becomes x and this becomes y and you have obtained the next r and then in the next iteration your r_2 becomes your x and your r_3 becomes y and then you obtain the next reminder and you keep on doing this process till you obtain a 0 remainder, the time when you obtain the 0 remainder that means you have 0 here; you stop. And you will output r_n as the overall GCD. Now, what is the guarantee that this iteration will eventually terminate what is a guarantee that this is not going to loop forever.

The reason that it will eventually terminate is that in each iteration, you are definitely reducing the value of your y by at least 1 because you are now taking; you are updating the sequence of remainders. So, you start with the remainder which r_0 and the other remainder being r_1. And thus in each iteration, your remainders keep on getting decremented at least by 1. So, at most, it will require a number of iterations to eventually obtain a remainder of 0.

So that means eventually the algorithm will terminate. And by applying this theorem, I can say that the GCD of a and b is same as the GCD of the 0th remainder and the first remainder and that is same as the GCD of; so you can treat this as a and b this is your b and r and then in the next iteration your role of a and b and b and r gets changed and so on. And when you obtain GCD of r_n, 0 that means you obtain the other remainder being 0. So, the GCD of r_n and 0 will be of course, r_n and that is overall GCD because you can come back all the way and say that this r_n will be the GCD of a and b as well.

(Refer Slide Time: 20:25)



So, the now, next question is what is the running time of the Euclid GCD algorithm: is it polynomial in the number of bits that I need to represent my a and b or not? Because that will be our measure get will be our measurement of time complexity. And there is a very interesting result attributed to Lame which says the following and we will use this theorem to conclude that Euclid's running time is polynomial in the number of bits that you need to represent your a and b.

So, what Lame's theorem says is the following: if there are total n iterations used in your Euclid's GCD algorithm then the value of b is same as the n + 1 Fibonacci number. So, very beautiful result, which relates the number of divisions which are used in the GCD algorithm with the Fibonacci sequence and the proof is as follows: we will give a direct proof. So, since we are assuming that there are total n iterations or equivalently n divisions which are performed.

So, by the way if you are wondering where exactly division is involved, this is step your division is involved. So, basically we want to find out how many times this x modulo y operation will be performed in the Euclid's GCD algorithm, I do not want an algorithm where this x modulo y operation is performed exponential number of times. So, if n divisions are performed, so, what I have done is I have listed down the various remainders which I will keep on getting in each updated iteration.

And then in the nth iteration, I obtain 0 remainder here namely r_{n-1} is completely divisible by r_n. So, my claim here I will make now a series of claims and using that I will conclude Lame's theorem. So, my claim is that each of these quotients q_1, q_2, q_{n-1} is at least 1 and that is trivial fact because they are at least 1 because at the first place we have used n divisions to get the output of the Euclid's algorithm that means definitely r_0 is not completely divisible by r_1.

And we obtained some remainder because that only we have gone to the second iteration So, that means q_1 is at least 1, similarly r_1 is not completely divisible by r_2 there was some remainder that means, q_2 is at least 1 and so on. The second claim is that the last quotient here q_1 is at least 2 and this is because q_n is the ratio of n – 1th remainder and nth remainder. And I know that n – 1th remainder is strictly greater than the nth remainder, because in each iteration x is strictly greater than y.

So, since now if I my quotient has to be an integer value, so I have numerator greater than denominator. And I know that, this fraction, r_{n-1} over r_n has to be an integer value. So, that means the minimum value of q_n is at least 2. Now, based on these 2 facts, let us derive the proof for Lame's theorem. So, I can say that the nth remainder where I stopped the algorithm is greater than equal to 1, it is not 0, it is greater than equal to 1.

And if you see the Fibonacci sequence, the first term is 0. The next term namely f_2 is 1. The next term is the summation of the previous 2 terms, which is f_3. The next term is a summation of these 2 terms which would 2 and so on. So, I can say that r_n the last remainder is at least the second Fibonacci number. Now, what about the previous remainder. So, the previous remainder r_{n-1} is q_n times r_n.

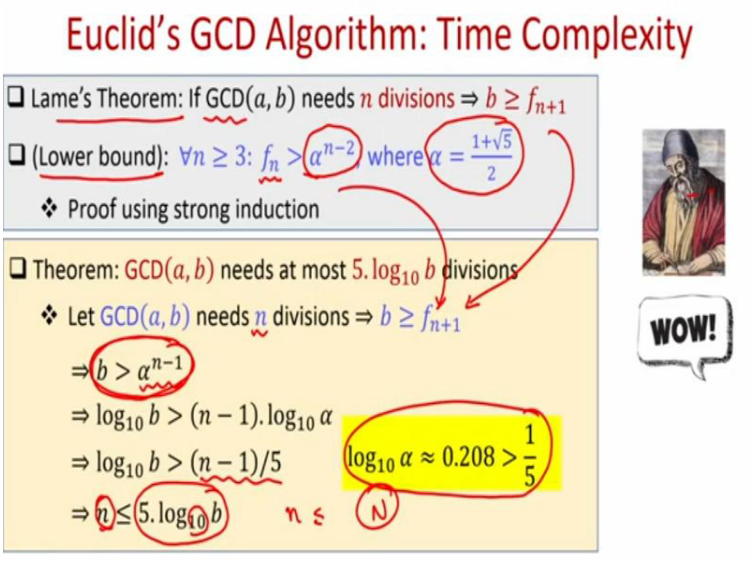
Now, I already know that r_n is greater than equal to 1, and I also know that q_n is at least 2. So, this is an exact equality, but I know that r_n is individually greater than equal to 1 (this value) and q_n is individually greater than equal to 2 and r_n definitive will be now, at least 2 times r_n , r_n is nothing but I have already proved that it is greater than equal to f_2. So, I obtained that the n – 1th remainder is as large as 2 times f_2.

And if you now compare it with the Fibonacci series here, this value is nothing but greater than equal to f_3. Now, I go to the previous remainder. So, remember, my goal is to say something about b, that is what Lame's theorem is. So, what I am saying is, I am going backward and trying to find out any equality relating all the remainders and then I will finally stopped with r_1 which is the value of b.

So, in the same way, I can say that the n - 2th remainder its exact value is this, I know that q_{n-1} is at least 1 and I already now have a relation or inequality involving r_{n-1} and hence, I can say that r_{n-2} is greater than equal to this. So, I can substitute q_{n-1} greater than equal to 1 and then I can substitute this inequality for r_{n-1} and hence I conclude that r_{n-2} is greater than equal to f_4. And now, if I keep on doing this process, I can come to the conclusion that r_2 is greater than equal to f_n.

And then finally, when I come to r_1 which is the same as b and r_1 is this value q times $r_2 + r_3$; q_2 is greater than equal to 1. So, I can say b is greater than equal to $r_2 + r_3$ but r_2 in the previous step I would have concluded is greater than equal to f_n and hence, I can conclude that b is greater than equal to $f_n + f_{n-1}$ which is same as the next Fibonacci number and that precisely is the claim. But, Lame's theorem directly does not help me to tell what exactly is the number of divisions,

(Refer Slide Time: 28:09)



Let us derive the exact number of divisions using Lame's theorem. So, I am stating Lame's theorem here. Now, let me recall a lower bound regarding the value of the nth Fibonacci number and relate it with the well-known Golden Ratio. So, it is a well-known lower bound that in the Fibonacci sequence if you focus on the nth term then it is greater than the golden ratio (α) raised to power n – 2 : (α^{n-2}) where the golden ratio, $\alpha = \frac{1+\sqrt{5}}{2}$ and this can be proved using strong induction I am not going to do that.

Now, based on this lower bound and Lame's theorem I can derive the number of divisions that are required in the Euclid's algorithm. So, the theorem, I can conclude that the Euclid's GCD algorithm will require at most these many numbers of divisions: $5*\log_{10}b$. So, I am deriving the answer in the base 10. But you can derive the answer to the base 2 as well because remember our time complexity is in terms of n, where n is equal to the number of bits that you need to represent your value a and b.

So, it will be basically$\log_2 b$. So, whatever argument I am giving here it can be modified easily to get the answer in terms of $\log_2 b$ as well. So, let us prove this theorem statement. So, imagine that there are n modular divisions or n mod operations performed inside your Euclidean algorithm then as per the Lame's theorem, we know that the magnitude of b or the value of b is at least as large n + 1 Fibonacci number.

And now if I apply the lower bound on f_{n+1}, I can relate it with the golden ratio, so I get a conclusion that b is greater than α^{n-1} and the value of $\alpha = \frac{1+\sqrt{5}}{2}$. So, if I take log on both sides of these 2 equations to the base 10, you can take the log to the base 2 as well and that will give you a different result, but more or less it will be same. So, if I take log to the base 10, I get this equation here there this relation.

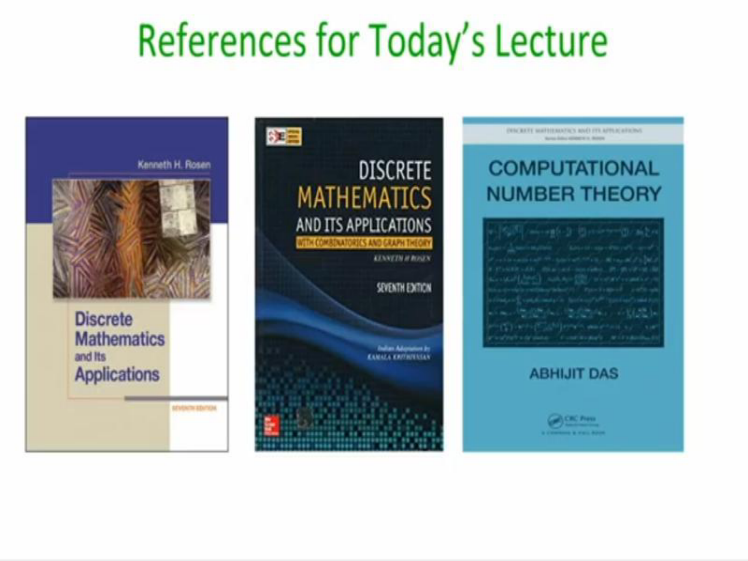
And now I can use the fact that the log of Golden Ratio $\log_{10} \alpha$ is approximately 0.208, which is greater than 1 / 5. So that is why I can write it as 1 / 5 here and that shows that n is less than equal to $5 \log_{10} b$. So, sorry for the confusion here, 'n' I used here for the number of divisions

and I was continuously under the impression that n is also the number of bits used to represent the values a and b.

So, let us use a different representation. So, let N denote the number of bits used to represent a and b. So, what I have derived here is the following: I have derived it, the number of divisions that you perform is upper bounded by roughly the number of bits that you need to represent your value b. So, we are getting it in the base 10. So, $\log_{10} b$ is not n, and will be something else. But whatever derivation we have done here. The step where we took the log to the base 10, if you do it with log to the base 2, that will give us the value of n in terms of the value of n somewhere so it would not be more than the number of bits asymptotically it would not be more than the number of bits that you need to represent your integers a and b. So that means Euclid's algorithm is actually a polynomial time algorithm polynomial in the number of bits that you need to represent your integers a and b.

That is a very remarkable fact because at the time when Euclid invented his algorithm, there was no notion of time complexity, there was no notion of polynomial time complexity, exponential time complexity and so on. So, without even worrying about how efficient it will be, Euclid gave us a very fantastic algorithm to compute GCD of 2 numbers whose running time is polynomial, and which can be used in practice.

(Refer Slide Time: 33:14)



So that brings me to the end of today's lecture, these are the references used. Just to conclude in this lecture, we introduced prime numbers, we discussed a naive algorithm for checking whether a given input number is prime or not, and its running time is exponential. And we also discussed about Euclid's algorithm for computing the GCD whose running time is polynomial in the number of bits that you need to represent your input values. Thank you.