Computational Number Theory and Algebra

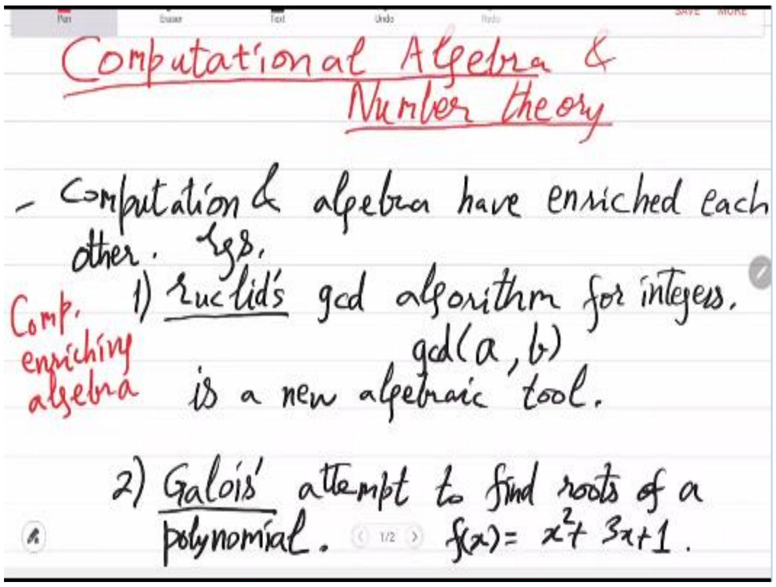
Prof. Nitin Saxena
Department of Computer Science and Engineering
Indian Institute of Technology-Kanpur

Lecture - 01
Introduction: Computation and Algebra

Okay, so we will study Computational Algebra and Number Theory. So basically, the background that you need is you should know some a bit of algebra and, but we will not work in pure algebra. So we will work with the algebraic problems and we want to solve them only algorithmically. Okay, so questions will be in algebra but ultimately we want algorithm, so we lead time complexity analysis and maybe space analysis as well.

But mainly time complexity. So we want to optimize the time to given a problem. It will be a finite input and we want to minimize the number of steps that your algorithm takes to solve it. So let us start with some broad motivation before moving to specific problems.

(Refer Slide Time: 01:26)



Computational Algebra & Number theory

- Computation & algebra have enriched each other. Egs.

Comp. enriching algebra

1) Euclid's gcd algorithm for integers. gcd(a, b) is a new algebraic tool.

2) Galois' attempt to find roots of a polynomial. $f(x) = x^2 + 3x + 1$.

So historically computation and algebra have enriched each other. So what I mean by this is that there are computational problems which are solved by algebraic means, methods. And while you solve these also new algebra gets created, okay. So both algebra is used is applied in computation, but also computation enriches algebra, pure algebraic and pure number theoretic methods.

So some very classical examples of this phenomena are first most stunning example is Euclid’s GCD algorithm. So this is the famous algorithm which given two integers finds the GCD which is the biggest number that divides both. Or analogously given two polynomials finds a polynomial which is highest degree dividing both, okay. So this is a very when it was solved it was not a practical problem, it was just solved out of curiosity.

But then later on it became a very useful problem. So Euclid’s GCD algorithm in particular for integers if you look at, so given two integers a and b and you want to find the GCD, the way it is defined that you want to find a maximum number c that divides both a and b, it seems that to find c you have to factorize a and you have to factorize, well you have to have factorize either a or b.

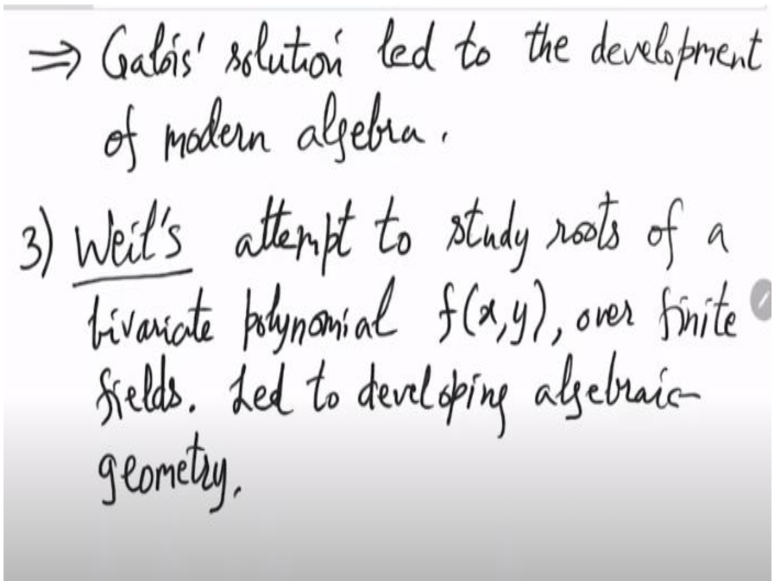
But that is a very hard problem, okay. So that is a problem, which is even now there is no fast algorithm. But thankfully Euclid’s GCD algorithm completely bypasses the factorization issue, okay. It solves it computes GCD in a very different way, and a very fast algorithm. So this actually becomes a new algebraic tool, Euclid’s GCD process. And then you can use this new algebraic tool to prove new algebraic theorems, okay.

So these application these things we will see in detail. For now just remember these keywords. Second is something more abstract. So that is Galois’ attempt to find roots of a polynomial, okay. So given a polynomial say univariate over so it has coefficients in integers. So suppose you are given a polynomial f x x square + 3x + 1 okay. So the presentation is simple but what are the roots of this?

So that is not completely trivial, at least it was not really trivial 1000 years ago. So formulas were found to actually compute the roots of a quadratic univariate and then you can ask the same question what if this is a cubic or what if this is a biquadratic or what if this is a quantic, right. So degrees 2, 3, 4, 5 and so on. So in that case can analogous formulas be found?

So Galois showed that and also people before him showed that up to 4 there is a systematic way, up to degree 4 you can find formulas, but beyond there it is impossible to find a formula, okay. So that impossibility result is a very famous result in algebra.

(Refer Slide Time: 05:58)



⇒ Galois' solution led to the development of modern algebra.

3) Weil's attempt to study roots of a bivariate polynomial $f(x,y)$, over finite fields. Led to developing algebraic geometry.

So this Galois' solution led to the development of really new algebra. In fact, modern algebra started with this. Third example is even more abstract. Or it leads to even more abstract theory in math, which is Weil's attempt to study roots of a polynomial, now bivariate. And in general, it can be multivariate, but let us start with bivariate. So a polynomial f with two variables x and y, okay and over finite fields.

So is there anyone here who does not know what is a finite field or what is a field? Raise your hand. So if you do not know what is a field then maybe this course is not for you. If you do not know what is a finite field, in assignment one there will be some exercise problems to warm up the concept of finite fields. So finite fields are a very important tool in computer science in many areas and even in many things in practice.

Both in Computer Science and Electrical Engineering finite fields are applied. So on the abstract side Weil studied around 100 years ago, given a bivariate polynomial over a finite field, what can you say about the structure of the roots? So roots over a finite field will be finitely many, which is not true over rationals or infinite fields. So there the roots can be infinitely many.

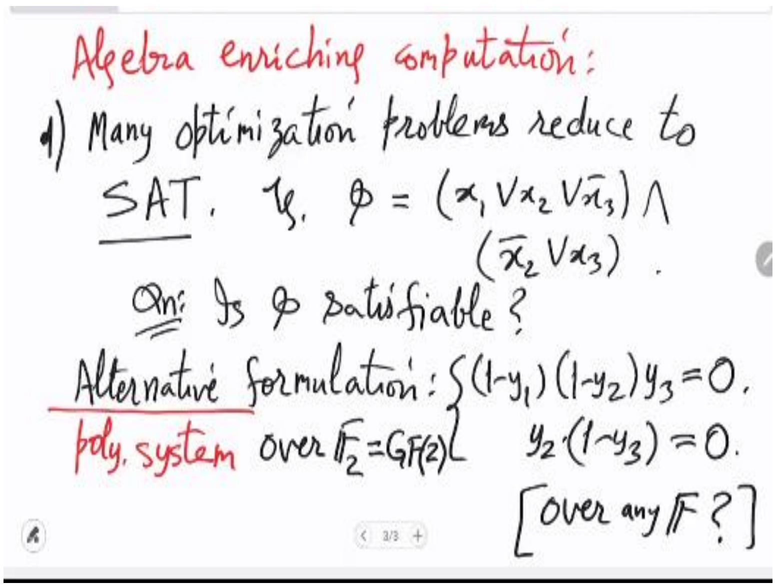
Because there are two variables. So one variable can take infinitely many values and possibly give a single y. So that already gives you infinitely many roots or the possibility of having infinitely many roots over finite fields; y has finitely many choices and so does x. So there are only finitely many roots. But then is there a structure on them? So this is what Weil studied.

And that led to the development of algebraic geometry in modern form, which we will not do here. This is again, just a broad motivation. So yeah, so these are the three examples. So Euclid's algorithm, Galois' you can say formula, or the impossibility of formula to find roots of a univariate. And third is what can you say about the root structure of multivariates over finite fields say?

So these are the examples of, you can think of them as actually, original questions here are all of in modern terms, it is all about algorithms. So you want to actually compute the GCD, you want to compute a root, and you want to compute root of a multivariate, right. So these are really computational problems. So you can think of these as computation enriching algebra.

And on the other side which probably is more interesting for computer scientists is how does algebra enrich computation? So in which places is algebra applied in practice, even in practice. So let us do that, see some examples.

(Refer Slide Time: 10:32)



Algebra enriching computation:

1) Many optimization problems reduce to SAT. Eg. $\phi = (x_1 \vee x_2 \vee \bar{x}_3) \wedge (\bar{x}_2 \vee x_3)$.

Qn: Is ϕ satisfiable?

Alternative formulation: poly. system over $\mathbb{F}_2 = GF(2)$ $\begin{cases} (1-y_1)(1-y_2)y_3 = 0. \\ y_2\cdot(1-y_3) = 0. \end{cases}$

[over any $\mathbb{F}$?]

So the first is a motivating example, which everybody here should know. That is many optimization problems reduced to satisfiability, right. So SAT. So SAT stands for satisfiability which is the problem of so SAT is what? Does everybody know the problem of SAT? You are given a formula like x 1 or x 2 or x 3 bar, right. So this is this you can call as a clause. It is a disjunction of three literals.

And x 3 bar is the NOT gate applied on x 3 right. So one clause and another clause. So you then do conjunction which is and so this is OR and NOT. And second clauses may be x 2 bar or x 3. So you are given a Boolean formula. We call it a Boolean formula, because it is a formula and the literals can take only 0, 1 or true false, false true values. So ultimately the value of phi if you fix the literals to false or true will be false or true.

So the question of SAT is whether phi can be made true, right. So which means that if you do not have any special experience about phi, then what you will do is just try out all possibilities. So there are eight possibilities for x 1, x 2, x 3 fixings and you will try all and then see whether it is true. It is equivalent to making all or each of these clauses true. So in small examples, it does not seem to be a hard problem.

But when the number of variables grow, and the clauses also grow in number then it is very difficult to keep track of correlations. So a variable appears in many clauses with or without NOT gates. And you do not know whether that variable will be biased towards true or towards false, okay. So then it essentially becomes looking at the whole space, which is 2 raised to n for n variables.

So that is the computational question. Is phi satisfiable? Right. So that is a kind of an abstraction of our computational question, but it is very real, because many optimization problems actually can be phrased in this form, right. That you must have seen so and the problems to which you can reduce SAT 2 are called NP hard problems. Right, so many problems, hundreds and thousands of problems reduced to SAT.

And SAT also reduces to some of them, many of them. So those problems are called hard problems. So there is a collection of equivalent problems. All of them are unsolved, at least in practice. And they are believed to be equally hard and believed to take exponential time. So what is that to do with algebra? Well, this Boolean question is already to do with Boolean algebra, but you can also come down to fields.

So you can make it more palatable in classical algebra. So you can, for example, look at an alternative formulation in terms of polynomials. So what you can do is convert a clause into a polynomial equation. So how will you convert x 1 or x 2 or x 3 bar into a polynomial equation? So remember that this will be 1 if and only if either x 1 or x 2 is 1, or x 3 is 0. Right yeah, so do you want x 1 times x 2 or do you want 1 – x 1 times.

So we use, to avoid confusion we use different variables. So you can write it as 1 – y 1, 1 – y 2 and y 3, okay. So this equation for example will force one of these three factors to vanish. So then, automatically either y 1 is 1 or y 2 is 1 or y 3 is 0, right. So that corresponds well with x 1, x 2, x 3's behavior in the first clause. That constraint, first constraint. And the second one analogously you can now write as y 2 times 1 – y 3 = 0.

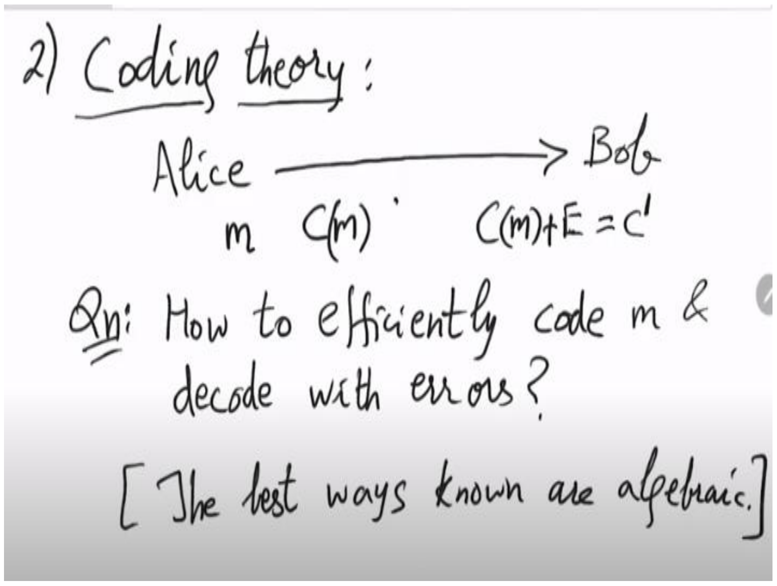
So that is the system. This is a system of two polynomials three variables and what is the field? Yeah, so this is over a finite field over F 2. So we will write it in this way or we can generally in mat-x it is also written like GF 2. GF is Galois field, but in computer science usually we just say F sub 2 subscript 2. So this is a polynomial system. Okay, that is the alternate formulation of SAT. Right?

So do you really need the field F 2 or can you do this expression, this translation in other fields? Can you do this over the field of reals or over the field of complex. You can actually because you just want I mean, this already is a is of that type, right? Because the first equation forces, one of these to be 0 or 1. So I mean, this is true for F 2, but also over any field. It is a translation.

So the field is not really important in this translation. The first equation forces one of these. So what you have learnt is bottom line is that SAT is a canonically hard problem in computer science and it is really a special case of polynomial systems over fields okay. So even checking whether a polynomial system over some field has a root just the existence of a root is something at least as hard as solving SAT, okay.

So all the optimization problems, such problems can be seen as a polynomial system. So that may help in understanding some computational problems because you have now an algebraic formulation. Optimization problems have a different life. They are mainly combinatorial. But, but they but this actually gives you also gives you an algebraic point of view. Any questions at this point? Okay, second is so let us call this first.

(Refer Slide Time: 19:27)



2) Coding theory:

$$\text{Alice} \longrightarrow \text{Bob}$$

$m \quad C(m) \quad C(m)+E = c'$

Qn: How to efficiently code m & decode with errors?

[The best ways known are algebraic.]

Second example is even more substantial because it is actually used everywhere in information transmission, which is coding theory. So what is coding theory? Yes. So whenever you transfer whenever you want to play with bits in practice, so you will have to physically realize it somehow. So in hard disk you physically realize the bits and over a wire, internet wire or whatever Ethernet wire you realize bits.

Or when you use a phone then you realize bits in terms of waves in the air. So all such physical realizations are error prone, like 20%, 30% of the translation is lost. So you are sending 0 physically but when it reaches the destination it will become 1, okay or it will become unreadable. So even though physical realization is so poor how do you manage to store information and work with it precisely, right?

That is the miracle. So that miracle is possible because of coding theory. Because of something called error correcting codes and believe it or not these error correcting codes are mostly algebraic number theoretic, okay. So the best ones are all using algebra with finite fields. So we will see some details later in the course. Question, you can model it as, so say Alice and Bob are two players.

And Alice wants to send a message to Bob over this channel okay. So Alice will send will wants to has a message m. So she will code it and the code will be sent in the beginning of the channel. But at the end of the channel, it will become something else, right because there will be errors. So some error will be added. So this is what Bob will get C prime. And from C prime now, Bob has to deduce m, right.

So there is a there is encoding, there is decoding and there is decoding with errors, right. So these are the algorithms that you have to come up with and you have to come up with guarantees. So suppose the, the error changes 50% of the bits then is it possible for Bob to deduce m from C prime? Why not? Exactly, so just think of one bit. I mean if Alice sent a single or Alice sent two bits and one of them got flipped.

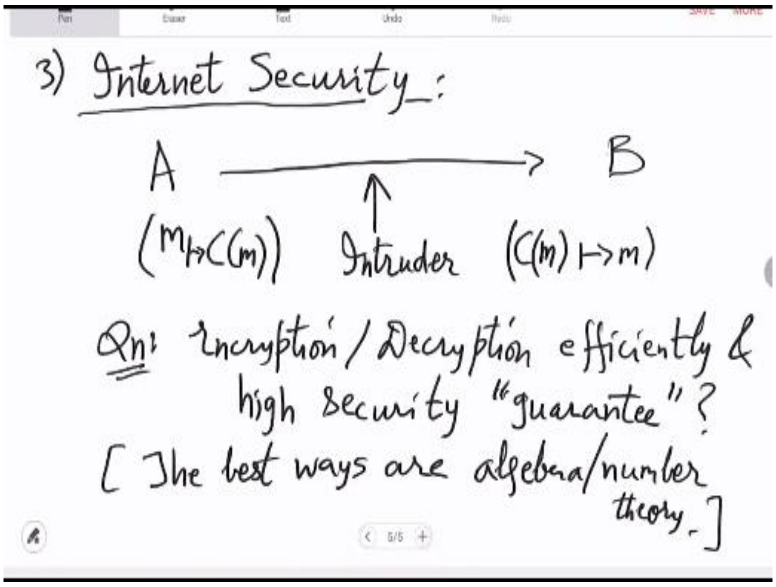
So C m basically was two bits and one of them got flipped, that is C prime. So Bob now has no clue what Alice wanted to send, right. So this cannot be this the error correcting code cannot be that good, there is a limit. So it cannot really handle 50%. But at the end of the course, you would have seen a method by which anything close to 50% can be handled, okay.

So it so error correcting codes can handle something extremely close. So between 49%, 50% even, but with some caveats. So at this point actually it should be it should seem impossible that such things exist. Okay, so and obviously you anything you want to do in algorithms, you want to do it efficiently. So if the message was 10 bits long you cannot you do not want to spend a lot of time in encoding.

And you do not want to make it very large right because there is always a limitation of time and space. So there has to be guarantees on how much time how much space and how much error. So how much is the tolerance of the error correcting code. So all those parameters are very good. So this is how information is stored in your computer and this is how it is communicated between computers, okay.

So error correcting codes are used. That is why even computers exist. So right, so the question here is how to efficiently code? So efficiency here is very important. Otherwise you can come up with trivial codes. But you want to do it efficiently. So how to efficiently code m and decode with errors? So the best ways are algebraic okay. So this is a major application of algebraic methods and we will come to this once we have done some basic computational algebra. Any questions?

(Refer Slide Time: 25:59)



3) Internet Security:

$$\begin{tikzcd} A \arrow[r] & B \end{tikzcd}$$

$(m \mapsto C(m))$ Intruder $(C(m) \mapsto m)$

Qn: Encryption / Decryption efficiently & high security "guarantee"?

[The best ways are algebra/number theory.]

So another equally important if not more important application is internet security or security in general. So again Alice wants to send a message to Bob through an insecure channel. But here, Alice is not really, so Alice has already solved the problem of error correction. But she is more worried about the message being hacked. Okay, so the message should be invisible to anybody who is intruding.

So an intruder is reading the channel. And the message should be sent in a way such that intruder is unable to deduce from C m. So now we will call it encrypted maybe. What do you want to call it? Let us say C m. So this is a cipher text that Alice will send and this will reach we can assume now that it will reach correctly here to Bob and intruder will also read C m.

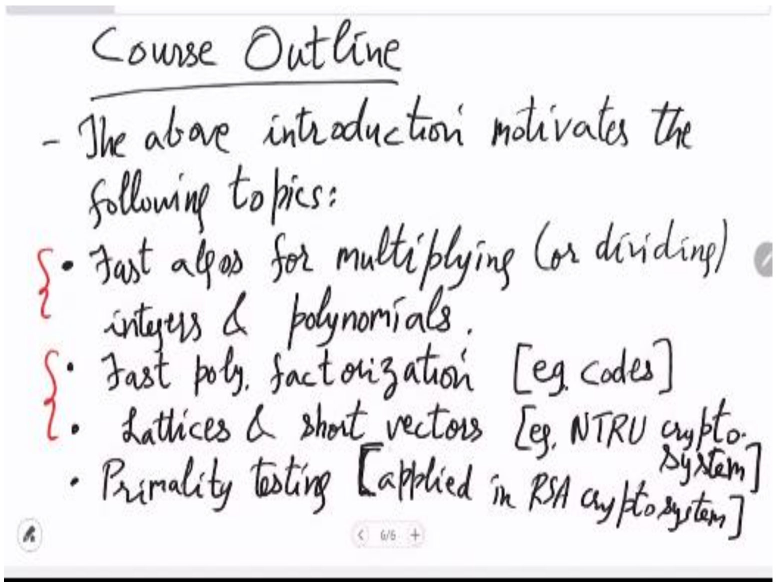
But from C m intruder should not be able to deduce m while Bob obviously should be able to deduce, okay. So you want a way to encrypt and you want a way to decrypt and you at the same time intruder should not be able to decrypt. Well, so there is a whole field and trillions of dollars are spent on this because all your banking transactions, shopping everything even the simplest things are using HTTPS at least.

So all that is based on certain protocols which again used algebraic number theoretic methods, okay. So much of the basic mechanisms we will study in this course. Okay so what goes in encryption decryption and why is it called secure? Now there are actually no known proofs of security. So the security is just on assumptions or conjectures. There are no mathematical proofs.

But we will just abstract out the mathematical problem that we believe to be hard, okay. So one of them is integer factoring. But there are also other problems that are considered hard on which these systems are based. So the question here is, again encryption decryption efficiently and high security guarantee. Those are the things you want. So again, here also the best ways are in fact, not just the best ways there are no other ways.

Every protocol uses some bit of algebra or number theory, okay. Any questions? So let me give a quick course outline.

(Refer Slide Time: 29:52)



Course Outline

- The above introduction motivates the following topics:
  - Fast algos for multiplying (or dividing) integers & polynomials.
  - Fast poly. factorization [eg. codes]
  - Lattices & short vectors [eg. NTRU crypto-system]
  - Primality testing [applied in RSA cryptosystem]

So the above introduction that we just saw motivates the following topics. Actually, it is fair to say that this course is only about two things. So one is multiplication and the other is factorization, okay. So given two numbers multiplying them, so algorithms about that and given a number factorizing it. So this course everything that we will do ultimately will boil down to just multiplication or factorization, okay.

So we will do it for integers and we will do it for polynomials. In polynomials, again there are two major areas of study. So one is we will start with univariate polynomials. So from integers we will go to univariate polynomials and from univariate polynomials we will go to multivariate polynomials okay. So as we do these transitions we have to develop a lot of machinery.

So that is how the topics are arranged. So fast algorithms for multiplying or dividing those things are actually essentially the same, you will see that. Integers and polynomials. These methods, these algorithms will be very algebraic. So strangely this we will first do multiplication of polynomials in a fast way and then we will use it to multiply integers, okay. It will be the other way around.

So multiplication of polynomials will be for some reason better to present first and then that will be used to multiply integers also efficiently. So at this point just to warm up your time complexity. So if you are given two n-bit integers you want to multiply them. So what will be the, how many bits will the product have at most? Sorry, n + 1 it will become if you just double it. But now you are multiplying n-bit times n-bit right?

So it can become now 2n bits. It is basically a squaring process. So number of bits will double. And yeah, so the input is 2n bits overall and output is also 2n bits. Now what is the algorithm and the time complexity that you have seen? You clearly know how to multiply numbers. So what is the time complexity of that in terms of this n? Sorry, n square? Yeah, so it is order n square.

If you just do it in the high school way, then this will be order n square. And when I say fast, we want to beat that, okay. So we actually want to get as close as possible to order n. So how will you get from order n square to something just above order n, right? So that will need a lot of work. So we will do that. Some of it is actually covered I think in other courses.

But generalizing all those things properly and then making it work for integers is much harder. So that you would not have seen in other courses. And then the opposite of this which is factorization. So fast polynomial factorization. Yeah, and this polynomial factorization has application in coding, okay. So the thing which I mentioned before. So it is used in coding.

So we will discuss that application once we have done this polynomial multiplication and polynomial factorization. Factorization is slightly more work than finding roots. Say you are given a univariate polynomial. So finding roots is you want to find an alpha such that f at alpha vanishes. factorization is slightly more because your polynomial may not have any roots like a quadratic irreducible times another quadratic irreducible.

That is given to you. From that you want to deduce the quadratic factors, right. So you do not want to go all the way to roots because roots become complex, for example. So you want to stop at a quadratic factor. So that we will do. This will have two parts because the algorithms are really based on what field you want to find factors in or over. So if you want to do this over rationals there is a completely different method from if you want to do it over finite fields.

In fact, we will first start with finite fields. Then we will use that machinery and do something else to find factors over rationals or integers. So that machinery will is called lattices, lattice theory and short vectors there. And this has so although it seems to be so exotic, that you are using lattices to find factors of integral polynomials, right which seems to be a useless exercise.

But believe it or not this has a lot of applications and in particular we will see this lattice based cryptosystem which is called NTRU. So in fact this finding short vectors is a very important problem in computer science. It has many applications. So we will only focus on how is the encryption and what is the decryption in this NTRU cryptosystem. But NTRU itself has now hundreds of variants.

So we will not be doing those things in that great detail. But we will just do the basic introduction to lattice cryptosystem and these algorithms are actually also used to do the reverse which is to break cryptosystems, okay. So if you can find short vectors then you can also break some cryptosystems. Those things I guess also we will not be able to cover.

But this is an important topic. Right. So what have we, by this point what would you have covered? So this is about multiplication. This is about polynomials, the opposite thing which is factorization. And then you do this for integers, integer factorization, right. So if you are given an integer, first you have to find out whether there exists a factor at all, right?

So which is the question of testing whether the number is prime. So that is primality testing. So again, maybe just as a warm up, given an n-bit number how do you test whether it is prime? What was the method you learnt in school? What is the time complexity of that? Sorry. Yeah, so that is a common misconception. Yeah, so your so I wanted to point out this misconception.

So if you have an n-bit number, which say you have written in a page. So it has some 1000 digits, n is equal to 1000. So that does not mean that the number is around 1000, the number is actually around 2 raised to 1000. So it is a huge number right. So for n bits the number is actually around 2 raised to n. And what you have learnt in school has time complexity square root of 2 raised to n which is 2 raised to 0.5 n.

So when n is 1000 or 2000 you are talking about 2 raised to 500, 2 raised to 1000 steps, right. And on the fastest computer in the world or all the computers of the world combined, how much time would it take to do 2 raised to 500 steps? It will take forever. It is literally infinity because it is far more than the time that universe has spent till now, okay. So it is literally infinity.

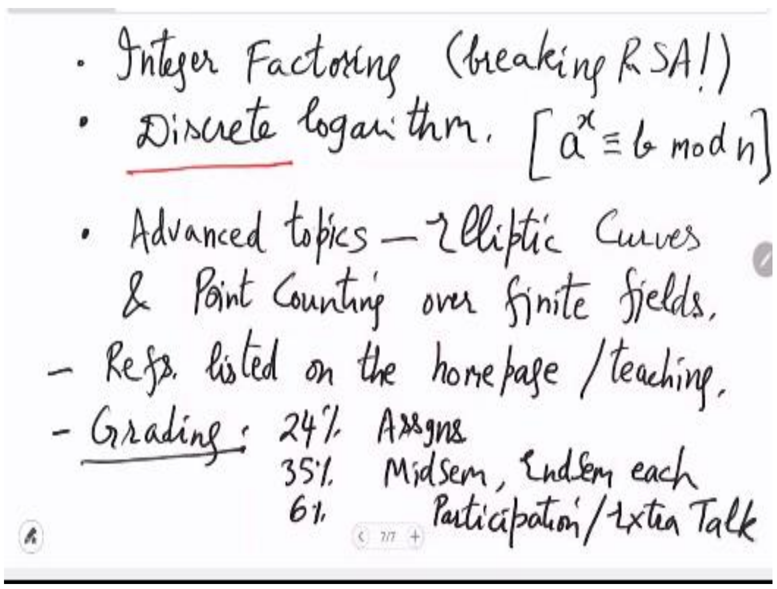
So even with all the computers combined, you cannot use high school method to check primality, forget factorization, right. So that is you are up against. So in this course you will see methods which make which bring it from 2 raised to n down to let us say n square or n cube, okay which will be the industrial grade primality testing algorithm.

So you can use 4000, 5000 bit numbers and within a second you get an answer, right. So and why is that important? Well, it is first of all interesting in its own right. But it is used in all these cryptosystems that are applied. So in particular applied in RSA. So whenever you click on a on an HTTPS link or you do SSH, you should always thank RSA cryptosystem because this is what the S in the HTTPS means, okay.

So this is all to do with RSA cryptosystem. So this system uses, to design it you need primality testing and then guarantee that you cannot break it is that comes from integer factorization, the fact that there is no good way to factorize till date except this high school method which will take infinite time. Although in this course, you will see some factorization methods but they are not very impressive.

So in the end, we will not have much time, but we will do some integer factoring for whatever it is worth.

(Refer Slide Time: 41:57)



- Integer Factoring (breaking RSA!)
- Discrete logarithm. $[a^x \equiv b \bmod n]$
- Advanced topics — Elliptic Curves & Point Counting over finite fields.

– Refs. listed on the homepage /teaching.

– Grading: 24% Assgns
35% MidSem, EndSem each
6% Participation/Extra Talk

So that basically if you can make it optimal or if you can make it practical then you can break RSA. Okay, but the state of the art is far from that. And something related to RSA is also discrete logarithm. So discrete logarithm is basically you are given just like logarithm. So you are given a and b and you want to find x, okay.

So a and b is given and you want to find the exponent x such that a raised to x is equal to b, which again, by high school method, it is a trivial it is an easy thing to do, because you can compute log of b and log of a and take the ratio. But here the all of that collapses because you are doing it modulo some number. Okay, so when you do it, do this mod n, then that high school method does not work. It does not, it is illegal.

So then it is which is why this word discrete, okay. So this discrete problem the only elementary method you can do is try out all x's. And how many x's are there? How many possibilities of x are there? Yeah, it is around n; n is again, n is huge; n is like 1000, 2000, 4000 bits, right? So you the space is too big to search.

So this is also a hard problem. Both integer factoring and discrete logarithm are hard, which is why RSA has survived to this day, RSA security. But there are some algorithms which are non-trivial. I think this much will already take up our time given in the semester. But if somebody wants to cover some advanced topics, then I will put ideas on the webpage.

So some nice advanced topics are to do with elliptic curves and point counting over finite fields. So some of the practical cryptosystems are actually based on elliptic curves and point counting or even hyper elliptic curves. In some respects they are better in some respects they are equal to RSA. But this machinery is far more advanced. So this needs a separate course in itself.

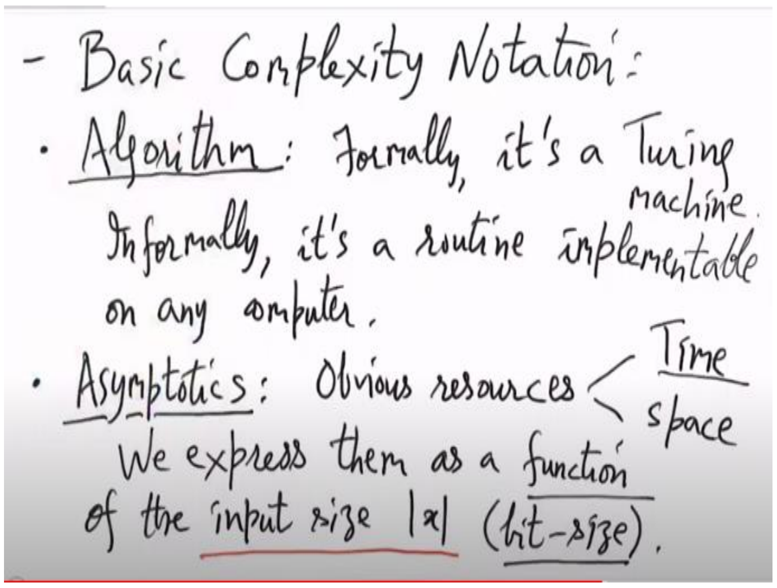
But if somebody has interest in this machinery, then they can give short talks. So I will put some references. Well any questions at this point about the motivation or course outline. So that gives you an idea of what you expect from the course. And if it seems too tough, then you have a few weeks to drop. Okay, do not take much stress. So references are many.

Each of these topics actually there are many, many lecture notes, surveys, papers that you can find, if you just simply search in Google. There are also Wikipedia pages on each of these topics. Giant literature is there. So you can, so I am not pointing to anything specific. You can read from wherever you want. Some references I will put on the homepage. So which you can just look at my homepage under teaching.

So you will, in fact you can also look at the lecture notes from the earlier offerings that you can see. Some words about grading. So we will have four assignments. So let us say that is 24%. And say you have equal midsem and endsems. And the remaining percentage is say your participation in the class or if you present an advanced topic as a independent lecture. So extra talk.

This is not compulsory, giving extra talk because there are so many students, we cannot have so many talk. Only if I find it interesting or and we have time then I can allow some of you to give extra talks. Any questions? Okay, so then in the remaining time we can start mentioning some of the basic notation that we will be using and you can brush up the details.

(Refer Slide Time: 48:32)



- Basic Complexity Notation:
  - Algorithm: Formally, it's a Turing machine.
    Informally, it's a routine implementable on any computer.
  - Asymptotics: Obvious resources < Time, space
    We express them as a function of the input size $|x|$ (bit-size).

So we will be using some basic complexity notation, which so all this is trivial for CSC students, but may not be trivial for non CSC students. So what is meant by the word algorithm? What does a non CSC student think about algorithms? Is there a definition of algorithm? So for all practical purposes algorithm is a C program, right. But if somebody is allergic to C then you want a definition independent of programming.

So a definition independent of programming is you define a machine and then say that whatever the machine does is an algorithm, okay. So that machine is called Turing machine. So formally algorithm is just a Turing machine. So every algorithm is a Turing machine and every Turing machine is an algorithm. It is one and the same thing. Nothing else is called an algorithm, okay.

So it is a mathematical object, it is a mathematical concept. It is not something vague. We are not leaving it vague. So since it is exactly a Turing machine, so given a problem okay, so maybe before that informally what is it? Informally, it is a routine that can be implemented on any computer. Implementable on any computer. So computers are believed to be or they are in a way they are Turing complete.

And the programming languages are also believed to be they are again Turing complete. So any algorithm any Turing machine can be implemented in any programming language on any computer okay. So informally, you can just think of that. What is the difference between a Turing machine and a computer? So computer is not a Turing machine. So why is that? There is some minor difference.

No, no. Yeah so it is only about finite and infinite. So the Turing machine technically it has an infinite tape which is infinite memory. In Turing machines, in Turing machines it is infinite memory and in computers obviously it is finite, it is limited by the primary and secondary memory. So except that there is no difference.

So in Turing machine we have to give infinite memory because so why do we put infinite memory in a Turing machine? Did you ever think about that? Yeah, so that is only because of the asymptotics. So we want to say that the Turing machine on this problem or on this input of n bits will take time let us say order n square.

But to make this mathematical statement, this statement has to be true for all n, which means that the computational power of the Turing machine should be infinite. At least the memory should be infinite. You cannot limit at any point the storage of the Turing machine. So just for that reason for the theorem statement to make sense, we give it infinite tape. So that brings us to asymptotics.

When you talk want to talk about complexity of time or space and other resources also. So the obvious resources for us are time and space. Time is more important for us because once we say that the time is limited, obviously, the space is limited by that right. Because just to go through some space, you spend time. So once we say the time is small automatically space is small.

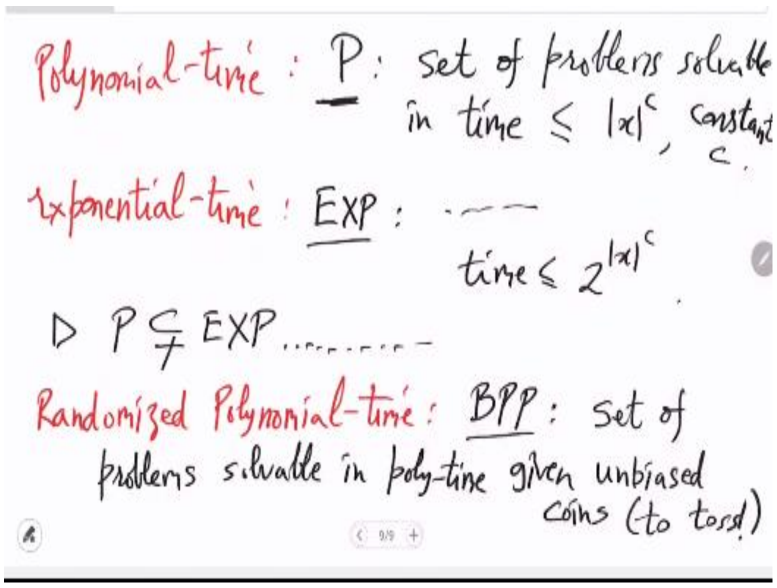
So I think in this course, we will never make any statement, hardly ever will make a statement about space. And then what is the definition of time? When we say that order n square is the time what does that mean? Of? Of what? No basic steps what? Yeah, but what does that mean? So there must be some machine on which the basic steps are happening. So that is the Turing machine.

So once you have this machine you have a well-defined steps. That I would not go into what are the steps, I think you just brush it up. Everybody I think has done TUC, students like CSC. So there are these steps in Turing machine and the just the number of steps is called time. And when you look at the number of steps as a function of n, which is the input size that becomes time complexity.

Okay that is the, that is the deal. So we express them so the resources spent as a function of the input size, as a function of the input size x. So this notation means number of bits okay. So this double bar covering x means how many bits are there in the input x. So that we usually call it n. And the number of steps of the Turing machine as a function of n is becomes the time complexity.

And similarly space complexity and other resources. So those resources we study in a different course. Here, we will not worry about other resources too much or at all. And once you have this asymptotics you can define complexity classes, which actually in this course, we will only worry about two complexity classes. One is polynomial time.

(Refer Slide Time: 56:59)



Polynomial-time : $\underline{P}$: set of problems solvable in time $\leq |x|^c$, constant c.

Exponential-time : $\underline{EXP}$: ------ time $\leq 2^{|x|^c}$.

▷ $P \subsetneq EXP$ ....

Randomized Polynomial-time : $\underline{BPP}$: set of problems solvable in poly-time given unbiased coins (to toss!)

And the other is exponential time. So polynomial time is the complexity class P, okay. So these are the problems set of problems solvable in time at most size of x to the c where c is an absolute constant. So c is a constant. So problems which are solvable in at most time n or n square or n cube and the set union is called P. Okay, this is the definition of P. informally, we will we call it polynomial time.

So we say that a problem is solvable in polynomial time if there is an algorithm of this time complexity. Exponential we give more time. So this is EXP class. So these are the same thing and time has to be at most 2 raised to x raised to c. This n raised to c goes in the exponent. So this is far more time available to you. So hence, x has far more problems than P has, okay.

And it is actually there is a proof that we cover in computational complexity course that P is strictly smaller than x, okay. So the problems that we come across in life are usually in x. And amongst them some of them are in P, some of them are not known to be in P and some of them are known to be in x - P. Okay, so this is the usual categorization of problems that you come across.

Are there problems which are outside EXP? Yeah, so should not come as a surprise that there are very hard problems. So there are many complexity classes, infinitely many above x also. But for now, we do not care about that. Actually sit in it. One more complexity class or we too compute is needed in this course that is randomized algorithms. So randomized polynomial time.

And so what is a randomized Turing machine or probabilistic Turing machine? Right. So the if the so the say the correct answer is yes then this algorithm will run for polynomial time and it will give an answer, give a answer. So it will be it will give the answer yes with probability, let us say more than two-thirds. And so equivalently it will give the answer No, which is the wrong answer, with probability at most one-third.

Probability at least two thirds. And yes with probability at most one-third. So you should think about the error probability. The error probability should be one-third or less. In fact, any constant below half will do. So just arbitrarily picking one-third. And there are also these results in complexity, which if you have seen before, you can brush up. That this is equivalent to saying that the error probability can be made arbitrarily close to zero.

So how do you think that can be done? Yes, so you have an experiment where at the event of something bad happening is less than one-third. So what you can do is you can repeat the experiment hundreds and thousands of times and then take a majority vote. Because the experiment, every experiment, or run of the experiment is giving you a one bit answer. It is either yes or no.

So you collect all these bits, independent experiments and take the majority vote. And then there is a nice theorem in probability and statistics which will help you bound the probability. Error probability will become arbitrarily close to zero. Okay, so this is why these algorithms are so popular in practice, because the probability of error becomes so small that the probability of your computer breaking down is far more than the probability of your algorithm making a mistake, okay.

And since you believe your computer you should also believe the algorithm. So that so this is why this is as good as a correct algorithm. So yeah what the Turing machine is doing here is solvable in polynomial time again given unbiased coins to toss okay. So basically whenever the Turing machine has a doubt or is find the question too hard the Turing machine just tosses a coin okay and moves forward.

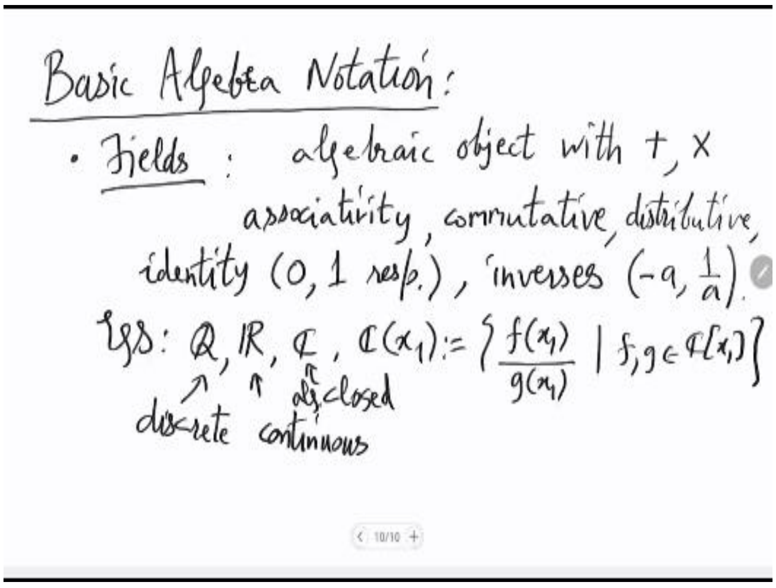
So in your C program for example, or in your Java program or whatever if you do a if you use a pseudo random number generator right, so it is the same thing in the case of Turing machine. So in every difficult step, the Turing machine is tossing a coin using the output of the coin. So it sounds strange, but the thing is that, that is not enough.

So ultimately, you as the designer of this randomized polynomial algorithm has to prove that this algorithm's output in the end has a error probability guarantee, right. So the algorithm has to be somehow intelligent. You cannot just say the algorithm cannot just for example, why cannot the algorithm just toss a coin in the very beginning? And if the coin says yes, the algorithm says yes, otherwise the algorithm says no.

So the problem of that will be that error probability can be as high as half right, which is not what the demand is. So the error probability has to be smaller than one third. So because of that gap, the algorithm has to be intelligent, it has to be somehow related to the problem. So a lot of intelligence goes in designing these algorithms. So these you can solve very interesting problems.

Okay, and all these problems come under BPP. So it is this is for bounded error probabilistic polynomial time. And so we have few more minutes. So let us come to basic algebra notation.

(Refer Slide Time: 1:06:03)



Basic Algebra Notation:

- Fields: algebraic object with $+$, $\times$
  associativity, commutative, distributive, identity (0, 1 resp.), inverses $(-a, \frac{1}{a})$.

Egs: $\mathbb{Q}$, $\mathbb{R}$, $\mathbb{C}$, $\mathbb{C}(x_1) := \left\{ \frac{f(x_1)}{g(x_1)} \mid f, g \in \mathbb{C}[x_1] \right\}$

discrete continuous alg. closed

So we will be using in this course fields and rings. So what are fields? So who can describe fields, what is a field in algebra? It is a set of elements such that with the natural properties of addition multiplication, right. So this is an algebraic object with addition and multiplication. And what you want is associativity, commutativity with addition and multiplication. And distribution should be there.

Addition should distribute an, multiplication should distribute on addition. Distribution should be there. Addition should distribute multiplication should distribute on addition and there should be identity elements, which will always say it with respect to multiplication. So for addition every element has an inverse which is a goes to -a. For multiplication, a goes to 1 over a, except when a is 0.

So – a, 1 over a okay. So that is the, those are the natural things. So when everything, all these properties are there, you call it a field. So what are the examples of fields? What are your favorite fields? Yes, so R is definitely a good field. Something smaller than R is Q. Are integers a field? Why not? Yeah, because only problem is division. Sometimes you cannot divide.

So the field which contains integers, smallest field is actually Q. And then Q is contained in R. Then R is contained in complex. Are there fields above complex? Or do you stop at complex? Yeah, no is a good answer. I wanted that answer. Any other? Yeah. So how can it be no? So the fields actually. So these are the univariate functions. So you can look at functions over a field and they also give you a field.

So fields actually never end. So over C also there is a field which is let us say C x 1 that is a field. So these are the univariate functions. So I should yeah maybe I should define this. So C x 1 is basically yeah so what you will do functions itself is not a good term. Let me make it precise. Right. So you take two arbitrary polynomials univariate over C and then you look at the ratio.

And then you study this set and you will realize that this is also a field, okay. So these this field is called the field of rational functions over C and every field has a bigger field of rational functions and hence this is actually an infinite tower, okay. The fields just never stop. Yeah all these are extensions of each other of the previous one, okay. And they have different properties of course.

So Q is kind of the most discrete field. There is no sense of analysis. R is a continuous field. Well there is a sense of continuity because for any two elements there is something in the middle right. So you cannot find two elements in R. I mean think of the real line. So on the real line you cannot find two elements such that there is nothing in between, right. So there is a strong sense of continuity here.

Complex has all these things and it also has some kind of closure. So what is this, this algebraically closed? Inverses. Yeah, so any univariate polynomial with complex so basically this f any f that you take, it will have a complex root right. This is a non-trivial theorem. This is called the fundamental theorem of algebra. This was proved by Gauss around 250 years ago. I think we are done.