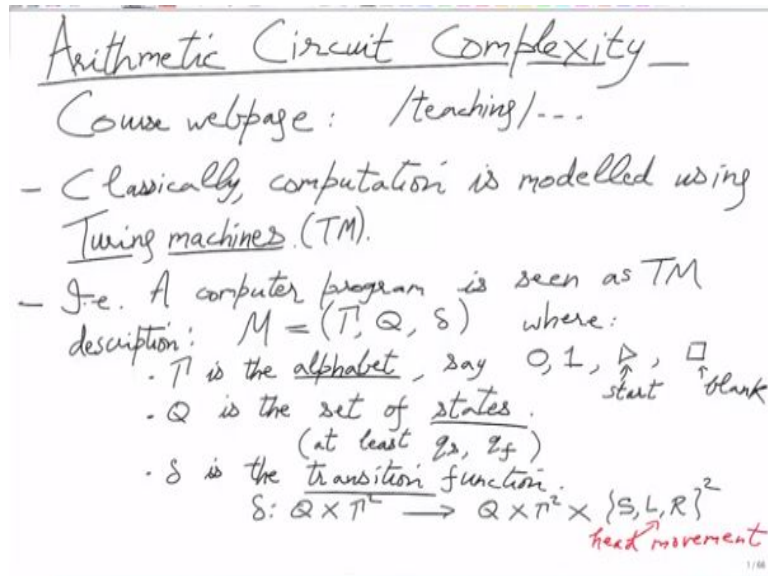


Arithmetic Circuit Complexity
Prof. Nitin Saxena
Department of Computer Science and Engineering
Indian Institute of Technology-Kanpur

Lecture - 01

So there is a course page.

(Refer Slide Time: 00:16)



There is a course page, for this you can go to my homepage; under teaching you will find CS748 for this semester. So in the course homepage you can see the lecture notes and assignments will be posted and mid sem end sem will also be posted there. They will all be takehome. And there are some ideas for advanced topics.

If you want to give a talk, half an hour talk then you can pick one of those topics that will be optional. Other grading distribution will be- four assignments mid sem and end sem. Maybe 30, 30, 40%. It is not fixed but around that. The topic is arithmetic circuit complexity. So arithmetic is probably not a very good term.

Now instead we use algebraic. This is really a course on algebraic complexity theory. This is an extension of the complexity theory course. In complexity theory you must have or in TOC you must have seen Turing machines. Instead of Turing machines we will use in this course a different model which will be in some sense stronger than Turing machines, and it will be more algebraic, for sure.

So classically computation is modeled as a Turing machine. This is what you see in theory of computation ultimately, we start with automata and pushdown automata and so on. But ultimately the real model which subsumes all the models is Turing machine. And so maybe I should quickly do a recap of Turing machines.

A computer program is seen as a Turing machine, is a Turing machine description. How do you see a C program as a Turing machine description? For Turing machine you can draw a transition graph or you can define a transition function. So that transition function is basically what a C program or any computer program describes.

You can translate, it is only a translational issue. So you translate a computer program into a transition function or this transition graph of a Turing machine. Now what do you do with memory? So if a computer program for example, wants to store something in a variable or if it is an array, then you may not even know how many variables.

The variables may be unboundedly many. So how do you simulate that on a Turing machine? For that we use the tape. That is it. The control, the transition function control is finite. That is your finite program and whatever memory the program is using that is actually unbounded, but for that purpose you have the tape in a Turing machine.

Tape is bounded. So this is how you can immediately translate any computer program into a Turing machine description. Let us go into a bit of notation for that. Turing machine notation would have an alphabet, state, and transition function δ . So where Γ , Q , and δ . So Γ is the alphabet of the Turing machine.

For us the alphabet, we can take it to be just 0, 1 okay that is what is enough to represent any other thing that you do in computers. Let us say the main elements are

0, 1 letters are 0, 1 but you may also need some special characters like start symbol (\triangleright) and blank (\square). So \triangleright is the start symbol, \square is the blank symbol.

On the tape, what you will read is initially all blank with a start symbol, let us say on the left end of your tape. So there is only one start symbol, everything else is blank, infinitely many blanks. Then once the computation, and you can assume also that the input is written in the initial part. That will be 0, 1.

Start the 0, 1 for the input string and then after that you have the blank, infinitely many blanks and on that the computation begins. The remaining part of the tape can be organized as the working space for the algorithm. We will use the simplest Turing machine description which is the left side, is the start symbol and then the tape stretches to infinity on the right.

And it is a single tape, but actually for one complexity class we will need another tape which we can call the work tape. But for most of the applications one tape is enough, except when you want to specify how much space was used, ignoring the input. If you want to ignore the input then it is better to talk about another tape, that we can call work tape. So there is this input tape and there is a work tape.

Q is the set of states. There are again two distinguished states and beyond which you may have your own states, but at least you should have the start state and the final state. So q_s and q_f . Okay so these are the two distinguished states that you will always have in a Turing machine. So the start is obviously, when the computation has not begun, and when the machine reaches q_f then the computation just halts.

And whatever is there on the tape is considered as the output. Now meaningful computation is one where q_s to q_f there are finitely many steps. We never talk about infinitely many steps. Infinitely many steps means that there is no computation, the computation field. So whenever we say computation we mean that q_s to q_f finitely many steps were taken and an output was given.

Finally, δ is the transition function. Transition function basically tells the Turing machine or the head of the Turing machine how to move and what to do. At any point of time the configuration of the Turing machine is given by what is there around the head and what is the state. Based on this what should be done next is decided by the transition function.

That is basically your one step of your C program execution. That is exactly what a computer program tells you. So δ , mathematically is a function that takes as input current state. I am using Γ^2 because I am assuming two tapes. There is an input tape, an input head which is reading input bit and the second one is the work tape head, which is reading the bit at the current work tape head.

Those three things will then decide how these heads should move and what should be the next state. What should be written by the head in the current position. The tapes can also be modified by the head. State to go to, what to write, and finally where to move. Either stay or left or right. So this is the transition function description.

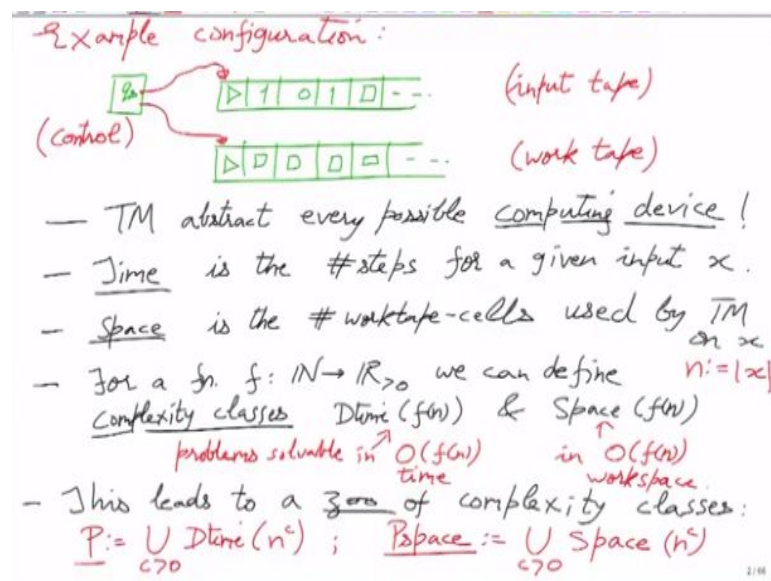
Since Q is finite and Γ is finite, δ is also finite. It just has a finite description and you can think of δ as your C program or your computer program. It is really the same thing. Let us use color. So this part is the head movement.

“Professor - student conversation starts” Can I find the start from this one. **Professor:** No, no the start is always at the left hand. **Student:** So it should be gamma minus? **Professor:** Ideally yes. Yeah. So I mean not all transition functions would make sense, but that would be taken care of by your program or by your algorithm. So you would always write a meaningful algorithm and then just translate it. **“Professor - student conversation ends”**.

In this notation, looking at the algorithm is scary, because the details here are too many data. I mean, this is completely non-intuitive. So we never actually work with this notation. We are doing it in the first class only to formalize what computation is

on a Turing machine. But obviously, when you try to solve a problem, then you find an algorithm in a very intuitive high level notation, okay, not this low level. An example, any questions about this since this page will be gone.

(Refer Slide Time: 11:58)



Okay, so an example configuration is so you have a control in the start state. q_0 is the state of this control and you have these tapes and the tape has cells. This is the input tape and this is the work tape, okay. So the start symbols are here and then you have the bits and then in the remaining ones you have the blank. Right. So the input here is for example 101 and the state is start state so the computation has not begun and there are two heads.

These for example are the two heads, the input tape head and the work tape head and then as the starting from this looking at δ , the configuration will change. That will be called one step. Okay and then so on. You do not know how many steps there will be because the tapes are infinite. Actually, because the work tape is infinite.

Since the work tape is infinite, you do not really know how many genuine different steps there will be. They could be anything from one to infinity. So this model basically is enough to capture any real life computation that humans have ever seen. Okay, there is nothing beyond this model and yeah any questions about this? So this models everything that we know.

Turing machines abstract every possible man-made device or even otherwise, okay. So till now it has always been true that whatever computing device you think of, either man made or natural its processes can be modeled this way. The reason why we are defining it is just to make sense of time and space. Time in this notation would be the number of steps for a given x .

You are interested in solving some problem and you are given an input x on the input tape and how many steps your Turing machine takes that is supposed to solve the problem that will be called the time taken to solve x . But we never really talk about a single input, right? We always talk about all inputs of length n , right?

So the number of steps should actually be, is always seen or it is meaningful to see the number of steps as a function of n , how many bits excess, right? We should, we never care about a specific input view, we actually work with all the inputs of length n and so the number of steps is just a function, that is the time complexity of a problem and space is the number of work tape cells used by this Turing machine on x .

Space is also a function of size of x which is, which we are calling n . So both time and space are functions of n and the space we do not consider the input length, okay. We just consider how much of the space of work tape was used. This is usually, this has no significance except in one complexity class. Otherwise, you can just look at one tape and everything is happening there.

Once you have defined time and space as functions of n , you can define complexity classes. This is what we do in complexity theory course, computational complexity theory course. I would not go into all those issues, but let me just talk about the main ones, in case you have forgotten. So, for a function $f : N \rightarrow R$ real valued, positive real valued, I would say, we can talk about complexity classes.

The most important one is deterministic time. Dtime $f(n)$ and the second important one is space $f(n)$. So these are, what is Dtime $f(n)$? This is the set of all those problems that can be solved on a Turing machine, on some Turing machine, in time, $O(f(n))$ and $Space(f(n))$ is a set of all those problems that can be solved in a Turing machine in work tape space, workspace $O(f(n))$.

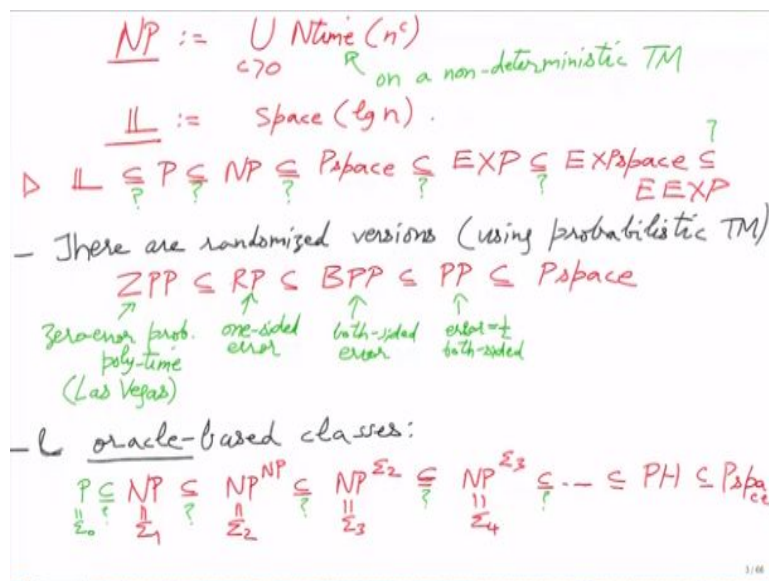
So problems solvable in $O(f(n))$ time and this is problems solvable in $O(f(n))$ space. Based on this what are the complexity classes you already know? what is the most natural specialization of f . If you take $f(n)$ to be a polynomial, then Dtime polynomial, overall polynomials will be polynomial time complexity class, P . This leads to a zoo of complexity classes.

There are hundreds of complexity classes if not thousands that are currently named and studied. We will only be talking about $P := \bigcup_{c>0} Dtime(n^c) \quad \forall c$. So if you look at all the problems, any problem or if you look at the set of all these problems that are solvable in n^c time for some c , c should be an absolute constant, right like 1000.

c is constant means that c is not a function of n , okay c is independent of n and c you are taking everything. So this is the complexity class P which is deterministic polynomial time. And correspondingly you have $Pspace := \bigcup_{c>0} space(n^c)$. we do not do D because D is not important in space. So a polynomial space is the class of those problems that you can solve in workspace n^c for some c absolute c .

And furthermore, you can define, you can look at variations of Turing machines. So for example, if your Turing machine has the ability to use non determinism which means that the transition function in one step has multiple choices, okay. Instead of transition function being a function, it is a relation. So if the transition if you have a transition relation δ then that is called an NDTM non deterministic Turing machine and the corresponding complexity class is NP.

(Refer Slide Time: 21:58)



$NP := \bigcup_{c>0} Ntime(n^c)$ This is Ntime is basically on a non-deterministic Turing machine. And finally, I have log space, $L := space(log n)$ which is a very small class, because these are the set of those problems. These are those problems that can be solved in log space. Okay, so this is the log space complexity class.

This is a very small class. So if you want to compare the classes then a simple consequence or sequence of containments is $L \subseteq P \subseteq NP \subseteq Pspace \subseteq EXP \subseteq EXPspace \subseteq EExp$

That class we have not defined, but there is a class where you can take Dtime f where f is an exponential function. So then you get EXP, okay. So problems that can be solved in time $2^{poly(n)}$ So instead of poly n, it is $2^{poly(n)}$ so 2^{n^c} for some c. So these are the problems in x. And then based on this you can also, based on exponential functions you can also define EXPspace.

So EXP will be then contained in EXPspace. And just like an exponential function you can look at a doubly exponential function. So $2^{2^{n^c}}$. And that will give you EEXP. Okay, yeah and so on. So there is no reason to stop. This is an infinite hierarchy. But

we do not know whether these containments are strict. Okay, so many of these classes could actually be equal. So what are the open questions?

So do we know whether $\log\text{-space} \stackrel{?}{=} P$. We do not know, right. That is an open question. Do we know whether $P \stackrel{?}{=} NP$. Yes, that is an easy guess. Do we know whether $NP \stackrel{?}{=} Pspace$. Yeah and so on. So any question you ask **you** it will be a question mark. So the reason is that whenever you are comparing different resources, then to date we do not have a good understanding.

log, space versus time questions or deterministic time versus non-deterministic time questions. All these will be open and the ones which are known is when you compare the same resource, so for example, P vs EXP . That is the same resource, $Dtime$. One has polynomial and the other has exponential. So that actually there is a theorem that they are different.

You know that P and EXP are different but then in the middle NP , $Pspace$ may go either way. That we do not know and similarly $Pspace$ is different from $EXPspace$. That is a strict hierarchy. There is a theorem which we cover in complexity courses. There are also randomized versions of this. You can, now look at a third variant of Turing machine which is probabilistic Turing machine.

Now the transition function is still a relation just like an NP , it is a relation, it can, from one configuration, it can move to two configurations. But there is a probability attached to those events. So let us say it moves to one of the two configurations with probability half. So okay so when you do that, then it is called a probabilistic Turing machine.

It is like your C program, that program is flipping a coin in every step. That is a probabilistic Turing machine and that gives you randomized classes. So using probabilistic Turing machine. So these are, for P there is, there are several versions $ZPP \subseteq RP \subseteq BPP \subseteq PP \subseteq Pspace$.

So ZPP is, it is what, do you already know? Zero-error probabilistic polynomial time. This I think is also called in older literature Las Vegas algorithm. Las Vegas algorithms have the property that on a given input instance x , the algorithm, whenever the algorithm halts it will give the correct answer, . The only tricky thing is that maybe the algorithm takes a long time.

But the probability of that is guaranteed to be small. So with high probability the algorithm will halt soon, like in polynomial time or polynomially many steps and the or the guarantee the other important guarantee is that whenever it halts it gives the correct answer. So this is why it is called zero error. It is also called, these algorithms are also called Las Vegas.

Yes, so the expected time complexity is yeah, I will not go into the formal definitions of these classes. In RP, so RP is randomized polynomial time. This is one-sided error, it is also called one-sided error. So if your string x is a yes string, then it may make an error. But if your string is a no string then it does not make an error, okay. This is one-sided error. BPP is both-sided error.

And this is called bounded probability or bounded error probabilistic polynomial time. Bounded error because wherever the, so algorithm will stop in polynomial time, there is no question about that. But when it stops its answer you have to take with some confidence. If it is saying yes in the answer then or it is saying no, the probability of being correct is more than let us say 66% okay.

There is no, I mean both side there are errors, but they are bounded errors. These are very good practical algorithms. In practice, you would be happy to use a BPP algorithm if it is fast. You do not really need the full power of P. You do not really need deterministic polynomial time in practice. So many practical algorithms are actually BPP algorithms.

The probability is taken as free in all practical applications. PP is probabilistic polynomial time. This is something very bad. So here the algorithm stops and

whatever it says the chance of it being correct is only half. It can be very close to half. This yeah there is a reason why this is not good. So error could be half here. So both-sided error and it is half. And all these problems, they can be solved in Pspace, okay?

There are many probabilistic versions, you can also look at the quantum model and then you will get different complexity classes. But that I will not mention here. There is a course running in parallel on quantum complexity, okay. And finally, there are Oracle based classes. So Oracle based classes are, yeah this you may or may not have seen.

For example, $NP \subseteq NP^{NP}$. So what does that mean? Right, so in the so you have a non-deterministic Turing machine and it has access to a subroutine that can solve for example SAT okay. So whenever this non-deterministic Turing machine wants to solve SAT instance, it just transfers the SAT instance to the Oracle Turing machine to the Oracle to SAT and an answer is given immediately.

That is considered one step okay. So yeah, so it is clearly a very impractical situation, you can never implement it. Because you can never get a subroutine to SAT but assuming that there is a subroutine what can you solve, okay? So those are the problems in NP^{NP} . What can you solve in a non-deterministic polynomial time? So this is called Σ_2 .

Okay this is defined as Σ_1 and then you can go crazy with this. So you can look at NP^{Σ_1} that will be Σ_2 . This is called Σ_2 and so on. Okay, so this is a hierarchy which is again not known whether it is tight or not, whether it is a strict hierarchy or not. These are open questions. Well, because, even in the base of the hierarchy this is an open question. P is Σ_0 , okay?

Everything here is an open question and this hierarchy is called yeah, so this hierarchy is called polynomial hierarchy. If you take the union of all these classes, in the limit it is called PH. Well, there is no limit, just the union. The union of all these is called PH

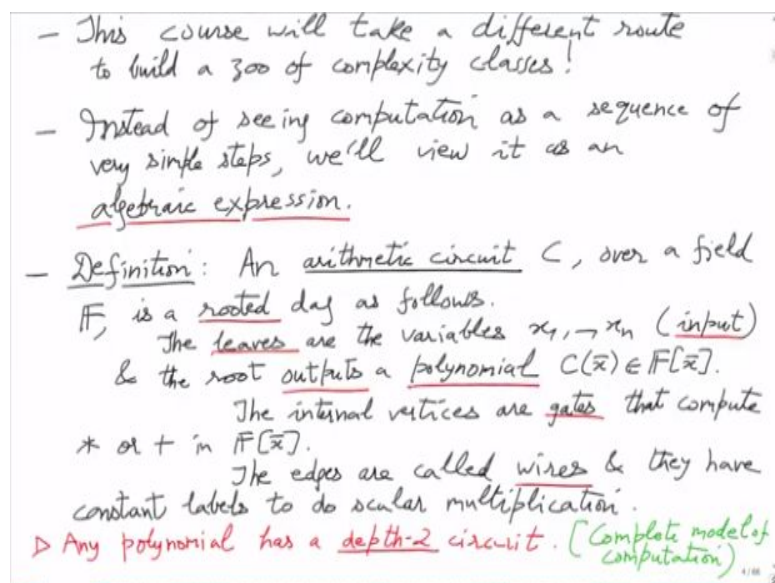
and everything in PH can be solved in Pspace. So you can see that, in inside, Pspace, there is a, there is a huge amount of diversity, okay?

This hierarchy is supposed to be an infinite hierarchy and this is all inside Pspace. Okay so Pspace contains pretty hard problems, believed to be even harder than NP although we do not know whether NP and Pspace are different. But if you believe this hierarchy to be strict then between NP and Pspace there is a, there are infinitely many classes which are all different, increasingly hard.

There are some natural problems which are actually in Σ_2 but not known to be in Σ_1 and in Σ_3 not known to be in Σ_2 . Yeah, but we will not be going into that I think. So these are the things you learn in a complexity course and then you compare these classes and you prove theorems about which one contains which one and so on.

In this course, we will take a different route. We will define, again complexity classes and we will study computation but using a different model, okay, and that model will be in many cases related to Turing machines, but it also will have different properties. Okay so we will, we will look at those things in detail.

(Refer Slide Time: 36:15)

- 
- This course will take a different route to build a zoo of complexity classes!
 - Instead of seeing computation as a sequence of very simple steps, we'll view it as an algebraic expression.
 - Definition: An arithmetic circuit C , over a field \mathbb{F} , is a rooted dag as follows.
 - The leaves are the variables x_1, \dots, x_n (input)
 - & the root outputs a polynomial $C(\vec{x}) \in \mathbb{F}[\vec{x}]$.
 - The internal vertices are gates that compute $*$ or $+$ in $\mathbb{F}[\vec{x}]$.
 - The edges are called wires & they have constant labels to do scalar multiplication.
- ▷ Any polynomial has a depth-2 circuit. (Complete model of computation)

This course will take a different route to build a zoo of complexity classes. We are doing this mostly for fun, but it is not all fun because if you prove theorems in this

different computational model, strong enough theorems then they will also mean something in the classical complexity classes, okay. So we are really studying we will ultimately we will really be studying natural problems.

In this model when we prove hardness it will also mean hardness generally. It will not, it will not just mean that it is hard for our algebraic model it will actually mean that it is hard in real life okay. So there will be a strong connection. And so this abstraction using algebra is highly motivated. See this is not just for fun. So any questions till now before I give some definitions of the algebraic model.

Instead of seeing computation as a sequence of simple steps, as a sequence of very simple steps, right? This is what a Turing machine does. Turing machine divides computation into many steps, each of the steps is trivial. There is a sequence of trivial steps and in the end something highly non trivial happens. This is how Turing machine use computation.

We instead want to view it as an algebraic expression. We will view it as an algebraic expression. So that will be the main point of departure from what we have seen before. And so what is this computational model? This we will call it arithmetic circuit as the title suggests. An arithmetic circuit, well let me first say in words before I write.

An arithmetic circuit is basically, it will have input in the leaves, it will be a tree where the input will be fed in the leaves. And then there will be gates like addition, multiplication gates, which will add or multiply the variables and that will give you polynomials. And after a sequence of such layers, in the end, the output will be given which will of course be a Polynomial.

The circuit is a tree that computes a polynomial based on the leaves as variables. So that is an arithmetic circuit. So an arithmetic circuit is, circuit C , since you want to add or multiply, you want to work over a ring or over a field. You can think of just

integers. Basically, if there is a variable X , you can multiply it by a number, let us say 10.

And then you can add another number and then you can square the whole thing and so on. , in the base, formally speaking, there should be constants with addition, multiplication operation, so this is what a ring is, but you can simply also think of integers. So over a field F is a rooted DAG directed acyclic graph as follows. There is this root which is important that will give you a single output.

The other important places are the leaves. So the leaves of the DAG of the tree, the leaves are the variables x_1, \dots, x_n . These are called the input variables. And the root of this tree outputs a polynomial. So this polynomial is $C(\bar{x})$. Okay, so \bar{x} is just the variables x_1, \dots, x_n . So what you have seen is you know the input, they are in the leaves and you know the output, it is in the root.

And the output is considered a polynomial. So this polynomial lives where? What is the polynomial ring where this polynomial lives? So this is the polynomial ring $F(\bar{x})$ right? So these are the set of polynomials in the n variables, constant from the field F , right? So now we are talking about something else right? In the case of Turing machine, we are, about computing a function that output 0 (or) 1 a decision problem.

Here it is not that is not what we are doing. Here we are not talking about computing a polynomial, right? The polynomial as a whole. This is not a functional question that we are solving. We are actually solving something more formal than a function. We are actually outputting a representation, the polynomial representation. The internal vertices are gates.

In the tree these internal vertices other than root and leaves, we call them gates and they are basically just doing addition, multiplication. Star or so multiplication or addition in the polynomial ring and write the, the internal. The edges in your tree , these are called wires, right? Since the whole thing we want to call a circuit it makes

sense that think of these internal vertices as gates and the gates are connected by wires and the current kind of flows from the leaves to the root, okay in that direction.

The wires can be used to multiply whatever is flowing on them by a constant, field constant, okay. Basically it is just scaling up whatever is whatever is fed into the wire. It can scale it up and then you can add two such things by a gate or you can multiply two such things by the multiplication gate. Basically this model can compute any polynomial, right trivially and they have constants, they have constant labels to do scalar multiplication. So this is the full model, okay?

Any questions about this? **“Professor - student conversation starts”** Analogous to Turing machine Cartesian problems this computes two constants, 0 and 1. So analogy in making. Then polynomial is just $0 \times \text{not taken}$. **“Professor - student conversation ends”**. No if you want to make an analogy with Turing machines then well then you have to talk about function.

Turing machine computes a Boolean function and here if you want to simulate the same thing then you can for example maybe you can say that I will only evaluate x i's at 0 (or) 1 and the computation will be modulo 2. Then the output I mean, although the output is still a polynomial over the finite field with elements 2. Even in that case actually, arithmetic circuit is computing something more than a Turing machine.

Because Turing machine will only give you an answer 0 (or) 1 but arithmetic circuit will give you the whole representation of that function. So you might have said that Turing machine computes a value while this circuit model computes a function, okay? It is actually, it gives you the function and then it is for you to evaluate it. You can evaluate it at any point.

So this from the very start it is actually a much stronger model. And it is highly algebraic, as you can see, it is not combinatorial. **“Professor - student conversation starts”** But in a sense this will actually be it is only computing polynomial and the transmissions in the Turing machine and the internal if you write, the internal working

as a function and that could be any function and not all functions could be captured as polynomials. **“Professor - student conversation ends”**.

Yeah. So there are differences for sure, yes. You, these two are not equivalent. **“Professor - student conversation starts”** Is there, like a natural way to see problems, like sorting, modelled as polynomials, intuitively, not directly. **Professor:** No, no. So to get to an equivalence between circuits and Turing machines, you have to look at the model of Boolean circuits. So Boolean circuits is where the gates are only computing and, or, not. **Student:** This is stronger than boolean function then. **Professor:** Well, in some sense it is stronger in some sense it is weaker. It is incomparable. So very strictly speaking these are three incomparable models, Turing machines, Boolean circuits, arithmetic circuits. But there are some similarities and you can still think of any one of these three as modeling real life computation. **“Professor - student conversation ends”**.

Once we have defined the model we have to define the resources here, right? When do we say that the circuit is a good circuit or it is a bad circuit? Because as you can see, well, already you know that it is a complete model. It can compute any polynomial, right? How do you do that? How do you model a polynomial as an arithmetic circuit? Right, a polynomial is a (sum of monomials with coefficients from the field, monomials) you can compute by multiplication gate.

And then when we have computed all the monomials you scale them up and then you use an addition gate, right? So this you can achieve in just two levels. Addition, multiplication and leaves. It is, it is a, maybe I should write it down. Any polynomial has a depth 2 circuit, depth meaning in the first layer you have addition, in the second in the bottom layer you have multiplication. So this is a complete model.

This is why we say that it is complete, complete model of computation. But that will not be enough that by itself is not enough because we also want to talk about the resources because ultimately you want to say that some polynomial is easy for circuits

and some polynomial is hard for circuits. For that, let us now define the parameters, The resources, the resource parameters, so that is basically something very natural.

(Refer Slide Time: 51:53)

- The size of the dag is called size(C).
 (sometimes, we include the bitsize of the constants on the wires)

- A max-path from a leaf to the root determines the depth(C).

- $\# \text{monomials} = \binom{n+d}{d}$, for an n -variate d -degree polynomial.
 $\binom{8}{4}, \binom{8}{0}$

- deg(C) is the degree of the intermediate polynomials computed in C.

- $f = (x_1+x_2)^8 - (x_1+x_2)^4$
 has 14 monomials

- The circuit size is small because of repeated-squaring, i.e. $(x+1)^{2^n}$.

The diagram shows a circuit for f . It starts with inputs x_1 and x_2 which are added to form x_1+x_2 . This sum is then squared repeatedly (indicated by asterisks and loops) to compute $(x_1+x_2)^4$ and $(x_1+x_2)^8$. These two intermediate results are then subtracted to produce the final output f .

The number of wires is called the size of the circuit. Let us say, number of, basically the graph size, the tree size. The size of the DAG is the size of the circuit, . Size of this graph, directed acyclic graph includes the leaves and also the edges and also the vertices. This the combinatorial representation size is the size of the circuit.

And sometimes you may also but so this is ignoring the size is ignoring something in the representation, what is that? It is ignoring the constants which are present on the edges or the wires, right. I mean in practice, somebody can object and say that what if the constants are huge. So ignoring it is not natural. Sometimes you also include the bit size of those constants.

Sometimes we include the bit size of the constants on the wires. Yeah, but formally speaking, we will not do that we will just, we will continue with the size as defined by looking at the graph only. And that is the basic resource in algebraic complexity theory if you only look at the graph size.

The question is always that given a polynomial what is the smallest graph you can design and naturally, the depth is just the length of the longest path from a leaf to the

root. A max-path from a leaf to the root determines the depth of C . You have size and depth, and we have already seen that depth 2 is enough actually. It can compute any polynomial in the polynomial ring.

But then what will happen to the size? Well size will be just as big as the number of monomials here. How many monomials are there in an n -variate d -degree polynomial,

Student: $\frac{n+d}{d}$. **Professor:** That is a bad thing. So number of monomials is equal to n plus d choose d for an n -variate d -degree polynomial, right? So $\frac{n+d}{d}$ is something like $\left(\frac{n}{d}\right)^d$ or it could be $\left(\frac{d}{n}\right)^n$

Depends on what is bigger, but for general setting this is exponential that you are really talking, about in fact I should put a constant here. So if you take $d = n$, so you are looking at $\frac{2n}{n}$, right. And $\frac{2n}{n}$ is like 2^{2^n} . This is clearly exponential in the arity and the degree of a generic polynomial. If you look at the representation at depth-2, then the size is necessarily very, very large for almost any polynomial.

It is exponentially large, right. That is the worst representation that you can have. So we are not interested in those representations, although they exist, or at least they exist. We are interested in the smallest representation, right. We will formalize that later, but you get the idea of the blow up that is happening. And finally, degree of C , so what is the degree of C for an arithmetic C ? Where, at the root? Yeah, so right.

We want to define degree to be the maximum possible degree at any intermediate at any vertex. Degree of C refers, or is the degree of the intermediate polynomials computed in C , okay. The reason is that it may happen that ultimately everything cancels out and at the root you get something very low degree, but that does not mean that the circuit computation needed that lower degree.

We here we only want to define bounds. So when we say degree of a circuit we mean the maximum possible degree at any vertex, not the degree of the final polynomial, although that is also important, but not in the definition, in the general definition.

Okay, so let us look at a small example. Let us look at the polynomial $f = (x_1 + x_2)^8 - (x_1 + x_2)^4$. If you expand this out how many monomials you will see, sorry, that is too less. This has 14 monomials. This first part has 9 and the second part has 5. And they do not cancel. They are of different degree where the first one is just homogeneous degree 8. The second one is homogeneous degree 4. You have 14 monomials.

Obviously you have a depth -2 circuit with how many, which size? Much more than 14 because you have to compute the monomials and then you have to add them. It will be 30 or so the size or even more. That will be a bad representation. There is a much more compact representation, right? The compact representation is sorry no, no but what is the circuit. So what is the DAG?

The DAG is you add, let me skip the arrows. You first compute $(x_1 + x_2)$. Yeah, then you use the output, right then you use the output twice to multiply, right? So this will give you square; $(x_1 + x_2)^2$ and you can do it once again. And that will give you, that will give you $(x_1 + x_2)^4$. So the 4 is computed, but you also wanted 8, right? So for that, let us do it once again.

Now you have both 4 and 8, you just have to add them with sign. Right, so this is the representation. That is your f at the root. So there are only 5 intermediate vertices. Overall the size is only, well the vertices are 7 and so many I think wires. But, you can see that it is a much smaller representation than what the polynomial in full expansion is, right?

This gives you an idea that using the circuit operations, you can actually compress the polynomial a lot. So one thing that we are using here is this repeated squaring. So this is a very useful technique. The circuit C , the circuit size is small because of repeated squaring here. And another example where the repeated squaring can do wonders is $(x + 1)^{2^n}$.

If you look at $(x+1)^{2^n}$ it has 2^n monomials. But by repeated squaring you can manage in n gates, right? So if say n was 100 then this polynomial, it is a huge polynomial. But it has a very small circuit. The circuit representation is a very natural way to compress a polynomial. Obviously, it is not always possible, right? So the question is: when is, when is it not possible?

When will all your clever techniques fail, and the only way to represent your polynomial would be depth-2 sum of monomials which is the worst? Right, so that is the foundational question in this area. And we still do not know the answer to that. Well, we know the answer in some sense, and we do not know the answer in general sense. So we will see as we proceed, right. In this example there are two more parameters, resource parameters.

So one is fan-in and the other is fan-out. So fan-in is the maximum indegree and fan-out is the maximum outdegree. which in this case is how much? Well it is actually 3. There is this top star which has fan-out, fan-out 3 but other than that it is 2. Yeah, so I drew it so that it is close to 2. So fan-in 2 is fine but fan-out 2 is what is giving you the kick.

Right because you are able to use the output multiple times. So you do otherwise what you will have to do is you will have to copy that and the copying will double the size. So when you are when copying is banned, then we call it a formula okay. So formulas are even more special than circuits.

(Refer Slide Time: 1:05:21)

- Fanin (resp. fanout) of a circuit is the max. indegree (resp. outdegree) of the graph.
A circuit with fanout = 1 is called a formula.
- Suppose $\mathcal{F} := \{f_i(x_1, \dots, x_i) \mid i \geq 1\}$ is a family of polynomials (call it a problem).
A family of circuits $\mathcal{C} = \{C_i(x_1, \dots, x_i) \mid i \geq 1\}$ solves \mathcal{F} if $\forall i, C_i = f_i$.

So fan-in respectively fan-out of a circuit. Fan-in is the max indegree respectively outdegree of the graph, of the DAG, the underlying graph. No. handwritten representation is formula. Because in a hand in this when you write by a pen on a paper, then there is no way to reuse computation. What you are writing in a line that is naturally a formula. On the other hand the cheating that you do in a C program right it is not in a line.

In a C program you define something called, call it a variable x and then you use x in multiple places. So that is a circuit okay but when you are writing just a algebraic expression in a line that is a formula. A program is a circuit if you look. In a program, like a C program, an expression that you have computed, once computed, you can use it, unlimited amount of time.

So that is like, that is exactly, the dependency graph is like a circuit. So a circuit with fan-out 1 is very special and natural, is called, it is called a formula. And there are numerous special cases of circuits which we will see in this course. Okay. Yes, so any questions at this point? So let me just define the notion of problem solving or solving a problem using an arithmetic circuit.

One thing that you may notice here is the size of the input is the number of variables which is n , right and for that n when you fix n on top of that you have drawn a circuit.

So it is conceivable that for different n the circuit is different. I mean, obviously it has to be drawn differently because the leaves have changed right. So for every so first observation is that for every input size, there is a different circuit.

When we talk about solving a problem or so what is the meaning of solving a problem that we are able to actually compute a sequence of polynomials. But then to compute a sequence of polynomials over different number of variables, you have to give a family of circuits. This is again a departure point from Turing machines. In the case of Turing machine just one Turing machine description was given to you.

Here you will need infinitely many in the worst case. I mean maybe all the circuits are highly correlated and you can come up with a even more compact representation. But, in general you have to give a circuit for every input size. Yes, in the case of Turing machine if you say that for a given n there is a C program and for different n there are different C programs.

That will really be a different definition of computation or resources. So suppose you are looking at a family of polynomials f_i , i -th polynomial, is in i variables. This is a family of polynomials. This is called a problem. Somebody has given you this problem which is a family of polynomials, okay? So i -th polynomial is i variate and to solve this problem by an arithmetic circuit means that for every f_i you will have to produce an i variate circuit.

So you will have to design a family of circuits. So a family of circuits solves F if $\forall i$, C_i is f_i , right. That is all. This is the notion of problem solving. Here the problem is not a single polynomial because well if somebody gives you a single polynomial then n is fixed. If n is fixed, then everything can be done in constant size, right? So that does not really give you asymptotics.

So to get asymptotics you actually need a sequence of growing arity, infinitely growing and for that, then you have to look at the circuit family and then you have to talk about sizes of the C_i 's, how is that going with i ? Using that using this now we can

define complexity classes and we can define what is hard and what is easy. Okay, so the notions, notion of hard polynomials, easy polynomials will now be based on this formalism of family.

But many times for simplicity of discussion, we will ignore the term family. We will just say a polynomial, but when I write a polynomial, it would be implicit that I am not talking about a single polynomial. But all such polynomials for growing n . Okay, so it will be implicit. We will not use this rigorous notation all the time because it becomes a mouthful to say that there is infinite family of circuit polynomials, infinite family of circuits. It will be implicit from now on. Any questions? Yes.

“Professor - student conversation starts” We look at the polynomial $(x + 1)^{2^n}$ right and we were allowed to copy so we had a small n size circuit. So but when in a formula we will have to create copies at every step. So naively that will still be 2^n right, because and but then do we actually know what is the formula size of $(x + 1)^{2^n}$.

2nd Student: You can prove that it is exponential size. **1st Student:** Is it indeed exponential size? **2nd Student:** Because you can show that the degree you need at least that much size for formula. Otherwise you cannot reach that. **Professor:** Yeah. So you can lower bound it by looking at the degree parameter. So formulas cannot exponentially blow up the degree. So formulas are actually very slow with the degree.

“Professor - student conversation ends”.

If the size of a formula is s , then the degree you can show is some polynomial in s , it cannot be more. But circuits can really blow up the degree. It can blow up to s raised to s because it can keep on multiplying the thing to itself again and again. That is repeated squaring. Yeah, that is a good point.