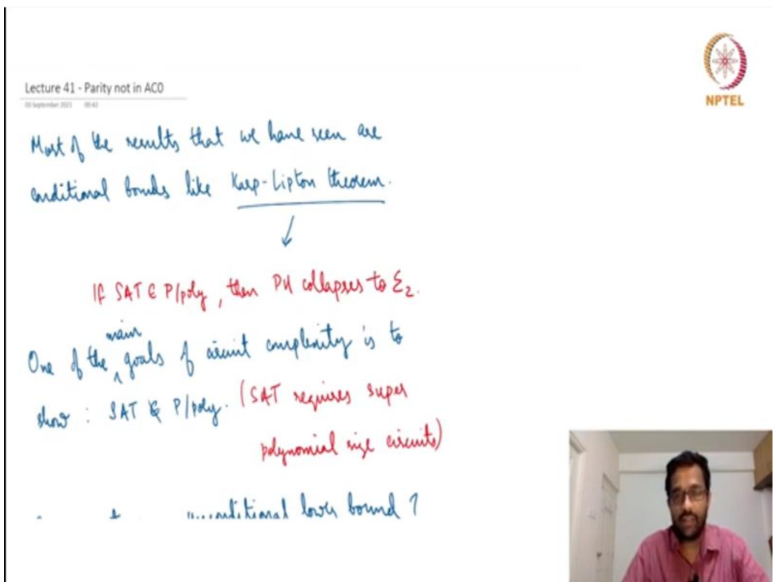
Computational Complexity
Prof. Subrahamanyam Kalyanasundaram
Department of Computer Science and Engineering
Indian Institute of Technology, Hyderabad

Lecture - 41
Parity Not in AC 0: Part 1

(Refer Slide Time: 00:15)



Hello and welcome to week 8 lecture 41 of the course computational complexity. In the past week we saw circuit complexity classes these were languages that could be decided by circuit families. So, we saw P by poly corresponding to circuit families of polynomial size and then AC and NC hierarchy where the circuits were restricted to polynomial size but there are also restrictions on the depth of the circuit.

So, NC i had for instance depth at most log n to the power i and AC was similar but then also allowed unbounded fan-in. So, we saw some languages that were contained in AC and NC and then we also saw that parity for instance was an NC 1. But then I mentioned that parity is not in AC 0. And today we will see the proof that parity is not in AC 0. So, therefore proving that actually AC 0 is not equal to NC 1.

So, this is one of the rare situations where we have an unconditional like separation. So, that is AC 0 is not equal to NC 1. We know for sure that these 2 are not equal because we have a language that is in one of them but not in the other. And most of the results that we saw and most of the results that are there even the ones that we did not see. In the case of complexity theory and particularly also in the case of circuit complexity theory are somewhat conditional.

So, they say things like Karp Lipton theorem. So, it says SAT if SAT had polynomial size circuits, then polynomial hierarchy collapses. It says if this thing happens then something very unlikely happens. So, this thing is very unlikely to happen as well. So, that is why SAT is unlikely to have polynomial size circuits. And so ideally what when circuits complexity theory like evolved as a field perhaps the original hope was to find an answer to P versus NP.

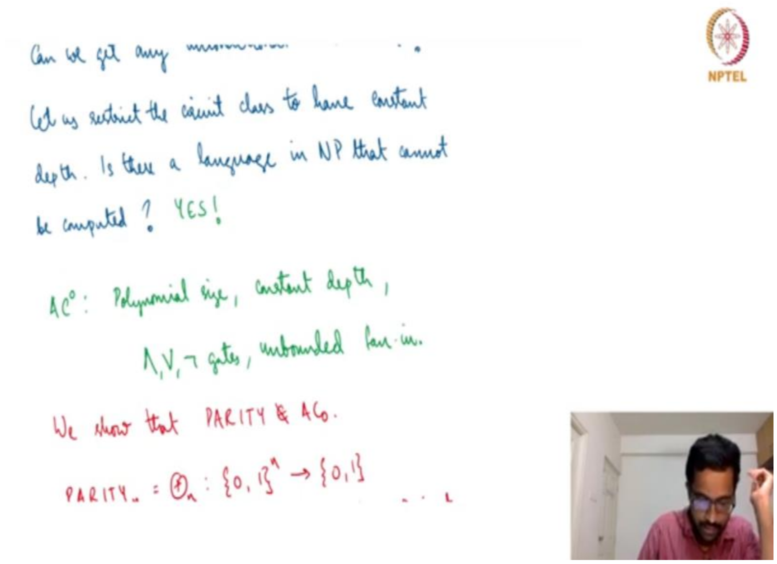
This is like in the case of many sub fields of complexity theory. So, one way to do that would have been to show that some exact some language exact does not have polynomial size circuits. Since we know that P has polynomial size circuits this would imply that P is not an NP. However, unlike like I just said we have Karp Lipton theorem which gives a conditional statement to the effect why SAT cannot be, why SAT may not have polynomial size circuits.

We do not know if any unconditional results of this type that will allow us to separate P versus NP. So, in fact the whole area of circuit complexity theory I have heard people tell me that very if you look at the kind of results that have come out of this area and if you look at the kind of people who worked on it smart brilliant people. And the amount of time they put into it the amount of results that we have is not in proportion to the amount of work.

That has gone into the area. So, it is an area where it is incredibly hard to get concrete results. So, what we will see today is a rare circuit lower bound. Especially lower bounds are very hard, so we will show that lower bound meaning some language or some language cannot be computed in a certain complexity class or in a certain algorithmic class. So, like I said we cannot we do not have any results stating that like sat is not in polynomial size.

So, now let us try to in the hope to get something let us try to restrict the class of circuits even more is there a language in NP such that a restricted class of circuits cannot it is not in a restricted class of circuits. So, what we will do let us say that is restricted so much so that it is allowed to have only constant depth. Now is there a language in NP and the answer is yes.

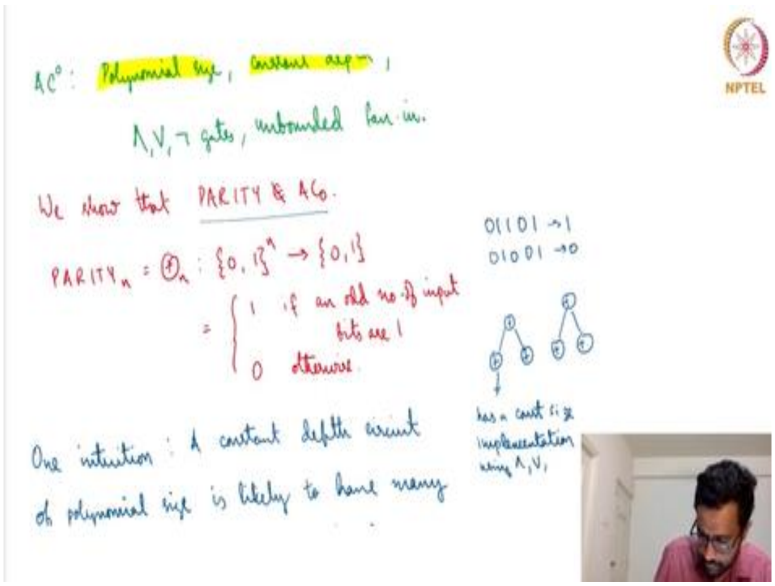
(Refer Slide Time: 04:56)



So, the answer is yes and the answer is parity, parity means it is the class of all strings that have an odd number of bonds. If you can compute it as a function as well but you can also view it as a language, so, all the strings that have all the binary strings that have an odd number of ones. And we will show that this language is not in the class of circuits that have polynomial size and constant depth. So, AC 0 is this class which has just to remind you this has polynomial size and constant depth.

And uses AND, OR and NOT gates of unbounded fan-in. Obviously AND and OR gates of unbounded fan-in NOT gates have gone fan-in one.

(Refer Slide Time: 05:43)

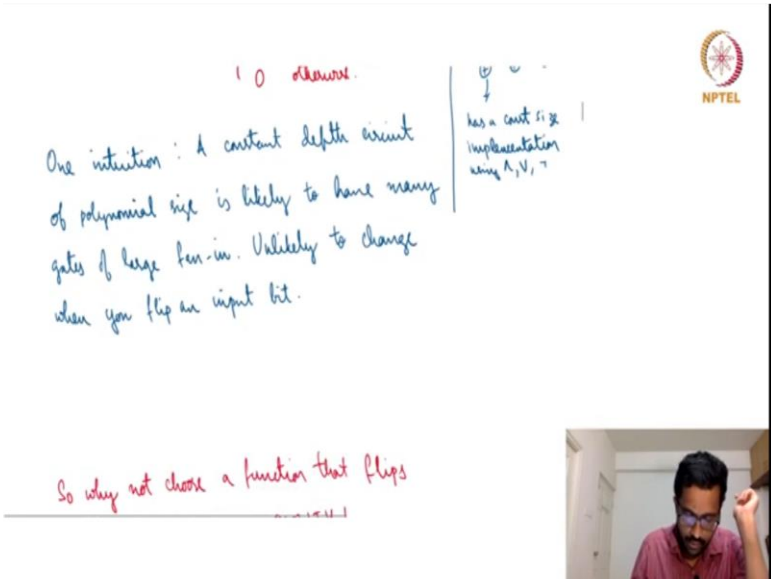


And we will see that parity is not in AC 0 and just reiterate parity is sometimes denoted with this O + plus symbol inside the O inside the circle. It is as a function you can view it as a function it is equal it is a Boolean function from 0, 1 power n to 0, 1. So, I think mostly through this proof we will be viewing it as a function not as a language. Parity of a string like a n bit string is 1 if and only if an odd number of input bits are 1.

So, parity of 0 1 1 0 1 is 1 whereas parity of 0 1 0 0 1 is 0. Because the first one has an odd number of one the second one has a even number. So, this is the parity function. So, it is a very simple function and in the previous lecture we actually saw that you could have a NC 1 circuit which means it is a circuit of log n depth that can compute this. So, it just made it just noted that we just noted that we could have a login depth tree where each consisting of the parity gate itself.

Where each branch is a or each gate is a two bit or two input parity gate. And this can be used to build a parity n function and each of this parity gate has a constant size implementation constant size implementation using AND OR NOT gates. So, that is why parity is in NC 1.

(Refer Slide Time: 07:55)



It was kind of a nice big result at that time. And there are many proofs of it and we will be seeing more of these proofs I will explain more about the proofs as we go along. So, why is this like one intuition as to why this is a good function to demonstrate not in AC 0. Because a constant depth circuit of polynomial size so it has polynomially many gates or it could have up to polynomial many gates.

But it is constant depth means it has n inputs but then constant depth means it is a rather flat circuit. So, since you have unbounded fan-in and you would expect it to have it to either independent of many inputs or if it is dependent on all the inputs or the majority of the inputs. Then the gates that are there would be rather very wide. Otherwise, you cannot really compute using only constant depth in very few number of levels in a constant number of levels you must complete the computation.

So, either you are ignoring many inputs or you are actually accounting for all the inputs but then they are all very broad gates that have huge fan-in. But if you have a gate with a huge fan-in and recall that we only have or not and AND gates. So, OR gates and AND gates that are the ones that can have huge fan-in. But if you have a huge fan-in even one if you have a huge fan-in OR gate even one input, being one fixes the output to be one.

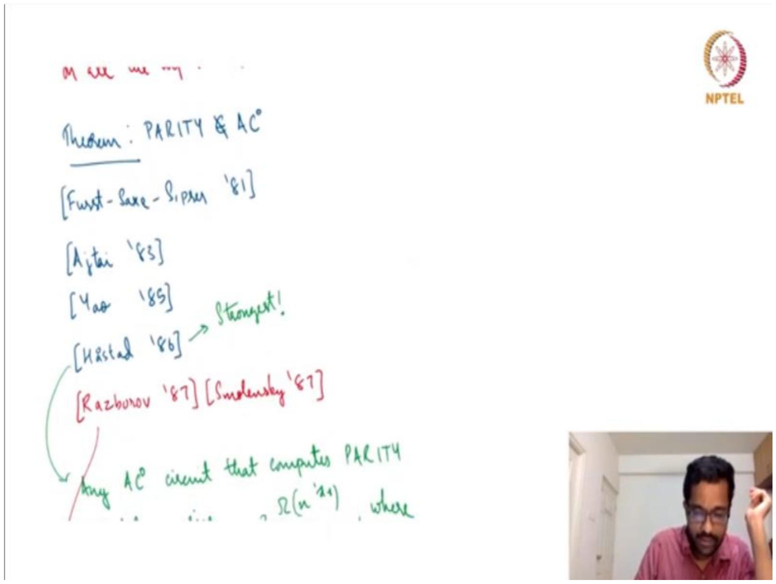
So, the other input flipping does not really change things. Similarly, if you have a huge fan-in AND gate even one input being 0 fixes the output to 0. If some other out input bits flips then the output will not change. So, the point is that there are a lot of for any input X or for any input any combination of the input bits there are most of the other bits if you flip it is unlikely to change the output.

So, and what I am talking is in general for any function that can be computed using a constant depth circuit. So, either lot of inputs are ignored in which case what I said still remains true or even if they are not in ignored, they are part of huge fan-in AND gates. So, which means it is very unlikely that the input the output flips when a specific input bit is flipped. So, this is the intuition this is saying that most of the input bits when you flip it does not change the output.

So, that gives us an intuition why parity is unlikely to be in this class because parity function is such that it just it depends on the number of input bits being. And if it is odd then the output is 1 if it is even the output is 0. So, whatever be the input, so, here I said two strings and the parity was 1 and 0. So, just one input bit when I flipped the output flipped and this is true for any input bit. If I had flipped the last bit for instance, I would have got 0 1 1 0 0 again the output flips.

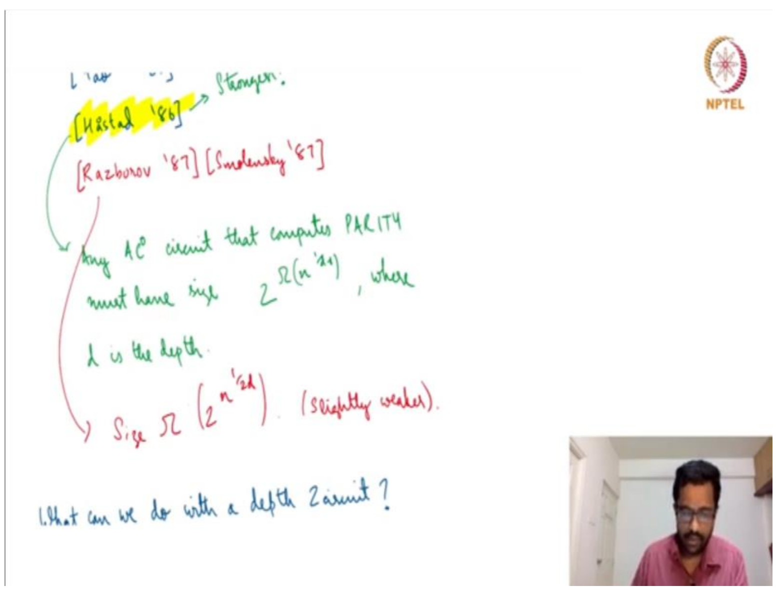
Because it depends on all the inputs and it also flips when you flip any input. So, that is one intuition as to why parity is unlikely to be in this kind of circuit with constant depth.

(Refer Slide Time: 11:48)



And this coming to the statement is just simply saying that parity is not in AC 0 and it has been proved by many people using different techniques. So, the first proof was by three people first Saxe and Siprer then Ajtai give another proof later then Yao gave another proof later then Hastad gave another proof. And then the proof that we will see today is by Razborov and approved by Smolensky in 87.

(Refer Slide Time: 12:21)



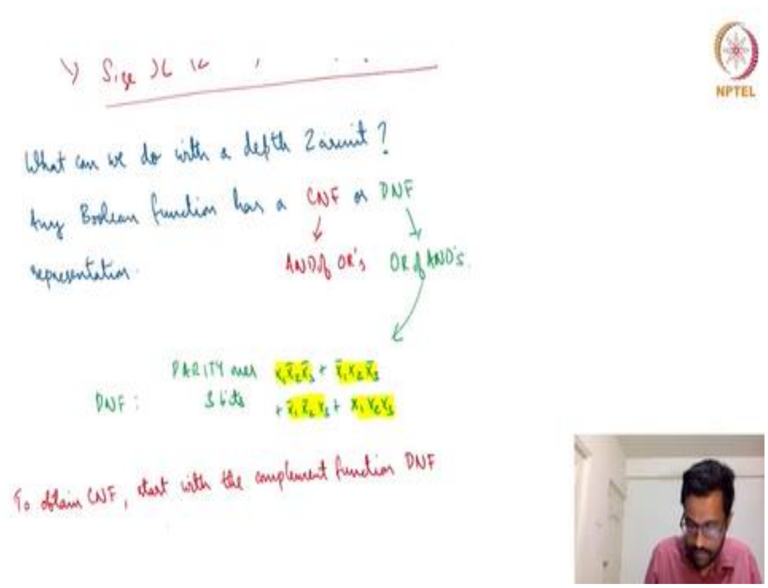
So, in fact the strengest result is by this is the strengest result in terms of the bound that we will get. Whereas Razborov Smolensky result that we will see today the proof of which we will see today. Actually, gives a gives a slightly lower slightly inferior bound but it has some other features which may be if I have time at the end of the lecture I will explain. So, let me just state what Hastad says and what Razborov Smolensky says.

So, parity is not in AC 0 is what it implies but I just stated. So, what Hastad says is that any AC 0 circuit that computes parity must have size 2 power order n power 1 divided by d - 1. So, this itself is a bit of a strain state statement because AC 0 implies polynomial so by AC 0 circuit I mean any maybe I should just rewrite it AC 0 instead of that I should say any constant depth circuit using AND, OR NOT gates.

That computes parity must have size at least this. So, this is obviously not polynomial 2 power n power 1 by d - 1 where d is the depth of the circuit. So, if the depth is 2 then this implies 2 power n must be the size or 2 power order n when depth is 10 this implies something when and for depth is 3 then this implies 2 power square root n for instance. And what Razborov Smolensky proved is slightly weaker in centre they gave the lower bound that is 2 power n power 1 divided by 2 d.

So, there is a 2 factor in the exponent of n in itself sits in the exponent of 2. But there is a two factor here which is not there in Hastad bond. But we will see the uh Razborov Smolensky proof we will see this proof.

(Refer Slide Time: 15:12)



So, before getting into the proof it probably helps to get a feel to what we can do with a fixed depth. So, what can we compute with let us say a depth 2 circuit, depth 2 meaning any from the output to the input there are only two gates any path. If you take from the root of the tree where the which is the output gate to the leaf which is usually the inputs. There are only two gates so, one thing to note is that any Boolean function has a CNF or a DNF approximation.

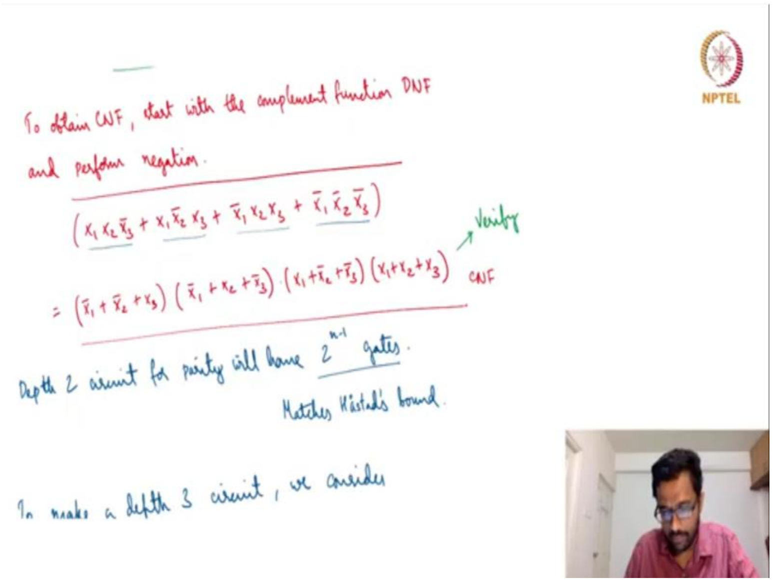
CNF, we have already seen CNF in this course, CNF means it is an expression using AND of ORs. So, it is an AND of ORs and DNF is an OR of ANDs. So, for instance DNF function let us say we have three bits and you want to compute the parity you can just collect what are so called the min terms. So, for which are the situations where the, so in other words you can just look at the truth table and gather all the situations where the output is one.

So, the parity function is one when you have exactly one output one input bit equal to one or all in or exactly three input bits be equal to one. So, the one input bit equal to one is the first three terms here x 1, x 2 complement x 3 complement this one. Then x 1 complement x 2, x 3 this is the second term is when x 2 is 1 and the other two are 0. The third term is when x 3 is 1 the other two are 0. And the last is when all three are one.

So, the first term is set only when 1 0 0 is the input second term is one set only when 0 1 0 is the input and so on. So, it is just basically breaking down the input into each possible scenario and it is cap each term is capturing a specific input n bit input and then you are taking an OR over all the n bit inputs. So, again I have used a different notation here so plus denotes OR and if I just write it side by side it is multiplication. So, this is a DNF expression for parity over 3 bits.

And you can get a CNF expression by taking the negation of parity and then taking the complement using De Morgan’s laws.

(Refer Slide Time: 17:50)



This may sound a bit confusing but let me just, so what is the negation of parity? It is all the terms that are not here right so the first term here x 1, x 2, x 3 complement is true big then it the input is 1 1 0. Second is true when the input is 1 0 1 third is true when the input is 0 1 1. So, basically all the cases when there are two ones and the last term is true when the input is all zeros. So, this is the negation of parity and there is a DNF expression for that.

And we can get a CNF expression by just taking the complement of this and using De Morgan's laws. So, De Morgan's law simply says that an negation of an AND function is the OR function of the negated things. AND negation of an OR function is just the AND function of the negative things. So, if you take a compliment over this. I think which is this is what I have tried here with the red big line, first we have the AND of each of the negation of these things.

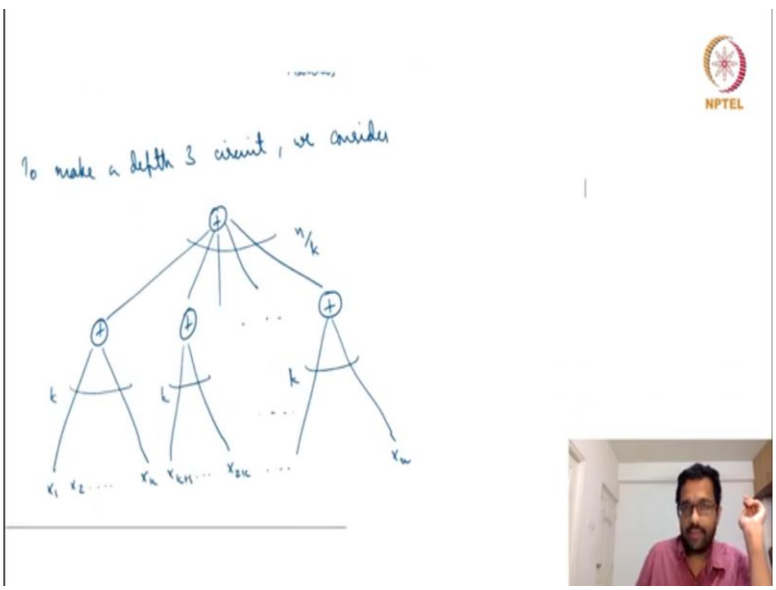
So, we have the AND of x 1, x 2, x 3 complement which actually further breaks down into the x 1 complement or x 2 complement or x 3. And then you have an AND where you have x 1 complement or x 2 or x 3 complement and so on you get this. So, you have this is the; what we have is the CNF form for parity and you can verify that this is indeed the case. Basically, you have four terms in the AND of the four terms and each of them will become 0 for a certain specific input at which point the output will be 0.

If the input is none of these four specific inputs, then the output will be 1. So, this is a DNF for parity again this is a AND of ORs and OR of ANDs. So, both of these the DNF expression over here as well as a CNF expression over here both of this this is a CNF for parity. Both of this can be expressed as depth 2 circuits it is again, we have unbounded fan-in and so it is not very difficult to see how you can express this as depth 2 circuits.

Again, if it is not clear you can work this out why how you can represent as depth 2 circuits because it is what I have written here. So, here how many terms are there it is three bit input and you have four terms. So, the depth to circuit either if it is CNF or DNF will have 2 power n - 1 gates. So, it is four here when you have three input bits if you had four input bits it would have been 8, 5 input bits you drive in 16. So, it is 2 power n - 1 gates.

And what does Hastad say? Hastad says that if the depth is equal to 2 then it is 2 power order n the d 2 -1 is 1. So, 2 power order n which is which is what we have here 2 power n - 1 is roughly 2 power order n. So, this matches Hastad bound.

(Refer Slide Time: 21:20)

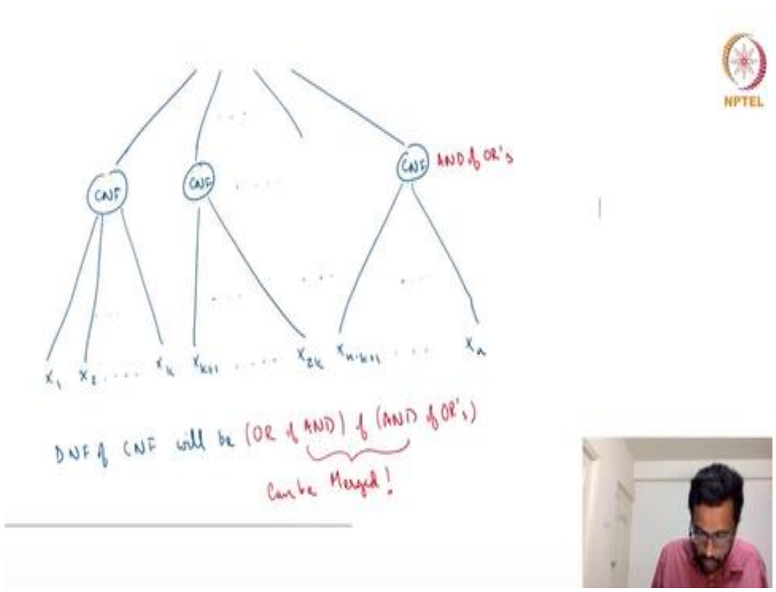


The next thing that we could possibly consider is depth 3 circuit. So, what we can see here is that when you when you have a depth 2, circuit of course you can compute parity. But then the size of the circuit becomes so huge we need 2 power n - 1 gates which is exponentially which we cannot have. So, this is what happens in the case of parity and depth 2. Let us see what happens when you have depth 3, one way to do it is to consider a 2 level tree for parity.

So, you first you compute priority of the first k bits then the next k bits and so on. So, like in blocks of k input bits first block second block and so on and then last block. Each block has k bits and then you have a top-level gate which takes the parity of these parities. So, this is a just a hierarchical thing and the parity of parity is also equal to parity. And this uses n divided by k +1 gates. So, n by k + 1 gates one top level gate and n divided by k gates at the second level.

Now what we could do is the top-level gate I could implement using DNF. DNF means it is OR of ANDs this expression the first one is written in green.

(Refer Slide Time: 23:00)



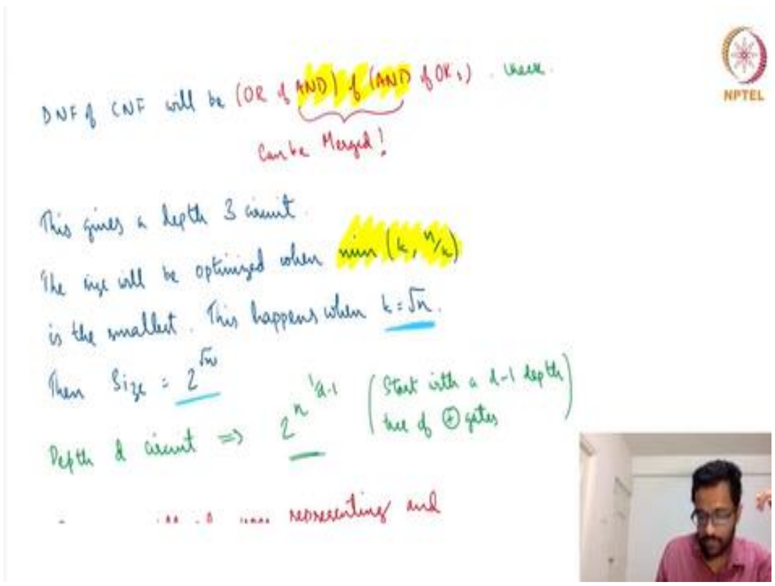
And the second level gates I implement all of them using CNF which is AND of ORs like this kind of experiment, the expression in red. So, therefore what we will have so the OR of ANDs will need 2 depth 2 and AND of ORs will need depth 2. So, one may think that this is depth 4 but actually it is depth 3 because I could merge these AND gates in from the DNF and the AND gate from the CNF.

I could just make the AND gates design it appropriately so that you can combine them into one AND gate or one set of AND gates. So, it will be OR of AND off AND of OR. So, whenever this

AND, AND are appearing nearby this AND these ANDs can be merged to just have one layer of AND gate. So, it is just an OR of AND of AND of OR circuit, this is again something that you can check. So, this will be depth 3 circuit and what will be the size here.

The size here will be optimized when this was our starting point, recall. You do it as k blocks and this is block 1, block 2 dot k blocks n by k blocks, block n divided by k each block had k inputs. So, and the number of gates will dependent on the number of blocks. So, here we have n by k + 1 gates is what I said but this is parity gate. And unfortunately, this is not a gate that is available to us. What we have is this? AND and OR and NOT gates of course.

(Refer Slide Time: 24:55)



So, the point I want to make is that the size will be optimized then it will be when k and n divided by k are minimized. So, the DNF or CNF both will the first one of them will use k gates the other one will use n by k gates. So, we have to minimize this function k or n by k and what is the k that will minimize this, minimize both of them? It will be balanced when these two are equal that will happen when k is set to square root n, k is square root n.

And then in that case the size will be exponential in square root n which is two power square root n. This is the size and this can be generalized to a depth d circuit where you can have d. Basically d - 1 you can start with the d - 1 tree of parity and you could implement them alternatively with

DNF and CNF. And therefore, you end up merging all the like OR of AND and you can merge OR of AND, AND of OR you can merge.

And at the end you will get end up getting a so d -1 parity gate tree is what I said. And at the end you will get a depth d circuit. So, maybe I will just write that as well start with a d - 1 depth tree of parity gates and then convert into DNF, CNF alternately. And again, this is optimized when at each level of this thing, the parity gates take in as input the d -1 the root of n and the depth of this entire circuit will be 2 power the d -1 root of n which is what we have written here.

Two power the d - 1 root of n and that exactly matches with what has that gave us it is 2 power the order d -1 root of n. So, till now we have been just talking about the importance of circuit lower bounds and then now we have seen how what will happen when we try to have a constant depth circuit that computes parity. And what we are seeing is that when the depth increases, I am able to control the size but still for a constant depth it is still exponential size.

But this is not a proof like maybe we are just limited in our creativity, maybe these ideas are the very simple straightforward ideas, maybe there are some very extremely clever idea that is not that looks nothing like this. But somehow manages to compute the parity function. So, this is the issue that all lower bounds have to deal with. You have to rule out all possible circuits or in the case of algorithm you will have to rule out all possible algorithms.

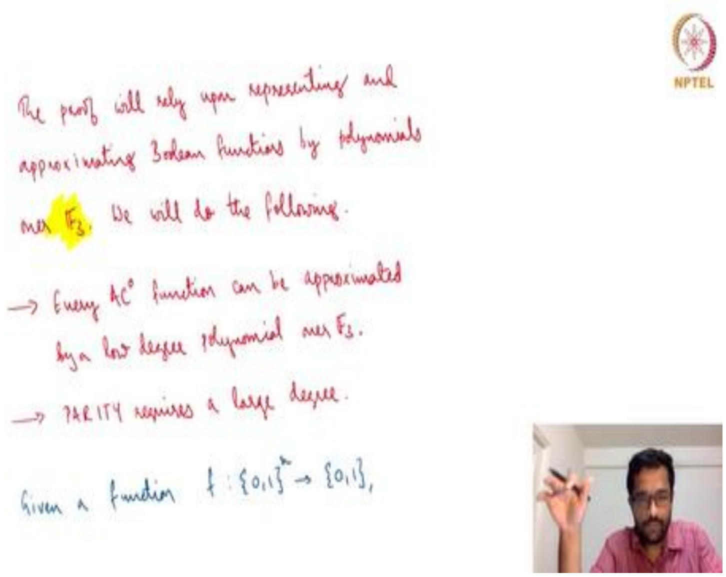
How on earth can one rule out all the possible algorithms or all the possible circuits in this case? So, maybe there are extremely different clever ways to make the circuit and maybe we are not and how do we make sure that our proof rules out each and every one of these possibilities. So, that is the reason why these bounds are hard that or that is one of the reasons why these lower bounds are hard and this particular proof is extremely interesting.

And we will use maybe two or three different techniques that you may or may not have seen. But they are extremely clever and one could actually be surprised as to how it was thought that we could use these techniques to obtain these lower bounds. Again, this proof was by Roseboro and

Smolinsky. So, the proof will rely on representing the parity function using a polynomial. In fact, it will rely on approximating this parity function using a polynomial.

So, I will say what I mean by representing using polynomial or approximating using polynomial. And we will actually you work on the F 3 field. So, F 3 is the field which has only three elements 0 1 and 2 and everything happens in the, modulo 3 sense. So, the easiest way to think of it is you only have 0 1 and 2 and everything happens in the modulo 3 sense. So, 1 + 1 is 2, but 2 + 2 is 4 but 4 is again 1, 2 + 1 is 3 which is 0, 2 times 2 is 4 which is again 1. So, you understand how multiplication addition happens both happens modulo 3.

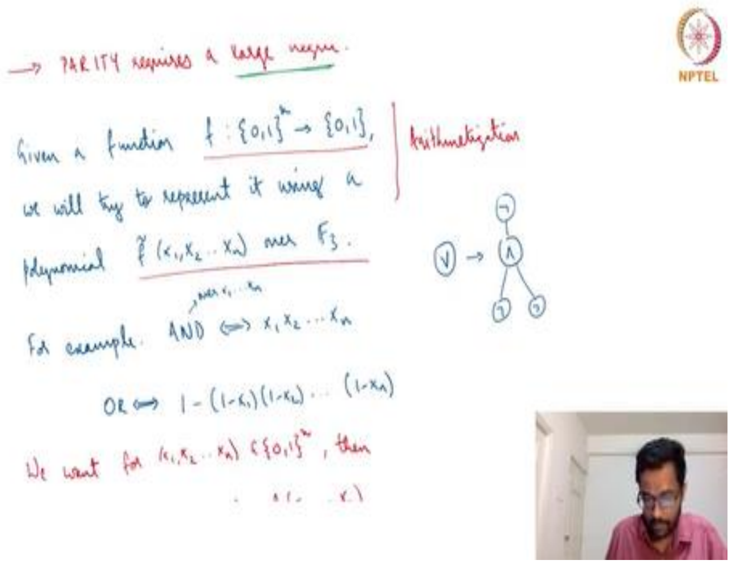
(Refer Slide Time: 30:24)



And the very very high-level outline of the proof is that any AC 0 function can be approximated by a low degree polynomial. So, any function that can be computed by an AC 0 circuit you have a low degree polynomial approximating it over F 3. So, over F 3, this is very important. But parity function needs a large degree over F 3, it actually turns out that even to approximate it requires a large degree. So, and that results in the contradiction.

One is saying that every AC 0 function requires a can be approximated using a small degree polynomial. But parity however requires a large degree. So, if parity were representable as an AC 0 function this would mean that this would lead to a contradiction. So, that is a very high-level idea of the proof. So, now I will explain what I mean by representing and approximating Boolean functions by polynomials.

(Refer Slide Time: 31:45)



So, this is kind of a sometimes it is called Arithmetization. So, we are going from or we are moving from a Boolean world where there is only 0 1 to some other world where there are maybe other things like a real number or in this case F 3. But and like in the Boolean world there is no multiplication, squaring such things are not there, addition, subtraction are not there, we only have ANDs and ORs.

But in the world of arithmetic world, you have addition, multiplication, squaring, division all of these are there. So, we will see what it means to represent a Boolean function using a polynomial. So, suppose the Boolean function is F, n bit input to a one-bit output. We will try to represent it using a polynomial f tilde over F 3. So, very simple way to explain is the AND function is simply the product. Let us say and function of over x 1, x 2 up to x 3, x n and over x 1 to x n.

This can be represented by the polynomial which is the product of x 1 to x n. So, what we want is when the input is from 0 1 power n, when the input is of a Boolean type then we want the output to be what we expect. So, when if you see when the input is 0 1 if at least one of the input bits is 0 the output bits be 0 which is what we want in the case of AND. And when the all the input bits are 1, the output will be 1 which is again what we want in the case of AND.

So, this is the AND function. How can you write the OR function? So, OR is not so straight forward. But one thing we can do is to kind of use the De Morgan's laws. De Morgan's law says that negation of AND is an OR of negations. So, you can write AND as so basically you can write OR gate as an AND gate negated AND gate and where each input is negated as well. So, you negate the inputs you make an AND and then negate the output.

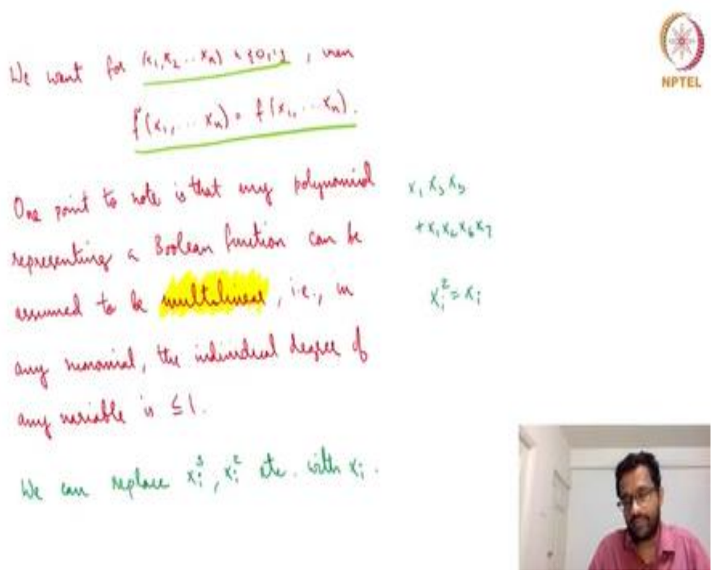
So, which gives us this expression 1 - x 1, 1 - x 2, dot dot dot 1 - x n. So, you can verify this. In the OR situation, we want the output to be 0 when all the inputs are 0. So, suppose x 1, x 2, all of them are 0 then 1- x 1, 1 - x 2, all of them will be 1. So, the product here will be 1 so 1 minus the product will be 0. So, when all the input bits are 0 the output is 0. When at least one of the input is 1, let us say x 1 is 1 then 1 - x 1 becomes 0 so this this becomes 0.

So, this entire product becomes 0 and 1 minus 0 is again 1. So, even if at least one of the input bits are 1, the output is 1 which is what we want in the case of OR. So, both AND and OR have representations using polynomial. But these representations you can see the degree of the AND representation the degree is n and even for the OR representation the degree is n. Because if you just compute the there is a x 1 x 2 up to x n term that is there.

It might have some coefficient plus or minus 1. But there is such a term that does not cancel out with others. So, this is these are ways to rip and this when the input is from the 0 1 range or not even range when the input is consisting of only 0 1 vectors then we get the desired output, but these are polynomials. So, you can even apply them in when the inputs are not 0 1. So, when x 1, x 2 everything is 2 then this gives us something.

So, you can even what if am saying is these are polynomials that have other outputs when the inputs are outside the 0 1 range. But the point is that we do not care about that. We only care about how they behave when the input is 0 1 vector. So, for when the input is from these 0 1 vectors then we want the polynomial which is f tilde to be equal to the actual function. So, again we have seen two examples AND and OR and what are the corresponding polynomials.

(Refer Slide Time: 37:02)

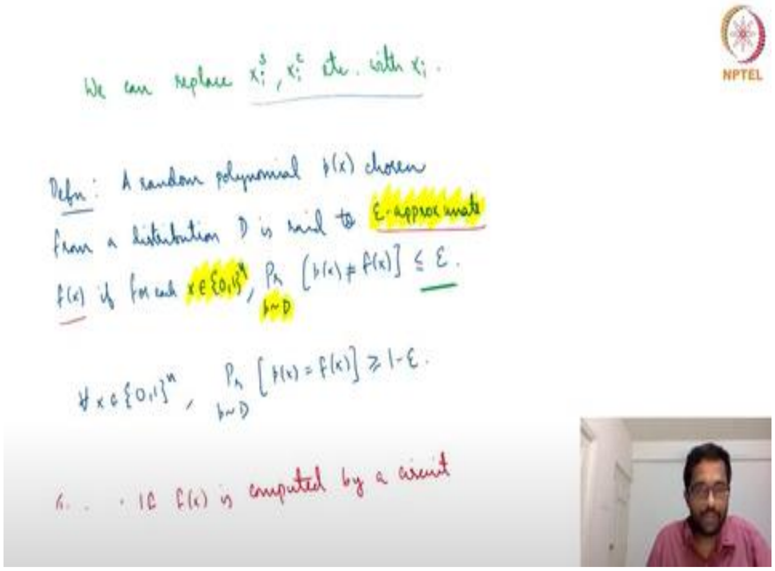


One important point is that the polynomial that expresses any Boolean function. So, these are all Boolean functions 0 1 functions we may assume that this polynomial is multi-linear. What is multi-linear mean? Multi-linear means that all the terms are of the form x 1, x 3, x 5 + x 1, x 2, x 6, x 7, all the terms are like this meaning there are no term with which has x 1 squared or x 1 cube or x 3 squared or x 7 cubed.

All the individual terms all the individual monomials have individual variables, none of them have degree bigger than one. There are no square terms with square or cubed or whatever. So, the degree of x 1 in any term will be at most 1 and so for any other variable degree of x 2 will be at most 1 x 3 will be at most 1 and so on. So, every term will be just either will be of this type, either 0 degree 0 or degree 1 for each variable. That is why it is called multi-linear.

Why this is the case? This is because we are only bothered about 0 1 inputs and 0 1 outputs. So, if you square x 1 then input is 0 1, x 1 is always so x i squared is x i itself when x i is 0 or 1 and x i cube is also x i itself. So, there is no point in doing that. So, any polynomial representing a Boolean function, we can just consider only multi linear polynomials. The individual degree of any variable is at most one and I have said here again.

(Refer Slide Time: 39:10)



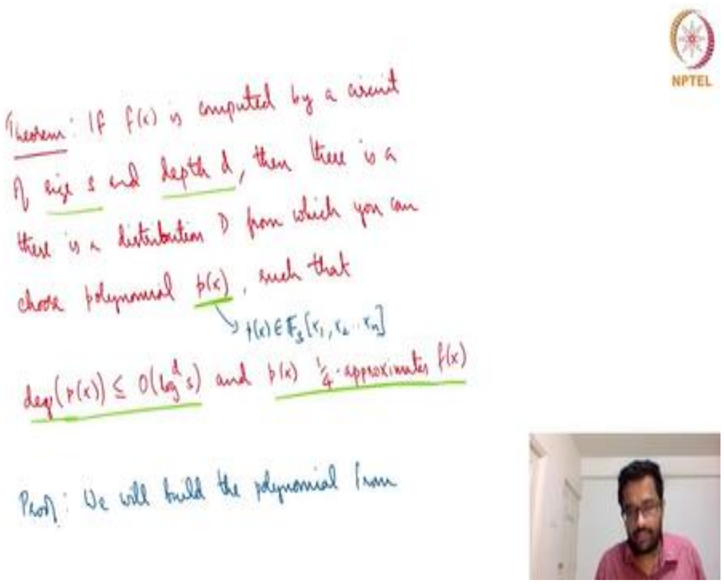
We can replace x i cubed, x i squared etcetera with x i. So, that is how we represent a Boolean function using polynomial. Now we will go to the notion of approximating a Boolean function with a polynomial. So, now when I say approximate it is a random thing that we will be doing. Suppose there is a distribution of polynomials, so there are many polynomials. And we are choosing this from these polynomials using some distribution.

And this is considered to epsilon approximate a Boolean function f, this is considered to epsilon approximate a Boolean function f, this is the key word epsilon approximate. If for any Boolean input any 0 1 input the probability that the function or the polynomial is not equal to the actual function value, this probability is at most epsilon. So, actually this has to be true for all the input sequences for any input x.

What is the probability that the polynomial is not equal to the actual function value and this probability is chosen over the distribution, all the polynomials in the distribution. So, fix the input x, for that fixed input what is the probability that the polynomial is not equal to the actual function. This probability must be at most epsilon when this happens, we say that the random polynomial or the random distribution approximates epsilon, approximates the polynomial again.

And this means if you take the negation for the all x the probability of the polynomial is equal to the actual function value it is at least 1 - epsilon. So, I am just taking the complement event here not even the entire negation.

(Refer Slide Time: 41:30)

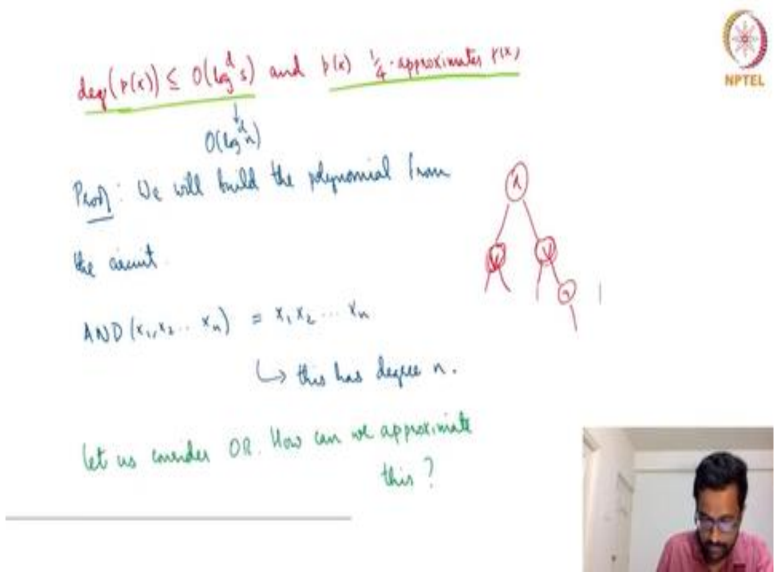


And the main the first theorem that we will prove so this is kind of more formal restatement of this every function in AC 0 can be approximated by a low degree polynomial. So, this is a more formal statement. If f x is computed by a circuit of size of size s and depth d then there is a distribution from which we can choose a polynomial p x which is over the F 3 which has coefficients from F 3 meaning coefficient 0, 1 and 2.

And everything happens modulo F 3 over the n variables. There is a distribution from which you can choose a polynomial such that one the degree of the polynomial in the distribution is at most log s power the depth and two the polynomials and the distribution one by four approximates f x. So, the one by four approximate f x which means that for any x chosen over the distribution the probability that the polynomial and the function disagree is at most one fourth.

So, this is saying that if you allow one fourth errors then the degree of the there is a polynomial that approximates 1 4 approximates the function such that the degree of the polynomial is at most log s power d, this is log s power d.

(Refer Slide Time: 43:22)



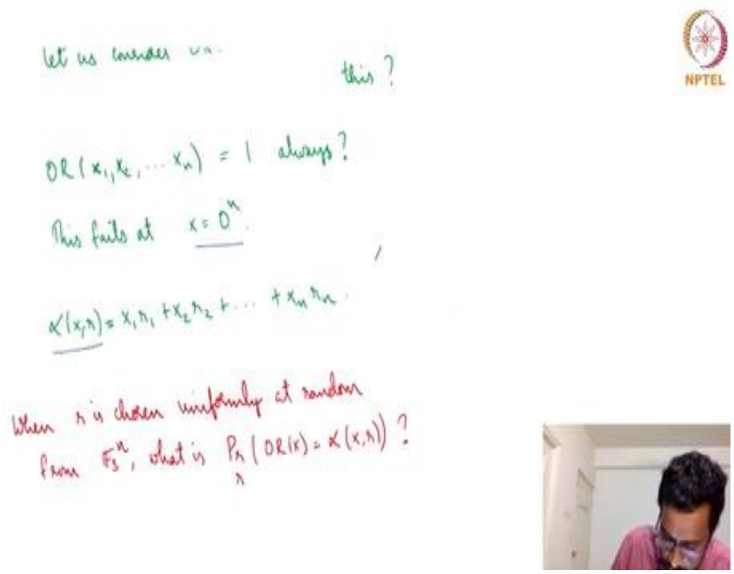
So, what we will do? So, it seems a very interesting statement. Though proof is rather direct what we will do is to look at the circuit AC 0 circuit and then we will just build a circuit using that. Basically, we will replace each gate with a polynomial and then at the end we will just compose these polynomials to get a big polynomial and each gate will be approximated by a polynomial. So, the entire function will be approximated by the big polynomial.

So, maybe I will say it now so suppose you have an AND of ORs let us say two ORs and maybe there is a NOT somewhere here. So, we will use the NOT function to approximate the NOT and we will use the OR function to approximate the OR and the AND function will be used to approximate the AND or the polynomial corresponding to the AND function. And that will take us inputs not the actual input but these polynomials as input.

The variables in the AND polynomial will be replaced by the OR polynomials here and here. So, it will be a polynomial if polynomials which is still a polynomial. So, this is what we will do. So, we have already seen that the and function of x 1 up to x n has degree n. So, this is not n factorial this has degree n. When in math one has to be careful this has degree n and we also saw an OR function that had degree n.

But unfortunately the depth we are allowed to have is order log s power d and recall we are dealing with AC 0 functions. So, s is polynomial so log s will be log n and d is constant so log n power some constant. So, this is something like order log n power sum some constant d, s is polynomial so log s is like order log n.

(Refer Slide Time: 45:49)



But so, we cannot use these AND and OR functions both of them were degree n. So, we will see how to approximate the OR function and we will use that. So, consider a very silly example. Consider one OR function which is very very low degree in fact degree 0 is to always output one. So, it is a polynomial that has degree 0. So, OR of x 1 to x n is just nearly 1. Now what happens here for all the 2 power n - 1 inputs this output is the correct answer.

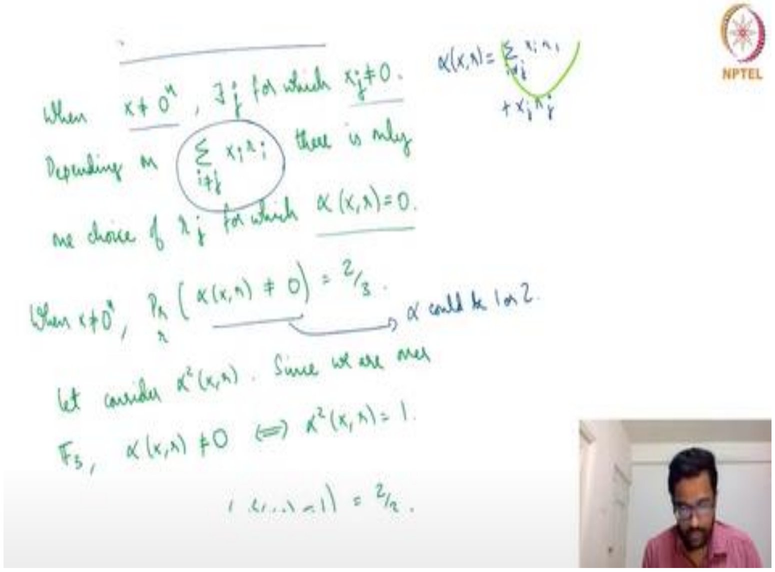
Because OR of any combination is 1 except for the all 0 combination. But when the input is all 0 this is always wrong and this is not desirable. We want for any fixed input the probability of the error to be at most one fourth. The probability of error should be at most one fourth. So, this is not good, because for at 0 power n the probability of error here is actually always there is an error. So, let us try to find another approach. So, this does not work.

So, consider the polynomial alpha x r which is simply so where x is the input and r is the random input, x is input vector and r is a random vector. Let us say random vector from it is all over F 3. All the inputs, all the coefficients now inputs is we are interested in 0 1 but x can in general be over F 3. So, random vector is also from F 3 meaning the random each coordinate of the random vector can be 0, 1or 2.

So, what is alpha x r? It is just a linear sum; it is just x 1 r 1 plus x 2 r 2 + etcetera up to x n and r n. Again, everything is over F 3 so just think of it as mod 3 that is the simplest way to think of it. So, suppose r is chosen uniformly at random from F 3 power n. There are three power and possible choices of r and each one of them is equally likely or in another way to look at it is r 1 is chosen at random uniformly and independently from 0 1 2, r 2 is chosen uniformly and independently from 0 1 2 and so on.

At each one of them each coordinate is being chosen from 0 1 2 uniformly and independently. What is the probability that? The OR function is equal to alpha so, for any x let us fix an x and then look at it and the probability being over the choice of the random string r. So, one thing is clear, when x is the 0 string alpha will always output 0. Because it does not matter what r is everything gets skilled. So, alpha x r is 0 when x is the 0 input.

(Refer Slide Time: 49:33)



What if x is not the 0 input? When x is not the 0 input meaning at least one coordinate of x is 1 we want the output to be 1. So, when x is not the 0 input there is a coordinate j for which x j is not zero. If it is not 0 again recall that we are interested in only inputs from Boolean input 0 1 power n. So, there is a j for which x j is not 0. Now depending on the sum of the other so, what is alpha? Alpha is just summation over x i r i over all i equal to 1 to n.

Now consider the summation of all the terms except the x j and alpha j term except the jth term. We know that x j is not 0, we do not know what alpha j is. So, you look at the sum and x we know x j is not 1. Now which alpha j will make the whole sum? So, what will happen to the whole sum can be just changed by changing the value of alpha j. For instance, if the rest of the term, sum up to 1 and x j is equal to 1. If alpha j is 2 then the whole summation is 0.

So, in other words maybe I just write it here, alpha x r is the sum of everything else i not equal to j x i there is an error here. It is x i r i + x j r j what I am saying is that whatever this summation is now depending on we know that x j is not zero depending on the value chosen by r j we can always make the total sum zero with for a specific r j. So, there is only one choice of r j for which alpha x r becomes 0. The whole sum alpha becomes 0 and this is an undesirable situation.

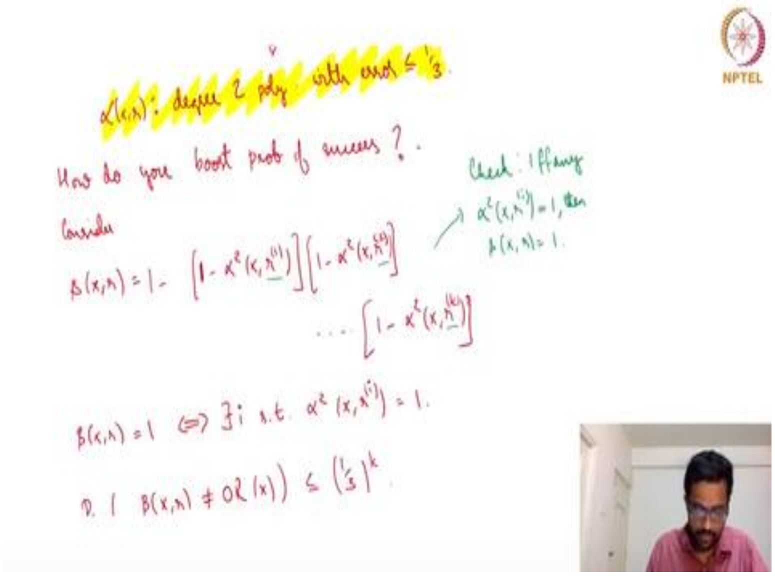
We are trying to compute r and we know that one x is not all 0 which means at least 1 bit is 1. And we want the alpha to be not zero so there is one the problem is there is one choice of r j for which alpha is not 0. So, for which there is exactly one choice of alpha for which alpha is 0. Then so just to summarize when x is not the all 0 vector there is the probability that alpha is not 0 is 2 by 3. For the other choices of r j, alpha will not be 0.

But notice alpha is a polynomial, so when it is not zero there are two possible values two other possible values it can take. We are all working over F 3, so it could be 1 or it could be 2. When you say when I say 0 not 0 it could be 1 or 2. Alpha could be 1 or 2 and we do not like 2 because we are trying to approximate or represent a Boolean function so we want the output to be 1. But we know we know it is not 0 so at least so far so good.

How do we get from 1 or 2 to always one? So, what we do is a very simple thing. We just take the square of alpha. So, the point here is that 1 square is 1 and 2 square in the mod 3 world 2 square is 4 and 4 mod 3 is also 1. So, both once if it is regardless of whether it is 1 or 2 its square will always be 1. So, whenever alpha is not 0 alpha square will be equal to 1. So, which means I can again rewrite when x is not the all 0 input.

The probability that alpha square is equal to 1 = 2 by 3. So, alpha was a degree 1 polynomial, alpha square is a degree 2 polynomial and when x is 0 there is no error.

(Refer Slide Time: 54:48)



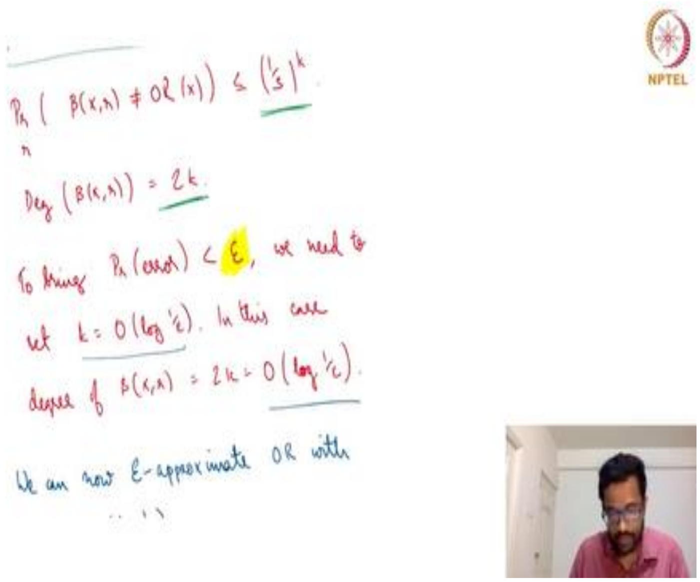
When x is 0, we already saw that it always outputs 0. When x is not 0 there is an error with probability one by 3 it may give 0 value. So, overall alpha x r is a degree two polynomial with error at most one third. What we wanted is a general approximation for any function with error at most one fourth and this r is going to be a building block. But so, we have some representation with error one third at most one third.

How do we improve the probability of success or decrease the probability of error? Well, we have seen this in the case of randomized algorithms. We just repeat it and then boost the probability of success. So, suppose you do several alphas or several alpha squares, each alpha having a separate set of random bits. And then then what do you do? You output so you know that when the actual output intended output is 0 there is no error, it always outputs 0.

The error happens when the intended output is 1 and sometimes it outputs 0. So, any one of these outputs being 1 we want to output 1. So, we again use the same trick, you do 1 - alpha square, 1-alpha square etcetera and k such times 1 - alpha square with r superscript 1 being the first set of random bits and superscript 2 being the second set of random bits and so on. So, there are k such sets and you can check that if any alpha square x r i = 1 then this whole thing I am calling beta.

Then beta x r = 1 or rather it is if and only if beta x r will be 1 if and only if this happens. So, this you can check that this. So, you have 1 - 1 – 1- everything take the product and then 1 - that. So, again this is like the similar De Morgan idea that we saw earlier which is in other words it is again the same thing that I have written here. Beta is 1 if and only if there is an i such that the ith alpha squared entry is 1.

(Refer Slide Time: 58:00)

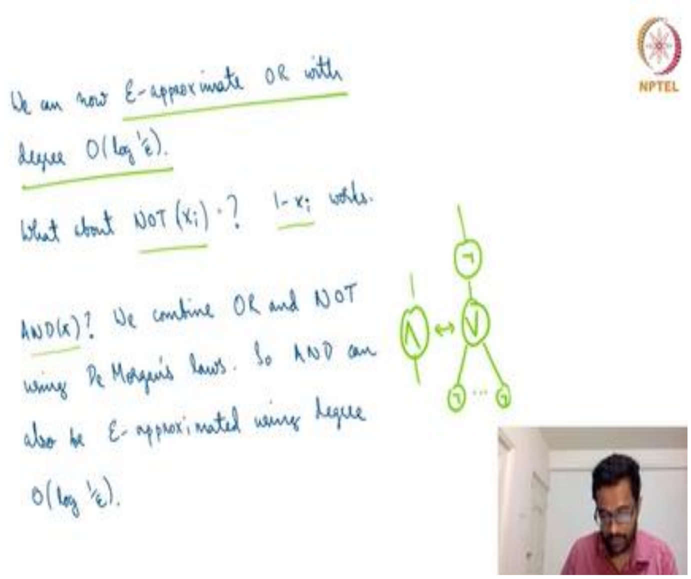


Now what is the probability that? So, the only probability of error now happens when all the k times you get the wrong answer. The intended answer is 1 but you output 0 in all the k times. So, what is the probability of error in 1 trial or 1 alpha? It is one third. We saw that it is one third so the probability of the beta a ring is at most one third power k. So, now you can see where the how the improvement is coming.

So, and the degree is simply alpha is of degree 2 so beta is you have k alpha is multiplying. So, beta is degree is 2k. Now what must k be so that or what must the degree be so that the error is sufficiently low. So, we want to bring the probability of error to some epsilon let us say. We want to bring the probability of error to the epsilon which means 1 by 3 whole power k should be epsilon which means k should be order log of 1 by epsilon and the degree of beta is just twice k.

So, which means the degree of beta also should be order log 1 by epsilon. So, if you want a epsilon approximation you better do, you better take degree order log 1 the epsilon which is still ok. Epsilon is some constant so it is still not dependent on n etcetera, so order log 1 by epsilon. So, now we have seen how to epsilon approximate for any epsilon the OR function using a order log 1 by epsilon degree polynomial. This is our main building block and it turns out that this is enough.

(Refer Slide Time: 1:00:16)

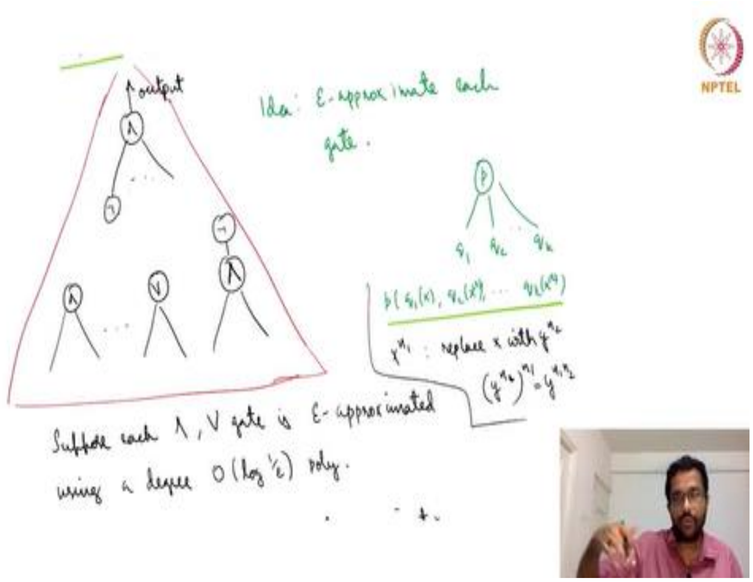


So, now what we have done is we have epsilon approximated or with a degree 1 order log one by epsilon. How do you epsilon approximate NOT? NOT is you can easily directly represent this you do not even need to approximate, just take 1 - x i. NOT is always fan-in one so you just take one minus x i. And how do you represent AND? We did De Morgan trick so AND can be represented by take the OR all the inputs also have NOT gate and the output also has NOT gate.

So, this is how you represent AND. And since we know that the NOT gates do not have any error the error of the AND gate will be same as the error of the OR gate. Whenever there is an epsilon error in the OR gate that will result in n that could result in epsilon error in the AND gate. And whenever the OR gate computes without error the AND gate also will compute without error. So, AND gate also can be computed or epsilon approximated using degree order log 1 by epsilon.

Now so we have seen how to epsilon approximate for any epsilon OR gate AND gate and NOT gate. NOT gate of course there is no error. What did the theorem want? Theorem wanted to 1 by 4 approximate any function with a log s power d degree. So, let us see how that happens.

(Refer Slide Time: 01:02:09)



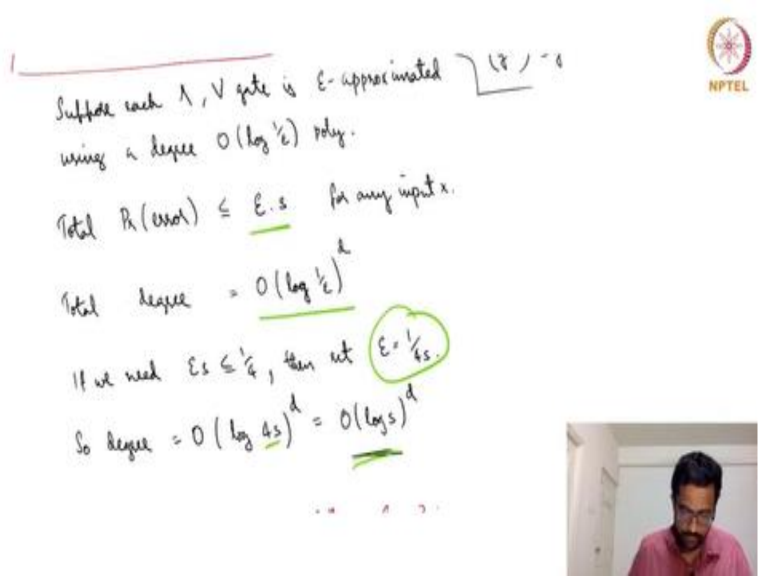
How do you 1 by 4 approximate a general depth d size s circuit? So, three things one by four approximate depth d size s circuit. So, what we do is we have some circuit which is an AC 0 circuit, some depth d size s circuit not necessarily AC 0. Let us say the circuit has AND gates and OR gates and NOT gates. We know how to epsilon approximates each one of them. So, the idea is epsilon approximate each gate, you look at the circuit you epsilon approximate each gate.

Now what does it mean? So, suppose there is a polynomial let us say P here that takes us inputs polynomials. Let us say some minute or q some q s then let us say q 1, q 2, etcetera q 1 q b 2 etcetera q k then what it means is that the polynomial takes this input p of q 1 of some x or whatever q 2 of maybe some x 2 something like this. So, basically it is a polynomials of q k of x k something like this. So, what I am saying is how it is just a composition of polynomials of other polynomials.

So, the degree will blow up. So, let us try to estimate the error and the degree for this. So, this is a size s circuit, each one of the gates we are replacing with an epsilon approximation. So, each of one of the gates has for any fixed input for any fixed input can result in an error with probability at most epsilon. So, the s gates each causing an error with probability epsilon. So, total probability of error is at most epsilon times s.

So, this is the union bound, an error could happen in this gate and another error could happen in this gate. But these errors could even coincide. But the worst case is that each error happens in a separate eventuality separate outcome. So, the worst case is when the probability of error is at most epsilon multiplied by the number of gates.

(Refer Slide Time: 01:05:09)



So, the total probability of error is upper bounded by epsilon times s. What is the total degree? So, whenever you do this kind of composite composition like I said here p of q or whatever. The degree of the resulting polynomial is the degree of p multiplied by the degree of q. So, to see why so suppose it is like saying I am computing x power n 1. Now I am replacing x with some y power n 2 so, I am replacing x with so maybe I will just write it again x power n 1 and replace x with y power n 2.

So, it is like y power n 2, whole power n 1 which is nothing but y power n 1 n 2. So, the degree is actually multiplying so n 1 and n 2 they multiply. And when you have depth d you could have up to d levels of such recursive polynomial. So, we know that at each level the degrees or order log 1 divided by epsilon. So, it is order log 1 divided by epsilon multiplied by order log 1 divided by epsilon and so on d times at most because depth is d.

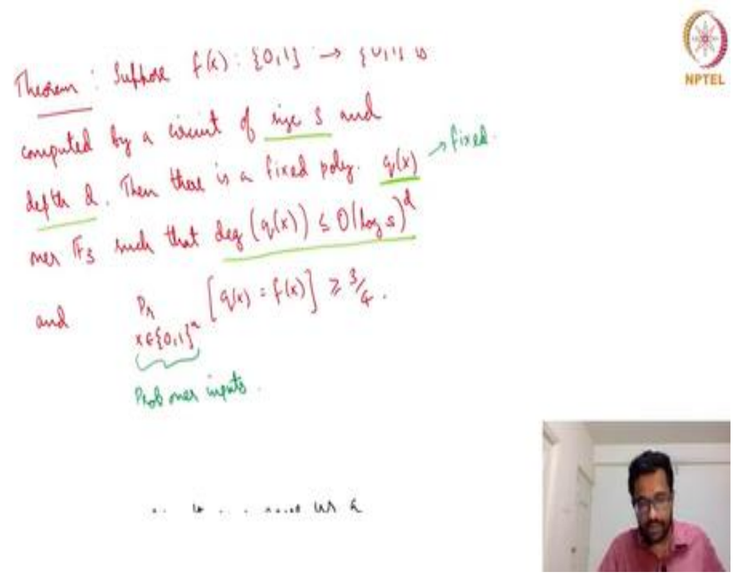
So, it is simply order log 1 by epsilon whole power d that is the degree. So, for this construction the error is epsilon s and degree is order log 1 by epsilon whole power d. Now what was the target again? The target was to was to show that for any size s and depth d circuit, you can 1 by 4 approximate it using log s whole power d polynomial. So, 1 by 4 was the target error. So, if the target error is 1 by 4 then all you need to do is epsilon s should be 1 by 4.

So, we set epsilon to be 1 divided by 4 s. Now we plug this value of epsilon into the degree. So, 1 by epsilon is 4 s or so you can just replace 1 by epsilon by 4 s here. So, order log 4 s whole power of d, but log 4 is simply log 4 + log s which is simply 2 + log s. But then anyway we have the O notation which can absorb that so it simply boils down to order log s whole power d. So, what we have here is this; what we set out to prove. What was that?

This was that we have a polynomial or polynomial chosen from a distribution of degree order log s whole power d that 1 by 4 approximate f x so f x is any polynomial. So, f x is any function that can be computed by an AC 0 circuit. We have shown how a polynomial chosen from a polynomial family from a distribution can 1 by 4 approximate it and where the degree of the polynomial is log s whole power d. So, this is how we get that.

So, this is like the first kinds of this first bullet point here every AC 0 function can be approximated by a low degree polynomial over F 3.

(Refer Slide Time: 01:09:34)



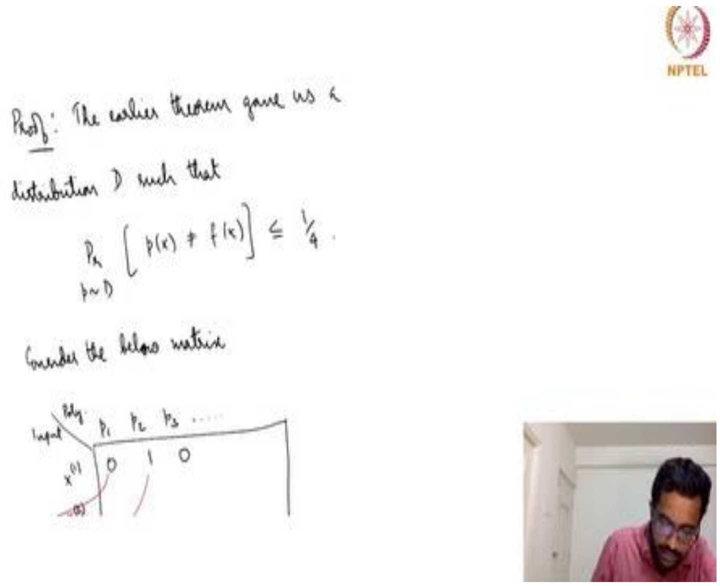
Now I will just say one more thing and then conclude this lecture. So, what we saw was a polynomial shows a distribution from which you can choose a polynomial that has bounded error. Now let us change the error around. So, we till now we had error defined as for a fixed input x, the error was over all the choices of random bits. Now instead let me just not have any randomness let us fix a polynomial and look at what is if you just vary the input inputs.

Now what is the probability of the errors? So, now the claim is that suppose you have a function f computed by an AC 0 circuit or circuit of size s and depth d. Now we are claiming that there is a fixed polynomial q that has degree order log s whole power d which is the same as what we had in the previous theorem such that the probability of q x = f x is at least three fourths. And in this probability, please note this probability over inputs not there is no randomness here.

Or the randomness is over the input choice of input, the polynomial is fixed. So, in the earlier case we showed a distribution of a family of polynomials and a certain distribution such that if you choose a polynomial from that distribution the probability of error for any input x is at most one fourth. Now we are saying that there is a specific polynomial, there is one polynomial we are not saying how to find it or anything.

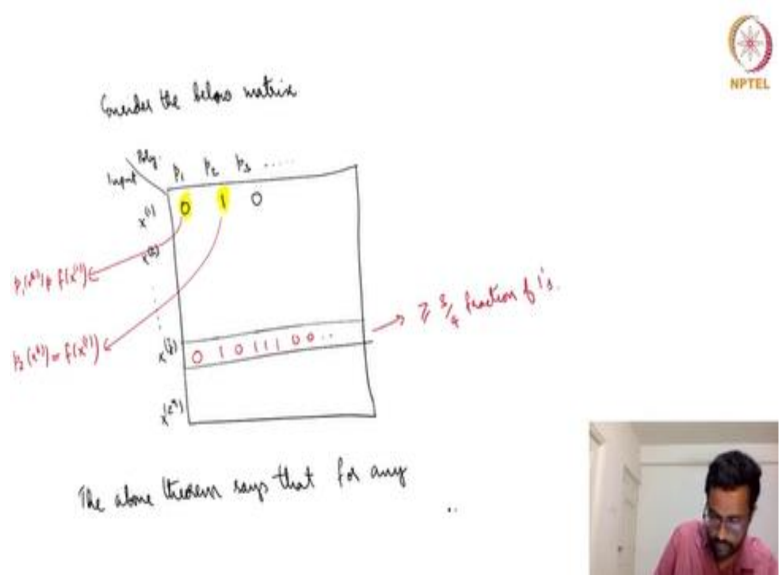
There is one polynomial that has the same degree log s whole poverty such that if you just vary the input if you choose the input randomly the probability that the polynomial is not equal to f x is at most one fourth or the probability of the polynomial equal to f x is at least three fourths.

(Refer Slide Time: 01:12:03)



And this just follows from the earlier theorem and it is very simple and easy to see. So, it is rather short. So, the earlier theorem gave us a distribution whenever the polynomial is chosen from the distribution. The probability that sorry the probability that P, the polynomial is not the same as the function is at most one fourth.

(Refer Slide Time: 01:12:33)



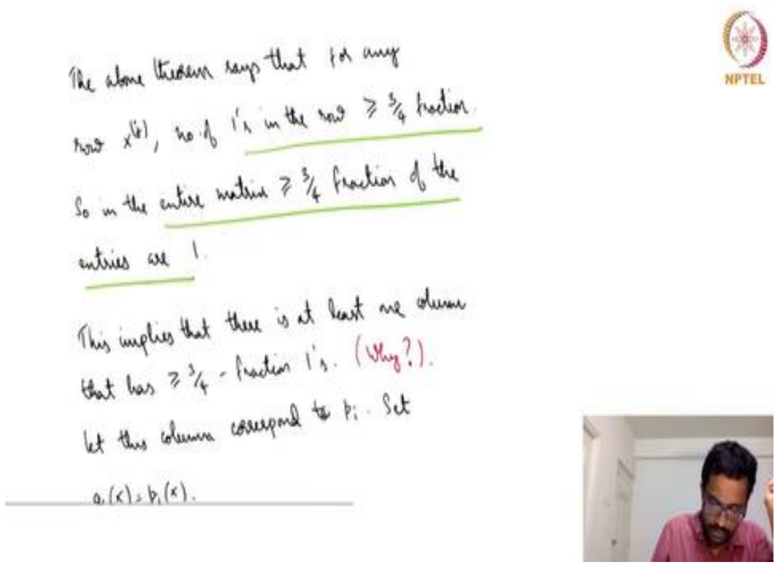
So, now let us make the following matrix. So, each row is indexed by an input sequence. So, assuming we have n input bits, we have two power n possible inputs that we are interested in all zero input going to all one input. So, I am representing them as x superscript 1 x superscript 2 x and so on. And these are the rows, the rows are the inputs. The columns are the various polynomials that are there in the distribution.

So, I do not know how many polynomials are there and the entries are 0 1. What do they indicate? 0 indicates that there is an error. So, if this 0 at the top left indicates that P 1 the polynomial P 1 computed at the first input x superscript 1 is not the same as the function at x square superscript 1 and this one that says that the polynomial 2 at x superscript 1 is equal to the function at x superscript 1. And likewise, we can fill up the matrix. What do we know?

We know that what did the theorem say this theorem says that for any input right let us fix an input x 1 or x 10 the probability of the polynomial about the polynomial is 2 is chosen from the distribution from the columns. The probability that a certain polynomial chosen from the distribution gives an error is at most one fourth or it agrees is at least three fourths which means that you fix an input let us say x j, it may have some 0 some 1 something.

So, it may have 0 1 0 0 1 1 something. When does it have an 0, it has a 0 when the polynomial does not agree with the input, sorry the function, 1 when the polynomial agrees. We know that once you fix the input the probability that the polynomial agrees is at least three fourths. So, which means in this row it contains at least 3 by 4 fraction of ones this is what I am saying here.

(Refer Slide Time: 01:15:07)



Number of ones in that row is at least 3 by 4th fraction. Now I can say this for any row. First row has at least three fourth fractions; second row has at least three fourth fraction and so on. Which means that if you look at the matrix overall the; entire matrix at least three fourths of its entries are one. If you look at the entire matrix at tree three fourths of the entries are one. So, this matrix is 75% ones. Now let us change the perspective, now let us let us try to look at columns.

I know that there is a matrix with entire fraction of ones is at least 75%. Now all I am saying is that there is at least one column that has 75% ones. All I am saying is that there is at least one column that has 75% ones. Why is this true? This is true. Because we know that 75% of the entries are ones. If there is no column with 75% once that means all the columns have less than 75% ones. Let us say all the columns have 70% ones.

That means the total fraction of ones is also in the entire matrix is also 70% that is a contradiction. So, there must at least be one column which 75% ones. In fact, there may be more than that. So, now let us say that column is P i, this column is P i that is 75% ones this means that the polynomial this column is a polynomial this P i has 75 or three fourth ones which means for this particular polynomial P i gives75% of the inputs.

This polynomial P i give the correct answer on 75% of the inputs. This column corresponds to P i now you look you choose that polynomial. So, the state the theorem says there is a polynomial q which agrees with the function on 75% of the inputs. So, P i choose P i to be that q that has at least 75% ones which one means an agreement with the actual function. So, that shows that there is a polynomial.

So, again the statement is that if there is a function f that can be computed by a circuit of depth d and size s there is a fixed polynomial. Now there is no distribution over F 3 again we are still over F 3 such that the degree is order log s whole power d. And if you randomly pick the input the probability that the polynomial agrees with the function is at least three fourths. So, that is what I want to show in this lecture.

And in the next lecture we will contradict this meaning we will say that so what we have shown now is that we first showed that every AC 0 function can be approximated by a low degree polynomial over F 3. And then we actually showed now that every AC 0 function there is a specific polynomial that agrees with the function on 75% of the inputs where the polynomial is also low degree. But then we will see in the next lecture the parity requires a large degree.

So, that will be the contradiction. So, just to summarize we saw, we motivated why it is important to have circuit lower bounds. Then we saw the intuition as to why parity may be the right function. We saw how for circuits of small depth we saw that the size has to be large if they were to compute parity in fact matching the bound by Hastad. And then we set out proving the Rasbora and Smolensky result.

We first saw that notion of randomly random polynomial approximating a function. We said that a function can be computed by a size s and depth d circuit there is a polynomial and a distribution or there is a distribution from which if you choose a polynomial. This polynomial will be over F 3 so the degree is order log is whole power d and the polynomial 1 by 4 approximates f. And then we saw that using this result we actually get a single polynomial over F 3.

That will agree with the function on three quarters many inputs so if we let the input change. So, there is no randomness on the polynomial, polynomial is fixed. Then it will agree with the input agree with the function on three quarters of 75% of the inputs and we will see why this is so. If it is an AC 0 circuit, the size s will be polynomial and depth will be constant then the degree will be again I have said this earlier the degree will be order log n whole power d.

In the next lecture we will see that this is a very low degree and we will see that to compute parity we need higher degree. We have already seen how much degree, how much size we needed and that will lead to the contradiction.