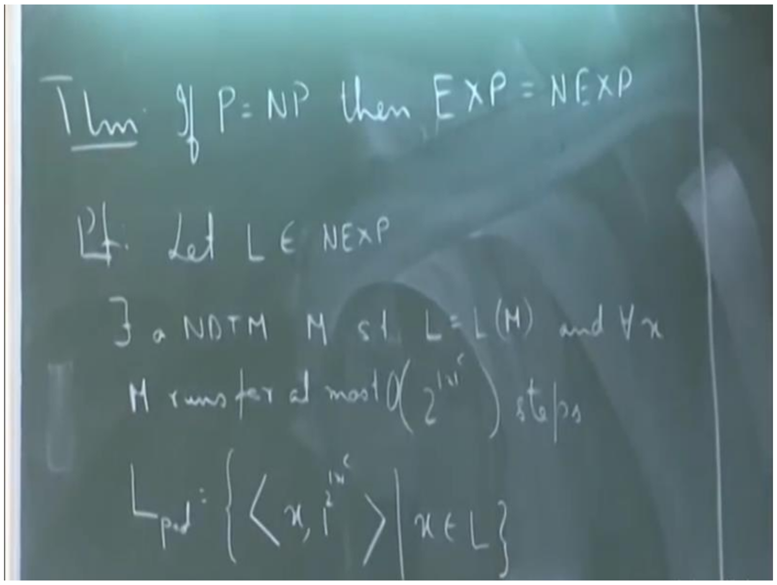
**Computational Complexity Theory**
**Prof. Raghunath Tewari**
**Department of Computer Science and Engineering**
**Indian Institute of Technology, Kanpur**

**Lecture -05**
**Introduction**

So let us get started. So last time we saw definitions of the complexity classes EXP, NEXP and so on. So today we will start by looking at a technique by which we can translate collapse of a class in some lower level, to collapse of classes in some higher level. So let me mention the theorem and then it will be clearer.

**(Refer Slide Time: 00:53)**



Thm: If P = NP then EXP = NEXP

Pf: Let $L \in$ NEXP

$\exists$ a NDTM M s.t. $L = L(M)$ and $\forall x$

M runs for at most $O(2^{|x|^c})$ steps

$L_{pad}: \{ \langle x, 1^{2^{|x|^c}} \rangle \mid x \in L \}$

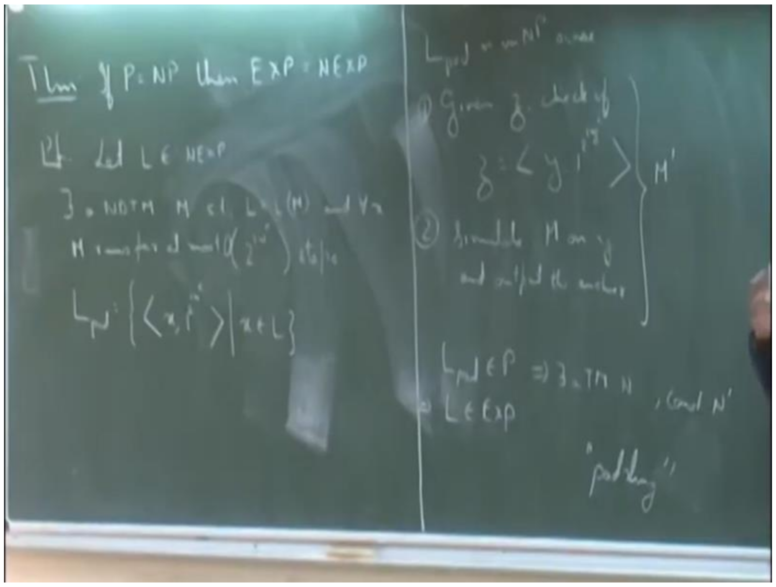
So if P=NP, then we can show that EXP=NEXP. So if I can show a collapse of these two classes in that is in the polynomial time setting I can show that even in the exponential time setting the analogous classes collapse. Or if you look at the contrapositive variant of this theorem if you can show that these two classes are not the same then that would imply that P and NP are not the same. So this is not very difficult to prove, let us just see how the proof goes.

So let L be a language in NEXP. So what does that mean? That means that there exist a non-deterministic turing machine M, such that L=LM and for all inputs x, M runs for at most 2 to the, x to the order of 2, to the x to the power c steps or some constant c, that is the definition. So what we will do is from L we will construct a new language which we will call L pad. So this language L pad is the language of all tuples of the form let us say a string x.

The string of all ones having length that is 1 to the power; 2 to the power size of x, to the power c. So you take the string x and concatenate it with 2 to the power, x to the power c 1 and which are the x that we consider for all x that belong to L. So what can we say about this language L pad? So we know that L belongs to NEXP. What can we say in more specifically in which class do you think this language will lie?

So we always count the running time as a function of the input size, so what are we doing here essentially. We are just adding some extra characters to the input. So characters that; we do not use but some exponential number of ones. So basically the reason why we do it is that, so that we can show that L pad will belong to NP. And that is not very difficult to see. So what we do is how do we construct an NP machine for L pad?

**(Refer Slide Time: 05:15)**



So L pad is in NP, since we can do the following. So first given let us say a string z. So we are just given a string z. Check if z is of the form some string y followed by 1 to the power, 2 to the power size of y, to the power c ones. So first check that the given string has this form or not? Scan through the input and do this and then if it has this form simulate the machine M on y. And output the answer.

So what is the running time of this machine the new machine that we are constructing? So first we are just scanning the input from left to right, that is if the input has length z that takes order z steps and then we are just simulating M on this string y. So we know that M runs for at most 2 to the power y to the power c steps. So the second step will also be order z. So that is why this entire computation it is a non-deterministic computation because M is non-deterministic.

But the good thing is that now it is linear in the size of the input the running time of whatever this machine. So we can give a name to this machine so let us say that this is some M prime. So now what can we say? We know that by our hypothesis P is equal to NP therefore L pad belongs to P as well. And from that we get that L belongs to EXP. Because if L pad belongs to P, take the deterministic polynomial time turing machine that decides L pad and just use that same machine to determine whether an input belongs to x or not.

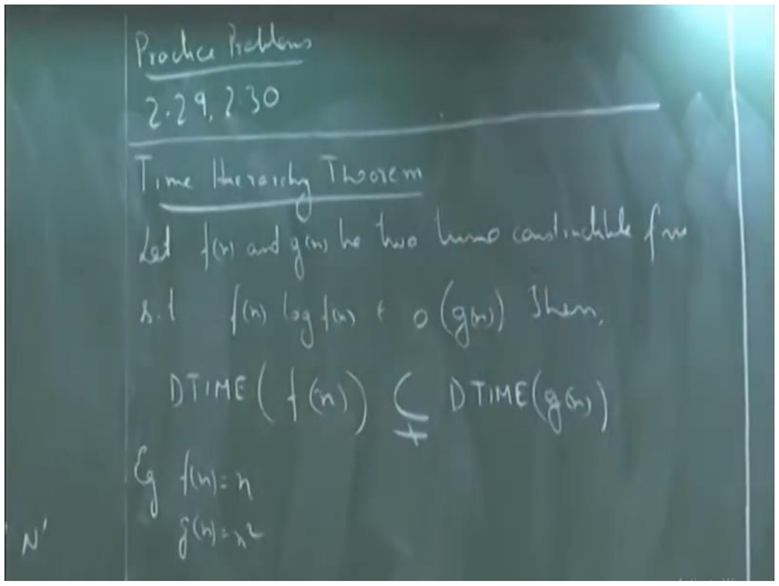
So that will still be exponential in the size of the input. Yes so I kind of brushed over it, but if you just think for it a while you will understand. So if L pad belongs to P, so that will happen, so what does it mean to say that L pad belongs to P? It implies that there exist some turing machine let us say N. A deterministic polynomial time turing machine N, which decides L pad what does it mean to say that N is polynomial? It is polynomial in the size of z.

So now what I do is that given this turing machine N, I just skip the first step. So I just take this turing machine N and from there I construct another turing machine N prime, which just skips the first step and takes an input for L and just does the rest whatever N prime was doing. So what I mean is that you just do not need to look at these bits any longer. You can just look at the y part and then output your answer.

So yeah so this technique is known as padding in complexity theory because essentially what we are doing is we are padding the given input by a kind of a redundant string of a certain length. I mean in certain cases where we are showing a result like this. That is we want to cascade I mean we want to translate a result which holds for some smaller class to some larger class. So one of the aim.

So, one of the goals of complexity theory is to prove separation between two complexity classes. So and this is not always possible I mean in fact we know very little about separation results but among the few that we do know.

**(Refer Slide Time: 11:04)**



Practice Problems

2.29, 2.30

Time Hierarchy Theorem

Let $f(n)$ and $g(n)$ be two time constructible fns s.t. $f(n) \log f(n) \in o(g(n))$ Then,

$$\mathrm{DTIME}(f(n)) \subsetneq \mathrm{DTIME}(g(n))$$

Eg $f(n) = n$

$g(n) = n^2$

N'

So we will discuss one today. And this is what is known as the time hierarchy theorem. And the technique that this theorem uses to prove that two complexity classes are separate is something that all of you actually have seen in your TOC course. Basically this uses the technique of diagonalization. So when you saw that the halting problem or some other problem is undecidable that is not computable using a turing machine you saw the technique of diagonalization.

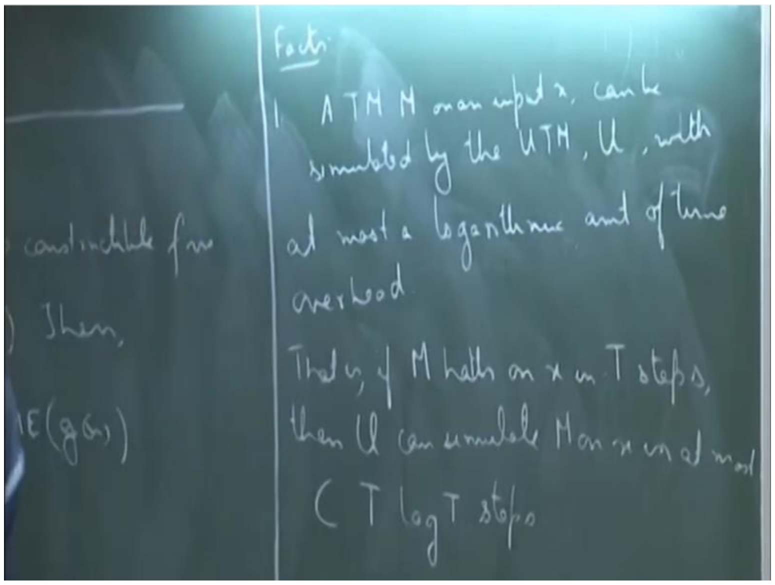
Where you thought of strings as machines and then you flipped the diagonal elements and got a language which was not decidable by any turing machine. So that is the technique that we also use here. So let me state the theorem first. So let f n and g n be two time constructible functions. Such that f of n times log of f of n is in small o of g of n. Then what we can show is that the classes the time classes that are determined by the functions f of n and g of n are separate basically.

That is DTIME f of n is strictly contained. So this is the notation for strict containment in DTIME g of n. So what this means is that there is a language in DTIME g of n that is not present in DTIME f of n. And this is true if we have time constructible functions which satisfy this criteria that is g of n is strictly larger, than f of n times log f of n. So just note that this is small o of g of n this is not big O.

So smaller means that it is strictly larger. For example, if you take can look at some small examples if you have a f of n= n and g of n= n 2. Then we can make this statement that there are languages in DTIME n 2 which are not present in D TIME n. But this again, I mean this hierarchy theorem is nice. I mean it does show that certain complexity classes are separate but again we cannot always use it.

For example, we cannot use it to show things like P and NP are separate. Because one thing is that they both are polynomial time machines. So it does not show separation between deterministic and non-deterministic versions of the same function. So before I begin the proof of this theorem let us look at some points.

**(Refer Slide Time: 15:51)**



Facts:

1. A TM M on an input x, can be simulated by the UTM, U, with at most a logarithmic amt of time overhead.
   That is, if M halts on x in T steps, then U can simulate M on x in at most C T log T steps

constructible fn

Then,

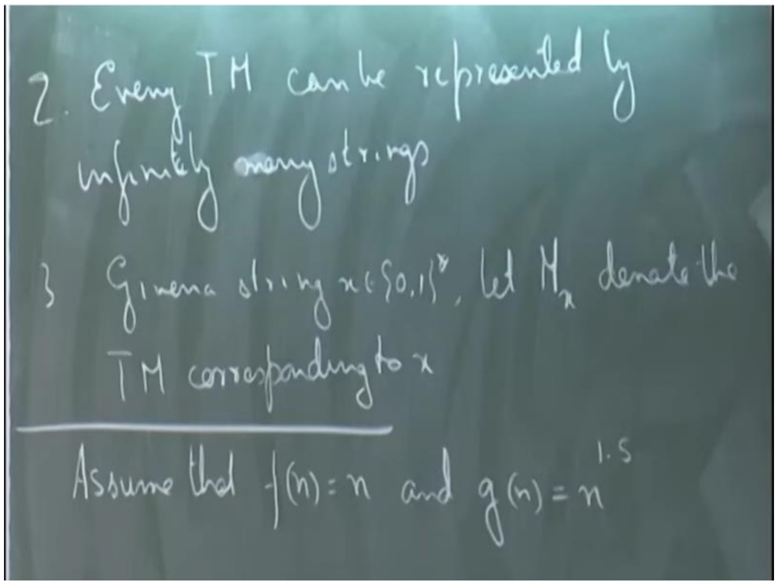
E(g(n))

So we know that again from our TOC course that there is a universal turing machine which is capable of simulating any turing machine. So it has it can take any turing machine and an input for that machine and it can simulate it on the universal turing machine. So we can say something little more than just bare simulation. We can say something about the time taken by the universal turing machine as well.

So a turing machine M on an input x can be simulated by the universal turing machine, let us say we will call it U with at most a logarithmic amount of time overhead. So let me make this more formal what we mean here. So that is if M halts on x in at most or say in in T steps. Then you can simulate M on x in at most some constant C times T times log T steps. So this is a constant which is dependent on the on the parameters of the turing machine M.

That is what is its alphabet size, how many states does it have and what is the nature of it is transition function which are all constants. I mean they do not depend on what is the length of the input. But apart from that if M can simulate x in T steps, then if I am using the universal turing machine to simulate M on x then that will take just an multiplicative factor of log T number of steps to do the same simulation.

**(Refer Slide Time: 19:31)**



2. Every TM can be represented by infinitely many strings

3. Given a string $x \in \{0,1\}^*$, let $M_x$ denote the TM corresponding to x

Assume that $f(n) = n$ and $g(n) = n^{1.5}$

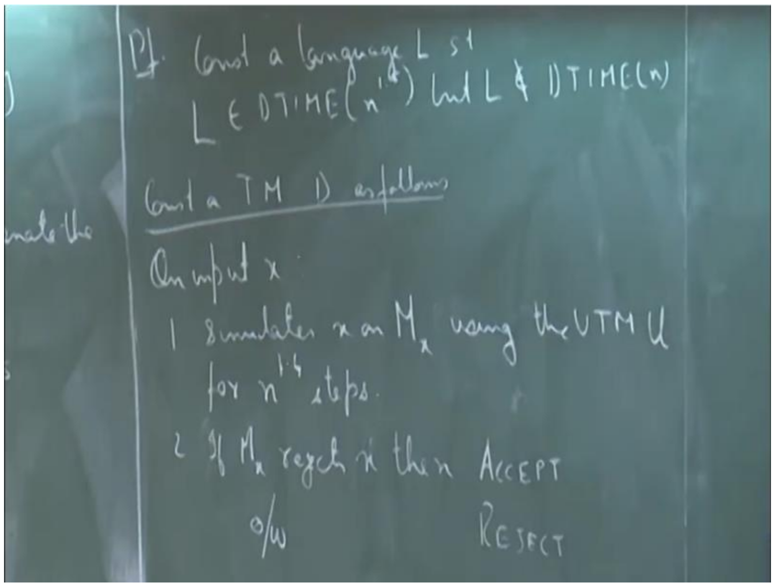
Every turing machine can be represented by infinitely many strings. Why is this true? So we always I mean we know that, we can fix an encoding of a binary string. So, that every string encodes a turing machine. But on the converse if I look at a turing machine then I claim that it can be represented by infinitely many strings. I mean that same encoding scheme satisfies this property as well.

So after you take a turing machine and encode it by some particular string you just pad it with some arbitrary number of ones. So basically the encoding that you now I mean the string that you now get will represent the same turing machine but just by ignoring the ones. I mean it is the same as looking at let us say program, so suppose if you have a C program it does not matter what that program.

I mean if you plug in comments into that c program you can plug any length I mean any number of comments of any length and that will not change what that program does. So there are infinitely many C programs which does the same thing basically by just plugging in arbitrary length comments. So that is the meaning of this statement. So and as a notation we will just use the following that so given a binary string x.

Let M of x denote the turing machine corresponding to x. So let us go ahead and prove the theorem. So actually again I mean what we will prove is not the general case for any arbitrary f of n and g of n. So we will basically prove the theorem for a particular choice of f of n and g of n. So we will assume that f of n is n and g of n is something like n to the power 1.5. And then will prove the theorem. And the general cases actually I mean will follow the same idea as we shall see.

**(Refer Slide Time: 23:27)**



Pf. Const a language L st
$L \in DTIME(n^{1.5})$ and $L \notin DTIME(n)$

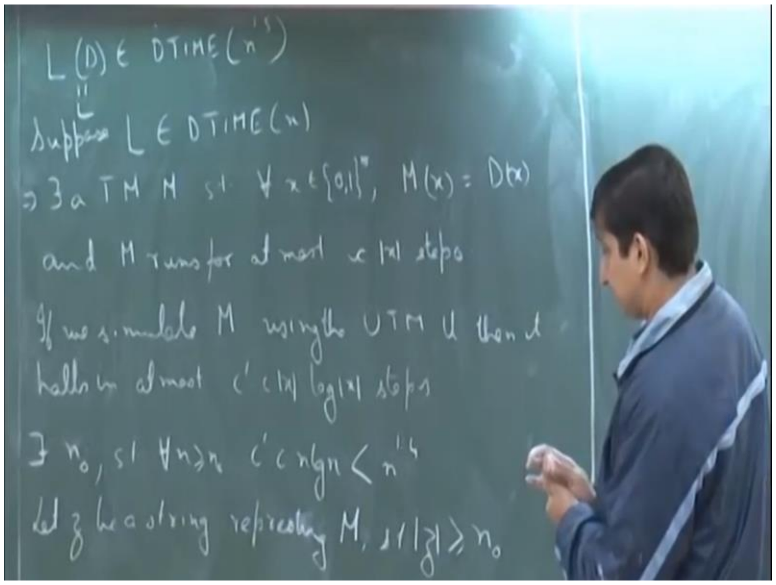
Const a TM D as follows

On input x :
1 Simulates x on $M_x$ using the UTM U
for $n^{1.4}$ steps.
2 If $M_x$ rejects x then ACCEPT
o/w REJECT

So we will construct a language L, such that L is in DTIME n of 1.5. But L does not belong to D TIME of n. So how do we define this language? So we will define a turing machine D. So will construct a turing machine D as follows. On input x what D does is the following. So it simulates x on M of x using the universal turing machine U for n to the power some 1.4 steps. I mean some number that is smaller than this.

And so the crucial point here is that it does not carry on this simulation for an arbitrary amount of time. Even if M x does not halt on x after n to the power 1.4 steps this machine D would halt and it will output the following. So if M of x rejects x, then go ahead and accept. Otherwise reject. So if within one n to the power 1.4 steps M of x rejects x, then D would accept and if either M of x accepts x or if M of x does not output anything on x within this many number of steps then you go ahead and reject in both the cases.

**(Refer Slide Time: 27:02)**



$L(D) \in \mathrm{DTIME}(n^{1.5})$
$\overset{\shortparallel}{L}$

Suppose $L \in \mathrm{DTIME}(n)$

$\Rightarrow \exists$ a TM $M$ s.t. $\forall x \in \{0,1\}^*$, $M(x) = D(x)$

and $M$ runs for at most $c|x|$ steps

If we simulate $M$ using the UTM $U$ then it halts in at most $c' c|x| \log|x|$ steps

$\exists n_0$, s.t. $\forall n \geq n_0$ $c' c n \log n < n^{1.4}$

Let $z$ be a string representing $M$, s.t. $|z| \geq n_0$

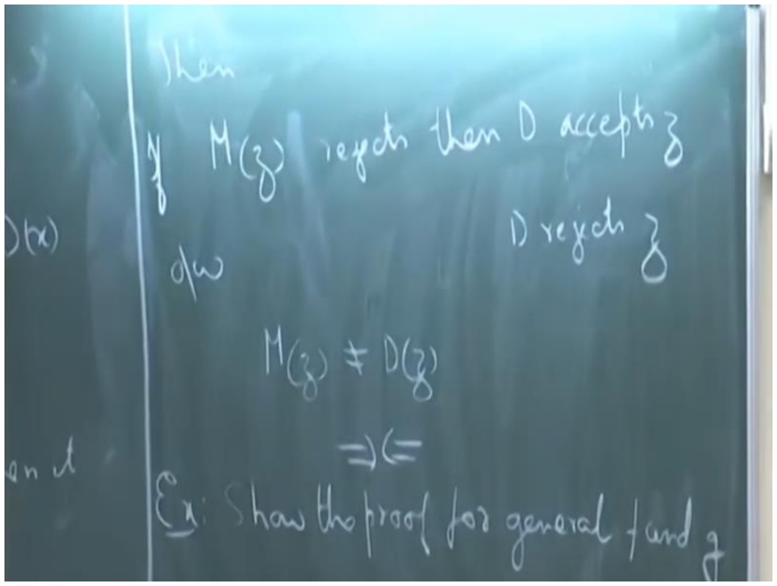
So what can we say about the language defined by D? So can we say this that L of D is in D TIME n to the power 1.5. So we can say this. Because by definition we are running this for at most this many number of steps. So therefore L of D belongs to this time class. Now suppose for and basically what we will show is that L of D does not belong to D TIME n. So for the sake of contradiction.

Suppose let me also in short just refer this to as L. Suppose L belongs to D TIME n as well. It means that there exist a turing machine M such that for all strings x M of x is equal to D of x and M runs for at most some constant c times the size of x number of steps. So, since we are assuming that L is in D TIME n. So now what we will do is we will simulate I mean since we were doing it for the case of D I mean there.

So we will simulate M using the universal turing machine U. So if we simulate M using the universal turing machine U, then it holds in at most some c prime times c of x times log of x number of steps. So this comes from this observation where now this c prime is basically this constant capital C that we had earlier. So what can we say about the asymptotic nature of these two functions.

So there exist some constant n o, such that for all n greater = n o, c prime c n log n is strictly smaller than n to the power 1.4. So this is the place where we use the fact that we have a function which is asymptotically larger than a logarithmic factor of the given function. So let z be a string representing the machine M such that z has length at least as large as n o. That is the cardinality of z satisfies this inequality. So why can again we assume that such a z would exist?

**(Refer Slide Time: 32:40)**



Then

if $M(z)$ rejects then D accepts $z$

o/w D rejects $z$

$M(z) \neq D(z)$

$\Rightarrow\Leftarrow$

Ex: Show the proof for general f and g

So now let us observe the following. So what is the output of this machine M of z. So what is happening to the output of M of z. So by definition so if you look at the definition of D that we have here. M of z is so, if M on z rejects, then D accepts z and otherwise D rejects z. Therefore for this particular z what we have that M of z and D of z they do not output the same answer by our construction.

And this is a contradiction because what we should have had is that this language I mean this machine should accept the same language that is for all strings x, M of x should have been equal to D of x. But now for this particular string that we have exhibited they have different output values. And now as an exercise you can show the proof for general f and g. I mean it is basically the same I mean instead of choosing these two particular functions if you take general f and g.

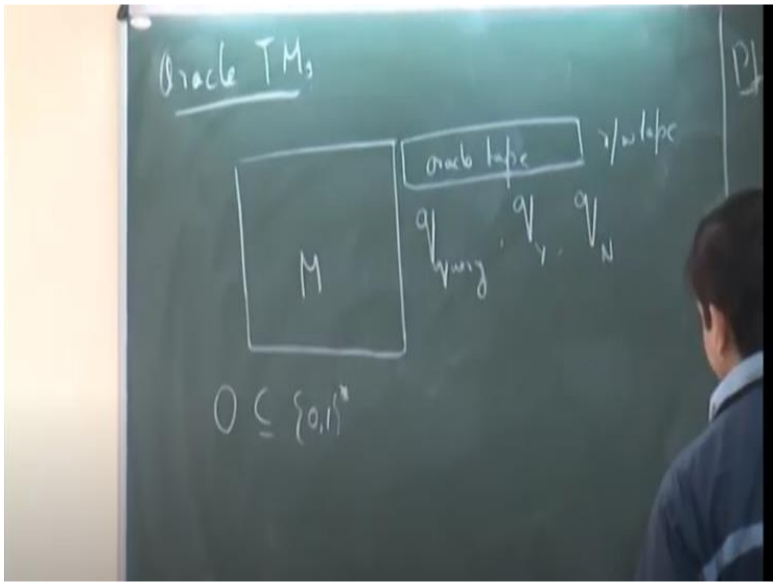
I mean the same kind of argument holds but you have to be a little bit careful. So let me ask a question at this point, so initially we assumed that the functions are time constructible exactly. So over here we are using that the function g of n is time constructible because so you are given this function g of n. Now you are given an input x, how can you run x on this machine M of x? For this many number of steps if you do not know what the value of that function is.

So, since you can compute that g of n in order g of n time which we implicitly are assuming here. So this is I have not stated here but we have the implicit understanding that we know what the value of this function here is because this is the function g of n. So given n we know what how to compute g of n and that is why we can do this particular simulation. This inequality might not hold.

So there can be a particular n for which c prime c n log n is greater than n to the power 1.4. So for that we do not even complete this thing. So it is only for those particular strings where we can complete this particular string this particular computation that we can claim this result. So this again I mean as you can see is using the technique of diagonalization. So but this is kind of different from.

Let us say the diagonalization argument where we showed the halting problem is not decidable. Because there we were there we had enumerated all particular machines. But here our enumeration is slightly different although we are enumerating an infinite number of machines, but we are only enumerating those machines which run for this much amount of steps. So that is the key observation.

**(Refer Slide Time: 37:24)**



So let us look at another concept. So the concept of oracle turing machines, so I will define this and I will state a result. But I would not prove it. I mean if you are interested proof is in your textbook but some important to understand what is happening. So what is an oracle turing machine? So an oracle turing machine is just like an ordinary turing machine. I mean it can be deterministic or non deterministic.

So we have a turing machine M but in addition to this it has an extra tape. So it has an extra tape which is referred to as the oracle tape. So this is a read write tape. So that is you can read from this tape as well as you can write on to this tape and the machine has three extra states. So it has a state q query, it has a state q let us call it yes and it has a state q No. So the idea is as follows. The idea is that what we want to do is that.

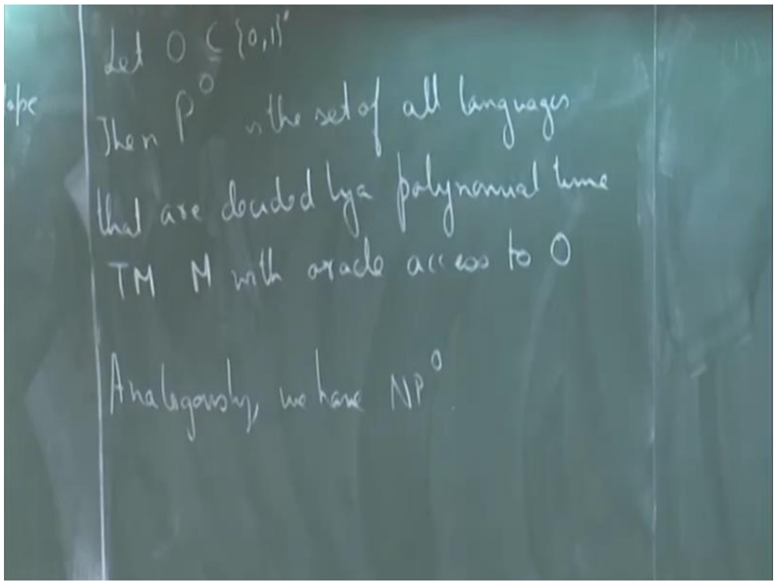
Now we want to look at turing machines which can query to some black box. So a black box is you can think of it as a closed box of some language that we do not know. And what this black box is capable of doing is if you provide this black box some string in one step it will tell you whether that string belongs to that corresponding language or not. So suppose if you have a black box for the language sat.

You provide the black box if boolean formula 5 and it will tell you whether 5 is satisfiable or not. So that is the idea that we want to formalize and represent it using turing machines. So the turing machine has an oracle tape and it has these three extra states. And there is some language this is the oracle subset of 0, 1 star, such that whenever the turing machine M enters into the state q query at any point during its computation.

In the next step the machine will go to either the state q y or the state q n depending on what is written on this oracle state. So the moment M enters q query it checks what is written on the oracle tape. If the string that is written on the oracle tape belongs to this language there is a fixed language O it will immediately go to q Yes. And if that string does not belong to the language it will go to q No.

And that is what and then it will proceed. And the good part is that, so that we just count that as one computation step. So maybe this O is a very complex language. I mean it has very high complexity, nevertheless we just think of going from this state q query to either the state q Y or q N as just one computation step. So that is the reason why we have this black box picture in mind. So now we can define a complexity classes based on this turing machine model.

**(Refer Slide Time: 41:13)**



Let $O \subseteq \{0,1\}^*$

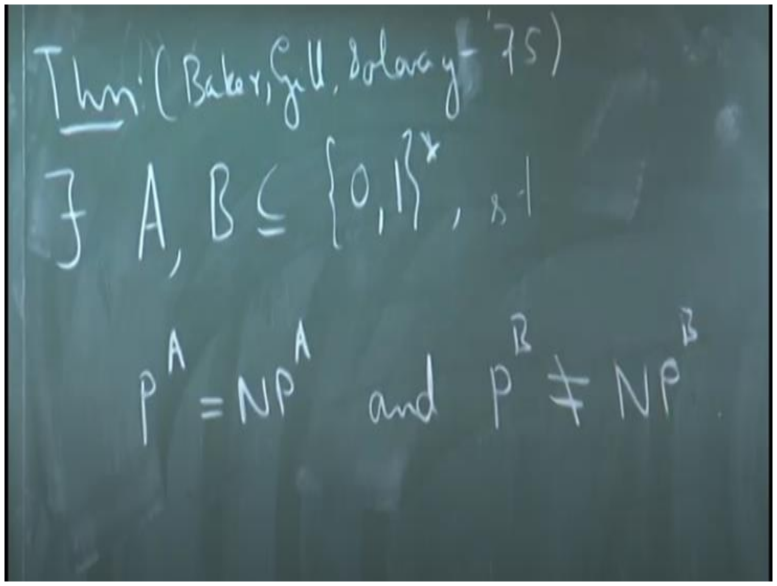
Then $P^O$ is the set of all languages that are decided by a polynomial time TM M with oracle access to O

Analogously, we have $NP^O$.

So let me just define this class P to the O. So let O be some language. Then the class P to the O is the set of all languages that are decided by polynomial time, time turing machine M with oracle access to O. So I formally did not define what it means to say that a turing machine has an oracle access to a language. But this is basically what the idea is. So I have a turing machine and it has oracle access to this particular language.

So then P to the power O is the set of all languages that can be decided by such turing machines. So analogously we have NP to the power O as well for any oracle O. And in fact this can be extended to any complexity class. If you have any complexity class you can think of an oracle for that complexity class and the class of languages that are defined by this notion. So now let us let me state the result that I mentioned earlier. So is this clear what this notion means?

**(Refer Slide Time: 43:37)**



Thm: (Baker, Gill, Solovay '75)

$\exists$ A, B $\subseteq \{0,1\}^*$, s.t.

$P^A = NP^A$ and $P^B \neq NP^B$.

So then what we can show is that the classes P and NP I mean for the classes P and NP there exist oracles with respect to which they are the same and their exist oracles with respect to which they are different. And this was a result that was shown in 1975 by Baker, Gill and Soloway. So kind of immediately after this theory of NP completeness picked up speed. So this was in 1975. So there exist languages A and B, such that P A = NP A.

And P B = NP B. So the first part of this theorem is not very difficult to show. So if you just think for a while you should get what A we should consider. So what do you think would be a suitable A? So the intuition is that A should be some very hard language. I mean in other words A should have complexity that is much larger than P and NP, so that it would not matter if we what the base classes are in some sense.

What choose A to be some language in EXP. So if you choose A to be some language in EXP we know that NP is a subset of EXP and P is a subset of EXP. So it really does not matter what the base machines are doing. If you give it any string well it has access to such a powerful oracle. So basically it can decide anything that is in EXP. So still I mean it still requires proof why these two would be the same.

But the idea essentially is that if you pick A to be let us say a language in EXP the equality holds. The more difficult part is to show this that what language do you pick as B such that these two classes are not the same. Because if you pick B to be something very hard, then they would be the same. If you pick B to be something very easy, then oracle does not help you much. So it has to be something of intermediate difficulty.

And this is where again the idea of diagonalization is used. So we construct a B again in an iterative manner using diagonalization. So we have a matrix and we flip the diagonal bits and we construct this language so that we get a suitable P. So this proof again if you are interested you can look at the book. It is not very difficult but it is good to know. So we will stop here.