Theory of Computation

Prof. Raghunath Tewari
Department of Computer Science and Engineering
Indian Institute of Technology, Kanpur

Lecture - 34
Reduction

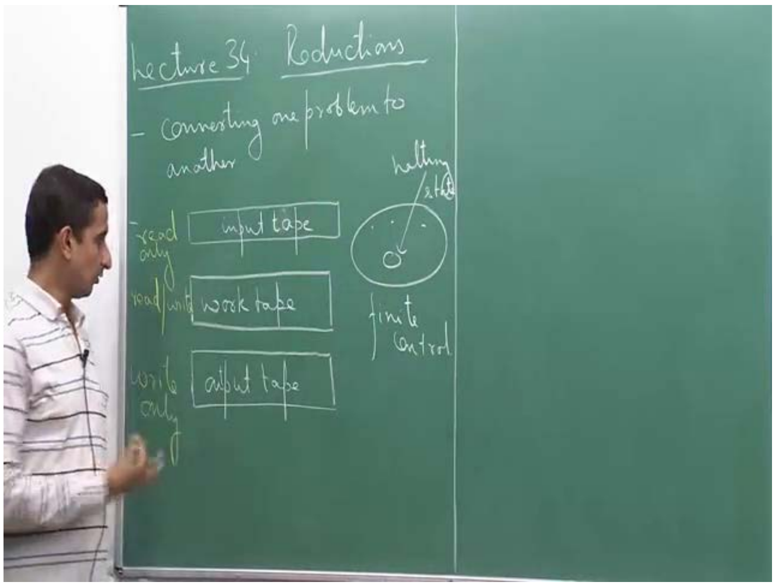
Welcome to the 34th lecture of this course. Today, we will talk about the concept of Reduction.

So, what is Reduction? Reduction in computer science means to convert one problem to another. In our day-to-day life and also in computer science, often we try to solve one problem by reducing it to another problem. For example, if I want to in computer science, if I want to sort an array, and let us say somebody gives me a black-box which is able to find the maximum element in an array, I can use that black-box and sort the array. So, how do I do it? I do it by repeated application of this black-box.

First, I feed my array to this black-box, it will give me the maximum element; I keep the maximum element out then I pass the reminder array that is the array left without the maximum element again to this black-box which will give me the maximum element of this new smaller array, and that will be my second maximum. And I keep on doing this until my array becomes empty. So, this will allow me to sort an array.

Also in for example, if I want to multiply two numbers, I can reduce it to the problem of addition. So, multiplying n times m essentially means that I want to add m to itself n times. So, if I can do addition, I can of course be able to do multiplication. So, there are various other examples where we somewhat we always where we use reduction without often observing that this is what we are actually doing. So, what will do today is we will formalize this concept, we will define precisely what we mean by reduction and then we will see its application with respect to showing that problems are un decidable.

(Refer Slide Time: 02:28)

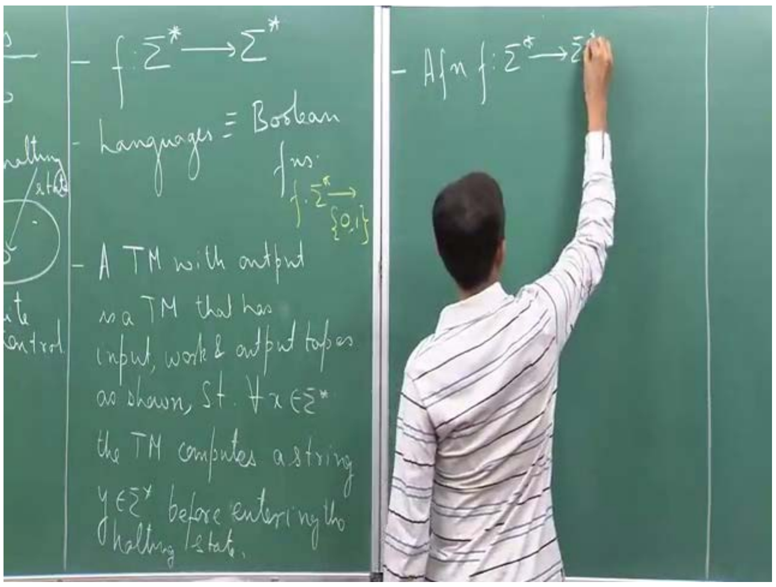


So, informally, as I say that reduction is converting one problem to another. Now for the purpose of reduction, so when we reduce one problem to another, we will use the Turing machine model; but we will slightly modify the model. So, our Turing machine model will now contain 3 tapes, so there will be input tape, there will be a work tape, and there will be output tape. Also in our finite control, so this is the finite control; in addition to all the states, we will assume that there is one state, which we call the halting state. So, this is the convention that we use.

So, what do these tapes signify? The input tape will signify the fact that it contains the input to the Turing machine. So, I can only read from the input tape, I cannot write anything to the input tape. And of course, I can scan both sides I can go back and forth and on the input tape, but I cannot write anything other than the input. The work tape is where I perform my work to come up with the solution. So, I can write on to the work tape, I can read from the work tape, I can erase the work tape, I can do everything.

And the output tape is a write only tape. The output tape has the property that I can only write bits to it, but I cannot read whatever I have written earlier. So, in some sense, it is a one directional tape. So, whenever I write a bit, I move one cell to the right and I keep on doing it I can never come back and read something. So, this input tape is as I said so the input tape is read only; the output tape is write only, and the work tape is of course, read write, I can do both read as well as write.

(Refer Slide Time: 05:51)



- $f: \Sigma^* \longrightarrow \Sigma^*$
- Languages ≡ Boolean fns: $f: \Sigma^* \longrightarrow \{0,1\}$
- A TM with output is a TM that has input, work & output tapes as shown, s.t. $\forall x \in \Sigma^*$ the TM computes a string $y \in \Sigma^*$ before entering the halting state.
- A fn $f: \Sigma^* \longrightarrow \Sigma^*$

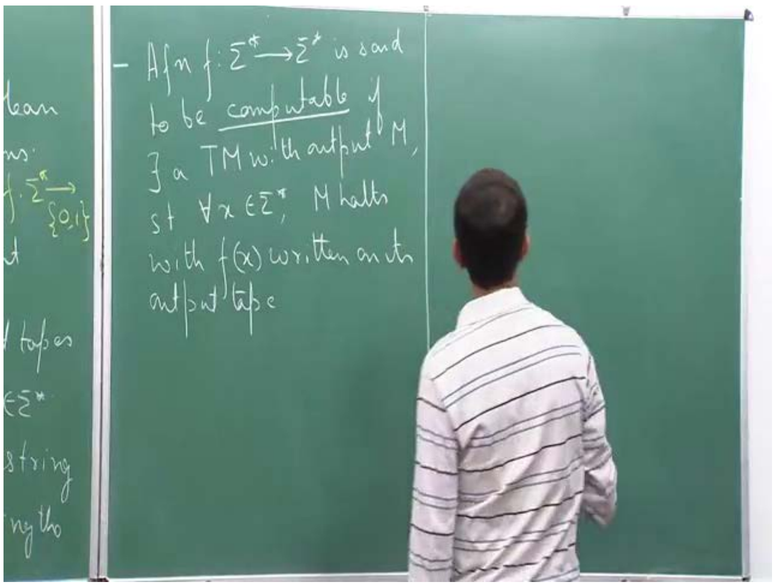
Now, what is the purpose of this halting state? So, what we want to do is that we want to compute functions or we want to output functions of this form. Let us look at a function, which takes a string as an input, it can take any string and it produces another string as an output. So, in the case of languages, one can think of languages as basically Boolean functions. So, functions which take a string as an input and produces 0 1, so that is a special type of function. So, I can think of languages as Boolean functions, because whenever I have a string that is in the language, I will map it to 1; and whenever I have a string there is outside the language, I will map it to 0.

So, languages are essentially equivalent to Boolean functions, so Boolean functions are those where the right hand side has. So, basically functions of the form f going from sigma star to 0 1. So, these are Boolean functions. Now I want to look at this general type of function, where actually I can write out a string, so often this necessary. For example, if I look at the sorting example, so in the case of sorting, I am given an array as an input, and I want to output another array which is the sorted array as output. So, I am not outputting a single bit, but I am actually outputting a string. So, it is essential that we have a way of computing such functions.

So, this Turing machine these particular types of Turing machine this generalization I should say is capable of doing it. So, what the Turing machine does is that it starts reading the input, it starts performing its computation, and during its computation it can write bits on to the output tape, it keeps on doing this.

And finally, when the machine enters the halting state, I stop; and whatever is written on the output state that is the output of the Turing machine. Let us call this a Turing machine with output, a Turing machine with output is a Turing machine that has input, work and output tapes as I showed. Such that for all x in sigma star, the Turing machine computes a string y belonging to sigma star and before entering the halting state; this y I will call as the output of the Turing machine.

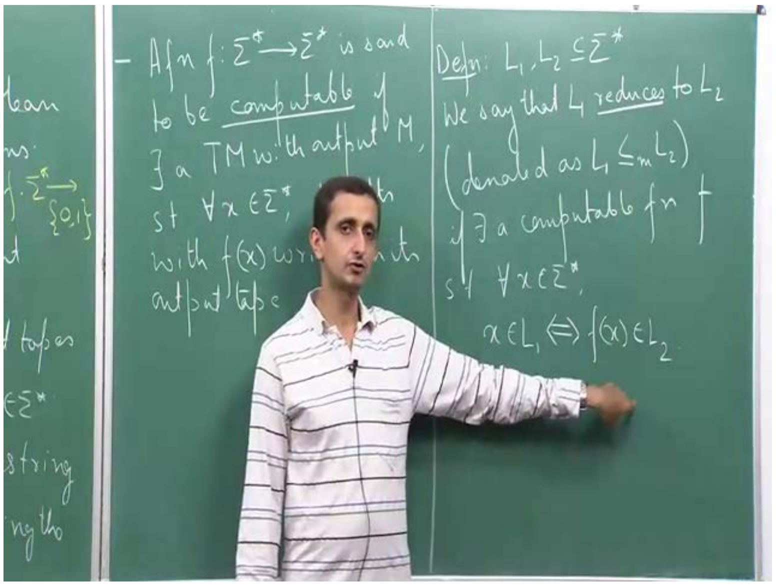
(Refer Slide Time: 10:12)



– A fn $f : \Sigma^* \longrightarrow \Sigma^*$ is said to be computable if $\exists$ a TM with output M, s.t $\forall x \in \Sigma^*$, M halts with $f(x)$ written on its output tape

Now with this, I define a computable function. So, a function f from sigma star to sigma star is said to be computable if there exist a Turing machine with output M, such that for all x in sigma star, M halts with f of x written on its output tape. So, this is when this happens we say that the function f is a computable function.

(Refer Slide Time: 11:50)



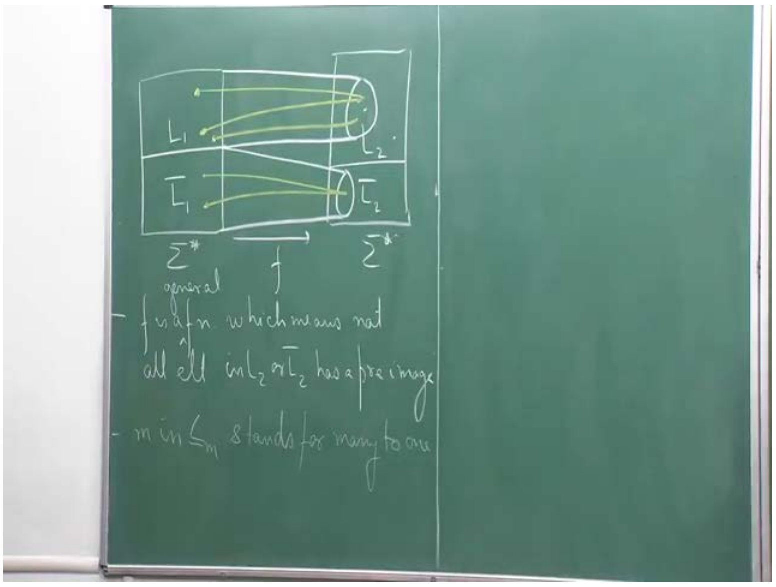
lean
ns.
$f : \Sigma^* \longrightarrow \{0,1\}$
t
tapes
$\in \Sigma^*$
string
ng the

- A fn $f : \Sigma^* \longrightarrow \Sigma^*$ is said to be computable if $\exists$ a TM with output M, s.t $\forall x \in \Sigma^*$, ... with $f(x)$ wri... on its output tape

Defn: $L_1, L_2 \subseteq \Sigma^*$
We say that L_1 reduces to L_2 (denoted as $L_1 \leq_m L_2$) if $\exists$ a computable fn f s.t $\forall x \in \Sigma^*$,
$x \in L_1 \iff f(x) \in L_2$.

Now we are in a position to define reduction. Let L 1, L 2 be two languages. We say that L 1 reduces, so reduces sign is basically, let me just say it we can write it. So, we say that L 1 reduces to L 2. So, this is denoted as L 1 with the less than or equal to sign L 2. If there exist a computable function f such that for all x belonging to sigma star x belongs to L 1, if and only if f of x belongs to L 2. So, a reduction is basically a computable function, which takes a string x and maps it to another string f of x such that it has the property that if x belongs to L 1 then f of x must belong to L 2. And if x does not belong to L 1 then f of x should not belong to L 2.

(Refer Slide Time: 13:42)



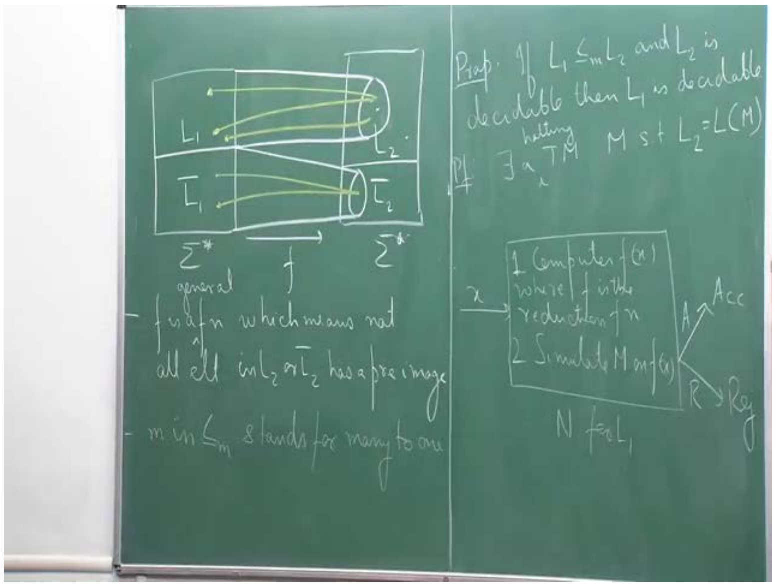
- f is a $\overset{\text{general}}{\hat{}}$ fn which means not all elt in L_2 or $\overline{L_2}$ has a pre image
- m in $\leq_m$ stands for many to one

So, pictorially, I can we can represent this as follows. So, suppose this is our universe of all strings again we have the universe. So, f is a function which goes from sigma star to sigma star. Such that so I have some subset of sigma star which is my language L 1. So, this is language L 1 and these are the strings which are not in L 1, hence they are in L 1 compliment.

Similarly, there is some subset, which is L 2 and L 2 compliment. So, we say that f is a reduction if L 1 is mapped to so let me put it here. So, if L 1 is mapped to some subset of L 2, the entire L 1; and the entire L 1 compliment is mapped to some subset of L 2 compliment. So, it is not necessary that every element in L 2 or every element in L 2 compliment must have a pre image. So, there can be elements outside also, there can be elements inside also. So, this is what a thing represents.

So, f is a function which means not all element f is a general function, I should say which means not all element in L 2 or L 2 compliment has a pre image. And we now in our notation, we had this small m as a subscript in our notation for reduction, so that m stands for many to one, so the m in stands for many to one. It essentially means that there can be many strings let us say in L 1, which is getting map to the same string in L 2, for example, I can have string here and here. So, maybe both of these get mapped to this guy, same thing can happen here also. So, it is a many to one function.

(Refer Slide Time: 17:18)

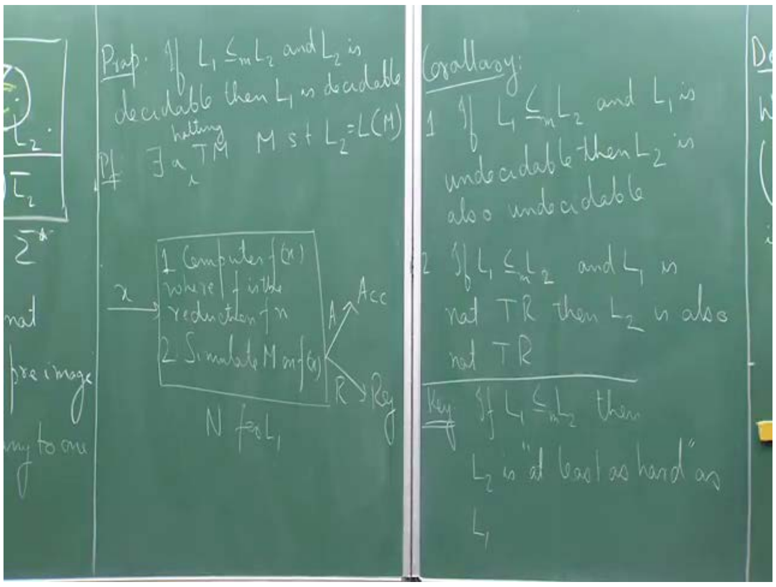

L_1
$\overline{L_1}$
L_2
$\overline{L_2}$
Σ^*
general
$\xrightarrow{f}$
Σ^*
- f is a fn which means not all elt in L_2 or $\overline{L_2}$ has a pre image
- m in $\leq_m$ stands for many to one
Prop. If $L_1 \leq_m L_2$ and L_2 is decidable then L_1 is decidable
Pf $\exists$ a halting TM M s.t $L_2 = L(M)$
x
1 Computes f(x) where f is the reduction fn
2 Simulate M on f(x)
A
Acc
R
Rej
N for L_1

Now, why are reductions so interesting? The reason is the following proposition. Let L 1 reduce to L 2, and put it if, so if L 1 reduces to L 2, and L 2 is decidable, then L 1 is also decidable. So, if I am able to reduce a language L 1 to a language L 2, and I know that L 2 is decidable, then it means that L 1 is also decidable. So, why is this, so the proof is not difficult to see? So, suppose I want to prove that L 1 is decidable. So, if I want to show L 1 is decidable, I have to construct a Turing machine for L 1. I know that L 2 is decidable. So, say L 2 is so since L 2 is decidable there exist a Turing machine M such that L 2 equals L of M, and not only a Turing machine it is a halting Turing machine.

Now, we will construct a Turing machine for L 1. So, what we will do in our Turing machine. Let us say I want to construct a Turing machine N for L 1. So, what the Turing machine N does is given an input x, it first computes f of x. So, why can we compute f of x, because f is a reduction, so where f is the reduction function, so this is step 1. In step 2, you simulate M on f of x. If M accepts, so if M accepts then you accept; and if M rejects, then you reject. Let us prove why this is correct. So, take any x, suppose if x belongs to L 1, then after I compute f of x, because I know that f is a reduction, since x belongs to L 1, f of x must belong to L 2.

Now when I simulate M on f of x, since f of x belongs to L 2, M accepts, hence I would accept. So, if x belongs to L 1, I accept; similarly if x belongs to L 1 compliment, then here I my f of x will belong to L 2 compliment. Hence when I simulate M on f of x it rejects, hence I finally, reject. So, it is a just a simple box, which is doing two things; it is first computing f of x and then it is simulating M on f of x, so that is the argument.

(Refer Slide Time: 21:27)



Prop: If $L_1 \leq_m L_2$ and L_2 is decidable then L_1 is decidable

Pf $\exists$ a halting TM M s t $L_2 = L(M)$

1 Computes f(x) where f is the reduction fn

2 Simulate M on f(x)

x, A, Acc, R, Rej

N for L_1

Corollary:

1 If $L_1 \leq_m L_2$ and L_1 is undecidable then L_2 is also undecidable

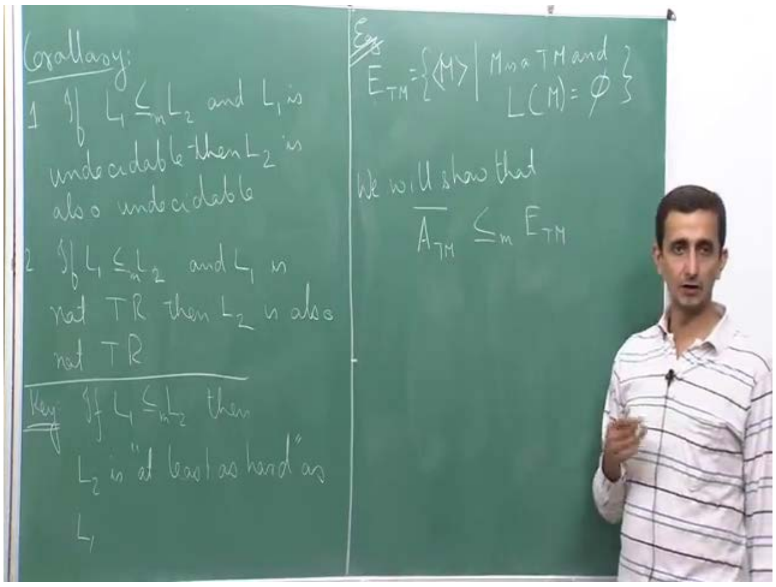
2 If $L_1 \leq_m L_2$ and L_1 is not TR then L_2 is also not TR

Key If $L_1 \leq_m L_2$ then L_2 is "at least as hard" as L_1

Now this has some interesting applications. So, as a corollary, we get two interesting I mean we can say two things. So, first if L 1 reduces to L 2, and L 1 is undecidable then L 2 is also undecidable. Now, this gives me a mechanism to prove that a language is undecidable. So, if I want to show that a language is undecidable, all I have to do is that to reduce an undecidable language to my given language. If I can do that then I would have succeeded in showing that my given language is also undecidable.

And similar technique I mean technique similar to what I have described here can be used to shown that if L 1 reduces to L 2; and L 1 is not Turing recognizable, then L 2 is also not Turing recognizable. The idea is the same. You can actually figure this out yourself. The key point here is that, so the key observation is that, so, this is what you should always remember that if L 1 reduces to L 2, intuitively what this means is that then L 2 is at least as hard as L 1. The language L 2 is either harder than L 1 or it is equivalent in hardness to L 1, it cannot be easier than L 1, so that is intuitively what reduction means. So, this is what you should keep in mind.

(Refer Slide Time: 24:36)



Corallary:

1 If $L_1 \leq_m L_2$ and L_1 is undecidable then L_2 is also undecidable

2 If $L_1 \leq_m L_2$ and L_1 is not TR then L_2 is also not TR

Key: If $L_1 \leq_m L_2$ then L_2 is "at least as hard" as L_1

~~Eg~~

$E_{TM} = \{\langle M \rangle \mid M \text{ is a TM and } L(M) = \phi\}$

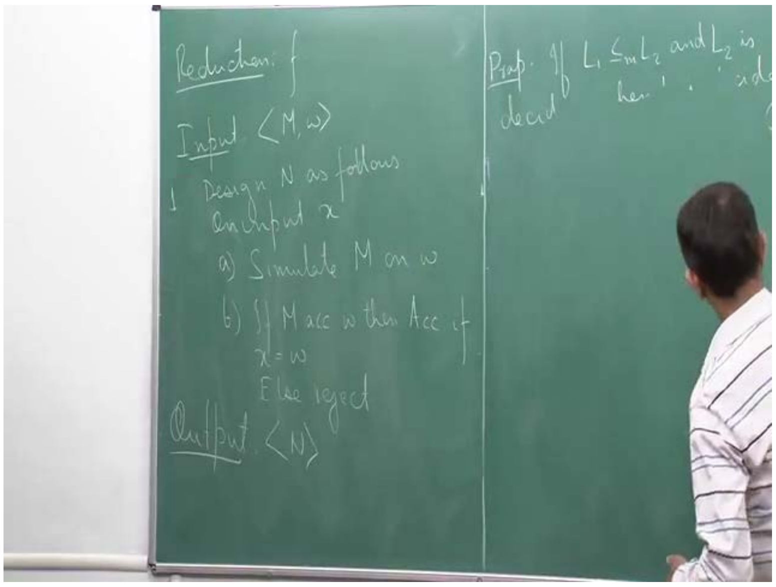
We will show that

$\overline{A_{TM}} \leq_m E_{TM}$

Now, let us look at an application of this result. So, consider the language E T M. So, this was an example. So, E T M is encodings of Turing machine M such that M is a Turing machine and the language of M is empty. So, E stands for empty, so this language we will prove is will prove that this is undecidable.

And the way we will show this is undecidable is we will use this corollary here. So, we will take an undecidable problem. The problem that we will take is A T M bar. So, we will show that A T M compliment reduces to E T M. So, not only we will this prove that E T M is undecidable, it will also prove that E T M is not Turing recognized, because A T M bar is also we saw this last time that A T M bar is not Turing recognized. So, what is the idea? Let me give the reduction. The reduction is nothing but giving a computable function.

(Refer Slide Time: 26:05)


Reduction: f

Input: $\langle M, w \rangle$

1 Design N as follows
   On input x
   a) Simulate M on w
   b) If M acc w then Acc if
      $x = w$
      Else reject

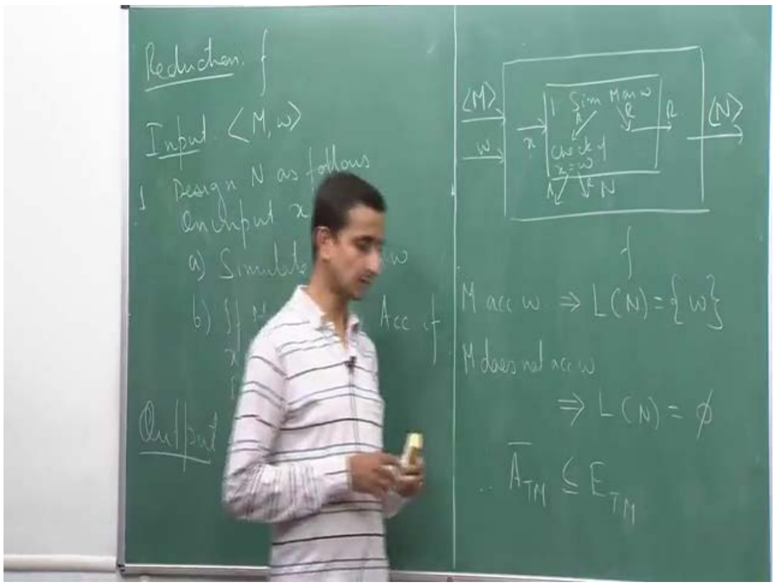
Output: $\langle N \rangle$

Prop. If $L_1 \le_m L_2$ and L_2 is

So, I want to design a function f that takes an instance of A T M bar that takes an input which is from the universe of A T M bar, and it produces an instance of E T M. Such that if so instance of A T M bar will be instances of the form M comma w, an instance of E T M will be just a simple machine M. So, I will take as input encoding of a machine, let us say M comma w, and I will produce as output let us say I will give this another name N such that if M does not accept w then this language is empty the language of N. And if M accepts w, then the language of N is not empty. So, that is what I am going to do. So, how do we design the machine? So, what we do at the beginning is design N as follows.

So, here is what N will do. So, on an input x, so x is the input to N. So, I have to define the Turing machine N. So, I have to define that what this Turing machine is going to do on every input, for example, if I am given an input x, what should the behavior of empty. So, what N will do is that first N ignores x. So, N would simulate M on w, if M accepts w, then accept if x is equal to w else reject.

(Refer Slide Time: 29:01)



Let us look at representation. So, I want to, so this is my reduction f that I want to design; inside this, I am designing this machine N. So, what N does is that. So, f takes as input two things; it takes an M and it takes a w, and N is taking some x as input. So, what it does is that first it will simulate M on w, if this accepts, and if this rejects, we do two things. So, if this rejects then N will reject; and if M accepts then check if so check if x equals w; if this accepts then accept else reject. Now, let us try to understand and finally, f will output the machine N.

Let us try to understand what is happening. So, suppose if M accepts w, so if M accepts w, observe that here I would accept w then the machine N going to check if x is equal to w; if x is not w, it is anyway going to reject. Which strings x will N end up accepting. So, N will accept only the string w. So, then the language of N is going to consist of only the string w that is the only string that it will accept. On the other hand if M does not accept w, then so if M does not accept w if M is rejecting or if M is going into an infinite loop, I will directly anyway reject or go into an infinite loop in other words it will not accept that x. So, in other words the language of N will be the empty set.

Now, I have shown that therefore A T M bar reduces to E T M; because if M does not accept w, then I have outputting a machine N whose language is empty as it is the definition of E T M. And if M accepts w, I am outputting a Turing machine whose language is non empty, so that is a no instance. So, here I am outputting yes instance, here I am outputting a no instance. Therefore, I have this reduction, which proves that E T M is undecidable.

So, I will stop here. Next time, we will look at more examples.