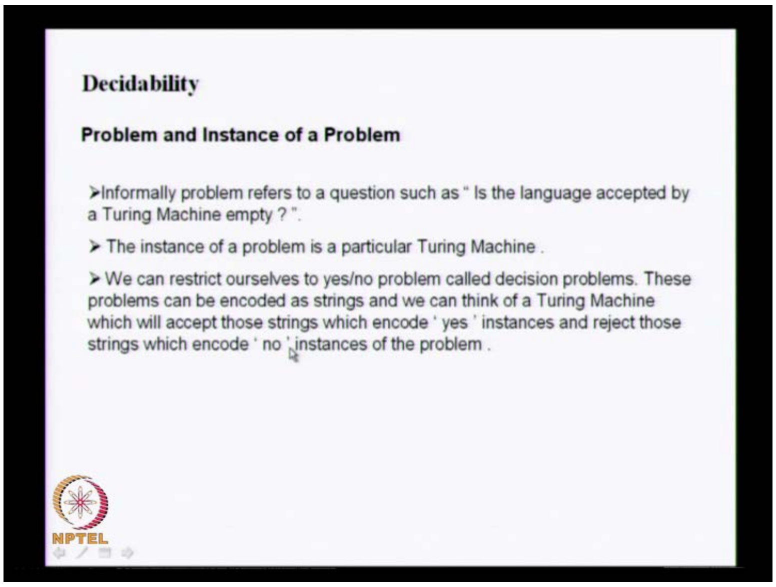
Theory of Computation

Prof. Kamala Krithivasan
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Lecture No. # 32
Problems and Instances, Universal TM, Decidability

In the last lecture, we saw about recursive sets, recursively enumerable sets and a few properties of them. And we also saw what is meant by a halting problem of a Turing machine, and how it is undecidable.

(Refer Slide Time: 00:34)



Decidability

Problem and Instance of a Problem

➤Informally problem refers to a question such as " Is the language accepted by a Turing Machine empty ? ".

➤ The instance of a problem is a particular Turing Machine .

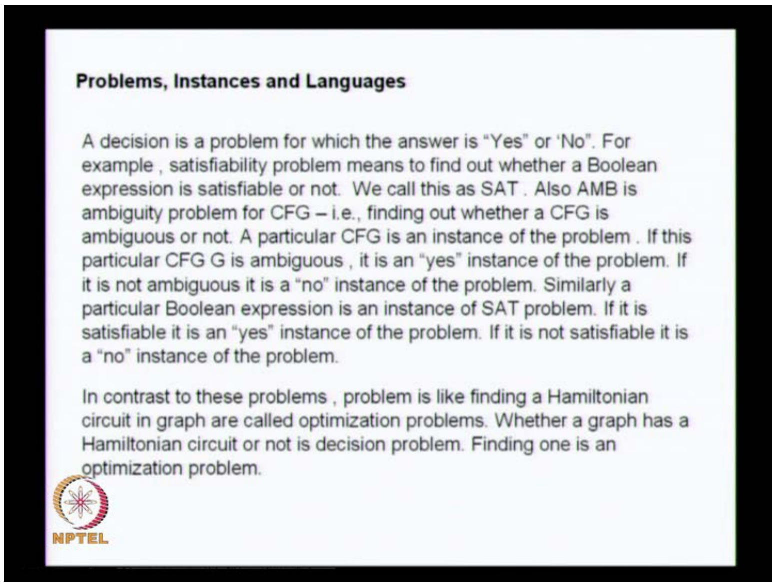
➤ We can restrict ourselves to yes/no problem called decision problems. These problems can be encoded as strings and we can think of a Turing Machine which will accept those strings which encode ' yes ' instances and reject those strings which encode ' no ' instances of the problem .

Let us consider some more results and facts about recursive sets recursively enumerable sets and also about decidability. Now, before going into decidability, what do you mean by decidability? Or when do you say that a problem is decidable? For this, you must have a clear understanding of what is a problem and what is an instance of the problem. Informally, if you say a problem refers to a question such as is the language accepted by a Turing machine; empty.

That is the problem or in other words, something like does a graph have a Hamiltonian circuit. And instance of the problem will be here, in this case, a Turing machine, when you consider a Hamiltonian graph problem, the instance is a graph. Now, we shall restrict ourselves to the problems where the answer is only yes or no. We call them as yes or no problem or decision problems and the problems can be encoded as strings. And we can look at the Turing machine which will accept all the yes instances of the problem.

That is the problem itself, it is encoded as a string and all those strings which correspond to yes instances of a problem form a language, and we can think of a Turing machine accepting that language. And the Turing machine will reject the no instances of the problem.

(Refer Slide Time: 02:20)



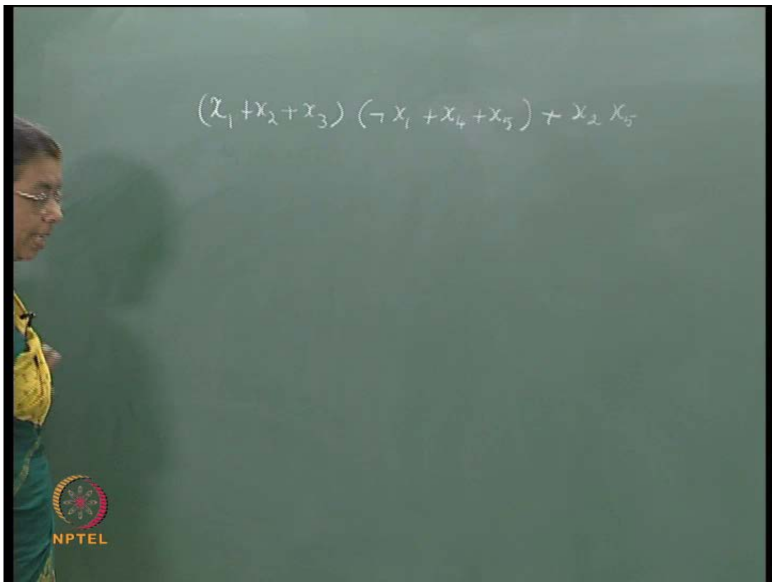
Problems, Instances and Languages

A decision is a problem for which the answer is "Yes" or 'No". For example , satisfiability problem means to find out whether a Boolean expression is satisfiable or not. We call this as SAT . Also AMB is ambiguity problem for CFG – i.e., finding out whether a CFG is ambiguous or not. A particular CFG is an instance of the problem . If this particular CFG G is ambiguous , it is an "yes" instance of the problem. If it is not ambiguous it is a "no" instance of the problem. Similarly a particular Boolean expression is an instance of SAT problem. If it is satisfiable it is an "yes" instance of the problem. If it is not satisfiable it is a "no" instance of the problem.

In contrast to these problems , problem is like finding a Hamiltonian circuit in graph are called optimization problems. Whether a graph has a Hamiltonian circuit or not is decision problem. Finding one is an optimization problem.

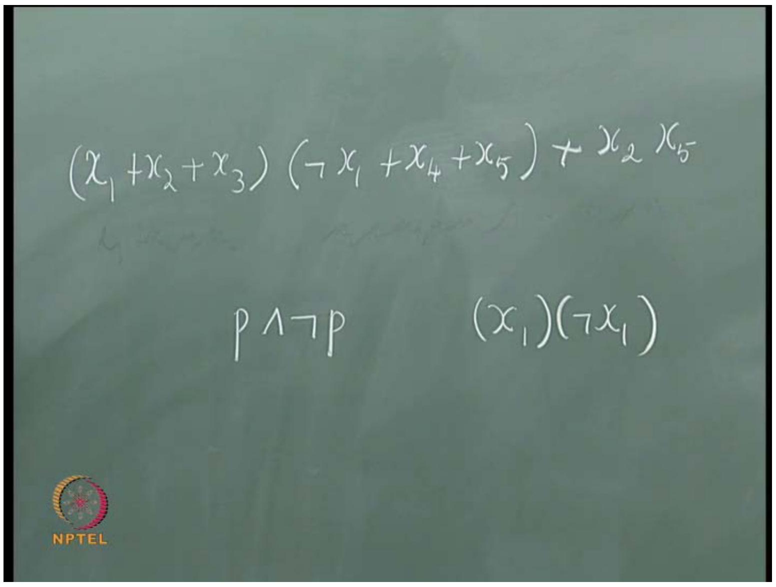
So, we will restrict ourselves to the decision problems. Of course, there is a connection between decision problems and the other problems we will see that in a moment. So, the decision problems the answer is yes or no. If you can design an algorithm, which will come out with an answer yes or no then the problem is decidable. If you cannot have an algorithm the problem is undecidable. For example, if you look at the Boolean satisfiability problem, what is that? It is to find out whether a Boolean expression is satisfiable or not. What is a Boolean expression? The Boolean expression is something like this.

(Refer Slide Time: 03:15)



So, you can have an expression take x 1 plus x 2 plus x 3 and not x 1 plus x 4 plus x 5 plus x 2 x 5 something like this. This is a Boolean expression, is it satisfiable or not? That is, does there existence assignments to the variables x 1 x 2 etcetera 0 or a 1 0 standing for false and 1 standing for true. Thus there exist in assignment which will make the expression evaluate to 1 or true. So, if it evaluates to 1, some assignment makes the expression evaluate to 1, then you say that instance has a solution. So, that is a Boolean that expression is satisfiable; that particular instance is satisfiable. There are instances which are not satisfiable.

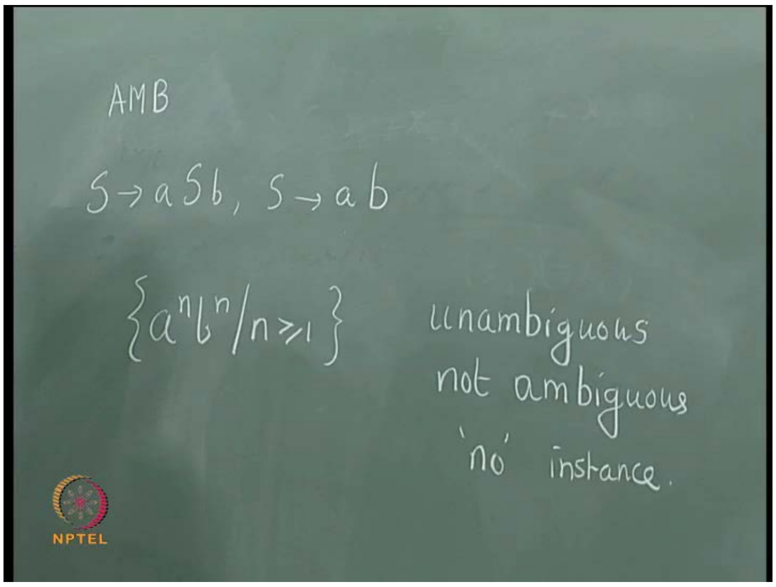
(Refer Slide Time: 04:32)



For example, something like you know p n naught p are this is in the proper (( )) form, x 1 naught x 1. That is x 1 and naught x 1 that will always be 0. So, there are expressions which are never satisfied, which will never take the value 1. So, the problem is given in expression of this form can one assignment make it equal to 1? Similarly, you have the ambiguity problem for context free grammars. What is that?

The ambiguity problem is finding out whether a context free grammar is ambiguous or not. Particular context free grammar is an instance of the problem. If this particular context free grammar is ambiguous, then that is a yes instance of the problem. If it is not ambiguous, then it is a no instance of the problem.

(Refer Slide Time: 05:55)

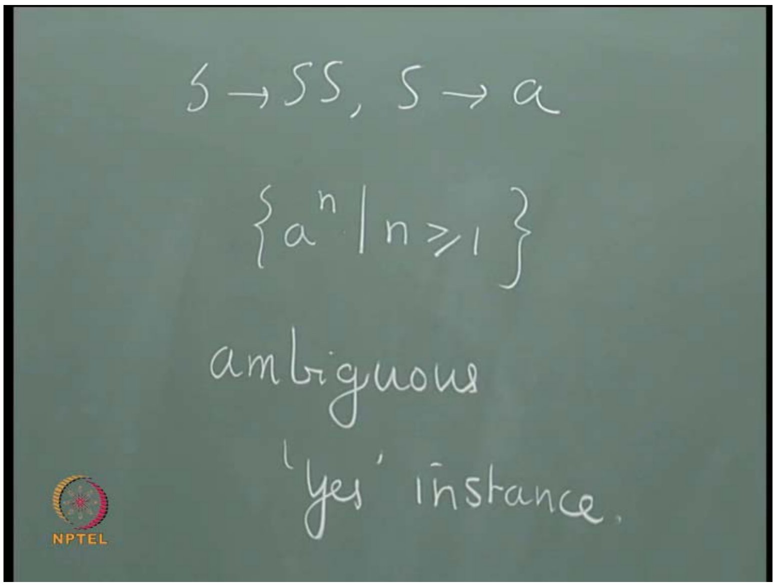

AMB

$S \to aSb,\ S \to ab$

$\{a^n b^n / n \geq 1\}$ unambiguous not ambiguous 'no' instance.

Now, let us take a grammar S goes to a S b, S goes to a b. I am only writing the rules S is the non terminals, small a and small b are terminal symbols. Now, this particular grammar, you know is unambiguous. Any string will have only one derivation tree. The language generated is a power n, b power n; n greater than or equal to 1. And if you take any string a power i, b power i, that will have only one derivation tree or one pass tree. So, this particular grammar is unambiguous or you can say it is not ambiguous. It is a no instance of the problem.

(Refer Slide Time: 07:07)



$S \to SS, \; S \to a$

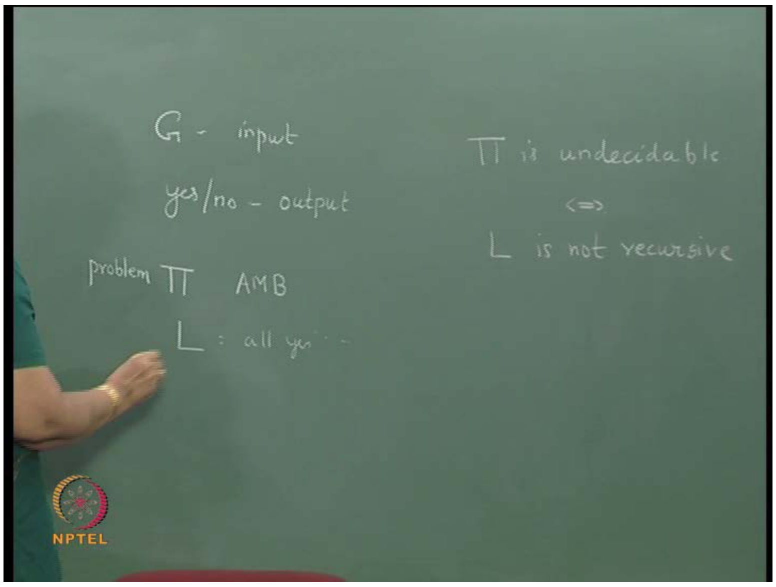
$\{a^n \mid n \geqslant 1\}$

ambiguous

'yes' instance.

Whereas, look at this grammar, S goes to S S, S goes to a. Here again, S is a non terminal and small a is the terminal symbol. These are the two rules. The language generated is a power n; n greater than or equal to 1. We know that this grammar is an ambiguous grammar so, this is ambiguous or it is a yes instance of the problem.

(Refer Slide Time: 08:20)



G - input

yes/no - output

problem Π AMB

L : all yes

Π is undecidable

$\Leftrightarrow$

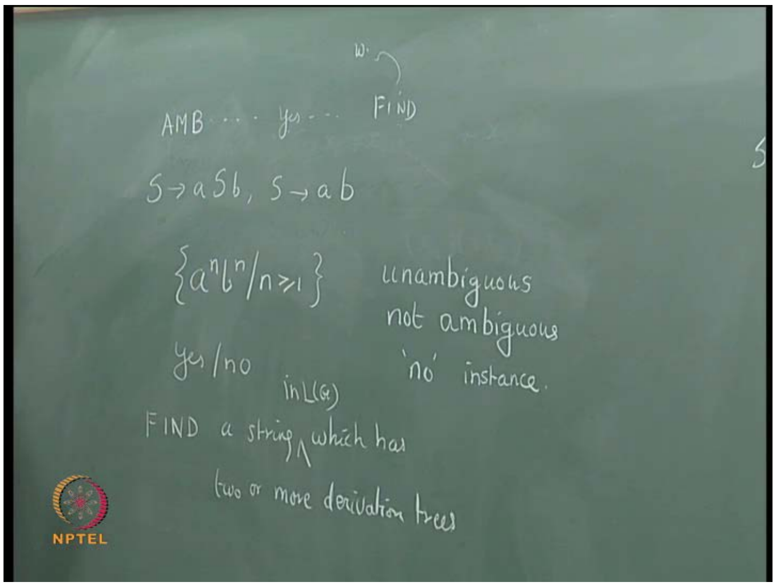
L is not recursive

Now, I know that this particular grammar is ambiguous because I am able to get two derivation tree for a cubed or something like that, you know, that this grammar is unambiguous. These are particular instances of the problem. For particular instances, you may be able to find out the solution. When you say that the problem is un decidable, what it means is, in general, there is no algorithm which will take a context free grammar G as input and come out with the answer yes.

It is ambiguous or no it is unambiguous. There is no general algorithm which will take any grammar G. G may be having 50 rules, it may be having 100 non terminals or it may be having 100 rules, 150 non terminals and so on and algorithm should take any such grammar and come out with an answer yes or no. What we want to say is, there is no such algorithm, that is, the problem is undesirable corresponding to a problem. Suppose, we denote the problem by a symbol pi, this is the ambiguity problem. There I will be corresponding language l which consists of all yes instances of the problem right.

Now, when you say that pi is undesirable, it would mean l is not recursive and vice versa. The problem pi is the problem. It is un decidable that is equivalent to saying l is not recursive. What is l? l consists of all yes instances of the problem encoded as a string, encoded as strings. Now, how is this encoding done, that we have to see. Under what condition, this one is not recursively enumerable? Under what condition it is recursively enumerable, not recursive? They are all points to be study.

(Refer Slide Time: 11:55)



w.

AMB . . . . yes . . . FIND

$S \to aSb,\ S \to ab$

$\{a^n b^n / n \geq 1\}$ unambiguous

not ambiguous

yes / no 'no' instance.

FIND a string in L(G) which has

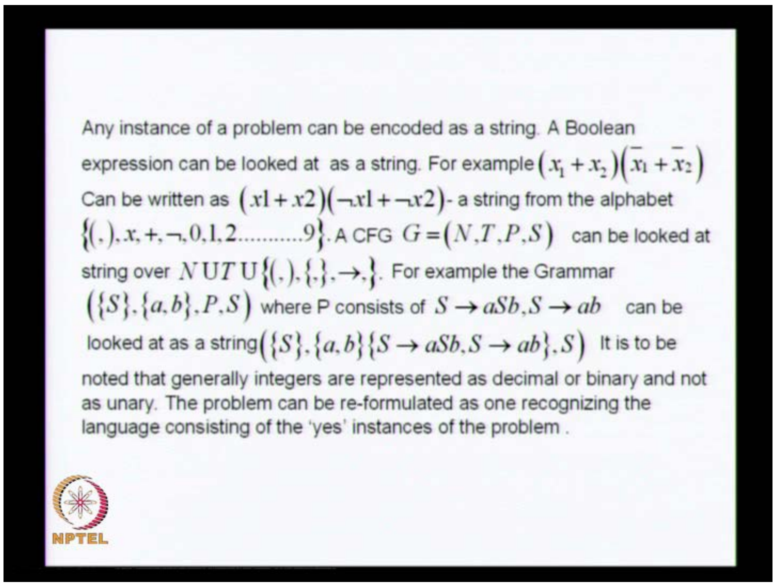
two or more derivation trees

So, we show that we see that for the sat problem or Boolean expression satisfiability, if it is satisfiable, that is, if there is an assignment, which makes the expression evaluative to 1, it is a yes instance of the problem. If there is no such assignment, it is a no instance of the problem. In contrast to these problems, the problems like finding a Hamiltonian circuit, they are called optimization problems. Whether a graph has a Hamiltonian circuit, it is a yes or no problem decision problem.

Whereas, find the Hamiltonian circuit is a optimization problems. In fact more or less we can convert one to the other and look at in a similar manner. For example, here the ambiguity problem is given a grammar. Is it ambiguous or not? That is the answer is yes or no. You can just look at it as an optimization problem. In this way, find a string which has got two or more derivation tree in, find a string which has two or more derivation trees, find a string of course, string is in l G.

Now, how can you relate one to the other? Suppose, the ambiguity problem is decidable or if the answer is yes, then in the find one by one, you can, the strings and find the pass trees and find out the string which has got two or more pass trees or derivation trees. The other way round, if the find gives you an answer, one particular string w. That means that particular string w has two or more derivation tree. So, the answer to the ambiguity problem is here. That way you can connect to one decision problem to one optimization problem.

(Refer Slide Time: 13:44)


Any instance of a problem can be encoded as a string. A Boolean expression can be looked at as a string. For example $(x_1 + x_2)(\bar{x}_1 + \bar{x}_2)$ Can be written as $(x1 + x2)(\neg x1 + \neg x2)$- a string from the alphabet $\{(,), x, +, \neg, 0, 1, 2 \ldots\ldots\ldots 9\}$. A CFG $G = (N, T, P, S)$ can be looked at string over $N \cup T \cup \{(,), \{,\}, \rightarrow, \}$. For example the Grammar $(\{S\}, \{a, b\}, P, S)$ where P consists of $S \rightarrow aSb, S \rightarrow ab$ can be looked at as a string $(\{S\}, \{a, b\}\{S \rightarrow aSb, S \rightarrow ab\}, S)$ It is to be noted that generally integers are represented as decimal or binary and not as unary. The problem can be re-formulated as one recognizing the language consisting of the 'yes' instances of the problem .

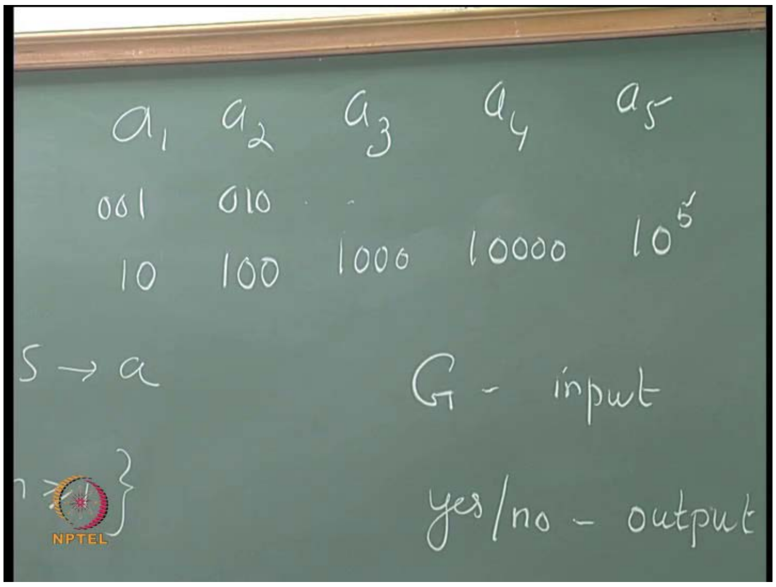
So, without loss of generality, we will restrict ourselves to decision problems and how can we encode a problem? As a binary string, any problem you can encode as a string. A Boolean expression can be looked at as a string. For example, this particular expression you write it as, without subscripts x x 1 plus x 2 naught x 1 plus naught x 2 and so on. What are the symbols which are being used? Left parenthesis, right parenthesis, the symbol x and the number, which follows x gives you the subscript. It is an integer using the symbols 0 to 9.

So, this string uses left parenthesis, right parenthesis, x plus naught 0 to 9. So, if taking this as the alphabet over this alphabet, this is a string right. Similarly, if you take a context free grammar, it can be encoded as a string. For example, this is the grammar and the non terminals are say S taking this particular grammar non terminal. There is only one non terminal s, there are two terminals symbols a and b set of production rules and the start symbol. And the production rules are given like this.

You may look at it as a string over this alphabet left parenthesis. Second symbol is this, third symbol is this and so on. So, what are the symbols being used? Left parenthesis and right parenthesis used in the beginning and the end, then when you denote the non terminal and the terminal as sets, you will be using this flower brackets and also in the rules you use the arrow mark. So, as a you can look at it as a string right, you can look the grammar as a string. This is how you encode a string.

There are one or two things you have to be careful about. Now, the alphabet size may become very large because you may have a number of non terminals. So, instead of that, we can just encode this in binary. We know how to do this. In the last class, we saw that any Turing machine, you can look at it in such a way that the tape alphabet consists of only 0 and 1 apart from the blank symbol.

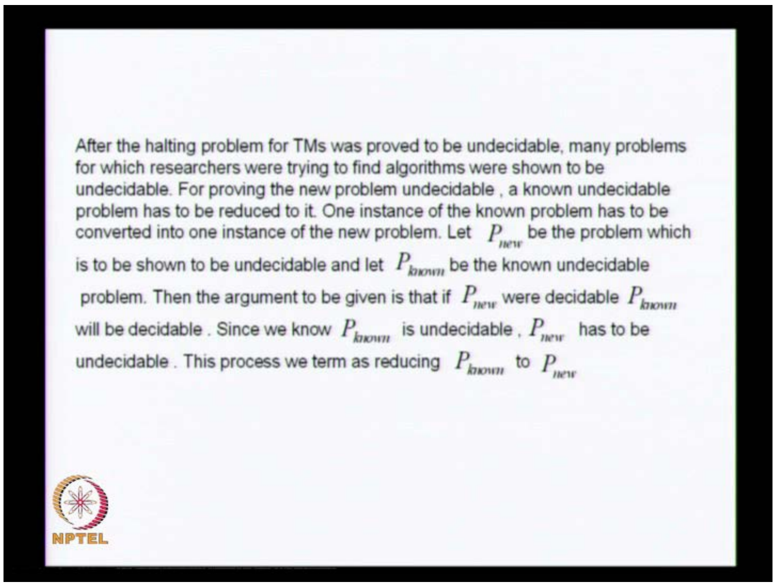
(Refer Slide Time: 16:23)



So, if this encoding use this k symbols say, a 1 a 2 a n, then to represent each symbol you can make use of log n to the base two symbols. So, for a example, suppose, I have a 1 a 2 a 3 a 4 a 5, I can use 5 like you know 0 0 1 0 1 0. Like that, I can use another way of encoding will be a 1 I represent as 1 0 a 2 I represent as 1 0 0 a 3 1 0 0 0 a 4 1 0 0 0 a 5 1 0 power 5 and so on that way also we can represent.

So, any problem you can encode as a binary string (( )). So, this is the connection between problems and languages. When you say that the problem is un decidable, it is equivalent to saying that the corresponding language which consists of the yes instances is not recursive.

(Refer Slide Time: 17:43)


After the halting problem for TMs was proved to be undecidable, many problems for which researchers were trying to find algorithms were shown to be undecidable. For proving the new problem undecidable , a known undecidable problem has to be reduced to it. One instance of the known problem has to be converted into one instance of the new problem. Let P_{new} be the problem which is to be shown to be undecidable and let P_{known} be the known undecidable problem. Then the argument to be given is that if P_{new} were decidable P_{known} will be decidable . Since we know P_{known} is undecidable , P_{new} has to be undecidable . This process we term as reducing P_{known} to P_{new}

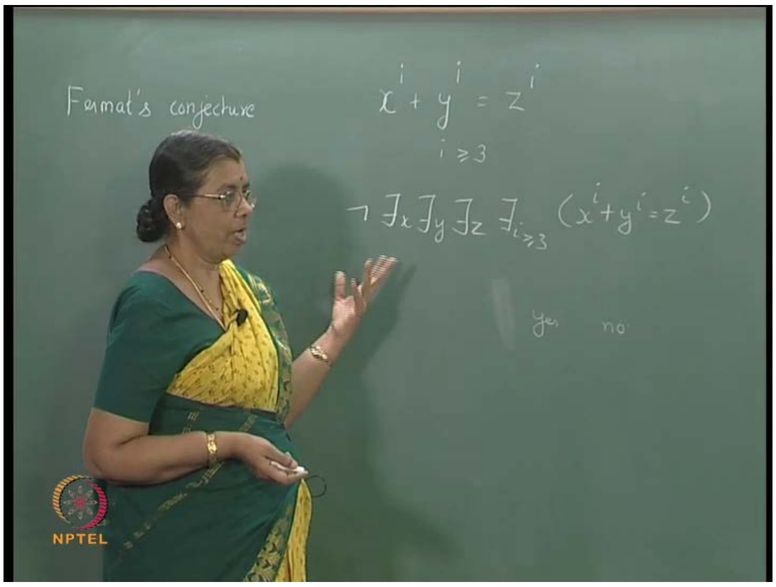
The halting problem for Turing machines was proved to be un decidable by Turing in 1936. Afterwards, many problems were shown to be un decidable. One of them is syllabus strength problem and problems like, you know, given to flow chart programmes do they do the same thing? Do they compute the same thing and sometimes given two flow chart programmes do they compute the same thing in the same manner? For example, factorial n you can compute by having 1 2 3 multiplied up to n or taking n initially, multiply it by n minus 1 and n minus 2 and so on.

Both of them will compute n factorial, but they do it in a different way. One starts with 1 and goes up to n, the other one starts with n and goes down up to 1. So, given two programmes do they compute the same thing? That is an un decidable problem. Even if they compute the same thing, do they do it in a same manner? That is again another un decidable problem. Hilbert strength problem is given, a polynomial with integer coefficients making it equal to 0, that particular equation will have number of roots.

If d is the degree, it will have d roots whether the roots are integers, rational, irrational, real, and imaginary that is to be seen. Does there exist an algorithm, which given a polynomial tells you, whether it will have integer roots? Polynomial with integer coefficient will it have integer roots? People have been trying that. Again as I told, particular instances of the problem you may be able to find out. Given one particular cubic equation or fourth degree equation where factorising you may be able to find out the root and tell whether it is integers are not.

But, in general the degree may be 100 the degree may be 200, the coefficients are all integers. Whether the roots are all integers are not, does there exist an algorithm? People have been trying this for a long time from say nearly beginning of the 20th century till Turing proved his result. At that time, they realized that there is no part in trying for that algorithm because no such algorithm exists. Again this is a very strong statement it tells you that nobody can ever find an algorithm for that problem.

(Refer Slide Time: 20:44)


Fermat's conjecture
$x^i + y^i = z^i$
$i \geq 3$
$\neg \exists_x \exists_y \exists_z \exists_{i \geq 3} (x^i + y^i = z^i)$
yes
no
NPTEL

At the same time, it through some light on problems like Fermat’s last theorem. What is Fermat’s last theorem? It is a Fermat’s conjecture. It says that you cannot have integers x, y and z such that i is also an integer and i is greater than or equal to 3. You do not have integers x y z and i; i greater than or equal to 3 such that this particular equation holds. Look at this. This is a single instance of a problem there is no set of a problem instances and so on. Some problems may have just one instance.

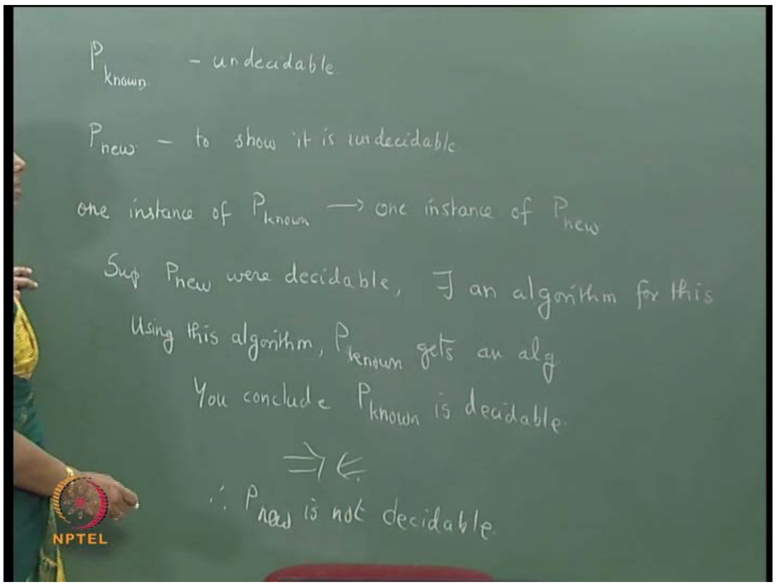
This is such a case, there is only one instance. x, y, z, i are not parameters; they are bound very well. Actually, it means that not there exist x there exist y there exist z there exist i greater than or equal to 0; x power i plus y power i is equal to z power i. So, x y is a dye or bound variables they are not parameters. So, this is only a single instance of the problem, either the answer is yes or no. It is decidable problem; we do not know we may not know.

Now, you know that is the conjecture is correct. Nearly 8 or 9 years back, it was proved that the conjecture is correct. But before that people thought, it may be true or it may not be true, but this un solvability of the halting problem showed that there should be an answer for that. Either it is yes or no you may not be able to prove that also. Even if the answer is yes it is a true statement, but by a result of Godel, any consistent axiomatic system for integer arithmetic will have true statement which cannot be proved.

So, people thought that probably we will never be able to prove that statement even though it is true. But finally, after some long difficult work this has been proved to be true; the conjecture has been proved to be true. That is to show that if a problem has only one instance, then it is a decidable problem. The other problems which we considered like ambiguity, they had several instances. For a particular instance, you may have an answer, but for the general problem there may not exist an algorithm.

Now, after this result, several other problems have been shown to be un decidable. How it was shown and how is it that you can show a new problem to be un decidable? Now, researches we are trying to find algorithms and we trying to show that problems are un decidable. Now, you want to prove a new problem to be un decidable. How do you do that? You have to take a known un decidable problem and it has to be reduced to the new problem. That is there should be one algorithm which converts one instance of the known problem to a one instance of the new problem.

(Refer Slide Time: 24:27)



P_{known} - undecidable

P_{new} - to show it is undecidable

one instance of P_{known} $\longrightarrow$ one instance of P_{new}

Sup P_{new} were decidable, $\exists$ an algorithm for this

Using this algorithm, P_{known} gets an alg

You conclude P_{known} is decidable

$\Rightarrow\Leftarrow$

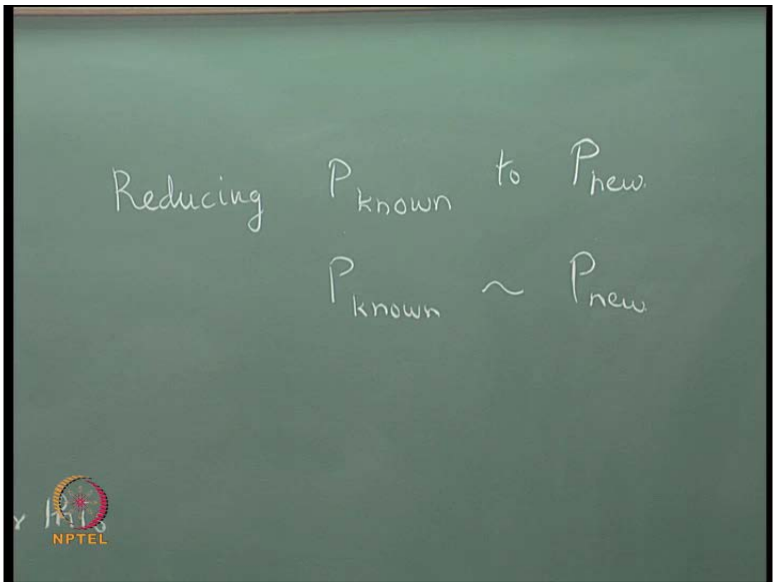
$\therefore$ P_{new} is not decidable

So, I have two problems. I have one problem p known; this I know is un decidable and I have a another problem p new. I want to show it is un decidable. How do I do this? There will be several instances here, several instances here so; there should be a way of converting one instance of p known to one instance of p new. So, while constructing one instance of p new from one instance of p known, we have to convert the yes instances into yes instances and no instances into no instances.

The argument will be like this, suppose, p new was decidable, that means there exist an algorithm for that. So, what you do is, take one instance of the p known problem. Convert it to an instance of the p new problem and call this algorithm. So, while constructing one instance of p new from one instance of p known, we have to convert the yes instances into yes instances and no instances into no instances. This algorithm will solve it. So, there is a solution for the p known also. If p new were decidable, there is an algorithm for this.

Using this algorithm, p known gets an algorithm. How does it get? One instance will be turned into one instance of p new and this algorithm will be caught and hence p known gets an algorithm for it. That is you come to the conclusion, you conclude p known is decidable, but we know that p known is un decidable, so you are arriving at a contradiction. So, that shows that p new is not decidable therefore; p new is not decidable. This you call as reducing p known to p new.

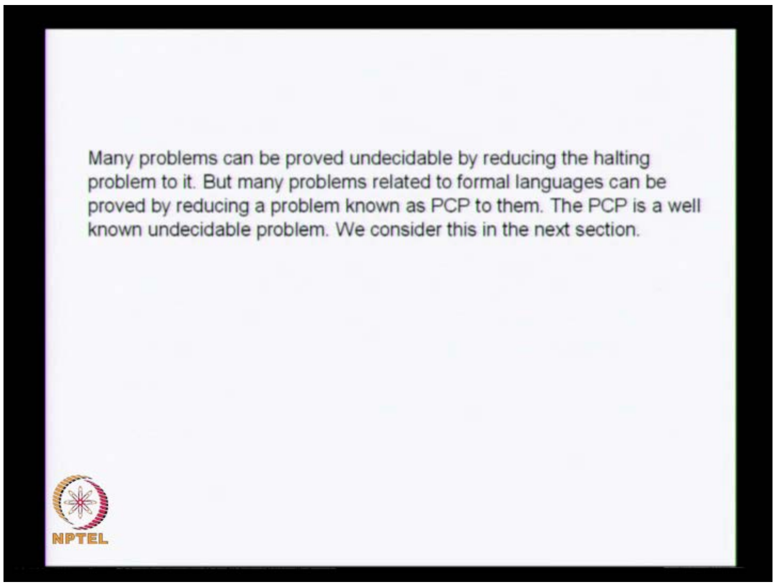
(Refer Slide Time: 28:10)


Reducing P_{known} to P_{new}

$P_{known} \sim P_{new}$

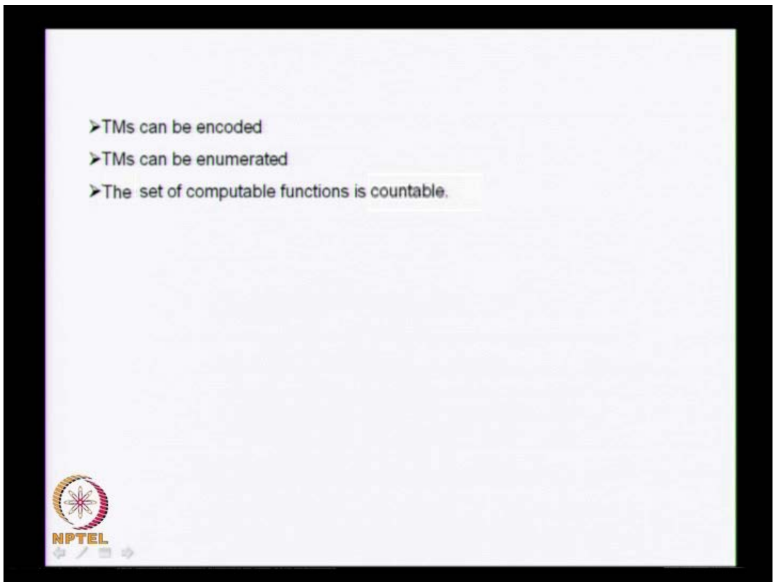
Or sometimes, this symbol is used, sometimes some other symbol is also used does not matter. But you say that p known is reduced to p new and you show that p new is un decidable.

(Refer Slide Time: 28:50)


Many problems can be proved undecidable by reducing the halting problem to it. But many problems related to formal languages can be proved by reducing a problem known as PCP to them. The PCP is a well known undecidable problem. We consider this in the next section.

Many problems can be proved to be un decidable by reducing the halting problem to it. But for problems related to formal language theory, like you know, ambiguity problem, whether a given context free grammar is ambiguous or not, we rather used another problem called p c p or post correspondence problem. So, first we prove that this is un decidable from the halting problem, then making use of this problem, we show that other problems are un decidable. We shall this later.

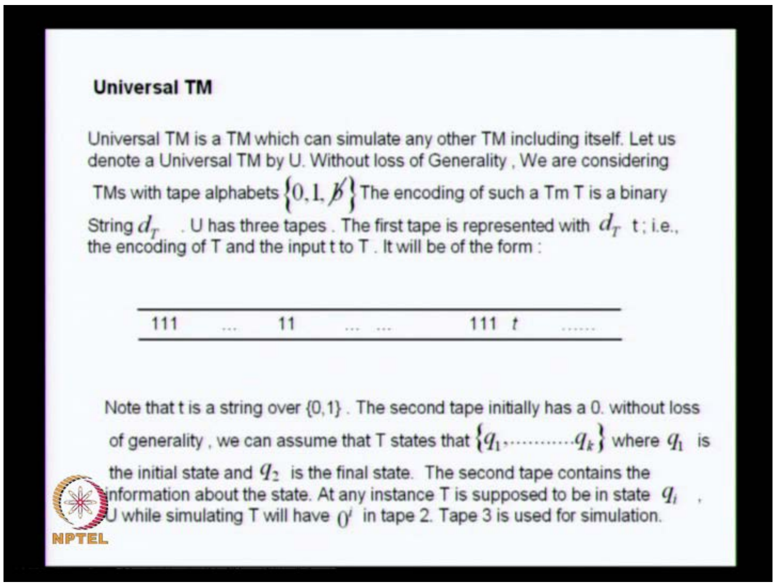
(Refer Slide Time: 29:32)



- ➢TMs can be encoded
- ➢TMs can be enumerated
- ➢The set of computable functions is countable.

Now, we have seen that a Turing machine can be encoded as a binary string. This we saw in the last lecture and Turing machines because of that encoding can be enumerated you can talk about the i'th Turing machine and you can also look at a Turing machine as computing a function. So, if you enumerate the Turing machine as t 1 t 2 t 3 etcetera, the Turing machine t i computes a function and those computable functions they are called Turing computable functions. And they can also be enumerated or it is a countable set of functions. The set of Turing computable functions is a countable set.

(Refer Slide Time: 30:28)



Universal TM

Universal TM is a TM which can simulate any other TM including itself. Let us denote a Universal TM by U. Without loss of Generality , We are considering TMs with tape alphabets $\{0, 1, b\!\!/\}$ The encoding of such a Tm T is a binary String d_T . U has three tapes . The first tape is represented with d_T t ; i.e., the encoding of T and the input t to T . It will be of the form :

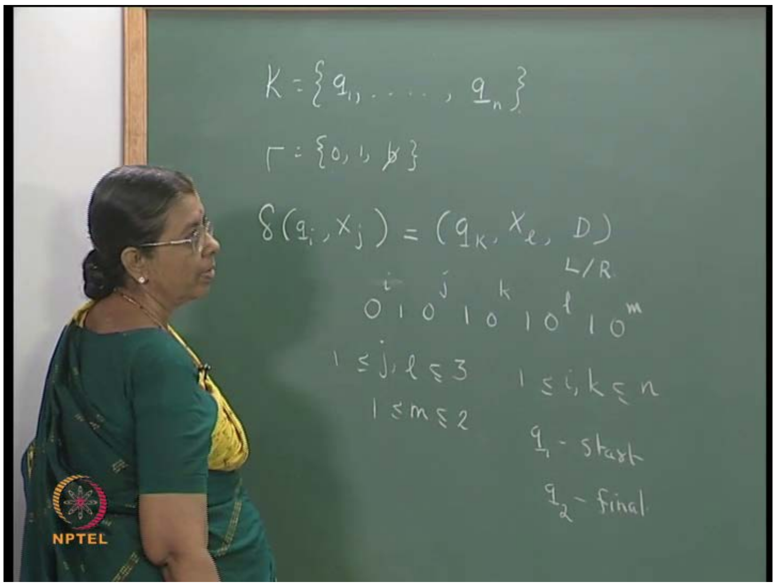
```
111   ...   11   ... ...      111  t     ......
```

Note that t is a string over {0,1} . The second tape initially has a 0. without loss of generality , we can assume that T states that $\{q_1 \ldots\ldots\ldots\ldots q_k\}$ where q_1 is the initial state and q_2 is the final state. The second tape contains the information about the state. At any instance T is supposed to be in state q_i , U while simulating T will have 0^i in tape 2. Tape 3 is used for simulation.

Now, let us see what is meant by a universal Turing machine. A universal Turing machine is a Turing machine which can simulate any other Turing machine including itself. Now, the universal Turing machine can simulate any Turing machine t, on a tape t, on an input t. Now, the alphabet for this is 0 1 and blank we saw that without loss of generality, we can assume only two letters apart from the blank symbol. So, we consider three symbols as a tape alphabet and the encoding of a Turing machine is a binary string d t. This also we have seen. The universal Turing machine given the encoding d t and the input t, it simulates t on t and behaves like t on t.

Now, the input will be of this form. The universal Turing machine will have three tapes. The first tape will contain the input and it will be of this form. The encoding of the Turing machine will be given first; the encoding dt. We know that the encoding begins with three 1(s) and ends with three 1(s) and are separated by two 1(s); the blocks are separated by two 1's and each block represents a move of the Turing machine.

(Refer Slide Time: 32:20)

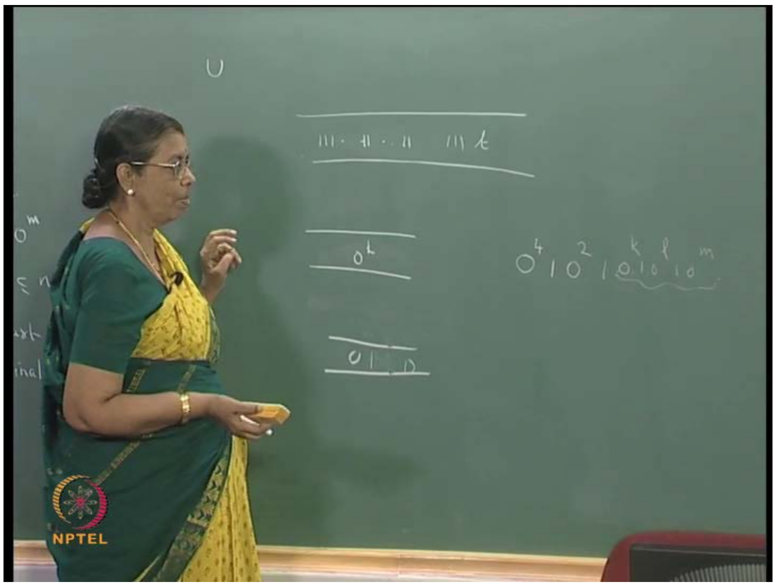

$K = \{q_1, \ldots, q_n\}$
$\Gamma = \{0, 1, \not b\}$
$\delta(q_i, X_j) = (q_K, X_\ell, D)$
L/R
$0^i 1 0^j 1 0^k 1 0^\ell 1 0^m$
$1 \le j, \ell \le 3$
$1 \le i, k \le n$
$1 \le m \le 2$
q_1 - start
q_2 - final
NPTEL

Let us just recall what we studied earlier. Suppose, the states of the Turing machine are q 1 q 2 q n, the tape symbols are 0 1 blank, the mapped given by delta of q i x j is equal to q k; it goes to state q k x l and the direction, which will be either l or r. This move means that it is in state q i reading the symbol x j and it will go to state q k replace x i by x l and then move left or right.

This move is represented by 0 power i, 1 0 power j, 1 0 power k, 1 0 power l, 1 0 power

m. Here, j and l will be 1 2 3; 1 means 0, 2 means 1, 3 means blank, m will be within 1 and 2; 1 means left move, 2 means right move. And i and k are states, they will vary from 1 to n. Without loss, we take q 1 to be the start state and without loss of generality, we can take q 2 to be the final accepting state.

(Refer Slide Time: 34:10)



Now, with this the universal Turing machine has three tapes. The first tape contains d t and t. Or in essence, it means that the first state will have the encoding and the tape t. We have seen that the encoding consists of blocks separated by two 1’s and each block is of this form. The second tape contains 0; the third tape is used for simulation.

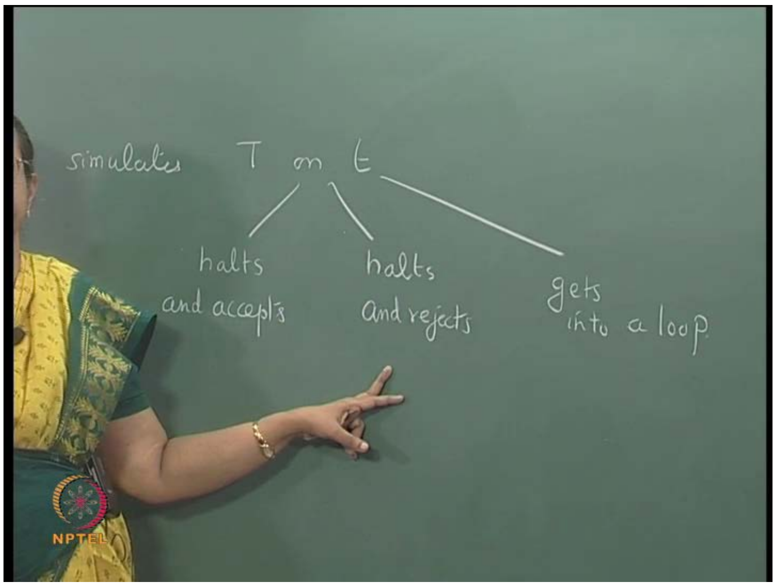
Now, initially, this has single 0, which means state 0. At any time, it may have some 0s, which will denote the states. So, you can have something like this. The third tape is used for simulation. So, initially, this contains 0 and t is copied on to this. That universal Turing machine before doing, it has to checks whether it is a proper encoding. We know that a proper encoding should begin with three 1’s, end with three 1’s and should have blocks. So, it checks whether it is of this form and we are considering only deterministic Turing machines now.

So, for one particular q i and x j, there should be only one map. So, the Turing machine first checks, it is a whether it is a proper encoding by checking whether it begins with three 1’s, ends with three 1’s and whether each block is of this form. And when it checks whether each block is of this form, it also checks whether i and k are between 1 and n, j and l are between 1 and 3 and m is 1 naught 2. If it has some other value, then it represents an improper encoding and the machine halts. A machine with improper encoding represents a Turing machine with no moves and so it cannot accept any string.

Now, once it finds it is a proper encoding, the initial state is q 1. So, 1 0 is here, t is copied on to the third tape and the behaviour of this Turing machine is simulated on the third tape. So, suppose, here the third, it is a binary string something like this. The third tape head is here it is reading a 1 and this contains a four 0s. That is, it means in state q 4, it is reading a 1, then it goes through this portion of the first tape and checks whether there is a block beginning with 0 power 4, 1; 1 is 2 so, 0 power 2, 1 and so on.

If there is a block, which begins like this, there is a move for this situation. That is in state q 4; it can read the symbol 1 and make a move. Then it records it, stores in its memory. What is given there something like 0 power k; 1, 0 power l; 1, 0 power m. So, it replaces this by 0 power k k 0s and prints a 0 or a 1 depending upon what is the value of l and then depending upon the value of m, it is moved left or right; tape head position of tape three will move left or right.

(Refer Slide Time: 38:50)



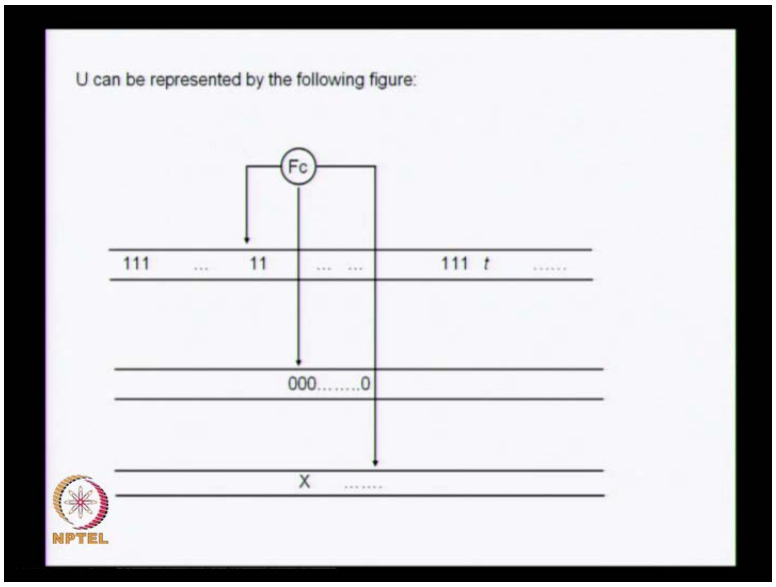
Thus you simulate one move of the Turing machine, in the universal Turing machine. So, if at some time, 0 0 appears in the second tape, that means you have reached the final state and so this string will be accepted. So, the possibilities for T on t, u simulates a Turing machine T on t. t is the input; small t is the input, capital T is the Turing machine.

It is given like this on the first tape. The possibilities are this halts and accepts in a situation where known next move is possible, it halts and it is not a final state. That means the string is rejected.

It halts and rejects or gets in to a loop. In this case also, the string will not be accepted. In this case, the string will be accepted, in these two cases, the string is not accepted. u while simulating T on t, if this halts and accepts it will also halt and accept. If it halts and rejects, this is also halts without accepting; that is, it is rejecting the input. In this case, u also will get into a loop. One step by step, it will simulate the move of t and it will get into a loop. This way, a universal Turing machine can simulate any other Turing machine, if suppose you take t to be u itself, it can simulate itself.

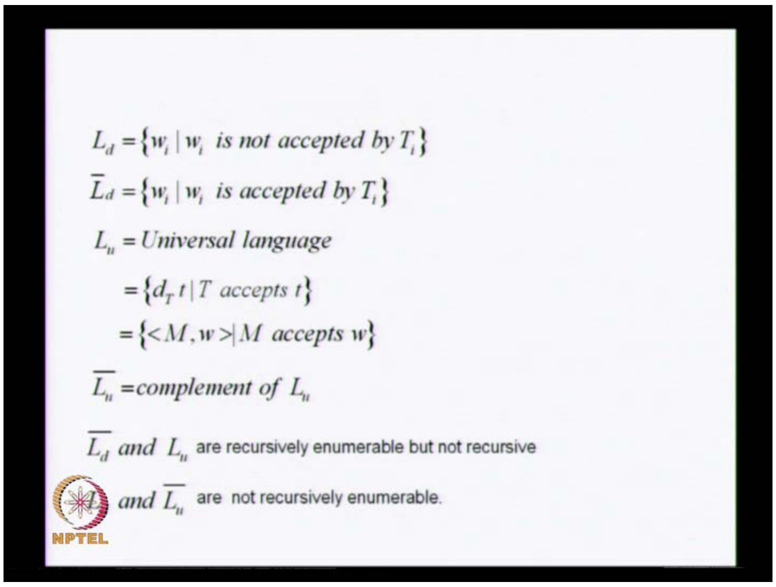
Now, one point we have to note is that we have used three tapes for the universal Turing machine. Actually, it is not necessary; we know how to simulate a multi tape Turing machine with a single tape Turing machine. So, let u be the universal Turing machine with one tape. You can make use of just one tape alone because a multi tape Turing machine can be simulated by a single tape Turing machine.

(Refer Slide Time: 41:10)


U can be represented by the following figure:

So coming back to this (No audio from 41:10 to 41:21)

(Refer Slide Time: 41:20)



$L_d = \{w_i \mid w_i \text{ is not accepted by } T_i\}$

$\overline{L_d} = \{w_i \mid w_i \text{ is accepted by } T_i\}$

$L_u = \text{Universal language}$

$= \{d_T t \mid T \text{ accepts } t\}$

$= \{<M, w> \mid M \text{ accepts } w\}$

$\overline{L_u} = \text{complement of } L_u$

$\overline{L_d}$ and L_u are recursively enumerable but not recursive

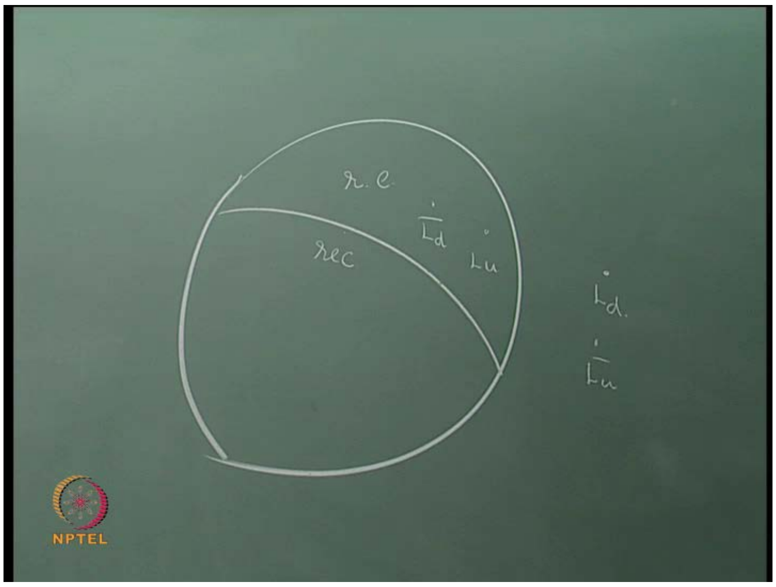
L_d and $\overline{L_u}$ are not recursively enumerable.

We have seen about the languages l d and l d bar. How did we define them? They we defined l d as w i; w is not, w i is not accepted by t i. l d bar is w i; w i is accepted by t i. Now, we can define a universal language as the language accepted by a universal Turing machine. Actually, it will accept strings of this form d t t, where d t denotes the encoding of t and t is the input to t. So, if t accepts t, this particular string will be in the language l u, if t does not accept t this particular string will not be in l u.

So, u we can define a universal language like this in this form. Sometimes, we use this notation; m w, m accepts w. That means the string is a binary string; you must realize that it is a binary string. Only for notation sake we use n comma w there is no comma or this less than symbol or right greater than symbol or anything.

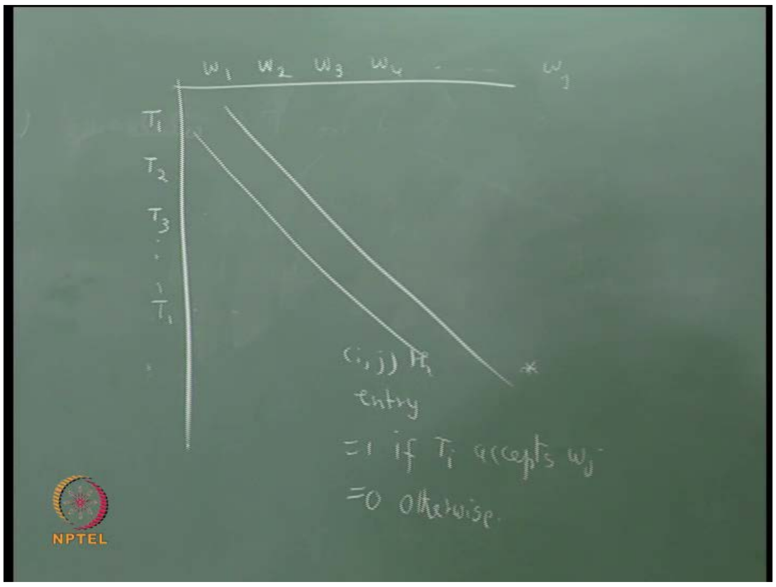
It really denotes something like this. The compliment of this language is denoted as l u bar. We can show that these two l d and l u bar, they are not even recursively enumerable whereas, l d bar and l u, they are recursively enumerable, but not recursive. So, in the last class I mean in the last lecture, we considered this recursive set recursively enumerable set.

(Refer Slide Time: 43:10)



For a language l and l bar, they can be both here, they can be both here. One can be here, one can be here but both cannot be here. If l and l bar are here, then they will come within this. That is what we have seen. So, in this case, we find that l d is here, l d bar is here, l u is here, l u bar is there. This is a thing. Now, recalling what we did in the last class.

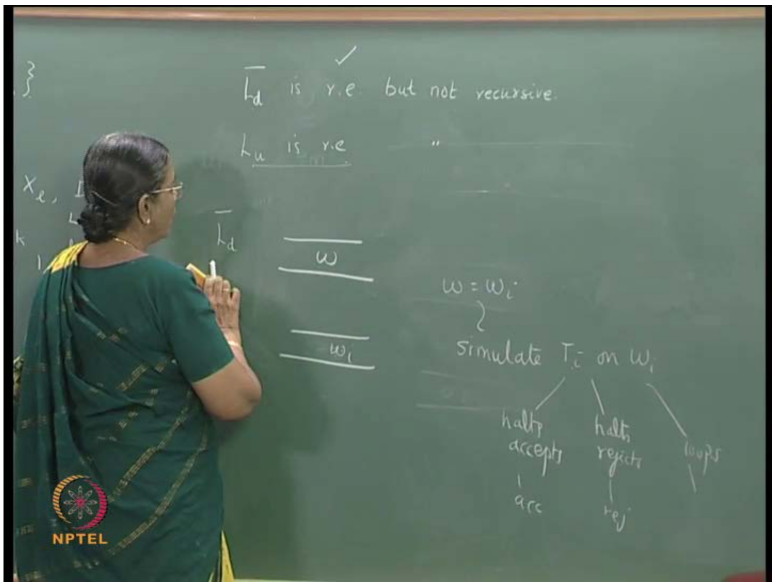
(Refer Slide Time: 44:02)



We have an infinite Boolean matrix where the strings over 0 1 are enumerated and in the column wise, you have the Turing machine. The i j'th entry, that is the entry in the i'th row and j th column is equal to 1 if t i accepts w j. That is if the i th Turing machine accepts the j th string, the entry will be 1. Otherwise, if it does not accept, it will be 0 and l d corresponds to the 0 elements of the diagonals.

If you consider the diagonals here, there will be some 0s and 1s. l d corresponds to the 0 elements, l d bar corresponds to the 1 elements and we have seen that l d is not recursively enumerable. Now, we will show that l u is recursively enumerable, but not recursive and l d bar is also again recursively enumerable, but not recursive. How do we show that?

(Refer Slide Time: 45:40)



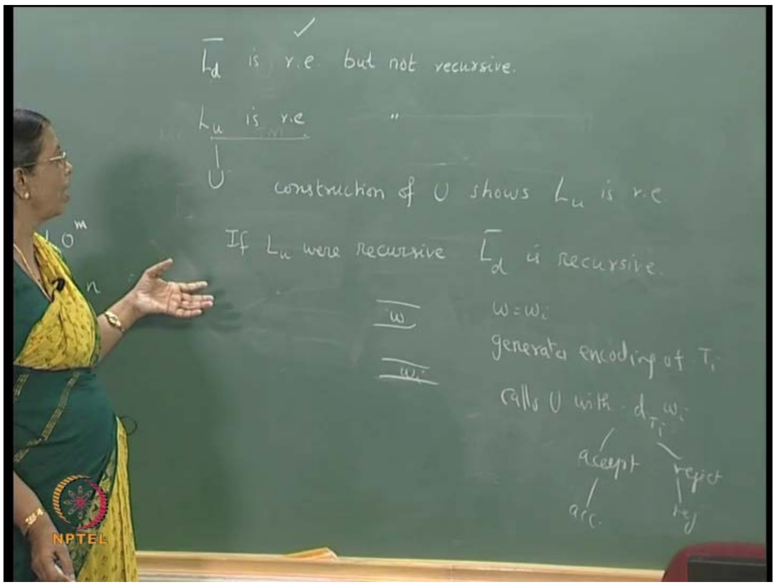
L d bar is recursively enumerable, but not recursive. How do you show that? Similarly, how do you show l u is recursively enumerable but not recursive? Now, for l d bar you show like this have one tape, the input w is given here. You can make use of another tape, where one by one you string; you generate the strings in that enumeration w 1, w 2, w 3, w 4 etcetera. Each time you compare with this. If they match, suppose, w i is generated here and it is matching with this.

That means the given input is the i'th string in the enumeration. Once you find that w is w i, after finding this, simulate T i on w i. You generate the encoding of the i'th Turing machine. What is that? It is an integer i. That is what we have seen earlier. So, you generate the encoding of the i'th Turing machine and simulate t i on w. The possibilities are it halts accepts halts and rejects or it gets into a loop.

So, the mission for l d bar is like this. It finds out that the given string is the i'th string in the enumeration. Then it generates the encoding and then simulate t i on w i. If it halts and accepts m also accepts, if it halts and rejects, this rejects. If it loops also, it may get into loop, it will not accept. Anyway, if t i is accepted by if w i is accepted by t i. It will be accepted by this machine m, which you are constructing. That is why you are able to construct a Turing machine for l d bar. So, it is recursively enumerable. So, l d bar is recursively enumerable.

Why it is not recursive? Because if it is recursive, the compliment of a recursive set is recursive so, l d will become recursive, but we know that l d is not even recursively enumerable. It lies outside so, in that case, we know that it is not recursive. So, we shown that l d bar is recursively enumerable, but not recursive. The next is same thing we want to show about l u. l u is recursively enumerable, but not recursive. How do you show that l u is recursively enumerable?

(Refer Slide Time: 49:26)


$\overline{L_d}$ is r.e but not recursive.
L_u is r.e "
U
construction of U shows L_u is r.e
If L_u were recursive $\overline{L_d}$ is recursive.
w
w_i
$w = w_i$
generate encoding of T_i
calls U with $d_{T_i}\ w_i$
accept
reject
acc.
rej

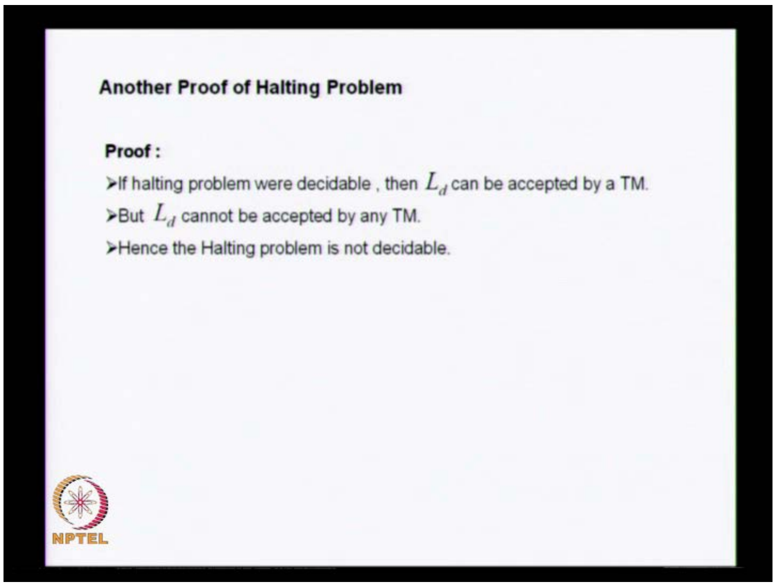
We have constructed a universal Turing machine u. So, you are able to construct a Turing machine for that. That means construction of a Turing machine, construction of u shows l u is recursively enumerable. Now, it is not recursive to show that, if l u were recursive, l d bar is recursive. You show that if l u were recursive, l d bar is recursive. How do you show that?

You can construct a Turing machine which halts, and which always halts for m l d bar.

So, how does it work? Given a input w, it finds out that it is the i th string in the enumeration. Then generates encoding of the i th Turing machine. Then it calls you with b of t t i and w i. Now, if l u where recursive, this will come out with the answer. This will be accepted or not accept or reject. There is no looping. If it accepts, that means t i accepts w i.

So, the mission you are constructing will accept and if it rejects, this will reject. That shows you have a mission, which always halts and it accepts l d bar, but that is not correct. That is not true. So, l u is recursively enumerable, but not recursive. By the same argument we know that l u is recursively enumerable, but not recursive. So, the compliment has to be not recursively enumerable. Because if it were recursively enumerable both of them will lie here and we know that both of them will lie here. We know that if both of them will lie, they will be shifted here right. So that way the have been proved.

(Refer Slide Time: 52:09)



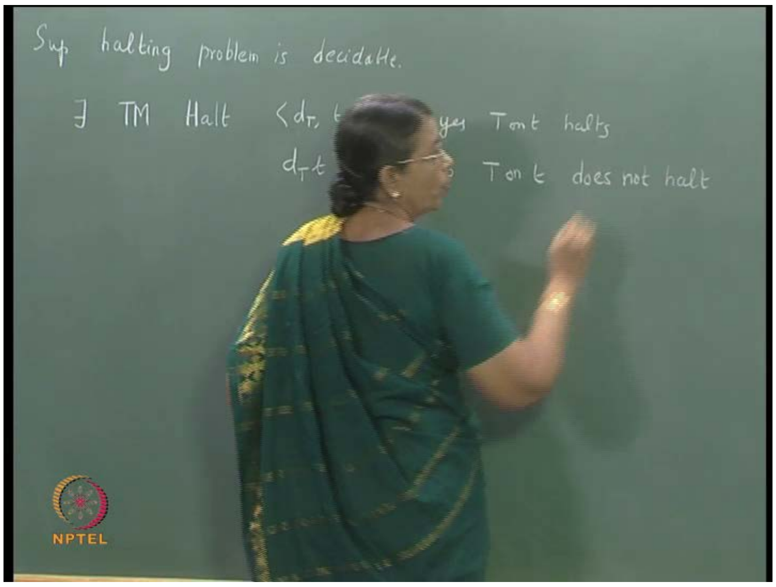
Another Proof of Halting Problem

Proof :

- If halting problem were decidable , then L_d can be accepted by a TM.
- But L_d cannot be accepted by any TM.
- Hence the Halting problem is not decidable.

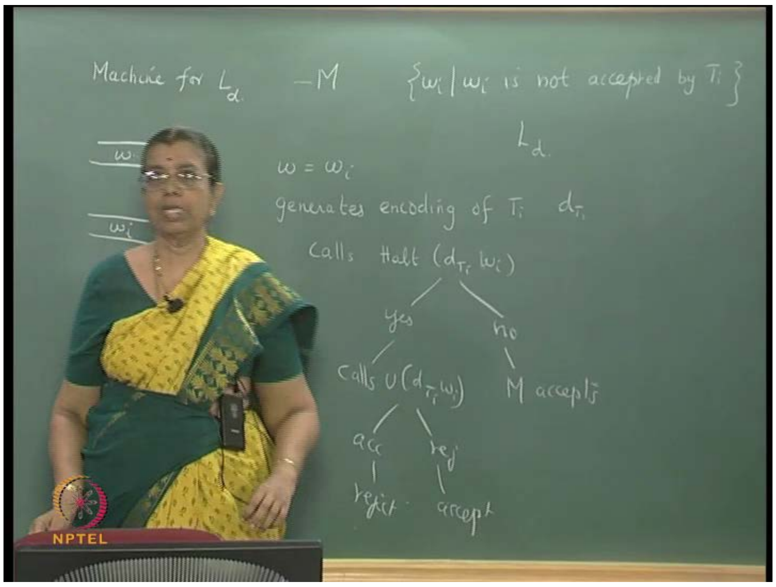
This gives another proof for the halting problem we have already seen one proof for the halting problem in the last lecture. Now another proof can be given for this. So, let us consider another proof for halting problem. It is like this.

(Refer Slide Time: 53:00)


Sup halting problem is decidable.
∃ TM Halt <d_T, t> yes T on t halts
d_T t no T on t does not halt
NPTEL

If the halting problem were decidable, then the language l d can be accepted by a Turing machine, but we know that l d cannot be accepted by a Turing machine. So, the halting problem is not decidable. So, the argument will be like this. Suppose, halting problem is decidable, then there exist a Turing machine halt. And for this, the input will be sum d t and t because the encoding it will be like this d t t. And it will tell you yes or no, T on t halts or T on t does not halt.

(Refer Slide Time: 54:00)



Now, making use of this, you can construct a Turing machine for l d as follows. Mission for l d, given an input w, it will keep on generating 1 2 3 and comparing and then find out that the given string is the i th string in the enumeration. That is it finds out that w is equal to w i and then generates the encoding. Encoding of the i th Turing machine calls that is, dt i calls. It calls as a subroutine, the mission halts with dt i and w i, now halt will always gives you an answer yes or no.

If it says no, that means the i'th Turing machine does not halt on the i'th string. It will get you into a loop. In this case, the mission, which you are going to consider m m accepts. If the answer is yes, that is t i and w i will halt, then it calls the universal Turing machine with d t i w.

Now, the universal Turing machine will say accept or reject. That is t i accepts w i or t i does not accepts w i. If it rejects, m will accept. If it accepts, m will reject. That way m (( )) strings of the firm w i, w i is not accepted by t i. If t i accepts w i, m will not accept it. It will be like this. If t i does not accept w i, it may be by stopping in a configuration, where the next move is not possible. That is taken care of this and m will accept that string. If t a gets into a loop and does not accept w i, it is this situation and in this case also m will accept w i.

So, this machine points out that the given string is i'th string in the enumeration and generating the encoding of the i'th Turing machine. First calls halt and then if it is says no, then it will accept. If it says yes, then it will call the universal Turing machine. Note that this will always halt because only in the case when t i halts on w i, we are calling u. So, in this case, the machine will always halt and come out of the sub routine right. So, if it accepts, m will reject; if it rejects, m will accept.

In any case, if w i is not accepted by t i, and then m will accept w i and this is the language l d. So, we are able to have a Turing machine for l d, but we know that, that is not possible and all this has come from the assumption that there is an algorithm for solving the halting problem right. And hence the halting problem is not decidable. There is no algorithm for solving the halting problem. So, thus we have seen the un decidability problems. We shall look into some more problems for this for Turing machine. These are for languages, we have to consider some results about properties, about languages accepted by turing machine, and also turing machines themselves.