Course on Theory of Computation
By Professor Somenath Biswas
Department of Computer Science and Engineering
Indian Institute of Technology, Kanpur
Lecture 40
Module 1
TMs can simulate computers, diagonalization proof.

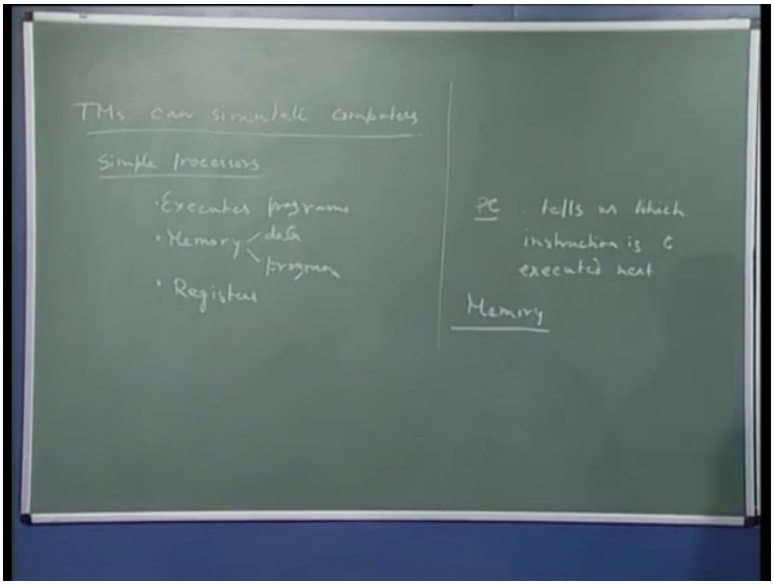
By now you must be convinced that the Turing machines are quite powerful as devices that can implement algorithms we have seen not just the basic model of Turing machines we have seen several extensions that is adding of some extra power or facilities to the basic machine and we have found that even with added power the class of languages which would be accepted after you know for example an extra tape is added something like that, that class of languages which can be accepted by Turing machines that class remains invariant.

Before we study the limitation of this class of languages which we had called recursively enumerable class of languages I would like to emphasize again that Turing machines are very powerful devices although they look very simple.

By now if you think a little you should be convinced that Turing machines can simulate computers as we know let me just spend a couple of minutes on this point that now when we say a Turing machines can simulate computers what we would like to say is that suppose you are carrying out an algorithm by means of computer as you know that particular algorithm can also carried out by a Turing machine.

To convince you of this statement what we can do is to consider a very simple processor a now again I would like to emphasize that just because I am considering a simple processor I am no way limiting the scope of this particular statement because even a simple processor can simulate that is known fairly or however elaborate processor that you might think of.

(Refer Slide Time: 3:12)



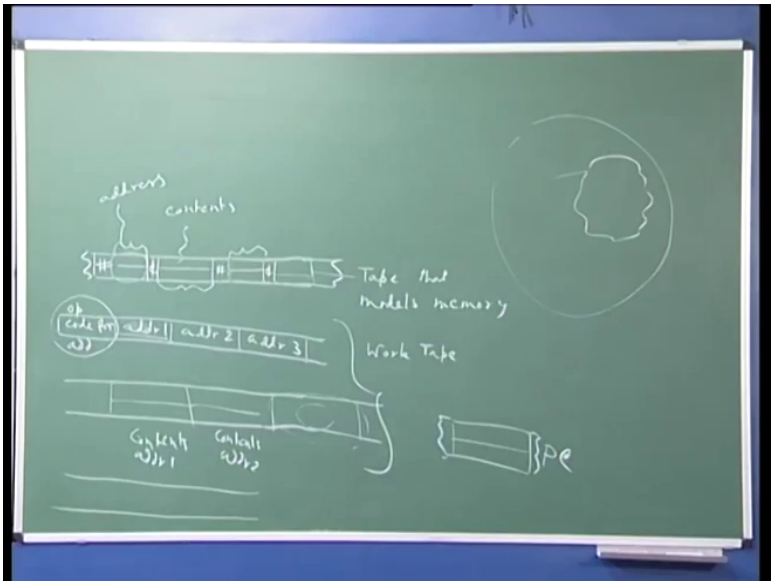
So what is a processor simple processor that can be modeled this way that a simple processor of course executes programs, it has memory and this memory contains both data and the program, right? And a simple processor can have some registers and registers also can store a some bit pattern, right?

So we are just talking of simple processors, alright? Now from our knowledge of you know computer organization we know how does a simple processor operate (())(4:10) what it does that at any given time there is some contents of program counter which is a register, now program counter tells me tells us which instruction is to be executed next and of course you know program is sequence of instructions that can be of course the execution of a program how it will execute in which sequence these instructions will be executed that can depend on data etcetera that all we know but basically there is some place which has the address of the next instruction, right? To be executed.

And then of course there is memory, and what is the essential part about a memory? That every memory cell has an address and in that address we that cell at that address contains some bit patterns, right? So a typical instructions could be like add, okay M1 and M2, M3 something like this, right? This is the this instruction is telling that add the contents of memory 1 cell 1 to memory 2 and the result should be placed in memory address 3. So one way of doing all this by a kind of machine that we have by machine I mean a Turing machine that we have we can imagine

our Turing machine has several tapes and one tape contains or models the memory that models memory, alright?

(Refer Slide Time: 6:38)



And this also may have you know may have another tape to do work etcetera we can say work tape just consider how an instruction like this could be added, first of all we need to say little more about what we mean by modeling memory we may say that we have some special symbol followed by a string and then again this special symbol followed by some string and then again here let us say in between I have a cell which also contains another special symbol let us say, alright?

So imagine this now what I will interpret that this part is the address after all address is string and this part between cent symbol and the next sharp symbol is the we are saying what would be the contents of the memory cell whose address is this. Now of course in normal day to day processors we have some bound which is the word length etcetera that kind of situation but here in this model we can have arbitrary sized contents, right? So now an instruction which is to be executed which is like this let us say which will that you know these parts of the instruction will tell me three addresses, right?

So imagine that here I had the contents of the addresses, alright? So M1 is an address, M2 is another address, M3 is another address, right? Alright? Now let us say this is the address 1 which is here M1 now if you see here three addresses, alright? And here there is the code for or you

know opcode basically that is why we that is what we say opcode for let us say addition which is a particular bit pattern.

Now what we the or our Turing machine can do that in another tape it can fetch the contents of this address 1, how it will do so? Because you see this is the bit pattern and therefore this of course goes to infinity maybe on both directions so it looks for this pattern in the places in those placed where addresses could be stored see that addresses are stored flanked by a sharp symbol and a cent symbol, right? So it will look for it will look for such a such strings in this tape and compare with this, right? Bit or you know symbol by symbol if they match then it knows that it has reached the correct address and then it can copy the contents here, okay in another and similarly it can so basically what I am saying is that it is not difficult to see that fetching the contents of some specified addresses can be done fairly easily in this way.

And now what does opcode addition involve you know executing that it would just involve that adding this and this if the contents of address 1 is here contents of address 2 is here so essentially we need to add and you can imagine that depending on whatever is the opcode here the Turing machine which is of course is going to be a big thing it will go to a particular part of its. So you know after all opcode is again a string the machine can decide you know to go to part of its machines program or you know after all a Turing machine can be seen as a lots of states and in each state being told what is to be done.

So it goes to an appropriate part of itself where these operations will be carried out essentially the operations what for addition is add this to this and maybe the result of the addition we may you know we can carry out that addition in another tape, right? And that addition is placed here contents of the result of the addition. And now the last part of this executing this simple instruction was to store the result in the address or in the cell whose address is address 3, again what we do is we will be scanning the memory tape the Turing machine will scan the memory tape so it will find out where this match is occurring, right?

Again remember that strings which are flanked on the left by sharp symbol and on the right by cent symbol these are addresses and when it finds then after that cent it should place this one, now of course it might mean you know creating space or squeezing some space but all those are simple operations by now you should be convinced that they can be done.

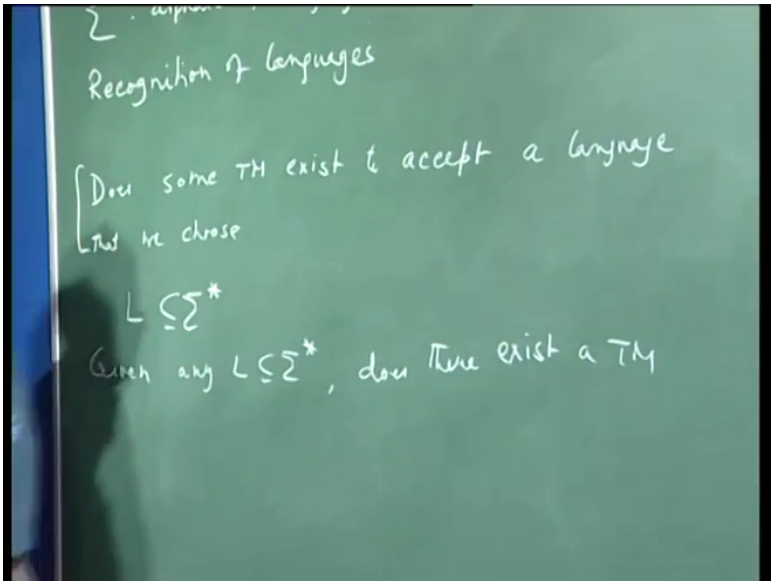
So after that it should carry out the next instruction, right? So what would the next instruction where will the next instruction be? Next instruction will be also there in the memory somewhere at a cell whose current address is the contents of the PC. So basically if PC was you know PC you can think of is somewhere again on some another work tape or something with the program counter.

So now these are all some tapes but this PC is program counter contents here now normally what we mean is unless the program counter value changes of the operation of the previous instruction we will add that what we say add 1 to the PC which really means go to the next instruction. So you know again you look for this address between sharp and cent wherever you find it skip the next contents part go to the next thing here, right? So whichever is the way either you know we can say physically that the instructions will be one after another on the tape.

So if you know you just carried out instruction corresponding to this address or contents of this address for the next you need to the address will be given here. So that you copy and that will have the correct PC value that is what you do after this and then from there from that address you fetch the next instruction in this tape and that then you will be updated the Turing machine will be updated to carry out the execution or the simulation of the execution of the next instruction.

So this is at a very gross level I have described that to you or I have tried to convince you that it is not difficult to see that a simple processor can be simulated by a Turing machine essentially simulating each instruction of the this simple processor program in an appropriate manner. Now therefore what we can say that Turing machines are of course fairly powerful in the sense they can do all that our ordinary or even extra ordinary computers can do and when I say that it can do whatever computers can do if we want to you know if we want to work on this statement in the sense if we want to find out what are the limitations of this statement in the sense that can Turing machines do everything that you might think of doable or it has its limitations there we need to be a little careful and therefore we will come back to the context in which we operate and that is recognition of language we call that a language is some subset of finite strings over an alphabet and let say the alphabet this is the alphabet of languages under consideration.

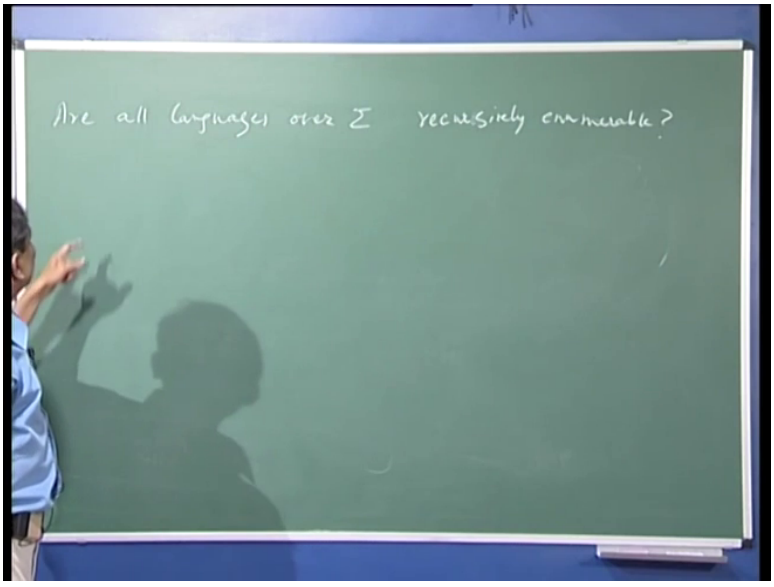
(Refer Slide Time: 19:02)

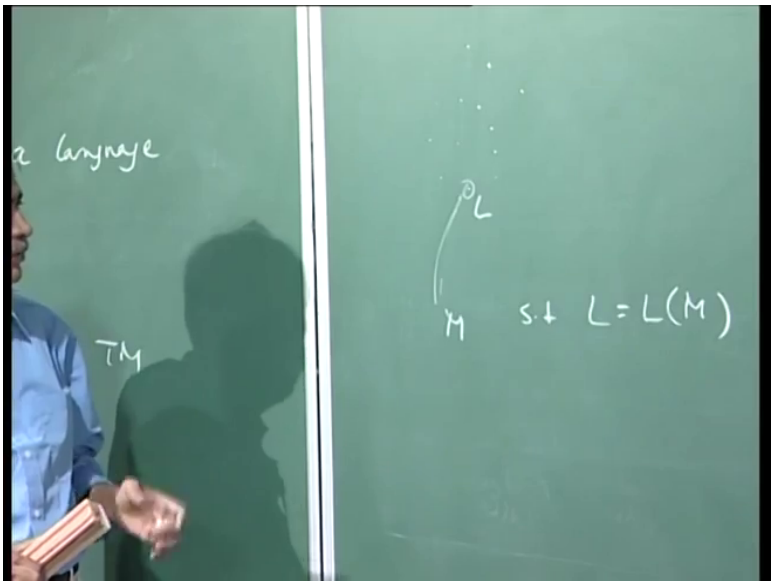


And now we may ask very simple question that does some Turing machine exists to accept a language that we choose, let me put it slightly this is not very nicely put the way we would like to say is that you see there are a language over sigma is of course a subset of sigma star, right?

So the focused question is given any L which is a language over sigma and you can ask this question does there exist a Turing machine to accept L.

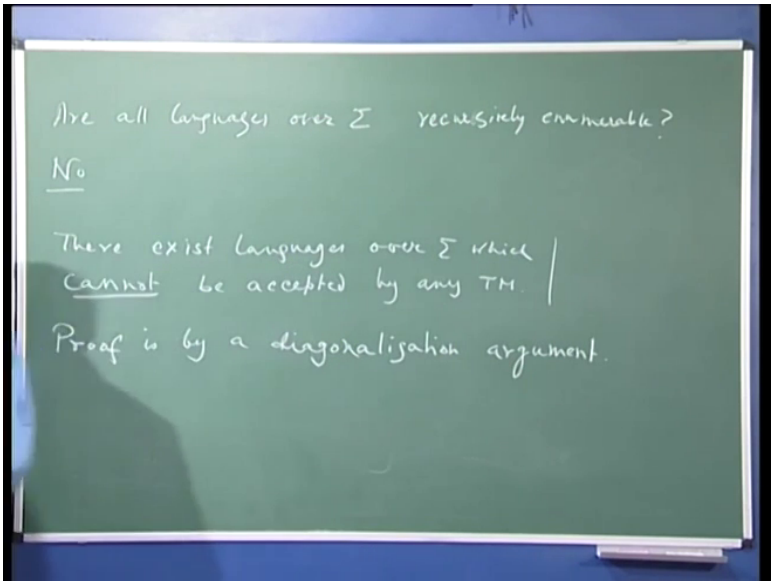
(Refer Slide Time: 20:57)





So in other words if we see what we are really saying is that we are asking the question are all languages over sigma recursively enumerable, right? That is the focused way of asking about the power of Turing machines so far as language recognition is concerned over some fixed alphabet, remember just once more the definition of recursive innumerability we say a language is recursively enumerable if there is a Turing machine to accept that language and the question that we are asking is there are so many languages over sigma star do you have for each language L do you have a Turing machine M such that L is the language accepted by this Turing machine of course different Turing machines will accept different languages that is fine or the same language maybe accepted by different several Turing machines but given a Turing machine that accepts a specific language and we want to know that the class of all Turing machines when we consider the languages they accept the languages the class of languages accepted by the class of Turing machines does that class exhaust all languages over sigma, right?

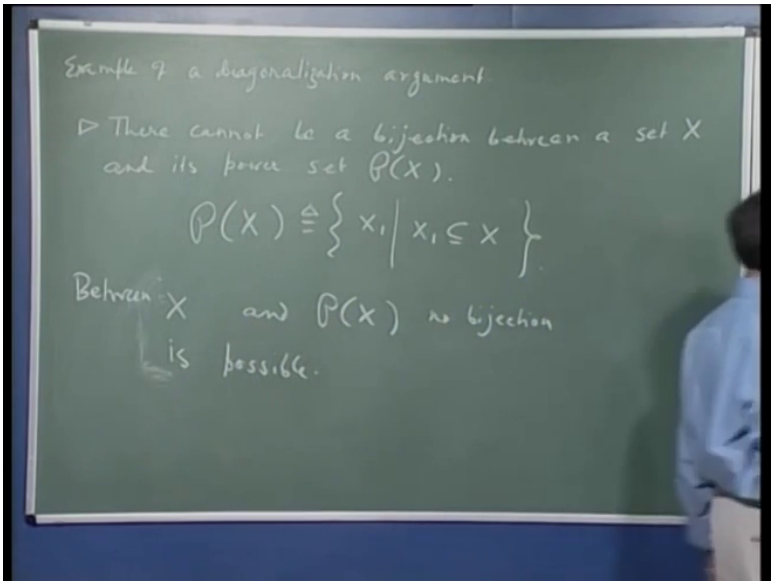
(Refer Slide Time: 23:02)



So this is the question and answer to this question is, no. So there are in other words what we are saying that there exist languages over sigma which cannot be accepted by any Turing machine, alright? You can see that if we prove this statement then we have proved limitations of what Turing machines can do so far I have been trying to emphasize that Turing machines are very powerful objects in the or devices in the sense that they can carry out all kinds of algorithms and as is the philosophy of this course we would like to understand the power or limitation of the power of any device that we are considering so in particular we want to know that what is the power or where what is the limit up to which Turing machines can be used, alright.

So this proof of this statement is to our I mean this statement that there exist languages over sigma which cannot be accepted by any Turing machine proof of this statement is by something called a kind of argument which is called diagonalization so proof is by a diagonalization argument this is a very powerful style of proof you know you know some proof strategies like induction, proof by contradiction etcetera this is a kind of this is a special kind of proof by contradiction and some of you might have seen the proof that there cannot be a bijection between the set of natural numbers and the set of real numbers that proof typically that we know of is by a diagonalization argument I would like to explain the diagonalization argument first by means of a simple but a fairly powerful proof of.

(Refer Slide Time: 26:30)

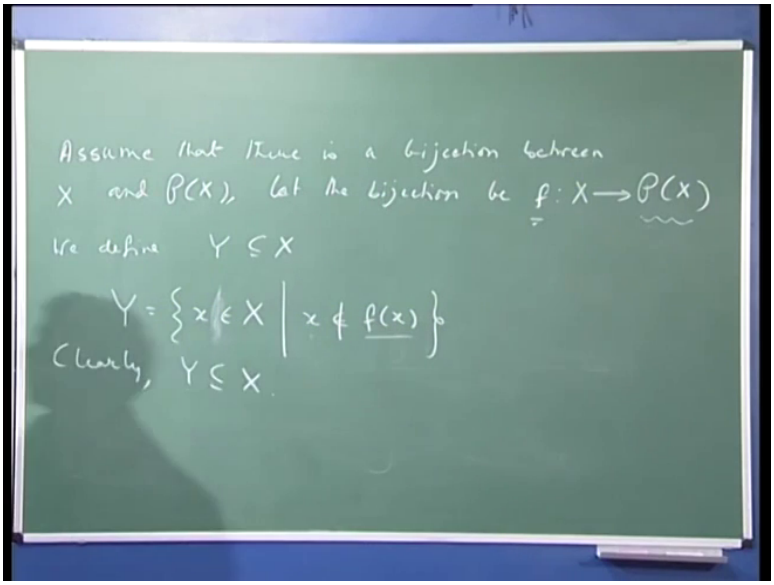


Now what I will prove is this statement that there cannot be a bijection between a set x and its power set it is normally denoted as you know script or p of x now and we read it is as power set of x, now what is the what is the power set of? So P(x) is by definition set of all x1 such that x1 is a subset of x, okay. So power set of x consists of the set of or power set of x is a set itself I mean as the name suggests and this set contains all subsets of x, so in particular it contains the empty set because empty set is a subset of x as well as the x itself because x is again the subset of itself.

What this statement says that between x and P(x) no bijection is possible, what we are trying to do? Let me just say it hear example diagonalization argument our diagonalization argument will prove this statement which is of course same as that statement. Now as I said the diagonalization argument is also a kind of a proof by contradiction and any proof by contradiction starts by assuming the negation of what you want to proof. So I should write here between x and P(x) no bijection is possible that is your statement which you would like to proof.

So negation of this statement is that there is a bijection between power set of x and x, right?

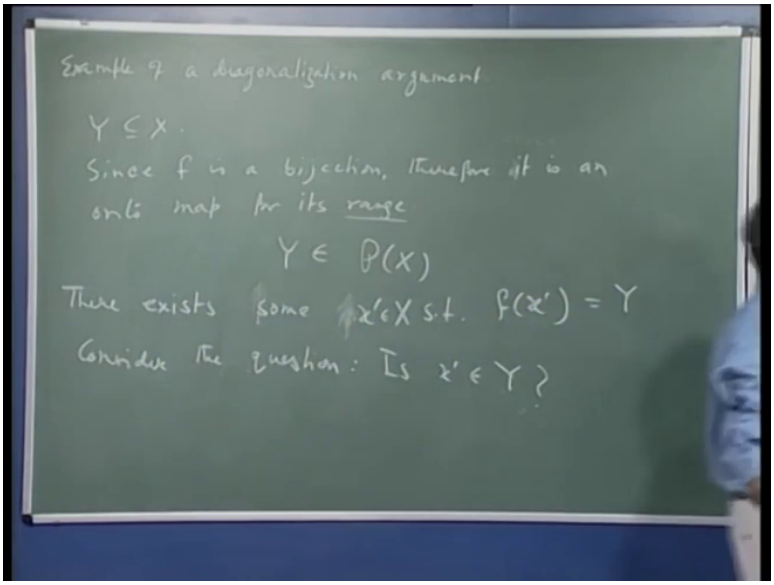
(Refer Slide Time: 29:42)

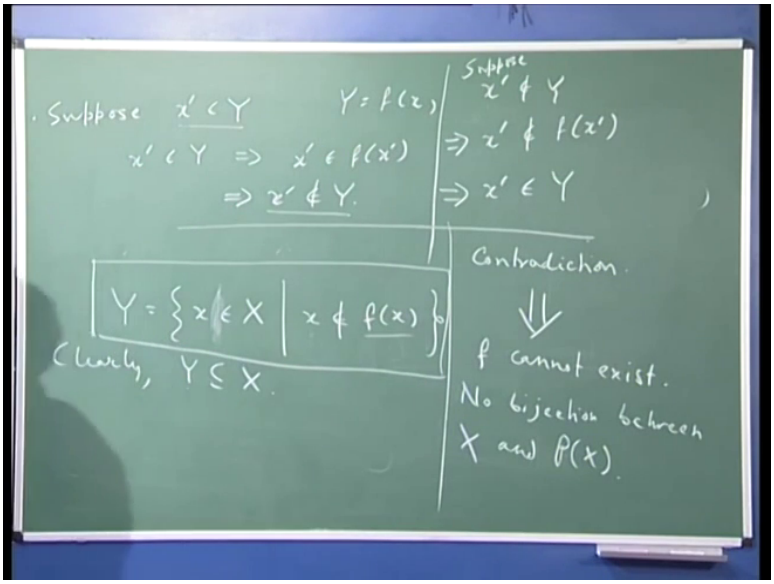


Assume that there is a bijection between x and P(x) let the bijection be f and this f maps x to P(x) being a bijection of course f inverse also exists and f is 1 to 1. Now what we will do is we defined Y which will be also a subset of x and the definition of Y is as follows that Y consists of x in x, right? Such that x is not an element of x, right? This looks so very (())(31:18) statement but its very interesting statement because we will see how it manages to this definition itself how it manages to prove this very general result, alright?

So essentially where how we are defining Y? This capital Y, we are saying look an element of x will be in this set Y if that element is not an element of the set or subset of x to which this small x is mapped to this is what it is, right? Now we can see clearly Y is a subset of x because Y by definition basically some elements of x, so capital Y is a subset of x. Now since it is a subset of x map f we will map it to of course some element you know x, x for every x we have this but you see because it is a bijection f is a bijection the at this remember this f is from this set to the power set so since f is a bijection for every element of this set there would be some corresponding element of x which alright is the map of so we are taking that element from here.

(Refer Slide Time: 33:23)





In other words so far as our purpose is concerned again Y is a subset of x since f is a bijection therefore it is an on to map for its range, right? And of course the range is the whole power set and in the range Y is there so there exists some let us say x prime such that f of x prime is the set Y I want to emphasize again that the existence of this x prime is because your f is an on to map on P of x and capital Y which you have defined in this manner is a subset of x and therefore it is a element of power set of x and since f is a on to map on this set there would be some element of the domain that domain of course is the set x itself capital X some element x prime of an element of X such that f x prime is Y, alright?

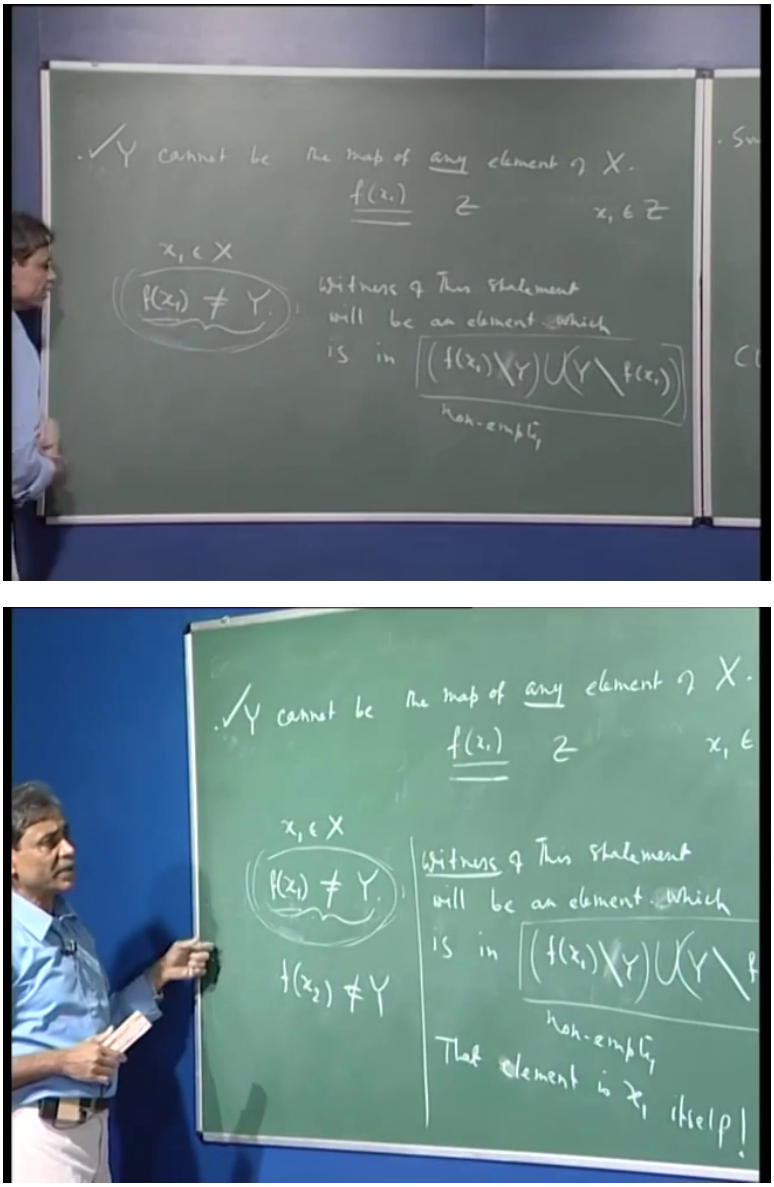
So this is straight forwards from the use of our definition of bijection and now consider the question is x prime an element of Y? right? x prime is a specific element if capital X and Y is a subset of capital X so it is meaningful question that is this element an element of Y. Now suppose if there are two possibilities either x prime is an element of Y or it is not an element of Y so we are taking the first situation suppose X is an element of Y, so now what does it is what is it saying that remember we are saying since X prime, right? We are asking this question is x prime an element of Y.

So x prime assume this so x prime an element of Y which of course Y by definition is f of x so I have x prime is an element of f of x prime but if x prime is an element of f x prime it cannot be in the set Y because in set Y you are only choosing those x's which are not elements of the set that x is mapped to by the bijection, so this really implies that x prime is not an element of Y, right? So as starting from the assumption x prime is an element of Y we are getting to the fact or the conclusion that x prime is not an element of Y so this is a contradiction so therefore this opposition is wrong the other what is the other opposition that x prime is indeed an element of is not an element of Y, right? Here we assume x prime is an element of Y and here we are saying x prime is not an element of Y.

Capital Y is by definition f x, right? So therefore we have x prime is not an element of f of x. Now if x prime is not an element of f x prime in that case you just look at this definition for Y this implies that x prime is an element of Y. Now here again the situation is starting from this opposition we come to the conclusion which is just the negation of that opposition. So there are only two possibilities either x prime is an element of Y or x prime is not an element of Y in both cases we are reaching contradiction which therefore means that our original assumption that there is a bijection f between the set X capital X and its power set that must be fall so this whole thing implies f cannot exist and therefore we have proved that there is no bijection between X and its power set, alright?

If you see what is really happening in this proof the crucks of the proof is in the definition that we made for Y and this definition made sure that no.

(Refer Slide Time: 40:31)



So basically the definition made sure that Y cannot be the map of any element of X, right? This is what we did. Now this statement that Y is not or cannot be the map of any element X you may think of I am just trying to explain how did you get to this definition and because even what our Turing machine context we will use similar arguments we need to be clear about the way these proofs diagonalization proofs work.

So this is something you were trying to prove that construct some Y such that this Y cannot be the map, what was the map we said? f the bijection map of any element of X, so basically we wanted to show that take element X of and we wanted to ensure that f of x is different from Y,

right? Now these are two sets, right? f of x is a set because f maps so f of x is a set of elements from x and this is also a set of elements also.

The way we achieve for a particular so here again consider a particular x let us say X1 and we want to ensure this condition that f x1 is not same as Y. So this statement is will be correct if for every element x1 we managed to do this, now two sets are unequal can be two sets can be unequal because either this has an element which this does not have or this has an element which this does not have, right? That is how to we make we can say two sets are not same.

So the width we can imagine the witness of this statement will be an element which is in the symmetric difference of these two sets, right? That is which one that element which is either here and not here or which is here and not here. An interesting thing that we did that we made x1 itself to be that element to be that witness, so let us write this sentence fully witness of this particular statement will be an element which is in that means all the elements which are in f x but not in Y union symmetric. So basically what is this saying? That this part consist of all elements which are in f x1 but which are not in Y and this consist of all elements which are in Y but not on f of x1.

So now this entire set if it is non-empty that would mean this non empty would imply this particular assertion that these two sets are not same and the way we have defined this set Y that we make x1 to be that element which is in the symmetric (())(45:28), right? How? Because you take your map, right? And see what x1 gets map to this is a particular set so just call it z, right? So f x1 is of course a subset of capital X which you are calling z, right? Now this element take this element x1 we are asking this question is in z? Right? If it is in z we will not put it in 1, right?

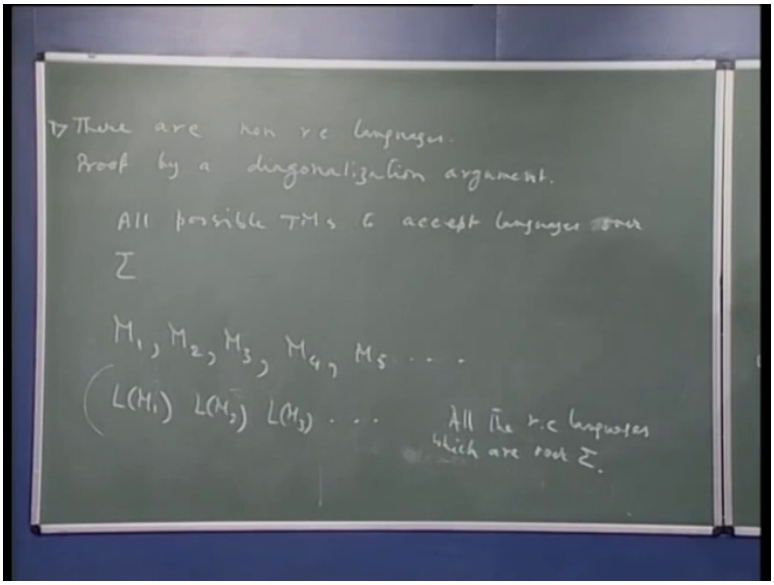
So in that case if x1 is in f of x1 in that case x1 will be here but not in here because by this definition since x1 is an element of f x1 we will not put it in Y so this set will be non-empty because x1 will be there or if x1is not an element of f x1 then we will put it in Y so this part will be non-empty in either case x1 is the witness of this. So we are taking every element and making it a witness of the fact that our constructed set is different from that subset of x to which x1 gets map to by a our assumed bijection and thereby we are you know proving that this particular statement is true, alright?

So again in summary what is the summary of what I am trying to say here if you see really a diagonalization proof you will see it is trying to proof non-equality maybe of two sets and this non-equality like here we said that you know we tried to proof whatever be the statement that assertion we broke we could break it down to maybe an infinitely many simple assertions like here that for any element take an element x1 make sure that f x1 is different from Y, right? And each one of these you ensure in defining the diagonal set this Y will be called a diagonal set, alright?

So we will see as I you know when you see for the first time such an argument it looks complex but if you go back examinate once more in the light of what you wanted to proof and the basic strategy of the proof so you wanted to proof in this case that there is no bijection between a set and its power set and you your proof idea was assume there is a map and then you constructed a particular subset of x such that it was not in the range of f.

So by that how did you do that? That this Y is different from the map of every element. So all these conditions one by one or you know conceptually you can see that for this condition x1 is the witness for another let us say x2 this x2 itself will be the witness and so on, so let me just complete that this case that element which is the witness, okay. So this is an example of a diagonalization proof this is fairly complex proof if you see if you are seeing it for the first time and it is a very powerful at least this particular example proof of diagonalization makes a very powerful statement we did not make any assumption about the set X capital X whose power set we are considering this X could be finite, could be infinite and even in infinite we are not saying that it is enumerable or not. So any set for any set the statement stands that there cannot be bijection between that set and its power set, okay.

(Refer Slide Time: 51:00)



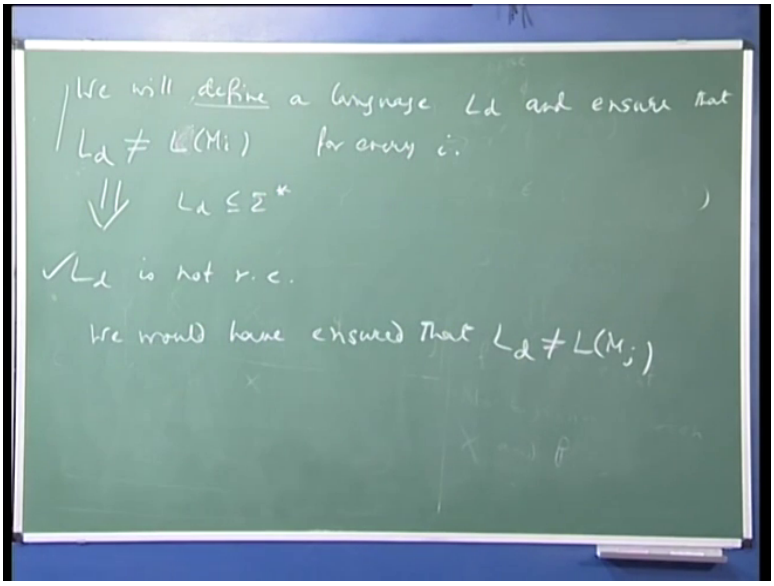
So now what is the statement that we would like to prove in this course about Turing machines is by this method first statement that we will prove is that there are so this is what we will prove that there are non r e language again the proof will be proof by a and let me give you the proof first of all from a very overall point structure first of all we will see that you know our case is simpler than the previous one in the sense that here everything is at least our Turing machines the class of Turing machines will be enumerable.

So what I mean by that is I can like up all possible Turing machines all possible Turing machine to accept languages over sigma. So this all possible Turing machines to accept languages over sigma they can be kind of lined up that means what I mean is I can the word enumerate, okay so the technical word but think of this way you it is possible for you to conceive of all these Turing machines to be in some sequence M1, M2, M3, M4, M5 and so on, right? And each one of these Turing machines accepts some language so the language accepted by this is of course L(M1) this is L(M2), this is L(M3) and so on.

And now you want to proof that there exist non r.e languages, if a language is r.e to accept a which is this which is a language over sigma then this has to be one of these languages, is it clear? Because these are the only Turing machines which will accept languages over sigma and these are the corresponding languages which are recognized by each of these machine. So essentially these are all the r.e languages which are over the input alphabet sigma.

Now in the same style that the way we created here a Y and made sure that Y is different in some way or way particular way what we are going to do here is to define a language we will call it Ld such that Ld is not L(Mi) for any i.

(Refer Slide Time: 54:53)



So we will ensure we will define a language Ld and ensure that Ld is not same as for every item, right? Once we manage to do that if I can manage to do then would immediately imply that Ld is not r.e, why? Because had it been re this would be one of these so in particular let us say had it been re it may be recognized by some M star which is somewhere in this lineup, so let us say this machine Mj suppose it if Ld was r.e then they would have been a machine in this enumeration of all Turing machines to accept languages over sigma let that be that machine be Mj the corresponding language would be Lm j but we would have ensured since we are doing it for all i we would have ensure that Ld is different from L Mj and thereby we prove again by contradiction that Ld is not r.e.

So two things that I need to do which we will do in the next hour, first of all prove this that Turing machines which accept languages over sigma that class of Turing machines can be effectively enumerated which means I can line them up one after another in some ordering so I can talk of ith Turing machine, i plus 1th Turing machine, nth Turing machine in that ordering and if at all there is a Turing machine to accept a language over sigma that will come somewhere here in this lineup and then I would define this language Ld which by definition will make sure

that Ld is different from all these Lm i and of course our Ld the way we will do it of course it will be a language over sigma and therefore Ld is not an r.e language over sigma, okay this is for a fixed alphabet sigma we will carry out all these discussion.