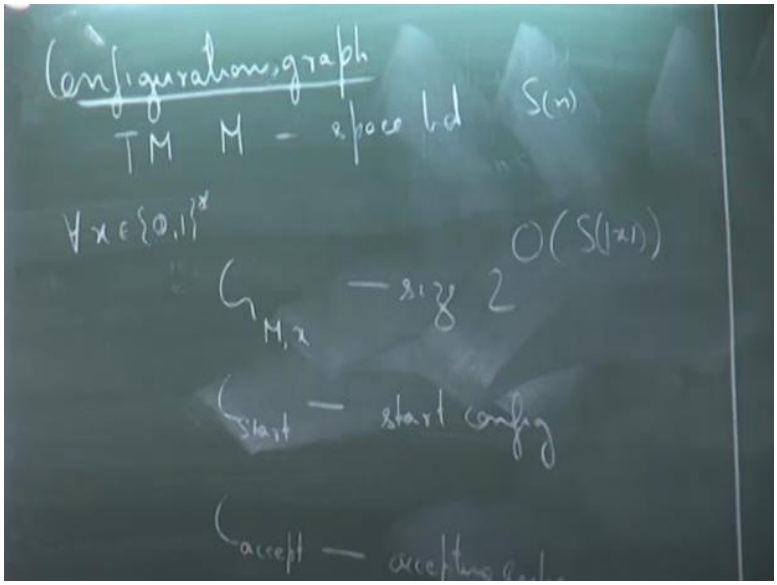
Computational Complexity Theory
Prof. Raghunath Tewari
Department of Computer Science and Engineering
Indian Institute of Technology, Kanpur

Lecture -07
Introduction

So last time we were talking about configuration graphs.

(Refer Slide Time: 00:27)

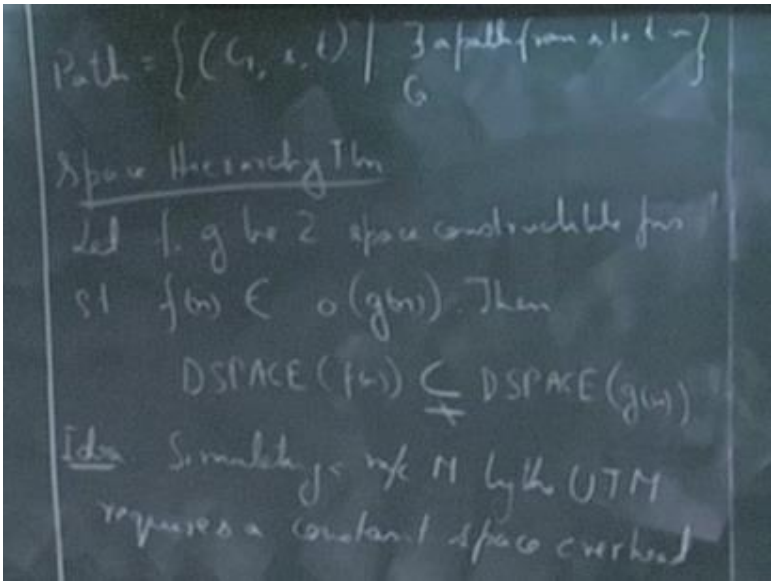


So we saw that for a Turing machine M that has a let us say a space bound some s of n. For any input x so for we can construct a configuration graph that we denote by G M, x and this has size how much? 2 raised to order s of the size of x. So how define this graph and under certain assumptions without loss of generality of course. We can assume that there exists a unique start vertex in this graph or there exists a unique vertex which we call C start.

So this corresponds to the start configuration and so I am just using the same terminology to denote both configuration and vertex. Instead of denoting a vertex by a separate notation c start denotes the start configuration as well as the vertex in G M, x corresponding to the start configuration and we also have a vertex C accept which corresponds to the accepting configuration.

So a general Turing machine can have many accepting configurations but last time we saw that how without loss of generality we can reduce it to just one accepting configuration. And the good thing about this thing this graph is that if x belongs to the language of this machine then there is a path from C star to C accept and otherwise not. So that is the property that we wanted to exploit. So keeping the same flavor of discussion, let us define this language which is a long similar line so I will call this path.

(Refer Slide Time: 03:25)



So this is somewhat more abstract. So the language path consists of all three tuples of the form. So a graph G a vertex s in G and another vertex t again in G such that there exists a path from s to t in G. So it is the all tuples of the form G, s, t such that there exists a path from s to t in the graph G. So, we will call this language path and we will, so this language and other restrictions of this language.

So when I say restriction I mean that I can restrict the class of graphs. So that will keep occurring in various forms along this course. So this is a very important language. So let us define a hierarchy theorem again analogous to the time hierarchy theorem that we discussed last week for space bounded classes. So it is called the space hierarchy theorem. So let f, g be 2 space constructible functions such that f of n belongs to small o of g of n.

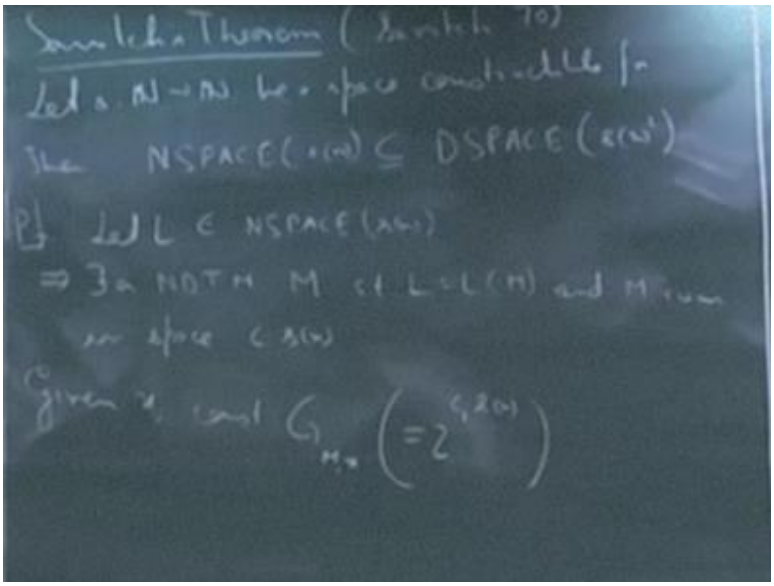
Then the deterministic classes defined by these two functions have a proper contain that is d space f of n is properly contained in d space g of n and the proof of this theorem is exactly the same as the proof of the time the time hierarchy theorem the reason. So if you recall the time hierarchy theorem there we had a log factor over here so fn times log of fn is contained was contained in the class of functions of small of g of n.

Then we had such containment and the reason for that was that to simulate a Turing machine using the universal Turing machine a log factor I mean, there was a factor of log occurring as a time overhead but if I want to simulate a Turing machine using again the universal Turing machine there is only constant amount of space overhead. So that is the reason why we do not have any additional factor here.

So I will just write the idea here simulating a machine M by the universal Turing machine requires a constant space overhead that is if m runs in some space s of n. If I simulated by the universal Turing machine it will run in space some c of s of n and the rest of the proofs is the same as the time hierarchy theorem. So I would not go into the proof of this. So what do we want to discuss next? So let us move on.

So we will now talk about a very important result in space complexity and this is something that we do not see in the case of time bounded complexity classes. So this theorem is known as Savitch's theorem.

(Refer Slide Time: 08:49)



So this was as the name suggests this was shown by guy named Savitch in 1970. So what this theorem says is that you can simulate the computation of any non deterministic space bounded Turing machine using a deterministic space bounded Turing machine with only a quadratic increase in the time I am sorry using only a quadratic increase in the space bound. That is so let s be a space constructible function, so which means that of course s is greater than log n s of n then NSPACE s of n is contained in DSPACE s of n square.

So this is something very counter intuitive I mean because if you look at the time classes we do not know if NP is contained in P or not. I mean had such a result been true for the time classes that would immediately show us that NP is content in P, which actually we believe is not true. So this is something which is very, very counterintuitive when the moment we go to space bounded classes.

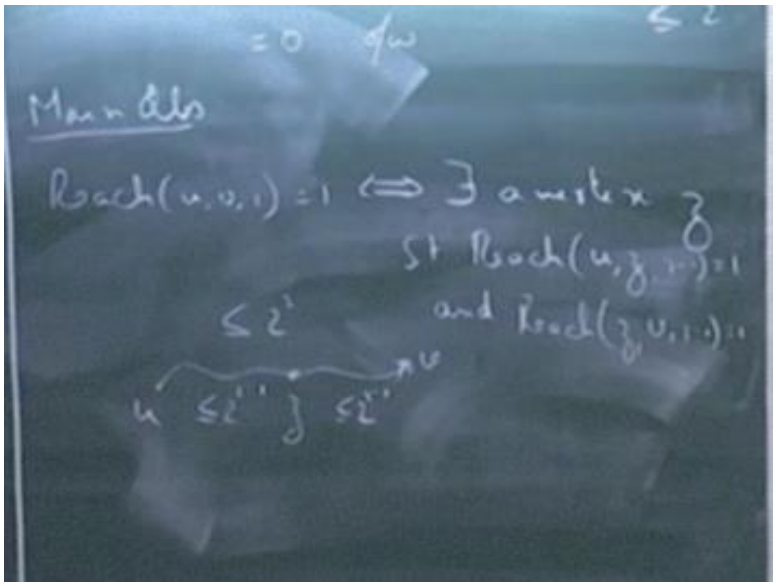
We have something and as we shall see this is not very difficult. I mean the idea of this proof is quite intuitive and quite nice. So let us look at the proof. So we will take some language L in NSPACE s of n. So let L be a language in NSPACE s of n. So what does that mean? So it means that, there exists a some non deterministic Turing machine M such that L is equal to L of M and M runs in space.

So let us say there is some constant C and it uses space C time's s of n on an input of length n. So what do we do next? We look at the configuration graph of this machine on any input x. So given an input x so given x so basically the goal is to construct a deterministic algorithm that uses at most this much amount of space and decides if any given x belongs to this language or not.

So given an x what do we what we do is we construct G M of x; I am sorry G M, x that is the configuration graph of this machine on input x. So how many vertex how many vertices does this graph have? So it is some 2 raised to the power order of s of n. So let me just denote it by this is some 2 to the power let me call it c 1 times s of n and so now we have reduced it to the configuration graph.

So, all we have to do check whether x belongs to the language or not is to check if there is a path from c start to c accept in this graph or not. So we will give a recursive algorithm for this problem. So let me define a predicate.

(Refer Slide Time: 13:57)



So we will define a predicate reach u, v, i. So, u and v are two vertices and i is some positive some non negative integer. So reach u v i is equal to 1. If there exists a path from u to v of length less than or equal to 2 to the power i and we say that this predicate evaluates to 0 otherwise either

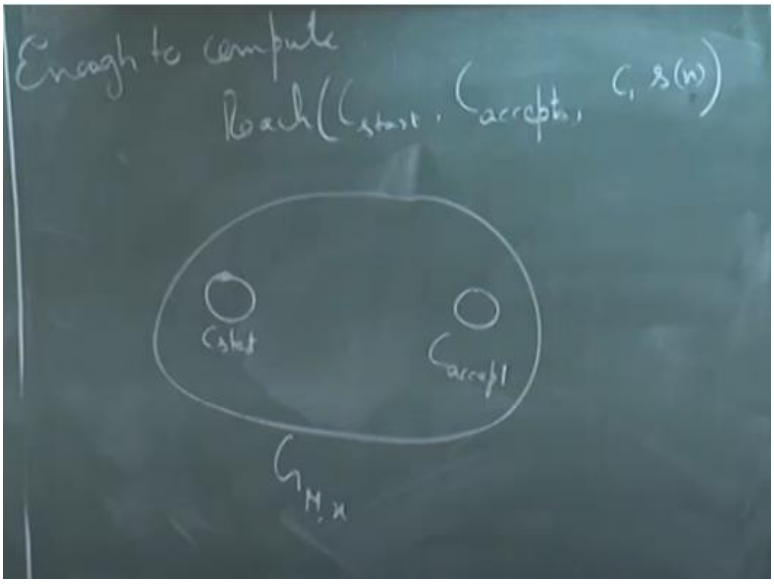
there is no path or there is the shortest path has length greater than 2 to the power i. So let us define an algorithm for this.

So before I give an algorithm, how to compute this let us look at an important property that this predicate has and that is the main observation behind this theorem. I will call it the main observation that is so reach u v i is equal to 1 if and only if there exists a vertex z such that reach u, z, i - 1 is equal to 1 and what, so reach z, u, i - 1 is equal to 1. So in other words what we are writing down here is that there exists a path from u to v of length less than 2 to the power i if and only if.

There is some vertex z in the graph such that there exist a path from u to z of length less than 2 to the power i - 1 and there is a path from z to v of length less than two to the power i - 1. In other words; if you look at any path from u to v. So, suppose this is my vertex v and this is my vertex u there is always a middle vertex. So that is the vertex which I will choose as my z so that if there is a path of length at most 2 to the power i - 1.

So this big thing has length less than 2 to the power i then these two small pieces will have length less than 2 to the power i - 1 each and this is the fundamental idea. So is this clear to everybody? But the shortest path between any two vertices will have length at most 2 to the power c 1 s.

(Refer Slide Time: 17:29)



So now so the claim is that it is enough to compute the predicate reach C start C accept with my i as c one s of n. So where n is the length of the input so we started with an input. So n is just my length of input. So if you are more comfortable you can write instead of modulus of x. It is the same thing. So is this clear to everybody, yes? Which part? So what do we mean by this predicate? So let us look at the definition of this predicate. So this is the definition.

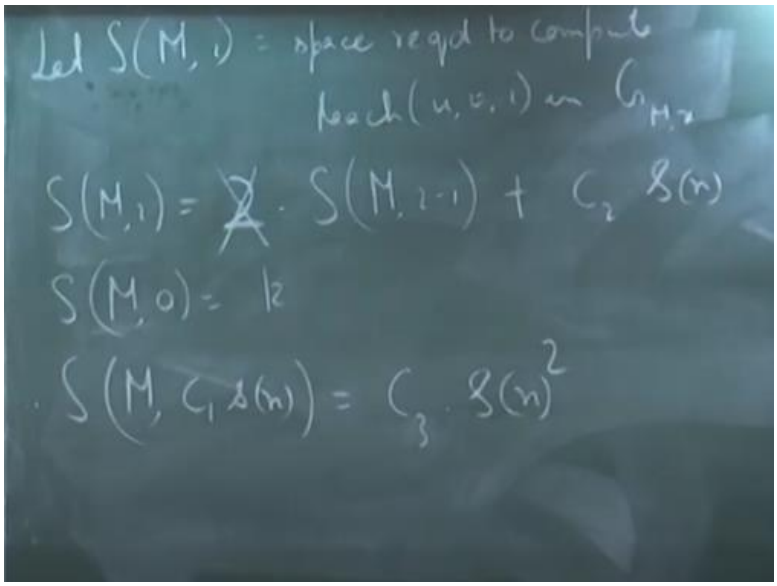
So in a graph g with two vertices u and v and some non-negative integer i, I claim that this predicate returns a one if and only if there exist a path from u to v of length at most 2 to the power i. So now if I look at this graph G M x, suppose this is my graph G M x and I have a c start sitting somewhere in it and I have c accept sitting somewhere. So all I want to do is check if there is a path from C start to C accept.

And, the claim is that if there is a path from C start to C accept there will be a path of length at most the number of vertices in the graph. I mean the path will not repeat any vertex because if the path repeats a vertex I can always construct a shorter path. So if there is a path of length at most 2 to the power c 1 s of n then if you just look at the definition of this predicate this will evaluate to true if and only if there is a path.

Clear? Any other questions? So now we have an algorithm that would solve the problem. The only thing left is to find out how much space it is taking. So this is basically our algorithm. So

we just feed the following parameters and we get the correct answer. So is everybody convinced that we will get the correct answer. So how much space it takes?

(Refer Slide Time: 20:34)



So let us did I use capital s somewhere? So let S M, i be equal to the space required to compute reach u v i in G M of x. So I am just; well M is not it is not so important to keep the parameter M because we always have a fixed M it is more important that we put i as a parameter of this function by which we denote how much space the machine is taking. So what do we want to how can we write a recursive relation for this function? So, S M, i.

So, how do we write a recursive relation in terms of lower values of i? Look at this algorithm two times S of M, i - 1, anything else? So what else we are doing? So of course we are making two calls to this predicate but also we are running a loop basically through all the vertices of the graph. So basically to check if such a thing is happening so how many bits do we need to represent a vertex of this graph? c 1 s n.

So let me just write it as some constant c 2 times S of n. So basically it is some order of S of n. So is this the correct recursion? So the point is that we are talking about space and the first thing that I said yesterday when we started discussing about space complexity is that space can be reused. So actually we can do away with these two so this two is not required. So when I am

computing these two predicates after I compute the first predicate I can just reuse the space to compute the second predicate.

So that is the good thing about space. So this is how the recursion starts and of course the base case is not very difficult. So S M, 0 is some constant I mean whatever space you need to look at the table. So therefore S M, so what is the; i that we are starting off with c 1 s of n is how much? It is basically some c 1 times c 2 times S of n square plus some k maybe. So I will just I will just write it as some c 3 times so this actually this was the small s.

So which is order S of n square. That proves Savitch's theorem. So that is a good point. So you can think it in many ways. So one way is to think of is a reduction. So given the input I just reduce it so basically construct the graph and then I am using the graph. So we will talk more about this form of reductions very soon. The other way to think of it is that I do not construct the graph at all.

So I have my input there so that x is given on my input tape so whenever I need a vertex so that is whenever I need a configuration of the machine on the input x I just look at the input and I construct that vertex from there so even without explicitly constructing the graph. So we will just construct the graph on fly basically whenever we need a vertex we will just go to the input will construct that whatever is the value of that vertex and then we will proceed.

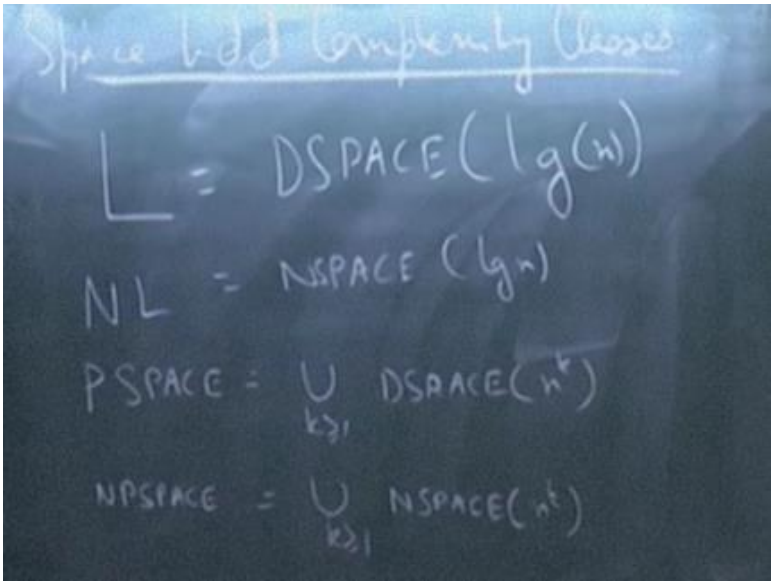
We do not care about time. So the only thing so the so that space should be included but as you will see that it would not be so how much extra space do we need? Well, we are not constructing the whole graph so that is the thing. So we have the input so yesterday or not yesterday day before yesterday we saw that so given an input how we can construct G M of x? So that takes only log space.

So just think about it, so that will take only log space but may be the output of this construction that is writing G M, x completely will take space much more than log of n but the thing is that so we would not write G M, x on our output tape at any time. So whenever I need a particular

vertex I will just go look at the input I will construct that vertex. Since the output tape is only a write only tape I never refer to the output tape for any information.

I only write out bits onto it. So I will just find out what is that vertex with to which I am referring and I will just use it so yes so some of these constants may I mean will become a little larger. But the main function will still be the same. Any other questions? So now let us look at some complexity classes that we can define for some fixed functions.

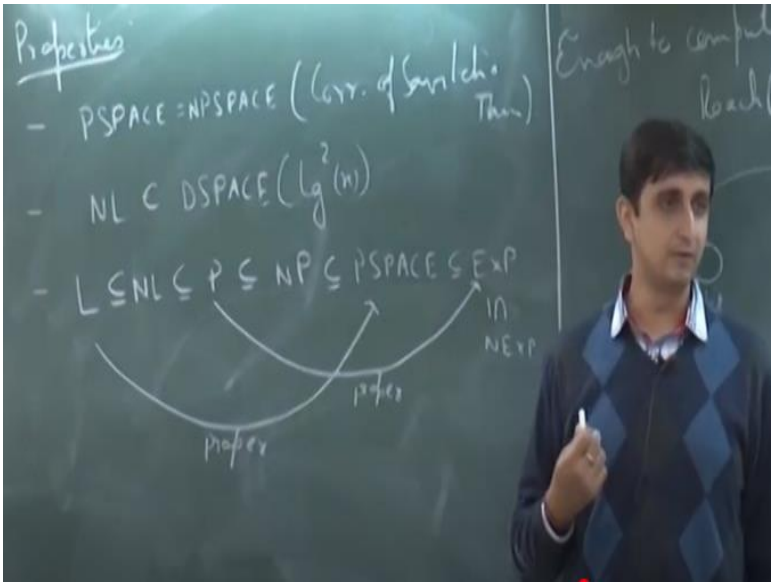
(Refer Slide Time: 28:07)



So we have the class L which is D SPACE of log of n. So if we choose our function to be log n so analogously we have NL which is the non deterministic variant of this class N SPACE log n. We have the polynomial variance of again these space classes. So we have P SPACE which is union over all k greater than or equal to one. D SPACE of n to the power k and we have NP SPACE which is union of k greater than one N SPACE n to the power t.

So here it does make sense to talk about functions which are sub linear. In other words things like log n because we can do as we saw we can do certain things using this much amount of space. In the case of time it did not make sense to talk about sub linear functions so what are so what can we say about these classes?

(Refer Slide Time: 30:07)



So let us discuss some properties. So can we deduce anything about these two classes from Savitch's theorem? So, P SPACE is equal to NP SPACE. So this is just a corollary of Savitch's theorem. So in fact this is what I was mentioning earlier that in the case of polynomial sized functions the space classes for deterministic and non-deterministic machines actually coincide and this is again and this thing does not happen in the case of time bounded classes.

So what tells can we say? NL belongs to D SPACE log square n. So this again holds by Savitch's theorem. What can we say about the relations between all the classes that we have seen so far? So let us start with L; of course L is contained in NL. By definition what can we say about NL and P? So last time we saw a theorem that we can simulate N SPACE s of n using D TIME 2 raise to order s of n.

Basically looking at the configuration graph and doing a reachability using some linear space reachability algorithm. So we can do NL using I mean we can put NL inside P by that same result. So P is contained in NP, what about NP and P SPACE? Why is that? So look at any complete problem for NP. Let us say you look at SAT. How can you solve SAT? By a P SPACE algorithm.

So what is the brute force deterministic algorithm that you know for SAT? Basically just cycle through all possible assignments of your variables and if there is some assignment for which the

formula gets satisfied you accept otherwise not and how much space are you taking for this? You are taking only order n space, take exponential time but you have to be careful this amount of space that you are using is only order n basically for remembering each and every setting of the variables.

So that proves that NP is in P SPACE; P SPACE is in EXP again this is by that same result that we saw last time that is N SPACE S of n is in D TIME 2 to the power order of S of n and this is in any NEXP. So these are sort of course P SPACE is equal to NP SPACE. So I am just not writing them separately. So this is kind of what we know currently and as it turns out so maybe we can say a little bit more about what we know right now.

For example we know that L is properly contained in P SPACE. So this is a proper containment by the space hierarchy theorem. So because this is a log n and this is union overall n to the power k and by that same argument we know that P is properly contained in EXP. So this is also a proper containment and that is all that we know about these classes as of now, if you want to study relations between these classes this is all that we know.

So, since L is properly contained in P SPACE what we know is that at least one of these containments have to be proper and similarly since P is contained in EXP at least one of these containments have to be proper but we do not know which one and in fact what most people believe is that probably all of these containments are proper but we do not know. So let us look at so we said that it does make sense to talk about log space classes. So let us look at a problem that lies in NL.

(Refer Slide Time: 35:53)

So recall the problem that we defined towards the beginning of this class that is the path problem. So we can show that path is in NL and the proof is again not very difficult. In fact I will just give an idea of the proof because I can just fill in the details later on. So the idea is very simple. So what you do is so you are given a graph g and you are given 2 vertices s and t. So starting at s just guess a vertex in the graph that is reachable from s.

I am sorry guess a vertex in this graph to which there is an edge from s. So you start at s and maybe it has many outgoing edges. So guess any 1 vertex and verify if it has an outgoing edge from s. Let us say vertex v. Now set that as your current vertex and carry on this algorithm. o initially s was the vertex at which you started you guess a new vertex check if there is an edge from s to b using the adjacency matrix or whatever representation of g that is given to you set v as your current vertex and do the same thing again.

So you said v as your current vertex and among let us say all the vertices that are reachable from v you guess one and you continue let us say some let us say this was v one I go to some v two and you continue in this manner for n steps, where n is the size of the vertex set if within this time you have reached t then you just halt your computation and accept and if you have not reached t then then again you halt and you reject.

So that is the algorithm so if t is reached halt and accept else reject in fact if t is reached within anything less than n steps you alt n accept it can come of course you can have a cycle in the graph and you can get a previous vertex also but again right so that particular computation would not be in n step so you are right. So that is the point so that is a good point that you raised so suppose you have a path from s to t so suppose this was your s and there is some path from s to t.

So we just argued a while earlier that if there is a path from s to t in the graph there is a path of length less than or equal to n but suppose along some computation it guesses a cycle in which case the path will be of length more than n but the point is that again there will be some guess along which it will guess the shortest path. So there can be some guesses which guesses this cycle in which before n steps it will not be able to get t.

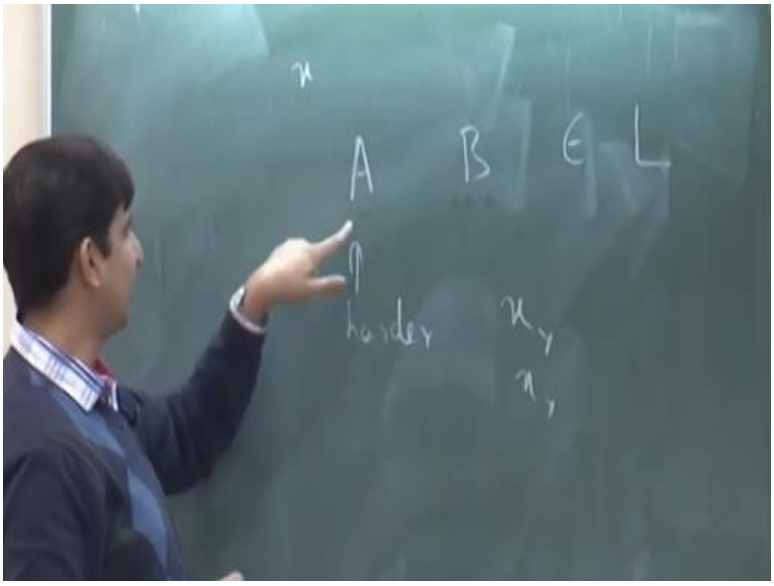
But there will be at least one computation which will guess the shortest path from s to t and that will be within n steps and as a result you would accept. So is that clear so the definition of a non deterministic algorithm is that there is some sequence of non deterministic choices that will lead you to an accept state, there is at least one sequence there can be many, so now the thing is that why is it taking login space.

So how much space does this algorithm use so what are we doing so firstly we need basically yes we need a variable to store the current vertex and maybe another variable to store which is the next vertex but these two variables can be reused so the moment I go to a new vertex I will set that as my current and I will find another vertex. So I will probably need two variables to store the vertices that I am using.

And I will also need a counter to record how many steps I have moved okay and that is so if i have n vertices in my graph the counter will need only log n bits. So basically i have about three or four variables therefore the total space taken by it is order log n. So the point is that whenever you are guessing a path you need not remember what are the previous vertices along that path that you have taken?

The moment you go to a new vertex you just forget the previous one but each time you have to check I mean each time you have to go to the adjacency matrix of the graph and check if the new guess vertex actually forms an edge with the current vertex so that check can be done very easily and that gives you a log space algorithm non deterministic log space algorithm is that clear. So now we will define the notion of log space reduction, so this is what I was telling earlier.

(Refer Slide Time: 42:06)



So we say that a function f from 0, 1 star to 0, 1 star is log space computable. So observe that now we are looking at functions which output strings actually, they are not just boolean functions is log space computable if there exist a log space bounded machine M with, so I will just write this within bracket with a write only output tape such that on any input x M outputs f of x on its output.

So the output tape has this property that it is only a right only tape so that is that means it only moves from left to right so after it writes out a bit it can only move to the to its next cell, it can never come back and read something. So now we say that a language or we write it just say that l subset or l 1 subset of 0, 1 star is said to be log space reducible to some other language l 2 if there exist a log space computable function f such that for all x in 0, 1 star x is in l 1 if and only if f x is in l 2.

So in other words basically I have a log space machine m which given any string x will output f of x on its output tape and if x belongs to the language l l one then f of x belong will belong to l two ~~ah~~ and vice versa. So the question is why do we need log space reduction so already we had a notion of reduction that we defined earlier that is polynomial time reduction but why do we need log space reductions any idea?

So let me just say that slightly more elaborately so what is the idea of reduction why do we need reductions exactly so when we want to show that some language is at least as hard as some other language. So the idea is that if a reduces to b it should kind of maintain the property that a is at most as hard as b. So suppose now let us take two languages, let us take a language A and let us take a language B.

Both belonging to let say the class l so they are languages which are quite small and suppose that A is much harder than B, so A is probably harder than B. So this is much harder to compute okay suppose we only had polynomial time reductions in our toolkit, what we can do is that we can always reduce A to B. How do we do that? So I pick a yes instance of B and I pick a no instance of B.

So let us say that I have some x yes and I have an x no I take any string x and I just run the algorithm for A and solve whether x belongs to A or not. So if x belongs to a I will map it to x y in B and if x does not belong to a and I map it to x y which is in B and since I had initially assumed that both these languages belong to l the algorithm that i am running to check whether x belongs to a or not is a polynomial time algorithm.

So in fact I can also assume that A and B let us say belongs to NL, so basically what I have shown is a harder language can actually be reduced to an easier language if my reduction is so very powerful that it can basically cheat the underlying classes. So that is the reason why our reduction should be weaker in power than the classes that I am looking at. So that is the reason why we need to go to log space reducibility. We will stop here.