Theory of Computation
Professor Subrahmanyam Kalyanasundaram
Department of Computer Science and Engineering
Indian Institute of Technology, Hyderabad

W12L65_Logspace Reductions and NL-Completeness

Hello and welcome to lecture 58 of the course Theory of Computation. In the previous lecture, we saw space complexity; we saw the classes L, NL, and PSPACE. We also saw relations between time-bounded complexity classes, like generic time-bounded complexity classes such as D time something, N time something, D space something, and so on, and the space-bounded complexity classes. In this lecture, we are going to talk about a notion of completeness for space-bounded complexity classes called NL completeness. So, this is similar to NP completeness, but here we are in a complexity class which is in NL. As we already saw in the previous lecture, NL is contained in P.

So, NL is much smaller than NP. So, NL is contained in P. So, this is, so just to give an idea of what we have seen so far, L is contained in NL; both of them are contained in P, and all of them are contained in, sorry, NP, right. So, now, NL completeness is actually talking about a notion of completeness, but not in NP, but in NL, right.

And then later we will see that the problem PATH is a complete language. So, what is PATH? PATH is, given a directed graph G and two vertices S and T, we have to determine whether there is a path from S to T. There is a directed path. So, all the edges are directed edges.

So, that is the question PATH. Given a graph and two vertices, is there a path from the first vertex to the second one? That is the problem PATH. So let us first see, first let us see whether this language is in NL, and later we will define the completeness notion formally. So PATH is like asking whether there is a path from the first vertex S to the vertex T. Clearly, this can be done by all the traversal algorithms like breadth-first search, depth-first search.

But it requires linear time, also linear space. So, it is in P, but it requires linear space. But NL means non-deterministic log space. We do not have linear space. We only have log space, logarithmic space in the size of the input.

But we have non-determinism. So, P does not have non-determinism, but NL has non-determinism. So what we'll do is we'll just directly explain the algorithm where we will take a guess-and-verify kind of approach, right? So we had seen guess-and-verify approaches for problems in NP, et cetera, but here we'll do something similar, not explicitly guess-and-verify, but something that roughly does the same thing, right? So this is the following algorithm. So we have, in the algorithm, we have some variables called v, is a

variable called w, and there is a variable called count. So, we first set v to be the start vertex S.

So, we have to determine whether there is a path from S to T. And the count we set to be 0. And we have this loop that runs from starting from count equal to 0 to count equal to n. So, what we do is we every time we guess a next vertex w. So, what we are doing is we guess a next vertex; maybe I will just indicate that in some other color.

So, we have the vertex S. So, we guess some vertex here. So, this is W. And we check whether there is an edge from S to W. If there is an edge from S to W, we continue.

If there is no such edge, we stop. So, we reject. So, if V to W is an edge, then we continue. If it is not an edge, we reject. So, focus on this thing.

If it is not an edge, we reject. Otherwise, we continue. How do we continue? What we do is, now we assign V to be the value of W. So, now this becomes V. Now we guess yet another W.

Let's say we guess a new W. Now we again do the same thing. We check whether there is an edge from V to W. If it is there, we continue. Otherwise, we again reject.

So now after this, we again, if it is there, we now make this as V. And that is what we continue. And every time when we find a W and it is a successful W, successful meaning there is an edge from V to W, we update the count. So, every time, so the count will be 2 at this point because first we guessed one W which worked and then we guessed the second W, it worked. So, now we are also, the count keeps track of how many guesses we made.

It also keeps track of how far away we are from the start vertex, so in terms of number of steps taken. So, it may not be the distance because we might have taken a circuitous route to come to V, but it may not be equal to the distance, but then this is the number of steps taken. So, what we are doing is we have these two variables V and W. So, V and W, and we guess the next one, make this W, and then check whether there is an edge from V to W.

If it is there, now we update this to V and then we guess another one and so on. And at every instance, we check whether the vertex that is guessed, whether we have already reached T. So, at every instance, we check whether we are already at T. So, V is where we are currently now with confirmed edges at each stage. V is where we are at, but W is our next guess.

So, we know for sure that there is a path from S to V, but W we are just kind of exploring whether there is a path from V to W. So, if at any point V is equal to T, we accept, meaning

we know that there is a path from S to V and suppose V is equal to T, we can accept. Otherwise, we continue this process. If at any point we make a wrong guess, let us say we guess this to be W the next time and there is no edge from V to W, then we reject. So, otherwise, if we are able to make a series of guesses starting from S and reaching T, then we accept.

It is a very straightforward approach. We basically, instead of making an entire sequence, instead of guessing an entire sequence of vertices leading to T, we are guessing the next vertex each time. And we are not even remembering what path we came from. We are just remembering V and W; we don't know how we reached V because if we were to keep track of the entire path, the number of vertices in the path could be something like n by 2 or n or n minus three-fourths n or something that is already linear in the number of vertices. So, we cannot afford to store that much; the space that we are allowed is only logarithmic in the input.

Right. So, we just remember V and we know for sure that there is a path from S to V. We do not know how we reached there, but we know there is a path, right? And where we guess the next one and then check whether there is an edge from this V to W and then once it is there, now we update V to V with the value of W and then we guess the next W and so on. So, every time the W is guessed and if it is correct, if it is a correct guess, V is updated to that, and at any time if V is equal to T, we accept, meaning we have found a path from S to T. So it's fairly straightforward. If there is a path from S to T, one of the many possible choices of W's will lead us to that path.

If there is no path, then we are not going to accept anyway, right? And one more thing, how does count feature into all of this? How does the variable count feature into all of this? We only run this loop from 0 to n. We don't want to run this forever. And the point is that if there is a path from S to T, there is a path from S to T that uses at most n edges. There are only n vertices. So, if there is a path from S to T, there is a path that does not repeat any vertices.

So, there is a, like we may end up taking a circuitous path, like the path could be something like this. It could go around and then come back, all that. But here we may be looping on some paths, looping on some vertices, but then we can forget the loop. We can remove the loop and there will be a straight path from S to T without repeating vertices and that can only have at most n vertices because that is the number of vertices that are there. So, we just need to consider the paths that are of length at most n.

So, that is the algorithm. Count gives us a terminating condition because, till n steps, if we have not found the path, we can stop and reject. So the correctness of this is straightforward,

and the space complexity is, we need to store v, we need to store the count, we need to store w; these three things are what we need to store. So v is a vertex, w is a vertex, and there are n vertices, so each, the name of each vertex, or the label of each vertex requires log n space. Also, the count is a number from zero to n, so that also requires log n space. It is like 3 log n or something or 3 log n space which is again O(log n) space.

And it is a non-deterministic algorithm that shows that the path is in NL. So, here we have an NL algorithm, non-deterministic log space algorithm that determines whether there is a path from S to T. Just one more comment, why does breadth-first search etc. not work? Because this requires marking each vertex as visited, not visited, or something like that. So we need to have a flag at each vertex of the graph.

Since the graph has n vertices, these n flags themselves take up n, order n space. So BFS, NFS, sorry, BFS, DFS are already taking linear space. Whereas this, where we don't remember the path. So after the end of this, we just know whether there is a path or not. We do not even know which path we came from because we are not preserving that.

We cannot afford to preserve that because we are bounded by the logarithmic space. So, the path is in non-deterministic log space. Now, let me explain log space reductions. So, this is similar to polynomial time reductions. So, we say a reduces to b in log space and is denoted by this notation $a \leq b$ with a subscript of l.

So, in mapping reducibility, we had $a \leq b$ with subscript m. In polynomial time, we had a subscript b. Here, we have a subscript l. And the reduction is fairly, the notion of reduction is fairly similar. We say that a reduces to b in log space if there is a Turing machine that computes a function that gives us this correspondence.

For every string in A, you get a string in B. For every string not in A, you get a string not in B. So that something in A gets mapped to something in B and something not in A gets mapped to something not in B. The only difference is in the amount of resources consumed by the machine that reduces. In this case, it is a deterministic log space bounded machine.

In the case of mapping reducibility, it was just some Turing machine. We just needed it to be computable. In the case of polynomial time reduction, it was a deterministic polynomial time Turing machine. So here it's a deterministic log space machine. So it's actually, we have seen that log space is a subset of polynomial time.

So this is a more constrained kind of reduction. So something reduces to something in log space, or something reduces something in polynomial time need not imply that the reduction is there in log space. But since log space is in poly time, if there is a reduction in log space, the reduction is certainly there in polynomial time as well because we saw that log space is contained in polynomial time. So what I am saying, I will just note it here, a

reduces to b in log space implies a reduces to b in polynomial time. But a reduces to b in polynomial time does not imply that there is a reduction in log space.

So this is wrong. This is not there. So this is something that one has to be careful about because log space is a more restricted class. Everything in polynomial time need not mean that it is in log space. So this is the definition of what is a log space reduction. So we say that there is a, A reduces to B in log space if this condition is satisfied. If there is a log space machine that computes a reduction, then the reduction is standard.

So every symbol in A or every string in A is mapped to something in B and everything not in A is mapped to something not in B. Now with this, we can define NL completeness. A language is NL complete if two conditions are satisfied. So one is that B is NL complete if B is in NL.

So we need membership in NL. And all the languages in NL should reduce to B in log space. So this is exactly like polynomial time reduction and NP completeness. Just that wherever we have NP, now we are replacing it with NL. And wherever we had polynomial time reduction, we are replacing it with log space reduction.

In NP completeness, we said that B is NP complete if B is in NP and all languages A in NP are reducible to B in polynomial time. Here we say the same thing, B is in NL, language B is NL complete if B is in NL and all languages A in NL are reducible to B in logarithmic space. So, we have a more constrained reduction requirement, log space reduction is more constrained than polynomial time reduction. So, we are allowing fewer resources, logarithmic space which is actually less than polynomial time. So, maybe just food for thought: Would it make sense if we use polynomial time reduction here? Would it make sense if we use polynomial time reduction? Because we are using log space here.

And the answer is no, it will not make sense because NL itself is contained in P. So if any language is in NL, and if you are allowed a polynomial time Turing machine, we can actually decide that language. So the goal of reduction is to convert and then solve the other language. So we want to convert A to B and then solve for B. But if you have a polynomial time machine available to do the reduction, we can, NL is contained in P.

We saw it in the previous lecture. So maybe I'll just note it here. So no, we saw NL is contained in P. So, poly time can be used to decide. So you can just use the polynomial time Turing machine to decide A, like why to reduce and all that. So it's, and then anything is reducible to anything in polynomial time.

Like if A is an NL, we can reduce, and if you are given a polynomial time Turing machine, we can reduce any A to any B, right? As long as you have something to map to. So the

point is that the reduction, polynomial time reduction is too powerful, way too powerful. So, if you remember we had this discussion when we taught, when we saw NP completeness. We said that we cannot use mapping reducibility because we could solve SAT completely and then map it to like if it is satisfiable map it to 1 and if it is not then map it to 0. If you have a Turing machine, if you just insisted that it should be computable, the reduction should be computable and no time bounds were imposed.

Similarly, if we insist that the reduction is in polynomial time and not in log space, we can do, we can solve or we can decide any language that is in NL in polynomial time. So it makes the whole point of reduction kind of meaningless, right? So that is the reason why we need a more restricted reduction. So in general whenever we have a notion of completeness we want the reduction to be less powerful. So in NP completeness we use polynomial time reduction which is less powerful than NP. In case of NL completeness we use log space reduction which is less powerful than NP.

Now, let us come to the main objective of why we need reductions. So, in polynomial time reductions we had this like if A reducible to B and B is decidable in polynomial time implies that A is decidable in polynomial time. We can convert A to B then we can decide B. So, for instance the idea went like this. Suppose, so A to B conversion takes let us say n squared time, then B decider let us say f of w cubed time, this much time then the whole thing is in polynomial time because the reduction is in n squared time. So, the length of f of w is will be in n power 6 sorry length of f of w is going to be at most n squared because in n squared time we cannot write something that is longer than n squared and then B decider runs in cubic of that.

So, n squared cubed is n power 6. So, this implies that A is decided in n power 6 time because you have to take the cube of n squared. The B instance itself could be as big as n squared. It may not be as big, but in the worst case, it could be. And then the running time of the B decider is cubic in the length of the instance.

So, n squared cubed is n power 6. So, anyway the thing is n power some constant, it is still polynomial time. So, we have a decider for A in polynomial time. Now, can we do the same thing for, can we say the same thing, otherwise the reduction is meaningless. Given A reducible to B in log space and B is in log space, can we infer that A is in log space.

Let us try to do what we did for polynomial time reduction. So, in polynomial time reduction, we converted A to B, we wrote the B instance, f of w was written and then we ran the decider B decider on f of w. But now running the decider on f of w and then we gave the output and we saw that the entire time taken is a polynomial. So, let us try to do the same thing for log space reduction. So we have a machine R that reduces W to F of W or X to F of X.

So this is W. So W will be there in the input and F of W will be there in the output. And then we have a machine that decides F of W. So we are basically, these three tapes are part of the reduction machine. So this is the input of the reduction machine, this is the work tape of the reduction machine, and this is the output tape of the reduction machine. So this input tape is read-only, work tape is read-right, and output tape is write-only.

So as far as the reduction machine is concerned. Now once having written the output, which is f of w, so output is f of w, now we run the decider for B. We run the decider for B. So, which takes f of W as input which is read-only for it and you have a separate work tape. So, now the thing is that when we are combining these two, the reduction and the decider for B, this is reduction from A to B R and M is the decider for B.

Now, we have an input W in the input tape of A which is the input tape. and we have a decision coming out of, actually from M, which is the B decider, so which is the accept-reject. So the work is done in all of these three tapes, which is the work tape of R and output tape of R, which is input for M, and the work tape of M. So now the work tape of this combined machine takes all of the space. And how much space is occupied by all this? So we know this is log space, but what is the length of the output of the reduction machine? So what is the length of f of w? That's what I'm asking here. What can we say about the upper bound of f of w? We need to write down the f of w longs, or we need to write down f of w.

So thing is that we have a log space machine. So the number of configurations could be 2^(O(log n)), 2^(O(log n)). This means it can run for 2^(O(log n)) time, which means the output could be of 2^(O(log n)) length, which is n power constant length. So this could be as bad as n power c, so that c is some constant and n is the length of the input. So the problem is that now this combined machine considers these three as the work tape. And we know that this is already taking, this is already taking up to n power c space.

Sorry, this may take up to n power c space. So now this is no longer log space, this is polynomial space. So we need only O(log n) space and n power c is much bigger than O(log n) space. So this is not a log space decider. Combining it in this way, it is not a log space decider for A.

So we cannot combine it like this. So the problem here, if you think, I'll just pause for half a minute just to think about it. So the problem here is that we are writing down the entire reduction. We are writing down the entire reduced instance f of w, and that is what takes polynomial space.

We cannot afford to do that. If we ignore this, the work tape of R is log space, and the work tape of M is also, the space required by M is log n^c, which is again, so maybe I will just

note that down here. The work tape of M requires log of n^c, which is again O(log n) space. So that is also okay. The work tape of M is okay, and the work tape of R is okay.

The problem is only with the input/output tape, the output of R and the input of M. So how can we proceed with the reduction without writing down f of w in full? So that is the question. So we do not want to write down f of w. How can we do this in another way? So we do not want to write down this.

So, we do the following. What we do is we directly start running the machine M. So, we do not take, so the input is here. So, here I have said that the input is called x. So, the input is here, let us say the input is here. We start running the Turing machine M without having any f of w written here.

We start running the machine M. And of course, it will need to look at the input. Any Turing machine will need to look at the input. So, suppose it at some point needs the first bit, the first symbol. Then at that point, we need the first symbol, then we start running R, right? So now the input x is written here, then we start running R till the point where we output the first symbol of the output, right? Which is the first symbol of the input of M. And then M can continue its execution. Now, let us say at some point M needs the 10th symbol of the input, right? M needs the 10th symbol of the input, and at which point we again restart R.

So now we restart R and basically we do not really care about what was output here. We do not even store any of these things. We just run R till we produce the 10th symbol of the output, which is the 10th symbol of the input of M. And once we have the 10th symbol of the input of M, we continue M till the next symbol of f of w is required or f of x is required.

So this is the algorithm. We start M, which is a decider for B. Now suppose the input is x, whenever it needs the jth bit or jth symbol of the input, jth symbol of f, we remember the j, whichever, so j could be 10, it could be 20, it could be 15, right? And we run the reduction machine till the jth symbol is produced. We don't store the rest of f. We just throw them away, right? Because if we try to store, then we are using up all the space.

We don't store any of that. We just count how many symbols we have output. Once we reach the jth symbol, we tell M and then M uses the jth symbol and continues. So now what is happening is that this works perfectly well. But the only issue is that we are re-running R again and again. Instead of producing f of w once, whenever any bit is required, so first maybe let's say the 10th symbol is required, it runs till the 10th symbol. Later maybe the 20th symbol is required, it again runs from the beginning till the 20th symbol. Maybe next time the 3rd symbol is required, but it had already computed the 3rd symbol, but it's okay, we again compute the 3rd symbol.

Again now maybe the 26th symbol is required, again it computes. So it is doing a lot of work again and again. The reduction machine is being asked to do a lot of work again and again, but that is okay. This is because we are not paying in terms of time; what we are paying is in terms of space. So, the only space that is used is the work tape of R and the work tape of M. Here we do not use anything; instead, we just throw away all the symbols except the symbol that is required.

So we end up recomputing many things. This is what I said here. We may end up recomputing many symbols. But it's okay. We don't, the time that is taken does not matter to us. Only the space matters to us.

And the space used is, so we already said why. The space used by R is at most log n. The space used by M is also at most log n. And here things are transmitted symbol by symbol. So that's also at most log n. We are not going to write down f of w or f of x as a whole.

So this is the way to perform a log space reduction. And the overall space usage by these two combined machines, because of the modified way in which we combined this, is logarithmic in the length of the input. So, which means we have like combining R and M, the reduction from A to B and the decider for M, we have a decider for A. So that gives us the following result: if A reduces to B in logarithmic space and B can be decided in logarithmic space, then A also can be decided in log space. Now, let us finally come to NL completeness.

So, NL completeness, we said that we already said this, but I will just repeat. B is NL complete if B is in NL and all languages A in NL can be reduced to B in log space. So, in some sense, like we said for NP, this is like the hardest problems for the class NL. Maybe I will just note that down here. Because if any of these problems we can solve in logarithmic space, then we can solve the entire NL in logarithmic space. Just like we said that if we have a polynomial time algorithm for any NP-complete problem, we can show that P is equal to NP.

Just like that we can say that if for any of the NL-complete problems, we can get a log space algorithm, log space decider that implies that L and NL are equal. So, just like P versus NP, L and NL are also, L is contained in NL, but we do not know if they are equal, right? This is something that is also an open question, not as popular as P versus NP, but still if you ask computer scientists, they will say this is also equally important or quite important, right. Okay, so now let us move to the proof that the language path is NL-complete.

So, path given directed graph G and two vertices S and T asks if there is a directed path from S to T. So, we need to show that this is NL-complete. In the beginning of the lecture,

we have already seen that path is in NL. This showed that path is in NL. Now we need to show the second part, that is, all the languages in NL are reducible to path.

This is what we have to show. All languages in NL are reducible to path. So, the idea is very similar. The idea is something that we have already seen. If a language is in NL, it has a, we can, if you recall in the previous lecture, when we showed that NSPACE(f(n)) is contained in DTIME(2^(O(f(n)))), we built this configuration graph for the NSPACE machine. Similarly, we can construct a configuration graph of the NL decider for A. We construct the configuration graph of the NL decider for A and all that we are doing is we are going to check whether there is a path from the starting configuration to the accepting configuration.

I think I mentioned it in the previous lecture, but I will say it again. This implies or this assumes that there is a unique accepting configuration because there could be, otherwise there could be multiple accepting configurations. So it's a quick exercise to show that if there is an NL machine that decides a certain language, we can consider an equivalent NL machine that decides the same language, equivalent means same language, with the same NL bound which has a single accepting configuration. So, basically multiple accepting configurations we have to convert into a single accepting configuration.

So, this is not that difficult. So, we can just, whenever it accepts, we ensure that the work tape is wiped and all the heads move to the leftmost point. So, then there is only a single accepting configuration. So, this is something that you can work out the details of. So, we have a configuration graph. And we are, so the NL decider accepts the string, the input string, if there is a path from start state to start configuration to accepting configuration.

So this is a very straightforward thing, right? So the correctness is fairly evident. Now all that, so all that we are, and it's a path problem, right? So we are given a graph, which is a configuration graph. The S is a start configuration, T is the accepting configuration. So everything checks out. The only thing that we have to ensure is that the reduction is computable in logarithmic space.

So this small bit that A reduces to a path in logarithmic space is all that we have to assert. So we have already, just by the construction of configuration graph, etc. If A(w) is in A, then the reduced instance, the configuration graph, the start state accepting, start configuration, accepting configuration is a yes instance of path if and only if the string is a string in A. Now, what we have to ensure is the fact that the reduction is in logarithmic space. So, first of all, the reduction, let us see the space. So what is the number of configurations in the configuration graph? Each vertex is a configuration.

So we discussed something similar in the previous lecture, but I will go over it again. The configuration comprises of the head positions of the two tapes, the input tape and the work tape. So input tape has length n, so the head could be in any of the n locations.

Work tape has space log n. log n possible head positions. Tape contents, so each the work tape, tape contents can change. Input tape is fixed. So each cell could have any of the gamma possible symbols where gamma is a tape alphabet. So number of configurations is gamma^log n, where gamma is the size of the tape alphabet.

And finally, we have the state which is q. So the number of configurations is the product of all this. And you can easily see that this can be written as 2^(O(log n)) where all the other small constants etc. can be absorbed into this 2^(O(log n)). But 2^(O(log n)) is also like polynomial space.

Anyway, we just need 2^(O(log n)). This is the number of vertices in the configuration graph. 2^(constant log n). Now, that means that there are 2^(constant log n) configurations. So, each configuration has a label or a name of the size of the logarithm of that which is each configuration has a label of size O(log n). So, if we have 100 configurations, we need only 7 bits. If you have 2^(log n) configurations, we need, or if you have 2^10 configurations, we need 10 bits. If you have 2^(O(log n)) configurations, we need O(log n) bits to represent a configuration, right? So now, now we need to explain the construction.

Number is fine. Now we need to explain the construction, like how to get the, obtain the graph given the, given the NTM decider for, sorry, given the NL decider for A, right? So, we have the NL decider for A. What we do is the NL decider, it lists all the possible, sorry, the NL decider is there and we have the reduction machine that looks at the NL decider and the reduction machine looks at the NL decider and produces a configuration graph. So, maybe I will just note this here. List down all the configurations of the NL decider of A.

So how does it decide this or how does it list now? It just, so we know the number of configurations is O(log n). So it just cycles all the possible strings of length O(log n). So this needs a counter of length O(log n). So maybe I will call it counter 1. All right, sorry, that I'm using that in the next place. So it needs a counter of O(log n), and once you produce a string, we just check whether it's a legal configuration.

So we need to list, so we have listed down all the configurations at the end of this. If it is a legal configuration, we list it down. So remember, we are not being charged for the output. We are only being charged for the work. Now next thing is, now we have to, so we have listed down all the vertices of the configuration graph.

Now, we need to list down all the edges. So, we need to go over all the possible edges. So, how do we do that? This is also, there is a small error here. So, we first go through all the possible configurations, which are already listed down in step 1. And for each configuration, we list all the possible successors. So we have one counter that lists all the configurations or goes through all the configurations.

And for a fixed configuration, we now list down all the possible successors. So we list down all the successors and then check whether, if the configuration is C10, we check if C1 is a successor of C10. So no, we do not list it. C2 a successor of C10? Yes, so then we list it. C3? No, we do not list it and so on.

So we just list down all the possible, all the correct successors of the configuration. So we have two nested counters or nested loops. Counter 1 goes through all the configurations. Counter 2, for each fixed configuration, goes through the list of successors. There are two counters here. So, both of these counters require space of O(log n) because each one of them is going over all the configurations, but O(log n) plus O(log n) is again twice O(log n).

And this completes the specification of the edges and vertices. So, vertices were discussed in step 1, edges are described in step 2. And then now G has been defined with vertices and edges. Now what is S? S is the starting configuration and T is the accepting configuration. And now we are asking, instead of asking whether X is in A, we are asking whether the NL decider of A has a path from the starting configuration to the accepting configuration. So this reduction is valid and also the space required for these counters.

These three counters are the ones that require space, but maybe I will highlight those. And the space required by these counters is everything is log n.

So, even the sum is log n. But the output size is not log n. But we are not being charged for the output. The output is just written. And that gives the entire configuration graph, which is polynomial in n. So that we already saw here. The output size is 2^(O(log n)) or polynomial in n. So, I have explained the details here, but if you want a slightly more detailed version, you can refer to the version in Sipser.

So, you can read the detailed proof in Sipser if you want to still understand the nitty-gritties of it. So, this shows that the language path is NL complete. That completes the proof that path is NL complete. Now, I will close with one brief exercise. So, the language ANFA, this is something that we saw in I think chapter 4. So, ANFA is the language where we are given the description of an NFA and a string and we are asked whether this string is accepted by this NFA.

And this turns out to be an NL complete problem. So, we have to show two things. One is that ANFA is in NL which is not that difficult. Then we have to show that all the strings in NL reduce to ANFA. But then we do not need to do that. We do not need to show that all the strings in NL are reducible to ANFA.

ANFA. Instead, we can take an existing NL complete problem and show that this problem is reducible to ANFA. So, we did, we listed and showed similar results in NP completeness. Here I am just using it. So, we can show that path is reducible to ANFA instead of showing

that all the languages in NL are reducible to ANFA. And just to give one line idea for this, so the path is like given a graph, is there a path from S to T.

So somehow we want to create an NFA there which the state diagram of this NFA should mimic this graph. So and there should be a string that takes it to an accepting state if and only if the original graph has a path from S to T. So maybe I will just say one more thing. If C is in NL and B is NL complete, and B reduces to C in logarithmic space, then C is also NL complete. So instead of showing all languages reduce to ANFA, we just need to show that an existing NL complete language reduces to ANFA.

So we use the same thing for showing NP completeness of various problems. So once we showed that SAT was NP complete, we just reduced from SAT or something that we already knew like clique or something.

So same idea extends here as well. Now, that is all I have for this lecture, lecture number 58. So, we defined NL completeness. So, we saw the language path. Path is a problem where we have a graph and we are asked to decide if there is a path from S to T. We first showed that it is in NL. Instead of using a breadth-first search or depth-first search, we are kind of guessing a path. Instead of explicitly guessing the entire path, we are just guessing the next vertex, next vertex on the fly and verifying that this is a valid path. Using non-determinism, we get an NL algorithm and we define log space reductions.

And we noted that we need log space reductions for NL completeness because the reduction has to be less powerful than the class that we are dealing with. It cannot be something like P, polynomial-time reduction. Then we noted that we cannot combine a reduction machine, a log space reduction, and a log space decider to get a log space decider, right, because the output of the log space reduction machine could be polynomial time or polynomial space which is too big. But instead, we explained this idea that instead of writing down the entire output, we can just compute the output on the fly, right, so we compute the output on the fly and send when whichever symbol is required. So, this will end up creating a lot of recomputing of the output of the reduction machine, but that's okay. We are not being charged for time.

We are only being charged for space. So we get the result that if A reduces to B in log space and B is in log space, A is in log space as well. So in fact, I'll just write down another result, which is similar in spirit. A reduces to B in log space and B reduces to C in log space implies that A reduces to C in log space. Basically, reductions are log space reductions are transitive. The proof of this is also very similar to the proof that we explained here.

Instead of computing and writing down the answer for one, we can just compute the reduction on the fly. Then we define NL completeness. So B is NL complete if B is in NL and all languages in NL are reducible to B in log space, then we showed that. So these are the hardest problems in NL. Then we showed that path is NL complete.

We already had seen that path is in NL. And the proof idea was to note that suppose A is an NL, we want to show that A reduces to path. So A is in NL, so A has an NL decider. And we look at the configuration graph of the NL decider. So, the configuration graph has 2^(O(log n)) configurations or polynomial in the length of the instance configurations. So we build a configuration graph for that NL decider and check whether there's a path from starting state to starting configuration to accepting configuration.

And then we said that this configuration graph can be generated using only logarithmic workspace. So the output, the configuration graph itself is of polynomial space, but the work involved in generating the configuration graph is log space. And that completed the proof that path is NL complete. We close by asking the exercise that ANFA is NL complete. So, I just stated two theorems which are similar.

So, you can try to show this similar to what we did in the NP completeness and similar to the proofs that we have already done. And that completes lecture 58. So, I will see you in lecture 59. Thank you.