[SQUEAKING]

[RUSTLING]

[CLICKING]

MICHAEL SIPSER: OK, welcome everybody. So we are here today, lecture number 21. Coming into the homestretch of the course. I'd say probably this last quarter of the course is a bit more technical, perhaps. So a little bit more abstract. Some of the theorems are going to be more difficult.

So I'll try to work through them slowly and answer your questions. But you know, I think you can expect to find the material a bit more challenging. As we started-- as we started with this theorem last time, about nondeterministic log space being closed under complement. So NL equals coNL.

We kind of only got about maybe a third of the way through that. So I'm going to start over with that, and spend kind of the first half of the lecture today talking about that. And then we're going to talk about the hierarchy theorems, which are a very important aspect of the complexity landscape.

Basically, they tell you that if you allow your favorite model, let's say Turing machines, to have more resources, then they can do more things. But we'll get to that in due course.

OK, so let us go back to-- oops, reminder to myself-- go back to the Immerman-Szelepcsényi theorem, which is that NL is equal to coNL. So as I mentioned, these are going to be the same slides as last time. And I'll just try to walk through them slowly. I hope you get it. But if you don't, ask me or the TAs questions.

So we're going to first, I mean, the thing we're going to show is that the complement of path is solvable in non-deterministic log space. We already know that path is solvable in NL. That's easy to do. You basically just start at the start node, and you guess the sequence of nodes, storing only the current node in your log space working memory, on your logs based work tape.

You guess the sequence of nodes on the different branches of the non-determinism. And if you ever get to the target node t, then you can accept. But how can a non-deterministic log space machine know or accept the complement of path? So it would have to accept when there's no path. And that is a lot harder. But big surprise to the complexity community that it is true.

So as we discussed last time, we're going to talk about computing functions with a non-deterministic machine. And that turns out to be a convenient way of looking at this. So we're going to have non-deterministic machines that have different branches of the non-determinism on some input, and it's supposed to compute some function value remaining on the tape.

But because of the non-determinism, you can imagine that different branches might have different function outputs. Well, that's not allowed. All branches must either report the value of the function that we're trying to compute, or they can punt. Basically, they can reject and say, well, basically I don't know.

So all branches must either report the correct answer, or they can say I don't know. And some branch must-- at least one branch must report an answer. Must report the answer. And that's what it means to be computing a function with a non-deterministic machine.

And we're going to show that certain functions can be computed with nondeterministic log space machines. In particular, this path function, which sort of incorporates both the positive and negative of both when there is a path and when there is not a path into the function, because the function has to answer yes when there is a path from s to t and no when there is no path from s to t.

So if you could do this, you're done. Because you can make a non-deterministic-- you can make an NL machine. So if you could compute the path function, you can make an NL machine, which would accept whenever the function says no.

And the other cases, and if the machine that's computing the function rejects, then you'll reject as well. But you accept if the function says no. And so therefore, you're going to be making an NL machine which does the complement of the path problem.

So if you can compute the path function, that would be great. So that's what we would like to be able to do. So as I mentioned, we're going to have two other values that are going to be relevant to computing the path function, which is what we're ultimately going to do. And that's going to be the number of nodes that you can reach from the start, from the start node in your graph, and the-- for-- R is the collection of nodes and c is the number of reachable nodes.

So shown on this picture. And here, if it's helpful to you to see it in a more formal [INAUDIBLE] is you can think of R as a function of the graph and the start node, of course. But sometimes we'll just call it R, when it's clear which graph and start node we're talking about.

R is the set of reachable nodes. So it's the collection u such that the answer is yes. And c is the size of R. So the way we're going to start is kind of an easy theorem, though this is still going to be relevant kind of at the end.

But for now, it's really more a practice with the concept that we have come up, you know, this function concept that we've just introduced. So I want to say that the path function, with an NL machine. Then I can compute the count with an NL machine.

So understand what a computing the path function means. That you have your NL machine, and every branch has to either say I don't know, which is reject. Or it has to have the answer, which it's going to be yes if there is a path, and no if there is no path.

And if I can be able to count the number of nodes that do have a path. The number of nodes for which the answer is yes. And this, I think if you're comfortable with the definition, this is more or less obvious, because what you would do is you would go through the nodes of G, one by one, and test using your path function what the answer is. Yes or a no.

And every time it's a yes, you add 1 to the count. Until you've gone through all of the nodes. And then you have your answer, which is [INAUDIBLE] and that's c.

Now, if the machine that's trying to compute the path function on the non-determinism rejects, that's OK. For computing c, that branch will reject also. But when you're-- some branch has to get the right answer on-- has to get the right answer.

And so then, you get the-- you know what's happening with that node. And so you then can either increment the count, or you move on to the next node. I think I'm trying to say-- I'm not sure if I'm making it any clearer by kind of repeating myself.

But OK, here. So you're going to start out with-- you're given the graph and the start node. We're trying to compute this value c, which is the number reachable from the start node. You start out with-- you have a counter, which you're going to set initially to 0.

And you go through every node of the graph. And if the path function computation says yes, you can reach u, then you add 1 to the count. It says no, you cannot reach u, then you just continue.

And maybe I should add another line here. If the thing that's computing rejects, then you also reject. And then at the end, you output-- so what we're going to prove is the other direction. If I give you the count, then I can answer the question for each node whether it's reachable or not.

And this is the thing. Because what it's saying that is I can give you the count, I'm done. If we can get that count, that's going to be enough. So maybe even before the check, and maybe we should just answer any questions.

Because if you're stuck here, then you're doomed. So I think it makes sense to try to understand what's going on at this-- because I think the real guts of this proof is coming on the next slide. Kind of the main idea. So I'm happy to take, if there's any questions about this.

I'll just wait for a second and see if you're typing away there. Well, why don't we go to the check in. Maybe that'll help. Not a very difficult check in. It'll come up. OK. Just a little practice with the concept.

So I'm going to give you some graph. It has 9 nodes. And I want to know the value of the count. So we'll assume that s, the start node here, is reachable from itself. And now what's the value of c? Are we good? I'm going to shut this down, give you another two seconds. Please get your answer in. OK, ready, set, end.

Yeah, the right answer is, in fact, E, which is 6. There is 6 reachable nodes in this graph. And that's what the value C is supposed to tell you, is how many nodes can I get to from s. And what I'm saying is that if I can calculate that in this sort of non-deterministic function sense, so if some branches can get that answer, then I can use that to test for each node, whether it's reachable or not, which is kind of a little bit of a miracle.

That's kind of surprising. Just knowing how many nodes are reachable will allow me to test whether each individual node is reachable, because that's-- no obvious reason why that would be. So there's going to be a procedure for doing that, which is on the next slide. And here it is.

So this is the key idea that we're going to repeat later. But so it's good to understand. This is the slide you really need to understand. So I'm giving the graph. Let's assume the graph has m nodes. Now, as I said. OK, so let's just say, what are we doing here?

Given that count, we can compute path. So we'll get the answer for every node in the graph, if I just know how many-- so I'll know, I can get the answer for whether a node is reachable or not if I just have-- if I just know how many reachable nodes there are.

So what I'm going to do is get that count of how many are reachable. Now, I'm going to go to-- I'm going-- so look. Let's see, what's the idea here, before we even jump into the algorithm.

The idea is let's say I know how many nodes are reachable, like 100 nodes are reachable. Now, what I'm going to do, what the algorithm that is going to do is find all 100 reachable nodes. [? Go ?] 1 by 1, but it doesn't matter. Sort of conceptually, it's going to find all reachable nodes.

And non-deterministically guessing them. So it's not sure in advance which ones they are. But it's going to guess basically 100-- it's going to guess some of the nodes as being reachable, confirm that the ones that it guesses are reachable are reachable, and then check to see that that number equals 100.

On some branch of the non-determinism, you will guess right, and you'll end up with exactly the right set of 100 reachable nodes. And then you'll see, is t one of those reachable ones? In which case you say yes. Or is t not one of those 100 nodes.

And then you know the answer is no. Because if you've guessed 100 nodes, and you know they're all reachable, and you know there are exactly 100 reachable nodes, then every other node is not reachable. So that's the spirit of this.

And that's, I'm just going to write that down here in the algorithm. Can you guys hear me still? Yeah, somebody said my audio is like flipping out. I am getting a sign or two of unstable internet. So if you need me to repeat anything, just send me a note. Good, thank you.

All right, so well, maybe I should speak slowly, if it's not coming through too well. So what we're going to do is go through all the nodes of the graph, one by one, and guess whether it's a reachable node or not a reachable node.

If we guess it is reachable, I'm going to also guess the path, which shows that it's reachable. And then I'm going to-- and I'm going to keep a count of how many reachable nodes I found. If that count agrees with the value c I started with, then I know I found them all. And if t is not one of them, then I know t is not reachable. That's the idea.

So here's my-- this is going to be a count of the number of nodes that I have found which are reachable. That's k. Now here I'm going to non-deterministically choose, is it a reachable node or not. I've just called it two branches of the algorithm, the p branch or the n branch. p means there's a path, an n means there's no path.

So if I guessed p at this point for this node u-- so I'm going to each of the nodes one by one. u is the current node. If I've guessed that it does have a path from s, then I'm going to guess that path, to make sure that it really is a reachable node.

If I fail to find a path, then this is one of the branches of the non-determinism that is going to fail. It's going to punt. It's going to say, I don't know under this branch. Because either you guessed wrong, and this node was not reachable.

Or if it was reachable, you failed to find a path, which shows you that it's reachable. There was some path, but you didn't guess the right one. Either way, you made a bad choice. You're just going to punt.

Now, if you have determined that t is that node that you've just shown is reachable, because at this stage, you did not fail, so you succeeded in showing a path to u, then-- and u equals t, then t is reachable, so there is a path from s to t, and you're finished. Because you know, you've got the answer you're looking for. And so now you can say yes.

Otherwise, if u is some other node, then you can just increase your count of the number of reachable nodes that you found. So you found a reachable node. If it's t, you're great, you're done. If it's not, you just include that in your count of reachable nodes.

Now, if you've guessed that the node is not reachable, OK, then you just proceed-- you're not going to-- you're just going to move on to the next node, because you're looking for a collection of reachable nodes.

Getting some questions here. But let me wait till the end here. Now, after I finish going through all of the nodes, so I'm finished with this loop here of going through all the nodes, now I see, did I find c reachable nodes, because k is the count of the nodes that I've found to be reachable. If that agrees with c, then I know I found them all.

If a differs from c, then something has gone wrong. Because I am told there are c reachable nodes, and I did not find c reachable nodes. So I made some bad guesses along the way. I guessed some node which really is reachable, I guess it was not reachable. So I didn't find them all. I'm going to punt.

But if I found them all, and I didn't end up accepting it, I didn't say yes at this stage, so t was not one of the ones I found unreachable, then I'm convinced that t is not one of those that are reachable. That was not one of those c nodes that I found which are reachable. And now, I can say no.

So let me take questions here, because I think we're, yeah, that's the end of this slide. This is kind of an important piece to understand. We can spend a couple of minutes trying to work through this.

So somebody is asking, how does nondeterministically pick a path fail? If you fail, what I mean is pick a path from s to u. So you have to go from s to whatever your current node u is.

So you're going to pick some path to u. You guessed u is reachable. Now you have to demonstrate it's reachable by picking a path from s to u. If you don't end up at u, and the pair-- you don't want to go forever on any branch. So you're going to limit it to m steps. Your path has to be of length m at most.

So after m steps, if you have not reached u by that point, you've picked a bad path, and you're going to reject.

So what's the difference between no and reject? That's a good question. Reject, in this case, is an I don't know. The algorithm could not make a determination based on the guesses that it's made. In this non-deterministic branch of the algorithm, it made bad choices, which doesn't allow it to reach a conclusion one way or the other.

Remember, this algorithm here is computing a function now. It's not an alg-- it can [INAUDIBLE] nondeterministic algorithm in the language recognition sense, this is a function computer. And so it has to get the answer to the path function, which is a yes or a no, or an I don't know on some branches. On some branches it's allowed to do that too. So no and reject are totally different.

This is the same thing we talked about last time. Why do we need two branches for p and n, if we're only going to have a proposal just to have the p branch? Well, but some nodes are not reachable. If you're going to look for-- if you have an unreachable node, so it's not in R, you can't get to that node from s, you have to skip over that node, because you're trying to find a subset of the reachable nodes.

So you're trying to pick that subset here one node at a time. So if you're only going to allow things-- you're going to require everything in the subset, there are going to be some nodes which are not reachable, and you're not going to find a path because they're not reachable, and you're going to end up rejecting all the time on that node. So you're going to be-- the algorithm will not work.

So I'm not sure I understand this question here, but somebody says, if t is reachable, we output yes on that branch. But don't we also output no on some other branch? That's a good-- let's see what happens if t is actually reachable.

How can we-- so if t is reachable, there's some branch that's going to output yes. We all agree with that. Or at least, if you're following, we agree. But could some other branch output no? If t actually is reachable. OK, that's a great question.

And no, that's not going to happen. If t is actually reachable, how could a branch output no? That must mean that it does not guess t as one of the reachable nodes. Because it's going through all of the nodes here, you know, it's going through all the nodes, and picking them as reachable or not.

If it picked t as one of the reasonable ones, then it's going to output yes. Because it will find, it'll either output yes, or if it doesn't find the right-- doesn't guess the right path, it'll end up rejecting on that path. But some path will end up saying yes, so if t is reachable.

And if you guess-- if you know t is reachable, and you guess t-- you guess u is reachable at the point when u equals t, you will end up outputting yes. The only way you could not output yes is if you guessed that node is unreachable.

But then your count is not going to add up right. Because you wouldn't-- you did not find all the reachable nodes. If t is one of the reachable nodes, and you know there are 100 reachable nodes, and you skipped over t as one of the ones that you say is unreachable, you at best can only find 99 reachable nodes.

And you're not going to end up saying no. You're going to end up rejecting. So it's a very good question. But you have to think through what's going to happen here. This c here is kind of a check. It's almost like, well, it's like a checksum, if you know what it is.

It makes sure that everything that-- if you got to c. If you got to-- if k equals c at this point, that means you actually found all of the reachable nodes. So c is kind of a check that you found all the reachable nodes.

Right? So if k equals c at this point, you have found every reachable node. And t was one of the ones that are reachable. You found t. OK, let's see.

Is the reason we do this with c essentially so that we know when we can stop guessing, and correctly identify if it's impossible to reach t? Well, it's not a matter of-- it's not a matter of stopping guessing. It's a check that we found everything. Because we're going to go through and do all the guessing for every node no matter what.

So we're not going to stop anything early, unless we find that t is reachable. Then we can stop early. But to show that t is not reachable, we have to go through the whole process.

How come we intuitively see that we don't have contradictory branches? That's sort of-- I was trying to say that just now. I hope that got through. You can't have contradictory branches, because if you got to this stage here, you have found all the reachable nodes.

So at this stage, if you got to six, you have made all correct guesses. You have found all the reachable nodes. You have convinced yourself that they're all reachable by guessing the path to them. And you've checked that you have the right number of reachable nodes, because it equals c.

So you must have found them all. So you cannot have a contradictory answer, because either t is one of the ones you found, in which case, you would have already said yes. Or otherwise, you found them all, and t was not one of them. And so you're going to say no, you can't have both things cannot happen. Let's move on.

So next thing we're going to do is the next slide is exactly the same as this slide. Except instead of saying is t reachable, I want to know is it reachable within d, within distance d. OK? So--

Which is going to mean exactly the same procedure. Can I get-- instead of asking can I get from s to t with a path of any length-- of course, it's going to be at most length m-- now I want to know, can I get to from s to t by a path at most length d. These are the number of edges in the path, say.

And that's the same procedure, because instead of-- I'm just going to cut things off at d. But if I know in advance how many nodes are reachable within d, I'm going to find all the nodes that are reachable within d and see, was t one of the ones reachable within d. It's the same exact idea. So here is the next slide, which kind of shows that.

So here is the definition. Path sub d means reachable by a path of length of most d. So R sub d is all of the ones that are reachable by a path of that length. And c simply is the count. It's the number that are reachable within d.

So if you understood the last slide, hopefully this slide will seem kind of obvious to you. I'm going to just highlight all the changes. So if I can now calculate c sub d, which is the number reachable by a path of most of length d, then I can test whether or not nodes are reachable by a path of that length.

First, I calculate c sub d. I go, I pick every node as being reachable within d or not. Now I just have to check that my path that I'm guessing has length at most d, instead of length of most m, which is what I had before.

Keep a count of the ones that I found. If that count equals c sub d, then I know I found them all. If it's not equal to c sub d, then I've made some bad choice along the way, and I can just punt and say I don't know. And if t was not one of the ones that I've shown to be reachable within d, then I know it's not reachable within d. And so I can say no.

So I don't know if this merits any additional questions. But this is really the same. It's just a repeat of the previous slide. What's kind of amazing is now the last slide is going to be again a repeat. Let me just foreshadow where we're going. But feel free to ask a question, on this, or on the first slide, if you didn't-- on the previous slide, if you didn't get that. Also, we can try to help you out with that.

The next slide, what I'm going to do is show how to compute all these c values. And I should mention, the value c, which is the total number of reachable, it's going to be the same as c sub m. Reachable with an m, the number of nodes of the graph. So if I can get up to c sub m I'm done.

And what I'm going to show you is that knowing c sub i, I can compute c sub i plus 1. Or c sub d, I can compute c sub d plus 1. Since I'm using d as my index here basically. So c sub 0 we know is just s. Well, it's just 1, because you can just start with s. That's the only thing reachable within 0.

And then, once I know that, I can figure out c sub 1, c sub 2, c sub 3, and so on, and then I get the c sub n, and then I have the count of the total number reachable, and then I can test the path function.

So the trick now is being able to count. Given c sub d, I would like to figure out what is c sub d plus 1. Now, how am I going to do that? What I'm going to do is that's my goal. What I'm going to do is something in between. I'm going to do a theorem just like this, but instead of giving c sub d, instead of computing paths of d, I'm going to compute paths of d plus 1.

So knowing how many are reachable from d, I'm going to give a test for whether things are reachable within d plus 1. And the fact is, that's easy, because this thing already tells me how to compute whether I'm reachable within d. And being able to be reachable from within d plus 1 means I have an edge from something that's reachable within d.

So if I can figure out which are reachable within d, well, and I just want to see, do I have an edge, do I have an edge from one of the nodes that are reachable within d. Then I'm reachable within d plus 1.

Then if I can test whether individual nodes are reachable within d plus 1, I can count how many nodes are reachable within d plus 1. That was that very first easy theorem that I showed.

So I know there's a lot of pieces here that you have to put together. But in the end, each individual piece is not that bad. I don't know how many of you follow me. Oh no, this is not supposed to be here. There we go.

So here is the last part, which again, is just a simple modification of what the previous slide had. So I'm going to show how to compute the path d plus 1 function. So testing if there's a path of length d plus 1 from s to some node t.

But only knowing how many nodes are reachable within d. So I'm going to find all nodes that are reachable within d, just like I did before, but see if any one of those nodes has an edge to t. Not necessarily that one is equal to t. Because that says that t is reachable within d. But I want to know, does it have an edge to t. That means t is reachable within d plus 1.

So if I find all the nodes that are reachable within d, and t turns out to be reachable from one of those by an edge, then t is reachable within d plus 1. And if d is not reachable from any of those nodes with an edge, then t is not reachable from d plus 1. I hope you're following me. I'm not sure you are.

So anyway, that's the algorithm here. And the corollary is that you can compute c sub d plus 1 from c d, because if you can count the past, if you can test for each node if it's reachable, as I mentioned before, you go through all the nodes, see whether the reachable d plus 1, and then count them up. Now I have c sub d plus 1.

And now I'm done. Because I'm going to compute each d plus 1 from the value d that I previously computed. I'm going to do that for all these. Should say 0 here, actually. And except, if the path says-- if the path function now says that the answer is no. Because I'm trying to do the complement of the path language. And reject if the path thing for m says yes. And that's my non-deterministic algorithm for the path complement problem.

Anyway, maybe you need to look at a little bit offline. It's presented in a little bit different way in the book. I don't know if that will be more or less clear to you. But I think this has been a little bit more unpacked for the purposes of the lecture.

So let's just see. I'm not getting any questions, which probably means I've lost a huge chunk of you. But the good news is we're going to move on to a different topic.

But feel free to ask a question on this if you want, or we're going to shift gears now to talking about the hierarchy theorems, which is going to be the second half of the lecture. Also, not so easy, I have to say. Probably a little less technical than this one is, but it's also there's going to be spending time on just mainly just one theorem.

But anyway, so looking ahead to where we're going, and then we'll have a break. What we've shown so far, these are the major complexity classes. I'm not including, let's say, the complementary classes, the co-NP type classes. These are the major classes we've seen so far.

And as we've seen, they form a hierarchy of containments. Some of those containments trivial, and some slightly less trivial. But we have not shown whether any of these classes are different. We've pointed out that there were some unsolved problems here, but do we know any of these classes differ from each other? Or could it all collapse down to L?

And the answer to that is we do know that PSPACE and L are actually different. That we can prove. And it relies on the theorem that says if you give a Turing machine more space, then you can do more things. So because PSPACE is a bigger bound than log space, we know we can do more things.

In fact, because NL is contained within log squared space deterministically and PSPACE is bigger than that, we actually can separate PSPACE and NL. So we're going to prove that today.

So basically, the idea of the theorem says that if you give a Turing machine a bit more time or a bit more space, then it can do more. So there are some conditions on that we have to-- we'll get into. One of the conclusions that we'll show is that time n squared, if you compare with time-- the things you can do in n squared time versus the things you can do in n cubed time, there are more things you can do n cubed time than what you can do in n squared time.

I mean, that's what you would expect. But it's not the case that everything we expect in complexity theory, we can prove. This is one of the things we can prove. So as you add more time, you can do more things.

So this is a proper subset here. So there are some things n time n cubed that are not n time n squared, and ditto for space. So that's going to be-- that brings us to our coffee break. And so feel free to shoot me any questions about what we've done so far, or anything else. And otherwise, we will launch our timer, and I'll see you in five minutes.

OK, so getting some good questions here. Could we also make a solution-- this is getting back to that, the logs-- the NL equals coNL. Somebody's saying could we just make another selection just by non-deterministically choosing C vertices, and then checking that they're all reachable.

That's effectively what we're doing, but be careful, because we cannot store C vertices. So that's why we're doing them one at a time. We can't guess all C vertices up front, because where are you going to store all that? We only have log space.

OK, another-- somebody is asking, how much working space do we need for storing the intermediate steps? I'm not sure what intermediate steps you mean. But if it's all to Ci values. You know, C going from C0 to C1 to C2 to C3, we don't store those.

All you need, you need C sub D to calculate C sub D plus 1. And then you forget C sub D. You couldn't store all the C values. But you don't need them all. You only need the most recent one to go to the next one.

OK, somebody's asking, can I go over why the complement of path in NL implies NL equal coNL? Because the complement of path is, essentially it's coNL complete. I mean, it is. So everything in NL is reducible to path. Everything in coNL is reducible to the complement of path by the same reduction.

And so if you can do the complement of path in any class, you can do all of the complements of NL languages in any class. And so you can do the complement of path in NL, you can do all of the coNL problems in NL. And so then NL equals coNL.

You have to think through the logic of it. That part is not hard. Why? It's enough to solve the path complement problem in NL. That does everything else. Because it goes through the same completeness phenomenon that we've been seeing.

Somebody is asking about the two set problem that we talked about last time. And is it, and you know, I pointed out that that's in NL. The two set problem. Well, the complement of the two set, problem, the unsatisfiable two set formulas. That's in NL language, because you can basically look for a contradiction non-deterministically in log space.

I think I probably won't be able to explain that in a minute. But maybe we'll have our recitation instructors cover that in recitation. It's a nice proof. Not very hard. But it's a nice proof.

It's something you have to do, you have to think about, you have to argue. But it's still, it's not super hard. Understanding the two set problem. And the complement of two set we showed, is in NL.

And because NL equals coNL, also the two set problem itself, without complemenation is in NL. And in fact, is NL complete. I think we're going to have to move on.

I'll stick around after lecture, in case there's any questions that I can answer quickly at that point. Sorry if I couldn't get to your question just now.

All right. Continuing on here. Shifting gears, the space hierarchy theorem. So as I mentioned, I think, maybe it's good to just go back to this slide here. We're going to do the time and space hierarchy theorems, which show that if you can do a little bit more-- if you give a little bit more time or a little bit more space, you can do more things.

We're going to do the space case first, because that actually tends to be slightly, for certain technical reasons slightly easier. So space hierarchy theorem.

So here is the statement of the theorem. So for any bound, think of s is going to be some space bound. And again, f has to satisfy some technical condition in yellow. Remember that it's yellow because that's going to be relevant later. So there's going to be some technical condition.

No matter what function you have, whatever space bound you have, as long as it satisfies this condition, which is a mild condition, but you need it. Whatever space bound you have, you can find a language A which requires exactly that much space.

So if f is like n cubed, we're going to find a language A that requires n cubed space. If it's n to the hundredth, we can find a language that requires n to the hundredth space, and cannot be done with n to 99-th space. Whatever it is, you can find a language that requires exactly that much space.

And if you like it a little bit more formally, so that means that it can be decided in that much space, but it cannot be decided in less space.

Framing it at a slightly different way in terms of our space classes, I'm going to define a notion which is kind-- it's not said this way in the book, but maybe it's a helpful way to write it down. It's space little o of f of n. So those are all the things that you can do by a function that's a little o of f of n in space.

So space little o of f of n is properly contained within space f of n. In other words, there's something here which is not in there. Picture it pictorially, I'm going to exhibit some language, some explicit language A, which I can do in this much space, but not in any less.

Now, you can sort of think of this as a little bit like the situation for context-free language and regular languages, where we exhibited a particular language that differentiated-- that was in context free but not regular. And we're going to kind of do the same thing now.

But the one key difference is that in the case of separating the context free and the regular, we could give a nice language, like 0 to the k, 1 to the k. Here, the language is not going to be so nice to describe. It's going to be the language that some Turing machine we're going to give decides.

But you're not going to be able to get a nice simple understanding of A. It's going to be whatever that Turing machine does. And in that sense, it's not a very natural language that's easy to sort of get your mind around.

So the outline, and really, you don't have to worry about this, but maybe it helps. It's really going to be a kind of a diagonalization proof. The way this machine D is going to operate. So D is going to give you my language A.

So D is going to be designed, and I'm going to show you D on the next slide. D is going to run within my target space bound, f of n. And here's the key. Here's the kicker. D is going to be designed to make sure that its language cannot be done in less space.

And the way it does that is it makes sure that its language is different from any language that is decidable by a Turing machine in less space. And it's going to be different in at least one place.

So any-- D is going to guarantee that its language cannot be done in little o of f of n space, because it's going to be different from every language that's doable in little o of f of n space somewhere. That's the point. And then the language A is going to be the language of this Turing machine D.

So it looks like a tall order. The D has to make sure that each-- that for every machine, its language differs from that machine's language, if that machine is running in little low o of f of n space. But it's basically going to be a diagonalization.

So for all of the different possible inputs to D, that input is going to actually code up a machine, on which we're going to make sure that we're different from that machine if it's a small space machine. Let's see if that--

So I can take a couple of questions here. Does f have to be computable? So that's going to be one of the conditions that we're going to have to guarantee. Where f satisfies a technical condition, yeah, it's going to end up being have-- it's going to be computable. But that's not enough.

Good, good question, though. OK, so let's move on from there. So here is-- now what my job is to give you this Turing machine D. So D, D's language is going to be my language A, which requires f of n space, cannot be done in less.

OK, oops. I need to-- I need the full slide here, so I have to take myself out. All right. Now, this is my goal. I want to exhibit this language A, which I can do in this much space, but not in any less.

And so I'm going to give this machine D, as I mentioned, where A is D's language. D runs in order f of n space. And that sort of-- that achieves this part. And D, make sure that its language cannot be done in any less space. So that achieves this part. So it's different from the language of any machine that runs in little o of f of n space.

So this is how D is going to work. I'm going to try to give you a little picture to help see to accompany the description. So D gets its input w, which is of length n. The very first thing D does is it marks off f of n space.

Because it's only allowed to use-- we're only going to allow D to use f of n space. Because otherwise, we're in danger of D not-- of A not being in space f of n. So D is going to guarantee that, by making sure it's going to mark off f of n space. And if it ever tries to use more than that, it just rejects.

But by virtue of that, we're sure that D's language is in space f of n. Because D is an f of n space Turing machine. And it's going to be decided. So this part so far is not too hard.

Now we're going to start getting into the meat here. So if w-- now, what we want to think of w as a description of a machine that we're going to feed-- that's going to run on w. So this is going to a little bit, you know, back to an earlier when we talked about diagonalization. So don't get thrown off by this.

We're going to think of w not only as the input to D, but it's also going to be the description of a machine. And if it turns out that w doesn't describe anything, it's just a junk w, then we're not interested. We're just going to reject on that w. We're only interested in the w's that do describe some machine M.

So if M-- if w describes some machine M, then we're going to run M on w. And we're going to do the opposite of what M does. That's the whole idea. We're just going to make sure that what we're doing is not the same as what M is doing.

So at a high level, the basic idea for this is not hard. So we're going to simulate M on w. If M rejects, then we'll accept. If M accepts, then we'll reject. We're just going to do the opposite.

And I think that is-- so we have to be careful when we do the simulation. This is a little bit of a detail, but this is a proof where you need to pay attention to some details.

The cost of simulating M on D is only a constant factor. Because if M uses a certain amount of space, when D is simulating M, you know, M may have a larger taper alphabet than D does, but D can then encode M's tape by using several cells for each of M's cells.

But it's only going to be a constant factor. And that's important here, because we have to make sure that if this was a big blowup, D would not be able to run M. I think I'm sort of arguing the details without making sure we understand the fundamental concept. So let me back up.

The point is that D is doing something the opposite of M. Now, D can't be different from every M, because D itself is a Turing machine, of course. But the thing is that D is only running with an f of n tape cells. So it has to be able to do that simulation of M within that amount of tape. If M is using a lot of tape, then D is going to use a lot of tape, and it's just going to reject.

So this is only going to really come into play, being able to simulate M, if M is using a small amount of space, so that D can do this simulation. Let's just see, maybe-- so there's going to be some issues here. But before I get to that, let's just see what your questions are.

How can a Turing machine know if w is encoding some other Turing machine? Oh, that's simple. You know, what is a coding of a Turing machine? It's just the standard-- we have a standard coding. It's just coding the rules of the machine.

So it has to have states, transition function, blah, blah, blah. So it just has to be some-- whatever our encoding for the Turing machine is, we can always test whether a string is a legitimate encoding of a Turing machine. So that shouldn't be bad.

Somebody says, why do we reject if we use more than f of n cells? Isn't it OK to use order f of n? Yes, it could be. But we have to cut it off somewhere. It might be-- it's OK, we could use 2 f of n. We could use 10 f of n. But we have to have some constant for D. And that's just simply constant 1. So D has to run within f of n cells. And that's going to be good enough for us.

OK, do we have to make sure that M runs in little o of f of n? So we can't really tell whether M is running in little o of f of n. All we can tell is whether we can finish the simulation. So that's actually going to be-- maybe you can just hold off on that question, because there is a point that we have to follow up on in that, which is just because we may or may not be able to finish simulating M on this w doesn't necessarily tell us what the asymptotic behavior of M is. But we'll have to look at that in a bit.

So somebody saying, what happens if M loops on w? That's going to be one of our issues we have to deal with. That's a good question there. Step two alone can use more than f of n cells.

Yes, step two alone can use more than f of n cells. If it does, we're just going to end up rejecting. So we're getting good questions here. Some of them I'm going to address anyway, so why don't we just move on.

So here is sort of a question. I think this is one of the questions that related to one of the ones that got asked. What happens if it runs in little o of f of n space? So remember, what we're trying to do is be different from every small space. You know, little o of f of n space machine.

So what if M runs in little o of f of n space, but has a big constant? So what I mean by that concretely, is suppose D is an n cubed space. So suppose we're trying to get A in n cubed space, but show it's not in n squared space.

So D is going to run in n cubed. And we have to make sure that any machine that's running in n squared space cannot do the same language. So we're going to be different from that. But the problem is that the machine M might be running in n squared space, but with a huge constant. So it might be running in a million n squared.

So that's still a machine that's running in little o of n cubed. And we have to be different from it. But for the particular w we're working on, we might not have enough space to run M, because of the huge constant.

The asymptotic behavior is only going to be relevant for large W. For small w, we may not see that. We may not have enough space to run M. So what are we going to do to fix that? We're going to run that M on infinitely many different w's. There's going to be infinitely many different w's that are all going to encode the same M.

And the way I'm going to do that is by thinking of w as representing M, but having an unbounded number of trailing zeros after that. So I'm going to strip off-- the very first thing I'm going to do with w, is I'm going to strip off the trailing zeros up until the final one. I'm going to remove those. And then take the rest, as the description of the machine.

So now, I'm going to have potentially w's that have an enormous number of zeros at the end, big enough so that I can see the asymptotic behavior of M, and that if M is really running in little o of f of n space, I'll have enough space to run M to completion on w. And so then I'll be able to be different from it.

So I'm going to showing that over here. So here's a very large W. I'm going to strip off the trailing zeros. The rest of it is just going to be M. And I'm going to run this M on the whole w. The entire w, without the zeroes stripped off. So now M is going to be running on a very large input, big enough so that D, which has asymptotically more space than M does, will have enough space to run M to completion.

Now, another question that got asked. What happens if M loops? That's going to be a problem, because D always has to hold. And if it just blindly simulates M, then D might be looping on M. Not if M is going to use a lot of space, by the way. Because then D is going to catch it in step one.

But if M loops on a small amount of space, then D might end up looping, as presently constructed. So what I'm going to do is I'm going to put a counter which makes it stop if it runs for 2 to f of n space. So basically, because that's how long D could possibly run without looping anyway, and M could be running without looping anyway.

And so we're going to run it for this number of steps. And I'm going to reject if it hasn't yet halted, as well as that. Because it has to be looping at that point anyway. And so it's not interesting for us. It doesn't matter what we're going to do if it hasn't halted. Because M is not a decider.

And the last thing is how to compute f. So I'll try to address some questions here in our remaining time. How to compute f. So to mark off f of n cells, we also have to compute f. I didn't think any of you guys asked that question, except maybe sort at the very beginning, about f being a computable function.

Certainly, f is going to have to be computable. But not only does it have to be computable, it has to be computable within the space bound. And that's just going to be a condition we're going to impose on f. It's so called space constructable, namely, that you can compute it within its own space bound.

And all nice functions that we care about are going to be space constructable. So it doesn't turn out to be an obstacle to applying the hierarchy theorem, but it is a condition that we need. It actually is not true without that condition.

Let's just-- oh, I have a check-in here. Maybe if we can take a couple of questions first. Some of you are anticipating my check-in actually, which is good. So let me hold off on those. Sorry, a bit confused about what is M. Can we say D is input M, and simulates M on-- yeah. So somebody is saying, can we say that D has input M, and simulates M on itself?

Yes, that's exactly what's happening. The reason why we're doing that is because we have to cover all possible M's. So as we get all possible inputs w, they're going to range over all possible M's. And so every possible M is going to get addressed to see if we can run it within the space bound, and be different from it.

D's job is to be different from each of those M's. But it's not-- again, there is some details here that got raised in these issues. But in a sense, this is just kind of more technical, I would focus on understanding what I originally wrote down, because that's the main idea. The rest of it is just implementation details.

So why don't I-- can I give an example of a nonspace constructable function? Yes. Log log log n space. You cannot compute log log log n space within log log log n space. And in fact, it's known that there's nothing new between constant space, which is just regular, and log log log n space. Anything you can do in log log log n space is a regular language.

So the hierarchy theorem doesn't want to apply there. Because well, it applies, but that's not a-- it's not space constructable. To find higher level large nonspace constructable functions, you can do it, but they're not easy to describe.

Let's do our check-in here. What happens when we run D on itself? I got a couple of people asking me about that. So this is just a good lead in to our check-in. And this a little-- you really have to understand what's going on, to see what does D do when if you feed in itself, maybe with some trailing zeros. Because remember, the algorithm strips off trailing zeros.

So what does it do in that case? So here are my options there. You can get to pick which one you think is the answer. So I'll give you another 30 seconds on this, because this requires a little bit of thinking. If you want to invest in it. All right.

Wrap this up, guys. 5 seconds to go. OK, I'm going to end it, so get your participation points in. A bunch of you have not said anything. Come on. I can see the count here, and it's three or four of you are not answered. Well, you're going to lose out. Closing.

All right. So the right answer is, in fact, C. It does reject. Let's just understand what happened. It's definitely we don't get a contradiction. I mean, this is an algorithm I just described. It's going to do something.

I'm assuming the people who picked E are having fun, as I did when I came up with the check-in. But not a question-- answer A is not going to be good either, because D has to be a decider. So it can't loop on anything. So the only sort of reasonable answers are reject, accept, or reject.

When you run D on itself, what's it going to try to do? It's going to-- the very first thing it's going to mark off f of n tape cells. And then it's going to get its input, which is itself, tries to simulate itself on the same input. That simulated D is also going to try to mark off f of n tape cells.

But due to some simulation-- there's going to be some cost to doing the simulation. When the simulated D is going to try to mark off f of n tape cells it's going to blow the original D's space bound and exceed the bound. And so D is going to reject upright in step one when it tries to get an input of itself.

It's very clear what's going to happen. It's just going to reject, because of-- for this, reject in particular. And notice this, you know, yes. You know, OK, let me not try to confuse it. OK, so that's all I want to say about this.

Let's now move in our remaining 7 minutes to the time hierarchy theorem, which is very has the same proof. But some of the technical details are slightly different.

So now, if I give you a time bound, again, we are going to have to face with the same notion, that you have to be able to compute f within x amount of time. So it has to be a time constructable. I'm not going to define that.

So there's a language A which requires that much time. So it has to be decided within that much time. But there's a slight difference here. And this is an artifact of the proof of the theorem, not because it's an absolute truth, as far as we know.

It's not that it's not decidable in little o of f of n. You actually, you can only prove something slightly weaker when you have one tape Turing machines. That it's decidable in little o over-- there's a slight gap in what you can prove.

So it's not only that you can't prove-- it requires little o. But little o of-- little o of f of n over log f of n is what you can prove that you get from this time hierarchy theorem. But let's not get caught up on that for now.

So the proof outline is the same outline as we had before. We're going to give a D that runs in order f of n time. So it ensures that the language is in that time complexity class time f of n. And it makes sure it's different from every machine that runs faster. By some significant-- by a log factor faster.

And so, why don't I show how that goes. The proof is, in some ways, almost exactly the same. I'm going to give a D, which runs in this much time, and it shows it's different from every M that runs in a lot less time.

Here is the algorithm for D. Now, it computes f of n. But it does something a little different with f of n. Remember, in the space hierarchy theorem, we marked off f of n space.

Now, this f of n is going to be used for a different purpose. It's going to be a clock. And you have to shut M down if it runs for more than f of n steps. Not if it uses more than f of n space. Because we're only interested in M's that use significantly less than f of n time.

So we're going to run M for f for some number of steps. Whatever M says, we're going to do the opposite. And only if we can actually finish that simulation, will we be able to be sure that we're different from what M is doing.

So this is the whole algorithm here. We don't have to do any further modifications. And where is that log n factor coming from? It's actually coming from a funny place. And you have to get into a little bit of the guts of this.

When you're simulating M on w, remember that M itself was described by w. So you're going to have to write down a copy of M, which is just as described by W. And then so you're going to-- and then you're going to have the tape that M is working on, which is starting out with w on it.

And you have to be, now you have to be a little careful how you manage that. Because if your description of M is just sitting at the beginning of the tape, as you're simulating M, every time you do one step in modifying the tape, you don't want to have to go back to the beginning of the tape to look up the next step of M.

So you actually have to carry M along with you as you're doing the simulation. And you can do that by expanding the tape alphabet of the tape so that you can effectively have two symbols on one cell. One is going to be for describing M, and the other one is going to be for just for the simulation tape.

And you'll be carrying M along with you wherever your head is, so you don't have to go very far to look up M. And so that's all possible, because that's going to add only a constant factor, because M is fixed in size, doesn't depend on-- for large inputs to M, M is fixed.

But the tricky thing here is the counter, to make sure we're not using too much time. The counter has size log f of n, because that's how big it has to count up to so you can shut it down if it's going to exceed f of n steps.

And let's see, you'll have to run for a certain amount of time. And keeping the counter nearby has-- the counter now, it can be pretty big. And so that's going to cost you a log factor of simulation cost to move that counter around all the time. And so that's why you have to run for only a log factor less, so that you can actually finish within f of n time, as you're required to do.

I realize that that's a mouthful there. And you may not have all understood that. It doesn't matter. It's not that critical. I think what I'm really more concerned is you understand the main idea of the hierarchy theorem. Some of these implementation details, if you don't get them, I wouldn't worry about it.

I feel I have to include them for completeness sake and to be honest with you about the proof. But if you didn't follow everything, that's OK. I do want to understand the main idea, though, of the algorithm, making sure that what it's doing is different from what every machine is doing if that machine runs in little o of f of n space or a small amount of time, little o of f of n over log f of n time.

And I think we're going to end here. So we're pretty much out of time. I'm going to stick around for a little bit, in case there's any questions here. Oh, there's one last check-in though.

Let's look at this. This is kind of an interesting sort of follow on to the hierarchy theorem. If you look at the two questions, does L equal P, and does P equal P space? These are both unsolved problems. What, if anything, does the hierarchy theorem tell us about those questions?

And it's kind of interesting, that there are actually-- well, I'll leave it to you to tell me if you can see what it might actually be telling you. Closing. Get your answer in.

OK, 1, 2, 3. I feel like I'm running an auction house here. And I should have a gavel. Yes, in fact, we know that these are separated. So it's not-- even though we don't know if L equals P, or P equal P space, they can't both be equal. Because then L would equal P space, and we know that's false. So at least one of these has the answer no.

So with that, let's wrap up today's lecture. Now we prove these hierarchy theorems. And why don't we just-- I'm going to shut us down here. But before-- well, I mean, we're over, so you can feel free to go. But I'll stick around in case anybody has any questions for a few minutes anyway. And then we'll call it a day.

Since we just showed space N is a proper subset of space N to the K for any K, why can't we also say space N is a proper subset of P space? Yes, space N is a proper subset of P space. Yeah.

So somebody just asked, we just showed that space N is a proper subset of space N to the K. Does that also say that space N is a proper subset of P space? Definitely.

Space N to the K is a proper subset of space N to the K plus 1, which is a subset of P space. So any fixed polynomial is going to be a subset of P space, because P space includes all the polynomials. Which of the two unsolved problems-- whoops-- do I think is more like likely to be true? Well, I think most-- I mean, I would bet that both of these are not equal. So both of these have answer no.

It would be weird, you know, I mean, you think L equals P? That anything can do in polynomial time you can do in log space? Log space is incredibly weak. And P space is incredibly strong. I would be shocked if either of these were equal. So we just-- the problem is that we don't have a method for proving problems are actually, have high complexity of any sort.

We don't know how to show things outside of L. Don't know how to show things outside of P. Except by using the hierarchy theorem. Diagonalization is the only method that we have for showing things are outside of classes.

And there's reason to believe, as well. This we'll get to, I think, next lecture, in fact. There's kind of reasons to believe that the hierarchy theorem type argument, which is diagonalization is not going to answer those kinds of questions. So we need a different method. And diagonalization is all we got.

Good question, though. If I didn't get to answer your question, you have a question for me, ask it again, because it's got buried. So it means we are very far from disproving P versus NP, is that right? It could happen tomorrow.

How can you tell? It doesn't-- it seems clear that the present state of mathematics, as of right now, is we don't have a clue how to answer those kinds of questions. And it's not obvious that we've even made any progress.

But you know, that's the nature of the game. That's the nature of the beast. Somebody gets a good idea, and all of a sudden lots of things can change. And that can happen at any point. Maybe one of you guys.

When would these results first-- the stuff. OK. The hierarchy theorem is old. That goes back to the very-- when time classes were first defined, I think it's one of the first results to show the hierarchy. And that's late '60s.

The NL equal coNL, I think I mentioned, was like mid-1980s. Much later. I mean, from your points of view, it was back in the cave age either way. But yeah, but the hierarchy theorem, that actually predates my coming into the field. But the NL equal coNL, that was something that I personally experienced how surprised people were.

I think I'm going to send you off, all off on your way. But good having you here. And have a good weekend, everybody. And I will see you on Tuesday. Bye-bye.