[CLICKING]

[RUSTLING]

[SQUEAKING]

[CLICKING]

MICHAEL SIPSER: OK. Hi, everybody. Let's get started. So, it's been a while since we came together in a lecture. Last week, we had the holiday. We had the midterm. So with that, what have we been doing? We finished the first half of the course about two weeks ago, where we talked about-- we were talking about computability theory. We have shifted into the second half, talking about complexity theory. So get your mind back to that.

We discussed the various different models and ways of measuring complexity on different models-- at least in terms of the amount of time that's used. And in the end, we settled on the one-tape Turing machine, which is the same model we had been working with in the first half of the course, and argued that though the measures of complexity are going to differ somewhat from model to model, they're not going to differ by more than a polynomial amount. And so, since the kinds of questions we're going to be asking are going to be, basically, whether problems are polynomial or not, it's not going to really matter which model we pick among reasonable deterministic models.

And so, the one-tape Turing machine is a reasonable choice. Given that, we defined time complexity classes, the TIME[T(n)] classes. We defined the class P, which was invariant among all of those different deterministic models in the sense that it didn't matter which model we choose, we were going to end up with the same class P. So that argues for its naturalness. And we gave an example of this path problem being in P. And we kind of ended that lecture before the midterm with the discussion of this Hamiltonian path problem. So we're going to come back to that today.

So today, we're going to look at non-non-deterministic complexity; define the classes' non-deterministic time or NTIME; talk about the class NP, the P versus NP problem-- which one of the very famous unsolved problems in our field; and look at dynamic programming, one of the most basic algorithm - polynomial-time algorithms and polynomial-time reproducibility - moving toward our discussion of NP completeness, which we will begin next lecture. So with that, let's move into today's content, which is, well, just a quick review. As I mentioned, we defined the time complexity class.

The time complexity class is a collection of all of the languages that you can solve in a certain time bound, within a certain amount of time. So the time n-squared, for example, is all of the languages or all of the problems that you can solve in n-squared time. We're identifying problems with languages here. And the class P is the collection of all problems that you can solve or all languages that you can solve in polynomial time. So it's the union over all time n to the k-- so n-squared, n-cubed, n to the fifth power, and so on. Union out of all of those bounds. The associated languages, that's the class P.

And we gave an example, this path problem. We gave an algorithm for path. And then, we introduced this Hamiltonian path problem. So if you remember? Instead of just asking given a graph, whether you can get from s to t, now we want to know can I get from s to t, but visit every other node along the way. So find a path that goes through everything else and gets from s to t. And I should say also it's a simple path, so you're only allowed to go through every node just once. And now, the question for this problem is can we solve that problem in polynomial time.

Can we somehow modify the algorithm for path to give us an algorithm that solves Hamiltonian path in polynomial time? Of course, we could solve Hamiltonian path with an exponential algorithm by trying all possible paths. And that will give a correct algorithm, but there are exponentially many different paths and trying them all will not give a polynomial time algorithm. So the interesting problem is can we solve that without doing that brute force searching through all possible paths. And that's a problem that no one knows the answer to.

Despite lots of effort, people have not succeeded in finding an algorithm for that. But on the other hand, we don't have any idea how do you prove there is no such algorithm. I mean, it's conceivable that one could prove such a thing, but we just don't know how to do it. And so, that problem is an unsolved problem and I just-- this isn't really a note to myself. What's kind of amazing, and this is what we're going to be showing over the next few lectures, that there would be very surprising consequences if you could find a way to solve Hamiltonian path in polynomial time.

Because what that would immediately yield is the polynomial time way of, say, factoring large numbers or solving a large number of other problems that we don't know how to solve in polynomial time. So as we mentioned, factoring is a problem that we only know at present how to solve with an exponential algorithm. And it doesn't seem to have anything to do with the Hamiltonian path problem. Seem very different. But, yet, if you can solve Hamiltonian path in polynomial time, then you can factor numbers in polynomial time.

And so, we'll see how to make that connection. That's what we're building toward with the next few lectures. OK. So happy to take any comments and questions on that, or we'll just move on, if you have any questions on our little review. Well, send questions along and we can stop at the end of various slides to try to answer them. And, of course, write to the TAs, who can take your questions while I'm lecturing. OK. So to start this off, we're going to have to talk about non-deterministic complexity as a variation of deterministic complexity.

So first of all, all of the machines in this part of the course and the languages, everything is going to be decidable and all the machines are going to be deciders. So what do we mean when we have a non-deterministic machine which is a decider? And that just simply means that all of the branches-- it's not just the machine halts on every input, but all of the branches halt on every input. So the non-deterministic machine is non-deterministic, it has lots of possible branches. They all have to halt-- all of them-- on every input. That's what makes a non-deterministic machine a decider.

And you're going to convert a non-deterministic decider into a deterministic decider. But the question is, how much time would that introduce? How much extra time is that going to cost? And the only way that people know at the present time for that conversion would be to do an exponential increase. Basically, to try all possible branches. And that's, of course, very slow. So first, let's understand what we mean by the time used by a non-deterministic machine. And what we mean by the time used is, we're looking at each individual branch individually, separately.

So a non-deterministic machine, we'll say, runs in a certain amount of time if all of the branches halt within that amount of time. So what we do not mean that the total amount of usage, the total amount of effort by adding up all the branches is at most T of n. It's just that each branch individually uses at most T of n. That's just going to be our definition. And it's going to turn out to be the right way to look at this to get something useful. So now we're going to define the analogous complexity class associated to non-deterministic computation, which we'll call non-deterministic time.

So non-deterministic time T of n is the set of all languages that you can do with a non-deterministic machine that runs in order T of n time. Just think back to the definition we had for deterministic complexity, the time class-- or sometimes people call it dtime to emphasize the difference. But let's just say we're calling it in this course time versus ntime. So TIME[T(n)] is all of the language that you can do with the one-tape Turing machine that's deterministic. But this here is a non-deterministic Turing machine for non-deterministic time.

So the picture that is good to have in your head here would be if you think of non-determinism in terms of a computation tree thinking of all the different branches of the non-determinism. All of those branches have to halt and they have to halt within the time bound. So imagine, here, this is T of n time. All of the branches have to halt within T of n steps for this, a non-deterministic Turing machine to be running in T of n time and to be doing a language in the NTIME[T(n)] class.

And by analogy with what we did before, the class NP is the collection of all languages that you can do non-deterministically in polynomial time. So it's the union over all of the ntime classes where the bound is polynomial. OK, so a lot of this should look very familiar, but we've just added a bunch of non-deterministic and a bunch of Ns in place. But the definitions are very similar. And one of the motivations we had for looking at the class P was that it did not depend on the choice of model, as long as the model was deterministic and reasonable.

And the class NP is also going to not depend on the choice of model, as long as it's a reasonable non-deterministic model. So it's again a very natural class to look at from a mathematical standpoint. And it also captures something interesting, kind of, from a practical standpoint - which we're going to talk about over the next couple of slides - which is that it captures the problems where you can easily verify when you're a member of the language. OK, so we'll talk about that. But if you take, for example, the Hamiltonian path problem.

When you find a member of the language, so that is a graph that does have a Hamiltonian path from s to t, you can easily verify that's true by simply exhibiting the path. Not all problems can be verified in that way. But the problems that are in NP have that special feature-- that when you have a member of the language, there's a way to verify that you're a member. So we're going to talk about that, because that's really the key to understanding NP-- this notion of verification. OK, so let me go-- there was a good question here. Let me just see if I want to answer that.

Yeah, I mean, this is a little bit of a longer question than I want to fully respond to but-- well, let's turn to my next slide, which maybe sort bringing that out anyway. Actually, it's a couple of slides from now. But I'll get to that point. So let's look at Hamiltonian, the hampath problem. And what I'm going to show is the Hamiltonian path problem is in NP. And I'm going to walk you through this one kind of slowly. So the Hamiltonian path problem, remember, we don't know if it's in P. But it is in NP. So it's in one of these. You can solve Hamiltonian path in polynomial time if you're a non-deterministic machine.

Why is that? Well, it's because of the parallelism of non-determinism, which allows you to kind of check all of the paths on different branches. So let me first describe how I would write down the algorithm. And then, we'll kind of try to unpack that and understand how that actually looks in terms of the Turing machine's computation. So first of all, taking the Hamiltonian path problem, you are given an input now, which is a graph and the nodes s and t where I'm trying to figure out is that Hamiltonian path-- again, which visits all the nodes, which takes you from s to t.

And we're trying to make now a non-deterministic machine, which is going to accept all such inputs which have a path. So the way this non-deterministic machine is going to work is it's basically going to use its non-determinism to try all possible paths on the different branches. And the way I'll specify that is to say, non-deterministically, we're going to write down a candidate path which is just going to be a sequence of m nodes, where we will say that's the total number of nodes of the graph. Remember, a Hamiltonian path, because it visits every node, is going to be a path with exactly m nodes in it.

So I'm going to write down a sequence of nodes as a candidate path. And I'm non-deterministically going to choose every possible sequence in this way. If you'd like to think of the guessing metaphor for non-determinism, you can think of the non-deterministic machine as guessing the right path, which is going to be the Hamiltonian path from s to t. But I think for this discussion, it might be more helpful to think about all of the different branches of the non-determinism. Because that's perhaps more useful when we're thinking about it in terms of the time.

I think you'll get used to thinking about it. You should be used to thinking about it in many of the different ways. but maybe the computation tree of all branches might be the more helpful one here. So now after we write down a candidate path sequence of nodes, now I have to check that this really is a path. And the way I'm going to do that is to say, well, now if I have just a sequence of nodes written down, what does it mean for it to be a Hamiltonian path from s to t? Well, it better start with s and end with t, first of all.

And we have to make sure that every step of the way is actually an edge. So each pair vi to vi plus 1 has to be an edge in the graph. Otherwise, that sequence of nodes is not going to be a legitimate Hamiltonian path from s to t. And it has to be a simple path. You can't be repeating nodes. These four conditions together will guarantee that we have a Hamiltonian path. And once we have written down a candidate sequence, we can just check that the sequence actually works. At this second stage of the algorithm, non-determinism isn't necessary. This is going to be a deterministic phase.

But stage one of the algorithm is going to be a non-deterministic phase where it's writing down all possible paths. Now, I'm going to try to unpack that for you so you can actually visualize how the machine is doing this. And then, of course, you know on each branch of the non-determinism, you're going to check to see whether the conditions have been satisfied. And on that branch, if the conditions were not satisfied, that branch is going to reject. Of course, one of the other branches might yet accept, so that's how non-determinism works. OK?

So I'd like to visualize this as the tree of the different branches of the computation of m on its input. So here is our non-deterministic Turing machine. Which? This one. And you provide it with the input G, s, and t. And how is the machine actually working? So when I say non-deterministically write down a sequence of m nodes-- look, this is getting into a little bit more detail than I would normally think about it, because we try to tend to think about things at a higher level. But just to get us started, I think it's good to think about this with a bit more detail.

So let's think of the m nodes as being numbered, having labels numbered 1 through m. And I'm going to think about them being labeled by their binary sequences. We're going to write down those nodes. That's how the machine is going to have to operate on those numbers for the nodes. We'll think about them as being written in binary. And now, as the machine is going to be writing down, let's say, the node v1. So it's non-deterministically picking the first node of the sequence. What does that actually mean in terms of the step-by-step processing of the Turing machine m?

Well, it's going to be guessing via a sequence of non-deterministic moves, the bits that represent the number of the node for v1. For example, v1 might be the node number 5 in the graph. Of course, non-deterministically, the machine is on different branches, picking all different possible choices for v1. Those are going to be the different branches here. But one of the branches might be-- and what I'm really representing here, these are-- I probably could have written this down on the slide here in tiny font.

But these are like the 0, 1 choices. That's why it's sort of a binary tree here for writing down the bits of v1. So here maybe this could be 101, representing the number 5, which might be the very first node that I'm writing down in my sequence. Some other branch is going to write down node number 6. Some other branch is going to write down node number 2. Because non-deterministically, we're making all possible choices for v1. That's what I'm trying to show in this little part of the computation of m on this input.

So then, after it's finished writing down the description of the node for its choice for v1, it goes down to choose what v2 is. Again, non-deterministically, so there's going to be more branches for each possible choice of v2. And so on, node after node. Then, it's going to finally get to the last node, vm. It's going to write down lots of choices for vm. And at this point here, we have completed the first stage of the algorithm. Now, there's some huge tree of all of the possible choices for the V's that have shown up at this point. OK?

And now, we're going to move into the second phase. So following this, there's going to be, here, a bunch of deterministic steps of the machine. So no more branching is needed because here, we've written down-- at this point, we've reached a point in each of those locations, where we've chosen one of the candidates, one of the possible branches-- one of the possible paths through the graph, I'm sorry. So here, we're guessing potential paths in the graph.

And now, we're going to check that we actually have picked a path that's a Hamiltonian path from s to t. OK? So each one of these branches is now going to end up accepting or rejecting. And the whole overall computation is going to accept if at least one of them ended up accepting, which means you actually found a Hamiltonian path. OK? I don't know if that's helpful to you or not, but that is-- if there's any questions on this, let's see. Question on is there something-- trying to draw a connection here between this and computation histories.

I mean, there is a pattern here that does come up often where you want to check that something starts right, ends right, and that all of the intermediates are right. So I think there is some deeper connection here. Probably too hard to explain but that has something to do with this Hamiltonian path problem. Why are we using binary representation? Well, we're going to talk about-- the algorithm would have worked equally well if we used base three or base five or base 20 as a way of writing down our labels for the nodes. But in a sense, it doesn't matter.

The alphabet has to be finite though, so that's true. I mean, that's why it's not just in a single step of the Turing machine that you would pick the node the choice for v1. You really have to go to a sequence of steps. Because each of the branches of the machine only has a fixed number of choices. So you can't, in a single step of the Turing machine, pick all the different possibilities for v1. That has to go through a sequence. OK. Now, let me do a second example, the problem of composites. So the language of all composites are all of the non-primes, written as binary numbers again.

So we'll talk about the base and the representation in a second. But just imagine these are all of the numbers that are not prime. And that language is easily seen to be a member of NP. Here is, again, the algorithm for that. Given x, we want to accept x, if it's not a prime number. So it has some non-trivial factor. So first, the way the non-deterministic machine is going to work is it's going to guess that factor. So non-deterministically, it's going to try every possible factor.

Y is going to be a number between 1 and x. But not including 1. You have to be an interesting factor, so not including one in the number itself. So something strictly in between. And we're going to then-- after we've non-deterministically chosen y, then we're going to test to see if y is really a factor. So we'll see if y divides evenly into x with a remainder of 0. If that branch successfully picked the right y, it's going to accept. And some other branch where it might have picked the wrong y, will not. And if x is really a composite number, some branch will find the factor.

Now, the base doesn't matter. Could have used base 10, because you can convert from one base to another. So this is really in terms of our representation of the number. But I do want to make one point here, that changing-- we don't want to write the number in unary-- writing the number of k as a sequence of k1s. That's not really a base. That's just an exponential representation for the number and that changes the game. Because if you make the input exponentially larger, then it's going to change whether the algorithm relative to that exponentially larger input is polynomial or not.

So an algorithm that might have been exponential originally when the number's written in binary might become polynomial if the numbers are written in unary. And I do want to mention as a side note, that the composites language-- or primes, for that matter-- both are NP. But we won't cover that. So whereas the Hamiltonian path problem is not known whether it's NP, the primes and composites problem are NP. So that was known. That was actually a very big result in the field. Solved by folks at one of the Indian Institute of Technology back about almost 20 years ago now.

So let's turn here, to trying to get an intuitive feeling for P and NP. And we'll return now to this notion of NP corresponding to easy verifiability.

NP are the languages where you can easily verify membership quickly. I'll try to explain what that means. In contrast, P are the languages where you can test membership quickly. By quickly, I'm using polynomial time. That's going to be, for us, that's what quickly means in this course.

In the case of the Hamiltonian path problem, the way you verify the membership is you give the path. In the case of the composites, the way you verify the membership is you give the factor.

In those two cases, and in general, when we have a problem that's in NP, we think of this verification as having-- we give it a special name, called a certificate, or sometimes a short certificate, to emphasize the polynomiality of the certificate. It's like a way of proving that you're a member of the language. In the case of COMPOSITES, the proof is the factor. In the case of HAMPATH, the proof is the path, the Hamiltonian path.

Contrast that, for example, if you had a prime number. Proving a number is composite is easy because you just exhibit the factor. How would you prove that a number is prime? What's the short certificate of proving that some number has no factor? That's not so obvious.

In fact, there are ways of doing it, which I'm not going to get into in the case of testing of numbers prime. And now it's even known to be in P, so that's even better. But there's no obvious way of proving that a number is prime with a short certificate.

This concept of being able to verify when you are a member of the language, that's key to understanding NP. That's the intuition you need to develop and hopefully take away from today's lecture, or at least by thinking about it, reading the book, and so on.

If you compare these two classes, P and NP. P, first of all, is going to be a subset of NP, both in terms of the way we defined it because, deterministic machines are a special case of non-deterministic machines. But also if you want to think about testing membership, if you can test membership easily then you can certainly verify it in terms of the certificate. You don't even need it. The certificate is irrelevant at that point. Because whatever the certificate is, you can still test yourself whether the input is in the language or not.

The big question, as I mentioned, is whether these two classes are the same. So does being able to verify membership quickly, say with one of these certificates, allow you to dispense with the certificate? Not even need a certificate and just test it for yourself whether you're in the language. And do that in polynomial time. That's the question.

For a problem like Hamiltonian path, do you need to search for the answer if you're doing it deterministically? Or can you somehow avoid that and just come up with the answer directly with a polynomial time solution? Nobody knows the answer to that. And it goes back, at this point, quite a long time. It's almost 60 years now. That problem has been around 60 years. No, that would be 50 years. No, 50 years, almost 50 years.

Most people believe that P is different from NP. In other words, that there are problems in P-- in NP which are not in P. A candidate would be the Hamiltonian path problem. But it seems to be very hard to prove that.

And part of the reason is, how do you prove that a problem like Hamiltonian path does not have a polynomial time algorithm. It's very tricky to do that, because the class of polynomial time algorithms is a very rich class. Polynomial time algorithms are very powerful. And to try to prove-- there's no clever way of solving the Hamiltonian path problem. It just seems to be beyond our present day mathematics. I believe someday somebody's going to solve it. But so far, no one has succeeded.

So what I thought we would do is-- I think I have a check-in here. Yes. And then we'll stop for a break.

Let's look at the complementary problem, HAMPATH complement. You're in the language now if you don't have a path. So is that complementary problem in NP?

For that to be the case, we would need to have short certificates of when a graph does not have a Hamiltonian path. I leave it to you. There are three choices. OK? Going to stop here, so make sure you get your participation credit here. I'm going to end the polling now.

Interesting. [LAUGHS] So the majority is wrong. Well, not wrong, we don't know.

I think the only fair answer to this question is C. Because we don't know whether or not we can give short certificates for a graph not to have a Hamiltonian path. If P equaled NP, then you can test in polynomial time whether a graph has a Hamiltonian path. And then the computation itself would be a certificate, whether it has a path or whether it doesn't have a path. Because it would be something that you can check easily.

Since we don't know for sure that P is different from NP, P could be equal to NP, then it's possible that we could give a short certificate. Namely, the computation of the polynomial algorithm. So the only really reasonable answer to this question is that we don't know.

Just ponder that. Those of you who answered yes, however, need to go back. And I put this here explicitly because I know this is a confusion for, well I can see, for quite a few of you.

When we have non-deterministic computation and you have a non-deterministic machine, you can't simply invert the answer and get back a non-deterministic machine. Non-determinism does not work that way. If you remember, the complement of a pushdown automaton is not a pushdown automaton. If you have a non-deterministic machine and you invert all of the responses on each of the branches, it's not going to be recognizing or deciding the complementary language.

I think that this is something you-- if you answered yes, you need to go back and make sure you understand why yes is not a reasonable answer to this question. Because that's not how non-determinism works. So you have a not complete understanding of non-determinism. And that's going to be really important for us going forward. I really urge you to figure out and understand why yes is not a good answer to this check-in.

OK, so I think we will-- we can talk about that more over the break. And so we'll return here in five minutes.

Somebody's asking about-- can infinite sequences be generated by the machine. When we're talking about, especially in the complexity section of the course, all of the computations are going to be bounded in time. So we're not going to be thinking about infinite runs of the machine. That's not going to be relevant for us. So let's not think about that.

How does a Turing machine perform division? How does a Turing machine perform division? Well, how do you perform division? [LAUGHS] Long division is an operation that can run in-- the long division procedure that you learn in grade school, you can implement that on a Turing machine. Yes, a Turing machine can definitely perform-- do long division, or division of one integer by another in polynomial time.

Another question. Can we generally say, try dividing y by x? Or do we have to enumerate a string of length y and cross off every-- no. I think that's the same question. I mean, if you have numbers written in binary, how would you do the division? You're not going to use long division. Anything such as the thing that's proposed here by the questioner is going to be an exponential algorithm. So don't do it that way.

Why does primes in P-- composites in P not imply primes in P? It does imply. If composites are in P, then primes is in P as well. When you have a deterministic machine, you can flip the answer. When you have a non-deterministic machine, you may not be able to flip the answer. So deterministic machine just having a single branch, you can make another deterministic machine that runs in the same amount of time that does the complementary language. Because for deterministic machines, just like for deciders in the computability section, you can just invert the answer.

There is an analogy here between P and decidability, and NP and recognizability. It's not an airtight analogy here, but there is some relationship there.

Somebody's asking me, what are the implications of P equal to NP? Lots of implications. Too long to enumerate now. But, for example, you would be able to break basically all cryptosystems that I'm aware of, if P equal to NP. So we would have a lot of consequences.

Somebody's asking, so composites-- primality and compositeness testing is solvable in polynomial time. But factoring, interestingly enough, is not known to be solvable in polynomial time. We may talk about this a little bit toward the end of the term if we have time.

The algorithms for testing whether a number is prime or composite in polynomial time do not operate by looking for factors. They operate in an entirely different way, basically by showing that a number is prime or composite by looking at certain properties of that number. But without testing whether it has-- testing, but without finding a factor.

Another question here about asking when we talk about encodings, do we have to say how we encode numbers, values? No, we don't have to. We usually don't have to get into spelling out encodings, as long as they're reasonable encodings. So you don't have to usually. We're going to be talking about things at a high enough level that the specific encodings are not going to matter.

Let's return to the rest of our lecture. When we talk about, say, this P versus NP problem. And how do you show that a problem might not be solvable in P, like the Hamiltonian path problem?

Many people who are not practitioners in the field, know about the P versus NP problems-- over the years, I've gotten many, many emails and physical letters from people about that. Since I've spent some time thinking-- I'm known as having spent some time thinking about it.

People claim to solve the problem, solve P is NP, the P versus NP problem, by basically saying, problems like Hamiltonian path or other similar problems, basically there's clearly no way to solve them without searching through a lot of possibilities. And then they go through a big, long analysis showing that there are exponentially many possibilities.

A lot of the proofs that claim to solve P versus NP, they all look like that. You only have to look-- somewhere in that paper, there's going to be a statement along the lines, "to solve this problem, clearly you have to do it this way."

And that's the flaw in the reasoning. Because just like for the factoring problem-- just like for the compositeness testing problem, you don't necessarily have to solve it by searching for factors. There might be some other way to do it. You might be able to solve the Hamiltonian path problem without searching for Hamiltonian paths. There might be some other process that you can use, which would give you the answer.

The class of polynomial time algorithms is very rich, can do many, many things. And I wanted to present to you one of the most important polynomial time algorithms. In a sense, you can make a certain argument that this is the most fundamental polynomial time algorithm. Some people might argue with me on that. And that's a process called dynamic programming, which I'm sure some of you have seen already in your algorithms classes, and some of you may not have.

Since you have a homework problem on it, I want to spend a little time describing it to you. And that's useful for solving this A CFG problem, which you may remember from the first half of the course, involving testing if a grammar generates a string.

So you remember this A CFG problem? You're giving a grammar, context-free grammar, and a string. And I want to know, is it in the language of the grammar. So that's going to turn out to be solvable in polynomial time, but only with kind of a clever algorithm.

Remember, it's decidable. We decided it by making sure that you are converting the grammar in Chomsky normal form. Then all the derivations have a certain length. You just try all the possible derivations of that length, and you accept if any of those derivations generate w. You may remember that from the first half of the course.

That immediately gives an NP type algorithm for this language. Because basically, you non-deterministically-- instead of trying the most sequentially, all of these derivations, you try them in parallel non-deterministically. So non-deterministically, you pick some derivation of that length. And you accept it if it generates the input.

This classically fits our model of NP. You can think of it as guessing the derivation and checking that it works. Or, in parallel, writing down all possible derivations. But this A CFG problem is classically an NP problem, a problem in NP.

And if you just imagine-- then what's going to be the certificate? If you found an input that's in the language, that's generated by the grammar, the certificate is the derivation.

So if you look at it that way, you might think, well, that's the best you can do. This is going to be an NP problem, in NP, and is not going to be a way of avoiding searching for the derivation. But that's not true. There is a way of avoiding searching for the derivation. You can build up the derivation using dynamic programming. And so that's what I wanted to describe for you, how that works. Also partly because it's a homework problem. And I think dynamic programming is a very important algorithm.

Before we describe what dynamic programming is, which is very simple, by the way, let's try to work up to it by making an attempt to solve this problem just using ordinary recursion. How would we solve the A CFG problem? So you're given grammar, you're given an input. Let's assume the grammar is in Chomsky normal form. That's going to be useful. So it's a Chomsky normal form grammar. And we want to see how to test if you can generate w.

And it's going to be recursive algorithm. The recursive algorithm is going to actually solve something slightly more general. I'm going to give you the grammar. I'm going to give you the string. And I'm also going to allow you to start at some other variable besides the start variable. I'm going to give you some variable, R, and I know I want to know, can I generate w starting at R? So that's my slightly more general problem, which is going to be useful in the recursion.

So the input now to this algorithm is the grammar, the input, and the starting variable. And now how is the algorithm going to work? It's going to try to test, can I get to-- is there some derivation, pictured here as the parse tree, for w starting at R? That's what the algorithm is trying to answer. Can I get w from R?

The way it's going to do that is it's going to try to divide w into the two strings in all possible ways. Which sounds like it might be exponential, but it isn't. There's only a polynomial number of ways to divide the string into two substrings. Just of order n, just depending where you make that cut. So that's not too bad. There's a polynomial, there's only n ways of making that division.

And also I'm going to try every possible rule that comes from R that generates two variables. So these are what's allowed in Chomsky normal form, R goes to ST.

For each possible way of cutting w into x, y, and for each possible rule, R goes to ST, I'm going to see, recursively, can I get from S to x, and from T to y.

So I'm going to use my recursion now. Now that I have smaller strings instead of w, I can apply recursion and try to answer it that way. And this algorithm will work. If I found a way to cut w into x, y, and I found a rule, R goes to ST such that S generates x and T generates y, then I'm good. I know I can generate w from R.

And if there's no way of cutting w up to satisfy that, or if I can't find any way to divide w into x, y, and a rule R goes to ST which makes this work, if all possible ways fail then you can't get from R to w.

And then you can decide the original A CFG problem now, by starting from the start variable, instead of just any old R. You plug in the start variable for R.

So this algorithm works, and it can be used to solve A CFG. But the question is, is it polynomial? And it's not. Because every time you're doing the recursion, you're essentially adding another factor of n. Because here, as we pointed out, this is a factor of n. But that's happening every time you're doing a recursive level.

And you can imagine, I'm just doing a very crude analysis here, depending upon how you divide w up. But roughly speaking, it's going to get divided in half each time, so there's going to be log n levels. So that means you're going to be multiplying n by itself log n times, or give you an n to the log n algorithm. That's not polynomial, because polynomials end with a constant, for some fixed constant. n to the log n is not going to be polynomial. This is not a polynomial algorithm.

Instead, you're going to have to do something just a little bit more clever. It's going to be the same basic idea, but relying on one little observation. And the little observation is that when-- this non-polynomial implementation that I've just described is actually pretty dumb. Because it's doing a lot of recomputation of things that it's already solved.

Why is that? Because if you look at the number of possible different subproblems here, once I give you G and I give you w, how many different subproblems of G, S, and T are there?

The number of strings here, all of those strings are going to be substrings of w. There's only roughly n squared substrings of w. I'm always going to be generating, in a subproblem here, some substring of w from some starting variable in the grammar. So because there aren't that many different substrings, and not that many different starting variables, the total number of possible problems that this algorithm is going to be called on to solve is going to be, in total, a polynomial number of different subproblems. There aren't very many of them. It's only something like order of n squared.

So if the algorithm is running for exponentially long time, it's solving the same subproblem over and over again. That's dumb. Why don't you just remember when you solve the subproblem, so you don't solve it again? Doing that enhanced recursion where you remember the problems you've already solved, it's called dynamic programming. I don't know why it has such a confusing name like that. Actually it's called by several different things. But anyway, that's known as dynamic programming. It's recursion plus the memory. And here is just repeating myself.

There are not very many different substrings, so every time in your recursion somewhere, you're going to be working with a substring. So there's not that many different subproblems that you can possibly solve. And just remember when you solve the subproblem, and not solve it again.

Let me just show you that algorithm again here, with the little modification. So first of all, let me give you-- this is the same algorithm from the previous slide. I'm just repeating it here without all the other stuff so we can look at it directly. Dividing it into x and y for each possible rule. And then recurse.

I'm going to add a little step 0 beforehand, which says, if I have G, w, and R, let me just check first if I've already solved that one before. So I have to keep track of the ones I've already solved. That's not too bad, because there's only order n squared possible different ones that I could be called on to solve. So I'm just going to have a little table where I'm going to remember those. And then every time I get a new one, I get one to solve, I'll check. Is it on that table? And what's the answer? So I won't have to rerun those.

I'm going to be basically pruning that tree so that it has only a polynomial number of leaves. And so the total size of that tree now is going to be polynomial. And so that's going to yield a polynomial running time. This, by the way, I only learned this myself, I'm sure you guys all know this who have taken this in the algorithms course, has a special name called memoization. Not memorization. Came from the same root, I think, but memoization, which is somehow remembering the results of a computation so you don't have to repeat it.

The total number of calls is going to be, at most, n squared, to this algorithm, because you're never going to be redoing work that you've done already. And when you actually have to go through it, the running time-- the total amount of time that you're going to need is going to be polynomial altogether.

I don't remember what my check-in is on this. Oh yeah. This is somehow related. And feel free to ask questions too, while you're thinking about this check-in. But the check-in says here, we've solved the A CFG problem in polynomial time. Does that tell us that every context-free language itself is also solvable in polynomial time?

Just mull that over, and please give me an answer to it. I hope you do better on this check-in than you did on the last one. But anyway, why don't you go ahead and think about that. I can take some questions in the meantime.

Somebody is asking here-- actually, I'm getting several questions on this. Why isn't it order n cubed or something greater than order n squared because of the variables? The variables don't depend on n. When you're given-- well, actually that's not true. No. You are right. Because the grammar is part of the input. So you might have as many as n different variables in the given grammar. So you are right.

There is potentially-- the grammar might be half the size of the input, and the input to the grammar w might be half the size of the input. So I didn't think about that, but you're correct.

There are potentially different numbers of variables in different grammars, so you have to add an extra factor, which would be at most the size of the input, because that's as many variables as you could possibly have. So it really should be, I think, order n cubed to take that into account as well. Plus all of the work that needs to happen in terms of dividing things up.

On a one-tape Turing machine, there's going to be some extra work just to carry out some of these individual steps, because with a single tape things are sometimes a little awkward. I think the total running time is going to end up being something like n to the 4th or into the 5th on a one-tape Turing machine. But that's a good point.

Somebody's saying, how can we be storing n squared strings in finite time? I'm not saying finite time. We have polynomial time. Every stage of this algorithm is allowed to run for polynomially many steps. As long as it's clearly polynomial, we can just write that down as a single stage.

Part two should say-- oh. There's a typo here. So use D. Thank you. That is a typo. I'm afraid if I change it on my original slide here, things will break in some horrible way. Let's just see. Did I completely wreck my slide? No, that's good. Yeah, thank you. Good point. Oops.

OK, how's our check-in doing? I think you're just about all done. Spent a lot of time on this. End polling.

As you may remember from the first half of the course-- so the answer is A, indeed. Remember that we showed A CFG is decidable, and therefore each context-free language itself is decidable, just because you can plug in a specific grammar into the A CFG problem. The very same reasoning works here.

If you have a context-free language, it has a grammar. You can plug that grammar into the A CFG problem. And then, that's polynomial time, you're going to get a polynomial time algorithm for that language. Good to review that. It's the same thing, same argument we used before.

I don't want to spend a lot of time on this. There's another way of looking at dynamic programming. We'll talk about this again maybe in a lecture, probably next lecture, just because I you have a homework problem on it. If you've seen dynamic programming before, this is going to be easy. If you haven't seen it before, it's going to be, I think, probably a little challenging.

Another way of looking at dynamic programming is the so-called bottom-up version of dynamic programming. And what that would mean is, you solve all of the subproblems first. You solve all the smaller subproblems before you solve the larger subproblems.

It's here on the slide. I'm not sure I want to talk it through. But basically, you solve the subproblems here where, start with strings of length 1, and then from that you build up to subproblems with the substrings are of length 2, and then 3, and so on. And each of those only relies on the smaller previously solved subproblems. So you can, kind of in a systematic way, solve all the larger and larger subproblems for larger and larger substrings.

That gives kind of a different perspective on dynamic programming. And for different problems, sometimes it's better to think about either this sort of top-down recursion based process, or the bottom-up process that I'm describing here. They're really completely equivalent.

The version that's described for this particular algorithm, which appears in the textbook, is actually the bottom-up algorithm. So you shouldn't be confused if you see something there which looks somewhat different. You basically solve all possible subproblems, basically filling out a table. Let me not say anything more about that here, since we're running a little short on time. There are really two perspectives on dynamic programming.

So moving on from there, let's shift gears. Leave context-free languages and dynamic programming behind. And so I'm moving toward understanding P and NP. And for that, we will introduce a new problem called the satisfiability problem. And that's one we're going to spend a lot of time on. If you tuned out a little bit during the dynamic programming discussion, time to get back on board.

The satisfiability problem is going to be a computational problem that we're going to be working on. And it has to do with Boolean formula. So these are expressions, like arithmetical formula, like x plus y times z, but instead of using numerical variables, we're going to be using Boolean variables that take on Boolean values, true, false. Or sometimes represented by 1 and 0.

The operators that we're going to be using are going to be the and, or, and negation operations. And, or, not.

I'm going to say such a formula, such a Boolean formula, we're going to call it satisfiable-- we'll do an example in a second-- if that formula value evaluates to true if you make some assignment of values to its variables.

So just like arithmetical formula will have some value if you plug in values for the variables, Boolean formula is going to have some value if you plug in Boolean values for its variables. And I want to know, is there some way to plug in values which makes the whole thing evaluate to true. The formula itself is going to evaluate to either true or false, and I wanted to evaluate to true.

Here is our example. Let's take the formula, phi, which is x or y, and x complement-- or, not x or not y. So the notation x with a bar over it, x complement, is just x bar, not x. It's just the way if you're familiar with the other notation, the not operation, which just inverts 1s and 0s. We're going to write it with a bar instead of the negation symbol.

I'm assuming that you've all seen Boolean algebra, Boolean arithmetic before, where the and operation is only true if both inputs are true. These are going to be binary and operations and binary or operations. Or is going to be true if either input is true. And not is true if its single input is false. Oops, just looked at the answer.

Here I want to know, for this Boolean formula, here is it satisfiable. Is there some way to assign values to the variables to make this formula evaluate to true? So for example, let's just try things.

Let's make x and y both true. So x is true and y is true. So x or y, well that's good, that's true. But now we have to do an and, so we need both sides to be true. So now we have x complement-- well we said we said x is true, so x complement is false, y complement is false. False or false is false. So now we have a true and false. That's going to be false. We did not find a satisfying assignment. But maybe there's another one. And in fact, there is.

If you make x true and y false, then both of these parts will evaluate to true, and then you'll have true and true. So we found a satisfying assignment to this formula. It is, in fact, satisfiable. So if you say x is 1 and y is 0, yes. This is satisfiable.

Now the problem of testing for a Boolean formula, if it is satisfiable, is going to be the SAT language. It's a set. It's a collection of satisfiable Boolean formula. And testing whether you're in SAT or not is going to be an important computational problem.

There was an amazing theorem which really got this whole subject going, discovered independently by Steve Cook in North America and Leonid Levin in the former Soviet Union, almost exactly at the same time, which made a connection between this one problem and all of the problems in NP.

By solving this one satisfiability problem in polynomial time, it allows you to solve all of the problems in NP in polynomial time. So if you could solve this problem set in P, then Hamiltonian path is also solvable in P. If you step back and think about that, it's kind of amazing.

And the method that we're going to introduce is called polynomial time reducibility. Let's do a quick check-in on this. This should be an easy one. Why don't you just think about, is SAT, the SAT problem that we just described here, is that in NP? Three seconds. You all there? OK. Ending polling.

Hopefully you're getting the intuition for NP that these are the problems-- to be in NP means that when you're a member of the language, there's a short proof or a short certificate of membership. And in this case, the short certificate that the formula is satisfiable is the assignment, which makes it true, also called the satisfying assignment.

So yes, this is an NP language, language that's in NP. There are a lot of things that we don't know in the subject, but this isn't one of them. We do know that SAT is in NP.

So let's talk about our method for showing this remarkable fact that, if you can solve SAT in polynomial time, then all of NP is solvable in polynomial time. And it uses this notion of polynomial time reducibility, which is just like mapping reducibility that I hope you've all grown to know and love in the first half of the course. But now, the reduction has to operate in polynomial time.

So it's the same picture that we had before, mapping A to B, transforming A questions to B questions. But now the transformation has to operate quickly. And we get that if A is polynomial time reducible to B, and B is polynomial time, then A is also polynomial time. Same pattern as before. If A is reducible to B and B is easy, then A is easy.

Here is kind of the essence of the idea, or at least the outline of the idea of this Cook and Levin theorem. That if satisfiable is in P, then everything in NP can be done in P. Which is because, we will show that all problems in NP are polynomial time reducible to SAT. That's the amazing fact.

So therefore, if you can bring SAT down into P by using this reduction, it brings everything else along with it, everything is reducible to SAT. So we just have to show how to do that.

There is an analogy that we had in the first half of the course, in one of our homework problems, if you may remember. We showed that A TM has the very special property that all Turing recognizable languages are mapping reducible to it. I think that was problem 2, or 2a, either in problem set 3 or problem set 2. I think problem set 3. That every Turing recognizable language is polynomial time reducible to A TM. And so, very similar picture. And there's a lot of analogies here that you can draw between the computability section and the complexity section.

With that, I know we're just about out of time. So let's just quick review of what we've done here. I will stick around for questions for a while. Is there a-- OK, that's a good question.

Is there a regular reduction analogy version to mapping reducibility? We had the general reduction for the computability section. And we had the mapping reduction for the computability section. Here, we're only going to be focusing now on the mapping reduction. So polynomial time reduction is, by assumption, going to be a mapping reduction.

Yes, there is a general polynomial time reduction notion as well. This is not required, but if you are curious about the general reduction and how to precisely formulate that, it actually appears in chapter 6 under Turing reducibility. That's the general notion of reducibility spelled out in a formal way. And there's polynomial time Turing reducibility as well. We're not going to talk about it in this course.

Other questions? Does NP correspond exactly to verification in polynomial time? For me to answer that as a precise question, we have to have a precise definition of verification. But with the right definition, the answer is yes. So you can define a verifier as a polynomial time algorithm that gives a certificate, that takes a certificate and an input to the language, and will accept if that certificate is a valid certificate for that input.

This is actually discussed in chapter, I think, 9 of the text. Now I'm forgetting already, what's where in the book. But yeah, you can think of NP in terms of verification as the definition.

Is proving P equal NP the same as proving that a polynom-- actually, I can even make the-- if you want, you can post public comments too. I should have done that in other cases. Is proving P equal NP the same as proving that a polynomial time non-deterministic Turing machine N has a polynomial time deterministic? Yeah.

Suppose we prove that P equals NP, which is the minority view, I would say, the small minority view. There are some people who believe that that is entirely possible, and might even be the case. But that's a very small group.

But yeah, if you prove P equal NP, that's the same as saying that every non-deterministic polynomial time Turing machine is going to have a companion deterministic polynomial time Turing machine which does the same language. That's exactly what it means.

Bye bye, everybody.