[SQUEAKING]

[RUSTLING]

[CLICKING]

MICHAEL SIPSER: So welcome, everybody. Welcome back. Let's get started with today's lecture. Where were we? On Tuesday, we covered, the main topic was the recursion theorem, which allows programs to self-reference. And we saw some applications of that, too.

So we gave a new proof that ATM is undecidable. We looked at this language of minimal Turing machine descriptions. And we had a short digression into mathematical logic, where we looked at how one shows that there are true, but unprovable, statements in any reasonable formal system.

So today, we're going to shift gears entirely. And we're moving into the second half of the course, where we are beginning a study of computational complexity theory. And we'll say a little bit about that during the course of the lecture, of course. But the main things that we are going to cover in terms of content that you will need is defining the complexity classes and the class P. And we'll prove a few theorems along the way. But that's the main objective of today's lecture.

Computability theory, which was the subject of the first half of the course, and which is what the midterm exam is going to cover, was a subject that was an active area of mathematical study in the first part of the 20th century. It really got-- it really dates back into the late 19th century, in fact, when people were trying to figure out how to formalize mathematical reasoning.

But it really got going in the 1930s with the work of Godel, and Church, and Turing, who really formalized for the first time what we mean by algorithm. And that allowed the study of algorithms to really get started. And it had its impact, as I mentioned, on the actual design, building, and thinking about real computers. The main question, if you kind of boil the subject down to a single question, is some language decidable or not.

In complexity theory, which got started kind of when computability theory more or less wrapped up as a subject, largely because they answered many of the questions that they-- they answered pretty much all of the questions that they were asking. So there really aren't interesting unsolved questions left in that field. And you really need mathematical questions to keep a subject alive, unsolved questions.

So complexity theory got its start in the 1960s. And it continues on as an active area of research to the present day. And I guess if you could boil it down, it would be is a language decidable with some restriction on the resources, such as the amount of time, or memory, or some other-- or some other kinds of resources that you might provide, randomness, and so on? All of those are within the area of computational complexity theory.

So let's get ourselves started with an example. Here is the language that we've looked at in the past, a to the k, b to the k. And let's look at it now from the perspective of complexity. All of the languages that we're going to be studying in complexity are all going to be decidable languages. So the question of undecidability in complexity theory is not really of interest. It's all decidable languages, but the question is how decidable. What sort of resources do you need to do the deciding?

So for this language A, how many steps are needed? Well, we're going to spend a little time just kind of setting up the definitions of the subject and kind of motivating them. So for this language A, when I ask how many steps are needed, well, it's going to depend upon which input you have in mind. Some inputs might require more steps than others.

So the way we're going to set the subject up, which is the standard way that people in this field look at it, and I think that applies to lots of examples outside as well, is that we're going to kind of group all of the inputs of the same length together. And look at the maximum cost, the maximum number of steps you need, to solve any one of those inputs of a given length. And we'll do that for each length.

And the way we're going to frame it is in terms of giving a maximum, or what's called an upper bound, on the amount of time that you need to solve all of those inputs of length n. That's what's sometimes called worst-case complexity. I'm sure many of you seen this already. But just to make sure we're all together on this, you might contrast that, for example, with what's called average case complexity.

Where instead of looking at the most difficult case among all inputs of length n, you take the average of all inputs of length n. And then you have to-- then it's a little bit more complicated, because then you need to have a probability distribution on those inputs and so on. We're not going to look at it, in this course, from that perspective. We're only going to be looking at what's called worst-case complexity.

So let's begin, then, by looking at this in more detail and taking as our starting point the theorem that says that on a one tape Turing machine, which is deciding this language A, a to the k, b to the k, you can do this on a one tape Turing machine, M, we're calling it. In at most some constant times n squared steps, for any input of length n, where the constant is going to be fixed independent of it.

So this is going to be-- having a constant in-- factor in the complexity is going to come up often. And so instead of saying this over and over again, we're going to use a notation that m uses order n squared steps. I'm sure many of you seen that terminology as well. But just for the purposes of making sure we're all together on that, there is this big O and little o notation. I'm expecting you to be familiar with that.

Big O is when you apply to functions, as it's done. You say f is big O of g, as for two functions f and g. It's basically if f is less than or equal to g, if you're ignoring constant factors. And you say f is little o of g if f is strictly less than g if you're ignoring constant factors. That's kind of one sort of informal way of looking at it. The precise definition is given up there on the slide.

And if you haven't seen it before, make sure you look at it in the book, where it's all carefully described and defined. So that you're comfortable with these notions. Because it's really-- we're going to be using this without any further discussion from here on. So let's get to the proof, then, of this theorem that you can do the language A in order n squared steps.

Not super hard. I think if I asked you to come up with an algorithm to solve A, this would be the algorithm that you would find basically. First, you would start off by scanning the input w to make sure it's of the right form. So a run of a's followed by a run of b's of some lengths. And if it's not of that form, then you're going to reject right away.

The next thing you'll do is then go to a repeat loop in the Turing machine. And if you imagine, here is your machine, here is the input w, you're going to go through that repeating repeatedly. Of course, you can do this in a number of different ways. But here's the way I have in mind for you, on this slide, anyway. We're going to scan the entire tape, crossing off a single a and a single b on a scan. And then you're going to keep doing that until you've crossed off everything.

Unless you run out of a's or you run out of b's. In that case, you're going to reject. If you run out of a's or b's before you run out of the other type, then you know that you started out with an unequal number. And so the machine is going to reject. If you've managed to cross them all off without running out of one before the other, then you'll accept. I know this is kind of obvious. But I think it's important to get us all together on this at the beginning. So here's a little animation of the Turing machine doing that.

I'm not showing the motion of the head. But you imagine the head scanning back and forth, crossing off these a's and b's, one a and one b on each pass, until they're all crossed off. And then it accepts. Unless of course, it runs out of a's or b's before the other type, then it rejects.

OK, now let's do a very quick, informal analysis of how much time this has taken. So the very first stage, I'm calling each of these things stages of the Turing machine to distinguish them from the steps of the Turing machine, which are the individual transition function moves. So this is the entire stage of the machine.

The very first stage takes order n steps, because you have to make a scan across the input. And then I'm not giving all the full detail. Of course, you're going to scan and then you're going to return the head back to its starting position. I'm not-- that's just going to be an extra factor of n, an extra n. And so that's where we're talking about being order n steps for the very first stage.

And then as you go through the repeat loop, each time you go through the repeat loop, you're going to cross off one a and one b. So that's going to mean you're going to have to do this roughly order-- you're going have to do this order n times in order to cross off all the a's and b's. So there's going to be order n iterations of this repeat loop. Each one of them is going to, again, require a scan. So that's order n steps for each one of the iterations.

So adding that all up, the very top row gives us the order n and then we have the order n iterations times order n steps is order n squared steps. And so the sum of these two is order n squared steps due to the nature of arithmetic when you have the big O notation. The dominant term overrides all of the others.

So that's, in a nutshell, how we-- well, this is our very first example of analyzing the Turing machine algorithm for a language. So let me ask you now whether there is some other Turing machine, some other one tape Turing machine which can do better than this Turing machine does in terms of how much time it takes.

So one idea that you might have, instead of crossing off a single a or a single b, maybe you can cross off two a's and two b's, or 10 a's and 10 b's. Well, that'll cut down the running time by a factor of 10. But from the standpoint of this theorem, it's still going to be an order n squared algorithm. And so that really doesn't change the amount of time used from the perspective of-- from our perspective, where we're going to be ignoring the constant factors on the running time.

So I ask you here, can you do better than just improving things by a constant factor? There we go. OK, so here's a check-in on this problem, on the problem A, deciding A on a one tape Turing machine. Can we do better than order n squared, as I just described in that theorem in that algorithm for that theorem? Or can you get it down to n log n? Or maybe you can get it down to order n? What do you think?

Obviously, we're just getting started. But just make your best guess. And let me just post that for you as a poll. What's most important to me is that you understand the terminology that we're using and the way we're discussing-- the way we're talking about it. Because that's going to be setting us up for the definitions that we're going to come a little bit later in the lecture.

So I'm going to close the poll. We're kind of all over the place on it. But that's good. Since we haven't really covered this material yet. In fact, B is the correct answer. We can improve this algorithm down to order n log n, but not all the way down to order n.

So let me give-- let me show you how you do it in order n log n on the next slide. And so here is a one tape Turing machine that can decide A by using only order n log n steps instead of order n squared steps. So this is a significant improvement. So I'll describe the Turing machine. Here, again, is the picture of the machine on an input. And the very first thing I need to say is that we're going to scan to make sure the input is of the right form.

And now, we're going to-- again, it's going to be making repeated passes over the input. But we're going to do it a little differently now. Instead of crossing off a single a and a single b, or some fixed number of a's and a fixed number of b's, we're going to cross off every other a and every other b. And that way, we're going to essentially cut the number of a and b in half.

And that's why the number of iterations is only going to be a log, instead of linear. So we're going to cross off every other a and every other b. And at the same time, we're going to keep track of the parity, the even/odd parity of the number of a's that we've seen and the parity of the number of b's that we've seen that have not yet been crossed off.

And we're going to compare those parities to make sure that they agree. If they ever disagree, we know we started off with different numbers of a's and b's. And I'll illustrate this in a second, just to make sure we understand this algorithm. So I'm going to write down as a little table of the parities that we've seen. And I'm going to illustrate the algorithm with a little animation.

So again, we're now going to scan across, crossing off every other a, and then every other b. But before we get to the b's, we observe, as we cross these off, that we had six a's. Now, I'm not saying we count them. We just keep track of the even/odd parity. That can be done in the finite control of the machine. Counting them would be more complicated. But just keeping track of the parity is something that the finite automaton could do.

So the parity in this case, because they were six, is going to be even. Now, we cross off the b's. Same, even parity, now we're going to return the head back to the beginning, I'm obviously not showing the head moving here. We return the head back to the beginning. And now, we scan across, again, crossing off every other remaining a and counting the parities of the remaining a's. So here now, it's going to be this one and this one are going to get crossed off.

And there were three a's, so that was odd parity. And the same for the b's, three b's, odd parity. And now we return our head back to the beginning, cross again, off every other a and every other b. So that was odd parity, there was just one. Crossing off the b, odd parity because just one. They all agree. So the machine is going to accept. Because everything is now crossed off. And the parities agreed along the way.

Let me just say for a second, obviously, if you ever get disagreement on the parities, then that the number of a's and the number of b's had to disagree. But how do we know that if the parities always agree that we actually did start out with the same number of a's as b's? And that, you could see that in a number of different ways, but perhaps a cute way to see it is there is actually-- if you look at these parities , here the sequence of parities, actually in reverse. So if you say odd, odd, even, and you look at the binary representation of the number of a's, which is six, so the binary representation would be 110.

The fact that you get odd, odd, even and 110 is not a coincidence. In fact, the sequence of parities in reverse that you get is exactly the same as the binary representation. You'd have to confirm that with a little proof. It's not hard. But that, once you have confirmed that, you can see that if the sequence of parities agree, then the numbers have to be the same, because the binary representations agreed.

OK, so now getting to the analysis here, again, order n steps to do the check. Log n iterations. Each scan takes order n steps. So the total running time here is going to be order n log n. It's going to be the log n times n. That's where the n log n comes from.

Now, question you might ask, could I do even better than that? Can I beat n log n by more than any constant factor? So can I do little o of n log n? And the answer is no. This is the best possible one tape Turing machine for this language. So a one tape Turing machine cannot decide A by using little o of n log n steps. We're not going to prove that. I'm not going to ask you to be responsible for the proof.

But in fact, what you can show is that any language that you can do on a one tape Turing machine in the little o of n log n steps turns out to be regular. So you can prove a rather strong theme here. Not super difficult to prove. But I don't want to spend a lot of time on proving grounds for Turing machines. Because really the whole purpose of setting this up using Turing machines is to talk about algorithms in general, algorithms in general, I'm not going to be focusing on the nitty gritty of Turing machines.

OK so what is my-- yeah, so I wanted to just stop and make sure that we are together here. So a brief pause. And feel free to send me any questions. OK, this is a good question. If we can keep track of parities, why can't we just keep track of the number of a's or b's? Well, you could keep track of the parity in the finite memory. And so you can do that with effectively no work.

But the finite memory cannot is not enough for doing a count up to some arbitrarily large number. Now, you could store the count on the tape. But that's going to cost you time to maintain that counter. And so that's not going to be so simple as keeping track of the parity in the finite memory.

Somebody is asking if a star b star is regular. Yes, a star b star is regular. So what? But a to the k, b to the k, which is our language, is not regular. That's the language we're looking at. So getting questions about what happens with multiple tapes. We'll talk about that in a second. Yes, we could do an order n steps on a regular computer, sure. But for this slide, anyway, we're looking at one tape Turing machines.

Getting questions about big O and little o. For something to be little o means it's less than any constant factor times the function. So you have to look at the definition in the book. I think enough people in the class probably know this and have seen this already, I don't want to spend everybody's time on it. But please review it in the book. Somebody is asking if you need to store the parity on the tape. No, you can just store it in the finite memory. I mean, storing in the finite memory seems to me the simplest thing.

Why don't we move on? Please feel free to ask questions to the TAs as well. We have two TAs at least here attending with me. So good, all right, all right, so we cannot do better than a one tape Turing machine-- on a one tape Turing machine than order n log n. And that's something we can prove, though we're not going to do it here.

However, if you change the model, for example, you use a two tape Turing machine, then yes, as a lot of you are suggesting in the chat, you can do better than that. So if we now have a two tape Turing machine, or a multi-tape Turing machine, you can do it in order n steps.

And that's actually the point I'm really trying to make here. So if you have here your two tape Turing machine, then two tapes, same language. Now, what we're going to do is copy the a's to the second tape. That we can do on a single pass. And then once the a's have been copied to the second tape, we can continue on reading the b's and match them off with the a's that appear on the second tape. So in order n steps, you can do the comparison, instead of order n log n steps.

And of course, if they match, you're going to accept, otherwise reject. So let's just see, here's a little animation demonstrating that. Of course, it's very simple. So here is-- here are the-- if you could see that, it came maybe a little too fast. Here, let's just show it again. Here is the heads moving across and the a's coming down to the bottom.

And now, the head, the upper head is going to continue on reading the b's. The lower head is going to go back to the beginning on the a's and matching off the b's with the a's. And that's how we can verify or check that they are the same number. So now in this case, they were the same number. So the machine would accept. If they were a different number, the machine would not accept.

And the analysis is very simple. Each stage here is going to take a linear number of steps, order n steps, because it just consists of a single scan. There are no-- there are no loops in this machine, no repeat loops. So question on this? All right, why don't we move on then?

Now, observe-- and the point I'm really trying to make is that on a one tape Turing machine, you can do it in n log n, but not any better. But on a two tape Turing machine, you can do it in order n. So there's a difference in how much time you need to spend, how many steps you need to spend, depending upon the model. And that's significant for us.

So the number of steps depends on the model. One tape Turing machine was order n log n, multi-tape was order n. We call that model dependence. If you contrast that with the situation in the complexity-- in the computability section of the course, we had model independence. The choice of the model didn't matter. And that was nice for us.

Because the theory of the decidability didn't depend upon whether you had a one tape Turing machine, or a multi-tape Turing machine, it was all the same set of decidable and recognizable languages. So we didn't have to worry about which model we're actually going to work with. We could work with any model, even just an informal model of algorithm would be good enough. Because we're going to end up with the same notion in the end.

Now that goes away in complexity theory. Now, we have a difference, depending upon the model. And from a mathematical standpoint, that's a little less nice. Because which model do you work with? If you want to understand the complexity of some problem that you have at hand, now you have to make a choice.

You're going to work with a Turing machine, or how many tapes, or you're going to look at some other model, and you're going to get different results. So it's somewhat less natural from a mathematical standpoint just to talk about the complexity of some problem. But we're going to kind of bring back something close enough to model independence by observing that even though we don't have model independence, as we did in computability theory, we can limit how much dependence there is.

So the amount of dependence is going to be low, as we will see, provided you stick with a reasonable class of deterministic models. So the dependence, though it does exist, is not going to be that much. It's going to be polynomial dependence. And we'll say exactly what that means in a second. And from our standpoint, that's going to be a small difference, a negligible difference that we're going to ignore.

So we're going to focus on questions that do not depend on the model choice among these reasonable deterministic models. Now, you may say, well, that's not interesting from a practical standpoint, because polynomial differences, say the difference between n squared and n cubed certainly make a difference in practice. But it really depends on what kinds of questions you're focusing on.

So if you want to look at something that's a very precise distinction, say between n squared and n cubed, then you might want to focus in on which model you want to be working with. And that's going to be more the domain of an algorithms class. But from our standpoint, we're going to be looking at other, still important, questions. But they are questions that don't depend upon exactly which polynomial you're going to have. We're going to be looking more at distinctions between polynomial and exponential. And still, there are important practical questions that arise in that somewhat different setting.

So with that in mind, we're going to continue to use the one tape Turing machine as our basic model of complexity. Since the model among the reasonable deterministic models in the end is not going to matter from the perspective of the kinds of questions we're going to be asking. So with that, so we are going to continue, then, it's important to remember that from going forward, we're going to stick with the one tape Turing machine model.

Maybe that's something you would have expected us to do anyway. But I'm trying to justify that through this little discussion that we had so far, thus far. So now, we're going to start defining things with the one tape model in mind. So first of all, if you have a Turing machine, we're going to say it runs in a certain amount of time. So if t is some sort of time bound function, like n squared, or n log n, we'll say the machine runs in that amount of time, like n squared or n log n. if that machine M always halts within that number of steps on all inputs of length n.

So it always halts within t of n steps on inputs of length n. Then we'll say that the machine runs in t of n time. So in other words, if the machine runs in n squared time, then the machine, when you give it an input of length 10, it's got to be guaranteed to halt within 100 steps, 10 squared, 100 steps, on every input of length 10. That's what it means for the machine to be running in that much time. And it has to do that for every n, for every input length.

And width that, we're going to come to the following definition, which is highlighted in color, because it's going to be-- we're going to be using this definition throughout the semester. So it's important to understand it. This is the definition of what's called the time complexity classes. And what I'm going to do is take some bound, t of n, and again, think of t of n like a bound like n squared.

So if you have time t of n or like time n squared, that's going to be the collection of all languages that you can decide within time n squared or within time t of n. So in other words, it's a collection of all languages B such that there's some one tape Turing machine, here we're focusing again on the one tape Turing machine, there is some deterministic one tape Turing machine that decides B. And that machine runs in that amount of time.

So this is a collection of languages. The time complexity class is a set of languages. I'm going to draw it now as a diagram. So if you take the language, again, that we've been using as our standard example, a to the k, b to the k, that's n time n log n, as we observed two slides to slides back, or three slides back.

So on a one tape Turing machine, you can do this language A in time n log n. So it's in the time complexity class n log n. This region here captures all of the languages that you can do in order n log n time. For example, that also includes all of the regular languages. Why is that?

Well, any regular language can be done on a one tape Turing machine in time order n, because the Turing machine only just needs to scan across. Doesn't even need to write, just need to scan across from left to right on the tape. And in n steps, it has the answer. So all of the regular languages are actually in time n, certainly a subset of time n log n.

And these all form a kind of a hierarchy. So if you increase the bound, you can imagine that the class of languages grows as you allow the machine to have more and more steps to do its computing. So these are all the languages that you can do in n squared, order n squared time on a one tape Turing machine, n cubed time on a one tape Turing machine, and so on, 2 exponential time, 2 to the n time on a one tape Turing machine. These are all collections of languages getting larger and larger as we increase the bound.

So someone is asking kind of-- let's see, let me get to some of these questions here. I'll try to get to them in order. So somebody says if you have-- a good question, if you have a regular computer, so an ordinary sort of random access computer, which we'll talk about that in a second, can do it in order n, can you do it on a multi-tape Turing machine also in order n time? Actually, I don't know the answer to that offhand. I suspect the answer is no.

That ordinary computers have a random access capability that Turing machines do not. And so that there are going to be some examples of problems that you can do with a random-- with a regular computer that you cannot do with the multi-tape Turing machine in order n time. I'd have to double check that, though, so we can-- it's also a question what we believe is true and what we can prove to be true. As we'll see, there are a lot of things that we believe to be true in this subject that we don't know how to prove.

Somebody is asking kind of an interesting sort of more advanced question. Is there some function f, some function t where it's so big that so that time t of n gives you all of the decidable problems? It would be a very big t. But the answer actually to that question is yes. But that's a little bit exotic. So let's not spend a lot of time on that right here. But happy to talk about that offline.

It's a good question here. Somebody's asking me does it mean that there are no languages between order n and order n log n, because I pointed out that anything below n log n is going to be regular. And so, as soon as you get below n log n, you can do it in order n. And yes, there is what's called a gap between order n and order n log n on a one tape Turing machine.

You don't get anything new from order n until you jump up. So from order n to order n log log n, nothing new shows up. So we'll talk about those kinds of things a little bit down the road, when we look at actually the relationship among these various classes, and what we call a hierarchy theorem, which shows-- how much bigger do you have to make the bound in order to be sure you'll get something new?

All right, somebody's asking is there a model which has the same time complexity as a normal computer? Well, I mean, there's the random access model, which is supposed to capture a normal computer. So let me-- these are all great questions, kind of more riffing off of this into more advanced directions. Let's move on.

Here's another check-in. Suppose we take-- this is a little bit of a check to see how well-- how comfortable you are with the notions we've just presented and whether you can think about some of the arguments that we've made and apply them to a new language. So take the language ww reverse, strings followed by their-- followed by themselves backwards. This language B are the even length palindromes, if you will.

What's the smallest bound that you need to be able to solve that language B? And I'll pose it as a-- pose that as a question for you. So which time complexity class is that language B in? Is it time order n, order n log n, n squared, so on? What do you think?

So we're about to come to the coffee break. So why don't we-- I'll answer any questions that come up. I think we're got everybody answered. So I'm going to end the polling. OK, make sure you're in if you want to be in. So the correct answer is, in fact, order n squared. It would be hard-- reasonable guess here would be order n log n.

I mean, you can come up with the same procedure as the one we showed at the beginning, the order n squared procedure for a to the k, b to the k works for ww reverse as well. You can just cross off, sweep back and forth, crossing off a symbol from w, and going across to the other side, crossing off a symbol from w reverse. And that procedure will give you an n squared and order n squared algorithm.

You might imagine you can improve it to order n log n. But you cannot. You can prove that order n squared is the best possible. I'm a little unhappy that a lot of you came up with order n, frankly. Because I already told you that order n is-- these are just regular languages. Anything that you can do in less than-- a little o of n log n is going to be regular. And we know this language is not regular.

So this was not a good answer. So please pay attention. And OK, so let us stop sharing. I will turn now to our break for five minutes. And I'm happy to try to take questions along the way as we're waiting for the time to end. So let's see, let me put this up here.

Let me try to take some of your questions. So someone is asking me about quantum computers as reasonable models of-- you may say a quantum computer is a reasonable model of computation. And that's fine. I would not say it's a reasonable model of deterministic computation, at least from our standpoint. Let's not quibble about the words. I'm not including quantum computers in the collection of machines that I have in mind right now when I'm talking about the reasonable models of deterministic computation that we're going to be discussing.

Let's see. Oh, because a bunch of people apparently are asking the TAs why all regular languages can be done in order n. So if you think about a DFA, which processes an input of length n with n steps, and a DFA is I'm going to be a type of Turing machine that never writes on its tape, so if a DFA can do it in n steps, the Turing machine can do it in n steps. And so therefore, every regular language can be done in order n steps on a Turing machine. Not sure where the confusion is. So please message me if you're still not getting it.

OK, somebody saying why are we using one tape Turing machines instead of random access? Wouldn't it be better to use the random access machines? If you were using-- if you're trying to do algorithms, yes. That's a more reasonable model. We're trying to prove things about the computation. And from that standpoint, we want to use as simple a model as possible.

Trying to prove things using random access computers is possible. It'd be very messy. So that's why we don't use random access machines to prove the kinds of things we're going to be proving about computation that are really the meat and potatoes of this course. So I mean, there's compelling reasons why you would want to use a simple model like a Turing machine, but not a powerful model like a random access computer.

So somebody's asking me, does the class time order n log log n have any elements? Yes, it has all the regular languages, but nothing else. Order n log log is it's only the regular languages. You have to go all the way up to n log n before you get something non-regular. Someone's asking me are we going to talk about how the random access model works? No. That's beyond the scope of this course, outside of what we're going to be doing.

We're going to talk about Turing machines. Not because we care so much about Turing machines. But I'm trying to prove things about computation. And the Turing machines are a convenient vehicle for doing that. Our candle has burned out.

Why don't we return, then, to the next slide. So everybody come back. So this answers one of the questions I got on the chat. What actually is the dependency between multi-tape Turing machines and one tape Turing machines? Can we bound that in general? Yes, we can.

We're going to show that converting a multi-tape Turing machine to a one tape Turing machine can, at most, blow up the amount of time that's necessary by squaring. No, I acknowledge it's a lot. But it still allows you-- but it's still small compared with an exponential increase. And we're going to be focusing, in this course, on things like the difference between polynomial and exponential, not between the different-- not between the difference of-- not the difference between n squared and n cubed. That's going to be less of a factor, less of an issue for us.

So the way I'm showing this theorem is that if you have a multi-tape Turing machine that can do a language in a certain amount of time, then it's in the time complexity class of that time bound squared. And the way I'm just saying that is because this is the bound that's utilizing the one tape model. So another way of saying that is converting multi-tape to one tape squares the amount of time you need at most.

So the way we're going to prove that is simply by going back and remembering the conversion that we already presented from multi-tape to one tape. And observe that if we analyze that conversion, it just ends up squaring the amount of time that the multi-tape used. So why is that?

So if you remember, let's just make sure we're all together on this, the way the single tape machine S simulates the multi tape Turing machine M is that it takes the contents of each of M's tapes, up to the place where there's infinitely many blanks. Obviously you don't store the infinite part.

But the active portion of each of M's tapes, you're going to store them consecutively in separate blocks on S's tape, on S's only tape. And now every time M makes one move, S has to scan its entire tape to see what's under each of the heads and to do all the updating. So to simulate one step of M's computation, S is going to use order of t of n steps, where t of n is the total running time that M is going to use. So why is t of m steps coming up here?

Well, that's because you have to measure how-- S is going to make a scan across its tape. How big can its tape be? Well M, if it's trying to use as much tape as possible, can use, at most, t of n tape on each of-- t of n cells on each of its tapes. So altogether, they're just going to be some constant number of times t of n cells on S's tape. Do you see that?

So each one of these is going to be, at most, t of n long. So this all together is going to be order t of n long. Because what can M do? It could send its head out, say the head on this tape here, moving as fast as possible to the right, using as much tape as it can. But you can only use t of n cells in t of n time. So this is going to be order t of n. So one step of computation is going to be t of n steps on S's computation. But M itself has t of n steps. So it's going to be t of n times t of n for the total number of steps that S is going to end up using. And that's where the squaring comes from.

Similar results, I'm not going to do lots of simulations of one model by another. I think that you'll get the idea. And you can, if you're interested, you can study those on your own. But you can convert multidimensional Turing machines to one tape Turing machines, one tape ordinary linear-- one tape, one dimensional machines. And the bottom line is that among all of the reasonable models, they're all what are called polynomially related if each can simulate the other with, at most, a polynomial overhead.

So if one of the machines can use this t of n time, the other machine that's simulating it would use t to the k of n time for some k. That's what it means for the two machines to be polynomially related. And all reasonable deterministic models are polynomially related. So as we've already seen, one tape and multi-tape Turing machines are polynomially related, because converting multi-tape to one tape blows you up by, at most, squaring. So k equals 2 in this case.

Multidimensional Turing machines, again, polynomially related, the random access machine, which I'm not going to define, but it's the machine that you might imagine-- you would, I'm sure they must define in some form in the algorithms classes, polynomially related. Cellular automata, which are just arrays of finite automata that can communicate with each other, similarly. All the reasonable deterministic models, again, classical models, I'm not talking about quantum computing, are polynomially related.

So we are-- that kind of justifies our choice in picking one of them, as long as we're going to ask questions which don't depend upon the polynomial. Let's then talk about the class P. So the class P, this is an important definition for us. This is the collection of all languages that you can do in time n to the k for some k on a one tape Turing machine.

Or as I've written it over here, I don't know if this notation is unfamiliar to you, but this is like just a big sum. But here, it's a big union symbol. It's union over all values of k of the time class n to the k. So this is time n, union time n squared, union time n cubed, union time n to the 4th, and so on. We call these the polynomial time decidable languages.

So we're going to be spending a certain amount of effort exploring this class P and other similar classes. Somebody's asking me why is it a union. I'm not sure how else you would write it. So if somebody-- if you have a proposal for a different way to write it, that's fine. But this is k, this is for all k. I don't know-- if you only had a limited finite number of k's, you could just take the biggest one. But since it's for all k, you need to write it as a union.

Now, I want to argue that the class P is an important class. And why has it had so much impact on the subject and in terms of applications as well? So one thing is that the class P is invariant for all reasonable deterministic models. What do I mean by that? So we have defined the class P in terms of these time classes here, which, in turn, are defined in terms of the one tape model.

So we have defined P by using one tape Turing machines. Now if we had defined P in terms of multi-tape Turing machines, we get exactly the same class, because one tape and multi-tape Turing machines are polynomially related to one another. And since we're taking the union over all polynomials, that polynomial difference is going to wash out.

Similarly, we could define P using any of the other reasonable deterministic models. And we get exactly the same class. So in a sense, we get back what we-- the situation that we had in computability theory, when the class of decidable languages didn't depend on the choice of model.

Here, the class P does not matter depending upon the choice of reasonable deterministic model. And we also kind of, even in the case of computability theory, we have to stick with kind of reasonable models that cannot do an infinite amount of work in one step. I'm not going to define what it means to be reasonable. That's, in a sense, an informal notion. But among all of those reasonable models, you're going to get the same class P.

The other thing that makes P important is that P roughly corresponds to the problems that you can solve in some reasonable practical sense. Now, not exactly, problems that require n to the hundredth time, you could argue cannot be solved in any reasonable sense. But if you think about it, for example, from the perspective of cryptography, cryptographic codes that people come up with are typically designed to require, or the hope is that they would require an exponential amount of effort to crack.

If someone found even an n to the hundredth algorithm to crack, that would crack a code, people would feel that the code is not secure, even though n to the hundredth is still large. So it's a rough kind of test. But it's still used as a kind of litmus test for practical solvability if you can solve it in polynomial time. You basically figured out how to avoid large searches if you can solve problems in polynomial time. We'll say more about that later.

But what I want to bring out here is that we have combined, here in the class P, something that's mathematically nice, mathematically elegant with something that's practically relevant. And when you have a combination of the two, then you know you have a winner. Then you know how you have a concept that's going to make a difference. And that's been true for the class P. This has been very influential within and without the theory of computation.

So let's look at an example. Let's define a new language we haven't seen before, though it's similar to procedures that we've looked at before. The PATH language, which is where I'm going to give you a graph G, two nodes in the graph, s and t, where I'm thinking of G as a directed graph. So directed means that the connections between the nodes in G are going to be directed. And that they have arrows on them. They're not just lines, but they have an orientation with an arrow.

So G is a directed graph that has a path from s to t that respects the directions. So such a-- I think I might even have a picture here, yeah. So imagine here, here is your graph. If you can see it, there are little arrows connecting the nodes. And I want to know is there a path from the node s to the node t. So that is a picture of a problem, an instance of a graph of a PATH problem.

And I want to find an algorithm for that. And I can show that there is an algorithm that operates in polynomial time for this PATH problem. And the algorithm, any of the standard searching algorithms would work here. But let's just, for completeness sake, include the breadth-first search algorithm that we have explored previously when we talk about finite automata.

So we'll mark s. And they'll keep repeating until nothing new is marked. And we'll mark all of the nodes that were reachable by a single arrow from a previously marked node. And then CFT is marked, After you have marked everything you can get to. So you're going to mark-- let's see, pictorially, here I think I have this indicated.

Yeah you're going to mark all of the things that are reachable from the start-- from the node s. And then see, after you can't mark anything new, whether the node G is marked. And if it is, you'll accept. If it is not, you reject. Now, we can analyze this, too. And I'm not going to be spending a lot of time analyzing algorithms here. But let's just do it kind of this one time.

We're doing a bunch of iterations here. So we're going to be repeating until nothing new is marked. So each time we mark something new, we can only do that, at most, n times. At which point, we've marked everything. So the number of iterations here is going to be, at most, n. And now for each time we mark something, we have to look at all of the previously marked nodes and see which things they point at to mark them too.

So this is going to be an inner loop, which again, has, at most, n iterations, because it's going through all of the previously marked nodes. And then once we have that, we can scan G to see-- to mark all of the new-- all of the nodes which we have not yet marked, whether they're connected with a previously marked node by an edge. And I'm being generous here, because I don't really care.

This can be done in, at most, n squared steps on a one tape Turing machine. I'm not going to describe the implementation. But I'll leave it to you as an exercise. But this is straightforward. So the total number of steps here would be n iterations times n iterations times n squared. So you're going to be, at most, n to the 4th steps needed. So this is a polynomial algorithm.

And whether I ended up with to the 4th, or n to the 5th, or n cubed, I don't really care. Because I'm just trying to illustrate that the total is polynomial. And that's all I'm going to be typically asking you to do. So to show polynomial time, what I'll be asking you to do is to show that each stage, each stage of this algorithm, should be clearly polynomial. And that the total number of stages, I'm sorry, this should say stages here, should be polynomial.

So each stage is polynomial. And after you're doing all the iterations, the total number of stages that are executed is polynomial. And so therefore, all together, the total running time, the total number of steps, is going to be polynomial. So that's the way we would write up polynomial algorithms in this class.

So let's see if there's any questions here. I don't want to get too far ahead of people. Let's see. Yes, in this theorem, I'm talking about one tape Turing machines, because we're defining everything in terms of one tape Turing machines. But now, at this point, when we're talking about polynomial time, my analysis is based on one tape Turing machines.

But in general, you could use any reasonable deterministic model on which to carry out your analysis, because they're all polynomially equivalent. So from the perspective of coming up with showing that a problem is in polynomial time, is in P, you can use any of the models that you wish that for convenient. Oh, that's a good question. What is n, thank you for asking that question.

n is always going to be reserved to indicate the length of the input. So here, n is going to be when we encode G, s, and t. And here, also, I haven't said this, but I'm assuming that the encoding that you're using is also somehow reasonable. I think we'll talk a little bit more about that in the next lecture, which is going to be after the midterm.

But you can cause problems if you intentionally tried to come up with nasty encodings, which will represent things with unnecessarily many characters. But if you try to be reasonable, then just use any one of those encodings, and you'll be fine. So yeah, so n is the length of the representation of the input. Let's see.

Someone's trying to dig into the actual how this is running here. Scan G to mark all y where xy is an edge. I'm saying that you can do it in n squared steps. If you have x, you can mark it in a certain place on the tape. And then as you're going to every other node, every other edge, you can go back and compare x with the x of the edge and then see-- and then find the y.

I mean, it's just going to be too messy to talk about here. I mean, I'll leave it as an exercise to you. I'm not going to try to fumble my way through explaining why you can do this in n squared steps. But it's not hard. You guys are all really hardcore algorithms folks. You want to know the algorithms for this, I'm not going to do that, sorry. High level picture here.

If you want-- if you want to look at detailed analyses, this is not the right course for you. Can k equal n? What is k? No, if you're talking about this k here, k cannot equal n. We're not looking at n to the n. These are all fixed k, it's like n squared, n cubed, but not n to the n is it going to be an exponential bound. And so that's not going to be included within this union.

So we're near the end of the hour. I'm going to introduce one last language here, called HAMPATH. And the HAMPATH problem is-- I'm going to ask now, again, for a path from s to t, but now a different kind of path, one that goes through every node of G along the way. So I'm looking for a path that goes, that hits every node of G, not just the shortest, most direct path. But in a sense, the most indirect path, the longest path that goes through from s to t that visits everything else along the way.

A path of that kind, that hits every node of the graph is called a Hamiltonian path. Because the mathematician Hamilton studied those and made some definitions about that. I'm not going to-- I don't actually know the history there. But I just know they're called Hamiltonian paths. So here's a picture. I want to get from s to t. But I want to sort of pick up everything else along the way.

So as you remember, the PATH problem itself can be solved in P. And what I'd like to know, can this simple modification, where I'm asking you to visit everything else along the way, is that problem also in P? And I'm going to pose this as a check-in for you. But actually, before I get to that, let me give you an algorithm for HAMPATH that doesn't work to show it's in P, because it's exponential.

So here's an algorithm for HAMPATH, where let's m the number of nodes in G. And what I'm going to do is I'm going to try every possible path in G and see if it's actually works as a Hamiltonian path, and accept if it is. And then if all paths fail, then I'll reject.

So I'm going to try every possible routing through G. If you want to think about it, think of m as every possible permutation of the nodes of G. And then you're going to see whether that's actually constitutes a path in G that takes you from s to t and goes through all of the nodes.

So this algorithm would work. This would give you a correct algorithm for the HAMPATH problem. The problem is, the difficulty is that there are so many possible paths that it's going to take you an exponential number of steps to execute this algorithm. It's not a polynomial time algorithm, because there are many possible paths that you could go through. If you're looking at it with a very crude bound, but you really can't improve that significantly, there would be m factorial, which is going to be much greater than 2 to the m paths of length m. So the algorithm is going to run for exponential time, and not polynomial time.

So my question for you is, I'm going to pose it as a check-in problem, is whether you could actually do this problem in polynomial time. So why don't you think about that as I'm setting up the question? So take the HAMPATH problem, just like the PATH problem, which I described with that marking algorithm, but now you want to hit every node along the way, can you show that problem as solvable in P?

And there's a whole range of possibilities here where either the answer is yes, you can see what the polynomial time algorithm is to definitely no, where you can prove there is no such polynomial time algorithm. And I'll put this is the check-in for you and see-- I'm curious to see what you come up with. Most people are getting it wrong. Well, wrong, I don't you know-- I'm not clear what wrong is here.

OK, are we done? Please check something. I see a few of you have not answered. But the poll is running out. OK, time is up. So in fact, as I think many of you know, but not all of you, this is an unsolved problem. This is a very famous unsolved problem, which is equivalent to the P versus NP problem that we're going to be talking about very soon, which, among other things, would be worth a million dollars if you solve it.

So for those of you who have answered a or e, please talk to me after lecture. And maybe we can work on it together. No, so yeah, I think most people would believe that the answer is no. But no one knows how to prove it at this time. So I'm interested in the folks who have come up with what they think are solutions.

And I should say that there are some folks who believe that there might be other outcomes besides just a simple no. Which might be proven eventually. So we're going to talk more about this. But this is the answer to the question, just for your-- just to make sure you understand is that it's an unsolved problem right now. So we don't know. Definitely yes and definitely no, at least according to the state of knowledge of which I'm aware, are not correct answers. But any of the others, well, who knows?

So I think that's the end of what I had to say for today. We covered complexity theory as an introduction, looked at different possible models, focused on the one tape model, introduced, based on the one tape model, these complexity classes, the class P. And we showed an example of this PATH problem being in P. Talked also about this HAMPATH problem, which we'll talk about more after the midterm. OK, so I'll stick around for a few minutes if you have any further questions. Otherwise, so let me just take questions here.

Somebody is asking me about my personal opinion on P versus NP. My personal opinion on P versus NP is that P is not equal to NP and that we will prove it someday. When I was a graduate student back in the mid '70s, I thought it would be solved by now. And in fact, I made a bet with Len Adleman, who I subsequently ended up becoming the A of the RSA code, that we would solve it by the year 2000.

And I bet what was then a lot of money for me, which was an ounce of gold, which I didn't end up-- which I did end up paying off to Len in the year 2000. So I'm not making any more bets. But I still believe that it will be solved. Hopefully I'll get a chance to see the solution. I spend a lot of time thinking about it myself. Obviously, I haven't solved it, otherwise we would know. But hopefully somebody will.

I'm gettings asked a question here that's kind of an interesting question, but I don't really know I'm sure I understand it. What's the largest possible runtime of a decidable problem? What is the largest decidable runtime? So anything that I can describe can be-- there are going to be algorithms that run for longer. You can define an algorithm. You can define a runtime, which would, in a sense, beats all other runtimes.

So that any runtime is going to be dominated by that extremely slow runtime. But it's not something that one can describe. I can describe it to you by mathematical procedure. But it's not going to be something like 2 to the 2 to the n. Somebody's here proposing a solution to the HAMPATH problem by presuming polynomial time. Why is the following flawed?

If s goes through all nodes and ends up at t, Well, s we're not-- you mean, I presume you mean starting at s, if we end up going through all nodes and end at t, the proposal is a little complicated here. Basically, if I can try to rephrase it, you want to try to-- from every possible node, you want to try to calculate a path to t and also a path from s to that node. And you can do that for all possible nodes. But there's no way to really combine them into a single path that visits all.

Solving for each node separately is not going to do the trick, because you have to somehow combine all that information into a single path, just one path that goes from s to t and visits all the other nodes along the way. And that that is not-- I don't see what your proposal, how that's going to actually work.

AUDIENCE: Professor Sipser. A question on the-- we were kind of talking about earlier, what we talked about today was defined for the one tape Turing machines, correct?

MICHAEL SIPSER: Yep

AUDIENCE: So and you said we could apply for the multi-tape ones, but are-- I don't know if we talked about earlier, if something is accepted by the one tape machine, can it be applied to the multi-tape Turing machine and vice versa? Are they interchangeable like that?

MICHAEL SIPSER: Well they're interchangeable only in the sense that the amount of time that you would need-- it's a different machine. If you have a multi-tape Turing machine for some language, you can convert it to a one tape Turing machine using the procedure that we described earlier in the term. And you'll get a different machine. It's going to run for a different amount of time by using that procedure.

The point is that the amount of time that the one tape Turing machine is going to run for is not that much worse than the multi-tape Turing machine's time. So if the multi-tape Turing machine's time was n cubed, the one tape Turing machine's is going to be the square of that. So it's going to be n to the 6th. But it's not going to be-- going from multi-tape to one tape is not going to convert you from polynomial to exponential. That's the only point I'm trying to make. It's going to convert from one polynomial to a somewhat bigger polynomial. But it's still going to leave you polynomial. I don't-- you don't seem-- I can't see your face.

AUDIENCE: No, I guess my opinion is just, especially when we were talking about earlier, when you were bringing it up, it just seemed like you could just turn anything to a multi tape Turing machine and completely cut the time out. If I had like something in n log n, if I did it in the multi-tape Turing machine, I have it in big O of n, you know what I mean? It just seemed like the multi-tape was so much more powerful. But then I guess not, with the explanations and the models we were talking about today.

MICHAEL SIPSER: Yeah, I would not say-- the multi-tape Turing machines are still pretty limited in their capabilities. And don't forget, when you have a multi-tape Turing machine, you have only a fixed number of tapes. I mean, you can also define variations of multi-tape Turing machines that have an increasing number of tapes as the input, either under program control, it can launch new tapes. Or it could just have more tapes depending upon the size of the input, that would also be a possibility.

That's not the model that we have defined. But you could define a model like that. But as long as the total amount of work being done by the machine at any step s going to be a polynomial amount of work, then you can convert it to a one tape Turing machine with only a polynomial increase in the bound.

You want to be careful of machines-- one thing I meant to say but didn't say, so here would be an unreasonable model, which you might think of as a plausible model. But it's not going to be a reasonable model from our standpoint. And that would be a model, for example, that can do full precision, say, integer arithmetic with a unit cost per operation. So each operation counts as costs 1. But I'm going to allow you to do, for example, addition and multiplication.

The thing that's bad about that, in terms of being unreasonable, is that after k step, each time you do a step, you could double the size of the integer by squaring it. After k steps, you can have an integer, which is 2 to the k long. And now, doing operations there is going to involve an exponential amount of work, even in any reasonable sense. In a theoretical sense, and also in a practical sense.

A model that operates like that is not going to be able to convert to a one tape Turing machine with only a polynomial increase, because it's doing an exponential amount of work potentially within a polynomial number of steps. So that's within a linear number of steps, within n steps. So that's an example of an unreasonable deterministic model.

AUDIENCE: Yeah, thank you.

MICHAEL SIPSER: Sure.

AUDIENCE: So I'm just curious, some idea just occurred to me. I guess if you have an Oracle Turing machine, basically just so that you could look at a Turing machine description and decide whether it's a decider or not, then it'd be a little bit interesting to think about what happens if you look at all-- if you have a description of a pair of the Turing machine and an input string, then you can look at for all size n descriptions of a pair, what's the most steps that it takes for such a machine to convert it. So then you'd have combined Turing machine and input string descriptions, where you could look at what's the longest it takes to convert. I don't know. It's just a random thought that occurred.

MICHAEL SIPSER: Yeah, so we actually-- we're going to-- we will talk about Oracle Turing machines later on in the term. These are machines that have access to sort of free information. And that actually turns out to be-- there's some interesting things you can say about what happens when you're providing a machine with, in a sense, information for free. That you might otherwise want to charge it for actually computing that information.

But let's just say we're going to allow it to get that information without being charged. And then how does that affect the complexity of other problems, for example? And so we will talk about that later. But too much of, I think, of a digression at this moment to try to define all that. But happy to chat with you about it on Piazza if you want to raise a question there.

AUDIENCE: Thank you.

MICHAEL SIPSER: Sure, no problem. Somebody's asking me about strategies for solving the P versus NP problem. We will talk also talk about that a little later in the term as well. But clearly, it seems beyond the reach of our present techniques to be able to prove that some problems really take a long time. Like that Hamiltonian path problem, it seems like there's nothing really you can do with that significantly better than trying all possibilities as I described in that exponential algorithm on the last slide.

But how do you prove that? Nobody knows. So lots of people have tried, including yours truly. I mean, I've spent a lot of time thinking about it. Haven't succeeded with it. But there is somebody, I think, someday, somebody will come up with the new idea that's needed to solve it. But it's going to clearly take some sort of a breakthrough, some sort of a new idea. It's not just going to be a combination of existing ideas, existing methods.

I think we're about 10 minutes past. A few of you are still here. I'm going to say goodbye to you folks. And shortly, I'm going to join my TAs for our weekly TA meeting. So see you guys. Thanks for being here.