[SQUEAKING]

[RUSTLING]

[CLICKING]

MICHAEL SIPSER: OK, everyone. Let's get started. Welcome back to Theory of Computation lecture number six. So we have been looking at a number of different models of computation. And last lecture actually was an important one for us, because we shifted gears from our restricted models, finite automata, pushdown automata, and their associated generative models, regular expressions, and context-free grammars to Turing machines, which are going to be the model that we're going to-- which is the model that we're going to be sticking with for the rest of the semester, because that's going to be our model, as we're going to argue today, for a general purpose computer.

So in a sense, everything we've been doing up till now or up until that point has been kind of a warm up. But it's been nevertheless has gotten a chance for us to introduce some important concepts that are used in practice and also will serve as good examples for us moving forward. And also just to get kind of in a sense on the same page.

So what we're going to be doing today is looking at the Turing machine model in a little bit more depth. One can define Turing machines in all sorts of different ways. But it's going to turn out not to matter, because all of those different ways are going to be equivalent to one another. And so we are going to stick with the very simplest version, the one we've already defined, namely the simple one tape Turing machine. But we're going to basically justify that by looking at some of these other models and proving equivalence.

So we will look at multi-tape Turing machines, we'll look at non-deterministic Turing machines, and we'll look at another model which is slightly different but it's still based on Turing machines called an enumerator. And we'll show that those all in the end give you the same class of languages. And so in that sense, they're all equivalent to one another. And that's going to be-- serve as a kind of a motivator, kind of in a sense recapitulate some of the history of the subject, and going to lead to our discussion of what's called the Church Turing thesis. So we will get to that in due course. And we'll also talk about some notation for notations for Turing machines and for encoding objects to feed into Turing machines as input, but we'll get to that shortly as well.

So let's move on, then. We will next go into a little review of what we did with Turing machines. And just want to make sure we're all together on that. It's a very important concept for us this term. So the Turing machine model looks like there's going to be this finite control. There's a tape with a head that can read and write on the tape, can move in both directions. The tape is infinite to one side and so on, as I mentioned last time.

The output of the Turing machine is either going to be a halt, accepting or rejecting, or loop that the machine may run forever. With three possible outcomes for any particular input. The machine may accept that input by entering q accept. May halt and reject by entering q reject. And it might reject by looping, which means it just never gets to the accept state or it never gets any halting state. It just goes forever. But we still consider that to be rejecting the input, just it's rejecting by looping.

And as we defined last time, a language is Turing recognizable, or as we typically write, T recognizable, if it's the language of some Turing machine, the collection of accepted strings from that Turing machine. Again, just as before, the machine may accept 0 strings, one string, many strings, but it always has one language, the collection of all accepted strings. Now, if you have a decider, which is a machine that never loops, which always halts on every input, then we say its language is a decidable language. So we'll say a language is Turing decidable or simply decidable if it's the language of some decider, which is a Turing machine that always halts on all inputs. So we have those two distinct notions, Turing recognizable languages and Turning decidable languages.

Now, as we're going to argue this lecture, Turing machines are going to be our model of a general purpose computer. So that's the way we're going to think of computers as Turing machines. And why Turing machines? Why didn't we pick something else? Well, the fact is, it doesn't matter. And that's going to be the point of this lecture. All reasonable models of general purpose computation, unrestricted computation in the sense of not limited memory, are all going to be-- all have been shown, all the models that we've ever encountered have all been shown to be equivalent to one another.

And so you're free to pick any one you like. And so for this course, we're going to pick Turing machines because they're very simple to argue about mathematically, and also they have some element of familiarity in that they feel like-- well, they're more familiar than some of the other models that have been proposed that are out there, such as rewriting systems, lambda calculus, and so on. Turing machines feel like a primitive kind of computer. And in that sense, they have a certain familiarity.

OK, so let's start talking about variations on the Turing machine model. And we're going to argue that it doesn't make any difference. And then this is going to be kind of a precursor to a discussion of a bit of the history of the subject that we're gonna get to in the second half of the lecture.

So a multi-tape Turing machine is a Turing machine, as you might imagine, that has more than one-- well, has one or possibly more tapes. And so a single tape Turing machine would be a special version of a multi-tape Turing machine. That's OK. But you might have more than one tape, as I've shown in this diagram.

Now, how do we actually use a multi-tape Turing machine? Well, you present the-- and we're going to see these coming up for convenience. Sometimes it's nice to be working with multiple tapes. So we're going to see these later on in the semester a couple of times as well. But for now, we're setting the model up so that the input is going to be presented on a special input tape. So that's where the input appears, and it's going to be followed by blanks, just as we had before. And now we have these potentially other tapes, possibly other tapes, which we call them work tapes where the machine can write other stuff as it wishes. And those tapes are going to be initially blank. So just all blanks on them. All of the tapes are going to be read and write. So you can write on the input tape. Obviously you can read off on the input tape, and you can read and write on the other tapes as well.

So what we want to first establish that by having these additional tapes, you don't get additional power for the machine in the sense that you're not going to have-- you won't have additional languages that you can recognize by virtue of having these additional tapes. I mean, you can imagine that having more tapes will allow you to do more things.

For example, if you have a pushdown automaton with two stacks, you can do more languages than you can with one stack. So it's conceivable that by having more tapes, you can do more languages than you could with one tape. But in fact, that's not the case. One tape is as good as having many tapes. And we're going to prove that. We're going to quickly sketch through the proof of that fact.

So the theorem is that a language is Turing recognizable. And when we say Turing recognizable, for now we mean just with one tape. So that's the way we've defined it. So a language is Turing recognizable if and only if some multi-tape Turing machine recognizes that language. So really another way of saying that is if you have a language that you can do with a single tape, you can do it with a multi-tape and vice versa.

So one direction of that is immediate, because a single tape Turing machine is already a multi-tape Turing machine that just so happens to have one tape. So the forward direction of that is-- there's nothing to say. That's just immediately true. But if we want to prove the reverse, then we're going to have to do some work. And the work is going to be showing how do you convert a multi-tape Turing machine to a single tape Turing machine. So if we have something that's recognized by a multi-tape Turing machine, it's still Turing recognizable. And what that means is that you can do it with a single tape Turing machine. So we have to show how to do the conversion. And I'll show you that I think in a convincing way but without getting into too much detail.

So here is an image of a multi-tape Turing machine during the course of its input. So we've already started it off. Initially it starts off with the other tapes, the work tapes being all blank. But now it's processed for a while and the head on the input tape now has moved off from its starting position at the left end. It's somewhere in the middle. It's written stuff on the other tapes.

And what we want to do is show how to represent that same information on a single tape Turing machine in a way which allows the single tape Turing machine to carry out the steps of the multi-tape Turing machine but using only a single tape by virtue of some kind of a data structure which allows for the simulation to go forward. So how is the single tape Turing machine going to be simulating, going to be carrying out the same effect of having these multiple tapes on the multi-tape Turing machine?

So here's a picture of the single tape Turing machine. It just has one tape. By the way, I should mention that all of the tapes are infinite to the right in the multi-tape Turing machine, just as we had for the single tape Turing machine. And now with this single tape Turing machine, it's going to represent the information that's present on these multiple tapes but using only the single tape. And the way I'm choosing to do that is going to be particularly simple. I'm going to just divide up, so here I'm saying it in words. It's going to simulate m by storing the contents of the multi-tape on the single tape in separate blocks.

So I'm basically going to divide up the single tape Turing machines tape into separate regions where each one of those regions is going to have the information that was on one of the tapes in the multi-tape machine. So for example, on the first tape it's got a, a, b, a. Well, that's going to appear here in that first block on the single tape Turing machine. Sort of seeing it float down to suggest that it's coming from this multi-tape Turing machine.

But that's really been developed by the simulation that the single tape Turing machine has. Obviously the single tape Turing machine doesn't have any direct access to the multi-tape machine. But it's going to be simulating. So this is how we're showing the data is being stored on the single tape machines tape. So in the second block, it's going to have the contents of the second tape of the Turing multi-tape machine.

And the final region of the single tapes, the final block so-called of the single tape's tape is going to have-- the single tape machine's tape-- it's going to have the rest of the contents of the last tape of m. And then it's going to be followed by the same infinitely many blanks that each of these tapes are going to have following the portion that they may have written during the course of their computation.

So that's what the single tape Turing machine's tape is going to look like during the course of its computation. It's going to have the information represented in these blocks. Capturing all of the information that the multi-tape Turing machine has in a perhaps somewhat less convenient way, because the multi-tape Turing machine, we didn't really make this explicit. And in part because it doesn't really matter, we didn't say exactly how the multi tape machine operates.

What I have in mind is that the multi-tape Turing machine can read and move all of its heads from all of its heads in one step. So in a single step of the multi-tape machine, it can obtain the information that's underneath each of its heads, feed that in through its transition function, and then together with the state of the multi-tape machine decide how to change each of those locations and then how to move each one of those heads. So it's kind of operating on all of the tapes in parallel.

So now how does the actual steps of the simulation of the single tape-- by the single tape machine, how does it go? So first of all, besides storing the contents of each of the tapes in S's single tape, there's some additional information that it needs to record. Namely, M has a head for each one of its tapes. But S has just a single head. And each of M's heads could be in some different location. In this case, as I've shown it, on the first tape, its head is in location three. On the second tape, it's in location two. On the third tape, it's on location-- on the last tape, it's in location one.

So where were we? We were simulating the multi-tape Turing machine with the single tape Turing machine. And we had to keep track of where the heads are. And so we're going to do that by writing the locations on those blocks. So we're going to have a special dot, as I've shown here, to represent the location of the head in that very first block. So the head's on the b. I'm going to dot the b. And I'm going to do the same thing for the locations of the other heads.

And how am I getting that effect? Well, we've seen something like that before, where we just expand the tape alphabet of S to allow for these dotted symbols as well as the irregular symbols. We also have expanded it to include the delimiter markers that separate the blocks from one another, which I'm writing as a pound sign. So we can just get that effect simply by expanding the tape alphabet of S.

A few more details of S that are just worth looking at, just to make sure we're kind of all understanding what's happening. So for every time M takes one step, S has actually a lot of work to do. Because one step of M can read and move all of those heads all at one shot. S has to scan the entire tape to see what's underneath those in effect virtual heads, which are the locations of the dotted symbols. It has to see what's underneath each one of those heads to see how to update its state to the next state. And then it has to scan again to change the location, change the contents of those tape locations, and also to move the head left or right by effectively now moving the dot left or right. So that's pretty straightforward.

There's one complication that can occur, however, which is that what happens if on one of the tapes, let's say, M starts writing more content than you have room in that block to store that information? So if M, it has 1, 0, 1, suppose M after a few steps moves its head into the originally blank portion of the tape and writes another symbol there. We have to put that symbol somewhere, because we have to record all that information. And the block is full.

So what do we do? Well, in that case, S goes to a little interrupt routine and moves, shifts all of the symbols to the right one place to open up a location here where it can write a new symbol for continuing the simulation. So with that in mind, S can maintain the current contents of each of of M's tapes. And let me just note that here. Shift to add room is needed. And that's all that happens during the course of the simulation. And of course, if S as it's simulating M observes that M comes to an accept or reject state, S should do the same. So that's our description of how the single tape simulation of the multi-tape machine works.

Let's turn to non-deterministic machines. Now, if you remember for finite, and I hope you do, for finite automata, we had equivalence between non-deterministic and deterministic. Finite automata. For pushdown automata, we did not have the equivalence. We didn't prove that, but we just stated it as a fact. There are certain languages that you can do with non-deterministic pushdown automata, which is the model we typically hold as our standard. So that some things you can do with non-deterministic automata that you cannot do with deterministic pushdown automata. You really need the non-determinism. They're not equivalent.

For Turing machines, we're going to get the equivalence back. So we'll show that anything you can do with a non-deterministic Turing machine in terms of language recognition, you can also do with a deterministic Turing machine. So remember, we kind of mentioned this briefly last time, the non-deterministic Turing machine looks exactly like a deterministic Turing machine except for the transition function where we have this power set, which allows for multiple different outcomes at a given step instead of just a single outcome that you would have in a deterministic machine. So we represent that with this power set here applied to the possibilities that the machine would have as the next step.

So now we're going to prove a very similar theorem, that A is Turing recognizable by an ordinary one tape Turing machine. That kind of goes without saying, since that's how we defined it. It's Turing recognizable if and only of some non-deterministic Turing machine recognizes the language. Again, remember we're using non-determinism in the way we always do. Namely that a non-deterministic machine accepts if there is some branch or some thread of its computation ends up at an accept. Other branches might go forever or might reject part of the way through. But acceptance always overrules then if any branch accepts. That's how non-determinism works for us.

So now the forward direction of this is, again, just like before, immediate because a deterministic Turing machine is a special kind of non-deterministic Turing machine which never branches non-deterministically on any of its steps. So that forward direction is immediate. The backwards definition is where we're going to have to do some work converting our non-deterministic Turing machine to a deterministic Turing machine. And we'll do so as follows. We'll take a non-deterministic Turing machine. Here is our picture. And we're going to now imagine its computation in terms of a tree.

And so here is the non-deterministic computation tree for n on some input w. So n for non-deterministic. Here is the tree. Somewhere down here it might end up accepting. And as I mentioned, that is going to be defining the overall computation to be accepting if there is some place here that's accepting. And the only way it can be non-accepting computation if there is no accept that occurs anywhere. And somehow if you're going to convert this non-deterministic machine to a deterministic machine, the deterministic machine is going to have to get that same effect. It doesn't have the non-determinism. So how does it manage that?

Well, it's really going to be carrying out the simulation kind of as you would imagine doing it yourself if you were told you had to simulate a non-deterministic Turing machine, which is you're going to search that tree of possibilities looking to see if you find and accept. If you do, then you accept. If you don't, well, maybe you'll go forever or maybe you'll manage to search the entire tree, and then you will say, no, I accept, no, I reject.

So let's just see, again, what the data structure of that deterministic machine's tape is going to look like. So the way this simulation is going to carry out, and you could program this in a host of different ways. And in fact, the textbook has a somewhat different simulation. But I think this one here lends itself a little bit better and maybe it's a little simpler for description in this setting. But there's lots of different ways of improving these things.

So m is going to simulate n kind of a little bit similar to the previous simulation. It's going to break the tape up into blocks, and it's going to now store in each block a different thread at a particular point in time of n's computation. So you imagine where m is going to be simulating n sort of doing the same parallelism but getting the effect of the parallelism not through the non-determinism but by maintaining copies of each of the threads on its tape and then updating them accordingly. So the idea is, I think, not too complicated.

So let's just see some of the details here. So here populating those blocks with the contents of n's tape on different threads. Maybe that corresponds to n's tape after the fifth step of n on each of those threads. You have these three possibilities, let's say, written down as three different blocks.

Now, remember in the case of the multi-tape simulation, we needed to record some extra information. In particular, we had to record the location of the heads by using those dotted symbols. We're going to do that again because each one of these threads might have their head in some different location. But we're going to have to do more. So let's do one thing at a time. We're going to store the head location. Here they are all written down as dotted symbols.

But now there's something that goes further. Because in a multi-tape case, there was one global state that the multi-tape machine was in, of course, at any given time. It's just that one has one finite control. It's in one state. But in the case of the non-deterministic machine that we're simulating now, each thread can be in a different state.

So whereas before, we could keep track of the multi-tape machine's state inside the single tape machine's finite control. Now there could be many, many different states to keep track of. So you won't be able to store that in a finite control anymore, because there could be some huge number of threads active at any given stage. And some of them can be one state. Others can be in different state.

And how do you keep track of all that? So we're going to have to use the tape. We're going to actually write down which state a given thread is in on that block. And we'll do that. I'll show it like this. So writing down the state of each thread in the block by writing down symbols which correspond to those states. So we're actually in a sense writing the states down right on top of the block using symbols that correspond to those states.

So again, we're getting that effect by increasing the tape alphabet of M to include symbols that correspond to each of its states. And we're going to write those symbols down. So here's the symbol for state q8 that says, well, that very first thread of n's computation is in state q8. It's in this third position on its tape. The head is on the third position there. And the tape contents is a, a, b, a. That's how we represent that thread of the computation including the state.

So M is going to carry out that step by step maintaining that information on its tape. Again, similar to before, there might be some special situations arising. So for example, n is non-deterministic. So a thread might fork into two threads. What do we do then? Well, sort of the reasonable thing. If there's a forking thread, M then copies that block, makes two copies of the block to correspond, or however many copies you need, to correspond to the possibilities that that thread is branching into. So you want to represent them all on different blocks of M's tape. All right?

And then if M ever finds out that on one of the threads it's entering the accept state, it can just shut down the whole computation at that point and say accept. So perhaps a little elaborate, but I think conceptually, hopefully, not too bad.

Now, let's move on then to talk about somewhat of a different looking-- well, sort of a different model, in a way, which has some historical significance and sometimes is a helpful way to look at Turing recognizability. And incidentally, you have a homework problem on this model called an enumerator or Turing enumerator.

So here it's going to operate somewhat differently. First of all, we're going to enhance the Turing machine model. It's going to now have just a single tape. But it's going to also have a printer. So we're now adding in this new device into the Turing machine called a printer. And the Turing machine has tape as before, except we never provide any input to the machine. The tape always starts out fully blank. It's only used for reading and writing. Only used for work. It's not where you present the input.

So how do you talk about the language of the machine? Well, the way you work this machine is you take this enumerator. It's a deterministic Turing machine with a printer, as I mentioned. You started off in blank tape. And it runs and runs and runs and periodically it can print a string under some sort of program control, which I'm not going to spell out for you. But you can imagine you might have to define the machine in such a way that when it goes into a certain print state, we're not going to set that all up, then whatever string is in a certain region of the tape maybe from the start point up into the head location, that gets printed out on the printer. And then it can print out other strings in due course.

And so periodically, you think of this printer as printing out a string under the control of the Turing machine enumerator. And so these print strings, these strings that could print out w1, w2, they appear. And the machine might possibly go forever and print infinitely many strings. Or it might go forever and only print finitely many strings. Or it might stop after a while by entering a halting state. Accepting or rejecting is irrelevant, but it enters a halting state. And then the strings that it's print out, the finitely many strings it printed out, that's the output of the machine.

Now, the language of the machine is the collection of all strings that it ever prints out. So we're defining language in a somewhat different way here. It's not the strings that it accepts. It's the strings that it prints. So we think of this machine almost a little bit analogous to the regular expression or the grammar in the sense that it's a generative model. It produces the language rather than accepts the strings in the language. It's not really a recognizer. It's a generator. And the language of the machine, as writing over here, for an enumerator here, we say its language is a collection of strings that it prints.

Now we will show that a language is Turing recognizable in the previous sense recognized by some ordinary Turing machine if and only if it's the language of some enumerator. And actually this is kind of a little bit of an interesting proof. You have to do some work. It's an if only if. Now neither direction is immediate. You're going to have to do some work in both directions. One direction is a little harder than the other. We'll start off with the easier direction. Let's say we have an enumerator. Hopefully you got this concept of this Turing machine which periodically prints strings when you started it off on the empty tape.

So now what I'm going to construct for you is that Turing machine recognizer defined from the enumerator. So this recognizer is going to be simulating the enumerator. So basically that recognizer is going to launch the enumerator. It's going to start off simulating it. Now, if you want, if it's convenient, we could use several tapes. We can use one tape to simulate the recognizer and you have other tapes available for convenience if you want, because we already showed multi-tape and single tape are equivalent.

So if you want, you can think of M as having multiple tapes. It's not going to really be that relevant to my conversation. But you're going to simulate E starting on the blank input, as you're supposed to. And then you're going to see whenever E prints in x, you're going to see if x is the same as the input to the recognizer. You get the idea? So we have a recognizer. It has an input string, maybe like 1, 1, 0, 1. And I want to know. I want to accept that string if it's one of the strings that the enumerator E prints out, because that's what the recognizer do. It's supposed to accept all the strings that the enumerator prints out.

So I got this 1, 1, 0, 1. What do I do? I fire up that numerator, I get it going, and I start watching the strings it prints out, comparing them with my input string, 1, 1, 0, 1. If I ever see it print out a 1, 1, 0, 1, great. I know I can accept, because I know it's in the enumerator's language. But if I compare and compare and compare, I never see the enumerator ever printing out my input string 1, 1, 0, 1.

Well, I just keep simulating. If E halts, well then I can halt and reject if I've never seen that string coming out as an output. If E doesn't halt, well, I'm just not going to halt either. I'm going to go forever. But then I'm going to be rejecting my input by looping. So that's the proof of converting in this backward direction, which is the easier direction.

Now let's look at the forward direction, which has a certain wrinkle in it that we're going to have to get to. So now we're going to be building our enumerator to simulate our recognizer. And the way we'll do that is the enumerator has to print out all of the strings that the recognizer would ever accept. So you kind of do the obvious thing. You're going to take-- the enumerator is going to start simulating the recognizer M on all possible strings. Sort of one by one, doing them in parallel, taking turns.

Maybe we didn't make this so explicit. But you can sort of timeshare among several different possibilities. Sort of like having different blocks for the machine. The enumerator is going to run the recognizer on-- well, let's just say it does it sequentially. It runs it on the empty string. It runs it on the string 0. It runs it on the string one. Runs it on the string 0, 0. These are all the possible strings of sigma star. You run on all of them. And whenever the enumerator-- uh oh. This is wrong. So whenever M accepts, then print-- I can't write very well. Print w, wi. Oops.

OK. So I'm going to just say it in words. I'm want to simulate M on each wi. Whenever you notice M accepting wi, you just print out wi. Because you want to print out all of the strings that M accepts. Now, there is a problem here. So hopefully you're not confused by this typo. Doing it sequentially like I just described doesn't quite work. So let me just back that up here.

Doing it sequentially doesn't quite work, because M might get stuck looping on one of the wi's. Like maybe when I feed 0 into M, M goes forever. Because M is rejecting 0 by looping. But maybe it also accepts this next string in the list, the string one. I'll never get to one by just feeding the strings into M one by one like this. What I really have to do is run all of the strings in M in parallel. And the way I'm going to indicate that is I want to simulate M on w1 to wi for i steps for each possible i. So I'm going to run M on more and more strings for more and more steps. And every time I notice that M accepts something, I print it out.

So I will fix this in the version of the file that I publish on the website. So if you're still not getting it, you can look there. But just to make things even worse, I have a check in, which is going to be about this one little point here.

So I got a question. Where do we get the wi strings? The wi strings are simply the list of all strings in sigma star. So under program control, we can make M go through every possible string. Like if you had an odometer, you're going to first get to the empty string. Then you go to the string 0, then the string one. You can write a program which is going to do that.

And the Turing machine can similarly write code to do that, to get to each possible string one by one. And then that's what the enumerator is doing. It's obtaining each of those strings and then feeding them into M, seeing what M does. What I'm trying to get at here is that it's not good to feed them in, run M to completion on each one before going to the next one. You really kind of have to do them all in parallel to avoid the problem of getting stuck on one string that M is looping on. All right.

So this is relevant to your homework, in fact. If I convert em to an enumerator in this way, does that enumerator always print the strings out in string order. String order is this order here. Having the shorter strings coming up before the longer strings and within strings of a certain length, just doing them in lexicographical order. So hopefully you follow that. But since these are not correct, this doesn't matter for us. Let me launch that poll. I'll see how random the answer is here.

So a fair amount of confusion. If you're confused, obviously you're in good company there. About to shut this down. Five seconds to go. OK, ending it now. All right. So the correct answer, as the majority of you did get, is in fact that the order is not going to be respected here when E prints things out. It might print out some things later in the order before earlier things in the order. What's going to control when E is going to print it out?

If M accepts some strings more quickly in fewer steps, then E might end up printing out that string earlier in the list than some other string that might be a shorter string. Because the order in which E is going to now identify that M is accepting those strings depends upon the speed with which M is actually doing the accepting.

So this is relevant to one of your homeworks, because what you're going to have to show is that if you start with a decidable language instead of a recognizable language, then you can make an enumerator, which always prints out things in order, in the string order. And vice versa. And if you have an enumerator which prints that things out in string order, then the language is decidable. So need to think about that. One direction is a fairly simple modification of what I just showed. The other direction is a little bit of more work.

So anyway, why don't we take our little coffee break here? So that's an interesting question. So one question is because sigma has an infinite-- sigma star is infinite, if I understand this question correctly, which says it's an infinite number of strings here. How does the machine even get started because it has to go through and enumerate all of it? No, it's not first writing down all of those strings. Because yeah, you can't write down-- you can't do this infinite step of writing down all the strings.

What I had in mind is you're going to take each string in turn. First you're going to take the first string on the list of sigma star, run M on that string, and then the next string in sigma star, run it on that string, and so on, string after string. And so you're never going to have infinitely many strings to deal with. It's just going to be more and more strings as you're going along. And the question I was trying to get at is whether you're going to run each one to completion before you get to the next one or you're going to kind of try to do them all in parallel, which is what you need to do to prove this there.

Won't there be infinitely many wi? Yes, so there are a-- similar question. Infinitely many wi. There are infinitely many wi. But it's just an infinite loop that we're in. One by one, you're taking care of more and more wi as you're going. Let's see here.

Yes. So another question is are we running all the strings in parallel? Yes, we are running all of the strings in parallel. But it's running them in parallel, but these are-- we're not defining a parallel Turing machine. It's running them in parallel using time sharing. It runs a little bit in this block, runs a little bit in that block and another block, and sort of shifts around and sort of does a few steps of each. And that's how it's getting the effect of the parallelism.

Question is would the enumerator essentially have a print tape and a work tape? Yeah, you can think of it as having a print tape. However you want to formalize it. It doesn't matter. I mean, I'm being a little whimsical with attaching it to a picture of a real printer. But yeah, you think conceptually you can have a print tape. However you like to think about it is fine.

How can we directly say that without knowing the program of the printer? We don't have to get into the structure of the printer. The printer is just something where you say print a string and a string comes out. That's all we know about the printer, and that's all we need to know. So there's no program for the printer. Sorry if I'm not understanding your question. But the question literally says, how can we say that without knowing the name of the printer?

So if the machine is decidable, why does it have to print all strings in order? For example, if one string is shorter, can it be decided later? Well, yeah. The reason why we want-- so the question is, if the machine is decidable, why does it have to print all strings in order? Because that's what I'm asking you to do in the problem. If you read the problem, I think it's number five on the P set. That's what I'm asking you to do. That's why you have to print it in order. Otherwise you wouldn't have to worry about printing in order. It's just because I'm asking you to.

OK, so I think we're out of time. Let us move back into our material. Second half, Church-Turing thesis. So Church, this is going back into the bit of the history of the subject back to the 1930s. Back then people were interested in formulating the notion of what do we mean by algorithm. They didn't even call it algorithm. Some people call it procedure. Some people called it effective procedure. Some people called it effective calculation. But people had in their minds-- mathematicians have been dealing for centuries, thousands of years, with procedures for doing things. That's a very natural thing.

And mathematical logicians, in particular Church and Turing, Turing somebody surely obviously you've heard of, Church maybe not. Church was Turing's thesis advisor, in fact. And they both were coming out of the mathematical logic field of mathematics and trying to use mathematical logic to formalize this intuitive notion of what we have had for centuries about what a procedure is, what is an algorithm.

And back in those days, they came up with different ways of formalizing it. So here we had this notion of algorithm, which is kind of intuitive concept. Turing proposed Turing machine as a way of capturing that in a formal way, a mathematically precise way. Other people came up with other ways of doing it. And back then, it wasn't obvious that all of those different formulations would end up giving you equivalent concepts, equivalent notions. And in fact, they proved in fairly elaborate detail that the different methods that people came up with, there was the lambda calculus, there was rewriting systems, there were several methods that were proposed for formalizing this notion, and they all turned out to be equivalent to one another.

Today that seems kind of obvious, even though I went to some effort to prove that just to give you a feeling for how those things go. If you have programs, if you have Pascal and Java, say, and thinking about what you can do mathematically in those-- I'm not talking about their ability to interface with Windows and so on, but just the mathematical capabilities. The capability of doing mathematical calculations or functions with a Pascal program or a Java program. It would be absurd to think there's some program that you can write in Java that you can't write in Pascal or Python.

And the reason is we know you can compile Python into Java and you can compile Java back into Python. That tells you that the two systems, two programming languages are equivalent in power. That wasn't obvious from the get go to these folks. So they observed that all of the different efforts that came at formalizing algorithm, all were equivalent to one another. That was kind of a breakthrough moment when they realized that all of the ways that they've come up with, and once they got the idea, they realized all reasonable ways of doing it are always going to be equivalent.

And so that suggested that they've really captured this notion of algorithm by any one of those methods, say a Turing machine. And that's what they took. You can't prove that, because algorithms are an intuitive notion. But the fact that we're able to capture that in a formal way, that's what we call today the Church-Turing thesis. So any of these methods captured the notion of algorithm as we described. And that has had a big impact on mathematics. Just give me one second here.

All right. Here's a check in on this one. So you know Alan Turing. So here are some facts which may or may not be true about Alan Turing. So now you get to pick all of the ones that apply based on your knowledge. Obviously this is more for fun or historical interest. But let's launch that poll. See how much you know about Mr. Turing. Check all that apply. OK, almost done? Please let's wrap it up. I think that's everybody. OK, two seconds. Please get credit for doing this. End polling.

In fact, it's kind of interesting here. You all do know that he was a code breaker. He worked, in fact, was part of a team, I think led the team, which broke the German code during World War II. The Turing test is a famous thing for how do you characterize when you have an intelligent machine. So he definitely worked in AI.

He worked in biology as well. He has a paper less well known to computer scientists. But if you look him up on Wikipedia, where I get all my information, he actually, and I knew this anyway, he has a very famous paper, an influential paper, on how, for example, spots arise on leopards and stripes on tigers and so on. Gave a kind of mathematical model for that which actually proves-- actually is quite-- does capture things in an accurate way, as was shown subsequent to that.

Was imprisoned for being gay. In fact, as far as I know from his history, he was not in prison for being gay. He was convicted of being gay and was given a choice to go to prison or to take chemicals to cure him from being gay. And he opted not to go to prison and take the chemicals. And sadly, he committed suicide two years after that. So he was treated very badly by British society and British government despite having the great work that he had done.

And you might think that he has been honored by appearing on a British banknote. That's also not true. But the good news is he's not currently on a British banknote, but he's going to be on a British banknote starting next year. So that's along with Winston Churchill and a number of other notable Brits. He's going to be on the 50 pound banknote.

OK, so let us continue. So as I mentioned, the Church-Turing was important for mathematics. And it has to do with these Hilbert problems. I don't know how many of you have encountered that. David Hilbert was widely considered to be the perhaps the greatest mathematician of his day. And every four years, mathematicians get together for an international congress, well, the International Congress of Mathematicians. That's been going on for over 100 years. And he was invited to the 1900 meeting to give a talk about anything he wanted. And what he decided to do during that presentation is present 23 problems that would be a challenge for mathematicians for the coming century.

And we're running a little short on time, so I'm not going to go into them. Some of these we'll talk about later. But the 10th problem here is about algorithms. The 10th problem on Hilbert's list called Hilbert's 10th problem is a problem about algorithms. And it says give an algorithm for solving what are called Diophantine equations. He didn't call it an algorithm. He called it some finite procedure or something like that.

But what are Diophantine equations? I'm glad you asked. Diophantine equations. What are they? Well, they're very simple. They're just polynomial equations, like does this polynomial equal some constant, for example. But where you're looking for the solutions, the polynomials are variables. You're looking for the solutions, but you're only allowing the solutions to be integers.

So here's an example. So I give you this polynomial. Here I'm setting it equal to 7. And I want to solve that equation. So it has these three variables, x, y, and z. So you have to find a solution there. But I'm only going to allow you to have plug in integers. And in fact, there is a solution in integers. 1, 2, and minus 2. If you plug it in, you'll see it works.

All right. Now, the general problem that Hilbert was addressing is suppose I give you a polynomial. Let's say it's set to 0. So we're looking for roots of the polynomial. But just some polynomial equation. You can always put it in this form. And I want to know does it have a solution in integers? And what I'm looking for is a procedure which will tell me yes or no. I want to know. Give an algorithm to answer that question for a given polynomial.

Or using the language of this course, defining this in terms of a language, decide this language. Give a Turing machine which will decide yes or no for any given polynomial. Yes there is a solution in integers. No, there is no solution in integers. That was what Hilbert asked in his 10th problem. Give an algorithm. Now, as we know now, it took seven years to get the answer that there is no such algorithm. It's an undecidable problem. And we'll talk about that a little bit later in the term. But D is not a decidable language.

Now, there was no hope of coming up with that answer in 1900 or even in 1910, because we didn't have a formal idea of what an algorithm is. That had to wait until the Church-Turing thesis told us that algorithms really are Turing machines, that there is a formal way of saying what an algorithm is. And once you had that notion, then you could prove that there is no Turing machine. But before that, you had only this vague notion, intuitive notion of what an algorithm is. And so there was no hope of ever answering that, because in fact, the answer is there is no such algorithm.

Now, I'll give this as a little exercise to you. We're a little bit running short on time. But this language D is in fact a recognizable language. So I would suggest you think about that offline. But basically, you can try plugging in different values for these variables. And if you ever find that it evaluates to 0, then you can accept. But otherwise, you just have to keep going and looking. So showing recognizability is very simple, but decidability is false.

Now let's talk a little bit about encodings for Turing machines. So we're going to be working-- encodings and Turing machines. So we're now going to be working with Turing machines going forward. The input to those Turing machines might be polynomials, might be strings. They might be other things too. We might want to feed automata into the Turing machines. So the Turing machine can answer questions about the automata. Well, don't forget, Turing machines take as their input just strings.

So we have to think about how we're going to represent something more complicated like an automaton as a string. And I'm not going to get into the details of that. You could spell it out in detail. But I think that's not too interesting to get into that. We're just going to develop a notation that says if you have some object, it could be a polynomial. There could be an automaton. It could be a graph. It could be whatever you're working with. A table. I'm going to write that O in these brackets to mean an encoding of that object into a string. And then you can feed the string into the Turing machine.

So that's how we're going to be thinking of presenting Turing machines with more complicated objects and strings as input, because we're just going to represent them as strings using ways in which I'm sure you're all familiar. I mean, that's how you deal with representing stuff when you write your programs anyway. But just to make it formal and that's the way we're going to write it down in these brackets. And if you have a list of several objects that you want to present together to a Turing machine, we'll just write them together within brackets like this.

Now, for writing Turing machines down, going forward, we're going to be using high level English descriptions. We're not going to be worrying about managing-- like the stuff we've been doing up till now, I'm not going to ask you to do it. We're not going to do that anymore, and I'm not going to ask you to do it. Managing where stuff goes on the tapes and all that stuff, that's too low level.

Because really now that we have the Church-Turing thesis, we're really interested in talking about algorithms. We're not that interested in talking about Turing machines, per se. We're interested in algorithms and what the power of computation is. So we're going to only be talking about Turing machines now in a higher level way. And only when we need to prove something about capabilities, we're going to come back to Turing machines and we're going to be proving things about their limitations and so on.

So our notation for running Turing machines is going to be-- we're basically going to put the Turing machine inside these quotation marks. And we're going to know that we could in principle write out the Turing machine in a precise way in terms of states and transition function and so on, but we'll never actually go ahead and do that lengthy exercise.

So quick check in here. So one of the features-- well, OK, let me not give this away. So if x and y are strings, I want to now have a way of presenting two strings as a single string as input to my machine, because I always think of my machine as getting a single input. So would you suggest one way of combining two strings into one which would be a good encoding is just to concatenate those two strings together. Would that be the way you would do it? Is that a good way to do it or not such a good way to do it?

So let's see here. I can get to that next. And think of what you would want to have happen in a good encoding. Ready, set, sold. Share results. Yeah. I think most of you get the idea that this is not a good way of combining two strings into one, because the problem is it's kind of ambiguous in a sense. If you combine two strings this way, what's important in an encoding is that the machine when it gets the encoding, it can decode them back into the original objects.

And if you're just going to be sticking the two strings together, you don't know where one string ends and the next string begins. And so it's not going to be a good way of combining things. You should find a little bit more clever way of either introducing another symbol or doing something a little bit more sophisticated, which would allow you to be able to do the decoding as well as the encoding in a unique way.

So getting back to that notation for a Turing machine. So here is the machine we've already seen once before for a to the k, b to the k, c to the k. I would write it now more simply than managing all of the tapes that we had the first time around, which we did last lecture. I would just say we give it an input w, check if w's of the right form with the things in the right order. Reject if it's not. Count the number of a's b's and c's in w. Accept if all the counts are equal and reject if not.

That's going to be good enough to be writing things at that higher level when you're going to be writing your algorithm descriptions. You just have to make sure that whatever you're doing you can implement. You don't want to be doing tests which are impossible or doing infinitely much work in one stage. That's not good. But as long as it's clear that what you're doing is doing only a finite amount of work in every stage and it's something that you could really implement, you can write it in a high level.

I wanted to spend a few minutes talking about problem set two, but we're a little bit running out of time here. Particularly there was this problem number five where you show that a language is Turing recognizable if and only if there's a decidable D. Where C is Turing recognizable where D is now a collection of pairs. We've got some questions about the notation, which I've hopefully answered now. So C is a set of x's such that there exists a string you can pair it with. So that the string xy is in D.

Let me try to give you a picture of that. So I want to think of D as a collection of pairs of strings. And it might be helpful to think of D kind of on the axes here. So if we have a pair of strings xy, which I have not yet packaged into a single string, I'm thinking of them as a pair of two objects at this moment, think of D as-- so just the x part is just below here on the x-axis. D you might just conceptually want to think of it as a collection of pairs.

So it's like a subset of all of these pairs. And C is all of the x's that correspond to any pair in here. Sometimes we call that the projection, because it's all the things that are below or the shadow. If you had a light sitting up here, it's all the things that are kind of underneath a D. So I've written C here kind of a little thicker, if you can see that. So that's the C language. And there's two directions here.

One is a lot easier than the other. If I give you a decidable D and I want to test whether something's in C, so I'll give you an x, I want to know is x in C? Well, you now have to test is there some y that I can pair with x where the pair is in D? So you can start looking for that y. If you ever find one, you know that x is in C. There are infinitely many y's to look for. But don't forget, you're only looking for a recognizer for C. So on the rejecting side, you're allowed to go forever.

So I'm kind of giving you a way to think about the easy direction. The hard direction, you need to think about how are you going to come up with that decidable D. If I give you C, you have to find a decidable D. And now you're bringing something down lower. You're starting with a recognizable language and a decidable language, which is going to sort of be counterpart to that and in a sense is a simpler language. And the way it becomes simpler is that y is going to help you determine whether x is in C.

And I guess the thing that I'll leave you with is, and maybe we can talk about this a little bit more on Tuesday, I'll try to leave a little more time if you want to test if something is in C, x is in C, but I don't want to let you go forever anymore, because I want to be a decidable language. And I'm going to use y to help you. What information would help you guarantee you get the right answer for x being in C? But that you would have to be sure you're getting the right answer.

So y could just be the answer. But then you don't know that that's-- you have to be convinced that you have the right answer. y just saying what it is, I mean, y could say that x is in D even when it's not true. x is in C even when that's not true. So what information would allow you to check that x is in C? What would be helpful information to check that x is in C where you would avoid ever having to go forever? A little bit too rushed here for that to be helpful.

Anyway, let's just conclude what we've done. Just brief summary here. And I don't want to keep you over time. So I'll just leave this up on the board. I will see if there's any-- so the lecture is over and you can take off as you wish. I will stick around for a couple of minutes. I have another meeting soon, but I'll try to answer some chats.

All right. People are commenting about this movie about Turing,*The Imitation Game.* If you haven't seen that, I recommend it. And is it a good idea to make y to be equal to the repeated looped string? Hm. I'm not sure about that. Thank you, everybody, for sending your kind notes. Church-Turing thesis. The question is is it in the book? Yes. It's in the book. Kind of similar to what I've already said, but yes.

OK, this is a good question here. The Church-Turing thesis. Somebody asked me, is it proved in the book? There's nothing to prove. The Church-Turing thesis is an equivalence between the intuitive and the formal. You can't prove something like that. You can just make-- it's really in a sense a hypothesis that the only thing you'll ever be able to compute is something that you can do with a Turing machine. I mean, that has something to do with the nature of the physical world. And I don't really think that's what people had in mind. It's that the kinds of things that we normally think of being able to do with a procedure mathematically is exactly the kinds of things that you can do with a Turing machine.

So somebody's pointing out, trying to be helpful here, maybe these folks are asking about equivalents of various different computation models being proved in the book, not the Church-Turing thesis itself. Well, I mean, the things that we proved in lecture are also proved in the book. About the equivalence of single tape machines, multi-tape machines, non-deterministic machines, those are all proved in the book. I mean, there's lots of models out there. So we could spend a lot of time proving equivalences. And there are books that do spend a lot of time proving equivalences. But I think what we've given is probably enough.

That's a good question. If you give an algorithm, you don't need to give the actual machine going forward unless you're explicitly asked to. But yes, you don't have to give the states and transitions anymore.

OK. So it's a little after 4:00. I think I'm going to move on to my next meeting. So thank you for being here, and I will see you on Tuesday.