ProgrammingAbstractions-Lecture09

Instructor (Julie Zelenski):Hey. Good afternoon. Assignment two coming in today so hopefully you brought some paper copies and did some E-submits and are ready to immediately take on your next assignment. There's no rest for the weary in this class. My understanding is that Miron calls the version where we just chat about some of the assignment the pain pole. I think that's really just much too negative and I'm gonna call it the joy pole instead, and I'm interested in getting a little bit of feedback on how much joy you had in doing your last assignment. First, let's just do the quick counts of time spent.

How many people think they managed to get the whole thing done in less than 10 hours? All right. That's a few of you. That's very good. Very good. Ten to 15? Kind of more of my target range, right, and so that looks like a big healthy chunk in there. Fifteen to 20? Okay. A little bit smaller and then more than 20? Anybody willing to – maybe not. Here's another question, which is how many of you think – so there's two halves, right, the random writer, the maze generating and solving, how many of you think you spent more time or found it more difficult or more time consuming to get the first one done, the random writer than the maze? How many people thought the maze is where you spent your time? Yeah. And of the maze part, how many thought it was harder to generate the maze than to solve the maze? A little bit. How many thought it was harder to solve the maze than generate the maze? Okay. So it looks like solving the maze may be where it comes down to taking the biggest chunk of joy was spent. Okay. Good to know. Hopefully one of the experiences or one of the take home points right after the end of that was certainly that templates provide a lot of leverage. Having these container classes, you can get a lot of things done; write these sophisticated programs with a small amount of code.

I saw a lot of people write who had a comparison function that just did not reliably compare things, all right, for example did not order them. The set really needs to know ordering and who's first, who's last, who's in between because that's how it stores them to later look them up. If your comparison function is not reliable, if you pass A and B and it'll say oh, A is less than B and if you pass B and A, it'll say B is less than A, right, you have a total inconsistency in your system, which the set will then make a hash of, right, it'll throw things in places that it doesn't look later because it took your word at it when it said it should go to this side or that side and it won't find it when it goes looking for it this next time so you do really have to have an ordering function that's reliable that every single time you pass it, it's the same things.

No matter which order they are, they're gonna tell you the same truth about their relative positions. That's one of the lesson learned from that. Assignment 3 is going out today and assignment three is basically a problem set. There are six problems on it. Each of them requires one recursive function and in a couple cases, a little tiny bit of support code around it. Those recursive functions are short. Typically, 10 lines at most. Some of them, six or seven in some of the easier ones. But that code is hard one. Right. You won't find those lines easy to knock out the way you could write out four loop and be done with it, right; there really is some thinking and careful planning and doing the recursive thing and

getting your head around it. So it is a small amount of code, but a very dense and complicated bit of code to generate so hopefully you will start that one early and give it time to gel because sometimes I think you'll see the problem the first time and not see the recursive insight. A day or two later, it will start to make more sense to you, but if you try to compress that all into the night before it's due you may not have enough time to let everything come together well. It is really important to get recursion started now. This is gonna be the recursion problem set.

The next problem we will do is the big venerable boggle that has recursion at the heart of it, plus a lot of other things going on so if you don't spend the time this week getting your recursive footing right, it's gonna start to snowball because recursion shows up from here all the way out through the quarter so this is the time. When you're working on this homework, be sure to get the help you need to make sure the concepts start to come together for you because this is kind of a key to doing well in the rest of the quarter.

So today I'm going, I'm talking about [inaudible] recursion. This is kind of corresponding to reader chapter five. I will do a bunch of examples today that kind of get at this idea with different domains. And then we will go on to recursive backtracking which is the topic from chapter six for Friday. We won't cover the later sections of chapter six. We're mostly gonna focus on the first two sections. There is a later section about game playing and strategy and stuff. It's very interesting and it's worth reading, but we won't formally cover it and use that material in this course right now. Okay. So kind of remind ourselves about thinking recursively and where we're trying to work out way toward.

Recursive [inaudible] is really the hard part. Somebody gives you a problem and they say solve this recursively, where are you gonna find you spend your time is saying, well, given this problem how can I find that self similarity in it, how can I break it apart in such a way that a smaller form of the problem within, right, if I made that recursive call and then used it was gonna solve the whole problem for me. Maybe there's more than one solve problem, maybe it's a divide in half, something like that, but that part is usually where you'll spend a lot of your time is trying to figure out how it is you can structure it to divide it in a self similar way. Once you have that plan about how you're getting a little bit smaller, a little bit simpler, it's usually not so hard to follow that to the logical conclusion and say well, if I keep doing that enough times, what do I get to that will really allow me to terminate this, the simplest possible case that all of those, you know, minimizing of the problem might eventually lead to something so simple we can directly solve it that we're working toward. Sometimes there's more than one base case, but ideally, there's exactly one that everything comes to. It kind of cleans up your code if you can make it come out that way.

There's a couple common patterns and we've seen some of these already and we're gonna see them again and again, but partly what you're trying to develop is a set of examples you've seen before that help you to kind of use those patterns to explore further problems in the future and so a very common way of dividing stuff is you have some collection, whether it's a vector or a string or a set of paths or something like that that

part of the recursion often is that you somehow separate one from them so when we were trying to choose the people to go to the flicks, right, we picked some student at random and said okay, this student is either in or out and so we separated one from the mass, leaving us an N minus 1 and then that N minus 1, right, was that recursive piece about that substructure saying, well, if I had the answer for the remaining N minus 1, right, I could use the information about this 1 I've separated out to join it together. So sometimes, that's the first person, sometimes that's the last person. Sometimes it's almost a random person that you're picking, but the idea of separating one or some small number leaving the N minus that number to work on. Sometimes it's a bigger chunk, it's a division in half or in thirds or something where I'm dividing and conquering the binary search doing that saying. We've got some information from the middle that tells us about what these two halves are in the case of binary search, we only do recurring one.

We'll see some cases we're actually able to divide it in half and recur on both, but where we're still just dividing the problem in easier, in this case, half as big versions of the same problem, but then we can attack recursively. And sometimes there's more like a choice. This is maybe the thematic thing for some of the later things when [inaudible] is that there's some choice of the options that remain and the recursions that attack it to make it one of those choices, which then kind of leads you to a state that has fewer choices to be made, fewer decisions and allows you try decisions from there to work your way to some base case. One of the things I'm also gonna be talking about today is some of the implications of the relationship between when you process this part you've separated out and when you make the recursive call. But it's actually totally not irrelevant whether I make the recursive and then try to process the one, or whether I process it and then do the recursion. In some situations, they'll produce different results and different answers that are interesting to know about how those things play out. So today, the theme is procedure recursion and procedural is differentiated from functional in a very minute almost a bit semantic about the definition, but it's worth telling you.

A functional recursion problem is one where you have a function returning an answer like is [inaudible] true/false. The factorial is a raised power telling you what the answer is. In a procedural recursion, we have functions that don't return anything so there's not an answer being computed in result of that, but there is recursive in [inaudible] so there's a procedure here that calls itself eventually bottoming out at some base case and as a result of all that recursion being done, something happens. The one I'm gonna do at the beginning here is some drawing pictures, some graphics that draw fractal graphics where there's not an answer being computed, but there's something being illustrated on the graphics window. So I'm gonna look at two examples of this because I think sometimes for some students the numbers and the strings aren't the best way to get it; sometimes visually really helps to open up the enlightenment for what recursion means in terms of the structure. We're gonna look at some pictures that have a self similar structure that repeat within itself that you if you look at it at one level of detail you'll see certain elements that, actually, if you look a little closer, you'll see that same element, but just repeated on a smaller scale within that and then within that again kind of moving down to that base case.

The way these things that we've written in code will give us some outer fractal that you draw that is part of its handling then makes recursive calls to draw these inner, smaller versions of the fractal within itself. Okay. So here is a little bit of code. So I'm not gonna tell you what it goes, but we're gonna sort of just imagine thinking it through and looking at this to see what kind of things it could produce. Draw fractal, let's give it an X, Y in width and height, which I'll assume is the bounding rectangle of what's being drawn and then it makes a call to some helper called draw triangle. If I assume that does what it seems like it does it says oh, it draws a triangle, right, that's inscribed within that rectangle. Next up I'm seeing this base case that's, like, well, if that triangle was particularly small, in this case, two tenths of an inch or smaller in either dimension, then we don't do anything further with it. We leave it alone. Otherwise, we do a little bit of computation here, a little bit of math to compute some coordinates and they're labeled the left, top and right, which is basically it's making a call on the left sort of sub region of this rectangle that's at the lower left corner, but half as tall, half as wide. Here's one that starts in the middle and is also half as tall, half as wide that represents a chunk out of the top, and then one that's to the right.

This produced something you've seen before, but you generated it a very different way last time. Right. In assignment one, we had you do the chaos demonstration, it produced this drawing, which is due to Sirpinski from a different mechanism, but in fact, it is a very recursive drawing when you look at it. Right. There's this outer triangle and then within it, there is a left most, a top most and a right most reproduction kind of the outer triangle at a smaller level of scale. If I look in each of those triangles, would I see that same self-similarity? If I just keep going down at each level of scale, I see the same thing, that there's a triangle which itself has these three left, right, top regions that exhibit the same structure the outer one did, but just a little bit smaller. It eventually gets so small that it stops drawing and that's what terminates our recursion. So that is one of the aspects we'll be looking at. Okay. I did the work first. What happens if I turn it around so I have the same base case right and the same order of left, top, right here, but I wait to draw the big triangle until after I've drawn all the other interior fractals so doing the processing for this one after I've done those. I'm gonna get the same picture, but it's gonna grow in a different way that only I've done the whole thing is the outer triangle; in this case, drawn on top which isn't actually very noticeable because the inner piece is kind of already made up the outer boundaries anyway.

Student:

Can you go slower?

Instructor (Julie Zelenski):Yeah, I slowed this one down a little bit. It isn't any slower in real life. I was kind of screwing with the pausing because I wanted to get it just right and then one of the pauses I think the pause is slightly faster than the other. But growing right from that lower left corner up then working all the way through the top and then all the way through the right and then the outer most something only being built is an after effect of once the inners have been constructed, so the very first one appearing way down here. Now, let me change my three recursive calls. It currently goes left, top, right. I'm

gonna change the code to go top, left, right. I'll get the same picture, but where is the very first triangle drawn? Is it small or big? It's small and where does it live within the triangle? The very, very top, right? So the top most point…the top small little triangle, it goes there. What is the second triangle that's drawn? The one that's just down to the left of that one so I should see the top most one, it's left most neighbor and it's right most neighbor and then we'll start working our way down from there. One thing that is sometimes helpful to think about in terms of visualizing what a recursive procedure is doing is to think of it as a kind of an upside down tree where here's my draw fractal to the outer one and then it makes these calls, right to top to left to right.

Realize that the way that the recursion proceeds is that it makes the call to the draw fractal of top, which then makes a call to the draw fractal of its top, which makes the call to the draw fractal of this top so it goes deep, right, until it hits that base case and so this was where I went top, top, top, top, top, top, top, right to the very top most one. Well, after it bottoms out here, it comes back to this one that said, okay, well, I drew the little tiny top, now, I need to draw the left and the right and so the ones that are fleshed out at the bottom here are the left and right of the top most one. And after those, it kind of comes back.

So sometimes thinking about it in terms of a tree and realizing that it is not sort of level by level or [inaudible], it's not like it draws all the outer ones and then the inner side of that, it really goes all the way down to that base case before it start unwinding and backing up and revisiting the pieces that kind of deferred as part of the recursion to come back to. So the very first call to draw fractal is sitting on the stack frame while all these other ones are active and when they finish their work and unwind, it picks up where it left off in drawing the other two thirds of the fractal. So this one should grow down from the top from there and then it should start to go to the left also from the top working its way down and then once it's worked out the whole of the left, do the right also from the top working its way down. So same picture in all these cases, right, am I doing the big triangle first, the inner triangle first, the left, the top, the right, but they show you that kind of the processing and how it would evolve is different. In some cases, this is gonna really matter to us. It's gonna actually change the outcome of the recursion how we're doing this so it's good to have a little bit of an idea how to visualize the different tracing through the calls. Any questions?

Student: Is it possible to draw by joining in three different places at the same time?
Instructor

Well, I mean, [inaudible] control, right, and so at any given point we're doing one thing at a time, so really without more fancy things, the answer is basically no. Let me show you more little recursive fractal. And this is based on a famous Dutch painter Mondrian, right, who was alive in the first half of the 20th Century there, one of the Cubists, the group he belongs to, and you've probably seen some of these paintings. They might look familiar to you with these very bright primary colors, a lot of horizontal and vertical lines bisecting the canvas and then some of those squares being filled in with bright colors. This is sort of one of the more famous of the pieces that he did.

And his little philosophy I quoted from there, right, was this idea that this is about harmony and beauty and you can just through lines and colors produce things that really have an aesthetic component to them. I think it's a pretty neat thing. But I'm not an artist, by any means, so I think the solution to that is if I want to create beautiful things I need to write computer programs that do them for me because if it was up to me, I won't do it with my own hands. So what I'm gonna show you is a technique for thinking of this as a recursive problem. If you look at what a Mondrian is and if you're sitting down to start this, you have a big canvas in front of you. It starts blank and you say, well, where should I begin? Well, let me just pick a random direction, horizontal or vertical, and then I'll pick a random placement for it so I decide I'm gonna bisect vertically and I pick a spot somewhere in the middle of the thing and then I do a bisection of that with my big black line and now I look at the thing that I have left and I've got well two smaller canvases; one to the left of the line, one to the right, which themselves I could imagine using that same strategy for. Well, look at it and decide whether to make a horizontal or a vertical line and then, occasionally, along the way decide to throw in a little color so when I have a canvas maybe wash it in red or blue before I go through my division.

And sometimes, I decide not to divide it at all. So this canvas that I've sectioned off here, I might just decide to leave it there and say, okay, that's good enough. I don't want to go any further on that one. That's not what I wanted to do. So it has these three options, divide the canvas horizontally, divide it vertically, do nothing and then when I have those two smaller canvases, I recursively can apply the same kind of Mondrian style affect to it and when I get to something that's too small, I can stop trying to divide it further. So I'll show you what the code looks like here and then I'll show it running to you. Most of it actually is a little bit gunky just because there's a little math of adding and moving and stuff like that around, but it has this idea of oh, here's the rectangle you're trying to draw a Mondrian in. If it's too small, in this case, anything under an inch I decided was too small. Otherwise, I fill the rectangle in the random color, which often, is white, sometimes yellow, sometimes blue, and then based on three cases of a random choice, I either decide to just leave it as is filled with whatever color I chose or I make a line, vertically in this case or horizontally in that case, and then recursively section off the two portions I've made to recursively apply a drawn Mondrian to draw some more Mondrian in those pieces.

Does it work? Any by work, the code works, but does it actually produce art? You can be the judge. Now, the good thing is see it's easy to make new ones if you don't like that one. I don't like that one either. A little too much blue. That's getting somewhere. I might put that on my wall and then at this point, I just started making them just as many as you want and just keep going. Some of them it stops early, it has nothing to do. My mother-in-law is an artist and I feel like I have to apologize to her whenever I do this because I'm really not trying to make fun of art as though actually a computer can do anything. The truth is, when you do this actually what you realize is that it's not art, right, what's truly aesthetic is something that actually is not generated by randomness and recursion, but it's still interesting how [inaudible] have that pattern to them that are sort of Mondrianesque just by simply taking this recursive strategy and saying that's a little Mondrian, this is a little Mondrian and if you assemble them all together, you've built a big Mondrian.

Millions of dollars right here I'm generating. I tell you, when I retire from teaching, that's gonna be my next career, turning out fake Mondrian's. All right. So any questions about little pictures and graphics and things?

Student: On the previous one, could you go over what's passed in each of recursive or draw a Mondrian?

Instructor (Julie Zelenski):Yeah. So the only thing it takes is it's just a rectangle. It says, this rectangle you should fill with Mondrian like things, so the outer rectangle is the whole canvas and then once I've done a bisection, right, I've picked some point in the middle, I've divided the rectangle into two smaller rectangles, the left and the right. It's not necessarily exactly, but I have one that starts in the origin and goes to the midpoint, one that starts in the midpoint and goes to the right side and so it's just basically taking the outer rectangle and dividing it. If I drew a picture here, it might help. You've got this thing.

You choose to bisect this way and so what you're passing, is this rectangle and that rectangle, and when this one decides to bisect this way, then it'll pass this rectangle and that rectangle and so at each stage you'll have these smaller rectangles that were pieces of the outer rectangle that you're continuing to draw Mondrian into and eventually they get so small that you say I won't further divide that up anymore. I'll just stop right here. And so the line actually is drawn on the way down it turns out. If you'll notice, the draw black line happens before it draws the thing, so draw the line and then draw a background on top around it and stuff. Either way it doesn't actually really matter. You could draw it on the way out and you'd end up with the same picture. If just would appear differently.

Student: Explain why you haven't returned something, like, a break or some other leads?

Instructor (Julie Zelenski):Well, in general, I just need something that says don't go through the rest of this code. I'm not trying to get out of a loop or a switch statement, right, so if I really want to leave the function, return is the quickest way to do that. The alternative is I could've inverted the sense of this task and said, well, if it's at least an inch wide and an inch tall, then get into this code and effectively it would say if, and it didn't pass if you didn't put a drop in the bottom and falling off, but either way, right. So I think I probably tend to prefer to do it this way just because I like to get my base case up from the front of my eyes seeing what it does and, typically, the right thing for the base case to do is identify that you're at the base case, do whatever simple processing is required in that case and then return the answer or just return right away saying, I've handled the base case and now let the rest of the code be and now in the recursive case going on.

This is very much the style I tend to use. If we're at the base case, do the base case stuff and return. Now I'm gonna hit you with the most classic recursion example known to all cs students worldwide. I wouldn't be doing my duty if I didn't teach you a little bit about the legend of Brahmin and its towers and how relevant it is in today's world that you know how to move graduated discs around. All right. So I'll give you the set up.

Apparently, there are these monks and there's a legend of the tower Brahmin that these monks, apparently, big troublemakers, monks, you know how they are, you gotta keep them out of trouble so the dyad here had this idea. Well, I'm gonna give them this task that's gonna keep them busy for a long time and that way they won't be causing problems. So what he did was he stacked up these three spindles, A, B, and C, big poles and he had these big heavy discs, they were made of solid gold apparently in the Brahmin religion and they are graduated so the big disc at the bottom and a slightly smaller one on top and then so on, and they're all stacked up let's say on tower A. I've labeled them, A, B and C just to keep track of which is which, and the dyad says yeah, I got all these discs all stacked up on this in A, but it turns out I'd really much rather have them on tower B.

Sorry. What can you do? I accidently put them there and I'm so strong and big and heavy I could move them, you on the other hand, right, puny little human beings that you are, are restricted to being able to only move one disc at a time because they are so inordinately heavy that it takes everybody's strength to move just even the smallest of the discs from one place to another. And he further in states these rules that the discs all have to live on a spindle so you can't actually just pick them up and start throwing them around in the desert. You're trying to move this whole tower from A to B. You can move one at a time. And in addition, because they're so heavy, if you were ever to put a small disc on a spindle and put a larger disc on top it would immediately crush the one underneath it. So that's disallowed as well. So you always have to keep an ordering of large ones on the bottom to smaller ones on the top. All right. That's the deal. That's the set up. Those trouble-making monks. Apparently, there's 64 of these things. All right. In computer science, I don't have any solid gold heavy discs, but I do have some very brightly colored plastic discs. So this is not so much the towers of Brahmin or the towers of [inaudible], I changed locations somewhere along the point of history; this is the towers of Fischer Price.

I did not actually steal these from my children, but they have one at home, which I keep waiting to see them show evidence of their recursive nature and I must say very, very disappointing. Okay. So I've got a tower, in this case, of six. This is A, this is B, this is C and I want to get this from here to here and I don't want to violate all those rules. I can't just start throwing the discs around and making a mess. I have to move it from here to here. I don't have a lot of choices here, but once it's there, for example, this yellow guy can't go on top of it because it would squash it, right? So I could kind of move yellow in there, so there's something a little bit about this idea that somehow there's gonna have to be some back and forth that is a part of what's going on. So let's go back and look at this for a second. So as I said, many of your classic recursive patterns involve sort of looking at the problem you have at hand and trying to decide is there some way you could separate it a little bit and one of the more likely ways to do it is say is there some way I can move one disc out of the way leaving N minus 1 that could be worked on recursively? Okay.

So it's probably not likely that I'm gonna be able to get, let's say the blue disc out of the middle very easily, so it seems like probably the two most likely candidates for that are either the top most disc somehow gets moved out of the way to uncover the bottom part

or the bottomless one, somehow I get this other tower out of the way. So let's start by thinking about the top one because that's certainly the easiest. I can get this top guy, as I said, and stick it somewhere. Well, let's say I put it over here in the temporary one. Okay. I get it out of the way and now I have the tower of N minus 1 left. Okay. That seems like it's kind of getting me somewhere and now I need to move that tower, N minus 1, over here, but the situation I made is I've actually made the problem, not easier in the situation, it's a little bit smaller but it's actually further complicated by the fact that given this small disc is over here occupying the small spindle means that, suddenly, this spindle is out of commission. So the purpose of moving this spindle might as not exact, right, because this guy is blocking anybody from getting there. So it seems like I have a problem that looks what I started with, but it's not actually any easier, in fact, it actually got a little bit harder. So let me back up from that.

Let me just let that be a dead end for now. What about this purple disc that is down here on the bottom? I'd like to get to the purple disc. Well, I can't just pick up the whole tower and move it, or can I? Well, let's think. I've got this tower of N minus 1 on top of the purple tower. If I were able to produce a delegate, a clone, a co-worker, right, who I could say, Hey, by the way, you know how to move towers, could you please move the N minus 1 discs that are on top of the purple one over to the temporary spindle using this other one as the spare? Well, that's kind of interesting. So if I could move that N minus 1 out of my way and over to that temporary spindle, then it would uncover this bottomless one which then is one step away from where it wants to be; it wants to move to the middle. So this is really where recursions are kind of buying us a lot of the heavy lifting. I want to get this tower from here to there. There's all these rules. It's very complicated. I can't think about it, it makes my head hurt, but if I just believed that I was in the midst of solving this problem that I haven't yet solved and if I had that solution and I believed and I had faith, right, and I believe that it will work, I move these guys out of the way, I move this over here, I move it back.

So let's see how much I can do without blowing it. All done. Don't ask to see it again. I'll show you a little code. I'll show you the code. Tiny, tiny little piece of code that solves the whole problem. I've got a [inaudible] coming in here, which is the height of the tower to move and I'm identifying the source destination attempt, the three spindles that are currently in play here with these letters, characters; if there's something to move – so I'm gonna talk about the base case in a second, but in the general case where I've got five things to move it says move the four discs that are on the source. So the source, let's say is A in this case, and now instead of moving them to the destination, move them aside to the temporary spindle using the destination as the temporary. Once you've gotten those out of the way, you move that single disc, the bottom of the disc from the source of the destination, and then you go pick up N minus 1 tower you left on temporary and stack it on top of you in the destination now using the source as your temporary. At any given stage, you've got three to go and then you have to move the tower of height off of you.

When you've got two to go, you have to move the tower of height of 1 off you. Base case you could imagine being pretty easy while single height tower, just one disc, could be exactly moved, this actually prefers going to even a simpler case, right, which is if you're

asking to move a zero height tower there's nothing to do. In fact, the base case for the zero is if N equals zero, do nothing for any positive number then it goes through the process of shuffling the ones off of you to uncover your bottomless disc and then moving them back on. So five little lines and all the power in the world. Question?

Student: Why use the destination as a temporary?

Instructor (Julie Zelenski):The idea is that if we're moving here from A to B, right, I need to leave this one open, right, so that I'll be able to move the purple one when the time comes. So I'm trying to move this out of the way, right, so here's where it's trying to go and in the process, right, you have these two places you can kind of shuffle things back and forth, where they came from, where they're not going and where they are going, so there's also the source of destination and it's pretty obvious; and then what happens is that whatever is left over is the one that gets used as a temp.

Student: Why can't you put those on the second one?

Instructor (Julie Zelenski):Well, if I put them on the second one, how am I gonna get the purple one underneath them?

Student: Well, you wanted to put them on the last one.

Instructor (Julie Zelenski):Well, my plan was to move from A to B actually. It doesn't really matter.

Student: Oh, okay.

Instructor (Julie Zelenski):I'm trying to move from A to B so in fact, C is the place where I'm just leaving stuff temporarily.

Student: Okay.

Instructor (Julie Zelenski):So you're gonna see that the stack frames here show how deep the recursion ever gets at any given point, right, at this case, having four discs it gets about four deep, actually five because it goes to the zero case and that you'll see it kind of go down and back as it shows it kind of moving the tower away and then getting back to moving the bottomless disc and then doing another deep recursion to move that tower back on top. And so at any given stage, right, they're all stacked up and the start/finish attempts start switching positions depending on what level of the recursion we're at. Each time we get to zero we hit our base case and unwind. Let me make it go a little bit faster because it's a little bit slow.

Student:[Inaudible]

Instructor (Julie Zelenski):[Inaudible]. All right. So I just moved the tower of height three all the way away and now let's move the bottomless one back and now it's gonna

start moving the tower that was shuffled away back on to it and then it will eventually stack it back up on the B where it wants to go. I can make him move the tower.

Student: Sixty-four.

Instructor (Julie Zelenski):Yeah. Sixty-four. We'll talk a little bit about why I wouldn't want to put in the number 64. You can see I'm kind of it around. Dancing, dancing, much better than I would do. I actually maintained the rules at all times. No discs getting smashed. I go up to six and so never getting lost, always keeping track of what it's up to, right, it's a very tricky thing for a human to do, but a very easy thing for a computer to be able to identify what part of the recursion we're at and what we're trying to do right now without getting confused about the other things that are also kind of ongoing.

They're all very neatly kept apart. So when people will ask you about recursion, everyone will want to know have you been exposed to the towers problem, and now, you have. So very powerful little piece of code, right, and it feels very much like a trick. I can remember sitting in a room, not unlike yourselves, a gazillion years ago and having someone explain the towers of Hanoi to me and just not believing. Just saying that can't work. You didn't do anything. All you did was call yourself before you were even done and that's just nonsense and that's just not right. And then after some amount of thinking about it and working it through, I did conclude that it actually did work, but at first, it really did seem like a trick. So if you're feeling that way, this is par for the course. It is a little wacky to get your head around, but it is sort of relying on this idea of the mathematical induction; if you can do it for the smaller version of the problem and then build on that to solve the bigger problem, as long as you make that logical progression from this problem to the smaller one down to that simple case, you will eventually solve the whole thing.

Student:[Inaudible]

Instructor (Julie Zelenski):Source destination in this case. Okay. So then there are two problems I want to do. I'll probably only do one today and I'll do one on Friday that are what I think of as the mother problems of all recursions that many, many problems in the end just boil down to an instance of either the permutations problem or the subset problem. So I'm gonna show you permutations and subsets sort of in gory detail and then I'll try to build with you about how many things actually are just a permutations problem or subset problem when looked at the right way and so once you kind of have these two in your arsenal, you're very much prepared to attack a lot of different recursion things using those same patterns. So the one I'm looking at here is permutations. You have an input string, let's say it's the alphabet A through Z, 26 letters, what you'd like to do is enumerate or print or list all of the ways you could permute the alphabet. So there's 26 letters in the alphabet, if you know anything about combinations and permutations, you realize there's 26 ways you can choose the first letters, A, B, C, all the way to Z and then that leaves you with 25 letters from which to pick the next one, right, so there's 25 ways to do that and then 24 ways to pick after that and so at each step you have one fewer

choices remaining to conclude the permutation; and then the number of permutations is then factorials in the length of the input, so 26 factorial, which is an enormous number.

It tells you about all the different ways you could rearrange the alphabet. For a smaller string, you know, A, B, C, right, there are three factorials for that, six different ways you could permute that. The same principle applies no matter how large the string is. So our goal is to write a function that, given an input string, will print all of the permutations of that string. That's the goal. Okay. So I've got A, B, C, D coming in, I want to be able to print D, C, B, A and C, A B, D and all these other variations. I'm gonna use the strategy that actually I described just the way I did kind of intuitively about how permutations are constructed. That at any given point, right, I have a choice to make, and that choice in this case, will be what's the next letter to attach to the permutations. So if I measure my goal by trying to build the permutation up one letter at a time. I start with an empty string so I'm gonna have the input and the output. The input is the letters I haven't yet used, so in the case of the string, A, B, C, it might be the whole string, A, B, C. I've got this output, which is what letters I've chosen so far. It starts empty. I look at my A, B, C and I say, Well, each of those letters could be the next one that's here, why don't I go ahead and pick one, pick A for that matter, put it on the output and then recursively call myself to say well given A is in the front, because you also print all the things that you can permute the remaining letters, B, C, behind it.

After the magic recursion has done what it's gonna do I can come back to the call I was at and say okay, well, now it's not just A in the front I need to try; I also need to try B in the front. So I tried B in the front and now permute the A, C, that I leave behind. I try C in the front and permute the A, B, that's left behind. So at any given stage of this recursion, right, I have these options, which are of all the letters remaining in the input, each of them needs to be tried as the next one to go and then I need to recursively permute on this slightly smaller form of the same problem where the output is one shorter of. I've removed one letter from consideration because I've picked it and the input is one longer. So at each level of the recursion, one letter is shuffled from the output to the input after I've done that N times, right, I will have created a permutation and then I can just print it. So I've talked about this. Let me just show you the code because I think that is where we can spend the most time illuminating what's trying to go on. So here's recursive permute. The form that I'm gonna use here is gonna take two arguments. I'll explain the need for it in a second, but largely what I'm gonna be tracking is that input and output.

What I have assembled so far is in that first parameter, so the things I have committed in the permutation I'm building right now, the rest is those letters that haven't been chosen that have been passed over so far that are remaining to be permuted and attached on to so far. If the rest is empty, that means everything has been attached in so far, I have a permutation and then I just print it. If it's not empty, then for every letter that remains in the rest – so imagine I'm starting with the whole alphabet and so far was empty, this four loop is going to iterate 26 times, it's going to take out the [inaudible] character and it's gonna append it to the permutation and then it's gonna remove it from the rest to show that it's been used so we won't accidently repeat it later in the permutation.

And then we make this recursive call saying okay, now having picked the letter N and put it in the front, here's the whole alphabet without N. So at the beginning we have everything in the input, nothing in the output. Well, once we pick something that shuffles on character from here to there. A subsequent call down here moves another character and then eventually we've emptied out the rest. There are other alternatives though, for example, up here it could be that we picked B and left behind A, C, it could be that we picked C and left behind A, B. Before it's all done, we're gonna have to try all of those things and so the very first level of the recursion is gonna make a number of calls equal to the length of the input. Each subsequent level, right, in here makes one fewer call at each branch. So if I had a 26 at the top, each of those 26 then makes a 25 underneath so there's 26 25s branching down there and then each of the 25 makes a 24. So this is an enormously wide tree. The depth is bounded by the total number of characters, but very, very wide. I'm just gonna show you a little part of the tree to get an idea. If I'm permuting A, B, C, D in the so far for the input, [inaudible] starts empty and there are four calls that are being made, first with A, then with B, then with C, then with D and in each case, removing that letter from what remains and then exploring.

So if I kind of further expand this call, it makes three calls, right, one having pulling B to attach on, one having attached C on, and one having attached D on leaving the remaining two characters. And then underneath that, picking that third character and then eventually pocking the fourth character which there's no choices there. So this is only a partial part of the tree, but it gives you an idea of what the shape of that recursion looks like. This is a tricky thing to get your head around. But it is a critical pattern to get in your arsenals to you know how to apply it. So what this is sort of that choice pattern, right; of the choices I have, I need to go through the process of making those choices, updating my state to show that choice has been made and then recursing from there – making that recursive call to further explore whatever choices remain having made this one, which caused there to be one fewer choices to be made. And in a permutation, you have to make 26 choices, right, for the alphabet. Well, you make one and then you have 25 to go, you make one, you have 24 to go. Each of those calls, right, is working its way down.

There's one little detail I'll mention and then we'll finish on Friday, which is that probably the way this function would be presented to a client or a user who wanted to get the permutations is it makes more sense for them to say here's the string I have, could you please just list the permutations. This notion that somehow I need to track what I've generated so far is very much an internal housekeeping part and so it effects permutations to look like this and then just immediately turn around and make a call to the real recursive function that set up the state, the housekeeping that was being tracked through it, so we'll call this thing a wrapper function. All it is one line of code that just sets up the right state to get the recursion going and kind of state managed. We will see that quite a lot in a lot of codes. They just know about it. Come Friday, we'll talk about subsets and then we'll start looking at recursive backtracking.

[End of Audio]

Duration: 52 minutes