[SQUEAKING]

[RUSTLING]

[CLICKING]

ANA BELL: Let's get started, everybody. So last lecture, we began talking about this topic of recursion. And it hopefully solidified a few really fundamental ideas about recursion that we're going to use in today's lecture. Today's lecture, the first half of it ish we're going to talk about recursion just to review on some actual numerical examples. But then the second half, which is the main event for today, is going to be recursion on non-numerics, so specifically recursion on lists. But the techniques we'll see on lists can be applied to other things that are non-numerics as well, like tuples, or strings, or things like that.

So let's start the review a little bit of a review of what we talked about last time and some of the big ideas by looking at this example. So we're going to write a recursive function for the Fibonacci sequence. And the Fibonacci sequence exists in nature in a lot of places. One specific place is you can model mating of rabbits using Fibonacci sequence. But we won't be studying that in depth today. We're just going to be looking at the sequence itself.

So just to remind you, the idea behind Fibonacci is we start out with two sort of basic values, Fibonacci of 1 is 1 and Fibonacci of 2 is 1. So in my table here, I've got these two starting values. And we can fill in the remainder of the table by basically saying Fibonacci of n is Fibonacci of n minus 1 plus Fibonacci of n minus 2. So Fibonacci of 3 will be 1 plus 1. Fibonacci of 4 will be 2 plus 1. Fibonacci of 5 is 3 plus 2. Fibonacci of 6 is 5 plus 3. And Fibonacci of 7 is 8 plus 5. Right. That's the sequence we all know and love.

OK. So our two base cases, if we're going to put this in mathematical terms, are Fibonacci of 1 is 1. Fibonacci of 2 is 1. And our recursive step right in terms of the math and slash programming lingo is going to be the Fibonacci of n is equal to Fibonacci of n minus 1 plus Fibonacci of n minus 2. So we put that in our function. So we slap a definition around that code and turn it into a nice function that we can run.

If x is 1 or x is 2-- those are our two base cases-- we just return one right off the bat. Right. Nothing to call. No functions to call. They're our base cases. But otherwise, we're going to return a value. And the thing we're going to return is a call to Fibonacci of n minus 1 plus Fibonacci of n minus 2 just like the mathematical definition said to do.

OK. So this is different than what we saw last lecture. Last lecture in our recursive step, we had basically just one function call to ourselves. So whatever function we had defined up here, we only had return some variation of that function down here with something else tacked on to it, like an addition of some value or something else. In this case, we actually have the function being called twice. OK. So we're going to see what implications this has as we trace through the code. And so as I trace through the code, I'll remind you of some of the big ideas that we learned last lecture.

So let's say that we wanted to calculate Fibonacci of 6. And so I'm going to illustrate a function call just by the name of the function with the parameter that I'm calling. So one of the big ideas from last lecture was that when you make a function call to a function that's recursive, you're going to trace through that function call and the environment for that function just as you normally would.

But as soon as you see another function call, so in this case, Fibonacci of 6 doesn't enter the base cases. It goes up into the recursive step. And it says I'm going to calculate Fibonacci of x minus 1 plus Fibonacci of x minus 2. So for this Fibonacci of 6 function call, let's follow along and say, well, Fibonacci of 6, we'll say I want to calculate Fibonacci of 5.

This is my question to you. Is it going to now calculate Fibonacci of 4? No. Very good because Fibonacci of 5 is a function call. Right. We need to explore what this function will return before Fibonacci of 6 can add the result of this, the return of this, to Fibonacci of 4. So that means then this new Fib(5) is an entirely new environment calling Fibonacci with n is equal to 5, completely separate than our original Fibonacci of 6 call.

So let's explore what Fibonacci of 5 is going to do. Well, in its function call, it's going to again go in the recursive step. It's going to figure out Fibonacci of 4. And then it's going to pause there because it needs to figure out what Fibonacci of 4 is before it finishes its other half to do Fibonacci of 3. So Fibonacci of 4 will now create a new environment. And now it has to explore its return.

So it figures out Fibonacci of 4 is, again, going into the recursive step to calculate Fibonacci of 3 plus something. But we don't know what that something is yet because we have to explore what Fibonacci of 3 is. So already we're four function calls deep and we haven't really done any work that we can see the result of. There's no values being passed back. All we're doing is exploring this path down until we get to some sort of base case that will kick off our conquer step where we pass values back up the chain. So Fibonacci of 3, again, is going to look at Fibonacci of 2.

And finally, we've reached a base case. So Fibonacci of 2 will immediately return. It doesn't make another function call. So Fibonacci of 2 will return a value and then Fibonacci of 3, in its function call, has the result for Fibonacci of 2 and then it's going to do plus that value plus Fibonacci of 1, 3 minus 2. So that's this one here. It can easily do that addition and return the value back up to Fibonacci of 3. So now Fibonacci of 3 has its first half ready.

So Fibonacci of 4-- sorry. So Fibonacci of 4 has its first half ready, Fibonacci of 3. So Fibonacci of 4 was trying to figure out what Fib(3) was. And it did. It was Fib(2) plus Fib(1), 2. So now it has a value for its first half here. And it needs to add that value to Fibonacci of 2, 4 minus 2. So it will explore that path. That's a base case. So all it does is return the value immediately and now Fibonacci of 4 has its value, whatever Fib(3) was that we figured out, plus Fib(2).

Now Fib(4), we have a value for it when we called Fib(5). So Fib(5) is now halfway happy because it knows what Fib(4) is. But it needs to add that to Fib(3). So Fib(5) is still halted. It can't return anything because now it needs to explore what Fib(3) is. Well, Fib(3) is going to do another function call. Right? So it's going to call Fib(2) and Fib(1), which are two base cases which easily return the value back up to Fib(3).

And now Fib(5) is happy because it knows this value. And now it knows this value. It can add them together. And Fib(5) now has a value that it can keep track of. And now, finally, Fib(6)-- we're not even close to being done, you guys. Fib(6) has a Fib(5) value so it has half of the things it needs to figure out what Fib(6) is because now it has to figure out what Fib(4) is.

And already you can tell what we're going to do next. We're going to start exploring the exact same way like we did before. Fib(4) needs to calculate Fib(3). It can't do Fib(2) yet because Fib(3) needs to calculate Fib(2) and Fib(1), pass back up the value. Fib(4) can now finish its job by calculating Fib(3) and Fib(2), pass up the value, and now, finally, Fib(6) has its two halves here, Fib(5) and Fib(4). And it can add them together and return the value.

OK. So a super inefficient algorithm because there's a lot of stuff going on but not much work being done until the end. We've got a bunch of base cases we get to. And then we can start building back up our result. And the reason why I say it's inefficient is because, well, we're exploring these paths. And as we go along the way, we figure out what Fib(3) is and what Fib(4) is. Right?

But then when we explore the right half of Fib(6) over here, we're actually recalculating these values all over again. That's why I said we're not even halfway done because when we got Fib(5), we had to explore Fib(4). And Fib(4), this branch down here, is basically a copy of this one down here. OK. So there's a lot of work being done here where you just do the same thing over and over again.

And so that leads me to say, well, what if we didn't have to do all this work all over again? If only there was some sort of data structure that we could use to keep track of things as we calculate them. Right? Right? To basically map one thing to another. So if we already calculated Fib(4) to be some value, why don't we just look it up? So any time we use things like keeping track of and looking things up, that should ring a little bell that says dictionaries can help us do that. And so what we can do is actually write a more efficient recursive Fibonacci function that it's still recursive but it uses dictionaries to keep track of values as we calculate them.

OK. And so this is the Fibonacci efficient function. So my name is fib_efficient. Notice we're still calculating Fibonacci of some n but we're going to pass in another parameter, a dictionary. And this dictionary will keep track of the Fibonacci values as we calculate them. So the key will be the n and the value will be fib of that n. And so down here you can see we're going to initialize a dictionary that has fib of 1 maps to 1 and fib of 2 maps to 1. Those are our base cases.

So let's take a look at our Fibonacci recursive function now that uses dictionaries. No longer do we need to think about the base cases as Fibonacci of 1 is this and Fibonacci of 2 is this. Now, all we need to do is say, well, let's look up the value in our dictionary. That's what our base case will be.

And we don't need to make a call to ourselves if the item is already in the dictionary. So we can just return the value associated with n in dictionary d if that n is already in the dictionary. So our two base cases down here will initially be in our dictionary. And as we figure out the values of Fibonacci, we'll add them to our dictionary.

And that's exactly what the recursive step will do. So else the value is not in our dictionary. So unfortunately, we have to calculate it, which is fine. We'll basically do that the first time through that sort of exploring the left half of our path. But that's pretty much the only times that we're going to calculate it. All the other times, we'll just look it up.

So this is going to be a little different than what we've seen before because I'm not, right off the bat, returning fib n minus 1 plus fib n minus 2. I'm actually still running the same recursive step, fib n minus 1 plus fib n minus 2, but I'm saving it in a variable. And that's totally fine to do.

And then before I actually return this value, let me add it to my dictionary. So this is simply just saying this dictionary at this particular n for this particular function is equal to this thing that I just calculated, just a straight-up dictionary addition, adding this item to the dictionary. And then after I've added it to my dictionary, I can return the answer-- or return that value. So still passing it back up the chain of function calls, but we'll save it first. Everyone OK with this code? OK.

So then this is the dictionary I mentioned where we initialize our two base cases. And then we can print the function. So let's trace through the code to see what exactly happens with these function calls now.

So we're initializing our dictionary, where we have n1 Fibonacci of 1 is 1 and n2 Fibonacci of 2 is 1, our base cases. Fibonacci of 6 again. We're doing the same function calls, so that means there's nothing stored for fib 5, so we still have to explore what value it will be. Nothing stored for fib 4. We're still exploring. Nothing stored for fib 3. We're still exploring. We've reached a base case.

So now, the first thing we do is check if it's in the dictionary. It is, so we just return the 1 directly. Check if the other half is in the dictionary. Return the 1 directly. And now, we've got a value for fib 3. Before returning it, let's store it in our dictionary. So I just calculated what fib 3 was. Let's put it in. The key is 3 and fib 3 is 2. So far, so good. It's pretty similar to what we've done before, except that we're storing this value in the dictionary.

So now, we explore the right half of this fib 4, fib 3 plus fib 2. It's already in the dictionary, so it immediately returns this addition. Now, we know what fib 4, is so we add it to our dictionary. fib 4 is 3. Explore the right part of fib 5, so fib 4 plus fib 3. Do we go further now? In the previous case, we explored 2 and 1. In this case, do we keep exploring? No, exactly, because our base case says, if 3 is already in the dictionary, simply return the value associated with it. So yep, there it is right there. We added it a while ago. We just return the 2 immediately. No need to go down this path.

So now fib 5 is done pretty quickly. So the right half. So that means we have the value for fib 5 and we add it to a dictionary. We explore the right half of fib 6. Remember, beforehand, I said we were not done. We don't need to explore this fib 4 anymore because we added it to our dictionary long ago. So now, all we need to do is look up the value associated with 4 from our dictionary.

So boom, there it is. And then we can just add fib 5 and fib 4 together and get the value for fib 6, store it in the dictionary, and in this case, it's the end. We don't need to do anything else with this value, passing it back or anything like that.

So we're not recalculating anything else. We're just checking the dictionary, and if need be, we calculate it. So it's an improvement. But how much of an improvement is it, actually? So if we run this function-- and it's in the Python code. You can play around with it yourself. If you run the function that we originally wrote, the one where we don't store anything in dictionary, if we try to calculate Fibonacci of 34, it results in 11.5 million function calls.

That's a lot of function calls, because even fib 6 had fib 3 being called twice, fib 4 being called-- fib 3 being called three times, fib 4 being called twice, things like that. So can you imagine how many times fib 3 will be called when we are trying to calculate fib 34? Probably thousands, if not more.

So overall, the number of function calls we're making is 11.5 million with our original code. But the efficient version only makes 65. It's not like we went from 11.5 million to 2 million. We went from the order of millions to tens, which is really, really impressive in terms of speed. So if you try to run this program, it'll take a couple of seconds for fib 34, but the efficient one will be instant.

And all of these function calls have some overhead. You need to create an environment in Python. It needs to pass these parameters. So all of these function calls take a lot of time, whereas a dictionary lookup is basically instantaneous. So in this particular case, we've given up some of our memory to store values.

The dictionary is storing 34 entries, which is not much, but there are applications where you can't spare 34 entries in your memory, in which case, you might spare some time to continue calculating without taking up some memory. So there's a little bit of trade-off between these two programs. One of them doesn't store, anything but is slow. The other one stores things, but is fast.

Let's look at one more example where we do Fibonacci on numerics. And I don't know when you'd use Fibonacci in your real world-- real life, but knowing all the possible ways you can make a score of x in basketball is a little bit more useful. So let's think about this problem recursively. Certainly, we could do it iteratively and brute force our way through all the possible combinations of scores.

So in basketball, you can make a basket that's worth 1 point, 2 points, or 3 points. So you can think about all the possible combinations you can make to give you some score of x. We're going to think about this problem recursively. So let's start with our base cases.

Base cases-- we've got three of them. So if we think about a score of 1, so if x is equal to 1, so that means if we have a score of 1 in basketball, what are all the possible ways we could have made a 1? Well, you could just score one point and then that's it. I just did 1 plus 0 just to emphasize that we're just scoring 1 and nothing else.

If we make a basket that's worth 2 points-- or if we have 2 points in basketball, what are all the possible ways we could have made 2? Well, we could have scored a 1 and a 1, or we could have just scored 2 right off the bat. So that's two possible ways to make a score of 2. And similarly, to make a score of 3, what are all the possible ways well? Well, we could have scored a 1, then a 1, then a 1, we could have scored a 2 and a 1, or we could have square root of 3 right off the bat. So that's three different ways you can make a score of 3 in basketball.

Everyone with me so far? These are our base cases. Because the recursive step will be very-- will blow your mind. It's so simple. So the recursive step looks like this. Now, somebody give me-- what's a reasonable basketball score for a team? 87? OK. It's been, probably, 25 years since I've played pro basketball in grade 5, you guys, so I forgot what's a reasonable score.

So 87. So let's say, now, we're not dealing with our base case. We're dealing with some number that's bigger than one of these base cases. How do we think about this problem recursively? Well, there's three possibilities. If I have a final score of 87, let's say that I think about the score of 86. If I know all the possible ways I can make a score of 86, all I need to do is add 1 to that score. It'll give me 87.

So that's one possibility here. But that's not the only possibility, because I could have a score of 85, and if I add 2 to that 85-- not two counts, just the score. If I have an original score of 85, if I just add 2 to that score, it gives me my desired score of 87. So if I know the possible combinations to make 85, then I know that all I need to do is add a 2 to my score and that will give me 87.

And then the last possibility is to know all the possible ways to make 84-- a score of 84-- because then I would take that score and add a 3 to it to give me 87. So I'm using my base cases to guide my recursive step. So the number of ways I can make a score of 87 is the sum of all the possible ways I can make 86 or 85 or 84, because if I've made 86, I would just add 1 to it, if I made 85, I add 2 to it, and if I made 84, I'd add 3 to the score.

So that's essentially what this recursive step is doing. I've got, these are all the possible ways I can make a score of 80-- x minus 1. So 87, 86. And that's just me calling my function. So score count x minus 1, score count x. Plus all the possible ways to make a score of x minus 2 plus all the possible ways to make a score of x minus 3. So if I add all these three ways together, I would get all the possible ways I can make a score of x. Does that make sense? OK.

So that's it. It's pretty clean code. It looks really nice. If we were to write this iteratively, it would be a mess, because we'd probably have a whole bunch of nested loops to try to brute force all the possible combinations of scores that we can make, and it wouldn't look very nice, very pythonic.

So let's do a trace of this code just to bring it all together. The trace will be very similar to the Fibonacci trace, except that now, we have three paths to explore before having a return value. So for a score of 6, I would explore, how can I make a score of 5? And of course, I will explore how can I make a score of 4 and 3, but I'm not there yet.

First, I need to explore how to make a score of 5, which is a function call. This one will explore how to make a score of 4, and of course, a 3 and 2, but not just yet. A score of 4 will lead us to our base cases. It's just how to make a score of 3 and 2 and 1. These are base cases. They immediately return and we know how to make a score of 4. A score of 3 is also a base case and a score of 2 is also a base case. So these ones will immediately return to give us the score of 5.

So now we know how to make a score of 5, we need to follow through how to make a score of 4, which is just 3 and 2 and 1. Whoops, I should change that to be a 1. And then how to make a score of 3. And that's just a base case. So very similar traits as the Fibonacci code. Questions about those examples? Are they OK? Do they make sense? OK.

So there is one exercise in the Python file. It's for at home. I would like you to try to memo-ize this code. So memo-ize means, basically, try to use a memo, like a dictionary, to store values as you calculate them, because you see that it's going to be just as inefficient as the Fibonacci code. So here, we're calculating a score of 4 again where we had calculated it way back here. And so try your hand at adding a dictionary to this code to try to speed it up.

So the second half of this lecture, we're now going to move away from recursion on numbers and having these nice mathematical operations that we can just translate to code easily and start looking at recursion on non-numerical things. And we're just going to look at lists, but again, as I said, you can apply these very similar codes to any sequences of values, tuples or strings or things like that.

So the reason why we're looking at lists is because lists are naturally recursive. So one of the motivations I gave at the end of the last lecture is that we have lists that can have elements that are other lists that can have elements that are other lists that can have elements of other lists. So without knowing how deep these lists within lists within lists go, it's going to be really hard to write iterative code. It's possible, but it's going to be really hard.

And instead, we're going to see that the recursive version of this code is going to be a lot more intuitive in the long run-- maybe not right off the bat, but definitely, it's a lot easier to write and to read. So let's think about lists in a recursive way.

So if we were doing iteratively, what we'd say is, we're going to loop through each element and do something. The problem we're going to solve is figuring out the sum of all the elements in a list to begin with. So iteratively, we just said we loop over each element in the list and keep it in our result. So I've got these state variables I talked about last time, result in e, that keep track of which element we're at and what the value is.

Recursively, remember, we're going to make all these function calls until we get to a base case, at which point, we're going to start to build up our result. So how can we think about this list recursively? Well, let's say that we have a list and we want to find the sum of all its elements. That's our original problem.

Now, let's say that we take the first element and we just extract it out. We have this list with a bunch of elements. Let's take the first one. We know it's a 10. And then let's consider the remaining elements, so the 20 onward. If I take my 10 and I know the answer to the sum of all the elements in 20 onward, then all I need to do to figure out the sum of my original list, this one here, is to say it's the 10 plus the sum of whatever the sum of the 20 onward is.

Now, the sum for elements 20 onward is the same problem again. It's the problem of finding the sum of all the elements in a list. It just so happens that our list is now our original list without that first element in it. Does everyone understand that? We've got our original problem and we've just made the same problem again, just a slightly different version of it-- all the list except for that first element.

So now, we do the same thing. Let's say this is our new list. We extract the first element from it and we consider the elements except for that first one as a new list. And again, if I knew what the sum of 30 all the way onto 60 was, all I need to do is add it to the 20 that I extracted and I would know the sum of this list. So we keep doing that. We take our list, extract the 30, and consider the remaining elements as a list.

Same deal. If I knew what 40 plus 50 plus 60 was, the sum of all the elements in this list, I just add it to the 30 and I have the answer to that problem. And we keep doing this, extracting an element and considering the remaining lists, all the way down to when we have a list with just one element in it.

Well, this is a pretty simple problem to solve. If I have a list with one element in it, the sum of the elements within that list is just the value of that element. It's just 60. So very simple problem. No need to keep going further, dividing this problem into smaller pieces. I already know the answer to this one. It's very simple. So this is our base case. And we know the sum of the elements in a list with length 1 is that element.

So once we reach the base case, we build back up our result. We take the 60 and we had extracted the 50 originally, so we're going to pass the sum back up to whoever called it, which was the function that extracted the 50. So now, the 50 plus the 60 is 110. Now, this 110 gets passed back up the chain. When we extracted the 40, we said, well, I'm going to add the 40 to the sum of the 50 and the 60, 110, which is 150.

Pass that answer back up the chain. When I extracted the 30, I said I was just going to add the 30 with the sum of the remaining things, which I figured out is 150. The 20-- I had extracted it-- becomes 20 plus the sum of everybody else, which is 180. So the sum is 200. And then finally, my original question was to extract the 10, add it to everything else, which is the 200 that we figured out. So the full sum is 210.

Does that make sense, this animation? OK. So we've got the division all the way down to the base case and building back up the result. So let's try to write it. So we're going to write it in pieces. So the function is called total_recur. It takes in a list l. We're going to recursively figure out the sum of all the elements in this list.

So we can have a base case when the list is empty. We can return 0. Up to you. Another base case, which is the one that I illustrated on the previous slide, is when the length of the list is 1. So when the length of the list is 1, what's the sum going to be? No need for recursion. It's just that element.

And so in these slides, what I've also included in addition to the code is a little example. So it helps you think about what the function returns. So in this base case, when the length of the list is 1, the list would look something like this. And all I'd need to do is return l at index 0, so the 50. And that's my sum. And that's what I'm doing here, returning l at index 0. Cool.

Now, the recursive step. Remember, in the recursive step, I extracted the first element and I said, let me save this first element. So here it is being saved as l at index 0. And I'm going to add it to something. So in this example here, I've got this list that's longer than 1. I'm extracting the 30, l index 0, and I'm going to add it to something.

Well, that something based on the slides. The previous slide, where I did the animation, is going to be us putting our trust in the fact that we write this function correctly. That something is going to be us figuring out what the sum is of 40 and 50. It's the same problem we're trying to solve right now-- the sum of 30, 40, 50-- except that now, I'm just going to take the sum of just the 40 and the 50.

So that something becomes the same function we're writing right now, total_recur, except that I'm not calling it on l, not the whole thing all over again. That would be bad. But I'm going to call it on L from index 1 onward, so essentially removing that first element. Is everyone OK with that?

So that's it. That's the function. Nothing else to write. No loop. We've basically written a function assuming that we wrote the function correctly, which is a very strange way to think about recursion, but that's essentially what it is. You're trusting yourself to write this function correctly such that your recursive step leads you to the base case so that you can build back up the result correctly. So there's a lot of trust involved in writing these functions recursively.

So I'm not going to go through the Python Tutor, but you should definitely go through it on your own as a practice for the quiz, things like that. Let's have you write this, then. So it's going to be a slight modification to the code we just wrote. So it's going to take in a list as its parameter, and instead of summing the elements in the list like we did-- 10 plus 20 plus 30, whatever-- I would like you to sum the lengths of the elements in the list.

So if I pass it in this function, it's going to sum the length of this 2 plus the length of this 1 plus the length of this 5, 2 plus 1 plus 5. So it'll be a very slight modification to the code that we just looked at. And here it is on line 70-ish. So think about the base case. If you have a list with one element in it, what do you return? And if you have a list with many elements, how can you put your trust in something that you just wrote to help you get to the answer?

What do you guys have for me? So let's start with the base case. And if you're having trouble, I encourage you to, just in a little comment, just write down what that base case looks like, like I did in the slides. It looks like this. So what would I return if I have a list with one element in it? Yeah.

AUDIENCE: [INAUDIBLE]

ANA BELL: Yep, exactly. So we would return the length of that element. So the length of whatever this is, ab, whatever. Awesome. How do we do the recursive step? Yeah.

AUDIENCE: [INAUDIBLE]

ANA BELL: Yes, exactly. Total len_recur with what list?

AUDIENCE: [INAUDIBLE]

ANA BELL: Yep. So we're going to extract that first one. So this will give us the sum of the lengths of everybody else.

AUDIENCE: [INAUDIBLE]

ANA BELL: Exactly. So we also need to add it to-- yeah, I'll add a 0. So it's fine to do it either before or after, because we're just summing these two values. So doesn't matter the order that you're summing them. So that's perfect. Any questions about this code? Yes.

AUDIENCE: [INAUDIBLE] is it more or less efficient than doing the classic first?

ANA BELL: Than doing the what?

AUDIENCE: The classic one [INAUDIBLE].

ANA BELL: So in terms of efficiency, this function will be slightly less efficient, I would say, because there's a little overhead in actually making a function call, whereas if you use a built-in operator, it's been optimized to work pretty fast. Yeah.

AUDIENCE: [INAUDIBLE] doing the plus equals, it's not doing this in the background?

ANA BELL: No. When it's doing plus equals, it's definitely not doing this in the background. Exactly. Yeah. But this is just-- I mean, I'm trying to show your recursion on something that you wouldn't typically use recursion on just to help illustrate the idea of recursion. Certainly, you can use an iterative algorithm, obviously, to calculate the sum of these elements. And it's more intuitive, more in line with what we've been learning so far. Excellent.

So now, let's look at a slightly different problem. So instead of finding the sum of all the elements in a list, let's tackle the problem of looking for an element in a list. Completely different, but we're still doing some sort of list operations.

We're going to start with an implementation that's not quite right. And we'll see why in a little bit. So let's follow the same sort of pattern that we've seen in the previous one. So let's consider a list of length 1. In this particular case, if I have a list with only one element in it, how do I know if that element is the one I'm looking for? Well, I'm just going to return this Boolean, whether l at index 0-- that element-- is the one I'm looking for, the e. So notice, this n list is passing in the list itself and the element I'm looking for. I think-- OK.

So then let's look at the recursive step. The recursive step, in this particular case, let's say it says, well else. We might think to say, well, if it's not the one I'm looking for, then let's look in the remainder of the list. So like we did in the previous case, let's apply the same function we're writing right now to all the elements except for the first one. And we're still looking for element e in those remaining elements.

So we can test it out. And if we actually run it-- again, please, I encourage you to do Python Tutor on your own. But we can test it out and say, if, in this particular case, 2, 5, 8, 1, if I actually run this code, it will give me true. So it found the 1 inside the list 2, 5, 8, 1, which is good. It's exactly what we wanted.

But if I change my input list slightly and I've got 2, 1, 5, 8, the element I'm looking for is here. The code will actually give me false, the one that I just wrote, which is not OK. I see the 1 is right over there. And so what exactly is going on? We can run the code here. So this is this code here. If you see that it gives you the incorrect value, one thing you could do when you're doing recursion is to put a print statement within the function itself.

So we can print, maybe, the list we're currently at and the element we're looking for and see exactly what's going on. So if I run it, it will say, well, first time through the function call, I'm looking for the number 1 in this list. The next time, I'm looking for the 1 in this list. The next time, I'm looking for the 1 in this list. And the last time, for my function call, I'm looking for the 1 in this list.

And already, we see something went wrong because as I was looking through these lists, I'm basically skipping over important elements. What this code is actually doing is only checking if the last element is the one you're looking for, because it basically ignores that first element in the code. The code here, yes, it extracts that first element, but it doesn't do anything with it. So that's our problem.

What we want to do is still look at further elements in the list. So that part of the code is correct. But we only want to do it in a certain situation. And that situation is when the element that we just extracted, l at index 0, is not the one we're looking for, the little else case.

So we still want to extract the first element if we have a list with more than one element in it. But as we've extracted it, check if it's the one we're looking for. If it is, return true. No need to keep searching the rest of the elements in the list. If it's not the one we're looking for, this else here, then we can look at the remaining elements in the list and run the exact same function we're writing to check if the element is in the remaining list.

Does this code make sense? Is it all right? OK. So the way I wrote this code is how I personally think about the problem. And if we run the code again, it'll give me the correct answers each time. But I wanted to mention that we can actually clean up the code a little bit and write it a little bit more Pythonically so it's a little bit nicer to read, it's more cleaned up.

But one of the things that was confusing for me when I first started learning recursion is that I would always see these beautiful, cleaned up versions of code that do the recursion, and that's not how we approach thinking about the problem. I can't come up with this nice form right off the bat. And this is one example, but there are certainly other examples of more complicated code where you see it and it's just-- it looks beautiful.

And yes, if I look at it, I can figure it out. And I say, OK, yeah, that makes sense. But I personally could never come up with it on my own. So as I was writing these lectures, I thought, well, how do I actually link about the problem? So I just went back one slide. And the way I think about the problem is to separate it into these smaller-- a bunch of different base cases, or a bunch of different cases.

And so that's what I've been trying to do in this particular lecture to help you guys understand recursion. It's, think about the case when we have a list with one element in it. How would you solve that problem? And then think about the case when you have a list with many elements in it. How would you solve that problem?

Yes, it's true. There are some pieces here that are repeating. So we've got L at 0 equal e is in a couple places. But you can do that cleanup later. So here, I've got two test cases that return-- two cases that return L at 0 so we can pop them into the same test case here. And then we can check if the length of the list is 0. We can add that test case. And else, we check the remainder of the list.

That's totally fine. And if it helps you think about the problem this way, that's OK too. But personally, for me, it was a lot easier to think about the problem in terms of a list with one element in it and then a list with many elements in it. And it's totally fine to have to write a little bit quote, unquote, "inefficient-looking" code to begin with.

Certainly, don't hardcode all the base cases. If length is 0, do this. If length is 1, do this. If length is 2, do this. But some reasonable base cases are OK to do. So this is just showing the simplified code.

One thing that I wanted to mention-- and hopefully, you've noticed this already-- is the function that you're writing, all of the returns from this function need to have the same type. When we wrote-- I'll go back a couple slides. When we wrote the function that calculated the sum of all the elements in a list-- so that's this one here-- what were we returning? Here, we were returning an actual number, and then here, we were assuming that this function returned an actual number that we can add to this actual number.

So every single return statement needs to return the same type of object, because if you don't, if you're assuming that the base case returns a list, but then at some point in the code, you're going to be working with a number or a Boolean, then Python, as soon as it gets that base case, is going to say, hey, you're trying to add a Boolean to a list. What's up?

And so in the summing of the list elements, all the test cases returned a number, and in this case where we are trying to return whether the element is in the list or not, notice, every single one of my returns is going to return a Boolean. So here, Boolean, here, a Boolean, and here, in the recursive step, I'm assuming that I'm just passing this Boolean back up the chain of command.

So very, very important thing. Again, something that was not made clear to me when I first started recursion. But once I knew this, it just made so much more sense and it helped me write my code better, more perfectly, right off the bat.

Let's look at a slightly different example now. So we've looked at taking the sum of all the elements in a list. We've looked at figuring out whether an element is in a list. Let's do something completely different. Still working with lists, let's say that we now have an input list that looks like this. So we've got a list. This is my list, beginning and end.

And this list only has list elements within it. So no integers. But its elements are lists. So here's one list element. Here's another list element. And here's another list element. So in this example, I've got a list with three list elements. What I'd like to do is to flatten this list, which means that I want to remove any semblance of sublists and take just all the elements of these sublists and put them top-level.

Does this task make sense? OK. So I'm not assuming I've got lists within lists within lists. I'm just assuming I've got lists with list elements that have integers or whatever in them. So again, let's think about the base case, let's think about the case when we have a list with one element in it, and then we can figure out the recursive step.

So if I have a list with one element in it-- again, I've got an example here on the right-hand side. It's a list with one list element in it. That's why I've got the double square brackets. If I wanted to flatten this, what could I do? I could just grab the element at index 0, because the element at index 0 is this inner list, and it is a flattened version of my list.

Else, what am I going to do? Well, let's do the same pattern. It seems to have worked so far for us. Let's do the pattern of extracting that first element. So grab element at index 0. So here, we would grab something like square brackets, 1, comma, 2, and concatenate it with something. OK, remember, when we concatenate a list with another list, it gives us a big list with all the elements in it, exactly what we're looking for when we want to flatten a list.

So the something we're going to add this l at index 0 with is just us flattening the remainder of our list. Again, same pattern we've been seeing already. So if I extract, in this example here, the 1, comma, 2 as a list, I'm going to concatenate it with the assumption that the function I'm writing will work correctly to flatten 3, comma, 4 and 9, comma, 8, comma, 7. So if I flatten that, this will give me just a list with 3, 4, 9, 8, 7 in it. And if I concatenate 1, comma, 2 with 3, 4, 9, 8, 7, that just gives me 3, 4, 9, 8, 7.

Everyone with me? Is that all right? OK, good. I see some nods, so that's actually a pretty good sign. OK, you are with me, right? Because now, it is your turn. So we're going to write a variation of whether an element is in a list. So I'm going to give you a very similar scenario to this flattened one. So I'm going to give you a list that contains list elements. So here's my list that contains list elements in it.

And what I'd like you to do is write a recursive function that checks whether this element, whatever the second parameter in my function call, is in these list elements. So not at the top level, like we wrote the at the last code to check if an element is in a list, but in these sublists.

So just to show you the difference, if I check whether 3 is in 1, comma, 2, comma, 3, that will be true. But if I check whether 3 is in the list containing the list 1, comma, 2, comma, 3, that's false, because it's checking whether the 3 is equal to this list. It's just doing a top-level equality here.

So let's have you write this code down on line 166. You may use the in operator to check if an element is in a list itself, but obviously, you won't be able to use in operator-- nor should you, because then we're not writing a recursive function-- to check if the element is within a list element. So have you work on it for a couple of minutes, and then we can write it together.

Does anyone have a start? So let's look at the case where we have one element in it. How do you check whether that element is within the list inside? So if-- this is our case with one element in it. The length of L equals 1. Yeah?

AUDIENCE: Um, I'm not sure, but [INAUDIBLE] e and L [INAUDIBLE].

ANA BELL: Yeah, exactly. e and L is the correct thing to do. L at index 0. So if this is our l-- that's why I added this little example here, so it can help us. So L at index 0 is this guy here, and all I need to do is check if e is in L at index 0. And I can just return that right off the bat.

I could do if e in L0, return true, else, return false. But e in L0 is already a Boolean. So I can just return that directly. else, we have a list with more than one element in it. So what do we do here? Remember, extract the first element and then do the rest.

So let's say this. Let's say the first element is L at index 0. That'll help us think about it a little bit. So before looking at the remainder of the list and calling our recursive function, what did we do when we checked if an element was in a list when we just had a plain list? We just said if e is in first, return true. else, return false.

But we don't want to do an else return false, because that's not quite true. else, we want to look at the remainder of the list. We want to see if the-- obviously, if the element is not in the first thing that I just extracted, this list here, then I would like to say, is it in the rest of this list, which is us calling the function that we're just writing all over again.

So we can return the name of this function in lists of-- what did I call it? Lists of lists. And then L from 1 onward with the same element we're trying to find. And of course, we can simplify this just like we could simplify the previous one. But it helps to think about it in these two cases, a list with one element and a list with many elements. Any questions about this? Yes.

AUDIENCE: [INAUDIBLE]

ANA BELL: This one? This one, we're considering a list with one list inside it. Yeah. We could include another base case, I suppose, if the length of L is 0 returned false. That would also work, because obviously, if the list is empty, then it's not in there.

So when do we use recursion? Obviously, a lot of the examples we've seen here, they're very intuitive to write iteratively. But I mentioned a couple examples last time, where it's more intuitive to use recursion. And specifically, I wanted to draw a little bit of a parallel to this thing when we learned about while loops. We said, well, what if we tried to code a little game that just used if and elses?

I said that we would have a bunch of nested if/else statements without a while loop, because we don't know how deep to make these if/else/if statements. And so a very similar idea exists with recursion and when to use recursion. So if I had a list with a whole bunch of lists in it, and those lists could have lists within it, and so on and so on, I don't know how long I need to-- how deep I need to make my code go.

So an example using a for loop would be to say, for each element in L, I'm going to say, I'm going to look at each element. I'm going to say, well, if you're not a list, then I can deal with you directly. But if you are a list, then I need to iterate over you. And so I've got this other iteration here for each j in i for one of those lists. Again, I would say, are you a list? If not, I'll deal with you directly. else, you are a list, so I do need to iterate over you.

And you can see this nested idea now comes into play here. And of course, we could try to use a while loop to optimize the code a little bit, say, while this element type is not a list, do this, things like that. But it leads to some really verbose code. And so recursion is a way for us to deal with these lists within lists within lists, and of course, when you have data structures that you don't know how long-- or how deep they go.

So I mentioned file systems and a set of operations last lecture as really nice places to use recursion. Scooby-Doo gang looking for their culprit, rooms that have doors that lead to other rooms that have doors lead to other rooms, they don't know how many doors they need to go through to get to a room without doors. Obviously, recursion, they should use. And then a bunch of other fun examples of places to use recursion.

So the last bit of class, I would like to work through this example where we're going to see the code to solve lists within lists within lists within lists. But before we do that, we're going to talk about-- so we're going to do that example in the context of reversing a list. But before we look at a list that has all these different sublists within it, let's look at a list that has just integers.

How would we think about this problem recursively to reverse all the elements in this list? So again, we're going to use the very same pattern we've been using all throughout today when we've been dealing with lists. We're going to take out the first element, extract it, and we're going to deal with the remainder of the list, basically, by running the same function we're writing on the remainder of the list.

So let's say I have my original list and I look at my first element, just like before. I'm going to extract it out. If I take this first element and I pop it at the end, and then I consider the remainder list, everything except for that first element that I put at the end, I can just call the same function I'm writing right now to reverse the remaining list, which means that I'm going to take this remaining list, grab the first element, pop it at the end, and deal with the remaining list.

Again, take the first element, pop it at the end, deal with the remaining list, until I have a list with length 1. How do I reverse a list that only has one element in it? It's just that list. Reversing a list L is just L. So that's the idea.

And notice that when we're building back up the result, we took that first element and we tacked it onto the end. So we're going to do another list concatenation kind of deal, except that the thing that I'm concatenating now, the first element, will be at the end. It'll be the second part of my plus. So I'm just giving you a heads-up. That's what it will look like.

So let's write the code. If the length of the list is 1, if I'm reversing a list with one element in it, just return that list. Easy-peasy. It's just the list itself. else-- and this is where the fun comes in. I've got something. So I'm going to do something concatenated with something else. So I'm extracting the first element. There it is, L at index 0. But it's sitting somewhere funny that we haven't seen it sit before. It's sitting on the second to the right of the concatenation. And that's fine, because what we want to do is take the element from the beginning of the list and tack it on to the end.

And there's something else that's funny about it. I've put it in square brackets. Now, again, I'm including this example to help us think about it. Why are those square brackets there? Think about what we want this function to return. Is it returning a number? No. Is it returning a Boolean? No. It's returning a list. This function, I want to take in a list and give me back a list, but where my elements are in reversed order.

So what I want to do-- you can already see this return over here is returning a list. So it'll be square brackets 10 or whatever. In my recursive step, if I'm concatenating, I want to concatenate this thing here, which I'll tell you about in the next slide. But I'm going to concatenate it with-- it's going to be a list with some other list. If I concatenate a list with a number, that L at 0 is-- L at 0 is a 10.

So if I concatenate a list with a number, Python will yell at me. So what I need to do is make that number that I just extracted, L at 0, be a list. So I'm just going to slap a square bracket around it and say, hey, Python, this is a list with one element in it. Does that make sense? Cool.

So then what that means is, I've got this 10 that I extracted. I'm going to concatenate something with that 10. And that's something is me putting my trust into the function I'm writing to say, that something is going to be the 20, 30, 40 successfully reversed, 40, 30, 20. If I can do that, 40, 30, 20, and I concatenate it with a 10, my job is done. I've successfully reversed 10, 20, 30, 40 to be 40, 30, 20, 10.

And so let's just do that. That's me putting my trust in this function I'm writing. I'm calling the same function again, saying, hey, I would like to reverse the remainder of the list, exactly as we have been in the past. Super weird to think about still because we're trusting something that we're writing. Cool.

So then let's test it out. Let's run it. So if I run it with list 1, 2, abc, Python will reverse my list. So it will print abc, then the 2, then the 1. Let's say I run it now with something slightly different. So I run it with this list here. How many elements does this list have? Test. You guys tell me. Three. Exactly. The first one is an integer, the second one is a list, and the last one is a list that's got a bunch of garbage in it, but as test, I don't care, because I just care that I have three elements inside.

And so when I run this function on test, it will reverse just the top level, because that's what this is doing. Nowhere in here did I say I want to reverse lists within lists. I didn't say, if you're a list, also reverse yourself. I just said, top-level, take this element, put it at the end. So when I reverse test, this funky-looking test over here, it will take that first element, put it at the end. The middle element stays where it is and the last element becomes first. is everyone OK so far? I'm worried there aren't many more questions. OK.

So that's good. But this is now not really what I'd like. What I'd like is, if I have lists within lists within lists within lists, and those lists have some sort of elements within them-- at the lowest level, I've got a list that's going to have some integer or string or whatever in it-- what I would like to do is to reverse those elements as well. So really, what I would have liked to have if I passed in this function here-- this list here-- is to say, well, why don't you reverse everything?

So I would like to have had gf as a list, and then the e, and then the d, and then the 1. And so this is where we're going to do that. So let's say I now have a list. So each one of these blue squares is my list-- or my list elements and my top level. And they happen to have some sort of lists within them.

How do I do this? Well, now that I have potential list elements, I need to have my recursive function test whether the element I'm currently considering is a list or not. If it's not, like the 3 and the 4, I can treat them in the exact same way that we treated them in this case. But if it is a list, as this one is-- this is a list element, and this is also a list element that has list elements within it. So that's even funkier. Then we need to consider them separately.

So let's take the code that we wrote in the previous slide, because it's a good start, extract the first element, put it at the end. That's what we did before. But before leaving, let's say, if you are a list, if you are a list, then also reverse yourself. So not only do I want, top-level, that list-- that element to go to the end, I also want to consider what you are. I don't want this last element to be 1, comma, 2. I want to reverse its elements too, to be 2, comma, 1. So in the end, what I want this to give me is 87654321.

So that deals with that first element being popped at the end there. Now, I consider my new list. And again, this is going to be a recursive step. The element at the front, again, I extract it. It's just a number. Nothing special here. So you just go to the end, just like before. Nothing to consider. Nothing to reverse for that 3. Again, the 4, just like before, it goes to the end. And now, what about this list with lists within it and so on?

Well, we've reached sort of this quote, unquote, "base case." So there's nothing to put at the end. But you can imagine being put at the end if there were other elements within it. So this one is going to stay as is. Sorry about that. This one is going to stay as is. But what we're going to do is going to say, well, you are a list, just like this one was a list. It was a list with two numbers in it. So you are also a list with two elements in it. So the first step I would like you to do is reverse yourself. So the 7, 8 will come to the front and the 5, 6 will go after it.

But its elements also are lists. So not only do I want to reverse you, but I want you to tell all your elements to reverse themselves. So the 5, 6 should reverse to become a 6, 5, and the 7, 8 should reverse itself to become 8, 7. Does that make sense? Conceptually, I think we got it. So we want to reverse as far deep as we can until we get to some numbers.

So let's write the code. We're going to do a very similar thing to what we've done in the past, write all of these examples following the exact same pattern. Consider a list with one element in it, and then consider a list with many elements in it. If I have a list with one element in it-- so here, here's a list. It's going to have only one element in it. If the list is a-- if that element within that list is a number, I'm going to do something different than if the element within this list is a list.

So what I actually want to do inside this if len(L) is equal to 1 is have two subparts depending on whether it's a list or not, because if it's just a number, I'm happy to just leave it as is. This number is already in place. It's already reversed. But if the element within it is a list, this element is one element inside my list is also a list, I want it to reverse itself. So if the length of the list is 1, I now check the type. If it's not a list, I do exactly the same thing as I did before. If it's not a list, you are already reversed. No need to reverse anything else. Yes, question.

AUDIENCE: I guess I'm a little bit confused by the one element [INAUDIBLE] first part.

ANA BELL: Yeah, so we're just dividing it into one element or two-- or more than one. So in the case where we have one element, this is my list. And this is the one element. And if I have an element that's a list itself, then this is still one element. Yeah.

AUDIENCE: Let's say you raise the brackets on the outside of the [INAUDIBLE].

ANA BELL: Yeah. This is now a list with two elements in it. Yep. Yep. Exactly. But I am considering the case where I have a list with one element. It happens to be another list. And what's inside it, I don't currently care, because-- yeah. So if it's not a list, it's already reversed. Otherwise, what do we do? Well, we want to ask it to reverse itself. And that's the function we're currently writing. Is that cool, I guess? OK. Again, a lot of trust going on here, you guys.

So we're calling deep reverse, this function we are currently writing, on this list element, L at index 0. It's our only element. And notice, again, I've got these square brackets around here because this function is supposed to return a list. So just like in the previous case, where I slapped on some square brackets around the number, I have to do it here as well. Everyone OK with this case? Because the recursive step is going to be even crazier.

OK, else, we have a list with more than one element in it. So we have a list with some stuff here, and then I have, potentially, another list and a bunch of other stuff here, like this, whatever it is. So then what I would like to do is, again, according to our sort of pattern that we've been looking at, is to say, I'm going to extract the first element in the list. So if I have a list with many elements, let's extract the first one and deal with it.

But again, I need to take care, because that first element may be a number or a string or whatever, or it may be a list. And I deal with these two cases separately. If it's just a number-- so that's this if case here. So if the type of L at 0, the thing that I've extracted, is a list, then what I need to do is what I had in the previous example.

I grab that first element, slap square brackets around it, and concatenate it with deep reverse of the rest of it. Exactly the same as the previous case, because it's just a number. I do what I did before, plop it to the end and we're done. And again, I'm making a function call here to myself.

else, this thing here, this L at index 0 that I've extracted, is a list. So not only do I have to call deep reverse on these guys here, but-- everybody together-- we have to call deep reverse on the first element as well, because it's a list. I can't just put it to the end. I want it to reverse all of its elements. So this is the code to do that.

This bit here, deep reverse L 1 colon, tells the remaining of the list-- the remainder of the list to reverse itself, exactly like we did in the integer case. All is the same. But we concatenate that, again, by putting square brackets around it, because we want to concatenate with the list. We concatenate that with deep reversing our element at index 0. So not only do we put this at the end to reverse it, but we need it to reverse all of its elements as well. There are no more lines to this code, but what are your thoughts?

AUDIENCE: [INAUDIBLE]

ANA BELL: I know. Yeah.

AUDIENCE: Why do you keep putting the square brackets?

ANA BELL: So yeah. So we put square brackets because we want to maintain the same structure of what the original list was. So if it's an integer, is, I guess, the simplest case to explain it. So if it's an integer, you can't concatenate the list with the integer. It will be a problem. So you want to concatenate the list with the integer inside a list as a single element.

So what we can do is we can simplify the code. Again, I personally think of this as a little bit easier to think about, just as I'm extracting out the case where I have a list with one thing and a list with many things. But you can certainly think of it like this. So if I have an empty list, just return an empty list. Else, I'm extracting the element at index 0 directly, and I deep reverse that-- the rest of the list-- concatenated with that element at the end. Again-- oops-- noting that we are putting this element as a list.

And else, we can deep reverse the rest of the list concatenated with deep reversing this guy here. So not only do we put it at the end, but we also reverse all of its elements. So this is the simplified version-- the simplified code. So this recursion that we saw, all these examples here that we applied to lists can actually be applied to any indexable ordered sequences.

The same code will work for tuples. The same code will work for strings, except for the one where-- because you can't have strings within strings within strings. But certainly, summing the elements in a list and checking whether an element is in a list will work for tuples as well, and some of these will work for strings, as long as you can do that operation on the strings, because these are all indexable sequences, so it shouldn't be a problem.

So lots of takeaways here with recursion. This last example, namely, it looks really nice in the cleaned-up form. And it's, what, five lines of code to solve this really kind of hard problem that you would otherwise have to solve using while loops and for loops and things like that. So I definitely encourage you to take a look through the Python Tutor links that I've put in.

My two tips, so the two big takeaways on recursion, is this thing about base cases-- or cases. Any time you have a return statement and you're writing a recursive function, make sure that every single return statement is returning something that is of that same type. Otherwise, you'll have type mismatches all over the place. And then the recursive step takes advantage of the fact that you are returning these kinds of types. So then those operations in the recursive step will work with those types.

And the second is, the function doesn't have to be efficient on the first pass. So the way we thought about the problem by separating it in a list with one element and many is easier for me to think about because I can wrap my head around the problem. And you don't have to write the most efficient code right off the bat for recursion. Certainly no need to do that. But you can definitely clean it up after you have something that works.

Many practice problems on the Python file for today. Many, many practice problems. Memo-izing the basketball. Obviously, I mentioned that. An example-- a little practice with no lists within lists, a practice with lists within lists within lists, and then I included three buggy recursion implementations for you to try to fix. So a little bit of debugging practice plus recursion practice. Thanks, all.