ERIC GRIMSON: Good morning. Professor Guttag has to be out of town today. My name's Professor Grimson. I'm going to be subbing for him. And just to show you how much we value education here at MIT, and especially in EECS, you get the incoming chancellor as a substitute teacher. So how cool is that? All right, maybe not very cool. But I'm going to talk to you anyway for the next 50 minutes.

At the end of last lecture, I think Professor Guttag introduced dictionaries to you, a really powerful type. It's got a great capability, which is it's a tool, a data type that lets you association almost any kind of structure with a key. So it could be that you're associating keys with numbers, you're associating keys with strings. But you could be associating keys with other things, including dictionaries.

And I don't know if he showed this to you or not, but those keys themselves can also be really powerful. It could be just a string. But the key could also be a tuple, or a list, an x- and y-coordinate, or a set of names, or other things. So it's a really great data structure to use.

Now, one of the things you could ask, though, is, gee, what if Python or whatever language you're using didn't come with dictionaries? Could we still get the same power? And the answer is sure.

So I want to show you a little example. So on your handout, if we look at this little procedure up here at the top, key search, I could build it out of lists. I could just use a list as my way of storing it.

So let's look at what this procedure. Says it says, OK, if I've got a list, and I've got a key, K, I could write a little loop that just walks down the list, saying, is the first part of each element equal to the key I'm looking for? And if it is, I'll return the second part of that element. And if I get to the end of the list, I haven't found it, I return none.

So notice I'm making a choice here. I'm assuming that I'm storing in the dictionary things that are lists too long, key and the associated value, another key and the associated value, another key and the associated value. But this would work fine. So if I didn't have dictionaries, I could build it. If I wanted to make things that had a more complicated lookup, I'd have to make sure that that equality test did it. But you could see how I might do it.

So the question then can be, so why do we bother with dictionaries, if we could just use it with lists? Here's my question to you. So how long in this implementation is it going to take me to figure out if something's in my dictionary? Oh my god, he's asking questions at 10 o'clock in the morning. This is really terrifying.

Somebody help me out. How long? Yeah?

AUDIENCE: Probably on average on the order the size of the list.

ERIC GRIMSON: Yeah. On average, it's going to take me half the size of the list. But that's the same thing as being the size of the list. So if that list is really long, I'm toast. If it's not in the dictionary, in fact, I'm going to have to go all the way through the list before I get to the end to decide it's not there. So this is not as efficient an implementation as we'd like.

Now, the flip that is you say, OK, how long does it take for the dictionaries, the built-in associated retrieval that you have inside of Python? And the interesting answer there is that that retrieval is constant. It takes the same amount of time independent of the size of the dictionary. And that's wonderful.

We're not going to talk about how today. You're going to see that later on in the term. But it is something that drives home, if you like, the point that different data structures have different costs. And while some are easier to implement in, they may not be as efficient, some may be a little more difficult to implement in, but they are much more efficient. And that's one of the great things about dictionaries.

The second thing I think you saw at the end of last lecture, and I want to just highlight in a slightly different way, is I believe Professor Guttag showed you a little example of a very simple translation function. We had a little dictionary. I'm going to give you a version right here, and un-comment it. In fact, there's a little dictionary that simply associates English words with French words. And yes, it's pretty simple, but it's a little way of pairing those things up.

And the idea would be if I have a sentence just a string that consists of a bunch of words, I'd like to translate that English sentence, if I can, into French. And let's look at the code that's going to do it. I'm going to, again, un-comment it here. It's on your handout. I want to walk you through it.

So first thing I'm going to do is actually write a little procedure the says, given a word in English, I want to find the corresponding word in French. And that's pretty easy. It says, if you give me a, word, and you give me a dictionary, I'm just going to look it up in the dictionary. That's just that associative retrieval. It does all the work to go into the dictionary, say, if that's a key for something in the dictionary, it's going to give it back to me. And in particular, I can get it back by simply retrieving it out of the dictionary.

And notice I'm doing this a little more carefuly. I'm not just directly going, and saying, give me the thing corresponding to the word in the dictionary, because it might not be there. So that test basically says, if the word is in the dictionary, it'll return true. And in that case, I get out the corresponding element. If it's not in the dictionary, I'm just going to be stuck with the English words, so we'll just return the word.

Now, let's look at the more fun part of it. I now want to translate a sentence. And the sentence is a string of words. I need to find where the words are. I need to look them up. I need to gather together a translation.

So notice what the heart of this thing does, right down here in this part of the loop that I'm going to highlight right there. Oops, I mis-did that. Let me try it again, right there. What's that doing?

It's walking down the sentence. A sentence is some big, long string. And it says for each character C in the sentence, I need to find where the word is. So I'm going to keep walking along until I find a place where there's a space. So that first test there is saying, if C is not a space, just add it onto the end of the word.

Oh, yeah, I need to have inititalized word, which I did right up here. I said, let's set word to just be an empty string. So I'm just walking along the sentence until I find a space. Until I do that, I'm just gathering together all of those characters. So I'm building up the word.

When I get to the place where in fact C is a space, then here's what I'm going to do. I'm going to translate the word using that procedure I just wrote. It looks it up in the dictionary. It either gives me back the French word, if it's there, and if not, it just gives me back the English word.

And, oh yeah, I'm doing a whole long sentence. So I need to add it together. So I'm simply going to add it to the end of what I've translated so far. And I'm doing a slightly funky thing in there. I'm inserting a space in between, just to keep the words separate. And of course, I need to have initialized translation up here, which I did, to be an empty string.

So what's this do? Walks along character by character. When it gets to a space, it says, I've got a word. Find the translation, add it to the end of this other internal variable, and keep going. And I'll do that until I get to the end of that sentence. And then I'll just return the translation. Nice little iterative loop. And it's simply using two pieces to make all of this happen.

I lied too you-- sorry, I didn't lie. I misspoke to you. Which is, what's that funky thing at the end? What's it returning? It's returning this strange thing this says, "Translation," starting with the first element, a copy of all of that, plus a translation of the word. So what assumption am I making about the inputs that causes me to do this strange thing at the end?

I'm actually making two assumptions about my input. But I'm making a particular assumption. Why do I not just return translation when I'm done? Why am I doing this last piece? Anybody help me out?

Boy, I notice everybody sits way back at the back, where I can't make any eye contact. How do I characterize words in my string? They have to end with a space. Ooh, is there a space at the end of my example? So I guess I haven't shown you an example. But I'm assuming if I give you a sentence, you don't usually have spaces at the end of sentences.

So I'm making an assumption, is that the words, except for the last one, are characterized by spaces. So this last little piece here is to get the very last word, and translate it. Because it won't be a space.

Let's try a couple of examples. Let's just un-comment these, and see if this does it. Yes, commenting them again is not a good idea. We'll go the other direction.

Looks like it did right. John-- we should have translated John into Jean. I don't know what the hell Eric is in French, but John mange du pan, Eric boit du vin. John has that right. He's eats bread. I like to drink wine. And, of course, tout le monde aime 6.00. Or in this case, everyone likes 6.00. You just got HASS credit, by the way, for listening to me do French really badly.

OK, as I said, I made an assumption, which that the words end in spaces. What's the other assumption I made here? I know you're just looking at the code. But I made another assumption inside of my code. And it's always good to figure out what those assumptions are. Yeah?

AUDIENCE: That the first word is a noun, so we don't have to translate it?

ERIC GRIMSON: I didn't assume that. I could if I was doing a more clever translation. But I'm actually making no assumptions linguistically here. I'll give you a hint. How many spaces do I have between words? Just one. I'm sort of building that into this assumption.

It's a reasonable assumption to make. But it points out, if I wanted to improve the code, I should think about these other cases. And of course, to build a real translation system, I'd need something much more sophisticated.

So this shows you little example of using the dictionaries. But I want to use this to lead into the main part of today's lecture, which is, why did I write this separate little procedure up here called translate word? I could have buried that inside the code.

And the answer is twofold. First one is it saves a small amount of code. If I'm going to use it in multiple places, I don't want to rewrite the code. And that actually is valuable, not just to save you typing. It means I only have to debug it once. It's one piece of code.

But the real reason I wanted to introduce it is it gives me what you might think of as modular abstraction. Those are fancy words that basically say I am isolating that function in one place. If I decide to change how I'm going to translate words, I don't have to search through all of my code to find all the places where I was using that. I just need to change the definition of that procedure.

And I agree, in a simple example like this, it looks pretty simple. But if you've got a million lines of code, you're using it 10,000 different places, you don't want to have to find all the places to change. So this is an example of modular abstraction, isolating where that thing is.

And that is an example, in fact, of a general problem-solving principle we're going to use, called divide and conquer. Divide and conquer is basically the idea of taking a hard problem, and breaking it up into some simpler pieces, or into smaller problems, where those smaller problems have two properties. Small problems are easier to solve than the original one. More importantly, the solutions to the small problems can easily be combined-- I want to stress easily-- to solve the big problem.

And we're going to look at a bunch of examples today to show that. This is a really old idea. Julius Caesar used it. He did it in Latin. My Latin is terrible, but it's something like divide et impera, which literally means divide and rule. That's how he created the Roman Empire.

The British knew this really well. That's how they control the Indian subcontinent brilliantly for several decades. Ben Franklin actually knew it. In particular, he knew how good the British were at this. So there's a famous quote. You may well remember-- when he signed the Declaration of Independence, he said, quote, "We must all hang together, or assuredly we will hang separately," meaning divide and conquer is going to be a real problem.

So second HASS credit for you. We just did some history. Why are we going to use it today? Boy, you're a tough audience, I noticed, by the way. That's all right.

We're going to use it today, because we're going to use it as a tool for problem-solving. And in particular, we're going to use it with one particular example. You're going to see divide and conquer later on in the term, when Professor Guttag talks about algorithm design. Today, I'm going to show you one great technique for doing divide and conquer kind of algorithms. And that is the technique of recursion.

How many people here have heard that term used before by computer scientists? OK, not to worry. If you've heard it, you probably think that it's a really subtle programming technique. I'll let you in on a secret. That's a PR job. It's what computer scientists tell you to make you think that they're much smarter than they really are. It's not subtle, and it's much more than a programming technique. And we're going to talk about it.

Now, it gets used not just as a programming technique. It gets used two other ways in computer science. And I'm going to talk about both. It's both a way of describing or defining problems, so it's a way of characterizing a problem independent of how we might implement it. And it is a way of designing solutions.

So an example of divide and conquer. Let me give you an example just to show you of why it's a way of defining a problem. Consider the part of the legal code that defines the notion of a natural-born US citizen, and remind you, to be eligible to run for president, which I hope you all do, you have to be a natural-born US citizen.

Definition has two parts. Part number one, anyone born within the United States is a natural-born citizen. Part number two, anyone born outside the United States, both of whose parents are citizens of the United States, is a natural-born citizen, as long as one parent has lived in the US.

A little more complicated. What does that actually say? Well, it's got two parts. And that's what a recursive definition has. The first part is simple. You're born here, natural-born US citizen.

Notice the second part. It says, you weren't born here, you might still be a natural-born citizen. But what you have to do? You have to decide if your parents are US citizens-- which they could be by having been born here, but there are other ways to be naturalized-- and that may require you determining if your grandparents are US citizens, which may require you to-- ah. So you may have to chain down several sets of problems.

Those two parts are what exactly we're going to use when we talk about recursive definitions. There's what we call a base part, which typically describes the simplest version of the problem. And then there is an inductive part that tends to describe how you reduce the problem to simpler versions of the same problem. So in fact, let me write those both down.

We have-- it's called a base case. This typically gives us a direct answer. It just tells us very simply, using very simple methods, whether this is something that satisfies that recursive definition.

And there is the recursive or inductive case. Here, you reduce to a simpler version of the same problem, plus some other simple operations. OK, again, a bunch of words. Let me show you some examples.

I'm going to start with three or four different examples, just to show you how quickly we do this. What I want you to see in all of these examples is that when I describe the problem, I'm describing it in terms of what's the simplest case, and then how do I build solutions to bigger problems from solutions to smaller versions of the same problems?

OK, here's the first case I'm going to do. Suppose I tell you I want to build a little procedure to do exponentiation, integer exponents. I want to compute b to the n. But I tell you I've got a really cheap machine, and all I can do is multiplication. So I can't use exp.

Mathematician-- a course 18 person would say, what's b to the n? That's b times b times b, n times. How could I solve this? Well, recursively, I'd like to say, suppose I could solve smaller versions of the same problem. And if somebody gave me a solution to that smaller version, how would I build a solution to the bigger problem?

Oh, that's easy. That's the same as b times b to the n minus 1. All right, I can see you're all going, well, duh. I guess that's what chancellors know. They're not very bright. Of course, you know that.

But notice what I did. I've now reduced it to a simpler version of the same problem, recursively. This basically says I wanted to solve b to the n. It's b times b to the n minus 1. Oh, yeah, but I've got to figure out when to unwrap this. So there's an if there. And that's true as long as n is greater than one-- actually, as long as n is greater than 0. We'll do it that way.

And if n is equal to 0, I know the answer is just 1. I know you don't believe me, but that's really cool. Why is it really cool? What do I have? I have a base case. And if it's equal to 0, I know the answer right away.

If n is bigger than 0, oh, let's assume that I've got somebody who will give me the solution to that smaller problem. When I get it, I can just use it, do the multiplication, and I'm done. And so, in fact, here is a simple piece of implementation of that, which I know is not in your handout, but I'm just going to show to you, because I simply want you to see the form of it.

It's an exact Python translation of what I just said. It says, if I want to take an exponent of b to the n, if n is equal to 0, just return 1. If not, solve a smaller version of the same problem right there. Call to the same procedure, same function, but different argument. And when I get that answer back, just multiply it by b, and return that.

This may look a little funky. This is the kind of thing that your high school geometry teacher would rap your knuckles with, although they're not allowed to do that anymore. You don't define things in terms of themselves. This is not geometry. This is programming.

And this is a perfectly legal recursive definition of a problem. It will stop, because it will keep unwinding those recursive calls until it gets down to the base case. And I'm going to show you an example of that in a second.

I want to show you a much nicer example of recursion. And again, part of my message here is when you get a problem, don't instantly start writing code. Think about, how do I break this down recursively?

So I have brought some very high-tech tools with me today. This is my version of the tower of Hanoi problem. How many people know the problem of the tower of Hanoi? Only a few.

OK, so here's the story. There's a temple in Hanoi staffed by a bunch of monks. In that temple, there are three tall, jewel-encrusted spikes. Mine aren't nearly as fancy. And there are 64 golden disks, all of a different size.

Stack starts out on one of those spikes, and the monks move one disk at a time. Their goal is to move the entire stack of 64 from one spike to another. And the rules are they can it of one disk at a time, but they can never cover up a smaller disk with a larger disk.

I have to tell you, I don't know what happens when they move the entire stack. I mean, the universe ends, or you all get A's in 600, or something equally as cool. Question is, could we write a piece of code to help the monks, to tell them how to move them?

All right, so let's figure this out. And I'm going to show you some examples. So I'm going to move a stack of size one. Well, that's not very hard. Watch carefully, because you're going to write code to do this. I want to move a stack of size two, so I've got to just make sure that I move the bottom one off. That's not so hard.

Now, I want to move a stack of size three. I've got to be a little more careful, because I can't cover up the smaller one with a larger one. But that doesn't look very hard.

Got the solution, right? Now, we go for stack of size four. This one definitely takes a little bit more care, because you really can't cover it up. But as long as you do it right, you can actually move the-- oops, and I didn't do it right. You've got to move the pieces in the right way. I do this for taking money off of Harvard students in Harvard Square, by the way.

Got it? Real easy to see the solution, right? You could write code for that right now. I'm not going to do five, but it's really easy to see the solution. Yeah. I blew it, too. I did one move I had to backtrack on.

Let's think about this recursively. What's the recursive solution? Break it down into a simpler problem, or a problem of a smaller size. Ah, here's the solution.

To solve this, I've got a stack, I've got a stack I'm going to, I've got a spare stack. What's the solution? You take a stack of size n minus 1, move it onto the spare stack. Now I've got a simple problem. I can always move a stack of size one. And then I move a stack of size n minus 1 to the target.

And, of course, how I move a stack of size n minus 1? Well, I just unwrap it one more. That's a really easy explanation, right? And it's really easy to write code to do exactly that. So let me show it to you.

Again, I know this isn't in your handout, but I just wanted to see it. And you could write this yourself. I'm going to write a little procedure right here called Hanoi.

What are my arguments? Going to tell how big a stack there is. That's m. And I'm just going to give it labels, the from stack, the to stack, and the spare stack. Look how simple the code is. Says, if it's a stack of size 1, just move it. I'll just print out the instruction. Move it from the from stack to the target stack, or the to stack.

If it's bigger than 1, what do I do? I move a stack of size n minus 1 onto the spare stack. I move a stack of size 1, which is what's left, onto the target stack. And then, I move that stack over here that's on the spare stack over to the target stack. It's what I just showed you right there. OK? So let's try it.

Yes, I know I still like French. We're going to do Hanoi. Move a stack of size 1. And we'll just give this some labels, just from, to, and spare. Well, duh. You just move it there. All right. Let's try a stack of size 2.

It's just what I did. I'm sure you remember that. Let's be a little more daring here. There's the solution to move a stack of size 5. I'll let you check it separately, make sure it's right.

One of the things you can also see here-- I'm not going to talk about it. You might think about it, ask your TA in recitation is, how long does it take to solve this problem? How long is it going to take those monks to actually move a stack of size 64?

I'll give you a hint. The answer is measured in billions of years. This is an exponential problem. And you can see that growth right away.

That's a separate topic. But notice coming up with that solution on your own, maybe not so easy. Thinking about it recursively, very easy to think about. And that's the way we want to look at it.

OK, let me give you another example of breaking a problem down recursively, and then writing the code to do. I want to decide if a sentence is a palindrome. Remember what a palindrome is? It is not an ex-governor from Alaska. It is a string of characters-- I guess an airport in Alaska-- it's a string of characters that have the property that they read the same front to back.

The most famous one in English-- which is, of course, amusing because it's attributed to a Frenchman-- is Napoleon supposedly saying, "Able was I ere I saw

Elba." Same thing back to front.

How would I write a piece of code to decide if something is a palindrome? I'm going to do it in a second. You've got it on the handout. But let's think about it for a second. What would the base cases be-- or base case? Somebody help me out.

What's a good base case here? What's the shortest possible sentence I could have? I? A? A? Don't worry about whether it's a legal sense of not. It might need a verb in there. Base case is presumably, if I've got a string one character long, it's a palindrome. If I've got a string zero characters long, it's probably a palindrome as well.

How would I take a longer string and break it down into a simpler version of the same problem to decide if something is a palindrome? Anybody want to help me out? Is that a hand up there, or are you just scratching your head? Well-- yeah?

AUDIENCE: Maybe take the first and the last element, see if they're equal. If they are, then cut them off, and--

ERIC GRIMSON: Yeah. What's a palindrome? The easy way to start it is take the things at the end, first and last character. If they're not the same, it doesn't matter what's happening in the middle. This thing can't be a palindrome. So let's check those.

And oh, yeah, if those are the same character, and I pull them off, what do I have? I have a smaller version of the same problem. I have a new sentence that's now two characters less.

Do the same thing. Say, is that a palindrome? So if these characters match, and that's a palindrome, I'm done. How do I tell if that's a palindrome? Check if their two end characters match, and the things in the middle.

So let's look a little piece of code to make this happen. I'm going to do it in a couple of pieces. Here's the first piece I'm going to write. I'm going to walk you through it.

First thing I'm going to do is I'm going to do this outside of the recursive call is I need to convert a string that's put in to make sure that, in fact, it's in a form I want.

So I don't care about the spaces. Able was I ere I saw Elba. The spaces aren't in the same place. That's OK. It's really the characters.

And I don't really care about capitalization. So this little procedure basically says, given a string, convert it all into lowercase-- and if you haven't seen that, I'm just importing from a module called string some built-in procedures-- and this one simply takes the string, and makes it all lowercase. And then what do I do? Well, just like we did before, I'm just going to walk down that string, gathering together all of the characters.

So this little loop just says, let's initialize ans to be an empty string. And then for each character in s, if it is a lowercase character-- and that little test simply does that. It says, if it's in the set of lowercase characters I'm going to add it to the end. And when I'm done, I'm just going to return ans. Going to return the answer. A little procedure.

And, by the way, this is a nice piece of programming style, as well. I want to separate out things that I want to do once from the things I'm going to call multiple times. So I don't need to re-check every time that my string is all lowercase, I'm just going to convert it out.

Now, let's look at how do we test this. Well, it's literally just a translation of what we said. But let's look at the pieces. It says, if I give you just a string of characters-- I've gotten rid of the spaces, I've made it all lowercase-- what does it say to do? I've got to check for the base cases.

And here, I'm actually going to be careful. We could have discovered this if we programmed. There actually are two base cases here, which is, is the string of length one, or is of length zero? Why would I end up with two base cases? Why don't I just check for a string of length one? Yeah?

AUDIENCE: [INAUDIBLE]

ERIC GRIMSON: Exactly. I can have an odd or an even number of characters. So as I'm clipping them off the ends, I might end up with nothing in the middle, or I might have one in the middle exactly. And we might have discovered it if we tried programming it. But right, exactly right.

So I can capture that by just saying, if the length is less than or equal to 1, which gets both of those. In that case, we know the palindrome will return true. Otherwise, what do we do?

Well, we do what the gentleman over here suggested. We take the first and the last character of the string-- again, remind you s with an index of minus 1 goes backwards 1, if you like, and gives me the last character. If those two characters are the same, I've now reduced it to a simpler version of the same problem. So I simply say, if that's true, and everything else is a palindrome, return true.

Now, if you've not seen this particular little funky form right here, that is taking string s, and saying, give me a copy of everything starting with the first-- which means not the 0th element-- and ending with everything up to, but not including, the last element. We'll use this to make copies of other kinds of lists. But that's all that's doing, is saying, give me what's in that string, throwing away the first and the last element.

And that gives me exactly the recursive call I want. I'm now saying what? If the first and last character are the same, and if what's left over is itself a palindrome, I'm golden.

Now, let me just wrap all of that up in a little piece here. I'll un-comment this. Which is simply going to say, I'm going to print out, or put a comment in it. And I'm simply going to put the two pieces together. Given a string, I'm going to convert it into all lowercase characters. And then I'm going to check, is this thing a palindrome?

So again, let's try some examples. So we'll pick on Professor Guttag. His name is almost a palindrome. Not quite. So we're going to give him a name change. It helps if I can type. Oh, good.

Guttag is not, but Guttug, whatever that means in German, is, because the Gs, the

Us, and the Ts all match up. Oh, and let's see. Let's try a couple of other ones here. And I actually typed these in. We'll check to see if Napoleon really was right when he used his palindrome. If you can't read this last one, it says, "Are we not drawn onward, we few, drawn onward to--" and I can't read the tail end of that-- "to new era." And if we try both of those, they are.

What's my point, you're wondering. I solved this problem by simply breaking it down into simpler versions of the same problem. That's the tool that you want. If you were just trying to think about, how do I keep track of my indices as I'm walking along? I'm going to come in from both ends, so I've got to add and subtract. And I got to make sure I'm checking things.

You could write a nice iterative loop that would do it. Actually, I'll take back the word nice. You could write an iterative loop that would do it. But it is not crisp, it's not clean, and it's really easy to screw up. Or you could say, let's take advantage of recursion. Let's just break it down into a simpler version of the same problem. You'd get a very nice, simple piece of code.

Now, this may still feel a little mysterious. I wouldn't blame you if it did. So let's do the following. I'm going to comment these out so that we're not constantly looking at them. And I'm going to show you what happens if we actually look inside of this thing to see what's going on. So just give me a second here. We're going to comment all of those out.

And let's build a version of this that just prints out as we go along. So I'm going to show you right here. It's the same basic pieces. And actually, I realized I need to leave is characters around. Let me go find my is characters part-- two characters part. Sorry, give me a second here.

Going to need that. So we'll un-comment that region. That's just doing the conversion for us. I think this is going to work. Let's look at what we're doing here.

This is exactly the same thing. But I'm just going to put some print statements in it, which as I'm sure you've already heard from Professor Guttag, is a good thing to do as well. I want you to see what happens.

So the only changes I'm making here are when I come into the thing that's doing the checking, I'm going to print out a little thing that says, what are you calling me with, so you can see how it does the work. And then, if it's a base case, I'm going to print out a statement that says I'm in the base case. And otherwise, I'm just going to print out something that says, here's what I'm about to return for the piece of the string I'm looking at. So it's just instrumenting, if you like, what I'm going to do inside of here.

And the other piece I'm going to do is I'm going to have a little space called indent, so that every time I call a smaller version of the problem, I'm just going to indent over a little bit, so you can see how it unwraps it. And if we do this, just run that through. Let's try it.

So if I call is palindrome print with Guttag-- it would really help if I could type, wouldn't it? Yes, it would really help if I could type. You're all being really polite, going, ah, he missed a character there. But we're not going to tell him, because he's going to have to figure it out for himself. Administrators cannot possibly do this.

Ah, notice what it did. This is how you can see the recursive unwinding. It said-- I'm going to check this thing out. It said way up there-- actually, I'll just do it here.

It said, I'm going to call it initially with Guttag. Oh, the two Gs worked so that recursively-- and notice the push-in here-- said, I'm calling it again with U-T-T-A. Oh, that one didn't work. So I didn't have to check any further, and I pushed it out.

On the other hand, if I do this, ha, you can see all of the stages. You can see it unwinding it. It says, to decide if Guttug is a palindrome, well, I check the ends. And I've got to check U-T-T-U. I don't know what that is. Something interesting. Which says, I got to check that.

Which means I've got a check, oh, there's that base case of the empty one. I can now return true-- and I'm going to open this up so you can see it-- from the base case, true for that, true for that, true for that. Let's do one last one, assuming I can type. I can't.

Well, you get the idea, right? It's a little messy. But we go up a little bit, you can start seeing-- I'm obviously just going to run out of those pieces there-- but notice what it's doing. It's stripping off each of those characters in turn. You can see how deep this goes in, which is why you've got all those weird spaces there as I keep going in.

But it starts off by saying, look at that, oh, look at that. I've got to look at-- [GIBBERISH] Which says, I've got to look at [GIBBERISH] You get the idea. But notice the key thing. I'm simply reducing it to a simpler version of the same problem.

You can also see that to get the answer out, I've got to stack up a whole bunch of things. All those indents are basically held operations, which is one of the properties of a recursive procedure. You have to hold onto intermediate things.

And that can cause, in some languages, some efficiency issues. But in terms of solving the problem, this is really easy. I've just broken it down into those simple pieces. And that's really nice.

So let me finish up with one last example of a recursive procedure. Again, my goal here is to let you see, what am I doing? I keep repeating this, but it's important. To solve a problem, figure out how to break it down into a simpler version of the same problem.

One of the classic examples of recursion is Fibonacci. How many people here know the Fibonacci numbers? A few more, good.

For those of you who don't, here's the story. Probably heard the phrase, "They breed like rabbits." It's been used to describe a population that the speaker thinks is growing too quickly. Works with rabbits. It works if you put two pennies in a drawer. There's a whole bunch of ways in which this happens.

The history of this is actually very old. It goes back to 1202, when an Italian mathematician named Leonardo of Pisa, also known as Fibonacci, developed a formula that he thought would help him quantify the notion of how rapidly do rabbits breed. And here was his model. His model is not great, and we'll see that in a second. But here's his model.

You start with a newborn pair of rabbits, one male and one female. You put them in a pen. You assume that rabbits are able to mate at the age of one month. And you further assume that they have a one-month gestation period. So they can produce offspring at the end of a month.

Finally, let's suppose that these mythical rabbits never die and that the female always produces one new pair-- that is, a male and a female every month from its second month on. The question is, how many female rabbits will there be at the end of a year? Hm.

OK, let's see if we can figure this out. OK, at month 0, when it starts off, there's 1 female rabbit and 1 male rabbit. At the end of the first month, there's still one female rabbit, but she's now pregnant. So there's still 1.

At the end of the second month, what do we have? Well, that initial female is still there. And she has produced one pair. So there is a second female.

At the end of the third month, that initial female has again been pregnant and has produced. And these two are still there. So there are 3.

At the end of the fourth month, both of these females have now produced offspring. And those 3 females are now pregnant. There are 5.

At the end of the fifth month, those 3 females have produced offspring. Those 5 are now pregnant. There are 8.

And at the end of the sixth month, there are 13. What did I just describe? I just described a nice recursive relationship. That says, gee, there ought to be a base case. And there ought to be a recursive case.

And, in fact, in this case, we can describe this very nicely. The number of females at month end is the number of females there were two months earlier-- because they've all given birth, so that's that many new females-- plus the number of females there were at the previous month, who are now all in whelp.

Ah, that's a recursive relationship. If I wanted to figure out how many females there are at month end, what do I have to do? Well, I just have to say, what's the base case? Because there's the recursive case. And the base case is-- let me do it this way. If n is equal to 0 or 1, it's 1. And as a consequence, I should be able to figure out how quickly do rabbits breed, at least according to 12th century Italian mathematicians.

A couple of things to notice here, by the way. Sort of similar to what I saw in the towers of Hanoi problem, I'm going to have multiple recursive calls. To solve this problem of size n over here, I've got to solve two smaller problems, and then do the simple operation of adding them together.

That's OK. It's going to change the complexity of my algorithm, but it's perfectly fine to have multiple recursive calls. And as I saw in my case of the palindrome, I may have more than one base case. That's also OK, as long as I can ground this out.

All right, so let's just look at it. How would we write this? Let me get rid of that, comment that out. We can write a little procedure to compute Fibonacci. And there's the procedure. It's in your handout.

I've got some things up at the top here that are just making sure that I got the right kinds of arguments to pass in, which is a cleaner, crisp way of doing it. But what do I say? I say, if either x is 0 or x is 1, two base cases, just return the answer, which is 1. Otherwise, solve two-- right there-- smaller sub-problems. And what's the simple operation? Just add the two things together, and return it out.

And if I do this, we can check to see if we actually compute Fibonacci properly. All right, let's do a test fib of 0. It says, ah, fib of 0 is 1. To do a test fib of 1-- there's my other base case. Aha, it says, there they go.

And now, let's do something a little larger. There it is. As it gets each computation it's doing, it says, to get fib of 3, I've got to solve fib of 0, fib of 1, and fib of 2. And there's the answer. And just to make sure we're really doing this well, just reproduce my table over there.

Do I expect you to get all of that code right away? No, although it's pretty easy to understand. What I do expect you to see, though, is notice what I did. I took a recursive problem, and broke it down into simpler pieces. And then I used the solutions of those pieces to give me the solution to the larger problem.

By the way, Leonardo Pisa had it wrong. In 1859, as I'm sure many of you know, a wonderful Australian farmer named Thomas Austin imported 24 rabbits from England to use as targets for hunting. 10 years later there were 2 million rabbits being shot and trapped in Australia.

And fib, while it grows fast, doesn't go quite that fast. So Leonardo of Pisa had it wrong. Of course, as you probably also know, Australia, and I think New Zealand, have the wonderful property of having more sheep than people. I don't know what that says, other than it's a great recursive problem.

Fibonacci, by the way, actually shows up in other places. Again, it may not be a perfect model of rabbit populations, but it has some other really interesting properties. One of them is, if you let n get close to infinity, and you look at the ratio of fib of n over fib of n minus 1-- it sounds like a strange thing to look at. It basically tells you how quickly does fib grow.

Does anybody know the answer to this? The golden ratio, 1 plus root 5 over 2, which Leonardo-- the other Leonardo, the really good Leonardo-- by the way, used to design buildings with. So there's a very strange connection.

The other one thing I like about this is that Fibonacci shows up, actually, a lot in nature. So, for example, did you know that the number of petals on most flowers is a Fibonacci number?

For example, black-eyed Susans, 13 petals. Field daisies, 34 petals. You might ask why. Oh, sorry. I'm out of time. I won't answer it today. But you can ask Professor Guttag next time around why it is that petals come in Fibonacci numbers.

What's the message of this lecture? Recursion, divide and conquer. Break a problem down into simpler versions of the same problem. Life is really easy. And you'll see Professor Guttag next time.