PROFESSOR: So we're going to pick up where we left off last Friday on recursion. Well first of all, can anyone tell me what recursion is or what a recursive function is? No one knows. OK.

AUDIENCE: To divide and conquer.

PROFESSOR: OK. It's a divide-and-conquer technique. How does it do it? It's a recursive function.

AUDIENCE: You have a base case. And usually a return function, you return the value of something. But because you keep returning the function back.

PROFESSOR: Right. So it's a function that calls itself. And it works by one, identifying a base case, which is the smallest sub-problem possible. And then in the other case, the recursive case, it tries to chunk the problem down into a smaller sub-problem that it then solves by calling itself.

So we went over a few examples. And one of the things that we wanted to talk about is that for many problems, a recursive function can be written iteratively, or actually for most problems. So there usually is some sort of a subjective choice in how to write a function. And it really comes down to ease of understanding.

So what we've done is we've taken a couple of the algorithms that we showed last week. And we've written them iteratively. We also have the recursive version. We'll compare and contrast and see where we would want to use a recursive function.

So on the screen on the left here, we have the multiplication version, the recursive multiplication, that we showed you last week. It's a little different because it turns out there's a couple of simplifications you can make. Can someone walk me through how this works? So what's my base case first?

AUDIENCE: 0

PROFESSOR: 0, right? And so obviously when n is 0, if we multiply by 0, then our result is 0. Now there are two recursive cases. And I'm not really sure how to explain this intuitively.

But let's say that my n is positive. So I'm multiplying by a positive n. Well, then all I'm going to do is take m and just add it to the recursive version of itself n minus 1 times. That's how to read that.

And then analogously, if n is less than or equal to 1, then I'm going to take negative m and add the recursive result of n plus 1 times m. It is not too intuitive right? So if we implement it iteratively though, I think it's a little easier to understand. Now this is also a subjective judgment, so you might disagree. You're free to.

Here's our base case again. n is equal to 0 or m is equal to 0, return 0. In this case though, if we don't have the base case, then we're going to initialize the result variable. And then for n is greater than or equal to 1, we're just going to enter a while loop and keep adding m to result n times. It's a little bit easier to understand I think than the recursive version.

And then same thing for-- Oh I have a bug here. Same thing for n is less than equal to negative 1. So I'm going to run the two versions of this function. So here's the recursive version and here's the iterative function.

They both return the same exact thing. They both work in generally the same manner. It's just that in one case we're using recursion to solve it, which I don't find too intuitive. And the other case, we're using WHILE loops. All right. So in this case in my opinion, writing this iteratively, it makes a little bit more intuitive sense. But in other cases, let's say good old Fibonacci, we can write the recursive version and then the iterative version.

So here's recursive Fibonacci. We have our base case or cases. And then we have our recursive case. You can almost rewrite the mathematical formula from this directly. All right now. Now what was it?

So here's the iterative version of Fibonacci. We still have our base case. But when we get to what was the recursive case, we have to do a lot of bookkeeping. We have to save off the previous Fibonacci what we're currently computing. And then we have to iterate, get the next Fibonacci and then save off the prior versions of it. This is all stuff that in the recursive version gets done for us by virtue of just calling another function.

So this is an example of a case where your recursive version is actually a little bit easier to understand. Doesn't mean that it's more efficient. And later on in the class we'll actually use this to talk about complexity. But the left version I think is easier to understand than the right version.

Are there any disagreements? If you disagree, I'm not going to bite you. So anyway, we can run this. And we can see that the output's identical.

AUDIENCE: What's x-range?

PROFESSOR: x-range? Probably something I shouldn't have put in there. x-range is like range, except that it returns what's known as a generator object that you can iterate over. So that I don't have to explain that right now-- Well actually, we'll probably talk about it later in the semester.

The difference is efficiency. Range will return an entire list to you. Whereas x-range is a little bit more conservative in how it manages it's memory for these purposes. But changing it won't make a difference in the program. And for a program as simple as this, range is perfectly fine. I just used x-range out of habit.

So we'll do one last example. And then we'll move on to a different topic. If I didn't mention it before, in problem set 4, recursion is highly recommended for the final portion of it. So it's kind of important you understand what's going on.

Anyway, so remember we looked at bisection early on in the semester. And we showed you an iterative version of bisection. This shouldn't really be unfamiliar to anyone at this point.

So all this is doing is finding the square root of a number using bisection search. And we set our low and our high, get our midpoint. And we just keep looping until we get a value that when we square it is close enough to x. And on each iteration we set our lows and our highs, depending on how good our guess was.

Now the recursive version looks like this. It has a few more lines of code. And before I launch into it, did we explain default parameters to you, for functions? So Python has this feature where if you have a function such as rec bisection search, you can specify that certain parameters are optional or you can give default values to them. So let's just show a simple example so I can get past this.

So if I define this function, this one's really easy. All it's going to do is print out x. I can call it like this, in which case it's going to pass 150 in and x will be 150 when the function's executing. See I'm not lying. Or I can call it like this. And it'll be 100.

So that's, in a nutshell, what default parameters do for you. They're useful in some instances, as in this example. So in this recursive version of bisection square root, we have a low and a high parameter that we specify. It's exactly equivalent to the low and the high parameter in this iterative version. This is a common idiom for recursive functions in Python.

If we're calling it for the first time, we're not going to specify in a low and a high. So low and high will be none coming into this function. And then we just set them as we did in this iterative version. And then we set the midpoint.

And then we have slightly different structure here. If the midpoint that we guess is close enough to the square root of x, then we just return the midpoint. On the other hand, if it's too low of a guess, then we're going to recursively call ourselves with the same x, same epsilon, but we're going to use midpoint for the low parameter. So midpoint, in this case, is here and the same high parameter.

And then if we've guessed too high, then our low parameter is low. And then our high parameter's the midpoint. So it's doing the exact same thing as the iterative version. We have recursive, iterative, recursive, iterative. Same thing, just different forms. All right. Before I leave recursion, does anyone have any questions, or want to ask anything, or complain? No? All right.

AUDIENCE: Do you use a lot of recursion in your work? Do you normally use iterative or recursion? Or is it just case by case?

PROFESSOR: It's case by case. It depends on the problem. And what we are trying to show here is that there are some problems that are better expressed recursively and others that are better expressed iteratively.

And by better, it's a very subjective term. In my mind, it means more intuitive, easier to understand. It allows you to focus on solving the problem rather than fiddling with code. On the other hand, sometimes efficiency comes into play. And we're going to be talking about that pretty shortly.

And in that case, you might want to do a recursive version because it's easier to understand. But it takes too long to run, so you write an iterative version. Computer programming, in a lot of cases, actually in all cases, is a bunch of trade offs. Often times you'll trade off speed for memory, elegance for efficiency, that sort of thing. And part of the skill of becoming good computer programmers is figuring out where those balance points are. And it's something that I think comes only comes with experience.

All right, so we've talked about floating point to death. But we just want to really emphasize it. Because it's something that even for experienced programmers, still trips us up. So the thing that we want you to understand is that floating point is inexact. So you shouldn't compare for exact equality.

So looking at the code here, I have to find a variable 10/100, which is just 10 over 100, and 1/100, which is just 1 over 100, and then 9/100, which is 9 over 100. And so in real math, this condition would be true. I add 1/100 and 9/100. I should get 10/100. So if we were not dealing in computer land, this would print out.

But because we are dealing in computer-land, we get that. And the reason is because of Python's representation. Now when you write print x, if x is a float variable, Python does a little bit of nice formatting for you. It kind of saves you from its internal representation.

So here is 10/100, as you just printed out. It's what you would expect it to be. But this is what Python sees when it does its math. And it's not just Python. This applies for anything on a binary computer. It's an inherent limitation. And you know we can get arbitrarily close, but never exact.

And then again, we have 1/100 and 9/100, Python will show us 0.1. So when you print these out, they'll look fine. If you were writing debugging code and you were wondering why if you compared x to y, it wasn't exactly equal, you would naturally print out x and then print out y.

But it would look equal to you. But the code wouldn't be working properly. Well the reason is, is that the internal application that Python is using to compare them is that. So what's the solution? It's a natural question. I don't know the answer.

AUDIENCE: If they're close enough, then it would be inside variance--

PROFESSOR: Right. We're going to say good enough. And the traditional way of representing that is epsilon. Epsilon, you've seen in your problem sets. And you've seen it in code before. And if you've come to office hours, someone's probably explained it to you.

Epsilon is the amount of error we're willing to tolerate in our calculations. So in Python-land, you can have arbitrary precision. Don't quote me on that though. But for purposes of this class, if you're using an epsilon it's like 0.0001, we're not going to get too upset.

All this function does, and this is a handy function to keep around, is it just tests to see if the distance between x and y are less than epsilon. If it is, then we say they're close enough to each other to be considered equal. So I don't like the function named compare. I don't think it's intuitive. Close enough is probably better. But it's also going to break my code.

Uh oh. This is an actual bug. Line 203. What did I do to myself? I commented out my definition is what I did. All right.

So if we compare the two values, 10/100 and 1/100 plus 9/100 and we use our close enough, our compare function, then yeah it's within epsilon. Again, notice here that we're using a default parameter. So if we don't pass in something explicitly.

So I can say something like this. Let's make epsilon really tiny. So if I make epsilon really, really tiny, then it's going to say no. So how you determine epsilon really depends on your specific application.

If you're doing high precision mathematics, you're modeling faults on a bridge or something, probably I want to be pretty precise. Because if you have the wrong epsilon, then you might have cars falling of the bridge or the bridge collapsing. And it would just be a bad day. So are there any questions so far about floating point? No.

AUDIENCE: You normally go as close as the areas will let you.

PROFESSOR: What's that?

AUDIENCE: You can't get as close anymore. How can you make the error that small. Because it's not going to get that close.

PROFESSOR: Well yeah, that's what I mean. So there is a limit to how close you can get. And it depends on the language and it also depends on the hardware. There are, and this group is getting a little bit more technical than I want, but you can define pretty precisely the smallest value that epsilon can be.

In a language like C, its defined as the minimum difference between two floating point variables that's representable on the host machine's hardware. So yeah, there is a limit. There are some math packages though, and we'll be using something called NumPy later on the semester, that allow you to do pretty high precision mathematics. Keep that in the back of your mind. But yeah you're right. You do eventually hit a limit.

OK so the last thing that I want to cover on floating point is that even though it's inexact, it's consistent. So let's say I define a variable 9/100 plus 1/100. And it's exactly what it says, 9/100 plus 1/100. Now we know that this is not going to equal 10/100, right. We just demonstrated that ad nauseum. And also, yeah, still defined.

The question though is, if I subtract 1/100 from this variable that I've defined, this 9/100 plus 1/100, will 9/100 now be equal to 9/100 plus 1/100? So in other words, will this be true? And the answer is yes.

And the reason is that if I'm adding or subtracting, even though 1/100 we know is an inexact representation, it's still the same. And so when we do the subtraction, we're subtracting the same inexact value. So this appeared as a quiz question at one point. It probably won't this semester. But it's something to keep in mind.

So any questions on floating point? If you're a mathy type and want to, look up IEEE 754. And this will give you all the gory details about representation and mathematical operations on floating point. And if you don't, then don't worry about it. It's not required for the class. OK.

So the next topic we want to cover is pseudocode. So can someone take a stab at defining pseudocode for me?

AUDIENCE: From what I gathered, it's basically you're writing out what you're planning on doing in just normal English.

PROFESSOR: I wouldn't just say normal English. But it's an English of sorts. And a lot of the difficulty that programmers have with writing programs or new programs is that we don't naturally think in computer languages. You think in English. Or well, you think in a human language.

So what pseudocode allows us to do is to kind of be in the intermediate. We still want to develop a step by step process for solving a problem, but we want to be able to describe it in words and not variables and syntax. Sometimes what'll happen is programmers will get so wrapped around kind of getting the syntax right that they'll forget the problem that they're actually trying to solve.

So let's walk through an example. Let's talk about pseudocode for Hangman. And because you've all done this on the problem set, I don't have to explain the rules right. So what would be a good kind of English first step for Hangman?

AUDIENCE: You have to choose a word.

PROFESSOR: Right. So let's not be too specific. We'll just say, select random word. OK. Now what would be another good step, next step.

AUDIENCE: Display the amount of spaces maybe?

PROFESSOR: So display a masked version of the word.

AUDIENCE: Exactly. Hide the word but display it.

PROFESSOR: Hide the word but display it.

AUDIENCE: Well, display the amount of spaces. You probably want to state how many letters are in the word at some point.

PROFESSOR: Ah, that's a good point. Where should that go?

AUDIENCE: That should probably be before the display.

PROFESSOR: OK. So tell how many letters. All right. Now what would come after this?

AUDIENCE: After display?

PROFESSOR: Yeah.

AUDIENCE: See how many letters you have to choose from.

PROFESSOR: OK.

AUDIENCE: First time you don't.

PROFESSOR: At first time you don't. But the nice thing about pseudocode is that we can barf things onto paper and then rearrange them as we-- It's sort of like brainstorming in a sense. You're trying to derive the structure. And it's easier to do like this than to try and do it in code. But yeah, you're right. You don't have to.

AUDIENCE: You would ask the person to put a letter.

PROFESSOR: OK. For a letter. And then what would come after that?

AUDIENCE: So then you want to check if it's the --

PROFESSOR: Check if it's in the word. All right. And?

AUDIENCE: If it is--

PROFESSOR: If it is?

AUDIENCE: Add it to the word.

PROFESSOR: Add, let's say, to correct letters guess.

AUDIENCE: Yeah.

PROFESSOR: OK. And if it isn't?

AUDIENCE: If it isn't, reject it.

PROFESSOR: Let's say--

AUDIENCE: You want to remove it from the options.

PROFESSOR: So if it's not, then we're going to remove from options. So letters remaining. Probably want to tell the user they're wrong too.

AUDIENCE: And use up a turn.

PROFESSOR: What's that?

AUDIENCE: You want to use up a turn.

PROFESSOR: I'm sorry.

AUDIENCE: Then you use up a turn. If you had a set amount of turns.

PROFESSOR: OK, so we're actually going get to that in a second.

AUDIENCE: I sent last week. [INAUDIBLE] game.

PROFESSOR: I actually played all of your Hangman games. It was quite fun. So again, yeah you're right. So we have a number of guesses that are remaining. And the thing is that we know that the user has a certain number of terms.

So we're probably going to repeat a lot of this. So at some point, we probably want to have a WHILE. It'll just say, WHILE we have guesses left remaining. By the way, the reason why I program computers is because my handwriting is horrible. So WHILE we have guesses remaining, we're going to keep doing all this. All right?

And then we're going to remove-- But is this the only stopping criteria? What if they win? So WHILE they have guesses remaining and the word is not guessed.

So this is in essence your Hangman program. It's English language. It's not easy to read because that's my handwriting. But it's kind of easy to understand it at an intuitive level. And the reason we're talking about this is because we're going to get to some more complicated programs as we move through the semester.

And a good starting off point for a lot of you, when you're trying to do your problem sets, is instead of trying to jump right into the coding portion of it, to sit down with a piece of paper, index cards, or a whiteboard and kind of sketch out a high level view of the algorithm. So that we can see this in code form. So let's say that I want to write a function that tests a number to see if it's prime. First question is, what is a prime number?

AUDIENCE: One where the only factors are 1 and itself.

PROFESSOR: Right. So a number that is only divisible by itself and 1. Are even numbers prime? Can they ever be prime? Really? What about 2? Right. So 2 is one of our special cases. All right. So what would be maybe a good starting off point for pseudocode to test primality, knowing those facts.

AUDIENCE: [INAUDIBLE]

PROFESSOR: All right. Can I erase this or should I leave it up? Because I can go over there. It's not like I'm erasing any deep dark secrets. There's no magic here.

All right so test number, if, what, equal to, say what? 2? Yeah. Why not? And maybe 3. Now what do I do if it is? Are they prime? So I'm done right. So I'm going to return true. Now what do I do if the number given is not 2 or 3?

AUDIENCE: [INAUDIBLE]

PROFESSOR: You're talking about the module operator. Right. So we will use that. That will tell us whether or not an integer divides evenly into another integer or the remainder after an integer is divided into another integer.

Let me ask you another question. What is the maximum value of an integer divisor for a non-prime number? So for a composite number.

AUDIENCE: So itself.

PROFESSOR: What's that?

AUDIENCE: Number itself.

PROFESSOR: OK, excluding the number itself. Well let's say that I have n as the number I'm testing, square root of n, because I'm not going to have a factor that's larger than that. And I ask that because there's a loop involved. So how would I go about this systematically?

I'd probably start at, let's say 5. OK. And then test if n modula, let's say 5, is equal to 0. Now if n is evenly divisible by 5, then that must mean that n is composite, because 5 is a factor. So if it is then return false.

Now what if it isn't? So that means that n is not evenly divisible by 5. Does that mean that the number's automatically prime? So after 5, what would be a good number to test, to move to?

All right. 6? No. That wouldn't be it. Because if 6 is a factor, then obviously it's not. Whatever.

So we're going to move on to 7. So basically we're going to test all the odd numbers. And this is going to be the same as that. So this repetition indicates here that I probably need a loop.

So instead of doing this, I want to say x is equal to 5 while x is less than-- We're going to test if x evenly divides n. And if it does, return false. And if it doesn't, then we just increment x and repeat.

And what happens when x becomes greater than square root of n? Well the WHILE loop's going to stop. And that also means that if I've made it to that point, then I've not found any numbers between 5 and square root of n that will evenly divide n. So that means that n is prime.

So if I translate this into code, it would look something like this. Now I see. So first we're going to check if n is less than or equal to 3. If it's 2 or 3, then we'll return true. If it's not 2 or 3, then that means it's 1 or 0. So return false.

So we've got those cases. And then we're going to iterate-- or if n is greater than 3, we're going to iterate-- now why would you go from 2-- we're going to integrate through all the possible divisors and check for divisibility. And if we evenly divide it, return false. And if we make it through the loop, we'd return true.

AUDIENCE: Does that RETURN stop the loop?

PROFESSOR: Yes. Well think about what RETURN is doing. You're in this function, test primality, And as soon as Python sees return, that's telling Python to kick out of the function and return whatever is after the return statement.

So this false here, it says return false, that means that it doesn't matter where you are, it's just going to kick out of the innermost function or the function that encloses that return and return that value. Any questions? All right. So testing primality, 1 is false, 2 is true, 3 is true, 4 is false, and 5 is true.

So it looks like the program works. And if no one has any questions, I'm going to move on to the last major topic. Everyone's good on pseudocode? All right.

AUDIENCE: What would the main purpose of pseudocode is for yourself when you're writing a program or when you want to explain it to other people?

PROFESSOR: Both. So the question was, is writing pseudocode useful for just understanding a program yourself or for explaining it to other people? The answer is both. I don't know.

It's the difference between showing someone the derivative of the function and then explaining that what you're doing is finding a function that gives you the slope of a function at that point. So it's one is more intuitive for some people than the other. A mathematician would understand the former pretty quickly. An English major would understand the latter maybe.

So when I explain my research to people, I don't tell them that I mess around with Gaussian Mixture models and Hidden Markov models. I tell them that I'm trying to figure out how people mispronounce words when they speak foreign languages. A lot easier for people to digest.

With debugging, what are bugs?

AUDIENCE: Mistakes.

PROFESSOR: Mistakes. And if you see one bug, there are probably many more. So when you're debugging, your goal is not to move quickly. This is an instance where the maxim fast is slow and slow is fast comes into play. You want to be very deliberate and systematic when you're trying to debug code. You want to ask the question why your code is doing what it does.

And remember, the first recitation I said, that your computer's not going to do anything that you do not tell it to do. It's not something that people do naturally. If you watch some of the TAs and sometimes a student will say, how do you find the bug so quickly? Well it's because I've been programming for 18 years. Professor Guttag's been programming for longer than that. So a lot of it is experience. And it's just when we've debug our own programs and when we were learning to program, it was as painful for us as it was for you.

So that said, you want to start with asking, how could your code have produced the output that it did? Then you want to figure out some experiments that are repeatable and that you have an idea of what the output should be. So after you do that, then you want to test your code one by one on these different test cases and see what it does. And in order to see what it does, you can use a print statement.

So when you think you found a bug and you think you have a solution to your code, you want to make as few changes as possible at a time, it's because as you're making corrections, you can still introduce bugs. Let's see. So a useful way to do this is to use a test harness. So when we actually grade your problem sets, a lot of the time the TAs will put together a set of test cases for your code.

So one of the things is a lot of the times when you get one of the problems or when you look at the problems, it'll have some example input and output. But that doesn't necessarily mean that we only test on that. There's additional test cases that we use. And it's not to trip you up. It's because there's a lot of different variations. And it's also, if you read the specification, you follow the specification, then you'll be fine. Moving on.

So let's look at an example. I have a function here, is palindrome. You've seen this before, right? Yes? Yeah. OK. So it's supposed to return true if string s is a palindrome.

And so I've written this function. And I've also written a test harness. Now there's a lot more code in the test harness, but it's pretty simple code. When you're writing functions, you want to think of the type of input you could receive. And you want to think of, what are the kind of boundary cases?

So the extremes of input that you can get. We call these boundary cases, edge cases. For the is_palindrome function, it would be like the empty string would be one. Or just a single character. These are the kind of minimum we can have or we could think of. On the opposite end of the spectrum, theoretically we could have an infinitely long string. So we're not going to actually test for an infinitely long string.

Anyway, all we're going to do is in our test harness, we're just going to run the function on these inputs. And we know that an empty string should be true. We know that a single character string should be true. We know that if I have a string that's two characters long and they're the same character, that should be true. If they're two characters, then it should be false.

And what I'm going to do now is I'm looking at kind of expecting what we call expected input. So after I've hit my edge cases, I'm going to look at all the strings of an even length and make sure that the function works properly. And then I'm going to look at strings with an odd length. And then once I get to this point where I've tested a number of different lengths, and in this case, it's just 2 through 5 or 0 through 5, if you want to include the edge cases. Then I'm going to say, well it looks like all tests are pass.

And I think that this function works pretty good for anything we can expect it to encounter reasonably. So the way that you use test harnesses, is every time you make a change to your program, you want to run the test harness, because that'll catch any bugs you may have introduced. And so I'm going to finish up with this really quickly because I know my time's up.

So I got a bug. It's telling me that one of my test cases failed. So line 299, which is this test case. So what we can do is now that we know that it fails, we can say, maybe printout our input and see what we have. And instead of just running, I'm just going to run that one test case that failed. So obviously this should be true.

And what we're seeing is that on the first call to is_palindrome, s is abba. And then on the recursive call to it, we only get bba. That means that we've only chopped of the front character. So you see the bug? Well here. So there we go.