PROFESSOR: So this week the last final topic we were taught-- we're really going to talk about in class is dynamic programming, so who can tell me what the key idea in dynamic programming is?

AUDIENCE: A way of [INAUDIBLE]. Instead of optimization tools, you can use [UNINTELLIGIBLE]

PROFESSOR: Sort of. The key idea is that you don't want to repeat computations that you've already done before. So you want to find a way to only do everything once. So it's a form of laziness. There are two key attributes for a problem though that need to exist for it to be solvable with dynamic programming. Can someone tell me what those are?

AUDIENCE: It's optimal [UNINTELLIGIBLE] The local optimization has those.

PROFESSOR: There's overlapping sub-properties. [LAUGHTER] You had all the right words, just not in the right order. An optimal substructure. OK. Can you describe the first one, or any one?

AUDIENCE: You get to the end solution [UNINTELLIGIBLE]

PROFESSOR: So when they say overlapping sub-problems, it means that we can take the big problem and we can break it down to a slightly smaller instance of the same problem. And then we can use the solution to that smaller sub-problem in the solution to our bigger problem. And then optimal substructure is closely related. It's saying that if we get an optimal solution for one of those sub-problems, then we can use that optimal solution for the optimal solution of the big problem. Does that make sense to everyone?

AUDIENCE: [INAUDIBLE]

PROFESSOR: What's that? It's closely related. And so it's-- yeah, it's closely related but not exactly the same. So to start off, we're going to do kind of a function that we're all familiar with, is Fibonacci, right. We've seen this one a billion times before. And so it's pretty obvious where the overlapping sub-problems are.

The solution to f of N is f of N minus 1, and f of N minus 2, right? And so if I combine the solutions to these two instances of Fibonacci, I get the solution for the big instance of Fibonacci right? So on the screen, you have an example function. You've all seen this before. And so what I'm just going to do is I'm going to run this Fibonacci function from 0 to 30, or N equals 29 actually. And we're going to look at how many steps it takes to execute.

When we get to Fibonacci of 29, it takes about 1.6 million steps to compute the value. And if we take a look at a plot here-- so the y-axis is semi-log, the blue dots represent the actual number of steps that the function took, so the number of times that it had to perform the computation. And then the blue solid line represents 2 to the N. The red line is the golden ratio to the N, which is the tight bound for this version of Fibonacci. And then the green line is a plot of a quadratic function.

So nothing really surprising there. It's pretty inefficient, and we established that before. So we could make this more efficient with dynamic programming, right? Let's draw out the call tree for f of 5. Doing a lot of writing. OK.

So if you look at this, we see that in a lot of places, we're repeating computations. So we repeat f of 2 three times. We also repeat f of 3, this computation, twice. The idea behind dynamic programming is that instead of repeating these computations over and over again, we're just going to compute them once.

So if we look at the implementation of this recursive Fibonacci, we see that the first recursive call is going down this side of the tree. And so the upshot is that this-- all the values from f5, f4, f3, f2, f1, f0, they all get computed before any of these branches. So what we can do is instead of recomputing f2 and f2 here, we can just

compute it here. Save off this value somewhere in some sort of a brain, and do the same thing for f3, and f4, but that's not going to matter.

So when we get to the second recursive call, say when we're calling Fibonacci of 3, we compute f of 2, get f of 1, we come back up to Fibonacci of 4, right? The next recursive call is going to be to Fibonacci of 2. But since we've already computed it, and we've saved it off in a brain, we can just look up this value instead of doing all the computation again. So when you have a very small tree like this, right, it's not a huge benefit. But if you have like a lot, like f of 100, this is going to make a huge difference.

Is anyone lost? OK. So why don't we take a look--

So this is an implementation of Fib that has an extra parameter called memo. And memo is the brain I'm talking about. This is where we're going to stash those computed values so we don't have to compute them again. And when it initially comes in, if memo is none, that is, we haven't passed in a dictionary or if it's like the first call to this function, then it's going to set a key of 0 to 0 and key of 1 to 1. Those correspond to f of 0 and f of 1, right, actually. No one pointed that out? All right.

So, now it's going to-- after it does that initialization, it's going to come down to this if statement, and it's going to check for whether or not N is in memo. What this is actually asking is, have we seen this number before? When we've been computing this number, the big Fibonacci number, right? So if it's down here, and it's looking at f of 2, it's asking, have I seen Fibonacci of 2 when I've done this computation before? And if it's in this tree, then the answer is yes because it's seen it down here.

If it hasn't seen it though, it's got to do the work. So it's got to actually do the recursive calls. And this is when it travels down this left branch of the tree. Does that makes sense? And all we're going to do is store it off here. And then at the end, we just return whatever's in our memory.

So why don't we take a look at how this runs. So remember, this is 1.6 million with the old implementation, and now it's only 57. And in fact, it's linear and exactly 2 to

the N minus 1, or 2 times N minus 1. Everyone follow that? Any questions?

AUDIENCE: There's not too much we could do.

PROFESSOR: No. It saves a lot of work for you. Because as soon as it-- so let's say that it's computed f of 4, this entire sub-tree here. As soon as it sees f of 3, it's going to look in it's brain, and you've already seen f of 3, and it says, I've already seen this, so I don't need to do all this computation here. I don't need to do all these recursive calls. I just have to return whatever's in the dictionary. That's where your savings come in.

So let's take a look at another example. The point is, you can have some really huge savings if you write your code right. So let's look at a different problem. Let's say that I have a robot. And this robot is positioned on the grid. Let's say that it has N rows, and N columns, or M columns. Your robot is starting out here, and it wants to get here. Your robot though is very stupid, and it can go only down and to the right. The question is, how many unique paths are there from the top left square to the bottom right square, given those constraints?

If you try and do this analytically, you'll probably hurt yourself. The easier way to do it, or at least I think so, is to realize that in order to get to g, there's only two places that it can come from. It can come from here, it can come from here, right. So the total number of unique paths coming into g are the total number of unique paths coming into this guy, and the total number of unique paths coming into this guy. So there's your overlapping sub-problems, right, and also your optimal substructure. If I can figure out these numbers, then I can figure out this number, right.

So-- and then these guys, the same condition applies. If I know these two numbers, then I can figure out this number. If I know these two numbers, then I can figure out this number. So one implementation-- well, one other thing. If I get down to this case where I have a 1 by M grid, how many different ways are there to get from here to here? One, right? And then same thing for M by N right? So the first crack at this, here's a recursive function. All we're going to do is, if we only have 1 row or 1 column, we return 1. Otherwise, we're going to look for the number of robot paths in

an N minus 1 by M matrix and then the number of paths in an N by M minus 1 matrix.

So let's take a look. We're going to do this on a 14 by 14 grid. And this will take a few seconds. OK. So there's a lot. 10 million unique paths, and it took about 20 million steps to figure it out. So we're going to pull the same trick for this problem that we did for Fibonacci. We're going to memorize, or memoize, the different paths or the different solutions, except our key is going to be a little different. It's just going to be N and M.

So again, if we have-- well, again, if N and M is not memo, then we need to compute it. If either is 1, then we remember it as 1. And if they're both greater than 1, then we're going to look at the number of paths in N minus 1 by M, and N times M minus 1. And then we also know that the solutions are symmetric, right. So it's going to be the same for an N by M and an M by N. And just return the solution. So let's try this out.

We get the same answer, but it only takes 104 steps to do it. So it's a pretty huge savings there. So the other-- this is an example of a top-down dynamic programming solution. We looked at the big problem, and we broke it down into two smaller problems. We looked at those two smaller sub-problems, we broke them down into two smaller sub-problems a piece. We can also go from the bottom up. And we might want to do this for a reason that I'll show in a second. So I think I needed that.

This is going to go from the bottom up. So instead of starting back here and asking how many ways I can get from these two guys, it's going to start at the beginning and ask how many ways I can get here. One, right. And then here let's imagine that we only have a 2 by 2. It's going to look at, again, the number of ways to get to the square to the left and to the square to the top, and add it here. Add it here. So what we're doing here-- what we're doing in this version is we're growing a grid towards a solution.

Does that make sense? This just sets up a matrix. The top row is going to be

initialized to 1. First column is going to be 1. And then for everything else, we're just going to add the row above and the column to the left. And return whatever's down in this lower right corner. So it gets the same solution, just in a different direction.

So now you might want to do this because imagine if instead of 14, we had 1,400. So this should crash. Python has a maximum recursion depth. If you have too many recursive columns, it's going to tell you, I can't go that deep, and kick you out. So I have 1,400 rows and 1,400 columns, that's a lot of recursion. But if we do it iteratively, where we have no recursion, that's the number of unique paths, which is a pretty sizable number. And that's the number of columns we made, which for a 1,400 by 1,400 matrix is not too bad, right. All right. So that was an easy one.

So now we're going to go to little harder one. Everyone doing all right? OK. This is a counting change problem. The idea is, let's assume I have a currency with coins of a certain value. I'm not sure where I got the $0.27 from, but that's apparently the value for the coins in our currency. So the problem is-- let's say that I have a sum total-- the question is, how many different combinations of coins are there that equal total?

The way to break this down is-- so does everyone understand the problem? So like if I say I want to give change $0.41 to a customer, then it's like a quarter, a dime, a nickel, and a penny, right? We can think of the problem in two ways. Well, we can break the problem down into two sub-problems by first considering what the total is-- what the number of combos would be if I use the largest coin in my set. We're using at least one of the largest coins. And then the other is the number of combinations if not using the largest coin. OK?

So this turned, actually turns, into a nicer sub-problem, which is if I have total minus largest-- so in this case, minus $0.27, what's the number of combinations for this sub-problem? You follow? And then this guy can be formulated as the number-- so I still have total number of combinations. But then I take out the largest coin, so coins is now, instead of 1, instead of having $0.27 as the largest coin, now has $0.25 as the largest coin. Does that makes sense to people?

And so the solution here to this big problem is the solution to-- or is the sum of the solutions to this problem and In this problem. Make sense? All right.

Why don't we take a look at the first version of this problem or this implementation. So we have three base cases. So the first base case is-- well, first let me explain the function. So, total, that's the amount that we want. Coins-- it's the set of coins in our currency. And I only have $0.05, $0.10, $0.25, and $0.27, so, oh well.

The first thing we do is check to see if total is 0. That means that we're trying to figure out what combination of coins is equal to 0. And of course, there's only one combination of coins. There are no coins. So that actually is 1. This case-- this is if we're trying a particular coin that's a little too large to fit into our total. So we wind up going below 0. That means we can't use that coin for this particular total, right. So there are no combinations that work for this.

And then this last case is, if we're no longer using-- if we have no more coins, and we still have stuff that we need to make up, like we still have some value in total, then that means that this particular combination that we're trying out also doesn't work, so we return 0. So that's this case, where we've taken out everything, so our set of coins is the empty set, is nothing. Does that make sense?

So our first recursive call is looking for the number of ways without the last coin. That's this case. And all we're doing is we're passing in the total we got before, and all the coins except for the largest one, so the last one. So we're stricken one off for each recursive call. And then the second case here is if we do use at least one of the largest coins. In this case, we're just going to subtract that off the total. And we're going to pass in coins as is. Make sense? And then we just return the sum. OK. So not too bad.

Everyone follow that code? All right. So this is without doing any memoization, it's just using recursion, breaking it down into two sub-problems and figuring out a solution. But now we're going to use dynamic programming to optimize this a little bit. So in this case, we have our little memoization dictionary-- you notice a pattern here? Yeah?

AUDIENCE: Do you have an infinite amount to supply those?

PROFESSOR: Yes. So you're assuming that you have an infinite number of $0.27 cent pieces, $0.25 cent pieces, et cetera.

AUDIENCE: [UNINTELLIGIBLE] just add 1's [UNINTELLIGIBLE]

PROFESSOR: And it's possible to get below 0. So let's see. This is all the same, right? No surprises there. But then what we're going to do is we're going to memoize on the total that we're trying to find and the largest value-- largest currency piece that we have. And if it's not in our dictionary, then we're going to compute it. And then once we get the solution, we're going to memoize it and return it. So let's see how this runs.

It could've been by 4. That make sense to everyone? No questions? Wow.

So now let's try a different formulation. This one's a little tricky. Let's imagine that I have a grid. And down the rows I have numbers up to total. OK, starting from 0. Across the columns, I have my different currency pieces. So the first two implementations we're going top down, now we're going bottom up. The way you read this is, if I have a total of 0, this is a number and my largest currency piece is $0.05, or $0.10, or $0.25, or $0.27, it's the number of coin combinations I have that will equal this total, right.

So in this first row, that's my base case, right. And then down here, we know this is all going to be 0, because if I have no coin pieces-- So now, the bottom-up way is, we're going to fill in this table here. So we're just going to iterate through all the totals that are possible. And for each total, we're going to iterate through all the largest coin denominations. Right? And then we're just going to do two checks. We're going to look at the current total we're looking at, so let's say that we're on row one, so our total is 1. And we're going to subtract the largest denomination of currency that we're looking at.

We start out here. So we're going to subtract 5 from total of 1. And that's less than

0. And when it's less than 0, all we're going to do is carry the number of combinations from the current total to the next smallest largest denomination. So if we're at 5 and we know we're less than 0, that means we're going to carry this value over. And we're now going to do the same thing for all these values.

Now let's get to an interesting case. So now we're looking at a total of 5, OK? When our largest coin is $0.05 and we subtract it from a total of 5, that's not less than 0, right? So that means we're going to go into the second branch of f statement. So again, the same thing. We're going to look at the number of ways with the largest coin. And to do that, we're going to use our table. We're going to look at the current total, which is 5, minus the largest coin that we're looking at, which is $0.05.

So that's going to index us into this row because 5 minus 5 is 0. And then we're going to look at the current coin, which is $0.05. So that's going to be 1, right? Well, I skipped a step. So we're going to get this value, right, so this is the number of ways with the largest coin. And then we're going to look at the number of ways without the last coin, so if this is the largest coin that we're looking at, then without it, this is all we have. So that's going to be 0. So 0 plus 1, that's 1.

And then for 10-- so again, this becomes 0. So the next interesting one is going to be 10. Is everyone following so far what we're doing? This would be 0. This is going to be 1 plus 0. And this is going to be 1 plus 1. Making sense? Make an error? No, OK. And then this is going to be 0 Everyone get the idea? So we're just filling in this table. What's that?

AUDIENCE: [INAUDIBLE]

PROFESSOR: Here?

AUDIENCE: There's only one way to make 10 plus 5.

PROFESSOR: Because-- the smallest-- so if I had 1 here, like so I was using a one piece, then this would be 2. But because we only have 5, there's only one way to make 10. Two 5's. Make sense?

AUDIENCE: [UNINTELLIGIBLE] because you can either use one 10 or two 5's.

PROFESSOR: Yes, exactly.

AUDIENCE: So if we're using the table that as-- so you have $0.05 as the largest coin, so there's only one way of doing it using $0.05 as the largest coin? How come that 1 doesn't go all the way across the row? Because if you have $0.25 as the largest coin you could use $0.05's.

PROFESSOR: You know what, you're right. Made an error. Actually, that would be 0.

AUDIENCE: [INAUDIBLE]

PROFESSOR: What's that? Yeah, but-- what can I say? It can be confusing. To prove that this crazy thing works, let's run it. So let's expand this. Actually, I'm going to use this. So all three versions-- what, did I make a mistake?

AUDIENCE: No, it's just how you expanded the window.

PROFESSOR: Oh, you like that?

AUDIENCE: That's what we-- well, like Tracy's quote is always really, really small. But.

AUDIENCE: No worries.

PROFESSOR: It's a program called Divvy, if you're interested. I think it's like $10 on the Apps store or something, or maybe it's $5.00. It just allows you to do those little grid things, so if you have lots of windows and stuff-- and if you need to increase your font size, that's a little bit more involved with Python.

Anyway, so they all give the same output except for the number of steps that they take, so the initial one took 855, the one with the memoization took 209 steps, and the one without memoization but from bottom up took only 337 steps, so-- it's just-- it's three different ways of attacking the same problem. And objectively, the last two are better than the first one.

We might think of cases where we would want to use say the table-based method

over their recursion method. Like if we had like lots of combinations to explore or some kind of situation where you got into a really deep recursion that was too deep for Python to handle and we got kicked out like we did for the number of robot paths. So that would be kind of like a criteria for which algorithm you would choose over the other.