SRINI DEVADAS: Good morning, everyone. So we come to the end-- one last lecture and puzzle. Today, we're going to look at a little coin row game and talk about, obviously, an algorithm to solve the game, code the algorithm. And I'm going to introduce this notion of dynamic programming to you which is an algorithmic technique that's going to be useful in this game. Though it's not essential to solving it, it's going to be important in improving the efficiency of this first algorithm that we come up with.

So the game itself is very straightforward. You have coins in a row given to you. So each of these has different values. And you can think of all of them as being positive, but it doesn't really matter. And your job is to pick coins to maximize total value.

And obviously, that's not very interesting. You just pick all of them if they're all positive. And if some of them happen to be negative, you wouldn't pick those. So that's not particularly interesting. But it becomes much more interesting if you add the constraint that if you pick a coin, you can't pick the next one, the immediate next one. All right?

So if I pick 3, I can't pick 27. If I pick 14, I can't pick 3, but I could pick 27. So clearly, I can alternate. Because I could pick, skip, pick, skip, et cetera, but I just can't pick, pick. But I could go pick, skip, skip, pick if I wanted a value that corresponds to that fourth location.

So there's nothing that's stopping me from doing 14 and 4. I mean, the only thing that's stopping me from doing that is I drop this 27, which is a large value, and it seems like a bad idea. But on the other hand, you can think of a different situation where you wanted 27 and 15. And so maybe you want to skip 4 and 5, right?

So what is the maximum value for this particular puzzle that you can come up with? And what are the coins that you would pick in this case? Someone?

AUDIENCE: 56.

SRINI DEVADAS: 56. How did you get 56?

AUDIENCE: By using 14, 27, and 15.

SRINI DEVADAS: You went 14, 27. And then you went skip, skip. You didn't want to pick 5. You could have picked 5. You can't pick 4. But you decided that 15 was much better. So you ended up picking

15, right?

And if, in fact, you had something like 11 here-- so let me just write that first. So if you did that, it's 56. And you're picking 14, 27, and 15. So is that right? Is that 56? Yes, that is 56. Good.

But if I had something slightly different, someone else tell me what I should pick here. What's the maximum value and what should I pick in terms of coins? Yeah, go ahead, George.

AUDIENCE: The maximum value is 57 if you just take every other coin, starting 14.

SRINI DEVADAS: So if you go 14, 27-- that's 41. Now you want to pick 5, so that's 46. And then you can't pick 15, but you get to pick 11. And so you go 57.

Yeah, so this one was alternation, pretty much strict alternation. So, well, you guys are getting good at this. So I'll give you a hard one.

Is that enough or shall we go for more? All right, this was the final question, final exam. You got 10 minutes-- seconds-- 10 seconds. [LAUGHS]

So clearly, this is not something you want to do manually. It's going to take a while. There's a lot of combinations here, right? There's a lot of combinations here.

And if you do this, you're going to go-- well, you can do it in a greedy way. You can say, I'll just do alternation. And so you go 3 plus 17-- you guys are going to have to help me here because these numbers are going to get large-- but that's 20.

And then I can't pick 23, so I go 20 plus 11, which is 31. 31 plus 4, 35. 35 plus 17, 52. 52 plus 34-- check my math, please-- 86. 86 plus 18 is 104. 104 plus 12 is 116, is a possibility.

But at this point, you have no idea-- I know, so I have some idea whether 116 is the highest or not. And I'll tell you that it's not, OK? It's not the highest, you can do better.

And you can maybe go and see if you can twiddle things here a little bit. But remember that you, obviously, just don't want to drop something. But if you just drop something, you want to pick something up.

Because if you just drop something, you'd lower 116. And if you want to do better than 116, you absolutely have to pick something up. Especially when you're alternating, clearly, you could have done-- I mean, can you do a little bit better?

My last thing that I picked was 12. And because of that, you know what happened. So can I do a little bit better than 116? Yeah, by dropping 12 and grabbing 15, and I get a plus 3.

And you might think that 119 is the optimum. But it turns out that's not even the optimum. And so there's a lot of combinations.

And I just looked at something local here to try and improve things. But maybe I should-- you know, it's probably was good that I picked the 34, but I don't really know that. I mean, it's possible that if I hadn't picked the 34, that I could've picked the 23 and the 17, and that is 40, right? So maybe that was better.

So this is non-trivial. This is a very simple-- at least to describe-- puzzle,. But this is non-trivial in terms of finding the optimum solution as it gets

--longer, the row gets longer. The numbers being large is not really the issue. It's the number of numbers that's the issue, right? So how are we going to solve this? What can we do?

And let's not worry about efficiency. How could we solve this? And obviously, we've been talking about recursive algorithms and we've solved nQueens, which has lots of combinations. Sudoku, even more combinations.

There's a lot of combinations here. But honestly, the number of combinations is dwarfed by the number of combinations we've already handled in these other puzzles, right? So yeah, so someone wants to give me some sense of how this could be done? Go ahead, Fadi.

AUDIENCE: So if I want to choose numbers here, we need to know that at least there's going to be a gap between-- at least a gap of one number between any two consecutive numbers.

SRINI DEVADAS: Yeah, you want a gap of at least one.

AUDIENCE: So what we can do with that, we can iterate, for example, if we have seven numbers, we have at most four numbers to pick. Or, for example, if we have more numbers, we then have more options. If we can do it, we can carry the gap slice between each two consecutive numbers and iterate over that, and see what combination gives us the high stuff.

SRINI DEVADAS: Yeah, you're basically trying to create some sort of exhaustive search. Your algorithm was iterating over the gaps. And there's many gaps, by the way.

You have a gap of one at least, and another gap of one, another gap of one if I had the strict

alternation. And you have to vary one of the gaps to two. And that, obviously, impacts the numbers you picked even for-- the gap might stay the same for the next pair of numbers, but the numbers will be shifted in the coin row sequence. And it's fairly complicated, OK?

If you are willing to give up efficiency, and you have a set of n items, then how many different ways, how many different subsets of n items do I actually have? When you have n elements-- you might have learned this in set algebra. But if you haven't, we'll go over it here real quick.

But how many combinations, how many subsets do you have when you have a set of n elements? You have to count the empty set and the entire set, right? So if I, say, have a set of one element, then the empty set is a subset of the one element set.

And then, obviously, the one element set is a subset of itself, so that's two, right? And then if I had two elements, A and B, then it's empty, A, B, and AB. That's four. So you get a pattern here? So it's 2 to the n, right?

So I can generate the 2 to the n subsets that correspond to the n-element set. And there's actually library functions in Python, powerset, that lets you do this. But that's cheating for this class because you can't use libraries, OK? Real programmers don't use libraries.

That's actually not true at all. [LAUGHS] That is probably the most dishonest statement made in this course. Real programmers actually exploit libraries. But introductory programmers should learn programming by programming from scratch, right?

But you can generate the 2 to the n subsets, for example, by just-- I can show you code for this offline-- by running through and incrementing an integer to go from 0 to 2 raised to n. You know, 2 raised to n could be a million, which is not that bad. And then this you could encode as-- you turn it into binary.

So if n happens be 4, then this turns into 0000, 0001. And each of these is a distinct subset, right? This means none of the elements are in there. This means that last element is in there. This means that all four elements are in there. And somewhere in here, if you have something like this, the second one is in there and the fourth one is in there.

So you could, essentially, use iteration to generate all of the subsets. And you're not done yet. What do you have to do now that I've given you all the subsets? It's two, three lines of code to solve our problem.

Someone tell me what those two, three lines of code are from a standpoint of operations required. I'm not looking at each of the subsets. What do I need to do for a given subset? Yeah, go ahead, Ryan.

AUDIENCE: To iterate through the subset and find the sum of all the values.

SRINI DEVADAS: That's right. I could iterate through the subset and find the sum of second plus fourth. I do that for-- do I do that for all the subsets and pick the maximum? Or do I do that for a subset of the subsets and pick the maximum?

AUDIENCE: from subsets

SRINI DEVADAS: What happens with this one? This is picking every number. So this is obviously going to be bigger than all of the other ones.

So did I have to enumerate all of them? What is wrong with this picture here? What is wrong with what I just said?

AUDIENCE: violates the constraint.

SRINI DEVADAS: Someone other than Ganatra, what is wrong with this solution which picks all the numbers? It's on the board, the answer is on the board. Yeah, go ahead, Ryan.

AUDIENCE: It violates the constraint.

SRINI DEVADAS: Yeah, it violates the constraint. So not all of the subsets-- I mean, if all of the subsets work, then you don't have to enumerate all of the subsets. You would just pick the numbers, all the numbers, right?

So you have to now check that-- and there is some code, too. I lied. I guess, I haven't finished with my dishonesty. But it's not two, three lines of code, it's a little bit more than that-- to check that particular subset, whether it violates the constraints or not.

And so what is the check-- it's going to be four or five lines of code-- what is the simple check if I give you a bit string? What is the simple check that you need to perform in order to check the violation of the constraint in terms of 0's and 1's? Yeah, go ahead.

AUDIENCE: No two consecutive 1's.

SRINI DEVADAS: Yeah, you do not do consecutive 1's. If you ever see two or more consecutive 1's, you've got a problem, right? So that's easy to check. And this constraint is no two consecutive 1's.

So you can imagine coding this up. And if you coded that strategy up, it turns out that this particular thing, it's a reasonable strategy if n is small. And it turns out that it would tell you that you want to pick 15, 23, 4, interesting enough, 17, 34. So double skip over there because you would really want to get to 17.

This 34, and 18, and you want 15. That makes sense. You don't want 12. And so this ends up adding up to 126, which happens to be the optimal. So you can certainly do that.

You can, as I said, use the powerset to save yourself some trouble. But you still have to code the violation of constraints, right? We want to do substantially better than this. Because if n happens to be a 50, 2 raised to 50 is way too large a number. And we don't want to be enumerating all of these subsets.

In fact, there's a way of doing this, certainly, in polynomial time. And it's going to be something where we're going to be able to run this algorithm in linear time. We can make a pass through this list and get that 126 number blazingly fast using the notion of memoization, which is related to dynamic programming.

And so this is a powerful algorithmic technique that you'll see over and over, especially if you take classes like 006 and 046. That gives you exponential improvements in runtime. Rather than 2 raised to n, growing as 2 raised to n, you grow as n, which is a huge difference, obviously.

Now, usually, when we talk about dynamic programming, we start with a recursive strategy, not this enumerative subset strategy, to generate the different solutions. And one of the problems with this is you're generating a lot of solutions, a lot of incorrect, invalid subsets. Because 1111 is an invalid subset. And there's a lot of them that are invalid.

A better approach, which would still be exponential, but it is better in terms of numbers than 2 raised to n, is to only generate recursively the legal combinations and then pick the best legal combination. And the legal combination, of course, in this case, as we described, was no two consecutive 1's, right? Now, we've done recursion. We've looked at recursive code.

So the key thing when you write recursive code is, remember, you want to get subproblems

that look like the problem that you're solving, right? They look exactly like the problem you're solving. Because then you can invoke yourself as a procedure with different arguments. And we kind of went through this a couple of times.

So this problem is actually less complicated than the tiling problem. It is basically a row of numbers. And let's just say you have numbers a, b, c, d, e, f. Try to think of a recursive strategy that says, I want-- let's call it Coinrow. And Coinrow is going to have a through f as an argument. That's the list. I'm just using pseudocode here.

And I want Coinrow to do some computation. And it's going to call Coinrow maybe multiple times. But the key thing is, what are the arguments that you have inside of those recursive calls for Coinrow? And that's the biggest question.

The easier question to answer is, what happens when you have the base case? And you end up, essentially, saying, you've got one coin on the row. So you've got one coin problem, right? What happens then? What is the base case?

What do you do? You pick the coin, OK? Because if you had two of them, if you happened to choose the base case to be 2, in this case, e and f, what would you do in that case? Compare the two and pick the bigger one, right?

But you have to be a little bit careful here. Because when you make the recursive calls, obviously, you can't have-- would this work if you picked 8? If I did something like this-- I do 8 plus Coinrow. So inside here, I do a plus Coinrow b through f.

If I wrote this code as a plus Coinrow b through f, what would happen? What would happened if I had Coinrow, and then I said I'm going to return a plus Coinrow b through f, and then I have a base case that says that if I have a single-- I still need a base case. This is just the recursive step. I need a base case so this would complete.

And then I say that if I ever have an argument of length 1, I'm going to return that argument. So what would happen with this code? Any ideas? Go ahead, Kye.

AUDIENCE: Pick all the coins.

SRINI DEVADAS: It would pick all the coins. It would just totally pick all the coins. Because it would go a plus, and then it'd go b through f. The constraint hasn't been encoded in here, right? So we would just pick all the coins. So that doesn't make sense.

So I have to-- I have some choices here. And so this was a giveaway that this would not work. Because it's not doing any choices, it's just picking all the coins.

So what happens if I pick a? What does it mean? If I pick a, what does that mean? What does that mean to my value that I want to return, which eventually needs to be the maximum value? And what does it mean with respect to the subproblems?

So the first question is easier. What does it mean with respect to my value? I need to add it to the value, right?

The second question is, what happens to the-- if I pick a, what happens to the coin row problem that remains? What can I say about the coin row problem that remains that you're going to tell me about? If I picked a-- so this might be pick the first one. Pick first, in this case, which is a. We'll just put that in brackets.

So what do I need to send as an argument to Coinrow? As one possibility, what do I need to send as an argument to Coinrow? Someone who has an answer? That should use the constraint. What do I need to send as an argument to Coinrow? Yeah, go ahead, I saw Kye first. Go ahead.

AUDIENCE: c through f.

SRINI DEVADAS: c through f, exactly. I cannot pick b right? So I encode the constraint in the recursive call by saying that if I picked a then I encode this as c dot dot dot f, OK? I'll just say c dash f.

And let me go ahead and erase this to give myself a little bit more room. But I'm not done yet. So if I ended up picking a, and I go-- if I don't write any more code and I ended up picking up a plus the value that I get from Coinrow c through f, and if I don't write any more code, what does that correspond to in terms of an algorithm?

We've actually used that algorithm. What does that correspond to if I only make one recursive call, and I pick a, and I go c through f? What does that correspond to with respect to an algorithm? Someone who hasn't answered?

This is the your last chance, last lecture. Yeah, you're thinking about it? Go ahead.

AUDIENCE: It violates the constraint because c and d and e and f.

SRINI DEVADAS: So Styliani right? So Styliani says, basically, it's an alternation algorithm. You're going to end up picking a, and then c, and then e, and then f. You can't pick f because you had to skip f. So it's alternation.

We did this alternation algorithm. We knew it wasn't going to be optimal. So clearly, it's not a solution to our problem, algorithmically speaking, because it's only going to give you the alternating coins.

And that might work out, but it didn't work out in the case of our longer problem, right? We got-- what was it-- 116. So that doesn't work.

So what else do I do? What is the other case that corresponds do what I have here? I can pick the coin and? I could skip the coin. I could pick the coin and I could skip the coin.

Skip first, a. Aha, what now can I pass into the Coinrow argument? What can I pass into the Coinrow argument if I skip a? Go ahead, Ryan.

AUDIENCE: b through f.

SRINI DEVADAS: b through f, exactly. So it's different, b through f. That's it, that's our algorithm. This is now four lines of code in terms of recursive calls.

One last thing, I did these two things, I'm going to get values back for each of these. Because I'm going to get a value back. That's essentially what I'm doing.

When I'm picking, I'm going to add the value a. And then I'm going to add it to the return value corresponding to Coinrow c through f. And then, in this case, I'm going to get simply the value corresponding to b through f.

What do I do with val1 and val2? What do I do with these two values? Do I look at them? Yeah.

AUDIENCE: return them?

SRINI DEVADAS: What is the value for the original-- what is the value that I want to return for the original problem? Just like with all of the things we've done, we've taken values of subproblems, solutions to subproblems, and we've actually done something with them to return the value or solution for the original problem. So what do I need to do?

What line of code do I need to write to return a value for the original coin row problem that had a through f as its arguments? It's one line of code. What is that line of code? Go ahead, George.

AUDIENCE: Put it in the max.

SRINI DEVADAS: Return the max. That's exactly right. You return max val1, val2. And so let me show you the code for that.

So this is a little bit better than the subset solution that we described. But as it turns out, it's not a whole lot better, and we'll talk about that. But this code now should not be surprising to you because we talked about it and effectively, collectively we wrote it, right? That was a collective coding exercise called cooperative coding. Not really, but close enough.

And so you see what I have up there. And there's only one thing that I didn't describe to you, which is something that we're going to get to in a second. But you see something called table in there, which is essentially something that keeps the maximum value corresponding to the smaller problems that we are looking at. So it keeps that around.

But you can see that there's a base case if you forget table. And there's a reason that table is in here. And I'm going to ask you why in a minute. But forget about table for a second.

And if len is 0, you're going to return 0. That's a base case. If len is 1, you're going to return that coin. That's also another base case.

And you might end up with one or the other. Because, obviously, you're skipping coins, right? So it's possible that you need to take into account both of these base cases.

You say, why do I need two base cases? Well, you're taking away a coin. So you might end up with something that's empty, right? So you need both of those base cases.

And then pick equals coin row, table. This thing over here is picking the coin. And then 2 colon says that you're dropping-- in our case, we picked a, but we dropped b and started with c. And skip is you're skipping it. So you just skipped it and you go row 1 colon. Makes perfect sense, right?

And I don't know if you guys have ever returned-- I think you have in exercises-- if you returned multiple values in the return statement. And so this thing over here is-- the 0 says

that I'm going to look at the first value, not the table. Because that's a value, and I'm going to add two values up. And I'm only concerned with this value here.

But I'm actually returning this table thing. And this table thing, as you can see from here, is simply computing the optimum values that correspond to problems of the entire problem and, essentially, the problems that are smaller. And so then you end up-- you're skipping from the beginning.

And so when you look at the table len row minus 1, that is effectively-- since you're going from the beginning of the list-- that is telling you what the optimum is, len row minus 1. Sorry, len row minus 1-- yeah, that's right. The number of entries in the table is len raw plus 1.

The total number of entries in the table starts with 0 always, so it's len row plus 1. And so table len row is the entirety, that's what you want for the original thing. And len row minus 1 looks at an optimum for that, skipping the first one, and so on. So you're skipping from the beginning, so remember that.

Why do you think we have this table in here if we're only concerned with the maximum value? Or are we only concerned with the maximum value? What did I do when I got these numbers up?

What else did I do other than pointing out these numbers? I also gave you more information, right? I just said 126. So if I just told you it was 126, would you believe me?

AUDIENCE: It is hard to check.

SRINI DEVADAS: Well, it's really hard to check. It was credible because I said 126 and then I told you by circling them what the numbers were, right? So if you don't have this table stuff in here, it turns out that you don't have enough information to figure out what coins were picked.

You'll get the correct answer. It is guaranteed to be a correct answer. But you don't have the information about 15, 23, skipping 11 and 3, and getting 4, and so on and so forth.

So we need more code that takes this collection of subproblems that each have been solved. So we actually solved all of the subproblems corresponding to picking this, and picking that, and then picking this, and so on and so forth. And you need to use those values in order to discover, using a traceback procedure, what the coins are. Or you could do more work in here.

I chose to-- you'll see why-- I chose to do it this way where I'm collecting up everything into table. And I'm going to eventually-- and we'll talk about this code, at least briefly. But this code, traceback, is taking the original problem and a table and is just iteratively going through-- there's no enumeration here, this is very efficient-- is iteratively going through row and table and discovering what the selected coins are.

And so that's something that we'll look at in just a minute. Actually not just a minute, but after we go back to this. And I'm going to tell you or ask you about this in terms of its computational complexity, OK?

So anyway, assume that we've done traceback. And it's not hard to do if you have the table of results. Then let's go back to this and let's talk about the recursive calls. And let's take a look at what's going on here with respect to how many recursive calls are made. And then we'll get to memoization in dynamic programming.

So let me just say, if I had Coins that-- this is the number of elements-- so I have 5. Then I'm going to call Coins 4 and I'm going to call Coins 3. And so here, I skipped the first one. Here, I picked the first one, and therefore, I had to skip the second one, so I have 3 here, right? That makes sense?

And then over here, what do I need to write here if I follow exactly the pattern? I'm going to write Coins 3, right? So I have Coins 3 over here, and here, it's Coins 2. And here, I'm just going to write the numbers down-- 2, 1. And again, I've got 2, 1, and so on.

Now, the base case-- you might keep going even after 2 because the base cases are-- you go 1, 0, 1, 0. What do you see in here that's a little bothersome? What do you see here that's bothersome?

What is that, symbolically? If I had some number of elements, n elements, what do I put here? It's n, right? And what do you see here? It's getting pretty big.

So down here, it's getting pretty big. So that's the problem. And one of the reasons it's getting pretty big is because you're doing work over and over. You see, Coins 3 here couple of places, Coins 2 in three places. And if this were larger, you'd see a lot of subproblems being solved over and over.

So this code that you see up on the screen is incredibly inefficient because it's solving the

same problem over and over, OK? In particular, you would solve Coins 4 exactly once, but Coins 3 twice, Coins 2 three times.

And if I just went up and made this 10, Coins 10, then you would see things that are solved tens of times. And eventually, you'll see things that are solved hundreds of times. You're just repeating work over and over.

You can actually write a recurrence relationship that says that if you had an n-element coin row problem, then your recurrence corresponds to the number of calls that are made. Number of operations therefore that are performed is A n equals A n minus 1 plus A n minus 2. And then the base cases are A0 equals 1 and A1 equals 1, because you're just picking that.

And so if you look at what A2 would be, A2 is 2. But then A3 is 2 plus-- initially it grows fairly slowly. But then, when you get to A4, it becomes 5, and so on. And have you seen this number before, this recurrence relationship before?

It's called Fibonacci. It's called the Fibonacci recurrence, and this actually shows up here. So this has a relationship to Fibonacci. And the number of computations in a recursive Fibonacci is exactly the same as in this problem.

But if I told you to compute the Fibonacci number F of n, can you do that quickly? If I wanted for you just to compute the Fibonacci number, then you would just do that iteratively like I'm doing here, right? The numbers would get bigger, but there's no reason to do exponential work for Fibonacci.

But here, of course, we're doing something much more sophisticated than just Fibonacci, in the sense that we are working with a coin row problem. But this should give you a sense of the efficiency that is possible here by thinking about the relationship between recursive Fibonacci that would look a lot like this structure that you see here-- just calling Fib of n. Sorry, for Fib of n, you're calling Fib of n minus 1 and you're calling Fib of n minus 2 recursively, you're adding them up. And the base cases are exactly the same as you have here, returning for the 0 case, F 0, you're returning 1, and so on and so forth.

And even Fibonacci, you end up having redundancy in the recursive formulation, which is exactly the redundancy that you see here, OK? And so basically, it turns out that if you want to go exponential to linear-- and you kind of see where this is going. The recursive was exponential. The iterative, in the case of Fibonacci, is linear.

So two things-- one, you can take this and you can turn it into iteration and make it linear. But that is actually a more dramatic transformation of the code. You can do something that is equivalent in efficiency to the iterative version-- both for Fibonacci and for our coin row problem, and we'll do it for our coin row problem-- that corresponds to, as I said, the more local modification of this code that eliminates redundancy, that eliminates redundant computations.

And all of these things are equivalent and they're all part of dynamic programming. And we have four lectures on dynamic programming in 006. So in 10 minutes or 15 minutes, you're just going to get some sense for what this is all about. And so this is really a preview or an elevator pitch perhaps, a trailer, right?

So you end up, essentially, doing equivalent work in the recursive memoized version of this code that I'll show you in just a minute that adds three lines of code to this and makes the complexity go from exponential to linear. And all you do in this code that makes this complexity equivalent into the iterative and that's very efficient is you remember the results. You memoize the results of the computation.

And you remember that you've solved Coins 4, which doesn't really help you in this particular instance. But you remember that you've solved Coins 3, so you don't end up doing all of this work. You end up just saying, OK-- I'm sorry, it depends on which direction you went.

Let's say that, for argument's sake, you went in this direction first, OK? It totally depends on which direction you went first. And since I drew this out and I don't want to change it, let's just assume that you went in terms of Coins 3 first. Which, in fact, is correct in terms of the code because if you see what I have there, I went in the pick direction first, which meant that I would go ahead and pick that first value. And then I'd go in the right direction, the right-hand side direction.

So if I did all of that, I do all of this work, now there's also memoization going on inside of here. So it's not like I'm doing all of this work. I'm not only getting it just a factor of 2 in terms of improvement. But definitely, once I do this and I get the value, I can just return that value here and I don't have to do all the work underneath this. That's why I put this in a square, OK?

And remember that this is being done recursively. So I'm going to do it for Coins 2. And so if I had multiple Coins 2's, then I wouldn't have to do that. And then over here, if I had a longer

coin row problem to begin with, there's a lot of memoization that goes on.

And you end up only solving-- this is the key-- you end up only solving each subproblem how many times? Once. Now the aha moment. How many subproblems are there? How many coin row problems are there, given an n-element coin raw problem?

How many? All of you together. How many coin row problems are there? n, that's it. There's only n coin row problems.

It's like you had a through f, you had b through f, you had c through f, you had d through f, you had e, f, and f. So if you only solve coin row problem once, and there's only n coin row problems, the complexity clearly cannot be exponential, it's going to be linear. Because whatever work you did for that problem is all that you have to do.

And so maybe there's a constant factor. But let's say that the number of operations-- obviously, this is just the max. It's not complicated. We know what that is.

So it would be the complexity of solving a subproblem times the number of subproblems. The complexity of solving a subproblem is constant. It's just a small number of operations. And the number of subproblems is linear.

So it's linear time, which is exactly the same as the iterative version. This is an incredibly powerful notion, which is why there's classes on this topic, just this very topic. And we spend a lot of time on this in 006 as well as 046.

So what I'm going to do, the main thing I want to do is show you this code that is an incremental modification of the code I showed you before that takes this complexity and turns it from exponential to linear using this notion of memoization. And then I want to tell you a little bit about the traceback code because that's interesting. But from a standpoint of just getting the value back, you're all good without even understanding traceback.

So all I've done here is I've taken table and I've turned it into a dictionary, OK? And I had to do a little bit of work here with respect to the memoization. So if I look at the base cases, it's pretty much exactly the same. I just have a memo in a table. And all of these are essentially the same. I'm sorry, all of these up here are essentially the same.

And then I'm looking at a particular problem. I see whether the memo table that I have-- if len row in memo, then if it's in the dictionary, that means I've already seen this problem before.

And I just return the values stored in that dictionary. And the value stored in that dictionary is, essentially, a pair which corresponds to not just the value of the problem, but also I'm storing-- I'm sorry, I take that back.

So memo itself is just like table. This memo here is simply because overall I want to return both the value and the memo table. So this memo table is exactly like the original table.

It's simply a set of key-value pairs that go memo 5 equals 56, memo 4 maybe equals 42, et cetera. So that's just a simple memo table. And I just need to return the memo. And so that's why I had this memo out here.

And this looks exactly the same as it did before. These three lines are exactly the same. This is an important line. So these lines were added because I wanted to memoize and look up what I solved before.

And this line was added-- well, added in the sense that it was added for memo, but I already had it for table. So it's just a simple replacement, so perhaps I shouldn't have said added. But this is important because I'm putting it into the memo table.

And it's really this if statement that's the key. Don't do redundant work. So if you get to this if statement, you're obviously getting to a return, which means you're not making these recursive calls, which is exactly going over here and remembering that you solved Coins 3 and returning the value of Coins 3 the moment you see it.

All right? That makes sense? Everybody is good with that? Excellent.

Let's just run this. And I want to show you what this gives you just so you get some sense of what's going on. There's a couple of different problems and I'm running them with the different code. And so you're getting the same answers. That's just for verification purposes.

So if you focus in on what happens here, this is, essentially, solving our first smaller problem. And it's telling you that if you have the entire problem with the seven coins, which is what I have up there, the first problem here, then what you have is 56. And if, in fact, you ended up dropping-- it you're looking at the subproblem that goes from 3 to 1, and you drop 14, the optimum value for that is 42.

And then, if you drop both 14 and 3, and you went with that, then you need to go backwards from here-- 56, 42. It's still 42. So if you drop 14, it's 42. If you drop 3, it's still 42 because 27

and 15 give you your 42. And that makes sense. And so you can see that.

And then this other thing here is the bigger problem. It says, table equals-- sorry, yikes, Python shell. Let me just point with the cursor. This large thing here is our bigger problem.

And you can see the table for this-- 126. If you dropped the first one, you'd still get 126. Then if you drop the first two, 3 and 15, you get, obviously, a smaller result, and so on and so forth. And this table that I'm getting back, which is basically the original table that you could be thinking of as a list, that turned into a dictionary when I wanted to memoize because I wanted to look it up very efficiently.

I could have left it, in this case, as a Python list. I did not need to use a dictionary. I didn't need the generality of a dictionary where the indices to a list can be strings or negative numbers. I'm just indexing into this.

So the dictionary is a bit of a red herring. I just happened to use it. But in general, you use dictionaries because you may have to look up things that are more complicated than just integer values. But here, I could just number the subproblems 0 through 7, what have you, and then just look it up.

So the bottom line is I need that information in order to compute or figure out what coins were selected. And so the last thing that I will show you is the traceback routine, which I just put up there briefly. But I just want to spend one minute on this and we will close.

And this traceback routine is taking the values that you saw and figuring out whether you picked a coin or not. So what it does is it looks at the entire thing and it knows that the entire thing is 56. And then it says, if I drop the first one, and I get 42. And it compares 56 with 42 and says, wait a minute, the difference there is 14.

And that is exactly the value of the first element, right? So what this tells me is that if I had the original problem, I got 56. If I drop the first one, I get 42 as optimal. So that implies that for the original problem, I picked the first coin. Because 56 minus 42 is 14, that's it.

That's pretty much all you have to do. You just have to do that over and over. And it's a little bit more complicated than that only because you don't want to get into negative indices for tables.

And so the only reason that first line-- table len row minus i equals equals raw i-- is there is for the corner case where you have a single-element list. And you don't want to get into a

situation where-- this thing here cannot be negative. If this is negative, then the program crashes.

So you don't want to get to this statement. It becomes negative, and it's a degenerate case. And you don't want to get there. This or takes care of that. You essentially check for the singleton equality.

And then you don't fall through the second-- if this part of the statement is true, then this doesn't get computed. So if table len row i equals equals row i is true, then you don't compute that, you don't get a negative index, and everything works on. But it comes down to 56 minutes 42 is 14, and you're all good to go.

All right, good. So that's all I had, well, for this lecture, for the class. I'm going to be co-lecturing 6006 next semester. I notice that some of you were preregistered, so I guess it's not goodbye. But it doesn't have to be goodbye for the rest of you. You know what I said-- happy to talk algorithms and computer science. Thanks for being an attentive audience.

[APPLAUSE]