PROFESSOR: All right. Good morning, everyone. Thanks for coming on quiz day. I have to have someone to throw Frisbees to. Empty chairs would be difficult.

So we're going to be doing dynamic programming, a notion you've learned in 6006. We'll look at three different examples today. The first one is really at the level of 006, a cute little problem on finding the longest palindromic sequence inside of a longer sequence. Sometimes it's called longest palindromic subsequence.

And as we'll talk about, subsequence means that it can be non-contiguous. So you could skip letters in a sequence of letters and you would still have a subsequence corresponding to that. Don't have to be contiguous.

And then we'll raise the stakes a little bit. So each of these problems gets progressively more complicated, more sophisticated. And you'll probably see problems here, at least alternating coin game, that are beyond 006 in the sense that it wasn't covered. Those kinds of notions weren't covered in 006.

So just in terms of review, I wrote this up here because I don't want to spend a whole lot of time on it. This is something that you should have some familiarity with from the recitation, for example, that you had on Friday and from 006. Dynamic programming is this wonderful hammer. It's an algorithmic technique that you can use to solve problems that look exponential in complexity. But if you can find this optimum substructure associated with the problem and its connection to it's subproblems, and if you can characterize that, then you can do a recursive decomposition of the problem where you show that you can construct the optimum solution from the subproblems.

And that's really the key step in dynamic programming, is step two. Once you've made this characterization, you write this recurrence out that relates the optimal value of a bigger problem to the optimal values of subproblems. And you compute the value of the optimal solution through a recursive memoization. And that memoization is really what gives you the

efficient algorithm, because you don't repeat the solution of subproblems.

You can also do this in an iterative fashion. Essentially, you're going to be computing things bottom-up. You might want to think about it as top-down when you write your recurrence. But ultimately, when you actually execute the program, you'll be computing things bottom-up, and you'll be checking the memo table to see if you actually solved this problem before. And that would be the recursive memoized case, which, in some sense, is a little bit easier to think about and translate directly from the recurrence.

But the other way to do it is to do it iteratively. And we'll take a look for this first problem as to how you'd do the two different ways at least from a conceptual standpoint, even though I might not write out the code for each of those cases. So a couple of choices here, recurse and memoize, or essentially do it iteratively. And the smaller subproblems would have to get computed first in both approaches.

The one thing that sometimes we don't spend a whole lot of time on is this last step, which is getting the exact solution. So a lot of the time, you stop with saying, I can compute the value of the optimum solution, the value in terms of the length of the longest palindromic sequence is 7 or 9. But what is that sequence or subsequence? That requires some additional coding, some additional accounting. The construction of this optimal solution typically requires some back-tracing and some information to be kept track of during the recurse and memoize step or during the iterative step.

So that's something to keep in mind. You're not quite done once you've found the value of the optimum solution. More often than not, you want the solution. So we'll talk about that as well a little bit.

But let's just dive in and look at this cute, little problem of longest palindromic sequence. And palindromes, of course, read the same front to back or back to front. Radar is a palindrome. Just as a trivial example, a single letter is a palindrome. Maybe I should have used a since that's actually a word. But here I got bb. That's definitely not a word, at least not a word that-- acronym, maybe. Radar is a palindrome.

Able was I 'ere I saw Elba, right? That's a palindrome. That's, of course, not a single word. But it's a famous palindrome, days of Napoleon.

But what we're trying to do here is, given that we have this notion of a palindrome, we'd like to

discover palindromes inside longer words or longer sequences. So what we have is a string, and we'll call it x1 through n, n greater than or equal to 1. And we want to find the longest palindrome that is a subsequence.

And so here's an example to get you guys warmed up. We'll have a couple of puzzles here in a second. So character. And you want to find the longest palindrome. And so you go, I'll pick c, I'll skip h, I'll pick a, I'll pick r, a, and c. And carac, which I guess is not a word either-- but it's the longest palindrome that corresponds to a subsequence of character. Right?

So the game here, as you can see, is to pick the letters that form the palindrome and drop the ones that don't. OK? And we're going to have to use dynamic programming to do this.

The answer will be greater than or-- 1 in length because we've defined a single letter as a palindrome. So it has to be-- if you have one letter in the input, well, you just pick that letter. But regardless of how many letters you have on the input, greater than or equal to 1, you know that you're going to get at least a one letter palindrome at the output.

So here we go. Let's say I have under-- and this is thanks to Eric here. I've got a couple of nice words-- underqualified. The person who gets me the longest palindrome wins a Frisbee. And if you want to code dynamic programming in the next two minutes and run it on your laptops, that's perfectly fine with me. That's not cheating.

So underqualified. So u is a palindrome. So we got that. Right? So one letter for sure.

What else? What's the longest palindrome? Shout it out. Go ahead. Shout it out.

[STUDENTS RESPOND]

PROFESSOR: D-E-I-- wow, that was quick. Deified. That's right. So right? Well, deified is to make somebody a deity. So that was you, a couple of you guys? Do you have a Frisbee yet?

AUDIENCE: No.

PROFESSOR: All right. All right. This is a little bit more difficult. I tried this on my daughter yesterday, so I know it's a little more difficult. Turboventilator. We'll call that a word. Turboventilator. Yell it out.

AUDIENCE: Rotor.

PROFESSOR: Sorry?

AUDIENCE: Rotor.

PROFESSOR: Rotor. OK, well, that's five. Can anybody beat that?

AUDIENCE: Rotator.

PROFESSOR: Rotator. So rotator. Rotator. So R-O-T-A-T-O-R, rotator. All right. Who is the rotator? You already have one? I want to throw this one. Right.

Good practice with the quiz, guys. Good practice with the quiz. No, no, no. These quiz jokes never go over well. I've been teaching for 27 years, and I still haven't learned that you don't joke about exams. But so nothing like this on the quiz. I don't want you studying the thesaurus as opposed to the textbook for the next few hours. OK? Nothing like this on the quiz.

All right. Those of you who are missing Python, who loved 6006 in Python, let's talk about how you would actually solve this using dynamic programming. And so what we want is lij, which is the length of the longest palindromic subsequence for xij. And we're going to have i less than or equal to j.

So that's essentially what we'd like to compute. And essentially when I've said this here, when I have lij, I have decomposed the overall problem into subproblems, and it was kind of fairly obvious in this case, because I'm going to have to go in order. Right? I mean that's the constraint. A subsequence does maintain the ordering constraint. It's not like I can invert these letters. That would be a different problem if I allowed you to do that.

So I'm going to start somewhere, and I'm going to end somewhere. And I want to have a non-null subsequence, so I'm going to have i less than or equal to j. I'm good with i being equal to j, because I still have one letter, and, well, that happens to be a palindrome, and it'll have a length of 1. So that's my lij.

And what I want to do is define a recursive algorithm that computes lij. So we'll just try and figure out what the recurrence looks like, and then we can talk about memoization or iteration.

So if i equals equals j, then I'm going to return 1, because I know that that's a palindrome by default. So that's easy. And what do you think the next check should be? If I look at this x sequence, and I have i as the starting point and j as the next point, what do you think the next check is going to be once I have-- if i is not equal to j?

AUDIENCE: If x of i equals x of j, then j equals as well.

PROFESSOR: Sorry. What was that again?

AUDIENCE: If x of i equals x plus--

PROFESSOR: Beautiful. You're just checking to see-- you're just checking to see whether the two endpoints are equal or not. Because if they're equal, then you can essentially grab those letters and say that you're going to be looking at a smaller subsequence that is going to get you a palindrome. And you're going to be able to add these two letters that are equal on either side of the computed palindrome from the subsequence.

So if x of i equals equals x of j, then I'm going to say, if i plus 1 equals equals j, I'm going to go ahead and return 2, because at that point, I'm done. There's nothing else to do. Else I'm going to return 2 plus L of i plus 1 j minus 1. So I'm going to look inside. And I've got these two letters on either side that are equal. So I can always prepend to the palindrome I got from here the letter, and then append the same letter. And I got 2 plus whatever value I got from this quantity here.

So so far, it's not really particularly interesting from a standpoint of constructing the optimum solution. But this last line that we have here, where we have the case that the two letters are not equal is the most interesting line of code. That's the most interesting aspect of this algorithm. So does someone tell me what this line is going to be out here? Yeah, go ahead.

AUDIENCE: If x of L i plus 1j or L i [INAUDIBLE].

PROFESSOR: Beautiful. That's exactly right. So what you're going to do is you're going to say, I need to look at two different subproblems, and I need to evaluate both of these subproblems. And the first subproblem is I'm going to-- since these two letters are different, I'm going to have to drop one of them. And I'm going to look inside.

I'm going to say-- in this case, I'm going to drop the i-th letter, and I'm going to get L i plus 1j. And in the second case, I'm going to drop the j-th letter, and I'm going to get ij minus 1. And that's it.

So it's a max. And there's nothing that's being added here, because those two letters, one of them had to get dropped. They weren't equal, so one of them had to get dropped. So you're not adding anything to this. So that's good.

And at this point, you're kind of done. Right? Whoops. Oh, nice catch. But you did drop something. All right.

So the thing that we've done here is gotten to step three. So just to be clear, we're not done-done, in terms of this chart here, because we don't have the code there that corresponds to actually computing the sequence. So it's not that hard. I'm not going to go over the code here. You can certainly look at it in the notes.

But you need a little bit of tracing backwards in this recursion to actually compute things. What is the complexity of what I wrote up there, though? Yeah?

AUDIENCE: Theta n squared?

PROFESSOR: Theta n squared. Do people agree that the complexity is theta n squared? Or is this gentleman an optimist? Is the complexity theta n squared? Tell me why the complexity is theta n squared?

AUDIENCE: Because each subproblem is-- that code right there just executed by itself is constant. But then there's theta n squared subproblems.

PROFESSOR: Absolutely. But if you actually implemented this code, and you ran it, and n was 100, how long would you wait for this code to complete? Look at it. What's missing?

AUDIENCE: The cache.

PROFESSOR: The cache, exactly. Well, you fixed your own little error there. It was a trick question.

So there's no recursion-- I'm sorry, there's recursion here, but no memoization. So this is exponential complexity. You will recur. In fact, the recurrence for that is something like T of n equals 1 if n equals 1, and 2T n minus 1 if n greater than 1. And this would be 2 raised to n minus 1 in complexity.

Now there's a single line of code, and you all know this, that would fix this. And that single line of code is simply something that says, right here, look at the lij-- and I'm writing this differently. I'm calling this now a 2D array. So that's why I have the open brackets and close brackets. So I'm overloading l here. But it's a 2D array that is going to essentially be a cache for the subproblem solution values. And then if you want to do the backtracing to actually compute the solution, you could certainly have that as an additional record that's connected to this very

same value. But that's implementation, and we won't really go there.

So look at lij and don't recurse if lij already computed. OK. So that's important to remember.

Now, if you actually put that, the cache lookup, hash table lookup, array lookup, whatever you want to call it, out there, then what you said is exactly correct. So our formula for computing the complexity of a DP, that you've seen a bunch of times and I mentioned in the very first lecture as well, is number of subproblems times time to solve each subproblem assuming or given that smaller ones are solved or the lookup is Order 1. So lookup.

Now you could say that hash table lookup is Order 1 on average, et cetera, et cetera. So what actually happens in the worst case, in this particular case and in most DPs, you can do things like perfect caching or, something that's even simpler in this case, is just a 2D array. There's not going to be any collisions if you just use i and j as the indices to the array. So you will definitely get an Order 1 lookup in this case, in most problems we'll look at in 046.

So if we just do that computation, which my friend over here just described, you do get your theta n squared, because you have theta n squared subproblems. And time to solve each subproblem, given that the smaller ones are solved, is simply going to be a computation of a max and an addition. So all of that is theta 1, because you're not counting the recursive calls.

So this is our first example. I'm done with it. Any questions about it? Any questions about DP in general? All right, good.

So little bit of review there. Not a particularly complicated question. Let's go to a different question corresponding to optimal binary search trees. It's a very different question. I don't think I need this anymore.

And it's kind of cute in its own way. One of things that's interesting about this question is it seems like a greedy algorithm should work. And we'll talk about that, as to why the greedy algorithm doesn't quite work.

So it's kind of similar to the interval scheduling and the weighted interval scheduling problem that we had back in February, where the regular interval scheduling problem, greedy worked, earliest finish time worked. But when it came to weights, we had to graduate to dynamic programming.

So here's our second problem, optimal BSTs. So what is an optimal BST? We have a bunch of

keys that we want to store in the BST, K1 through Kn. And we'll assume that the way this is set up is that K1 is less than K2, da, da, da, less than Kn. And just to make our life easier in our examples, we just assume that Ki equals i. That's not necessarily required for anything we're going to do next. It's just for the sake of examples and keeping things manageable.

So I got a bunch of keys. And clearly there are many different binary search trees that can come, whether they're balanced or unbalanced. Many different binary search trees can be consistent with a given set of keys. If I choose the root to be Kn, then I'm going to get this horribly unbalanced tree. If I chose the root to be somewhere in the middle, then I'm going to get something that looks a little better, at least at the top level. But again, if I messed it up at the next level, I'd get something that's unbalanced.

So there's clearly many different BSTs. I'm not talking about balanced BSTs here. But we're going to define an optimality criterion that's a little bit different from balanced BSTs, because it's going to have this additional cost function associated with it that corresponds to the weight of the keys. So what is that?

Well, I'm going to have weights associated with each of these keys corresponding to W1 through Wn. And the easiest way to motivate you to think that these weights are an interesting addition to this problem is to think about these weights as being search probabilities. So what you have, for argument's sake, is a static structure that you've created. I mean, you could modify it. There's nothing that's stopping you from doing that. But let's just pretend for now that it's a static structure corresponding to this BST-- has a particular structure. And chances are you're going to be searching for some keys more frequently than others.

And the Wi's tell you what the probabilities are in terms of searching for a particular key Ki. So you can imagine, and we won't have this here, that you take-- the Wi's all sum up to 1, if you want to think of them as probabilities. Or I'm just going to give you numbers. I don't want to deal with fractions, don't particularly like fractions. So you can imagine that each probability corresponds to Wi divided by the sum of all the Wi's, if you want all of the probabilities to sum up to 1.

So think of them as search probabilities, because then you'll see what the point of this exercise is. And the point of this exercise is to find the BST T-- so we're actually constructing a binary search tree here. So it's a little more interesting than a subsequence, for example-- it has a richer structure associated with it-- than an exponential number of possible binary search trees

that are associated with a given set of n keys that are all binary search trees. They're consistent, unbalanced, balanced-- unbalanced in one way versus another way, et cetera, et cetera.

So we want to find a binary search tree T that minimizes sigma i equals 1 through n. Obviously, we're going to have this Wi that's the game here. Depth in that T-- so this depth is for that T. What is the depth of the node? And K of i plus 1. And I'll explain exactly what this and tell you what precisely the depth is. But roughly speaking, the depth of the root-- not roughly speaking. The depth of the root is 0. And the depth of 1 below the root is 1, and so on and so forth.

And so, as you can see, what you want to do is collect in-- roughly speaking, you want to collect the high weight nodes to have low depth. If Wi is high, you want the multiplicative factor corresponding to depth T of that node, i, i-th node, to be small. And if Wi is small, then you don't mind the depth number to be higher.

You only have a certain number of low-depth nodes. You only have one node of depth 0. And you have two nodes of depth 1, which means that you only have one node where that quantity there is going to be 1, because you're doing 0 plus 1. And you have two nodes where the quantity is going to be 2, and so on and so forth. So you have some room here to play with corresponding to the BST structure, and you want to minimize that quantity. Any questions so far? All right, good.

So the search probabilities would be an example. So in terms of a more concrete application, you could imagine you had a dictionary, English to French, French to English, what have you. And there are obviously words that are more common, let's say, in common speech than others, and you do want to do the translation in a dynamic way using this data structure. And you could imagine that in this case, the search probability of a word is associated with the occurrence of the word, the number of times over some normalized number of words that this particular word occurs. And that would be the way. So it makes sense to create a structure that minimizes that function, because it would minimize the expected search cost when you want to take this entire essay, for example, and convert it from English to French or vice versa.

So this can-- if these are search probabilities, then this would minimize-- this cost function here would minimize expected search cost. Make sense? Yeah.

So that's the definition of the problem. And now we have to talk about why this is complicated,

why this requires dynamic programming. Why can't we just do something fairly straightforward, like a greedy algorithm? And so let's look into that.

Let me give you a really good sense for what's going on here with respect to this cost function and maybe a little abstract by giving you a couple of concrete examples. First off, we got exponentially many trees, exponential in n. Let's say that n equals 2. So the number of nodes is 2. Then remember that I'm assuming that the Ki's are all i's, that 1 and 2 would be the case for n equals 2 and 1, 2, 3 for n equals 3, et cetera. So I could have a tree that looks like that. And I could have a tree that looks like this. That's it. n equals 2, I got 2 trees.

So in this case, my cost function is W1 plus 2W2, and in this case, my cost function is 2W1 plus W2. Just to be clear, what this is the K, and it's also the i. So the numbers that you see inside, those are the i numbers, which happen to be equal to the Ki numbers. And the weight itself would be the Wi. So I put W1 here, and the reason it only gets multiplied by 1 is because the depth here is 0, and I add 1 to it. The depth here is 1, and I add 1 to it, which is why I have a 2 out here.

Being a little pedantic here and pointing things out, because you're going to start seeing some equations that are a little more complicated than this, and I don't want you to get confused as to what the weight is and what the key number is and what the depth is. There's three things going on here.

So over here you see that I'm looking at W1 here, which is this key. So the 1 corresponds to the 1. And this has a depth of 1, so I put a 2 in here, and so on. So so far, so good?

When you get to n equals 3, you start getting-- well, at this point, you have 1, 2, 3, 4, 5-- you have five trees. And the trees look like this. I'll draw them really quickly just to get a sense of the variety here. But I won't write the equations down for all of them.

There you go. So those are the five binary trees associated with n equals 3, the binary search trees. And this is kind of cool. It's nice and balanced. The other ones aren't. And I'm not looking at--

Should I have another one here? I guess I should have. This is a-- oh, no, no, no. So I'm not doing mirrors. So there are a bunch of other trees that have the same equation associated with two-- no, that's not true. Because these are it. I was going to say that you could put 3 in here, but that wouldn't be a binary search tree.

So this is it. And we've got a bunch of trees. This one would be 2W1 plus W2 plus 2W3 by the same process that I used to show you that W1 plus 2W2 for the n equals 2 case. You just go off, and that's the equation.

And so your goal here in an algorithm-- clearly this is not the algorithm you want to enumerate all the exponentially many trees, compute the equations for each of those trees, and pick the minimum. I mean, that would work, but it would take exponential time. But that's what you have to do now. That's the goal. You absolutely want the best possible three that has, for the given Wi's that you're going to be assigned as constants, you do want to find the one tree that has the minimum sum, and you want to do that for arbitrary n, and you want to do that in polynomial time.

So the first thing that you do when you have something like this is forgetting about the fact that we're in a dynamic programming lecture or a dynamic programming module of this class, when you see a problem like this in the real world, you want to think about whether a greedy algorithm would work or not. And you don't want to go off and build this dynamic program solution, which is, chances are, going to be more inefficient then a greedy solution, which, if it produces the optimum answer, is the best way to go.

So an obvious greedy solution would be to pick K of r in some greedy fashion. And what is the obvious way of picking K of r to try and get a greedy solution that at least attempts to minimize that cost function that we have there?

AUDIENCE: Highest weight.

PROFESSOR: Highest weight, right? So just pick K of r to be highest weight. Because that goes back to what I said about if Wi is high, you want this value here to be small. So that's essentially your greedy heuristic. So K of r should be picked in a greedy fashion.

So what you do is you pick K of r as the root in this particular problem. And you could certainly apply this greedy technique recursively. So you do that for every subproblem that you find because when you do this choice of the root for Kr in the current problem that you have, you immediately split the keys into two sets, the sets that have to go on the left-hand side of Kr and the sets that have to go on the right-hand side of Kr.

So in this case, you know that you're going to have a bunch of keys here that correspond to Ki to K4, and over here, you're going to have-- oh, Kr minus 1, I should say, because you already

have Kr out there. And the keys here are going to be Kr plus 1 to Kj, if overall I'm going to say that eij is the cost of the optimal BST on Ki, Ki plus 1, all the way to Kj.

So I'm kind of already setting myself up here for dynamic programming. But it also makes sense in the case of a greedy algorithm, where this greedy heuristic is going to be applied recursively. So initially, you're going to have E(1, n)-- so if you want to compute E(1, n), which is the cost of the optimal BST on the original problem, all the way from 1 to n, you're going to look at it, and you're going to say, I have a bunch of keys here of different weights that correspond to K1 through Kn. I'm going to pick the Kr that corresponds to the maximum weight, and I'm going to use that as the root node, and I'm going to stick that up there. And the reason I did that is because I believe that this greedy heuristic is going to work, where this maximum weight node should have the absolute minimum depth.

So I stick that up there. And then my BST invariant tells me that Ki through Kr minus 1-- remember, these are sorted, and they go in increasing order, as I have up here. You're going to have those on the left, and you're going to have those on the right. And then you've got to go solve this problem. And you could certainly apply the greedy heuristic to solve this problem.

You could go essentially find the K that has the highest weight. So you look at Wi through Wr minus 1, and you find the K that has the highest weight. Actually, you're not going midway here. It's quite possible that Kr is K of n. It just happens to be Kr is K of n. So that means that you have the highest weight node up here, but it's also the biggest key. So all of the nodes are going to be on the left. So this greedy heuristic-- and now you're starting to see, maybe there's a problem here. Because if you have a situation where the highest weight node is also the largest node, you're going to get a pretty unbalanced tree.

So I can tell you that greedy doesn't work. And when I gave this lecture, I think it was a couple of years ago, I made that statement, and this annoying student asked me for an example. And I couldn't come up with one. And so I kind of bailed on it, went on, completely dissatisfied, of course. And by the end of the lecture, the same student came up with a counter-example. So I'm going to have put that up here and thank the student who sent me an email about it.

And so here's a concrete example. It turns out I was trying to get a tree node example for a few minutes and failed. It's, I think, impossible-- I haven't proved this-- to find a tree node example with arbitrary weights for which the greedy algorithm fails. But four nodes, it fails. So that's the good news.

So here's the example, thanks to Nick Davis. And it looks like this.

So I claim that the greedy algorithm for the problem that is given to me would produce this BST. And the reason for that is simple. The highest weight node happens to be node 2, which has a weight of 10. So I would pick that, and I would stick that up here, which means, of course, that I'm going to have to have 1 on this side, and I'm going to have to have 4 and 3 on the other side. And since 4 has a higher weight than 3, I'd pick that first. And then 3 would have to go over here.

So if I go do the math, the cost is going to be 1 times 2-- and I'll explain what these numbers are. I did tell you it was going to get a little cluttered here with lots of numbers. But if you keep that equation in mind, then this should all work out.

So what do I have here? I'm just computing-- this is W1. So you see the first numbers in each of these are the weights. And so this would be W2 and et cetera. And you can see that this is at depth 1, which means I have to have 2 here, because I'm adding 1 to the depth. So I have 1 times 2, 10 times 1, because that's at the root, 9 times 2, et cetera. And I get 54.

It turns out the optimum tree-- you can do better than that if you pick 3, and you go like this. And so what you have here is the cost equals 1 times 3 plus 10 times 2 plus 8 times 1 plus 9 times 2, and that's 49. So I'll let you look at that.

The bottom line is because of the way the weights are set up here, you really want to use the top three spots, which have that the minimum depth, for the highest weight nodes. And so you can see that I could make these weights arbitrary, obviously, and I could break a greedy algorithm as this little example shows. So we got no recurs here. We got to do some more work. We're going to have to use DP to solve this problem. The good news is DP does solve this problem.

So let's take a look at the decomposition. And as with anything else, the key is the step that corresponds to breaking up the original problem into two parts or more that essentially give you this sort of decomposition. And we don't quite know. So this is really the key question here. We don't quite know what the root node is going to be.

So what do we do when we don't know what to do? What do we do in DP when we don't know what to do? This is a very profound question. What do you do?

AUDIENCE: Guess.

PROFESSOR: You guess. And not only do yo guess, you guess all outcomes. You say, is it going to come up heads? Is it going to come up tails? I'll guess both. You have a die. It's going to get rolled, a 1, 2, 3, 4, 5, 6. Guess them all, right?

So if you're going to have to guess that the root node could be any of the keys, and it still-- I mean there's a linear number of guesses there. That's the good news. It's going to stay polynomial time.

So we're going to have to guess. And once you do that guess, it turns out that decomposition that you see up there is exactly what we want. It's just that the greedy algorithm didn't bother with all of these different guesses. And the DP is different from the greedy algorithm because it's going to do each of those guesses, and it's going to pick the best one that comes out of all of those guesses. So it's really not that different from greedy. So we'll keep that up there, leave that here. Let's go over here.

So the recursion here is not going to be that hard, once you have gotten the insight that you just have to go make a linear number of guesses corresponding to the root node for your particular subproblem, and obviously this happens recursively.

So there's a little something here that that I'll point out with respect to writing these equations out. So what I have here, just to be clear, is what I wrote up there. Even when I was talking about the greedy algorithm, I had the subproblems defined.

So the original problem is E1 through n. The subproblems correspond to eij. And given a subproblem, once I make a root choice, I get two subproblems that result from the root choice. And each of those two subproblems is going to be something-- and this is something to keep in mind as we write these equations out-- they're going to be one level below the root. So keep in mind that the original eij subproblem corresponded to this level, but that Ei r minus 1 problem and the E r plus 1j problem are at one level below. So keep that in mind as we write these equations up.

And so what I want to do here is just write the recurrence. And after that, things become fairly mechanical. The fun is in the recurrence.

So I got Wi if i equals j. And so I'm at this level. And if I just have one node left, then at that level, I'm going to pay the weight associated with that node, and I'm going to have to do a

certain amount of multiplication here with respect to the depth. But I'm just talking about eij as the subproblem weight, just focusing in on that problem. So if I only have one node, it's the root node. And that weight is going to be Wi, because the root node has depth of 0, and I'm going to add 1 to it. So for that subproblem, I got Wi. Keep that in mind, because that's the only subtlety here with respect to these equations, making sure that our actual cost function that we're computing has the correct depth multiplicands associated with it.

So then we have our linear guessing. And I might have said max at some point. We want to get the min here. And I had minimize here, so I think I wrote things down correct, but at some point, I think I might have said we want to maximize cost. But we do want to minimize cost here corresponding to the expected search.

So we're doing a linear, and we're doing a min. And we're going to go off and look at each of the different nodes as being the root, just like we discussed. And what I'm going to do here is I'm going to simply say, first, that I'm going to look at Ei r minus 1, which corresponds to this, and I'm going to have plus E of r plus 1, j.

And at this point, I don't quite have what I want because I haven't actually taken into account the fact that the depth of the Ei r minus 1 and the r plus 1j is 1 more than the eij. So it's 1 more than that. And I also haven't taken into account the fact that, in this case, I definitely need to add a Wi as well.

Because the root node is part of my solution in both cases. In one case, it ended my solution, the top case. But in this case, it's also the root node, as you can see over on the left, and I've got to add that in there. So I definitely need to add a Wi here. And what else do I need to add? From a standpoint of the weights, what other weights do I need to add to this line? Yeah? Go ahead.

AUDIENCE: First of all, shouldn't that be a Wr?

PROFESSOR: First of all, that should be a Wr, you're exactly correct.

AUDIENCE: And you want to add all the weights.

PROFESSOR: And you want to add all the weights, right. You're exactly right. I have two more Frisbees, but I need to use them for something else. So you get one next time. Or do I have more than that? No, no. These are precious Frisbees here. Sorry, man.

And you corrected me, too. Shoot. This is sad.

So that needed to be a Wr. But you also need to add up all of the nodes that are in here, because they're one more level. And now you see why I made the mistake. I don't usually make mistakes. But what I really want-- actually it's more like-- I don't know, a few per lecture. Constant order one mistakes.

I'm going to say this is Wi, j, where Wi, j is simply the sum of all of the weights from i to j. And this makes perfect sense, because the nice thing is that I don't even need to put an r in there. I could choose a particular r. Wr will be in there. But all of the other nodes are going to be in there anyway. So it doesn't really matter what the r selection is corresponding to this term here. I will put it inside the minimization. This bracket here is closed with that, but you can pull that out, because that doesn't depend on r. It's just going to be there for all of the cases, all of the guesses.

So that's it. That's our recurrence relationship corresponding to the DP for this particular problem. You can go figure out what the complexity is. I have other things to do, so we'll move on. But you can do the same things, and these are fairly mechanical at this point to go write code for it, trace the solution to get the optimum binary search tree, yadda, yadda, yadda. Any questions about this equation or anything else? Do people get that?

Yeah, go ahead.

AUDIENCE: What is the depth input?

PROFESSOR: So the depth is getting added by the weights. So basically what's happening is that as I go deeper into the recursion, I'm adding the weights and potentially multiple times, depending on the depth. So if you really think about it, this Wij, I added all of these weights. But when you go into this recursion, into the ei of r minus 1, well, you're going to see i to r minus 1 in the next level of recursion. So you would have added the weight again.

So it's not the case that the weights are only appearing once. They're, in fact, appearing this many times. So that's what kind of cute about this, the way we wrote it. Any other questions? All right, good. So we're done with that.

So one last example. This is, again, a little bit different from the examples of DP we've looked at up until now, because it's a game. And you have an opponent, and you have to figure out

what the opponent is going to do and try to win. I suppose you could try to lose as well. But let's assume here, same thing with minimization and maximization, most of the time you can invert these cost functions, and DP will still work. But let's assume here that you want to win this game.

So the game is an alternating coins game where we have a row of n coins of values V1 through Vn. These are not necessarily in any particular order. And n is even. And the goal here is select the outer coins. So select either the first our last coin from the row, and then the opponent plays. Remove permanently, but add it to your value and receive the value.

So I need two volunteers to play this game. And you want to maximize the value. The winner gets a blue Frisbee. The loser gets a purple Frisbee, because blue is greater than purple. And you might ask, why is that.

Well, if you went to a beach, and you saw water this color, and you went to another beach, and you saw water this color, which beach would you pick? Right? This is Cozumel. This is Boston Harbor. So blue is greater than purple. Don't use this proof technique in the quiz.

So do I have a couple of volunteers to play this game? Two of them over there. Are you waving for him? Yeah. I see one. You can come down. Another volunteer? Yeah. Over there. You don't get your Frisbees yet.

So we're going to make this really fair. What's your name?

AUDIENCE: Josiah.

PROFESSOR: Josiah.

AUDIENCE: Tessa.

PROFESSOR: Tessa. Josiah and Tessa, I'm going to write out a bunch of-- it's going to be a fairly short game. And we're going to be really fair, and we're going to flip a coin to decide whether Josiah goes first-- you can pick heads or tails-- or whether Tessa goes first. Actually if you win, you can let her go first if you want. But you get to choose.

AUDIENCE: All right.

PROFESSOR: So pick.

AUDIENCE: Heads.

PROFESSOR: That's heads. Do you want to go first, or do you want to let Tessa go first?

[LAUGHTER]

AUDIENCE: You should go first.

PROFESSOR: All right. So Tessa, you get to go first.

AUDIENCE: OK, 6.

PROFESSOR: 6, OK. So let's just say T. So you get to choose either 25 or 4, Josiah.

AUDIENCE: I think I'd pick 25.

PROFESSOR: You think you'll take 25. So that's j. So it's not on the 4 and 19 over here because those are gone. So your turn again.

AUDIENCE: OK. Can I take a minute, or should I just go?

PROFESSOR: Well, you can take 30 seconds.

AUDIENCE: 19.

PROFESSOR: 19, OK.

AUDIENCE: 4.

[LAUGHTER]

AUDIENCE: All right, 4.

PROFESSOR: 4. All right. And 42?

AUDIENCE: Yeah.

PROFESSOR: All right, 42. This is one strange game. Now, we get to add up the numbers. This is going to be tight. 4 plus 39 is 43. 43 plus 25 is 68. 42 plus 19 is 61. 61 plus 6 is 67. Ooh. Well, you get Frisbees. Well, blue for you. I mean, you could give her blue if you like.

AUDIENCE: I prefer purple.

PROFESSOR: You prefer purple. Thanks. Good job.

[APPLAUSE]

No offense is intended to Josiah and Tessa, but that was a classic example of how not to play the game. First off, Josiah could have won regardless of the coins that were up there, regardless of the values of the coins, if he'd chosen to go first. So this game, the person who goes first is guaranteed to not lose. And by that, I mean you may have a situation where you can have a tie in terms of the values, but you're guaranteed to not lose.

Now, he ended up winning anyway, because there were other errors made during this, and I don't have the time to enumerate all of them.

[LAUGHTER]

So we're just going to go move on and do the right thing. So let me first tell you, outside of DP, just in case you play this game over Spring Break or something, how you can win this game without having to compute complicated recursive memoization DP programs for this case.

So let's say I have V1, V2, V3, all the way to V n minus 1 and Vn. And remember n is even. So you've got to pick n over 2 coins if you're the first player and n over 2 if you're the second player.

So what the first player does is simply compute V1-- take the odd numbers-- and n is even, so you've got V n minus 1 being odd. Compute V1 plus V3 all the way to V n minus 1, and compare that with the even positions, V2 plus V4 all the way Vn. So in this particular instance, Josiah, who went second, if you just look at the odd positions, which is, in fact, what he ended up picking, it was 4 plus 39 plus 25, which was 68. So you can do this computation beforehand. You compute this to be 68, in our case. Compare that with 67. And you don't give up your first-player advantage. So you're going with the first player.

But now you say, I know that I can set this up-- and that wasn't the order that he did this-- but I know I can set this up so I always have a chance to get V1, V3, V5, et cetera. Because let's just say that the first player Josiah started first. And he decided, in this case, that the odd values are the ones that win. So let's say he picks V1.

At this point, Tessa sees V2 through Vn. So you're just looking at that. Now Tessa could either pick V2 or she could pick Vn. In that case, Josiah, who's the first player, could pick V3 or Vn

minus 1 and stick with his rule of picking the odd coins. So regardless of what Tessa does, the first player, Josiah, could pick odd if odd was the way to go or if the values were such that, even with the way to go, he could go even. So no reason for the first player to lose.

But let's just say that you're a nasty person, Michael Jordan nasty. You want to just press your opponent. You want to make sure they never want to play you again. So now you have a maximization problem. So this is clearly not DP. You don't need DP to add up a bunch of numbers.

But let's say that you want to, given a set of coins, V1 through Vn, you want a dynamic strategy that gives you the maximum value. This odd versus even is a bit constraining, because you're stuck to these positions. And it's good to be in that situation if you just want to win and you don't care how you win. But if you want to maximize, then it's a more complicated problem.

And so we have to talk about how you would do something that would give-- maybe it would be V1 and V4 and something else, depends on the values. How would you get to a situation-- if you're the first player. We'll just stick with you're the first player. You know you can't lose using this strategy. But not only do you want to not lose, you want to get as many coins as possible-- let's say this is money. More money is better, and that's your goal. So people understand that.

A little trick there with, I guess, a greedy algorithm you can think of. Or it's not really even something that you might want to call an algorithm. It's a little computation. And our goal now is to maximize the amount of money when assuming you move first. And so this is going to be a little bit different from all of the other problems we've looked at, because we have to now think about what the opponent would do. And there's going to be a sequence of moves here. You're going to move first, so that one is easy. You've got the whole problem.

But now you have to say, the opponent is going to move and pick a coin. And then you have to think of your subproblems as the potential rows of coins that are going to be different, depending on what the opponent does. And through this process, you have to maximize the value. So it's really pretty different from the other couple of examples that we looked at.

So we're going to have to do a little bit of set-up before we get to the point where we can write something like this, which is the solution to our DP. And so let me do that set-up. But it's not super complicated. And part of this is going to look kind of the same as previous problems

we've looked at.

So Vij is the max value we can definitely win if it is our turn. And only coins Vi through Vj remain. And so we have Vii. Then there's only one coin left, and you just pick i.

Now, you might say, but that's never going to happen if there's an even number of coins and I'm playing first, because the other player is going to be at the end of the-- is going to be the person who picks the last coin. But we need to categorize Vii because we need to model what the other player is going to do. So we can't just say that we're going to be looking at an even number of coins in terms of the row of coins that you look at, which is true, that when you get to move, you only see an even number of coins if you're the first player. But you do have to model what the opponent does . So we need the Vii.

And what you might see, of course, is a board that looks like Vi i plus 1, for some arbitrary i. So that's why I have it up there. But remember that you're only picking coins on the outside, so it's not like you're going to have a gap. I mean, you're not going to have V3 left and V7 left. There's no way that's going to happen. You're just going to keep shrinking, taking things from the left or the right. So it's going to be i and i plus 1. That make sense?

And so in this case, what would you pick? Vi and i plus 1. You just pick the max. Because at the end of this, either you did it right or you did it wrong. Either way, you're going to improve your situation by picking the max of Vi or Vi i plus 1. So there's no two things about it.

So here, you're going to pick the maximum of the two. And you might have Vi i plus 2, which is an odd number of coins that your opponent might see. It gets more complicated for Vi and i plus 2. We're going to have to now start thinking in more general terms as to what the different moves are. But we've got the base cases here. All I did here was take care of the base case or a couple of base cases associated with a single coin, which is what your opponent will see and pick, or two coins, which is your last move. So with DP, of course, you always have to go down to your base case, and that's when things become easy.

So we have to talk about two things and put up our recurrence together. The two things we have to talk about are what you do when you move, and that's actually fairly easy, and the second thing is what the model of the opponent looks like when you're waiting for him or her to move. So let's take a look at your move. And I've got V1. Let's look at Vi. Vj here, dot dot dot, Vn. So that's what you see. And you're looking at i and j. And at this point, you're seeing all

those other coins, the outer coins have disappeared into people's pockets. And you're looking at this interval.

So I'm going to write out what Vij should be. And keep in mind that we want to maximize the amount of money. And we want to say that you should be able to definitely win this amount of money, regardless of what the opponent does. So I want to do a max, obviously. And I have two choices here. I can pick Vi or I can pick Vj. It's not like there's a lot of choices here.

So if I pick Vi, then-- let me go ahead and say I pick Vi here. And here, I pick Vj. Let me draw this out a little bit better. So I've got to fill in these two arguments to my max.

So this is easy. I'm going to have a plus Vi here. Going to have a plus Vj here because I picked the appropriate value. And now, this is also not that difficult. What exactly happens? What can I put in here if I pick Vi?

AUDIENCE: Vi plus 1.

PROFESSOR: Yeah. Vi plus 1 to j. So the range becomes i plus 1 j. I have to be a little careful here in terms of whether I can argue that it's actually the V that I put in here. So the subtlety here is simply that the Vi plus 1 j is not something that I see in front of me. This is the complication. Vi plus 1 j is never a board that I see in front of me, whereas Vij was a board that I saw in front of me. So I have to model the boards that I see in front of me, because those are the boards that I have control over that I can maximize. I cannot would be Vi plus 1 j in there, simply because I don't quite know what that is, because of what I get eventually is not something I control. It's going to be the board after my opponent has moved.

So all I'm going to do is I'm going to say the range becomes-- range is i plus 1 j. And I'm going to say something else in a second here. The range is i j minus 1. And in both of these cases, the opponent moves.

So in order to actually write out my DP, I'm going to have to now look at the worst case situation in terms of the board I get back, because the only times I'm adding values is when I see a board in front of me and I pick a coin. And I'm going to have to say now the opponent is going to see an i plus 1 j-- or an i j minus 1, excuse me, and might do something. And I'm going to get something back. I'm going to assume that the opponent is just as smart as I am, knows DP, taken 6046, et cetera, et cetera. And I still want to get the maximum value that I can definitely win.

And so we need to look inside of that a little bit. And it's not that hard if you just make the assumption that I just did, which is the opponent is going to do-- I mean might not necessarily do the right thing. But you have to assume that the opponent knows just as much as you do and is going to try and do as well as possible.

So let's do that. That's the last thing that we have to do here. And it's just one more equation. So here's the solution. We now have Vi plus 1 j subproblem with the opponent picking. And the simple observation is that we are guaranteed the min of Vi plus 1 j minus 1 or Vi plus 2 j.

In this case, the opponent picks Vj, and in this case, the opponent picks Vi plus 1. And you're guaranteed the minimum of these two, because the opponent can only pick one coin. So that's the simple observation that lets you jump ahead to your next move, which is really what the DP is interested in, because that's the only time that you're actually executing something in terms of picking coins from the board and adding to your value. But we did have to model that as to what the maximum value was that you could win, jumping ahead of this move. And you have the min in here because you're assuming that this is a definite guarantee. That's what we want. It's possible that the opponent plays a different game from the one that we think he or she is going to play, which maximizes his or value. But the min is guaranteed.

So now that you've made this observation, it turns out we just have to take that, and we'll be done. So let's see. Let me erase this. It's the last set of equations. And it's just plugging in that observation into what I wrote before.

So we have Vi j equals max. This is the outer max that I had up there already, so that's the same max. And put these giant brackets here. And inside, I'm going to plug in this min, which is something that corresponds to the value that I would win in the worst case after the opponent plays the best possible move. And that would be Vi plus 1 j minus 1, V1 plus 2 j plus Vi. This Vi is the same as this one up here.

And you've got a plus Vj here. And I didn't actually do this, but in terms of writing out what the opponent would do in other subproblem case, but it's really pretty straightforward. You have a problem that corresponds to the i j minus 1 problem. And the opponent could pick the i-th or could pick the j minus 1th. If the opponent picks the j minus 1th, you get Vi j minus 2, and you need to take the min of that. In the other case, where the opponent picks i, in which case you get i plus 1 j minus 1.

And that's our DP. That's our DP. So the big difference here was the modeling that you had to do in the middle. We didn't actually have to do that in any of DPs we've covered in 046 at least up until this point.

Before we talk about complexity, that should just take a minute, people buy that? See that? Good.

So with all of these problems, the complexities are fairly straightforward to compute. The complexity here is simply, again, as before, the number of subproblems times the time it takes to solve the subproblem. You can see here are that these are all constant time operations.

So assuming that the Vij's have been computed, it's theta 1 time to compute a subproblem. So that's your theta 1. And as before, there are n squared subproblems. Sorry. Let's just do number of subproblems times theta 1. It's just theta n squared. So that was good.

All right. So good luck for the quiz. And don't worry too much about it. See you guys next week.