PROFESSOR: OK so any pointed questions before we go over blackjack?

AUDIENCE: Before you start blackjack can you go through the rules again?

PROFESSOR: The rules of the game?

AUDIENCE: Yes.

PROFESSOR: OK. Well I'm going to do something better than going through the rules. I'm going to abstract the rules away. So I'm going to say that a game of blackjack has, let's start here. So a game of blackjack has a deck. And we have x-ray vision or some other illegal way in which we know what the deck looks like from the beginning. How many cards in a deck?

AUDIENCE: 52.

PROFESSOR: OK. So say from 0 to 51 because we like zero-based indexing. So the way that the game starts is you get two cards, the dealer gets two cards. What is the only decision that you make in the game if you already have perfect information?

So you know the deck already. You don't need to wait for each card to know what it's going to be. So as the game starts you have to make one decision. What's that decision?

AUDIENCE: Hit or stand.

PROFESSOR: Well that's assuming you care what you get each time. So if you have to look at the card then you have to decide each time if you hit or stand. If you know all the cards in advance--

AUDIENCE: How much to bet?

PROFESSOR: I think our game has that fixed.

AUDIENCE: When to restart?

PROFESSOR: So it is hit or stand, but I want to restate that. Because you don't actually have to decide after every card if you want to hit or stand. You know what the cards are going to be in advance.

AUDIENCE: When to restart? Like how many cards to pick?

PROFESSOR: How many cards to pick. Yeah that's pretty much the same thing. So you don't have to decide every time hit or stand, you know ahead of time I want to hit three times, or I want to hit once. Because you know the entire deck. You don't need to actually look at the cards.

So each game the only decision that I make is-- suppose this is one game-- the only decision that I have is h, how many cards I'm going to hit. Right? And then after I make this decision the game will eat up some cards. I get two cards, the dealer gets two cards, then I hit some cards then the dealer gets some cards. And this is the number of cards that have been consumed.

Some number of cards that were played in this game. And the other thing that comes out of it is how much money I make. For our simple rules it's either I make $1, or nothing, or I lose $1. Fair enough? Everyone follows?

So I'm going to abstract all this in a helper method. I'm going to say that I have a method called round outcome. Where I say, look out of the entire deck I'm going to start at card i, so this is where my round starts. So I tell it which cards I start at. And how many cards I'm going to hit.

And it gives me back a tuple, where the first item is how many cards are played. So number of cards played. And the second item in the tuple is how much money do I make. h is how many cards I'm going to hit. So i says that I already played this many cards in previous games.

So a game starts with a full deck. And then I play some rounds. Each round I do a whole exchange with the dealer, where I hit some cards, and then the dealer hits some cards. And then I win or lose some money.

AUDIENCE: I don't understand why h and cp are different.

PROFESSOR: So cp is how many cards are played in total. And this looks at the initial cards. So first I get two cards, the dealer gets two cards. And then after I hit some cards the dealer also has to hit. And the dealer has a pretty find algorithm, right?

So the number of cards played includes the cards that were dealt initially and the cars that the dealer has to hit. So this little thing has all the blackjack rules encoded in it. Everything is already there.

AUDIENCE: So blackjack, that's roughly 2h? cp is 2h, something like that?

PROFESSOR: 4 plus 2h maybe. Because you get two cards, he gets two cards, and if you both deal h on average then--

AUDIENCE: So is cp the number of cards this round?

PROFESSOR: This round, yeah. OK so intuitively I already know, so I already know the entire deck. So the decision that I have to make is, would I play optimally every round? Or do I want to maybe lose the first round and leave some good cards for later?

So I have to figure out how many cards I'm going to hit each time so that overall I make the most amount of money when I leave the table. And when I run out of cards I leave the table. The dealer says shows over, gotta go.

Do the rules make sense? OK so I propose we approach this in two ways. First we model it as a graph problem, because we've already done this a few times. And then we model it as a dynamic programming problem. And we see how the two are related. Make sense?

Is this too simple for everyone? You guys already get everything? So how would I

model this is a graph problem?

AUDIENCE: Do you already know the order that you're going to get all these cards in?

PROFESSOR: Yep.

AUDIENCE: So I in lecture we briefly talked about having your starting node go to a bunch of other nodes that would be your potential next move, and then you just calculate your shortest path distance from there to the x value.

PROFESSOR: So that's the general approach. How do we do this for cards? So what would be nodes? What are the most intuitive nodes you could think about?

OK maybe intuitive for me. So nodes show our state, right? Show the state that the game is currently in. Sorry?

AUDIENCE: Your hand. Like your current cards. Is that your state? So from I guess 2 to ace.

PROFESSOR: From?

AUDIENCE: From 2 to ace, in terms of the number of choices you have.

PROFESSOR: OK so the problem is you have four cards of each type, right? And you don't know where they show up and everything. So I think this my code more complicated than it needs to be.

AUDIENCE: Like the rest of the cards in the deck that are remaining?

PROFESSOR: OK.

AUDIENCE: Is that a good [INAUDIBLE]?

PROFESSOR: How many cars I have left in the deck, right? So basically I would have one node for each of the cards here. And that says I start a new game at this card. OK so I'm going to draw some circles.

Say these are our nodes. Maybe I draw a bit too many of them. Where do we always start?

AUDIENCE: The left side.

PROFESSOR: First one, right? We start with 52 cards. So circle 0 means that we played 0 cards, we have 52 cards left. When do we draw an edge between nodes? What does an edge mean?

AUDIENCE: That's how many cards you've chosen, so the number cp.

PROFESSOR: OK so the number cp. It means I played a game, right? So one edge is a game. And it goes from one state to the next state.

So if I'm, say I'm at node i. How do I draw the edges? Say I'm somewhere here. So I already played i cards.

AUDIENCE: Iterate through all h's, cp's.

PROFESSOR: OK. So for h in what to what? What's the smallest h?

AUDIENCE: The smallest h is 1.

PROFESSOR: Really? Do I have to hit?

AUDIENCE: Wait isn't it 4? Because you're always dealing out 4.

PROFESSOR: Yes, so I might as well not count them, right? So h is how many cards I hit after the initial ones were dealt. So that I can start at 0, a nice and easy number. And where do I end? Rough approximation.

AUDIENCE: You could go to infinity and then break. It's still 11 because 11 at most-- oh no, 11 minus 4, 6. You need to know the rules of the game.

PROFESSOR: OK if you know the rules of the game it's that. If you don't know the rules of the game it's 52 minus i. OK so what's first thing I do? So how do I draw an edge representing a game where I hold h cards?

AUDIENCE: So you draw an edge from your current place to the output of round outcome with the 0 element.

PROFESSOR: So then let's store this output somewhere. Let's say o is round outcome. What do I give round outcome?

AUDIENCE: i and h.

PROFESSOR: See I picked good names. They're exactly what I have there. So I draw an edge from i to what?

AUDIENCE: To the output of round outcome, which is o. So it's 0.

PROFESSOR: OK.

AUDIENCE: Or that the node at o is 0.

PROFESSOR: Yeah. So suppose I'm at i and I've already played five cards, right? So say i equals 5, for example. And I know that if I hit once the dealer will also have to hit once. So in total I've played 6 cards.

And suppose I won. Then the output would look like this. 6 cards were played and I won. So plus 1. So I would draw an edge from 5 to what?

AUDIENCE: [INAUDIBLE]

PROFESSOR: OK, from 5 to 11 hopefully. So what's 11? 11 is 5 plus all of 0. OK, what am I missing there?

AUDIENCE: i to i plus [INAUDIBLE].

PROFESSOR: Yep. So this tells me how many cards I played in this game. I want to look at each game separately. I don't want to have to keep track of previous states. So this output is localized to this game.

It tells me how many cards have been drawn in this game in total. And how much money I made in this game. If I already played i cards before starting the game, after I play all of 0, the total number of cards is i plus 0.

Tiny detail, but you'd probably lose a point off of an exam or something if you forget it. OK so keep track of your state. It makes sense to write down this is my state, and then make sure that you're always representing it. So what's the cost of the edge?

And then our answer would be what path do I want? I want to make the most amount of money, so that's the longest path. How do I convert this to a shortest path problem, because this is what you know how to solve? Where I put it?

AUDIENCE: There.

PROFESSOR: Good answer, there. Right there. OK so this builds the graph, then I run some algorithm on it. What's the best algorithm I can run on it?

AUDIENCE: Dykstra because you can [INAUDIBLE]. But you can add everything on. The lowest negative edge weight is 1, or negative 1.

PROFESSOR: Are there negative cycles?

AUDIENCE: No.

PROFESSOR: Everything goes right, right? So even if I don't get hit any card, at least four cards will be played. So all these arrows go right. So then I heard a fancy term that I like. Can someone say it again? What's this graph?

AUDIENCE: DAG.

PROFESSOR: DAG. All the edges go one way. So this is a DAG. And that means that I can run what algorithm? Sorry?

AUDIENCE: [INAUDIBLE].

PROFESSOR: Topological sort plus DFS, the one that we talked about last time when everyone was out for Thanksgiving. So you have to believe me that it exists, or look at lecture notes. So top sort plus DFS will give me the shortest path in order of V plus E.

So this is better than Dykstra, which is E plus V log V. OK? OK I'm guessing most of you don't know how that algorithm works. Well good news, we're going to do this

using dynamic programming.

So we're going to represent the graph implicitly. And we're going to write everything without needing to know the algorithm. If you know it, it helps because you can see how they relate. But if you don't we should still be able to solve the problem.

But before we do that, does everyone understand this? Some nods would be nice so that I can feel good. Yes? Almost? OK. Questions?

AUDIENCE: Like the number 5 there, that's only if you hit, right? Or that's the number of cards remaining. Is there a particular reason you picked 5?

PROFESSOR: So I was choosing an example for i. So good question, what does i mean? So this means I'm-- assume that I'm starting a game. So starting-- sorry, a round. A new round. And I have already played i cards in the previous round.

So I started a new round and the first 5 cards are out of the deck. What's the best strategy I can have? What's the most money I can make?

AUDIENCE: But there's some nodes that connect to the next card over that won't necessarily end the game, so there aren't going to be any earnings, right? You could have 5 connect to 6, and that wouldn't--

PROFESSOR: Here I'm just trying to emphasize the point that all the edges move, all the edges point forward. But yeah, the edges have to go at least across four nodes. So assume there are some more nodes around here.

AUDIENCE: OK.

PROFESSOR: Yes?

AUDIENCE: You were saying how you have to be careful with something or else you'll lose a point. What was that?

PROFESSOR: How you draw your edges. Or when we switch to dynamic programming, what you write in your recursion. OK so last chance to ask a question before we do a

conceptual jump and use another algorithm. OK how do we do this using the dynamic programming?

So what are we going to have instead of nodes? Sorry? Well so you still have states in both cases. But here you represent them with nodes in a graph. In dynamic programming you represent them using-- yeah, the states are basically sub-problems.

And what do we, when we compute stuff, what do we use? Numbers is a vector or in a matrix. So there's no graph to work with. There are no extra algorithms to call. We just straight up compute to the numbers, we trade the answers to the problem.

So we're going to have an array. How many elements in the array? Can anyone guess? All right, I heard 52.

If I'm at element i, say i equals 5 because we used that before, what does this represent in the dynamic programming formulation? It's very similar to node 5 there. So what does it represent?

The fact that we're starting a game after we played the first five cards. Sorry, we're starting a round after we've played the first 5 cards. And we want to maximize our earnings from here on. So then the problem is, how do we maximize our earnings starting here?

So given that the deck has these cards, so the cards from 5 to 51. What's the maximum amount of money we can make by playing optimally? So maximum number of money we can win by playing optimally, starting around at card i. Starting at i.

So if I want to compute this-- by the way, speaking of bad variable names, we did this before. When people don't know how to name this array they name it dpi. I think we did that in our PSet. So this is the most useless name you can have for the array.

It just tells you that we're using dynamic programming. But it doesn't really tell you

what it means. So we're going to go for it, because it's nice and easy to write.

So if I want to compute dp of i, how do I do that?

AUDIENCE: x [INAUDIBLE] problems.

PROFESSOR: OK, so what are the sub-problems?

AUDIENCE: The function at i's past 5 where the i's are [INAUDIBLE] through this manner.

PROFESSOR: OK so I'd want to have something very similar to this, right? What are my decisions? So what are my choices? What do I iterate over?

AUDIENCE: How many hits.

PROFESSOR: Yep, exactly the same thing as before, right? Starting a node at i. I was starting a node at i here. So the choices are exactly the same.

So I'm going to start by looking at this line. Do I need to make any changes? Or do I copy straight over? So this is the algorithm for computing dp of i.

For i in 0 to 52. Sorry h in 52 minus i. All right. I hope we can do a bit better than that.

AUDIENCE: So we know that it's a max of 6, right? So can we just put that in instead? That's given the rules.

PROFESSOR: Sure. If you're smart you can. I'm not, so I'm just writing this. So this helper function that I have here, that I call the magic that implements the rules of bag check will save me. So if I say that oh, I want to hit 10 times, and that's impossible, then it'll probably give me an earning of minus infinity, which makes sure I never choose that path.

So all that is hidden in there. You're smart enough in blackjack so you can write 6. I'm not so I can't. So 52 minus i. OK what I do next?

Do I copy this next line? Or do I change it? Yeah let's copy it over. Sounds good.

How about this? Do I copy this? Trick question. Come on guys. So do I copy this line or not?

Do I have a graph here? Can I draw edges? OK so I'm not going to copy it. What do I do instead?

By trick I mean easy. So what do I do instead? I compute my answer directly. So if I hit h cards, what am I looking at?

AUDIENCE: Do you mean like a function?

PROFESSOR: A function, sorry?

AUDIENCE: You add the dp of o 1 or of o 0.

PROFESSOR: OK so first let's see if I have i cards, and say I hit, I do the same exact thing that I did before. I look at i is 5 and h equals 2. So then that function gives me the same answer, 6 1. So then I know that after this I'm going to end up in a state where I played the first 11 cards.

So I'm going to end up at 11. How much money did I make overall? OK, so 1 in this case. So how much money I made is o of-- I think it's o of 1. And after I land here, how much money I'm going to make? Assuming I'm still playing optimally.

AUDIENCE: [INAUDIBLE].

PROFESSOR: dp i plus.

AUDIENCE: [INAUDIBLE]. OK. So i plus o of 0 is used to compute-- so dp of i plus o of 0 is used to compute dp of i. This is the same thing as--

AUDIENCE: dp is a function here?

PROFESSOR: No. So you're wondering what the hell? Why will that work, right? Let's get to that in a minute. That will work, we have to make it work.

So here I'm drawing an edge from i to i plus o of 0. And the cost of the edge is

minus o of 1. So here we're looking at edges. Here I'm assuming that they already computed the answer here using some black magic. It's already available.

And I want to compute the answer here. So I have the cost of the edge plus whatever I had here. So if, suppose I know that if I start here and I finish the deck, I can make $20. So suppose I know that this is 20. What will the answer be here?

1 plus 20 which is? So if I hit-- how many times did I say there-- if I hit twice I guess I'll make 21. So this is a possible answer. And I have to go over all possible answers.

So this is how much I'm making if I hit h cards, right? Now I'm looking at multiple choices here. This is the answer for each choice. Which answer do I want in the end?

The largest. OK. So let's say I'm going to start with a choices array that stores all the answers. So here I'm just going to pend the answer, the possible answer. Choices append this guy. And then at the end of the for loop I'm going to take the max of choices, and I'm going to assign it where?

AUDIENCE: [INAUDIBLE].

PROFESSOR: So I promised I'm going to compute dp of i. I just finished computing dp of i. Now there's a little problem here. In order to compute this guy, I need to already have the answer for this guy. And maybe for some other guys here.

So an answer here depends on future answers. The arrows here are the same as the arrows here, right? They represent possible moves in the game. At the same time the arrows here represent dependencies. This answer depends on this answer, this answer depends on this answer, so on and so forth.

When we hear the word dependencies what do we think of? Topological sort. PSet, which one? PSet 6 still brings painful memories? Not anymore. We have a new one.

So in order to compute this I need to compute the answer to a few other sub-problems. To make sure that I have these answers ready by the time I compute

this. So to make sure that this code doesn't crash I have to compute all the answers to the sub-problems in the topological sort order. That's where topological sort fits in here. What's an obvious topological sort, if all the edges are pointing this way?

AUDIENCE: [INAUDIBLE].

PROFESSOR: Yep, thank you guys. So start at the easiest problem. What do you do if you have one card? And then go look at bigger and bigger and bigger problems, until we tackle the hard problems of what I do with the entire deck. So when I compute this problem I'm going to go to iterate how? From where to where?

AUDIENCE: 52 minus i to the 0. You're actually--

PROFESSOR: So this is i.

AUDIENCE: Oh, that's i.

PROFESSOR: So where do i go from here?

AUDIENCE: 51 to 0.

PROFESSOR: All right. So now whenever I access this guy I know it's already computed. So the code isn't going to crash. OK and this thing is my topological sort. So the advantage of this is that the code is a lot smaller, right?

Here I'm building the graph. So I'm calling some graph methods. And then I would have to have the code for computing the shortest path in a DAG. And then I would have to have some code for extracting the answer using that.

Here this is all the code. It's a few lines. And it's because the graph is represented implicitly. The topological sort is represented implicitly. The edges are represented implicitly.

So this looks like magic. But if you know where to look you'll find the items. You'll find the things that tell you what the graph looks like. Yes?

AUDIENCE: So that tells you how much money you can make, but does it tell you can--

PROFESSOR: How you make it? Not yet. Parent pointers. Yeah that's a good point. Let me see how we're doing on time. OK. I can talk parent pointers. Am I missing anything else?

There's one decision that I'm missing, that I missed here too. If things look bad for you, if you know you're going to lose money, what can you do? Walk away. How much do you get?

AUDIENCE: 0.

PROFESSOR: Yep. So you always have an edge that takes you all the way out with cost 0. So the way I represent that here is I start with a choice of 0. OK now let's do parent pointers. What's the easiest way of doing parent pointers?

AUDIENCE: Keep track of the max of-- [INAUDIBLE].

PROFESSOR: So I want to keep track-- for every answer here I want to keep back of the h that led me to that answer. So in the dynamic programming vector, instead of just storing the maximum cost, I'm going to store the maximum cost and the number of hits I have to make to get there. So instead of having one number here that's 21 I'm going to have two numbers.

I'm going to have 21 and the 1 that says you have to hit-- sorry, 2-- you have to hit twice. And then you'll go on this arrow and so on and so forth. And if you know how many hits you have to make you can follow this-- you can follow these parent pointers and they will tell you how to play the entire game. You start at 0 and play the entire game.

Do we want to change the pseudocode to do that? OK, it's not too hard. How many people want to see the pseudocode changes? All right, I guess I don't have to write it then. OK any questions on this?

So the change is really simple. Instead of storing one number you store a tuple. And then because tuples are sorted the right way maximum still works. You don't have to change that, you just have to change what you store down there.

AUDIENCE: So you just add h in, into your--

PROFESSOR: Yeah, you add one more parentheses for the tuple. Wait, I already have two, OK, never mind.

AUDIENCE: You have to go through your choices, so you just--

AUDIENCE: Is there any difference in making a separate dictionary for parent pointers? Does it make any difference in run time?

PROFESSOR: Absolutely no difference running time. The code is, it might be more complicated, it might be more simple, depending on how your brain works. It's easier to patch existing code to add in parent pointers this way. If you're writing new code it might be easier for you to have a separate dictionary. This is fewer lines of code though. OK any questions on this? Yes?

AUDIENCE: Could I generalize and say that if you have a topological sort you can do everything backwards. Otherwise you should use memoization.

PROFESSOR: So actually you're doing it in the order of the topological sort. You're not doing it backwards.

AUDIENCE: Oh, OK sorry. If you have a topological sort then you should do it in that order. But if you don't have a topological sort then you should do memoization.

PROFESSOR: If you don't know the topological sort. But there has to be one, right? Because otherwise you have infinite loops. If you have an infinite loop in your dependency graph, then you're not going to have an answer.

So that means you're dp formulation is bad. Fortunately for all the problems that we have the topological sort is pretty obvious. It either grows from 0 to the problem size or the other way around.

AUDIENCE: So then memoization is?

PROFESSOR: So memoization is, it's more of a proof of concept thing. It shows you that if you

have the recursion, everything else can be done automatically. So like if you build a graph then you can run top sort and get the answer, you don't have to think about it. We think about it because the code is smaller if we do it this way.

If I'd have to write memoization I would add 4 or 5 more lines, right? But the point of doing it that way is, all you need is that recursion. If you have this-- so this is the magic part. If you have this, so this line here of what your choices are and max, how you combine them, then everything else is mechanical. Once you've solved enough problems everything else is just follow the process.

So this is the equivalent of in graph problems, the hard part is figuring out what the state is. Once you know what the state you know that these are the vertices, and you know how to draw edges between them. And then you know what algorithm to run. So the hard part is still knowing what the state is. Anything else?

So this is dynamic programming. Smaller code. This is the graph approach. They essentially compute the same thing. This is more code, this is less code. And if you see the correspondence between them then you understand the problem a little bit better.

The main point is when you have a new problem you can approach it either way. If you see the dynamic programming solution right away write it down, you're done. If not, draw the graph. Think of what the state is, draw the edges. And then after that you can write the math.

OK let's talk about a new problem. Let's talk about the problem that shows up on interviews. People excited about interviews? OK suppose you have a sequence of numbers, I'm going to draw a sequence here.

And you want to find the shortest increasing sub-sequence. So you get to choose some numbers out of these numbers. And they have to form an increasing sequence.

So for example this is a sequence. It happens to be increasing. This is also a sequence, but it's not increasing. So it's not a valid answer. And I want the longest

sequence, the longest sub-sequence that is increasing. Does the problem make sense?

How do we solve it? Do we want to solve it using dynamic programming or using graphs? OK so votes for dynamic programming. Votes for graph.

Well too bad, it looks prettier as a graph. So how do we solve it as a dynamic programming problem? What are the sub-problems?

AUDIENCE: The largest sub-sequence

PROFESSOR: Starting somewhere, right? I'm going to go off that answer because I know how to go off of it better. So say start here. Say start at 4. Or actually say I start at 3.

I have two choices. 5, which is closer to me. And 4. Well I have a few more choices, but they're further away. Whatever.

So these are my choices starting at 3. If I decide that I'm going to go from 3 to 4 and the next number I choose is 4, now I want the longest sub-sequence starting at 4, right? It still has to be longest sub-sequence. So from here on, no matter what happened before, my behavior still has to be optimal.

If instead I chose 7, I don't care what happened before. The behavior still has to be optimal. So a sub-problem says start at number i. So starting at number i. By the way we're going to use zero-based indexing again because we like it.

So starting at number i, what's the longest increasing sub-sequence I can get? So the length of the blah, blah, blah. The length of, you get the point. OK so I'm going to have an array again, right? Which stores the answers.

The array is going to be named dp. If I have N numbers I'm going to have N elements, from 0 to N minus 1. Suppose I'm at element i. And suppose this original array is called a. I'm in the mood for good variable names today. So how do I compute dp if i? Let's write some pseudocode for it.

AUDIENCE: [INAUDIBLE] N minus i--

PROFESSOR: So what's h?

AUDIENCE: The number of steps we want to take--

PROFESSOR: So if I'm going from 3 to 4 h would be what?

AUDIENCE: 2. 3.

PROFESSOR: OK so I'm going to have to do additions and subtractions, and this is going to confuse me. So how about I propose this. What you say is perfectly valid, but instead, to make sure I don't make too many mistakes, I'm going to look at the number I land at. At the index directly.

So I'm going to say I start at i and end at j. So the next step is j. And then your h is j minus i.

So I'm not going to look at the number of numbers I hop over, all I care about is where do I land. So what's the next number in the sub-sequence? If I do it that way, where do I start?

AUDIENCE: i plus 1. So I can choose the same number twice, right? So plus 1 to n. And then I'm going to have a choices array here that I start, initialize with nothing. And then what's the candidate, if I'm at j? So what answer am I looking at?

AUDIENCE: dp of j.

PROFESSOR: OK. So if I'm at i, and I'm considering choosing j as the next element, then my sequence will be-- my sequence length will be dp of j almost. Plus 1. OK, can I choose all the-- can I go through all the j's? Can I go from 3 to 2.

AUDIENCE: No. The number at j is greater than--

PROFESSOR: If the number at j is greater than the number at i then I have this new choice, dp of j plus 1. What do I do with it?

AUDIENCE: Stick it in choices.

PROFESSOR: Stick it in choices. Sorry this is append. And afterwards? And by the way, this thing is under the if.

AUDIENCE: [INAUDIBLE].

PROFESSOR: OK and I'm missing one choice this way. What's my default choice? So what's the sequence length if I just stay there?

AUDIENCE: [INAUDIBLE].

PROFESSOR: So if I decide to not choose anything after 3 then I have a number, 1. Small detail, again one of those things that costs you one point if you get it wrong. OK so I have a default. So I know that this is going to be well-defined, and I have all my possible choices. Yes?

AUDIENCE: dp of j representing a [INAUDIBLE].

PROFESSOR: So it's saying, if I'm at i, and the next number in the sequence is j, what's the longest sub-sequence-- the length of the longest sub-sequence starting at j? So let's run the dp for this example actually. Let's get a feel for why it works and how it works.

So I'm going to copy it again here. 8, 3, 5, 2, 4, 9, 7, 11. So this is a. And dp is here. Where do I start by the way?

AUDIENCE: [INAUDIBLE].

PROFESSOR: So I have the algorithm here, how do I iterate? For i in--

AUDIENCE: N minus N is 0.

PROFESSOR: N minus 1 all the way to 0. So in this case we're going to start at 11, right? The default choice is 1. Do I have any other choice? Can I go forward? Nope. So this is going to be 1.

Now for 7 my array of choices has a default of 1. And then for-- let me write the indices too, so I don't get confused. 0, 1, 2, 3, 4, 5, 6, 7. And these are i's. So we're at 7, i equals 6. For j equals 7, is a of j greater than a of i?

OK. So then 7, 11 is a possible choice, right? So if I choose 11 as the next point in my sequence, what's the total sequence length? 2. And 1 plus dp of 7 equals 2. So this is good. So far the answers add up.

So I have 1 and 2 as my candidates for the answer 2dp of 6. What's the maximum? All right. Works so far. How about 9? What are the possible answers for 9? So what's choices?

First there's 1, there's always 1. And then for j equals 6, will the if be true? No. I can't add a 7 after a 9, right? So go to the next one. For j equals 7, will the if be true?

So this append will happen, right? What will be appended in the array? And this means that if I'm at 9, and then the next element is 11, the longest sequence I can get has length 2.

OK, what's the answer for 9? So if I start at 9 the longest sequence I can make has length 2. Let's look at 4 now.

AUDIENCE: [INAUDIBLE].

PROFESSOR: What is?

AUDIENCE: Just in general that's-- the problem is defined as, OK in this case you go from 9 to 11. Or do you have to go from 9 to the next element?

PROFESSOR: So this is the longest sub-sequence if I do-- so the longest sub-sequence I get overall. I don't have to go to the next element. So if my problem looks like this, what is the best answer?

AUDIENCE: It's almost defined as--

PROFESSOR: So the problem is defined as, this is your first element in the sub-sequence. What's the best answer you can get?

AUDIENCE: I thought in that case it would be 1 because there's nothing following it that's

greater.

PROFESSOR: This is greater, right?

AUDIENCE: Unless it skips.

PROFESSOR: So it's a sub-sequence, not a sub-string, which means it can skip. I hope I got these right. So you can skip, otherwise the answer would be a bit easier to compute. OK how about 4?

So let's start with 1 because that's the easy one. And then?

AUDIENCE: 1, 3, 3, 2, right? Because it's going to be 3 for that one.

PROFESSOR: OK.

AUDIENCE: And then 3 for this one.

PROFESSOR: 3 for this one.

AUDIENCE: And then 2.

PROFESSOR: So all these are bigger, so all of them are possible next candidates. And these are the sequence lengths that I can get if I choose them. Final answer?

AUDIENCE: 3.

PROFESSOR: 3 maximum.

AUDIENCE: But for setting the parent pointers you'd want to take the closest thing, right?

PROFESSOR: As long as it's a maximum I don't care. So what are possible parent pointers here? The 2's, right? So either this or this. Do I care which one I chose?

AUDIENCE: No, I guess I don't.

PROFESSOR: As long as I choose a 2. From a 3 I know I have to go to a 2. I can't go to 1 because otherwise it wouldn't be as long as possible. And then from 2's I have to go to 1, and

I don't care which one. OK how about 2, what's dp of 2?

Does everyone else see it? So these are all possible choices because they're all bigger than 2. And I get 1 if I don't choose anything, 4, 3, 3, 2. So 4 is the biggest answer. Let's look at this one, this one's a bit interesting, 5.

So what are the choices here? 1 if I don't look at anything else. Then? There's a 3, 4, then 9. A 3, 4, then 7. And? And a 2 for the 11.

So this if is going to skip these two elements, which I can't use to make an increasing sub-sequence. And then it's going to look at these ones, and it's going to add 1 to the numbers here. And I get 3. OK. What is the answer for 3?

AUDIENCE: 4.

PROFESSOR: And what is the answer for 8?

AUDIENCE: 3.

PROFESSOR: Right, the choices are 9 and 11. Starting with itself and then 9 and 11. So now what's the longest-- what's the answer overall for this problem? So it's not dp of 0, right?

Before when I had blackjack I knew that I have to start at the first card. So the answer was dp of 0. In this case it's not dp 0, it's the maximum of all the dp's here.

Because I can start my sequence anywhere I want. So I have to take the maximum. And that's the overall answer, which in this case is 4. OK does it make sense now? Somewhat?

So if you don't understand please look at how you'd represent this as a graph. The idea is that the numbers are nodes and you draw an edge between numbers, where the first number is smaller than the second number. Write that formulation, write the shortest path for that, and see how that matches to this.