PROFESSOR: Today's our last lecture on dynamic programming, the grand finale. And we have a bunch of fun examples today on piano, guitar, *Tetris* and *Super Mario Brothers.* What could be better? We are, again, going to follow this five-step plan to dynamic programming, define sub-problems, guess something in order to solve a sub-problem, write a recurrence that uses that guessing to relate different sub-problems, then build your dynamic programming either by just implementing as a recursive algorithm and memorizing or building it bottom up.

For the first, you need to check with the recurrence is acyclic. For the second, you need an actual topological order. These are, of course, equivalent constraints. Personally, I like to write down on topological order because that proves to me that it is acyclic, if I think about it. But either way is fine.

Then we get that the total running time is number of sub-problems times time per sub-problems, and then we need to solve our original problem. Usually it's just one of the sub-problems, but sometimes we have to look at a few of them. So that's what we're going to do. And we have one new concept today that all these examples will illustrate, which is a kind of second kind of guessing.

We've talked about guessing in part two here in which we saw the obvious thing. In the recurrence, we usually take the min of a bunch of options or the max of a bunch of options. And those options correspond to a guessed feature. We don't know whether the go left or go right, so we try them both. That's guessing.

But there's another way to guess. So two kinds of guessing. So you can do it in step two. Let's see what I have to say about these. In step two and three, you are guessing usually which sub-problems to use in order to solve your bigger sub-

problem. So that's what we've seen many, many times by now. Every DP that we've covered except for Fibonacci numbers has used this kind of guessing. And it's sort of the most common, I guess you might say.

But there's a higher level of guessing that you can use, which we've sort of seen in the knapsack dynamic programming, dynamic program, which is when you define your sub-problems, you can add more. Add more sub-problems to guess or you can think of it as remembering more features of the solution. And we just leave it at that.

Essentially what this does-- so remember with knapsack, we had a sequence of items. They had values and sizes. And we had some target knapsack, some capacity. We wanted to pack those items into that knapsack. And the obvious sub-problems were suffixes of the items. Because we always know suffixes, prefixes, substrings, those are the obvious things to try.

But suffixes wasn't quite enough. Because if we looked at a suffix, we didn't know of the prefix that we've skipped over how many of those items were in-- and in particular, how much of the capacity we'd used up. And so we needed to add more sub-problems to remember how much capacity had we used up in the prefix. We did that by multiplying every sub-problem by s different choices, which is how many units of the knapsack still remain.

So in some sense, we're remembering more about the prefix. You can also think of it as-- in the more forward direction, we have the suffix problem. I'm going to solve it s different times, or s plus 1 different times. I'm going to solve it. What if I had a big knapsack? What if I had a smaller knapsack? What if I had a zero-size knapsack? All of those different versions of the problem. In some sense, you were solving more sub-problems. You're, in some, sense finding more solutions to that sub-problem. You're looking at a suffix. And I want to know all these different solutions that use different amounts of the knapsack.

So in that sense, you're just adding more sub-problems. But from a guessing perspective, you're remembering more about the past. We're going to see a bunch of examples of this type today. We'll always use this type, but we'll see more of this

where the obvious sub-problems don't work and we need to add more.

So the first example is piano and guitar fingering. This is a practical problem for any musicians in the audience. How many people here play piano, or have played piano? OK, about a quarter. How many people have played guitar? A few, all right. I brought my guitar. I've been learning this semester. I'm not very good yet, but we'll fool around with it a little bit.

So the general idea is you're given some musical piece that you want to play. And on a piano, there's a bunch of keys. You have all these keyboards, so you know what a piano looks like, more or less. It's just like a keyboard, but only row. It's crazy. Each key that you press makes a note, and every key has a different note.

So it's very simple from a computer scientist's perspective. You want to play a note, you push the key. But you could push it with any one of these fingers. Humans have 10 fingers. Most humans. I guess a few have more. But you want to know, which finger should I use to play each note? It may not seem like a big deal. And if you're only playing one note, it's not a big deal.

But if you're going to play a long sequence of notes, some transitions are easier than others. So let's say we're given a sequence of n notes we want to play. And we want to find a fingering for each note. So fingering, so let's say there are-- I'm going to label the fingers on your hand, 1 up to f. For humans, f is 5 or 10, depending on if you're doing one hand or two hand stuff.

I think to keep it simple, let's think about piano, right hand only, and just you're playing one note at the time, OK? We're going to make it more complicated later. But let's just think of a note as being a single note. OK, or you can think of guitar, single note, left hand is playing things. You want to assign one of these fingers to each node.

And then you have a difficulty measure, d. And this you need to think about for awhile musically, or anatomically, how to define. If we have some note p and we're on finger f and we want to transition to note q using figure g, how hard is that? So

this is-- the p and q are notes to play. I guess p stands for pitch. And f and g are fingers.

So this is how hard is it to transition from f on p to g on q. There's a huge literature on for piano. There are a lot of rules like, well, if p is much smaller than q, unless they're stretched, then that becomes hard. And if you want to stretch, you probably need to use fingers that are far away from each other. If your playing legato-- so you have to smoothly go from one note the other-- you can't use the same finger on both keys. So if f equals g and you're playing legato, then p better be the same as q sort of thing.

There's a weak finger rule. You tend to avoid fingers four and five, these two. Apparently going from-- I'm not much of a pianist-- so going from between three and four, which I can barely hold them up, it's kind of difficult, is "annoying." That's what I wrote down. So between three and four transitions you try and avoid. And so you can encode into this function. It's a giant table. You can just put in whatever values you want that you're most comfortable with. And music theorists work a lot on trying to define these function so. So you can do that.

And for guitar, maybe I should do a little example. Get this out. I can't play much, so bear with me. Bet you didn't think this would happen in 006.

[LAUGHTER]

So let's see. So let's say you're trying to play your favorite song.

[STRUMS "SUPER MARIO BROTHERS" THEME]

[LAUGHTER]

OK. So when I'm playing that, I have to think about the fingering. Which finger is going to go where to play each note? OK, so the first notes are actually open, so it's really easy. And then I go up to holding the first fret on the fifth string. OK, and I'm using my index finger because everyone loves to use their index finger. And in particular because the very next note I'm going to play-- well, actually it's down

here.

Then the next note is going to be this one. So I'm holding on the third fret of the bottom string. And then I've got to transition over here. And actually, usually I do it with my middle finger. I don't know quite why I find that easier, but I do. OK, and so I've actually played that opening a zillion times with lots of different things. This is the one I found to be the most comfortable.

And there's this issue, right? If your pinky is here, where can I reach with this finger? Where can I reach with this finger? It gets difficult. And in particular, it's very hard for me to reach down here when my pink is there. And so you can encode that in this d function however you want. You get the idea.

[APPLAUSE]

Thanks. I'll skip to our lessons. [? We're ?] [? worth ?] [? it. ?] So let's solve this with dynamic programming, OK? That's the cool thing. So we can do it. And we follow our five step procedure. So the first thing is to define sub-problems. What are the sub-problems for a set-up like this? What are the three obvious candidates? Do you remember last lecture? How many people know the answer? Just checking. One person. Go for it.

AUDIENCE: Prefixes, suffixes, and substrings.

PROFESSOR: Right. Prefixes, suffixes, and substrings. We have a sequence of notes. We're not going to worry about the sequence of fingers. I don't think that's too big a deal. That's what we're finding. What we're given is a sequence of notes, so we should try suffixes, prefixes, or substrings. I'll just tell you, suffixes are fine. Kind of. So a sub-problem will be suffixes, so how to play notes from i onwards.

Intuitively, we want to figure out, how should we play the first note? And then we go on to the second note and so on. So we're applying them one by one from left to right from the prefix side. And so we'll always be left with a suffix.

OK, then we need to guess something. What's the obvious thing to guess, given I

need to play notes i onward? Think little harder. This one you shouldn't have to think. That's what I tell you. Try suffixes, try prefixes, try substrings. Yeah?

AUDIENCE: Maybe which finger to just put around i?

PROFESSOR: Yeah, which we're going to use for note i. Our whole point is to assign fingering. The first note here is i. So let's think about i, what could you do for i? We'll try all the possibilities. Which finger to use for note i? OK, now the really hard part-- because it's impossible-- is to write a recurrence. This is wrong, by the way, but it's the first thing to try.

So this is what I want to ask you to do because it's not possible. But intuitively, what we might try to do is we're trying to solve DP for i. And we want to find-- this is difficulty, so you want to minimize difficulty. So we'll take a min over all of our guesses of what it would take to solve the rest of the notes, to play the rest of the notes, plus somehow the cost of playing the first note. So what's the cost of playing the first note? And then is going to be a for loop over fingers.

OK, that's going to be the min. We want to try all possible fingers for note i. Then we have to play all the remaining notes. And then there's this transition cost where you're going from note i to i plus 1. So it's going to be something like d of if-- we know that we use finger f to play i-- then we have to go to note i plus 1. But then the problem is we have no idea what to write here, because we don't know what finger we're going to guess for note i plus 1. So this cannot be known.

OK, but it's the first thing you should try, because often this works. For simple DPs, that's enough for sub-problems. But we need to know more information about what we're going to do next. And this seems very worrisome, maybe now we have to guess two things. Do we have to guess more than two things? Turns out two things is enough. But we cannot use this type of guessing. We need to use-- we need to add more sub-problems. More sub-problem, more power.

So any guesses what we could do for sub-problem? A couple of right answers here. Yeah?

AUDIENCE: Maybe like all the suffixes [INAUDIBLE] like the i, for all written i's, like all the possible fingers for i?

PROFESSOR: All the possible fingers for i in the sub-problem. Yeah, good. How to play-- it's still about the suffixes. We're still going to use that. But we're going to suppose we already know what finger to use for the first note, note i. OK, this is a little weird, because we were guessing that before. Now we're just supposing someone tells us, use finger f for that note. This will work. That's the one I had in mind.

But the question becomes, what should we guess? Anyone else? You clearly get [? a pillow. ?] I don't know how many you have by now. Have another one. That's tough. This is not easy to figure out. Now, given that that's our sub-problem, what is the next thing to guess? Do you have an idea?

AUDIENCE: I got an idea to define it. Like either the next or previous finger for the--

PROFESSOR: The next or previous finger. Well, I'm looking at suffixes. So I only care about the next one. Yeah. I see what you mean by next or previous. But what we mean is note i plus 1, that's the next thing we don't know about. So we're going to guess finger-- we'll call it g-- for note i plus 1. And now magically, this recurrence becomes easy to write. So it's almost the same thing.

I wish I could just copy and paste this over, but I can't. It's not a digital blackboard. Are there digital blackboards? That would be cool. Someone should make that. I don't know why switched from open parens to square brackets, but I did. Then we have-- I think it's just the obvious thing, if i plus 1 g. Ahh, this is a slightly wrong, though. It's a copy paste error.

This should really be DP of i comma f, because now sub-problem consists of two things-- which suffix am I in, and what's my finger for note I? And so when I call DP, I also have to provide two arguments. It's going to be DP of i plus 1 comma g. And then I'm looping over g. I'm trying all possibilities for g. That's the recurrence.

So if I want starting with finger f on note i, how do I solve the suffix from i? Well, I guess what finger am I going to use for the very next note. Then I have to pay this

transition cost for f on i to g on i plus 1. Yeah, OK. So slightly, I'm cheating the notation here. This probably should be the note, what is note i, and this thing should be what is note i plus 1.

If you think of this d function just being given notes, pitches that you need to play, instead of indices into the array. It doesn't really matter, but that's how I defined it before. OK, so I have to pay this transition cost. What does it take to make that transition from i to i plus 1? And then what does it take to do the rest of the notes, given that now my finger is-- or now finger g is playing the note i plus 1?

So we transition from f to g, and that's now kept track of in the sub-problem. This is the magic of defining more sub-problem. We needed to know where our finger used to be. And now we're telling it, oh, your finger right now is finger f. Finger f is the one that's currently playing the note. And then afterwards, g is the finger that's currently playing the note, and we can keep track of that.

You could also define this to say, oh, f was the finger that was used for the previous note, note i minus 1. But it's just a shifting of the indices here. You can do i minus 1 to i instead of i to i plus 1. But this is, I think, slightly cleaner.

OK, and then we have a DP, right? We've just memoized that recurrence. We get a recursive DP, or you could build it bottom up. If you were building it bottom up, you'd want to know a topological order. And this requires a little bit of care because there's two parameters. And so it's going to be a for loop over those two parameters in some order. And I believe the right order is for i has to go from right to left because this is suffixes.

So I would write reversed range n python if there are n notes. And then within that loop, I would do a for loop over f. If you reverse the order of these for loops, it would not be in the right order, I'm pretty sure. But this one will work. You can check it. And then to solve our original problem, here we require a little more work because none of these sub-problems are what we want to solve because we don't know what the first finger is.

We know what the first note is. That's note 0. But what finger goes there? I don't know. And DP of 0 requires us to give it a finger. Give it the finger, ha. Give it the finger for whatever is the first note. So this is pretty easy though. We just take a min over those choices. Which finger should we give it?

That should do it. So we don't know what finger to start with. Just try them all, take the min. This is just like the guessing that we did here, just a slightly simpler version. There's no transition cost because there's no transition. We weren't anywhere before. Just what finger do you start with? I don't care what finger I start with. It's how I transition from one note to the next that's hard. OK, done. That's the DP.

Now, if this is not obvious or not clear, I think it's easier to think about it in the DAG form. So let's draw all the sub problems. We have here a two dimensional matrix of sub-problems. We have the different suffixes on the one hand. So this is it, it stats a 9, goes to n minus 1. And then in the other dimension, we have what finger to use from 1 to f.

And so in each of these positions, there a note. There's a sub-problem. Race. I wanted to get five rows because there are five fingers. And then our transitions basically look-- if we're at finger one on this note, we can go to finger one on the next note. Or we can go, if we're not legato, or we can go to finger two on the next note, or finger three or finger four or finger five.

And then if we're starting with finger two, we could go to any one of these. So you get a complete bipartite graph, which you usually draw like this. That is how graph theorists draw complete bipartite graphs. OK, but I tried to draw a little more explicitly here. It's just any possible transition.

And for each of these, the point is you can compute the D cost, because you know what figure you were at. You know what finger you are going to and what note you're starting from and what note you're going to. Those are the four arguments you need for D. So you put those weights on, and then you solve shortest paths on this DAG. And that is exactly what this DP is doing, OK?

Except there's no single source here, which is kind of annoying. And so you need to take this min over what's the shortest path from here, what's the shortest path from here, from here, from here, from here. Of course, you don't actually need to do that by running single source shortest paths f times. If you're a clever shortest paths person, you would add an extra source note, connect that with 0 weight to all of these sources. So put 0s on there.

And then do single shortest paths from here. And you will find the best way. You don't really care where you started, so this is trying all the options. That's exactly what we're doing here. But here I'm doing it with the shortest paths trick, here I'm doing it with guessing and taking a min like DP style. OK, so that's how to do piano figuring and guitar fingering for single hand, one note at time. Questions?

And this even worse for aliens if you have arbitrarily many fingers on your hand. I guess we should figure out what's the running time. So we have sub-problems. We see how many sub-problems there are here. There's n times f sub-problems. How much time, or how many choices are there for our guess? Well there's f different choices for what finger we use.

And when we do this min, we spend theta F time. Because there's a for loop over F, we're doing constant work assuming D lookups take constant time. This is theta F time. So we multiply those two things together, and we get the total time, the number of sub-problems which is n times F, and we multiply them by theta F for each sub-problem. So this is nF squared. And given F is usually pretty small, it's almost linear time. So that's a pretty good algorithm.

But in reality, you tend to play multiple notes at the same time. In music, typically you're playing a chord. With piano, you're playing several notes with one hand, maybe several notes with another hand. Two handed piano, it's crazy. You could do four handed piano, make it a little more exciting. With the guitar, play-- I don't know very many chords, but I know at least one. You play, I don't know.

This looks like something. That's a G chord. Do I know any others? And that's an E chord. All right, you get the idea. I mean, for each of these chords, different people

use different fingers, even for a single cord. So it's sort of a personal taste how you're going to define your difficulty measure. But I could play an E like this, or I could-- I don't know, play it like this. Or I could play like this.

And there's lots of crazy ways to put your finger here and your finger here and your finger here. And for each of them, you could define some difficulty. And then, of course, is a transition from one chord to another. And because there's different ways to play different chords, that wasn't a very good example because they all look pretty bad. Well, this one for example, this is the G again.

I could use my-- one, two, three, four-- fourth finger here, or I could use my fifth finger. My instructor says we should use our pinky because people tend not to use their pinky. But it makes a difference what I'm going to transition to next. Maybe my pinky really needs to go over here next and I should free it up for later, or maybe it's better if this one's freed because then I can move it somewhere else. So that's what we'd like to capture in a generalized form or this dynamic program, and we can do it.

So I'll try to do it quickly so we can get on to the other examples. All right, other fun stuff. Actually, there's another fun thing with guitar, which is that there's more than one way to play each note. There are six strings here. And you could play like this note for the Super Mario Brothers. I could also play that doing the fifth thing here. It's slightly out of tune, but those sound almost the same.

Or I could play on the 10th fret on the third string. That's the same as bottom one. So a lot of options, so you also like to capture that. This is actually not too hard. You just need to generalize the notion of finger to what finger you're using and what string you're using. So there are f different choices for what finger you're using. If you use a generalized guitar, there's s choices for what string you're playing. There's a lot of different guitars with various numbers of strings, so we can just generalize that.

And now it's not only, which finger am I going to use, but what sting will I play it on? And then you can still define a difficulty measure like this for this set up, depending

both on the finger and the string. And then the running time grows slightly. It's now n times F squared S squared, because now I have to take the product of F and S. OK, so that's first thing.

But then if I wanted to do multiple notes, well, you can imagine it's a similar type of deal. It's going to get harder though. First thing we need to generalize is the input. Before the input was a sequence of notes. Now it's going to be a sequence of multi-notes.

So notes of i is now going to be, let's say, a list of notes that all need to be played at once. And conveniently, it's probably going to be, at most, F notes, because you really can only play one note with each finger pretty much. I guess you could try to play two notes at once on a piano with a finger, but eh. It sounds difficult. For a guitar, it's at most s notes. You can only play one note per string, more or less.

So that's our input. And now we need to adjust the dynamic program. And I think I'll tell you how to do this. Basically, now you need to know where all your fingers are. So you go from one pose to another pose, from one chord to another. Different ways to finger that. Which fingers are and which strings and which frets on the guitar, which fingers are on which keys on the keyboard.

But you just need to know all that. And all your fingers might be doing something, and you've got to know for each finger what note is it on, or is it not being used at all. So how many different ways to do such a mapping are there? I mean, this is just a function. So it's the number of targets of the function. So how many of these are there. Gosh, well, I guess we said there are, at most, f notes.

So f plus 1 is the maximum number of possible things each finger can do. And we raise that to the power of the number of fingers. That's the possible mappings of what all of my fingers could be doing. It's exponential in f, not so great. But if f is 5, it's all right. And then-- well, then you just generalize the rest. I don't think I'll write it down in detail.

But our sub-problems now are going to be-- let me switch boards here. How do we

play these multi-notes from i onwards, given that we're going to use that pose-- or I called it the state of all my fingers-- for the first notes of i is now a whole bunch of notes. So given I'm now going to play those notes with this particular finger assignment, how do I play the rest? And then what we'll guess is the entire finger assignment for the next set of notes, i plus 1-- the next chord, if you will.

And that guessing involves now F plus 1 to the F time. And then we just write the recurrence in the same way. So we're basically generalizing here we call the finger, now it's an entire pose for your hand. Instead of F, you might write H for hand or something. And so the running time in this situation is going to go up to something like n times F of plus 1 to the F. Did I miss anything? Probably have to square that. 2F.

Before it was F squared, now it's just F plus 1 to the F squared. So if F is small, this is all right. Otherwise, not so great. This is the best algorithm I know for chord fingering. Questions? Just trying to make it practical, solve the real life problem. I would love, I think-- I don't know if this has been implemented, but someone should implement this in some-- I don't know, score program, musical score program. I would love as learning guitar, it'd be great for someone to just tell me how to finger things. Then I can retroactively figure out why using the dynamic program.

All right, let's move on to Tetris. All these problems are going to have the same flavor. You can solve them with basically the same dynamic program. It's all about figuring out what should the sub-problems be. So let me-- does anyone here not know Tetris? OK, good. No one's willing to admit it. So you've got these blocks falling.

But I'm going to make several artificial constraints. First of all, I tell you the entire sequence of pieces that are going to come. This is more like a Tetris puzzle. OK, we're given sequence of n pieces that will fall. For each of them, we must drop the piece from the top. OK, and if you're a fancy Tetris player, you can let a piece fall and then rotate it at very end to do some clever, clever thing. I disallow that. You always have to push the drop button. So the piece starts here, it goes instantly to

the ground. This will be necessary. I don't know how to solve the problem without this constraint.

OK, and then the other weird thing-- this is very weird for Tetris-- full rows normally clear, but now they don't clear. This is like hardcore Tetris. You're guaranteed to die eventually. The question is, can you survive these n pieces? That's the question. Can you survive? Oh, I've got one other constraint. This is actually kind of natural. The width of the board is small, relatively small, because we're going to be exponential in w. In real life it's 12, I think?

AUDIENCE: Ten.

PROFESSOR: Ten, sorry. It's been a while since I wrote my Tetris paper. So all right, these are all kind of weird constraints. If you don't make all of these constraints-- oh, also the board is initially empty. That's like level one of Tetris. If all of these things are not the case, which is regular Tetris, if you just have the first thing then this problem is called NP-complete. We'll be defining that next class. So it's computationally intractable.

But if you make all of these assumptions, the problem becomes easy, and you can do it by dynamic programming. So how do we do it? We define sub-problems just like before. The obvious thing to try is suffixes. How do we play a suffix of pieces i onwards? How to play those guys.

And just like fingering, this is not enough information, right? Because if we're going to play from pieces i onward, what we need to now is what the board currently looks like. I said here the board is initially empty. That's not going to be the case after you place the very first piece. So in general, after we've placed the first i pieces, we need to know what the board looks like.

And here's where I'm going to use all of these assumptions. If you always drop things from the top and rows don't clear, then all you really care about is how high each column is. This is what you might call the skyline of the board. OK, now in reality, there might be holes here because you drop things in silly ways. Maybe you

drop a piece like this.

And then I claim, because I'm dropping things from infinity from the sky, I really don't care about that there's a whole here. I can just fill that in and say, OK, that's my new skyline. Because if you can't do these last minute twists and if lines never clear, that's going to be gone. That material is wasted. OK, so all I need to remember is how high is each column.

So I should say given the board skyline. Now, how many choices are there for that? It's quite similar to this function, the fingering. Let's see. There's the height of the board, different choices. It's going to be h. For each column it could be anywhere between 0 and h, so I guess h plus 1 if you want to get technical. And then we raise it to the power w, because there's w different columns and each of them is independent choice. So this is going to n times that different sub-problems.

And here's what I need the is small because this is exponential in w. So it's reasonable in h, but exponential in w. OK, then what do I guess? Any suggestions what to guess?

AUDIENCE: So where the new piece falls, as in [INAUDIBLE]?

PROFESSOR: Yeah. What should I do with piece i? There's not that many choices. I can rotate it zero, one, two, or three times. I can choose someplace to drop it, but those are my only choices. So it's just how to play piece i. And given that guess, you can figure out how the skyline updates, like I did here. If I drop that piece like that, then I fill in this part and recompute my new skyline.

So it's going to be something like 4 times w different choices, roughly-- 4 for the rotation, w for the x-coordinate. And so the running time is just going to be the product of these. n times w times h plus 1 to the w. Open problem, if I drop any one of these assumptions, can you get a dynamic program that's reasonable? Could you do it if w is large? I don't know. Could you do if rows do clear? That's the least natural constraint here. I don't know. Puzzle for you to think about. I'd love to know the answer.

You can obviously do the rest of the steps, right? You can write down the recurrence. It's the same thing. You take the min over all guesses. What are we minimizing? Hmm. I guess here the question is survival. Can you survive?

So this is one of the first examples where the answer is a Boolean value, true or false. But if you think of true or false as 0 and 1, then it's still a maximization problem. You want to maximize. You want 1 if possible. Otherwise, you'll get 0 when you maximize. So you can write the recurrence using max. And in the base case, you have truth values, true or false. And you'll see, did I survive? Did I die? That sort of thing.

I want to go on to Super Mario Brothers, because everyone loves Super Mario Brothers. has? Anyone not played NES *Super Mario Brothers 1*? Aww, you got to play it, man. You're the only one. You can play it on an emulator. Maybe not legally, but you can play it on an emulator and just see how it is.

So what I'm going to talk about next, in theory, works for many different platform games, side-scrolling platform games. But *Super Mario Brothers 1* has some nice features. In particular, a nice feature is that whenever anything moves off of the screen, it disappears from the world. So the monster moves off, it's gone.

You can think of there's a static level there. When the level comes into screen, when a monster comes on screen, then it starts acting. But as soon as you move the screen-- you can't actually move backwards in *Super Mario 1,* but as soon as you move forwards and that character is offscreen, it's gone. So in a sense, that part of the level reset to its initial state. Now, as long as your screen is not too big-- and thankfully, on NES screens were not very big. It's 320p, or whatever. This will work.

If you are given the entire level-- so let's say there's n bits of information there-- and you have a small screen, w by h screen, w and h are not too big. Then I claim we can solve Super Mario Brothers by dynamic programming. So let's say we want to maximize our score. Want to run through the level and maximize your score, or you want to minimize the amount of time you use. You're doing level runs. Pick your favorite measure, all of those can be solved.

And the way to do it, this sort of general approach to all these DPs is we need to write down what do I need to know about the game state. I'll call that a configuration. What can we care about for Super Mario Brothers? Well, I guess everything on screen. This is a bit tricky, but there's stuff on screen. There are monsters and objects. For the monsters, I need to know their current position. For the objects, I need to know-- like, is there a question mark box? Did I hit it already? Did I already get the coin or did I already get the mushroom?

So for each of those things, there's some amount of information you need to store. How much information? I think something like constant to the w times h should do. That's saying for every pixel on the screen or for every square on the screen, however you-- whatever you define the resolution here to be. Let's say for every little unit square in Mario land, is it a brick? Is it a hard brick, or has it been a destroyed brick? Is a monster there right now? Is Mario there right now? All these kinds of information.

OK, so there's a cost number of choices for each pixel. You can write them all down. You might also want Mario's velocity. I had to play it again just to check that there is indeed velocity. Turning around is slower than going forward. You do accelerate a little bit. So you've got to remember that. There's probably only a constant number of choices for what your velocity is.

What else? Ah, I want to remember the score. You want to maximize score. And let's say you also-- how much time is left. There's a time counter. If it hits zero, you die. Now, these are kind of annoying, because they're integers. They could be kind of large. So I'm going to say the score could be capital S big, and time could be capital T big. So this'll be a pseudopolynomial algorithm.

The number of configurations in total here is the product of these things. It's exponential in w and h. And then multiply by S and T. So that's the number of configurations. And that's also going to be our sub-problem. I guess we should also write down where is the screen relative to the level. OK, how far to the right have you gone? That's another w. That's not a big deal.

OK, given this information, you know everything you need to know about playing from here on. And the time counter's always going to keep ticking. So you can draw a graph of all configurations, just enumerate all of these things. It's this many of them. And then draw, for every configuration, what are the possible things I can do? I could push this button. I can push the A button, I can release the A button. I can push the B button, I can release the B button. I can push the up arrow.

Those are all the things you could do. It's a constant number of choices. So each vertex will have constant out degree. If you did this, what configuration would I reach? Just draw that whole graph. Do shortest paths. Or dynamic programming, these are your sub-problems. There are no suffixes here. These are your sub-problem.

And then you take a max, if you're trying to maximize score or max if you're trying to maximize time, minimize the time you use. This is time remaining. And you can relate each sub-problem to a constant number of other sub-problems. So your running time will be this, because you only pay constant time per sub-problem. And now you can solve Super Mario Brothers optimally, as long as your screen is not too big and as long as your scores and times don't get too big either, because we're only pseudopolynomial with respect to S and T.

Questions? All right. That's-- yeah?

AUDIENCE: So are we going to be trying to memoize all of these possible configurations?

PROFESSOR: If you do the recursive version, you will end up memoizing all of these configuration values. Well, anyone that's reachable from the initial state. Some configurations might not be reachable, but the ones that are reachable you're going to start doing them. When you finish doing them, you will memoize the result.