PROFESSOR: All right, today we continue our exciting adventure into dynamic programming. Are you excited? I'm excited, super excited. Dynamic programming, as you recall way back before Thanksgiving, is a super exciting powerful technique to design algorithms, especially to solve optimization problems where you want to maximize or minimize something. Last time, we saw how two algorithms we already knew-- namely, how to compute the nth Fibonacci number and how to compute shortest paths via Bellman-Ford-- are really dynamic programs in disguise.

And indeed for, at least for Bellman-Ford, that's how they were invented, was to apply a general technique which we're going to see today in full generality, more or less-- most of this is generality-- in five easy steps. And we're going to see that technique applied to two new problems which are much more interesting than the ones we've already solved-- namely, how to make your text look nice in a paragraph, where to break the lines. That's text justification. And how to win and make loads of money at blackjack. So lots of practical stuff here, and we're going to see one new technique for general dynamic programming.

These are some things I wrote last time. Actually, one of them I didn't write last time. In general, you can think of dynamic programming as a carefully executed brute force search. So in some sense, your algorithm is going to be trying all the possibilities, but somehow avoiding the fact that there are exponentially many of them. By thinking of it in a clever way, you can reduce the exponential search space down to a polynomial one, even though you're still not being very intelligent you're still blindly trying all possibilities. So that's the brute force part.

In more detail, the three main techniques in dynamic programming are the idea of guessing, the idea that, oh, I want to find the best way to solve a problem. Let's pick

out some feature of the solution that I want to know. I don't know it, so I'll guess the answer-- meaning I'll try all the possibilities for that choice and take the best one. So guessing is really central to dynamic programming.

Then we also use a recursion, some way to express the solution to our problem in terms of solutions to sub-problems. So it's usually very easy to get a recursion for a lot of problems as long as they have some kind of substructure. Like shortest paths, we had that some paths of shortest paths were also shortest paths, so that was handy. Usually the recursion by itself is exponential time, like even with Fibonacci numbers.

But we add in this technique of memoization, which is just once we compute an answer we've stored in a lookup table, if we ever need that answer again we reuse it instead of recomputing it. So we store it. We write down in our memo pad anything that we compute. Those techniques, all these techniques together give you, typically, a polynomial time dynamic program-- when they work, of course. Memoization makes the recursion polynomial time. The guessing is what is doing a brute force search. And magically, it all works if you're careful.

Another perspective-- kind of an orthogonal perspective or another way of thinking about it, which I think should be comfortable for you because we spent a lot of time doing shortest paths and expressing problems that we care about in terms of shortest paths even if they don't look like it at first glance-- dynamic programming in some sense is always computing shortest paths in a DAG. So you have some problem you want to solve, like you have text you want to split up into lines so it looks nice in a paragraph, you express that problem somehow as a directed acyclic graph.

And then we know how to compute shortest path in directed acyclic graphs in linear time. And that's basically what dynamic programming is doing. I didn't realize this until last week, so this is a new perspective. It's an experimental perspective. But I think it's helpful. It's actually-- dynamic programming is not that new. It's all about how to be clever in setting up that DAG. But in the end, the algorithm is very simple.

And then we had this other perspective-- back to this perspective, I guess. In general, we have-- the real problem we want to solve, we generalize it in some sense by considering lots of different sub-problems that we might care about. Like with Fibonacci, we had the nth Fibonacci number. We really just wanted the nth Fibonacci number. But along the way, we're going to compute all f1 up to fn.

So those are our sub-problems. And if we compute the amount of time we need to solve each sub-problem and multiply that by the number of sub-problems we get, the total time required by the algorithm. This is a general true fact. And the fun part here is we get to treat any recursive calls in this recursion as free, as constant time, because we really only pay for it first time. That's counted out here. The second time we call it, it's already memoized, so we don't have to pay for it.

So this is, in some sense, an amortization, if you remember amortization from table doubling. We're just changing around when we count the cost of each sub-problem, and then this is the total running time. OK, so that's the spirit we saw already. I'm going to give you the five general steps, and then we're going to apply them to two new problems.

So five easy steps to dynamic programming. Unfortunately, these are not necessarily sequential steps. They're a little bit interdependent, and so "easy" should be in quotes. This is how you would express a dynamic program, and in some sense how you'd invent one, but in particular how you would explain one.

OK, let me get to the main steps first. First step is to figure out what your sub-problems are going to be. Second part is to guess something. Third step is to relate sub-problem solutions, usually with a recurrence. I guess always with a recurrence. Fourth step is to actually build an algorithm.

And we saw two ways to do that last time. One is to use recursion and memoization, which is the way I like to think about it. But if you prefer, you can follow the bottom up approach. And usually that's called building a table. And that one's basically to turn our recursion and memoization, which is kind of fancy, into a bunch of for loops, which is pretty simple. And this is going to be more practical, faster, and so

on. And depending on your preference, one of them is more intuitive than the other. It doesn't matter. They have the same running time, more or less, in the worst case.

Then the fifth step is to solve the original problem. All right, so we've sort of seen this before. In fact I have, over here, a convenient table. It's called cheating. The two problems we saw last time, Fibonacci numbers and shortest paths. And I've got steps one, two, three, four-- I ran out of room, so I didn't write five yet. But we'll get there.

So what are our sub-problems? Well, for Fibonacci, they were f1 through fn. So there were n different sub-problems. And in general because of this formula, we want to count how many sub-problems are there. So number of sub-problems is-- this is what we need to do algorithmically. And then for analysis, we want to counter number of sub-problems for step one. And so for Fibonacci there were n of them.

For shortest paths, we defined this delta sub k of sv. This was the shortest path from s to v they uses at most k edges. That was sort of what Bellman-Ford was doing. And the number of different sub-problems here was v squared, because we had to do this for every vertex v and we had to do it for every value of k between 0 and v minus 1. v minus was is the number of rounds we need in Bellman-Ford. So it's v times v, different sub-problems, b squared of them.

OK, second thing was we wanted to solve our problem. And we do that by guessing some feature of the solution. In Fibonacci, there was no guessing. So the number of different choices for your guess is one. There's nothing. There's only one choice, which is to do nothing.

And for shortest paths, what we guessed was-- we know we're looking for some path from s v. B Let's guess what the last edge is. There's some last edge from u to v, assuming the path has more than one edge-- or more than zero edges. When could the edge possibly be? Well, it's some incoming edge to v. So there's going to be indegree of v different choices for that. And to account for the case that that's zero, we do a plus 1. But that's not a big deal.

So that was the number of different choices. In general if we're going to guess something, we need to write down the number of choices. For the guess, how many different possibilities are there? That's our analysis.

OK, the next thing is the recurrence. That's step three. We want to relate all the sub-problem solutions to each other. For Fibonacci, that's the definition of Fibonacci numbers. So it's really easy. For shortest paths, we wrote this min. In general, typically it's a min or a max, whatever you're trying to solve here. We're doing shortest paths. You could do longest paths in the same way.

So you compute them in of delta sub sk minus 1 of su. The idea is we want to compute this part of the path, the s to u part. And we know that has one fewer edge, because we just guessed what the last edge was. Except we don't really know what the last edge was, so we have to try them all. We try all the incoming edges into v-- that's this part-- and for each of them we compute-- I forgot something here. This is the cost of the first part of the path. Then I also need to do plus the weight of the uv edge. That will be the total cost of that path.

You add those up, you do it for every incoming edge. That is, in some sense, considering all possible paths. Assuming you find the shortest path from s to u, that's going to be the best way to get there. And then use some edge from u to v for some choice of u. This will try all of them. So it's really trying all the possibilities. So it's pretty clear this is correct if there are no negative weight cycles. You have to prove some things. We've already proved them.

It's just slow, but once you add memoization, it's fast. Now, how long does it take to evaluate this recurrence, constant time, if you don't count the recursive calls or count them as constant? Over here, we're taking a min over n degree of v things. So we have to pay n degree of v time, again the recursions as free. But for each one of them, we have to do an addition. So it's constant work per guess.

And this is quite common. Often, the number of guesses and the running time per sub-problem are the same, the constant factors. Sometimes they're different. We'll see some examples today. OK, step four. Let's see. So here we evaluate the time

per sub-problem. Once you have the recurrence, that becomes clear. You want to make sure that's polynomial. Often these are the same.

And then we add the recursive memorize or build a DP table. I'm not going to write those. We did it for Fibonacci last time, shortest paths. Pretty easy. And in general, what we need to check here is that the sub problem recurrence is acyclic. In other words, that it has a topological order so we can use topological sort. We don't actually use topological algorithm usually. You can just think about it.

In the case of Fibonacci numbers, it's clear you want to start with the smallest one and end up with the biggest one. You can't do the reverse, because then when you're trying to computer the nth you don't have the ones you need, the n minus 1 and n minus 2. But if you do it in this order, you always have the one you need by the time you get there.

In general, there's a DAG there-- and for Fibonacci, it was like this. Every node depends on the previous and the second previous. But you just choose a topological order, which is here left to right, and you're golden. And these are actually the for loops you get in the bottom of DP.

For shortest paths, you have to think a little bit. You have to do the for loop over k on the outside, the for loop over V on the inside. The reverse does not work. I won't go through that, but we drew the DAG last time. And that's the main thing you need to do here. And then, of course, you use this formula to compute the overall running time, which is just multiplying this quantity with this quantity. Total time.

Then there's just one last step that usually isn't that big a deal, but you have think about it. You need to make sure that the problem you actually cared about solving gets solved. In the case of Fibonacci and shortest paths, this is pretty clear. I didn't write it. We can do it on here. Solve the original problem. Fibonaci, it is Fn. And this is one of our sub-problems, so if we solve all of them, we're done.

For shortest paths, it's basically delta sub v minus 1 of sv for all v. That's single source shortest paths. And by our Bellman-Ford analysis, that gives us the right

shortest paths. There are no negative weight cycles.

And sometimes this requires extra time to combine your solutions to get the real thing. Here of course, we just have the answers, so writing them down does not take very long. So that's the dominant running time-- which I didn't write, I should have written in under for here-- this ends up being n, this ends up being VE. OK, I don't want to spend more time on those examples. Let's go to new things.

So first problem we're going to look at today is text justification. And the informal statement of this problem is you're given some text-- which means a string, a whole bunch of characters. And we want to split them into good lines. The rules of the game here are we're going to, like in the early lectures of document distance where you have some definition of splitting a document into words separated by spaces.

And what we want to do is cut. We can only cut between word boundaries. And we want to write some text, it's going to have some spaces in it. Then there's a new line, something like that. And we want to justify our text on the right here. And so we'd like to avoid big gaps like this because they look ugly, they're hard to read.

Now, if you use Microsoft Word-- at least before the latest versions-- they follow a greedy strategy, which is very simple. You pack as many words as you can on the first line, then you go to the next line, pack as many words as you can on the second line. Keep going like that. And that strategy is not optimal. If you use LaTeX-- as some of you have been doing on problem sets, and I think also new versions of Word but I'm not sure-- then it uses dynamic programming to solve this problem. And that's what we're going to do here.

So let me specify a little bit more about what we mean here. So the text we're going to think of as a list of words. And we're going to define a quantity badness. And this is an anesthetic quantity, if you will. I'm going to tell you what LaTeX uses. But this is sort of how bad it is to use-- or let's say, yeah, words i through j as a line.

So this is python notation. So it starts at i and ends at J minus 1. That'll be convenient. So I have this list of words. And if I look at words i through j minus 1 and

I think of what happens if I pack them in a line, well, they may fit or they may not fit. So there are going to be two cases. If they don't fit, I'm going to write infinity. So that's really bad.

So I have some notion of how wide my line can be. And if the sum of the lengths of those words plus the sum of the lengths of the spaces as small as possible is bigger than the width of my screen-- or page, I guess-- then I say they don't fit, and then I define badness to be infinity-- meaning, I never want to do that. This is actually LaTeX sloppy mode, if you want to be technical.

Otherwise, it's going to be page width minus total width cubed. Why cubed? Who knows. This is the LaTeX rule. And squared would probably also be fine. So this is the width of the page minus the total width of those words, which you also have to include the spaces here. You take the difference. You cube it. And so when this is small-- I mean, when these are very close-- then this is going to be close to zero. That's good. That means you use most of the line.

When the total width is much smaller than the page width, then this will be a large value. You cube it, it will be even larger. So this will highly discourage big gaps like this. And it will very much discourage not fitting. So there's a tradeoff, of course.

And the idea is you might-- in the greedy algorithm, you make the first line as good as you can. But it might actually be better to leave out some of the words that would fit here in order to make the next line better. In general, it's hard to tell, where should I cut the lines in order to get the best overall strategy? What I'd like to minimize is the sum of the badnesses of the lines. So it's a sum of cubes, and that's really hard to think about.

But that's what dynamic programming is for. You don't have to think. It's great because it's brute force. OK, so the first thing we need to do is define sub-problems. This is, in some sense, the hard part. The rest will follow easily. So I think actually it might be easier to think about, for this problem, what would be the brute force strategy? How would you try all possibilities, exponential time? Suggestions? Yeah?

AUDIENCE: Try all partitions of the words that don't fit?

PROFESSOR: Try all partitions of the word, so-- of the string of words. So I mean, it could be it all fits in on one line. It could be it's split into two lines. I try all possible splits there. In general, I'm guessing for every word, does this start a line or not? That would be all ways. And so there are 2 to the n. If I have n words, there's 2 to the n different splits. For every word I say yes or no, does this is begin a line?

So what I'd like to figure out is where those lines begin. That was the point of that exercise. So any suggestions? Maybe it's actually easier to jump ahead and think, what would I guess in my solution if I have this big string of words? What's the natural first thing to guess? Yeah?

AUDIENCE: Guess how long the first line is?

PROFESSOR: Guess how long the first line is, yeah. We know that the first word begins a line. But where does the second line begin? So I'd like to guess where the second line begins. That's-- so you know, I have the beginning of a line here and then I have a beginning of a line here at the fourth word. Where does the second line begin? I don't know. Guess.

So I'm going to try all the possible words after the first word. And say, well, what if I started my second line here? At some point I'm going to be packing too much into the first line, and so I abort. But I'll try them all. Why not? OK, that's good.

The issue is that once I've chosen where the second line is, of course the next thing I want to guess is where the third line begins. And then I want I guess where the fourth line begins, and so on. In general, I need to set up my sub-problems so that after I do the first guess I have the problem of the original type. So originally I have all the words. But after I guess where the second line begins, I have the remaining words.

What's a good word for the remaining words? If I give you a list of words and I want from here on, it's called-- what? A sub-problem, yes. That's what we want to define. It's called a suffix of the array. That's the word I was looking for. It's tough when I

only have one word answers.

So my sub-problems are going to be suffixes. Which is, in python notation, i colon. They call it splices. And how many sub-problems are there if I have n words? Two? Sorry?

AUDIENCE: 2 to the n.

PROFESSOR: 2 the n? That would be a problem if it's 2 to the n. I hope it's only n. Originally, we said, OK, for every word, we're going to say, is this in our out? Is this the beginning or not? That's 2 to the n. But here, the idea is we're only thinking about, well, what are the words that remain? And it could be you've dealt with the first 100 words and then you've got n minus 100 left, or it could be you've dealt with the first thousand words and you've got n minus 1,000. There's only n choices for that.

We're only remembering one line, this is the key. Even though we may have already guessed several lines, we're just going to remember, well, OK. This is what we have left to do. So let's forget about the past. This is what makes dynamic programming efficient. And we're just going to solve it, solve these sub-problems, forgetting about the past.

So the sub-problem-- I'm not going to write it here-- is if I give you these words, never mind the other words, how do I pack them optimally into a paragraph? I don't care about the other words, just these words. So this is a different version of the same problem. Initially, we have n words to do. Now I have n minus i words to do. But it's again text justification. I want to solve this problem on those words. That's just how I'm going to define it.

This will work if I can specify a recurrence relation. As we said, what we guess is where to break the first line, where to start the second line for those words. OK, so this is-- it could be the i plus first line. It could be the i plus second line-- or sorry, word. Some word after i is where we guess the second word. The number of choices for the guess is at most n minus i. I'm just going to think of that as order n. It won't matter.

The third part is we need a recurrence relation. I claim this is very easy. I'm going to-- I didn't give this problem a name, so I'm just going to write it as DP of i. So this is going to be the solution to that suffix, words from i onward. And I'd like to-- what I want to do is consider all possible guesses. So I mean this is going to be pretty formulaic at this point. After I've set up these ideas there's pretty much only one thing I can write here, which is I want to do a for loop. That would be the for loop of where the second line can start.

I can't start at i, because that's where the first line starts. But it could start at i plus 1. And this special value of n will mean that there is no second line. OK, so DP of i-- now I want to do this for loop in order to try all the possible guesses. j will be the word where the next thing starts. So then what do I write up here? If I make this guess-- all right, so I have word i is the first word of the first line. And then word j is the first word of the second line.

And then there's more stuff down below. I don't know what that is. But how can I use recursion to specify this? DP of j, exactly. I guess if I'm doing recursion, I should use parentheses instead of brackets. But if you're doing it bottom up, it would be square brackets. So that's just DP of j. That's the cost of the rest of the problem.

And I can assume that that's free to compute. This is the magic of dynamic programming. But then I also have to think about, well, what about the first line? How much does that cost? Well, that's just badness of ij. And we've already defined that. We can compute it in constant time. Dynamic programming doesn't really care what this is. It could be anything. As long as you're trying to minimize the sum of the badnesses, whatever function is in here, we just compute it here.

That's the power of dynamic programming. It works for all variations of this problem, however you define badness. So you might say, oh, that's a weird definition. I want to use something else instead. That's fine, as long as you can compute it in terms of just i and j and looking at those words.

OK, now I need to do a min over the whole thing. So I want to minimize the sum of the badnesses. So I compute for every guess of j, I compute the cost of the rest of

the problem plus the cost of that first line. And this, is in some sense, checking all possible solutions magically. OK. That's the recurrence.

We need to check some things. I guess right now we just want to compute how much time does this cost, time per sub-problem. To do this for loop, basically I do constant work-- all of this is constant work-- for each choice. So there's order n choices, so this is order n. Now we have to check that there's a topological order for this problem or for these sub-problems.

And this is easy, but a little different from what we've done before because we have to actually work from the end backwards, because we're expressing DP of i in terms of DP of larger values of i. j is always bigger than i. And so we have to do it from the right end back to the beginning. And n minus 1 down to 0. I didn't actually define DP of n. There's a base case here which is DP of n equals 0.

Because the meaning of DP of n is I have zero words, the nth word onward. There is no nth word. It's 0 to n minus 1 in this notation. So I don't pay anything for a blank line. OK, so that's our top logical order. This one, of course, is instantaneous. And then we work backwards. And always whenever we need to compute something, we already have the value.

The total time we get is going to be the number of sub problems-- which is n times the running time per sub-problem. which is order n, which is order n squared. And in the worst case, it is indeed theta n squared. Although in practice it's going to work better, because lines can't be too long. So that's the running time.

Then finally we have to check that the original problem actually gets solved. And in this case, the original problem we need to solve is DP of 0 because DP of 0 means I take words from 0 onwards. That's everybody. So that's the actual problem I want to solve. So we work backwards. We solve all these sub-problems that we don't directly care about, but then the first one is the one we want. And we're done.

So in quadratic time, we can find the best way to pack words into lines. Question?

AUDIENCE: [INAUDIBLE]

PROFESSOR: DP of j is returning. It's like this. So DP of-- this is a recursive definition. Imagine this is a recursive function. I wrote equals, which is Haskell notation, if you will. But normally, you think of this as like def DP of i is return min of this. This is python. So it's returning the cost. What was the best way to pack those lines from j onwards? That's what DP of j returns.

So it's a number. It's going to be a sum of badness values. Then we add on one new badness value. It's still a sum of badness values. We return the best one that we find. Now, this does not actually pack the words. That's a good-- maybe your implicit question. It's not telling you how to pack the words. It's telling you how much it costs to pack the words.

This is a lot like shortest paths where we didn't-- it was annoying to actually figure out what the shortest path was. Not that annoying, but that's not what we were usually aiming to do. We were just trying to figure out the shortest path weight. And then once we knew the shortest path weight, it was pretty easy to reconstruct the paths.

So maybe I'll take a little diversion to that and talk about parent pointers. The idea with parent pointers is just remember which guess was best. it's a very simple idea, but it applies to all dynamic programs and lets you find the actual solution, not just the cost of the solution.

We did the same thing with shortest paths. We even called them parent. So when we compute this min, were trying all choices of j. One of them-- or maybe more than one, but at least one of them actually gave you the min. That's usually called the arg min in mathematics. It's what was the value of j that gave you the minimum value of this thing. So I mean, when you compute the min, you're iterating over every single one. Just keep track of which one was the best.

That's it. Call that the parent pointer. Do I need to write that? Here, parent-- parent of i is going to be arg min of that same thing. So it's a j value. It's the best j value for

i. And so we store that for each i. It cost no more work, just a constant factor more work than computing the min. We also write down the arg min.

So we're already storing the min in the DP table. DP of i would get sorted to be that. We also store parent of i. And then once we're done, we start with our original problem and we follow parent pointers to figure out what the best choices were. So we start at 0 because we know word zero begins a line. And then 0 will be the first line. Then we go to parent of 0. That will be where the second line begins.

Then we go to parent of parent of 0. That will be where the third line begins. OK, because these were the best choices for where the second line begins, this is the best place where the second line begins. Given that this is the first line, this is the best line where the second line begins given that this was the first line. So that's really the third line given this was the second line.

Little confusing, but you just a simple for loop. You start with 0 because that's our original problem. You keep calling parent of the thing you currently have. In linear time, you will reconstruct where the lines break. So you can use this technique in any DP. It's very simple. It's totally automatic. Just like memoization is a technique that you can apply without thinking, you could even write a program, given a recursive algorithm, would turn into a memorized recursive algorithm. It's totally automated.

Same thing with the bottom up DP table. As long as you know what the topological order is, just make those for loops and then put exactly the recursive call but turn it into an array call. Boom, you've got a bottom up algorithm. Totally automatic, no thinking required. Parent pointers also, no thinking required. As long as you're following the structure of trial guesses compute some value-- just remember what the guess was-- you reconstruct your solution. That's the great thing about dynamic programming is how much of it is automatic.

The hard part is figuring out what to guess and then what your sub-problems are, or the other order. Whatever works. Any other questions about text? I would like to move on to blackjack. OK, now I brought some cards, because some of you may

not know the rules to blackjack. How many people know blackjack? OK. How many people do not and are willing to admit it? A few, all right. So this is for you and for fun, entertainment.

So I'm going to bring Victor up to help demonstrate the rules of blackjack. We're going to play standard Casino blackjack as in the movie *21*, or whatever. So I'm going to just do a random cut here so I can't sheet. You have a tablet, that's scary. You're going to look at strategy.

VICTOR: Nothing special.

PROFESSOR: All right. Hopefully you do not have x-ray vision. So the way it works is there's a dealer player and one or more players. We're just going to do it with one player to keep it simple. I'm going to be the dealer. So my strategy is actually totally deterministic, there's nothing interesting. Victor has the hard part of winning.

So to start out, I believe we deal to you first, then to me, then to you, then to me. So let's hold up these cards, Victor, so that people can see them. You don't get to see one of my cards. That's some peculiarity of the rule. And if the sum of our cards goes over 21, we lose the game. Victor first. I cannot have a value more than 21 in these hands, because I only have two cards.

You have a value of-- ha, ace. Great. An ace can be a 1 or an 11. That's the fun rule. So this is either an 8 or an 18. And so Victor has a choice of whether to take another card or not. What would you like to do?

VICTOR: Standard strategy says stand.

PROFESSOR: He stands. So he's going to stick to that. At this point, my cards flip over. I have 17, which is same you, which I believe means-- I forget about tie rules.

VICTOR: I have 18.

PROFESSOR: You have 18. All right.

VICTOR: See? The strategy works.

PROFESSOR: So that's good. I'm going to hit in the hope that I have a small card that will push me right above you. But I do not. I lose. I'm sad.

VICTOR: It says always stand on a 17.

PROFESSOR: Oh, always stand on 17? Huh. All right, never mind. Thanks. Yeah, I still lose. The game is over. My strategy is always stand on a value--

VICTOR: Stand on 17.

PROFESSOR: 17 or higher. And if I have a value less than 17, I always take another card. So let's do it one more time to get it right. So I'm going to deal to you, deal to me, deal to you, deal to me. So hold up your cards. You have 18 again. Are you cheating?

VICTOR: I still have to stand.

PROFESSOR: You still stand, according to tablet. So I, in this case, have a 20. And so this I win. So you get the idea. Let's say in each case we're betting $1. So at this point, we'd be even. He won $1, I won $1. But in general, slight-- I think it's balanced.

VICTOR: For these rules, there's a 1% advantage for the house.

PROFESSOR: 1% advantage for the house. Interesting. All right, well, that's beyond this class. What we're going to see is how to cheat in blackjack. So this is going to be-- I encourage you to try this out at casinos. Just kidding. This is a little bit difficult to actually do in a casino unless you have an inside man. So if you have an inside man, go for it. It's guaranteed to win you lots of money because it's going to play optimally.

In perfect information blackjack, I suppose that I already know the entire deck. Suppose somehow either I get to put the deck there, or I have some x-ray vision. I get to see the entire deck ahead of time. And then somebody's going to play through a game over and over with me-- or not over and over, but until the deck is depleted-- and I want to know in each case, should I hit, or should I stand? And I claim with dynamic programming you can figure that out-- using exactly the same

strategy as text, actually.

It's really for each word, should I start a new line or not? Same problem here. It's slightly more complicated to write down. So let's say the deck is a sequence of cards. And I'm going to call it c0, c1 up to cn minus 1, n cards. And you are one player. First is the dealer.

I don't know how to solve this for two players, interesting open problem. But for one player I can do it. Let's say $1 bet per hand, I think they're called. I'm not sure. Per play? Per box? Whatever. You're not allowed to double. You're not allowed to split. All these fancy rules are harder to think about, although you might be able to solve them as well.

So the idea is I have some cards. Should I hit or should I stand? I don't know. I'll guess. So our guessing-- let's jump ahead to the guessing part-- is whether we want to hit or stand given a card. Actually, it would be easier to think about an entire play, an entire hand. We're going to guess, how many times should I hit in the first play?

So initially, four cards are dealt. I look at my hands. Actually, I don't really look at my hand. I'm just going to guess ahead of time. I think I'll hit five times this time. I think I'll hit zero times this time. I mean, I'm just going to try them all. So I don't really have to be intelligent here, OK? It's kind of crazy but it works.

Our sub-problems, can anyone tell me what our sub-problems would be, In one word or less? Less would be impressive. Yeah?

AUDIENCE: Where you start the new hand.

PROFESSOR: Where do you start the new hand? Yeah. So it's going to be suffixes of the cards. So at some point we do a play, and then we get to ith card. And then the rest of the game will be from the ith card on. So it's going to be suffix ci colon, I guess would be the notation here. It's a bit awkward. These are the cards that remain.

And so the sub-problem is, what is the best play? What's the best outcome given $1 bets? How much money can I make-- maximize my winning, say-- given these cards

onward? Who knows what happened to their earlier cards, but just these are the cards. I'm left with. Number of sub-problems is-- hmm? n. How many choices of i are there? n choices.

This really important. It's really useful that we're thinking about suffixes. It's not that some subset of the cards have been played. That would be really hard, because there's exponentially many different subsets that could be left. It's always a prefix that gets played, and therefore suffix is left. And there's only n suffixes, remember that. We're going to use it over and over in dynamic programming.

So now we need to solve the sub-problem. Starting from ci, what's the best way to play? Well, the first four cards are fixed, and then we guess how many hits are left. So it's going to be something like n minus i minus four different possibilities for-- I mean, that would be the maximum number of hits I could take all the remaining cards. That would be the most. And let's see, so the number of choices-- I'll just say it's, at most, n. I don't have to be fancy here.

OK, now we go to the recurrence. So I'm going to call this blackjack of i. It's going to be the solution. I want to solve this sub-problem from i onwards. What's the best play? And I guess it's going to be a max if I'm measuring winnings. And what's the winnings if I decide to hit this many times? It's a little bit hard to write down the exact formula. I'm going to write a rough version which is the outcome of that first play. It's going to be either I lose $1, we tie, or I win $1.

So if we end up with the same value, you actually-- in most versions-- you get your money back, nothing changes. The bet is nullified. So that's a zero outcome. But if we're only betting $1, these are the three possible outcomes. You can compute this, right? If I told you how many times you hit, then you just execute through those cards and you compute the values of my hand, of your hand versus the dealer's hand. You see, did anyone bust? If so, they lose. Otherwise you compare the values and you see which is bigger or smaller.

This is easy to do in linear time. No biggie. What's useful here is that the dealer strategy is deterministic. So after you know how many cards you take, what the

dealer does is force, because he just looks. Do I have 17 or greater? If not, take another card and keep repeating that. So it's a deterministic strategy. In linear time, you can figure out what the outcome is.

Then you also have to add the outcome of all the remaining cards, which is just BG of j. This is recursion, super easy. We do this for all choices of j. It's like a range of i plus 4 up to n, I think. Sure, that'll work. I should probably put an if here, which is if it's a valid play.

There are some constraints here. If I've already busted, I can't hit again. So in fact what you have to do in this for loop is say, well, maybe I take another hit. Maybe I take another hit. At some point I go over 21, and then you have to stop the for loop. So I'm writing that as an if. You can also do it with a break, however you want. But that's-- you're considering all possible options, all valid options of play.

For each of them, you see what the outcome was after the dealer takes some more cards. This is actually a little bit funny. Sorry, this should really be the number of hits in range from, let's say, 0 to n. Maybe you don't hit at all. And then j is a little bit tricky, because this is actually i plus 4 plus the number of hits plus the number of dealer hits. OK, so you have to run this algorithm to compute what happened, which computes how many times a dealer took a card. That's how many cards got consumed. And so that's-- if you do i plus 4 plus that plus that, that's how many cards are left, or where the cards resume. And then you call BG on that.

So we're, in general, from BG of i-- if you think of the DAG-- there's some position, maybe i plus 4 happens. Maybe it doesn't happen. It depends on what the dealer does. We're going to depend on i plus 6, i plus 5 maybe. It's going to be all of these possibilities. These are all different plays. And then on each of these edges, we're going to have plus 1, 0, or minus 1. Those are the outcomes, whether I won or lost or tied.

And then we're just computing a shortest path in this DAG. It's actually really easy if you think about it that way. This is just how many cards are left. From that position, you just see what are all the possibilities? What are all the edges that I could go to?

What states could I to go to next? How many cards are remaining? How much did it cost me or win me? And then take longest paths in that DAG.

That will give you the exact same answer. That's what this dynamic programming is doing. In the lecture notes, there's more details where I actually tried to write out this function, this recurrence as an algorithm. You could do it, assuming I've got everything right. It's not that hard. The order here is just the same as the order we did before. The running time is going to be cubic in the worst case, because we have-- it's a little non-obvious, but we have n sub-problems. For each of them, we have n choices. And for each choice we have to run the dealer strategy.

And so that conceivably could take linear time. Here I'm assuming a general value of 21. If 21 is actually constant, it only be constant time to play out a single hand, and then it's quadratic time. So it depends on your model of generalized blackjack. But that's it. And get some flavor of the power of dynamic programming, we're going to see it's even more powerful than this in the next two lectures.