PROFESSOR: Today we're going to solve three problems, a problem called Parenthesization, a problem called Edit Distance, which is used in practice a lot, for things like comparing two strings of DNA, and a problem called Knapsack, just about how to pack your bags. And we're going to get a couple of general ideas, one is about how to deal with string problems in general with dynamic programming. The first two and our previous lecture are all about strings, certain sense or sequences, and we're going to introduce a new concept, kind of like polynomial time, but only kind of, sort of-- pseudo polynomial time.

Remember, dynamic programming in five easy steps. You define what your sub problems are and count how many there are, to solve a sub problem, you guess some part of the solution, where there's not too many different possibilities for that guess. You count them, better be polynomial. Then you, using that guess-- this is sort of optional, but I think it's a useful way to think about things. You write a recurrence relating the solution to the subproblem you want to solve, in terms of smaller subproblem, something that you already know how to solve, but it's got to be within this list. And when you do that, you're going to get a min or a max of a bunch of options, those correspond to your guesses. And you get some running time, in order to compute that recurrence, ignoring the recursion, that's time for subproblem. Then, to make a dynamic program, you either just make that a recursive algorithm and memoize everything, or you write the bottom up version of the DP. They do exactly the same computations, more or less, and you need to check that this recurrence is acyclic, that you never end up depending on yourself, otherwise these will be infinite algorithms or incorrect algorithms. Either way is bad. From the bottom up, you really like to explicitly know a topological order on the subproblems, and that's usually pretty easy, but you've got make sure that it's

acyclic. And then, to compute the running time of the algorithm, you just take the number of subproblems from part 1 and you multiply it by the time it takes per subproblem, ignoring recursion, in part 3. That gives you your running time. I've written this formula by now three times are more, remember it. We use it all the time. And then you need to double check that you can actually solve the original problem you cared about, either it was one of your subproblems or a combination of them. So that's what we're going to do three times today.

One of the hardest parts in dynamic programming is step 1, defining your subproblems. Usually if you do that right, it becomes-- with some practice, step 2 is pretty easy. Step 1 is really where most of the insight comes in, and step 3 is usually trivial, once you know 1 and 2. Once you realize 1 and 2 will work, the recurrence is clear. So I want to give you some general tips for step 1, how to choose subproblems, and we're going to start with problems that involve strings or sequences as input, where the problem, the input to the problem is string or sequence. Last class we saw text justification, where the input was a sequence of words, and we saw Blackjack, where the input was a sequence of cards. Both of these are examples, and if you look at them, in both cases we used suffixes, what do I call it, x, as our subproblems. If x was our sequence, we did all the suffixes, I equals zero up to the length of the thing. So they're about n, n plus 1, such subproblems. This is good. Not very many of them, and usually if you're plucking things off the beginning of the string or of the sequence, then you'll be left with the suffix. If you always are plucking from the beginning, you always have suffixes, you'll stay in this class, and that's good, because you always want a recurrence that relates, in terms of the same subproblems that you know. Sometimes it doesn't work. Sometimes prefixes are more convenient. These are usually pretty much identical, but if you're plucking off from the end instead of the beginning, you'll end up with prefixes, not suffixes. Both of these have linear size, so they're good news, quite efficient. Another possibility when that doesn't work, we're going to see an example of that today, is you do all substrings. So I don't mean subsequences, they have to be consecutive substrains, i through j. And now for all i and j. How many of these are there? For a string of length n? N squared. So this one is n squared, the

others are linear. Out of room here. Theta n. So you obviously you prefer to use these subproblems because there's fewer of them, but if sometimes they don't work, then use this one, still polynomial, still pretty good. This will get you through most DP's. It's pretty simple, but very useful.

Let me define the next problem we consider. For each of them we're going to go through the five steps. So the first problem for today is parenthesization. You're given an associative expression, and you want to evaluate it in some order. So I'm going to-- for associative expression, I'm going to think of matrix multiplication, and I probably want to start at zero. So let's say you have n matrices, you want to compute their product. So you remember matrix multiplication is not commutative, I can't reorder these things. All I can do is, if I want to do it by sequence of pairwise multiplications, is I get to choose where the parentheses are, and do whatever I want for the parentheses, because it's associative. It doesn't matter where they go. Now it turns out if you use straightforward matrix multiplication, really any algorithm for matrix multiplication, it matters how you parenthesize. Some will be cheaper than others, and we can use dynamic programming to find out which is best. So let me draw a simple example.

Suppose I have a column vector times a row vector times a column vector. And there are two ways to compute this product. One is like this, and the other is like this. If I compute the product this way, it's every row times every column, and then every row times every column, and every row times every column. This subresult is a square matrix, so if these are-- say everything here is n, and this will be an n by n matrix.

Then we multiply it by a vector and this computation has to take, if you do it well, it will take theta n squared time, because I need to compute n squared values here, and then it's n squared to do this final multiplication. Versus if I do it this way, I take all the rows here, multiply them on all the columns here, it's a single number, and then I multiply by this column. This will take linear time. So this is better parenthesization than this one. Now, I don't even need to define in general for an x by y matrix, times a y by z matrix, you can think about the running time of that

multiplication. Whatever the running time is, dynamic programming can solve this problem, as long as it only depends on the dimensions of the matrices that you're multiplying.

So for this problem, there's going to be the issue of which subproblems we use. Now we have a sequence of matrices here, so we naturally think of these as subproblems, but before we get to the subproblems, let me ask you, what you think you should guess? Let's just say from the outset, if I give you this entire sequence, what feature of the solution of the optimal solution would you like to guess? Can't know the whole solution, because there's exponentially many ways to parenthesize. What's one piece of it that you'd like to guess that will make progress? Any idea? It's not so easy.

AUDIENCE: Well, wouldn't you need the last operation?

PROFESSOR: What's the last operation we're going to do, exactly. You might call it the outermost multiplication or the last multiplication. So that's going to look like we somehow multiply a 0 through ak minus 1, and then we somehow multiply aK through an minus 1, and this is the last one. So now we have two subproblems. Somehow we want to multiply this, somehow-- I mean, there's got to be some last thing you do. I don't know what it is, so just guess it. Try all possibilities for k, it's got to be one of them, take the best. If somehow we know the optimal way to do a0 to k minus 1 and the optimal way to ak to an minus 1, then we're golden. Now, this looks like a prefix, this looks like a suffix. So do you think we can just combine subproblems, suffixes and prefixes? How many people think yes? A few? How many people think no, OK, why?

AUDIENCE: So, for example if you split, if you were to split, like [INAUDIBLE]?

PROFESSOR: Yeah. The very next thing we're going to do is recurse on this subproblem, recurse on this subproblem. When we recurse here, we're going to split it into a0 to ak prime minus 1, and ak prime minus 1, or ak prime to ak minus 1. We're going to consider all possible partitions, and this thing, from ak prime to ak minus 1, is not a prefix or a suffix. What is it? A substring. There's only one thing left. I claim these are usually

enough, and in this case substrings will be enough.

But this is how you can figure out that, ah, I'm not staying within the family prefixes, I'm not staying within the family suffixes. In general, you never use both of these together. If you're going to need both, you probably need substrings. So if just suffixes work, fine. If just prefixes work, fine, but otherwise you're probably going to need substrings. That's just a rule of thumb, of course. Cool. So, part 1 subproblem is going to be the optimal evaluation parenthesization of ai to aj minus 1.

So that's part of the problem here. We want to do a0 to n minus 1. So in general, let's just take some substring in here and say, well what's the best way to multiply that, and that's the sorts of subproblems we're getting if we use this guess. And if you start with a substring and you do this, you will still remain within a substring, so actually I have to revise this slightly.

Now we're going from ai-- to solve this subproblem, which is what we need to do in the guessing step, we start from ai, we go to some guest place, ak minus 1, then from ak up to aj minus 1. This is the i colon j subproblem. So we guess some point in the middle, some choice for k. The number of choices for k is-- number of possible choices for this guess, so we have to try all of them, is like order j minus i plus 1. I put order in case I'm off by 1 or something. But in particular this is [INAUDIBLE]. And that's all we'll need. So that's the guess.

Now we go to step 3, which is the recurrence. And this-- we're going to do this over and over again. Hopefully by the end, it's really obvious how to do this recurrence. Let me just fix my notation, we're going to use dp, I believe. For whatever reason, in my notes I often write dp of ij. This is supposed to be the solution to the subproblem i colon j.

I want to write it recursively, in terms of smaller subproblems, and I want to minimize the cost, so I'm going to write a min overall. And for each choice of k, so there's going to be a for loop, I'm going to use Python notation here with iterators. So k is going to be in the range, I think range ij is correct. I'm going to double check there's no off by 1's here. Says i plus 1j. I think that's probably right.

Once I choose where k is, where I'm going to split my multiplication, I do the cost for i up to k, that's the left multiplication, plus the cost for k up to j, plus-- so those are the two recursive multiplications. So then I also have to do this outermost one. So how much does that cost? Well, it's something, so cost of the product ai colon k times the product ak colon j. So I'm assuming I can compute this cost, not even going to try to write down a general formula, you could do it, it's not hard, it's like xyz.

For a standard matrix multiplication algorithm. But whatever algorithm you're using, assuming you could figure out the dimensions of this matrix, it doesn't matter how it's computed, the dimensions will always be the same. You compute the dimensions of this matrix that will result from that product, it's always going to be the first dimension here, with the last dimension there. And it's constant time, you know that.

And then if you can figure out the cost of a multiplication in constant time, just knowing the dimensions of these matrices, then you could plug this in to this dynamic program, and you will get the optimal solution. This is magically considering all possible parenthesizations of these matrices, but magically it does it in polynomial time. Because the time for subproblem here--

We're spending constant time for each iteration of this for loop, because this is a constant time just computing the cost. These are free recursive calls, so it's dominated by the length of the for loop, which we already said was order n, so it's order n time for subproblem, ignoring recursions. And so when we put this together, the total time is going to be the number of some problems, which I did not write.

The number of problems in step 1 is n squared, that's what we said over here, for substrings. So running time is number of subproblems, which is n squared, times linear for each, and so it's order n cubed, it's actually theta n cubed. So polynomial time, much better than trying all possible parenthesizations, they're about 4 to the n parenthesizations, that's a lot. Topological order here is a little more interesting, if you think about that.

I can tell you, for suffixes, topological order is almost always right to left. And for prefixes, it's almost always left to right, for increasing i, decreasing i. For substrings, what do you think it is? Or for this situation in particular? In what order should I evaluate these subproblems?

AUDIENCE: [INAUDIBLE].

PROFESSOR: This is the running time to determine the best way to multiply-- that's right. So yeah, it's worth checking, because we also have to do the multiplication. But if you imagine this n, the number of matrices you're multiplying is probably much smaller than their sizes. In that situation, this will be tiny, whereas the time to actually do the multiplication, that's what's being computed by the DP, hopefully that's much larger, otherwise you're kind of wasting your time doing the DP.

But hey, at least you could tell somebody that you did it optimally. But it gets into a fun issue of cost of planning verses execution, but we're not really going to worry about that here. So, in what order should I evaluate this recurrence, in order to-- I want, when I'm evaluating DP of ij, I've already done DP of ik and DP of kj, and this is what you need for bottom up execution. Yeah.

AUDIENCE: Small to large.

PROFESSOR: Small to large, exactly. We want to do increasing substring size. That's actually what we're always doing for all of those subproblems over there. When I say all suffixes, you go right to left. Well, that's because the rightmost suffix is nothing, and then you build up a larger and larger strings, same thing here. Exercise, try to draw the DAG for this picture. It's a little harder, but if you-- I mean you could basically imagine-- I'll do it for you.

Here is, let's say-- well, at the top there's everything, the longest substring, that would be from zero to n, that's everything. Then you're going to have n different ways to have substrings of, or actually just two different ways, to have a slightly smaller substring. At the bottom you have a bunch of substrings, which are the length zero ones, and in between, like in the middle here, you're going to have a

much larger number.

And all these edges are pointed up, so you can compute all the length zero ones without any dependencies and then just increasing in length. It's a little hard to see, but in each case-- Yeah, ah, interesting. This is a little harder to formulate as a regular shortest paths problem, because if you look at one of these nodes, it depends on two different values, and you have to take the sum of both of them.

And then you also add the cost of that split. Cool. So this is the subproblem DAG, you could draw it, but this DP is not shortest paths in that DAG. So perhaps dynamic programming is not just shortest paths in a DAG, that's a new realization for me as of right now.

OK. Some other things I forgot to do-- I didn't specify the base case. The base case for that recurrence is when your string is of length 0 or even of length 1, because when it's length 1, there's only one matrix, there's no multiplication to do, and so the cost is zero. So you have something like dp of i, i plus 1 equals zero. That's the base case. And then step 5, step 5 is what's the overall problem I want to solve, and that's just dp from 0 to n, that's the whole string.

Any questions about that DP? I didn't write down, I didn't write down a memoized recursive algorithm, you all know how to do that. Just do this for loop and put this inside, that would be the bottom up one, or just write this with memoization, that would be the recursive algorithm. It's totally easy once you have this recurrence. All right, good.

How many people is this completely clear to? OK. How many people does it kind of make sense? And how many people it doesn't make sense at all? OK, good. Hopefully we're going to shift more towards the first category. It's a little magical, how this guessing works out, but I think the only way to really get it is to see more examples and write code to do it, that's-- the ladder is your problem set, examples is what we'll do here.

So next problem we're going to solve. Dynamic programming is one of these things

that's really easy once you get it, but it takes a little while to get there. So edit distance, we're going to make things a little harder. Now we're going to be given two strings instead of just one. And I want to know the cheapest way to convert x into y.

I'm going to define what transform means. We're going to allow character edits. We want to transform this string x into string y, so what character edits are we allowed? Very simple, we're allowed to insert a character anywhere in the strength, we're allowed to delete a character anywhere in the string, and we're allowed to replace a character anywhere in the string, replace c with c prime.

Now, you could do a replacement by deleting c and inserting c that's, one way to do it, but I'm going to imagine that in general someone tells me how much each of these operations costs, and that cost may depend on the character you're inserting. So deleting a character and then inserting a different character will cost one thing. It will cost the sum of those two cost values. Replacing a character with another character might be cheaper. It depends. Someone gives me a little table, saying for this character, for letter a, it costs this much to insert, for letter b it costs this much to insert, this much to delete, and there's a little matrix for, if I want to convert an a into a b it costs this much to replace.

Imagine, if you will, you're trying to do a spelling correction, someone's typing on a keyboard, and you have some model of, oh, well if I hit a, I might have meant to hit an s, because s is right next to an a, and that's an easy mistake to make if you're not touch typing, because it's on the same finger, or maybe you're shifted over by one. So you can come up with some cost models, someone could do a lot of work and research and whatnot and see what are typical typos, replacing one letter for another, and then associate some cost for each character, for each pair characters, what's the likelihood that that was the mistake?

I call that the cost, that's the unlikeliness. And then you want to minimize the sum of costs, and so you want to find what was the least set of errors that would end up with this word instead of this word. You do that on all words of your dictionary and then you'll find the one that was most likely what you meant to type. And insertions

and deletions are, I didn't hit the key hard enough, or I hit it twice, or accidentally hit a key because it was right next to another one, or whatever.

OK, so this is used for spelling correction. It's used for comparing DNA sequences, and DNA sequences, if you have one strand of DNA, there's a lot of mutation-- some mutations are more likely than others. For example, c to a g mutation is more common than c to an a mutation, and so you give this replacement a high cost, you give this one a low cost, to represent this is more likely than this.

And then at a distance will give your measure of how similar two DNA strings are evolutionarily. And you also get extra characters randomly inserted and deleted in mutation. So, it's a simplified model of what happens in mutation, but still it's used a lot. So all these are encompassed by edit distance.

Another problem encompassed by edit distance is the longest common subsequence problem. And I have a fun example, which I spent some hours, way back when, coming up with. I can't spell it, though. It's such a weird word. Hieroglyphology is an English word and Michelangelo is another English word, if you allow proper nouns, unlike Scrabble.

So, think of these as strings. This is x, this is y. What is the longest common subsequence? So not substring, I get to choose-- I can drop any set of letters from x, drop any set of letters from y, and I want them to, in the end, be equal. It's a puzzle for you. While you're thinking about it, you can model this as an edit distance problem, you just define the cost of an insert or a delete to be 1, and the cost of a replace to be 0. So this is a c to c prime replacement.

It's going to be 0 if c equals c prime, and I guess infinity otherwise. You just don't consider it in that situation. Can anyone find the longest common subsequence here? It's in English word, that's a hint. So if you do this you're, basically trying to minimize number of insertions and deletions. Insertions in x correspond to deletions in y, and deletions in x correspond to deletions in x. So this is the minimum number of deletions in both strings, so you end up with a common substring.

Because replacement says, I don't pay anything if the characters match exactly, otherwise I pay everything. I'd never want to do this, so if there's a mismatch I have to delete it. And so this model is the same thing as long as common subsequence. I want to solve this more general problem, it's actually easier to solve the more general problem, but in particular, you can use it to solve this tricky problem. Any answers? Yeah. Hello. Very good. Hello is the longest common subsequence. You can imagine how I found that. Searching for all English words that have "hello" as the subsequence. That can also be done in polynomial time.

So how are we going to do this? Well, I'd like to somehow use subproblems for strings, suffixes, prefixes, or substrings. But now I have two strings, that's kind of annoying. But don't worry, we can do sort of dynamic programming simultaneously over x and y. What we're going to do is look at suffixes of x and suffixes of y, and to make our subproblems we need to combine all of those subproblems by multiplication.

We need to think about both of them simultaneously. So subproblem is going to be solve edit distance, edit distance problem on two different strings, a suffix of x and a possibly different suffix of y. Because this is for all possible i and j choices. And so the number of subproblems is?

AUDIENCE: N squared.

PROFESSOR: N squared, yes. If x is of length n and y is of length n, there's n choices for this, n choices for that, and we have to do all of them as pairs, if there's n squared pairs. In general, if they have different lengths, it's going to be the length of x times length of y. It's quadratic. Good. So, next we need to guess something, step 2. This is maybe not so obvious, let's see. You have here's x, starting at position i. You have y starting at position j.

Somehow I need to convert x into y, I think it's probably better if I line these up, even though in some sense they're not lined up, that's OK. I want to convert x into y. What should I look at here? Well, I should look at the very first characters, because we're looking at suffixes. We want to cut off first characters somehow. How could it--

what are the possible ways to convert, or to deal with the first character of x? What are the possible things I could do? Given that, ultimately, I want the first character of x to become the first character of y.

AUDIENCE: Delete [INAUDIBLE].

PROFESSOR: You could delete this character and then insert this one, yes. Other things? There's a few possibilities. If you look at it right, there are three possibilities. And three possibilities are insert, delete, or replace. So let's figure out how that's the case. I could replace this character with that character, so that's one choice. That will make progress. Once I do that, I can cross off those first characters and deal with the rest of the substrings.

Let's think about insert and delete. If I wanted to insert, presumably, I need this character at some point. So in order to make this character, if it's not going to come from replacing this one, it's got to be from inserting that character right there. Once I do that, I can cross out that newly inserted character in this one, and then I have all of the string x from i onward still, but then I've removed one character from y, so that's progress.

The other possibility is deletion, so maybe I delete this character, and then maybe I insert it in the next step, but it could be this character matches that one, or maybe I have to delete several characters before I get to one that matches, something. But I don't know that, so that's hard to guess, because that would be more time to guess. But I could say, well, this character might get deleted. If it gets deleted, that's it, it gets deleted. And then somehow the rest of the x, from i plus 1 on, has to match with all of y, from j on.

But those are the three possibilities, and in some sense capture all possibilities. So it could be we replace xi with yj, and so that has some cost, which we're given. It could be that we insert yj at the beginning, or it could be that we delete xi.

You can see that's definitely spanning all the possible operations we can do, and if you think about it long enough, you will be convinced this really covers every

possible thing you can do. If you think about the optimal solution, it's got to do something to make this first character. Either it does it by replacement or it does it by an insertion. But if it inserts it later on, it's got to get this out of the way somehow, and that's the deletion case. If it inserts it at the beginning, that's the insertion case, if it just does a replacement, that's the replace case. Those are all possibilities for the optimal solution.

Then you can write a recurrence, which is just a max of those things, those three options. So I'm going to write, I guess, dp of ij, yes, of i,j, but now i,j is not a substring. It's a suffix of x and a suffix of y, so it corresponds to this subproblem. If I want to solve that subproblem, it's going to be the min of three options.

We've got the replace case, so it's going to be some cost of the replace, from xi to yj. So that's a quantity which we're given. Plus the cost of the rest. So after we do this replacement, we can cross off both those characters, and so we look at i plus 1 on for x, and j plus 1 onwards for y. So that's option 1. Then comma for the min.

Option 2 is we have the cost of insert yj. So that's also something we're given. Then we add on what we have to do afterwards, which is we've just gotten rid of yj, so x still has the entire string from i on, and y has a smaller string. Comma. Last option is basically the same, cost of the delete, deleting xi, and then we have to add on DP of i plus 1j.

Because here we did not advance y but we advanced x. It's crucial that we always advance at least one of the strings, because that means we're making progress, and indeed, if you want to jump to step 4, which is topological ordering-- sorry, I reused my symbols here, some different symbols. Head back to step 4 of DP, topological order.

Well, these are suffixes, and so I know with suffixes I like to go from the smaller suffixes, which is the end, to the beginning. And, indeed, because we're always increasing, we're always looking at later substrings, later suffixes, for one or the other. It's enough to just do-- come over here. To just do that for both of the strings, it doesn't really matter the order. So you can do for i equals x down to zero, for j

equals y down to zero, and that will work.

Now this is another dynamic programming you can think of as just shortest paths in the DAG. The DAG is most easily seen as a two-dimensional matrix, where the i index is between zero and length of x, and the j index is between zero and length of y, and each of the cells in this matrix is a node in the DAG. That's one of our subproblems, dp of ij. And it depends on these three adjacent cells.

The edges are like this. If you look at it, we have to check i plus 1, j plus 1, that's this guy. We have to check ij plus 1, that's this guy. We have to check i plus 1j, that's this guy. And so, as long as we compute the matrix this way, what I've done here is row by row, bottom up. You could do it anti-diagonals, you could do it column by column backwards, all of those will work because we're making progress towards the origin.

And so if you ever-- if you look up at a distance, most descriptions think about it in the matrix form, but I think it's easier to think of it in this recursive form, whatever your poison. But this is, again, shortest paths in a DAG. The original problem we care about is dp of zero zero, the upper left corner.

So to be clear in the DAG, what you write here is like the cost of, the weight of that edge is the cost of, I believe, a deletion. Deletion, oh sorry, it's an insertion. Inserting that character, this one's a cost of deletion, this is a cost to replace, so you just put those edge weights in, and then just do a shortest paths in the DAG, I think, from this corner to this corner. And that will give you this, or you could just do this for loop and do that in the for loop, same thing.

OK. What's the running time? Well, the number of subproblems here is x times y, the running time for subproblem is? I'm assuming that I know these costs in constant time, so what's the overall running time of that, evaluating that? Constant.

And so the overall running time is the number of subproblems times a constant equals x times y. This is the best known algorithm for edit distance, no one knows how to do any better. It's a big open problem whether you can. You can improve the space a little bit, because we really only need to store the last row or the last

column, depending on the order you're evaluating things. To even get down to linear space, as far as we know, we need quadratic time.

One more problem, are you ready? This one's going to blow your minds hopefully. Because we're going to diverge from strings and sequences, kind of. So far everything we've looked at involves one or two strings or sequences, except for [INAUDIBLE]. That involved a graph, that was a little more exciting. But we'd already seen that, so it wasn't that exciting.

OK, our last problem for today is knapsack. It's a practical problem. You're going camping. You're going backpacking, I should say, and you can only afford to take whatever you can fit on your back. You have some limit to capacity, let's say one giant backpack is all you can carry.

Let's imagine it's the size of the backpack that matters, not the weight, but you could reformulate this in terms of weight. And you've got a lot of stuff you want to bring. Ideally you bring everything you own, that would be kind of nice, convenient, but it'd be kind of heavy. So you're limited, you're not able to do that.

So you have a list of items and each of them has a size, si, and has a desire, a value to you, how much you care about it, how much you need it on this trip. OK, each item has two things, and the sizes are integers. This is going to be important. It won't work without that assumption. And we have a knapsack, backpack, whatever, I guess it's the British, but I don't know, I get confused. Growing up in Canada, I use both, so it's very confusing. Knapsack of total size, S.

And what you'd like to do is choose a subset of the items. If you're lucky, the sum of the si's fit within s, then you bring everything. But if you're not lucky, that's not possible, you want to choose a subset of the items whose total size is less than or equal to s, in order to maximize the sum of the values. So you want to maximize the sum of values for a subset of items, of total size less than or equal to S.

You can imagine size as weights instead of size, not a big deal, or you could have sizes and weights. All of these things generalize. But we're going to need that the

sizes/weights are integers. And so the items have to fit, because you can't cheat, you can't have more things than what fit, but then you want to maximize the value.

How do we do this with dynamic programming? With difficulty. I don't have a ton of time, so I think I'm going to tell you-- well, let's see. Start with guessing. This is the easy part to this problem. We should also be thinking about subproblems at the same time. Even though I said we're leaving sequences, in fact, we have a sequence here, we have a sequence of items.

We don't actually care about the order of the items, but hey, they're in an order. If they weren't, we could put them in an order, in an arbitrary order. We're going to use that order, and we're going to look at suffixes of items. i colon of items. That's helpful, because now it says, oh, well, we should be plucking off items from the beginning. Starting with the i-th item, what should I decide about the i-th item, relative to the optimal solution? What should I guess?

AUDIENCE: Is i included or not?

PROFESSOR: Is i included or not, exactly. Is item i in the subset or not. Two choices, easy. Of course, those are the choices. If I do that for everybody, then I know the entire subset. Somehow I need to be able to write and this is what's actually impossible if I choose this as my subproblem.

I want to write DP of i, somehow, in terms of, I guess, DP of i plus 1. And we'd like to do max, and either we don't put it in, in which case that's our value, or we put it in, in which case we get an additional v i in value. OK, but we consume in size, and there's no way to remember that we've consumed the size here.

We just called DP of i plus 1. In this case, it has everything, all this. In this case, we lose si of S, but we can't represent that here. That's bad, this would be an incorrect algorithm. I would always choose to put everything in, because it's not keeping track of the size bound. There's no capital S in this formula, that's wrong. So, to fix that, I'm going to write that again, but a subproblem is going to have more information, it's going to have an index i, and it's going to have remaining capacity.

I'm going to call it capital X, at some integer at most S. We're assuming that the sizes are all integers, so this is valid. The number of subproblems is equal to n, the number of items, did I say there are n items? Now there are n items, times capital S, really S plus 1, because I have to go down to zero. But n times S, different subproblems. Now for each of them I can write a recurrence, and that is DP of i comma s, is going to be the max of DP of i plus 1s.

This is the case where we don't include the items, so S stays the same. Actually I should write x here, because it's not actually our original value of s. x is the general situation. The other possibility is we include item i, and then we give DP of i plus 1. We still consume item i. We now have x minus si as our new capacity, what remains after we add in this item. And then we add on vi, because that's the value we gain from putting that item in. That's it, that's the DP, pretty simple.

Let me say a little bit about the running time of this thing. Again, you check there's a topological order and all that, it's in the notes. The total running time, we spend constant time to evaluate this formula, so it's super easy. The number of subproblems is the bottleneck. So it's n times s.

Is this polynomial time? You might guess from the outline of today that the answer is no. This is not polynomial time. What this polynomial time mean? It's polynomial and n, where n is the size of the input. What's the size of the input here? Well, we're given n items, each with a size, each with a value. If you think of the sizes and values as being single word items, then the size is n.

If you think of them as being ginormous values, at most, the size of this input is going to be something like n times log s, because if you write it out in binary you would need log s, bits to write down those numbers. But it is not n times s. This would be the binary encoding of the input, but the running time is this. Now s is exponential in log s, this is, at best, an exponential time algorithm. But it's really not that bad if s is small, and so we call it pseudopolynomial time.

What does pseudopolynomial mean? It just means that your polynomial in n, the input size, which might be this, and in the numbers that are in your input. Numbers

here means integers, basically, otherwise it's not really well defined. So in this case we have a bunch of integers, but in particular we have s. And so there's S and the si's. This is definitely polynomial in n and s. It is the product of n and S. So you think of this as pseudoquadratic time, I guess? Because it's quadratic, but one of the things is pseudo, meaning it is one of the numbers in the input.

So if the number is big in k bits, so I can write down a number that's of size 2 to the k. So it's kind of in between polynomial and exponential, you might say. Polynomial good, exponential bad, pseudopolynomial, it's all right. That's the lesson. And for knapsack, this is the best we can do, as we'll talk about later. Pseudopolynomial is really the best you could hope for. So, sometimes that's as good as you can do and dynamic programming lets you do it.