[SQUEAKING]

[RUSTLING]

[CLICKING]

ERIK DEMAINE: All right, welcome to the grand finale of dynamic programming in 6.006. 4 of 4. Today we are going to focus in on a particular type of problem that we saw at the very beginning with Fibonacci, which is when you have an integer input, and a natural thing to do with that integer input is look at smaller versions of that integer.

And this is going to lead us to a new notion called pseudopolynomial time. We've talked a lot in this class about polynomial time being a good running time, but pseudopolynomial is a pretty good running time, and we'll talk about that. And it relates to these integers. We'll only look at two new examples-- rod cutting and subset sum-- but then we're going to review all the examples we've seen from a kind of diagonal perspective.

So as usual, we're applying our SRTBOT framework, subproblems, relations, topological order, based cases, original problem, and time. Quick review-- so the hardest part is getting the right set of subproblems. And there's some natural choices. For sequences, we try prefixes, suffixes, substrings. For integers, like in Fibonacci, there's a given number, and we want to compute that at the n-th Fibonacci number.

What we ended up doing was solving Fibonacci numbers for all input numbers-- input integers between 0 and that number n-- or in this case, capital K. And that's a general technique, and we'll see two more examples of that today. Otherwise, we take products of these. And often, that's enough, but sometimes we need to add more subproblems in what we called subproblem expansion, often with extra constraints that let us remember some state about the past.

My canonical example of that is the piano fingering, where we had to remember what our fingering assignment was, in some sense, from the previous step in order to compute the transition cost. And this is a very powerful technique. You can also use it to play *Super Mario Brothers* optimally. If you have a constant size screen and all you need to remember is what's in the constant size screen, if everything outside that screen resets, you can just add that state as a parameter to your subproblem and you'll be able to solve *Super Mario Brothers*-- anyway very useful.

And that was the focus of last lecture. We won't talk about it much here today. And then we relate these subproblems recursively, and this is basically the test of whether your subproblem definition was correct is, can you write down a recurrence relation-- which is just a recursive algorithm? And there's a nice general procedure for how to come up with these relations, which is to just think up some question about the subproblem solution that, if you knew the answer, reduced to smaller subproblems.

And then you just locally brute force all the answers to that question, which I like to think of as guessing the answer correctly, and then just directly calling the recursive things. But then, at the end, you have to pay for that guess by looping over all possible guesses in order to guarantee that you actually find the right one.

So once you identify this question, it's very easy. DP is all about just brute force anything that you want. And usually that leads to pretty good running time, as long as the number of possible answers to that question is polynomial. Then we need to check this relation is acyclic, and then it's-- often reduces to finding a path-- like a shortest path or something-- in a DAG-- a subproblem DAG.

We need some base cases. We need to make sure we can solve the original problem using one or more subproblems, and then we analyze the running time as usually a number of subproblems times the amount of non-recursive work in the relation plus however much time it took us to solve the original problem.

So that was our framework. We've seen it four times now, slightly refined each time. We've mostly added some general techniques for subproblem definition and how to write relations. So let's do a new example, which is rod cutting.

This will be pretty straightforward, but it will serve as a contrast to the next example we talk about, subset sum. So what is the problem? The name rod cutting comes from the book *CLRS,* but it's actually a pretty practical problem-- maybe not for cutting rods. But you could imagine you have some resource of a given length.

Because I have been wood-- hardwood shelf shopping recently, I like to think, if you have a big plank of hardwood and you get some price for selling that length, but you could also cut that plank into multiple pieces of various lengths that sum to L. And you could sell them individually, and maybe you make more money that way. And that's what this problem is all about.

So you're given the value of every possible length you could cut. We're going to assume all lengths have to be integers, scaled so that that's true. So capital L here is the original length, little L is a candidate length of a piece you might cut, and v of l is the value for cutting off a length L rod-- sub-rod. And we're going to assume, when we cut, we don't lose any material.

You could probably adjust for that, but it's not terribly interesting. And we want to know, what is the best way to split up our big rod of length capital L into various rods of small L length-- potentially different lengths? So I'll call this the maximum value partition. In mathematics, this is called partition, a bunch of numbers that sum to a given number.

And we want to maximize the total value, naturally. So here's an example. Let's say our original length is 7 and we have this table for lengths 1, 2, 3, 4, 5, 6, 7-- all the different lengths. I'm going to write down a value that's an integer for cutting off a rod of that length and selling it. It's like the sell price.

It's presumably monotonic, but doesn't have to be. Maybe people really like buying powers of 2 length or something, and so those sell higher. So this doesn't have to be monotonic, but in this example, it is. And so I have this rod of length 7, and I could sell it directly for $32, let's say, but I could also split it into, say, a length 1 rod and a length 6, or a length 1 rod and two length 3's, or length 3 and a 4-- anything that sums to 7.

And probably the most natural thing to do for this problem is a heuristic. This would be a greedy-- bang for a buck heuristic is what is usually called-- it's to take the ratio-- how much money do I get for a given length? Divide v of L by L, and try to maximize that. If I had to pick a single item and sell many of that type that, would be the optimal thing to do.

So this has a ratio of 1. That's bad. This has a ratio of 5. That's better. And you stare at it long enough-- I believe 6 is the best-- has the highest ratio, slightly more than-- I can't divide-- slightly better than-- let's see.

AUDIENCE: Slightly worse than 4--

ERIK DEMAINE: Slightly worse than 4-- was 4 the best? Slightly worse than 4? What do you mean?

AUDIENCE: 31 over 6 is slightly less than 4?

ERIK DEMAINE: Oh, slightly-- I see. The ratio is slightly less than 4. Thank you. Yeah-- which, all of these others are somewhat worse. If I double the 3 value, I get 26, which is quite a bit less than 31. The closest competitor, I think, is 2, because if you multiply this by 3, you get 30. So if I sold three 2's, I get $30, but if I sell one 6 which-- is the same quantity-- I get $31, a slight improvement.

And so this item maximizes bang for buck, for that ratio. And so one natural partition is 6 plus 1. I sell one out of length 6, and that leaves a lot of length 1. And this will give me $31 for the 6, and $1, which gives me $32. But this turns out to not be the best, which is actually the same as if you just sold it outright.

But in fact, you can do better by selling-- this is not obvious-- stare at it for a while-- a 3 and a 4-- also sums to 7. Then we get 13 plus 18, which is hopefully bigger-- 33. Nope, I did not get it right. That's too small.

We're going to take these two and sell 3 plus 2 plus 2, and I get 13 plus 10 plus 10. Remember, 2 was a close competitor for the ratio for 6, so it's a little better to sell 2's-- two 2's and then a 3, because then we get $33. And that turns out to be the best for this problem. And it seems really tricky to figure this out. In general, there are exponentially many different partitions-- can't afford to try them all. Question?

AUDIENCE: Can I have negative values?

ERIK DEMAINE: Can I have negative values in here? I think that will work fine. I'm not allowed to have negative lengths or 0 lengths. I don't want 0 lengths to actually give you something, because then I'd just cut infinitely many 0's. But the v of L, I think, could be negative. Yeah?

AUDIENCE: Do I have to use a whole bar?

ERIK DEMAINE: Do I have to use a whole bar? In this problem, yes. I think it wouldn't change much, if you didn't have to use the whole bar. We can think about those after we write the DP. So let's solve this with SRTBOT.

So what's the input to this problem? I didn't mention this is an integer length-- positive integer length. So we have one input, which is an integer L, and we have another input, which is-- I guess it's an array of integers. So this is a sequence and this is an integer.

So if we look at our list of nice subproblems, we could try prefixes, or suffixes, or substrings of the value structure, or we could try, for that integer, smaller integers. That's actually what I prefer. I think the way to think about this is to jump ahead to, what do we want to-- what feature of the solution do we want to guess?

And presumably, we should think about, what is some length of rod that we will cut and sell? So I have this big rod. Maybe I sell the whole thing. Maybe I cut off a thing of size 1 and sell it. Maybe I cut off a thing a size 2 and sell it. But I have to sell something, unless I'm not selling anything.

There's only L different choices for what rod lengths to cut off first, and so we can just brute force that in order L time. So what problem do we get if we cut off an integer of some small L length? Well, we just get the same problem with a rod of length capital L minus small L. The values don't change. It happens that I won't use the big values, if I cut off some amount of the problem, but I like to think of this as we're just-- all we're doing is decreasing big L.

And so my subproblems are going to be, for each small L, less than or equal to big L, solve that problem. So x of L is max value partition of length L-- little L-- for little L equals 0, 1, up to big L-- so just the same problem, but with different choices for big L. So this is using this integer subproblem.

Now, in this example that happens to correspond to prefixes of the v array-- because, if I only have length up to little L, then I really only need the prefix of the value array up to little L. So you could think of it that way. That's also fine. But I think this way's a little more generalizable.

OK. So I claim this is a good set of subproblems, because I can write a recurrence relation, which is actually pretty simple. Like I said, we want to choose some piece-- so we're given a lot of length little L. I want to choose how much of that rod to sell in the next piece. So I could cut off something off length 1, or I could sell the whole thing, or cut off any piece size in between.

And the money I will get for that is whatever the value of that piece is plus, recursively, the maximum I can get from all the remaining pieces-- sorry-- from the remaining length, which is little L minus p-- so very simple inside the formula-- just the value for the first piece. We're guessing, what is the first piece we'll cut off and sell?

We get the value for that piece, and then recursively, we see what's the best we can do with the remainder.

Now, we don't know what size the first piece should be, so we just brute force it. We try all possible choices for p and take the max that we get out of this formula over that choice. And that's guaranteed to find the best overall, because we must cut off some piece. Now, if you wanted to allow not selling anything, in the case of negative numbers, you could just add a 0 to this max, and then you might stop early if there's nothing left that's worth selling.

OK, topological order, it is very simple. We just have these capital L different problems, and if you look at-- oh. Yeah. So we're looking at L minus p. p is always at least 1, so we're always strictly decreasing L in this recursive call. And so as long as I solve the problems in order of increasing little L, I'll guarantee that, whenever I'm solving x of little L, I'll have already solved all the things I need to call.

So if you're writing a bottom-up DP, this would just be for loop L equals 0 up to big L, in that order. This is equivalent to the statement. But the key is to check that this is acyclic, because we're always referring to smaller L, and so increasing L is a valid topological order in this subproblem DAG defined here, where there's an edge from an earlier thing to evaluate to a later thing to evaluate.

OK, base case would be x of 0. That's the first thing we want to compute here. And indeed, this formula doesn't make much sense if I have x of 0, because I can't choose a number p between 1 and 0, even non-strictly, and so what does it mean? Well, if I have a rod of length 0, I can't get any money out of it, so that's it-- just 0. That's an assumption, but a reasonable assumption. You don't get something for nothing.

OK, then we have the original problem, which is just the length L rod. And then the time to compute this thing-- how many subproblems are there? Capital L plus 1-- so we'll say theta L subproblems. And we'll multiply by the amount of time to evaluate this max-- just a constant number of things in here, not counting the recursive call.

And so it was spent-- takes little L time. Little L is certainly, at most, big L, and so we'll say big O of big L time. It's actually a triangular number, but it will only affect things by a constant factor. So we get L squared time, and we're done-- so very simple, straightforward DP. At this point, we've seen much more complicated examples than this.

But it highlights a question, which is, is theta L squared polynomial time? So is this a reasonable running time? And I claim the answer is yes, this is a reasonable polynomial running time. You might say, well, of course, it's a polynomial. Look, this is a polynomial-- L squared. That's a quadratic polynomial.

But it's a quadratic polynomial in L. And we haven't really thought about this too hard, but what does it mean to be polynomial time? And this is a notion that's properly called strongly polynomial time. We won't worry about that strongly too much in this class, but if you look this up online, you'll see the difference.

Polynomial time means that the running time is polynomial in the size of the input. And the size of the input is, for our model, measured in words-- machine words. Remember, our good old word RAM, W bit words. It's been very useful, because it lets us assume things like adding two numbers is constant time, as long as these numbers fit in a word, as long as they're at most W bits.

And generally, we assume all the things we're manipulating fit and a machine word, because that's what-- the case where we normally work. And so the natural way to measure the size of an input-- so in this example, in this problem, rod cutting, the input is a single number L, and this value array, v of L, which is capital L numbers-- integers.

Doesn't mention integer value here. Throughout this class, we assume that all the integers, unless otherwise specified, fit in a word. So we've got one word here and L words here, so the total size of the input to this problem is L plus 1 integers. So in this problem, input size is L plus 1, which we can think of as just L. It's theta L.

So I explicitly didn't want to use n in this problem, because usually we use the letter n for the problem-- not quite always. But input size always means input size, and so here, we can compute it in terms of the specification-- involves L plus 1 word inputs. And so polynomial time should be polynomial in that input size-- in L plus 1, in our example.

So of course, L squared is polynomial and L plus 1-- so good. Yes. OK, the next example I'm going to show is going to be very similar, but the answer will be no-- make it more interesting. But it'll still seem pretty reasonable. So we'll come back to that issue in a moment. Let me first show you what the subproblem DP looks like for this problem-- again, took me a while to draw, so please admire.

So this is the same example of these values-- 1, 10, 13, 18, 20, 31, 32-- drawn here on a graph. It's the complete graph oriented in this increasing way. I think this is called a tournament in graph theory. So we have the base case over here. This corresponds to a length 0 rod, where we get no money. And over here we have our full right of length 7, and claim the best we can do is 33.

And what's happening here is, for every value like 3-- I could sell a rod of length 3 for $13-- there's an edge that goes from each vertex to one 3 higher, and those all have a weight of 13 on them. And then what we're essentially trying to do is find a longest path from here to here-- longest because we want to maximize the sum of values that we get.

And we know, if we negate all the weights, that's the shortest path problem, so we could solve that with shortest paths in a DAG. But I've drawn here the shortest path tree from the base case. So it actually tells us that, if we had a rod of length 7, the best thing to do is to sell it directly for 31. The bold lines here are the shortest path tree-- or the longest path tree, I guess.

And if we had something like 10, we should sell it directly. If we have something of length 20, we should sell one thing of length 2 and another thing of length-- sorry-- one thing of length 4. Then we should sell one thing of length of 2, and one of length 2, and we get 2 times 10 points. And for the 33 case, we sell one thing of length 2, one thing of length 2, and then one thing of length 3 is optimal. So you can read lots of information from this. And if you write down the DP code, this is effectively what it's computing, from left to right.

OK, let's move on to a second problem today, which is subset sum. So here we are given the same multiset. Multiset means that I can repeat numbers. So this is just a sequence of numbers. But I'd like to use set notation, because I want to use subsets, because it's subset sum.

So this is n integers. And we're also given a target sum, and we want to know, does any subset sum to the target sum? This is actually a similar problem to rod cutting, because rod cutting-- we also had to split up our number L into different values that seemed to capital L. So capital T here is playing the role of capital L.

But before we were allowed to cut into any lengths, and so it was easy. In particular, L sums to L. Here, presumably, T is not in this multiset, and we need to find a combination of numbers here that add up exactly to T. Sometimes that's possible. Sometimes it's not. We're only allowed to use each number once, or as many times as it appears in the subset.

OK, so here's an example. Say A equals 2, 5, 7, 8, 9. And two examples are T equals 21 and T equals 25 for that same set. So can I get 21 out of these-- this involves arithmetic. That's hard. Let's see. If I add 7 and 8, I get 15-- not quite what I want. I'm going to cheat, look at my answer.

Yeah-- close. I see-- 5, 7, 9. So this is a yes answer to the question-- does there exist any subset-- because 5, 7, and 9 sum to exactly 21. And T equals 25-- I don't know a good way to prove to you that there's no way to write 25 with these numbers, other than I wrote a program to try to all subsets and-- or you could write a program that runs the DP that we're about to give.

And it will output no. There's no succinct way, as far as we know, to prove to someone that the answer's no for a given target sum. There is a nice, succinct way to prove the is yes. I just give you a subset-- we'll talk more about that next lecture. But these are some examples of the question you might ask, and the answer that we're looking for.

This is what we call a decision problem. In its original form, we're just interested in a yes or no answer. It's a single bit. Of course, in the yes case, we might actually want to find the set, and we can do that as usual with parent pointers, just like in the bold lines over here. We'll get to that in a moment.

Most of the problems we've been seeing with dynamic programming are optimization problems, and we're trying to minimize or maximize something, and so we always put a max on the outside in the relation. Here we're going to have to do something that involves boolean values-- yes or no, true or false.

OK, so let's solve it. This is actually also going to be pretty straightforward, in that we can just use our standard sets of subproblems. So just like the previous problem, we have, on the one hand, a sequence of integers, and on the other hand, we're given a single integer T.

And what turns out to be right is to use prefixes or suffixes on this sequence and integers less than or equal to T. Let's think about why. So that was SRTBOT for rod cutting. This is SRTBOT for subset sum.

Again, I'll look for-- look ahead to, what feature of the solution should I guess? Well, I have these n numbers. Each of them could be in my subset or not. So I have this binary choice. For each AI is it in S or not? That's a lot of questions to answer. I can't answer them answer them all at once, but I could just start with the first one and say, well, is a0 in S? Yes or no?

There's two answers-- locally brute force. If I do that, what happens to my problem? What new subproblems do I run into? What do I want to recurse on? Well, I've eliminated a0. That will leave a suffix of the AIs. So suffixes on capital A seem like a good idea.

And what about my target sum? Well, it depends. If I put a0 in my set S, then the target sum for the remainder is T minus a0. So T went down, and so I need in my subproblems to represent smaller target sums also. So this is how you figure out what subproblems you should use.

You could just try prefixes on this, suffixes on this, substrings on this. And yes or no-- do I include smaller versions of T here? But you can also just think about-- trying to write a recurrence relation first, see what things you were naturally recursing on, and then formulate the subproblems that way. So that's what I like to do.

So I'm going to have a subproblem for each suffix. So that's x of i. And for each target sum-- and I use these capital letters for the actual target sum so that I can use lowercase letters for smaller versions of them. So this is going to be, does any subset-- remember, don't-- first, most important thing is to define what your subproblems are.

Don't just say, it's the same problem, but where I replace blah with blah. It can be very ambiguous. Does any subset S of the suffix A from i onwards sum to little t? And we're going to have this subproblem. The other important thing is to say how many subproblems there are and what your indices can vary over.

So i is going to be between 0 and n, and t is going to be between 0 and big T. OK. So remember, subproblems is n plus 1 times 2 plus 1, or theta n times T. Cool. Now I claim I can write a relation, like I said, by-- so we have this suffix from A-- from I onwards in A, and so I'm just going to-- because I'm looking at suffixes, I want to keep with suffixes.

So I should try to guess what happens to A of i, because then what will remain is A of i plus 1 onwards. And A of i can either be in my subset S or not, so there's two choices. So x of IT is going to be-- involve two things. So there's an operator here, and then I have a set of two items, which is-- I could choose to not put A of i in my subset.

In that case, I've eliminated A of i, and what remains is A of i plus 1 onwards. And I didn't change my target sum. I haven't put anything in S, so I still want to achieve the same sum. So this is the case where AI is not an S.

And then the other case is I put A of i in S. In that case, again, I've eliminated A of i, and so what remains to figure out is what happens to A of i plus 1 onwards. But now my target sum is different, because I put a number in my set. And so among these items, they should sum up to not little t anymore, but now little t minus what I just added-- which is AI.

So then, if, in this subproblem, I get something that sums to t minus AI, and then I add AI to that set, I will get something that sums to exactly T. And so that's a valid solution to this problem. And because we have brute forced all possibilities-- there were only two-- if we combine these in the suitable way, then we will have considered all options.

Now, what do we want here? Normally I'd right max or min, but this is a decision problem. The output is just yes or no. So this is a yes or no answer. This is a yes or no answer. This is a yes or no answer. And so what I want is or. In Python, this would be called any-- just, are any of these things true?

Because if there's any way to construct a set that sums to T, then the answer to this is yes-- cool-- so very simple, actually. This is one of the simplest reoccurrences. But we're solving what I think is a really impressive problem. We're asking, is there any subset? There are exactly 2 to the n subsets of A here. And we're, in some sense, considering all 2 to the n of them, but because we split it up into n local choices that are binary, and do them only one at a time-- this is the local brute force versus global versus.

Global brute force would be trial subsets, sum them, see which ones add up. But we're, in some sense, collapsing a lot of these choices and reusing subproblems. That's the whole point of dynamic programming. For the rest of the sequence, from i plus 1 onwards, I don't really care exactly which subset you choose. I only care what it sums to.

And that collapses a lot of different options into the same thing. I really just want to know, is there any way to do it with exactly this sum? If I said, give me all of the different subsets that sum to T that would be exponential time, but because I'm just asking a yes or no question, this choice only takes constant time, and we get to some them, instead of product them-- because we're using memoization. That is the beauty of dynamic programming and the-- this time analysis rule that we only have to sum over subproblems because we only compute each subproblem at once.

Without memoization, this would take exponential time, just like Fibonacci. With memoization, magic. I just think it's beautiful. So even though it's one of our simpler DPs, I think it's an impressive one. OK.

Topological order-- well, let's look at these function calls. So here we have to be a little bit careful. When we call x recursively, we always increment i. But sometimes we don't decrement t. Sometimes t doesn't go down. So if I wrote decreasing t here, that would be bad, because sometimes I call with the same value of t, and I don't want to get into-- I want to ensure that this has already been computed when I try to compute this.

So i is the right thing, and we should say decreasing i. It doesn't actually matter how we order with respect to t just any order that is decreasing-- i will be good, because these function calls always increase i. OK, we need a base case.

Let's see. Given this aspect, I think the natural thing is to have a base case when my suffix is empty, which is when i equals n. So this is x of n, t-- for any little t, because we don't have a lot of control how t is changing. But this is easy. So this is saying, if I give numbers, what sums can you represent? The only sum I can represent is 0.

So if t equals 0, then the answer is yes. And otherwise, the answer is no. So that's my base case, and that's enough. And we needed this base case, because if we wrote x of n, t, this would try to call x of n plus 1, which doesn't make sense. But x of n colon is a natural suffix, which-- we only allowed i to go up to n, and that's the empty suffix.

So this is enough. We need the original problem, which is the entire string from 0 onwards, and capital T for little t. That's our target sum. And then the running time-- as I said, there are n times t subproblems-- theta. And the amount of work we spend for each one that isn't recursion is constant. We just do a couple subtractions, additions, do an or and recursive calls-- so constant time. So n times t is the running time of this algorithm.

Let me show you an example as a subproblem DAG. These are hard to draw, but they're easy to read. I think they're helpful to show what's really going on in this DP. Remember, every node here corresponds to a possible subproblem we can have. So we have the choices for little t on the top, choices for little i on the left.

So the original problem we care about is i equals 0, T equals 6. This is the same example that I showed you before, where we have 2-- or no, it's a different example. Sorry. My new set a is 3, 4, 3, 1-- four numbers here. My target value is 6. This is definitely possible. I can add 3 and 3.

This shows that a doesn't have to be sorted. We're not assuming anything about the order of a, and we're allowed duplicate values. And we see indeed, there's a y here for yes, it is possible. And how did I compute that? Well, I just drew a bunch of arrows. So there's vertical arrows here, always going from each problem to the next one above it, because we have-- this dependency xi of t calls x of i plus 1, t.

The calls are going in this direction, so the dependency order is you have to compute things lower down before you compute things up here. And indeed, down here is the base case, where we right yes for the first problem and no for all the others, because we don't have any numbers. We can't represent anything above 0.

And then we have these blue lines. I just drew them a different color so they stand out-- hopefully. And they correspond to the numbers here. So our first choice is, do we include a0 which is 3? So that's only possible if we're trying to represent a number that's greater than or equal to 3-- which reminds me, I forgot to write down-- in this DP, I need to say this option is only an option if AI is less than or equal to T. Just move my comments here. Those are the same.

So this notation means I put this item in this set that we take an order of only if this condition holds. Otherwise, I omit it, because it's not a choice. Why is that important? Because I don't want to call x on a negative value of t. We only are allowed to call x here, when t is between 0 and capital T. So that's subtlety, but important for a correct DP.

That's why there's no edge from here, for example, that goes through 3 to the left, because there's no vertex there. There's no subproblem. So only for little t, from 3 onwards, we have this edge that goes 3 back, and that's just the same pattern over and over. Then our next number is 4, and so we have these edges that are-- go 4 to the right.

Then our next number's 3, so we can have edges that go 3 to the right. And then our next number is 1, so we have these nice diagonal edges that go 1 to the right. And then what's happening in here at each stage-- let's take an interesting one-- maybe this vertex-- is we look at each of the incoming neighbors and we take the or.

So the incoming neighbor here has a no. Incoming neighbor here has a yes. And so we write a yes here, which means that, given just these numbers, 3 and 1, we can represent the number 3-- namely, by taking this edge of 3-- sorry-- of length 3 and then just going straight down to a base case. That's yes. That's representing 0.

So in this case, we-- this example-- I didn't draw the parent pointers here, but they're in the notes-- this yes is possible-- not from going this way, but from going this way. So we take the number 3. Then we go down, which means we skip the number 4. And then we go left, which means we take the number 3. And then we go which means we skip the number 1. So we can represent 6 as 3 plus 3-- cool.

So subset sum-- not only can we solve the decision problem, but by following parent pointers in the yes instances, we can actually find a valid subset. Question--

AUDIENCE: You added that condition to the relation?

ERIK DEMAINE: Yeah.

AUDIENCE: Is it possible to deal with that by spending your number of base cases?

ERIK DEMAINE: Good question-- it's a generally useful technique to add if conditions to the cases to only when they apply. You can write a lot of DPs that way, and that's why I wanted to stress it. You could, instead of adding this condition, allow the case that little t is negative. But you have to think about, how negative would it be?

You might say, well, maybe t between minus big T and plus big T is enough, but I don't think so. It should be we're at some value t and we subtract some AI. We don't know how the AIs compare it to big T. Probably they're less than or equal to big T, because they're not useful if they're bigger than big T.

So you could first prune the AIs to guarantee all the AIs less than big T. Then minus big T to big T would work. Otherwise, it's minus the maximum AI up to big T would be enough, I think. Yeah?

AUDIENCE: Does this restrict to only positive integers in the input?

ERIK DEMAINE: I am implicitly assuming here that all the AIs are positive. I think you can solve it with negative numbers, but it's not--

AUDIENCE: Maybe do it in recitation?

ERIK DEMAINE: Maybe we'll do it in recitation. It's not a trivial change to this DP. I've definitely thought about it before. Yeah, so I should have said positive integers here-- cool. All right, so that's subset sum. But we come back to this question of, is this a good algorithm? Is it polynomial time?

So we have this running time, n times t. So a new question is, is n times big T polynomial? And the answer is no. Why? Because for this problem, what is the input size? How many words of input do I have? Well, I have these n integers, and then I also have the target sum. So similar to up above, our input sizes n plus 1.

But now the labels really matter. Before it was L plus 1, and a running time was a function of L. Now our input size is just n plus 1, but our running time is a function of both N and T. Not polynomial in input size n plus 1. You cannot write that n times capital T is less than or equal to n plus 1 to the 27th power, because you just don't know how N and T relate.

Maybe T is, at most, N Then we're happy. But maybe it's not. Maybe T is 2 to the N. Why could it be 2 to the N? Because what do we know about capital T? We assume implicitly throughout the course the T fits in a word. That means it's a W bit number. So that means it's, at most, 2 to the w.

And you might think, well, we know about a relation between w and n. So we know T is, at most, 2 to the W, if it's W bits. And we know that our assumption is always that w is at least log n. That's the word RAM trans dichotomous assumption. But notice, we don't know anything about an upper bound on w. In order to upper bound T in terms of n, I'd need to say that w is at most log n. Then this would be at most n, and we'd be done. But that's not very interesting.

And generally, we allow w to be arbitrarily large with respect to log n. It just has to be at least log n just to be able to index stuff. But w could be really big-- much bigger than log n. For example, w equals n is not a crazy idea. If I have a machine that can represent n numbers, why not represent n numbers, each of which is n bits long?

Maybe it takes a little more time to compute on those-- might want to scale your running times. But it's quite reasonable to think about n bit numbers. And then this is like n times 2 to the n, so this would actually be exponential time. If w is 2 to the n, n times t is exponential in the problem size, which is n plus 1. And that's just an example. w could be bigger.

So this is not what we call a polynomial algorithm, or strongly polynomial algorithm but it's still pretty good, right? As long as we know that capital T is not ginormous, then this is a great algorithm. And we capture that notion with a concept called pseudopolynomial.

It's not the best term, but it's the established term. It's like polynomial, but not quite. And the definition of this term-- so we have definition of strong polynomial time. Now, pseudopolynomial time-- I guess I'll write time here, though you can measure other things with pseudopolynomial-- is that we're polynomial in the input size, just like before, and the input numbers-- input integers.

OK, so in this problem, the input integers are capital T and a0, a1, up to n minus 1. So what we want now is a polynomial, or what some people call a multinomial, where our variables are all of these integers, and we want to be polynomial in them. So we're allowed to take some constant power of capital T and some constant number of the AIs. We can't just take the product of all of them. That would be big.

So it's a-- I guess I should say constant degree polynomial. And indeed, this a-- you might call it a quadratic polynomial in the input size and the-- and one of the numbers. So this running time is pseudopoly, but not poly.

And so while normally, we think of polynomial time as good, exponential time is bad, pseudopolynomial is what we normally call pretty good, to be informal. Yeah. So in particular, pseudopolynomial implies polynomial in the special case if the input integers are all, at most, polynomial in the input size.

This should sound familiar. This is a constraint we've seen before. This is the condition when radix sort runs in linear time. Radix sort runs in linear time exactly when all the input integers are polynomial bounded in n, which is the size of the array, which is the input size. So same condition is appearing again, so this is sort of a fundamental setting to think about.

And let's see. So in particular, other structures we've seen, like counting sort and direct access arrays, are also pseudopolynomial. Any others?

AUDIENCE: Fibonacci--

ERIK DEMAINE: Fibonacci--

AUDIENCE: Radix?

ERIK DEMAINE: Sorry?

AUDIENCE: Radix--

ERIK DEMAINE: Radix sort-- yes-- technically, also radix sort-- are all-- they're not strongly polynomial, but they are pseudopolynomial. We thought about this is a dependence on u, which we got rid of using hashing. But if you just use the order u running time for, say, build, that-- U is bound on the input integers, and that's only good when this is polynomial.

In general, it's pseudopolynomial. Same with counting sort-- we had an order u. Now, we improved this in radix sort to this running time that was n times log base n of u-- plus n, if you want to be careful. So-- or put a ceiling. Now, this is a running time that's a little bit better. And this is usually called weakly polynomial-- don't want to spend much time on this.

But weakly polynomial is just like pseudopolynomial, but instead of being polynomial in the input size and the input integers, you're polynomial in the log of the integers. So this is better. And it's almost as good as polynomial-- as strongly polynomial. We won't really use this notion. The only place it appears in this class is in radix sort, but future classes-- you might care about-- so the nesting is the best thing you can be is strongly polynomial.

We just call this polynomial in this class. Then we have weakly polynomial, which is almost as good, but you have this logarithmic dependence on the numbers. And then the next level is pseudopolynomial. This is not as good here. This really only works well if the numbers are small. Logarithmic dependence is pretty good, because even if they're exponential, this is polynomial.

Sounds funny, but log of an exponential is polynomial-- all right, enough about pseudopoly. The last thing I want to talk about is reflecting on all of the dynamic programs we've seen so far, and characterizing them according to the big techniques that we used in SRTBOT. The first big technique was, how do we define our subproblems?

Did we take prefixes, and suffixes, or substrings of a sequence? Did we have multiple sequences and have to take products of those spaces? Did we have integers and have to take smaller versions of those integers? Sometimes that leads to pseudopolynomial algorithms, sometimes not.

And sometimes we also had subproblems that were defined in terms of vertices. This just happened in shortest path problems, because that's the only setting where we saw DP over graphs. So let me characterize all the DPs we've seen in lecture-- not the recitation ones for now-- in red.

So for example, with the bowling problem, with bowling pins, we only had to deal with prefixes or suffixes, because-- we could just think about what happened for the first couple of pins, for example. Also, for LCS, we had to take two different sequences, but then we just guessed what happened at the-- with the first two items of each sequence. And so that only left us with suffixes.

With longest increasing subsequence, again, we just guessed whether the first-- or maybe we assumed that the first item was in the longest increasing subsequence, and then we tried to guess what the next item was. But that, again, eliminated everything in between, so we were just left with the suffix.

This leads me to another characterization of the dynamic programming techniques, which is for-- in addition to these basic subproblems, we often added constraints and expansion. And LIS an example of what we call non-expansive constraint, where we just added a constraint to the problem, which was I want this first item to be in my longest increasing subsequence.

But that didn't actually change the number of subproblems, so it didn't expand the number of subproblems. This is, I think, the only example we saw with this feature. Most of the other times, when we added a constraint, we also increased the number of subproblems, which we'll get to.

OK, also, in a certain sense, there's multiple ways to think about this. One Floyd Warshall is a problem-- or we define subproblems based on prefixes of the vertices. We had vertices 1 through n, and we said, is there a shortest path using vertices just 1 to i? So that's a prefix of the vertex set in a particular order.

So you can think of Floyd Warshall as being-- as involving a prefix. You can also think of it as you're given an integer i, and you're only allowed to use vertices less than or equal to i, and so it's also kind of an integer subproblem. I will leave it up there.

Also, the two examples we saw today-- rod cutting kind of-- you could think of it as either a prefix on the values, or I would prefer to think of it down here, where we had an integer and we were considering smaller integers also. But subset sum definitely had a suffix, in addition to having an integer subproblem.

So rod cutting you can put down here, because we looked at smaller integers-- or up here. But subset sum really is up here and down here, because we both looked at suffixes and smaller values of T. OK. Fibonacci also fits down here. Fibonacci was another case where we had a number n, and we looked at integer smaller than n.

Good-- I think I've done these. Now, what problems involve substrings? We saw two of them. One was the alternate in coin game, because we were taking from left or right, and so we had to look at substrings in between. And the other is parenthesization, where we had to get something in the middle, and that left the prefix before it and the suffix after. Both of those are typical ways where you get substring problems.

OK, so pseudopoly-- both of these are pseudopoly, and those are the ones that we've seen that are dynamic program pseudopoly. And then, with vertices, it's all the shortest path problems, where we also had a parameter, which is the vertex. These are natural, because the goal of single shortest paths is distance to each vertex, and so naturally, we had a subproblem for each vertex-- for DAG shortest paths, for Bellman-Ford and for Floyd Warshall.

OK, back to subproblem expansion-- we saw a couple of examples-- alternating coin game and parenthesization-- sorry-- not parenthesization-- yeah, parenthesization-- sorry-- arithmetic parenthesization. So here we considered two different versions of each subproblem-- one where I go first and one where you go first. And that was really handy, though not necessary.

For parenthesization, it turned out we needed both min and max in order to solve max. So we really only cared about max, so we doubled the number of subproblems to make it solvable. For piano and guitar fingering, we increase the number of subproblems by a constant, or f-- or some f to the f, or some-- for 5 fingers, this is a reasonable constant-- for some amount of state that we wanted to keep track of of the past.

And one example where we had linear expansion sort of is Bellman-Ford. So here we were expanding by how many edges are in the shortest path. So we really only cared about finding shortest paths. We said, oh, what about at most i edges? So you can think of that as adding a constraint. And then there's n different variations of these subproblems, which leads to expansion.

You can also think of it is just adding a single constraint, which is I care about the number of edges. And that input is an integer, and then we're looking at the natural sub problem for integers, which is we care about up to the length n minus 1. And now let's consider all lengths smaller than n minus 1.

And finally, the other main feature we had is, in the recurrence relation and all these shortest path DAGs, how many incoming edges did we have? How many different branches did we have to consider, and then combine in some way? So we saw a lot of examples with constant branching-- Fibonacci, where we just-- it's the obvious two-way branching; bowling, where we had a couple of choices at the beginning; LCS, longest common subsequence, where we had a couple of choices what to do at the beginning; alternating coin game-- similarly, do we take from the left or the right-- so just two choices.

Floyd Warshall, there's two choices. Do we use that vertex or not? Subset sum-- do we include this item in the subset or not? So that was all constant branching. In a lot of the graph problems, we got order degree branching, which leads to an order e term in the final running time-- namely, DAG shortest paths and Bellman-Ford.

And then a lot of examples had linear branching-- in particular, longest increasing subsequence, where we didn't know what the next increasing item was, so there are N possible choices for it. Parenthesization, where we had to choose anybody in the middle as the last operator-- and rod cutting that we saw today-- the first rod we cut could be any length.

And then finally, once you've considered-- recursed on all these subproblems, you have to combine them somehow. And in a lot of the problems, we actually just take one-- the one best choice, and that is our final solution. And in those problems, the final solution ends up being finding some kind of shortest path in a DAG.

But there are a few cases where we actually took multiple solutions and combined them together to get the overall solution. And this includes Fibonacci, which is we added them-- not too interesting. Floyd Warshall, we concatenated two paths. And parenthesization is maybe the most interesting, where we had to take two parts-- the prefix and the suffix-- how to solve them and then multiply or add them together.

And so these problems are not well-represented by shortest path in a DAG-- still a DAG involved, but it's like a multi-path thing. And then finally, the original problem-- often it's just a single subproblem is the original problem. And there are a few examples-- namely, DAG shortest paths, and longest increasing subsequence, and Bellman-Ford, and Floyd Warshall, these were the order that we covered them-- so the three shortest paths, and then longest increasing subsequence, where here, because we added this condition, we had to try all the possible choices for this condition.

Did we also have one here today? No. OK, so in fact, in retrospect-- or we know this from the beginning, but for you, in retrospect, these four DP lectures were all about showing you these mean techniques of dynamic programming, from how to set up simple subproblems to different types of basic subproblems to constraining or expanding those subproblems, and having initially very simple branching, and then getting to bigger branching and different kinds of combination.

We wanted to show you all these ideas in some order. And if you look back at the sequence that we covered problems, we're slowly adding these different main techniques to DP, and that's why we chose the examples we did. There are, of course, many more examples of DP-- very powerful technique. But that's the end.