[SQUEAKING]

[RUSTLING]

[CLICKING]

ERIK DEMAINE: All right. Welcome back to 006 and our dynamic programming quadruple of lectures. We are over halfway through, into lecture three of four. Today, we're going to follow up on this idea of problem constraints and expansion that was mentioned, especially, towards the end of last lecture. We saw an example of subproblem expansion by a factor of 2 in the two-player game with coins where we wanted to have two versions of the game, one where I go first and one where you go first. So this was expanding the number of such problems by a factor of 2.

Today, we'll see a bunch more examples of this idea, including in one setting we've seen already, which is Bellman-Ford, which you can think of as a dynamic program. Maybe it's a good time to mention that Bellman invented dynamic programming in the '50s. Same Bellman in the Bellman-Ford algorithm. This was actually an independent discovery by both of them-- and other people. He invented dynamic programming, and then, a few years later, he applied it to solve the single-source shortest paths problem. We saw them in the other order. We saw single-source shortest paths first, because it was a little easier. And now we're seeing the general framework that this fits into. And so we'll see how that works.

Why did Bellman call dynamic programming dynamic programming? Mostly because it sounded cool, and he was trying to impress government agencies giving him grants. I mean, how can you argue with something as cool-sounding as dynamic programming? But there is some logic to it. Programming is a reference to an old form of this word, which means optimization. And generally, we're trying to optimize things. And instead of optimizing according to some static kind of approach or program, we're doing it dynamically. This is a reference to the local brute force we're doing to optimize at each stage. You can't tell, at the top, what you're going to do in the middle. And so it's kind of-- each subproblem is behaving differently. And so, in that sense, dynamic. And it sounds cool. All right.

Then we'll go to all-pairs shortest paths. We'll see a new algorithm for that that's not asymptotically any better, but it's nice and simple, and another way to-- a cool way to see subproblem expansion. And then we'll look at a couple of sort of practical problems-- parenthesizing arithmetic expressions and a real-world problem, piano and guitar fingering, so assigning a fingering how to play a piece. And we're going to do that with our SRTBOT framework. Quick recollection of what that is. We define subproblems. And we saw how to do that for sequences. We try either prefixes, suffixes, or substrings. We prefer prefixes and suffixes because there's fewer of them. If there's more than one sequence, we take the product of those spaces.

And then the idea we're going to stress today is, we can always add subproblems to make the next step easier. In particular, adding constraints to subproblems, in some sense, lets us remember things about the subproblem that's calling us-- or about the past, is one way to think about it-- or in general, to "remember state." Just by adding more subproblems, we can remember more stuff-- not just what things we're working on, but some context.

And that's what we'll see lots of examples of today. It will hopefully make sense by the end of the lecture. Then, we need to relate these subproblems with the recurrence relation-- actually, we usually just call it a relation. And the key idea here is to come up with a question that, if you knew the answer to that question, you could reduce the subproblem you're trying to solve to smaller subproblem solutions. This question is sort of the fundamental aspect of-- some fundamental aspect of a solution.

Typically, when you're dealing with suffixes, you want to ask some question about the first item, s of i. When you're dealing with prefixes, you want to ask a question about some-- near the last item, s of i minus 1. And for substrings, who knows? Somewhere in the middle. We'll see an example of that today. Once you have-- oh, this is a reveal. Something coming later. Once you have identified such a question, the dynamic programming approach is, don't be smart about how to answer that question-- just locally brute force. Try all possible answers to the question. For each one, recurse, and take the best solution according to whatever metric you're trying to do, typically minimization or maximization.

Another way to think of this local brute force, which I like to think in my head-- so maybe some people like this, some people don't-- is to think about guessing the answer to the question. So maybe the answer could be 0, 1, or 2. And my algorithm will say, guess which is the right answer. And I'll assume that my algorithm correctly guesses the answer and analyze that. So we can think of the program as going straight through-- guessing the answer and then recursing, and then combining the solutions however. But then, at the end, of course, we can't assume that the guess was correct. In fact, we have to loop over all of those guesses. So it's the same thing. Just, if you think of it this way, there's less looping in your program. But when you analyze it, definitely, the loop is really there. You have to pay for all of the possible answers. OK.

Then we need to make sure this relation is acyclic, get a subproblem DAG. And I like to specify an explicit topological order to make that clear. We have base cases for the relation. We have to solve the original problem in terms of these subproblems, and then we analyze the running time, usually, as the number of subproblems times the non-recursive work in the relation. So recursion is free because we're multiplying by the number of subproblems. You can also sum over the subproblems. Sometimes that gives you a tighter bound. And then, of course, we also have to add on the running time we spend in the original problem. All right. That was a quick recap.

Now, one problem we saw in this framework two lectures ago-- the first lecture was single-source shortest paths in a DAG, which, a lot of dynamic programs actually can be reduced to a single-source shortest paths in a DAG. In fact, the reverse is true. Single-source shortest paths in a DAG can be thought of. The DAG relaxation algorithm we saw is essentially a dynamic program where the subproblems are delta of s, v. The relation of delta of s, v is the min. So what are we thinking about here? We're guessing-- this is sort of a new phrase I want to add. Actually, I wrote it right here.

The last edge, u, v, on a shortest s to v path. That's a delta of s, v. The problem we're trying to solve is, find shortest s to v path. And it's some path, and we don't know what it looks like, but some feature of the solution of this path we're trying to find is, well, what's the last edge? It comes from some vertex here, unless the path is of length 0 and s equals v. That's a special case dealt with in the base case. Otherwise, there's some vertex u before v. We don't know what it is, so we're going to locally brute force, or guess what the right answer is.

So look at all incoming edges from u to v, and for each of them, take the recursive shortest path from s to u, plus the weight of the edge. So this is the guessing or local brute force perspective. And the reason this works is because G is acyclic, and so it has a topological order. Otherwise, if the graph has a cycle-- and that's the case we want to go to next-- this recursive call from delta of s, v to delta of s, u will have loops in it. And so you'll never evaluate this recursion. You'll never finish. You'll never minimalize. You'll be sad. It will take infinite time. So don't use this algorithm unless you have a DAG. For DAG, it's great. It's linear time. So that was a little review.

Now, here is a single-source shortest paths in general graphs, which we know as Bellman-Ford, but rephrased into the SRTBOT framework. So we defined this problem, in the Bellman-Ford lecture, delta sub k of s, v. Remember, this was the weight of a shortest path from s to v that is restricted to use, at most, k edges. This made the problem feasible. We ended up taking the product of the graph into all of these different subproblems, in fact. But from the perspective of dynamic programming, we think of this as a subproblem constraint.

What we really want is the shortest s to v path, but that's hard. So we're going to break it up into smaller pieces. For each k between 0 and v, we're going to say, well, let's think about this problem restricted to use, at most, k edges. And for simple paths, if there are no negative weight cycles, we only have to go up to v minus 1. We proved that. And we know we're-- then we're happy. But that's sort of the last thing we want to solve. So if we look down at the original problem we want, it's delta sub v minus 1 of s, v for all v. So if we could solve these subproblems, as we saw with Bellman-Ford, we can solve the original problem-- unless there are negative weight cycles. And we use delta sub v of s, v to check whether they're negative weight cycles. I don't want to repeat that, but that's the all in the original problem.

And then we can take this relation that we wrote for DAG shortest paths and just port it over here. So remember, for a general graph, this has cycles, so we can't use it, because we're just referring to arbitrary delta of s, vs, so there's no reason to expect no cycles there. In fact, the call graph is exactly the graph G, so it has a cycle if and only if G does. But now, over here, I'm running exactly the same formula for this min, but I added a sub k here and a sub k minus 1 here, the observation being, if I've already guessed what the last edge u, v is for this path, then if this whole thing has length at most k, then this piece has length at most k minus 1.

So I only need to solve delta sub k minus 1 of s, v here. And so that's what I wrote-- sorry, of s, u. That's what I wrote here. There's one other thing, which is, as I mentioned a little while ago, it could be that we use fewer than k edges. And so let's consider the case where we don't follow the last edge, and we just add to this min the shortest path using, at most, k minus 1 edges. That's one option for having, at most, k edges. If we wrote equality here, then we would remove that, like the last problem section. OK, good.

The key observation here is that this recurrence does not have cycles. Just by adding this index, if we solve these problems in order of increasing k, all of the references from delta sub care in terms of delta k minus 1. And so this is now magically acyclic. This is why Bellman-Ford worked. But now, I think in a very pleasing way, we're taking our graph acyclic. And by spreading it out over various copies of k and referencing always the smaller one, we get an acyclic graph-- acyclic relations. We have base cases, like normal.

And then we can analyze the running time in the usual way, which is summing-- how much does this cost? We're going to take the cost of computing the relation, which is the number of incoming edges to v-- that's this theta-- and then some overall subproblem. So sum over k and sum over v. Wrote it as a sum instead of a product because this thing is theta E. The sum of incoming edges over all vertices is exactly the size of E. And then we sum-- there's no k in this formula, so we just multiply by v and we get v E. So our good friend Bellman-Ford recast kind of the opposite way. Before, we were using relaxations. Now, we're just writing the min explicitly. But essentially the same computation, just in a different order. OK, cool. Those are reviews of old algorithms, but in this new framework to show how powerful it is.

Let's look at another example, which is our friend-- right here-- all-pair shortest paths. A few lectures ago, we saw Johnson's algorithm, which solves this very well. So one option we could use is just this same set of subproblems, but for all u and v. For u and v and v-- that's what we really care about. And then we could say k up to v. Something like that. So this would work, but it would give the same running time as running Bellman-Ford v times. This is one solution we know, but not the best solution we know, for how to solve shortest paths.

This would give v squared E, which is at most v to the 4th. For dense graphs, it is theta v to the 4th. So this is v times v times v. But worst case is v to the 4th what I would like to show you now is a different way to solve this problem-- almost identical, but gives v cubed running time, which, for dense graphs, is really good. So we're going to reduce this 4 down to a 3. And this is an algorithm called Floyd-Warshall. It's definitely not an obvious algorithm. It's a very cool idea. And it's a nice example of a different way-- remember, this is subproblem expansion. We took the problems we cared about multiplied by this choice of k.

Here, we're going to do the same thing, but define it differently. We're going to define those subproblems differently. First, I want to number the vertices, starting at 1 for convenience. So let's see, y-- and then we're going to define some subproblems, which are delta-- I'll call it u, v, comma k. Maybe, to avoid conflict, I will write a d here. This is just a definition local to this algorithm. So I want the weight of shortest s to v path. So far, just the same as the problem we actually want to solve. This is delta of-- sorry, u, v. But I'm going to add a constraint, which is, using only vertices in u, v and 1 to up to k.

This is the divine inspiration to define subproblems this way. It's a different constraint than the one we saw here, which was using, at most, k edges. So in some sense, what's slow about this algorithm is that we have to loop over all of the incoming vertices. This is expensive. Costs order the degree, which ends up with an E term. We're going to try to convert that E term into a v term by just knowing which vertex to come from-- which sounds impossible.

But it turns out, if you write the subproblems this way-- so I'm naming the vertices. And say, well, let me just find a path that uses u, v and the vertex labeled 1. There's only, like, two options. I could go straight from u to v, or I could go from u to 1 to v. And then, how about with 1 and 2, and 1 and 2 and 3? The same vertices. By label instead of by counting them. Slight tweak, but it turns out this speeds up the algorithm. So let me first tell you how many subproblems there are.

This is v cubed subproblems. Two choices for u and v, and I have v choices for-- sorry, I have v choices for u, v choices for v, and v choices for k, roughly. v cubed of these. But the key thing is-- that sounds like a lot. But the relation becomes cheaper now, because I can just write delta u-- sorry, d of u v k is the min of two things, d of u, v, k minus 1 and d of u, k, k minus 1, plus d of k, v, k minus 1. It's a strange formula because we're using indices like k for vertices also, as well as u and v for vertices, but we're also using k as a counter. But we can do this because our vertices are numbered. OK.

So the idea is the following. We have vertex u, have a vertex v. And we've already found the shortest path that uses vertices 1 through k minus 1. That's what this quantity is. So it could be-- and now, we're trying to add in this vertex k and think about, what are all the paths that could go from u to v using 1 up to k? Well, it could be the shortest path from u to v using 1 up to k doesn't use vertex k. That's the first option. Just use vertices 1 up to k minus 1. Maybe we don't need to use k. Or it could be that the path goes through k.

Well, it starts at u, and it goes to k, and then it goes from k to v, all using-- for simple paths, if we assume there are no negative weight cycles and you have to run Bellman-Ford to detect them-- but assuming no negative weight cycles, the shortest path from u to v through k must start by using vertices less than k and then use vertices less than k again. And that's what I've written here. It's from u to k using k minus 1 up to k minus 1 labels, and from k to v using labels up to k minus 1. The cool thing here is, the min only has two terms. So this takes constant time non-recursive work. So this is, k is not on the shortest path. And this is, k is on the shortest path. Cool. So this is constant non-recursive work.

If we jump ahead to the running time for this algorithm, we get the number of subproblems, which is v cubed, times the amount of work per subproblem, which is constant. So that's just a v cubed algorithm. For dense graphs, this is really good. When E is v squared, then this is the same thing as v times E, so as good as Bellman-Ford. It's not as good as Johnson for sparse graphs. Sparse graphs, remember-- or, in general, with Johnson, we got v squared log n plus v. And this is always spending v cubed. But it's cool because it's a very simple algorithm. Let's quickly write the topological order. All we need is to guarantee that we solve problems in increasing k order, because every reference here is to a smaller k for the third argument.

And so, for example, you can just write a triply nested loop-- k equals 0, 1 up to v, and u and v, and v and v. So if you wanted to write this algorithm bottom-up, you just write this triple four-loop and then plug in this recurrence relation here. And think of d as a table, as a set mapping, instead of as a function call. And boom, in four lines, you've got your algorithm-- except you also need a base case. The base case here is u, v, 0. So I have to define what that means. But when k equals 0, the 1 through k set is empty. So all I'm allowed to use are my vertices u and v.

There are three cases for this-- 0 if u equals v. It's w of u, v if there's an edge from u to v. And it's infinity otherwise. OK. But easy base case, constant time for each. And then the original problems we want to solve are delta u, v, size of v. Because I remember the vertices 1 through size of v. And if k equals size of v, that means I get to use all of my vertices. So that is regular shortest paths. This is assuming no negative weight cycles. We already know how to do negative weight cycle detection, so I'm not going to talk about that again.

But then, this will be my shortest pathway because-- yeah. I implicitly assumed here that my path was simple because I imagined that I only use k once-- 0 or 1 time. And that's true if there are no negative weight cycles. Cool. And we already did the time part of SRTBOT. So v cubed algorithm. Very simple. Basically, five lines of code, and you've got all-pairs shortest paths. And if your graph is dense, this is a great running time. If your graph is not dense, you should use Johnson, like you will in-- or like you have implemented in your problem set. Yeah.

AUDIENCE: So how does this this compare to just running Dijkstra's algorithm a bunch of times?

ERIK DEMAINE: What about using Dijkstra? Let's compute. Running Dijkstra v times is the running time of Johnson. Running Dijkstra a bunch of times is great if your graph has only non-negative edge weights. Then you should just run Dijkstra. You get this running time. And for sparse graphs, this is superior. If you have negative edge weights, you should run Johnson, which is Bellman-Ford once and then Dijkstra three times. And we're comparing this to v cubed, which we just got.

So this-- I mean, how these compare depends on how v and E relate. On the one hand, maybe v is theta E. That's what I would call a very sparse graph. And it's quite common. Then, the running time we get here is v squared log v-- roughly v squared. On the other hand, if we have a very dense graph, v is theta E squared-- which, for simple graphs, is the most we could hope for-- then this running time is v cubed. If you know the v is near E squared, then this is giving you v cubed anyway, from the vE term.

So why not just use this algorithm? And often, you know, a priori, whether your graph is very sparse or very dense or somewhere in between. If it's somewhere in between, you should still use Johnson's algorithm, because you're going to get the benefit from sparsity and only have to pay this vE instead of the v cubed. But if you know ahead of time, constant fraction of the edges are there, then just use-- or you have a small enough graph that you don't care, just Floyd-Warshall because it's simple and fast.

Good question. Any other questions? This is an example of subproblem expansion-- a very non-intuitive one, where we use some prefix of the vertices. But notice, it's prefixes again-- a number of the vertices from 1 up to v. And I took a prefix of those vertices. So I just solved the problem using prefix vertices 1 through k. So it's actually a familiar idea. If all you had seen are all dynamic programming examples of prefixes, suffixes, substrings, actually, it's pretty natural way to solve shortest paths-- maybe even more natural than this. Anyway, all right.

Enough shortest paths. Let's solve two more problems that are more in our standard wheelhouse that will involve sequences of inputs, not graphs. First one is arithmetic parenthesization. First, let me define this problem. OK. We are given a formula with, say, plus and times. Let me give you an actual example-- 7 plus 4 times 3 plus 5. Now, when you read this, because you've been well-trained, you think, OK, I'm going to multiply 4 and 3 first because that has a higher precedence, and then I'll add the results up.

But what I'm going to let you do is parenthesize this expression however you want. For example, you can add parentheses here and here. You must make a balanced parenthesis expression, a valid way to pair up-- or not just pair up, but a valid way to evaluate this expression. Any order you want. I could be inconsistent. I could, for example, do this sum, and then do this product, and then do this sum. But some kind of expression tree over this. And each one evaluates to something. This is 11 and this is 8, and so this is 88. And my goal is to maximize that computation. And I claim that this is the way to maximize that particular example. Let me write it in general, and get my notation to match my notes.

Given a formula a 0, star 1, a 1, star 2, a 2, and so on up to star n minus 1, a n minus 1, where each a i is an integer, as we like in this class, and each star i is either plus or times. OK so I'm using star as a generic operator. I chose star because it is the superposition of star on top of a times symbol. So it's clear. So you're given some formula, any mixture of plus and times that you like, involving n integers. And your goal is to place parentheses to maximize the result. So you can try all of the combinations here.

If I, for example, take the product of 4 times 3, I get 12. If I do that first, I get 12. Then if I add 5 and 7, I get 24, which is less than 88. And I check them all, and this one is the maximum for that example. Interesting problem. It's a bit of a toy problem, but it's motivated by lots of actual problems, which I won't go into here. To apply this framework, we need to identify some subproblems. This is a sequence problem. We're given a sequence of symbols. And so the natural thing is to try prefixes, suffixes, and substrings.

I'm going to jump ahead and think about the relation first. I want to identify some question about a subproblem or its solution that would let me reduce to smaller subproblems. This is a little trickier. This is very different. We're not always doing something on the left or on the right, or we can't assume there's something happening on the left, because maybe we take a product in the middle first. If I take a product in the middle first, then I have some result here, but I still have three things. I have the thing to the left, I have the thing in the middle, and I have the thing on the right. It turns out to be very messy to think about what the first operation is. Because we can think of this as a tree, where we take a product here-- we take a sum of 7 and 4 and 3 and 5 over here and then take the product at the root.

But I don't know what the tree is, right? I only know these numbers and these operators, but I don't know how to organize this tree. The idea is, if you think of this tree, what is the one thing that's easiest to identify? It's the root. The root corresponds to the last operation I do in this computation. The last thing I did was take a product. And that's a lot easier, because if I guess who is at the root-- which operator is at the root-- that naturally decomposes into the left subtree and the right subtree. And those will always be substrings. We kind of know this. This node corresponds to everything left of this operator, and this substring or this subtree corresponds to everything to the right of the operator.

So this is our idea, is we're going to guess which operation, star i, is evaluated last-- or, in other words, at the root. So this is the question. It has n possible answers-- I guess, actually, n minus 1 from operator 1, operator n minus 1. And so we'll just brute force all of those choices. I wanted to start here because-- to realize that if, I choose some star i in the middle, which might be the right thing, like in this example. Star i is the middle one-- middle operator. I naturally decompose into everything to the left of that operator and everything to the right of that operator. This is a prefix. This is a suffix.

So you might think, oh, my subproblems are all prefixes and all suffixes. But that would be wrong, because if you have a bunch of operators-- and say you choose this one to be last. So I have a prefix here and a suffix here. And then there will be some-- within this suffix, I'll choose some operator to be the root of that one, and I have a prefix and a suffix of this suffix. But in particular, I will have to evaluate this subproblem, which is a prefix of a suffix-- in other words, a substring. So never use a mixture of prefixes and suffixes. If you need both, you probably need all substrings. So our subproblems are going to be substrings. OK.

I'm not going to write the subproblems quite yet, because there's another idea we need. So what do I need to do with the substring? I'm going to guess the middle operator and then evaluate the left substring, evaluate the right substring. What am I trying to do with those substring? I guess I'm trying to solve this problem, which is, place parentheses in order to maximize the result, and then return what the result is. And I can use paren pointers to reconstruct what the parentheses actually are.

Once I guess what the last operator is, it enough to maximize the part to the right and maximize the part to the left? Will that always maximize my sum or product according to what this operator is? And if you think about it for a while. Yeah. If I want to maximize the sum, I should maximize the two parts. And if I want to maximize a product, I should maximize the two parts. That seems right. Except, I didn't say that my integers are positive. That's true if your integers are positive. But to make this problem more interesting, we're going to allow the integers to be negative. For example, 7 plus minus 4 times 3 plus minus 5. So I just added a couple of minuses to a couple of the numbers here. Then it's no longer best to pair them this way.

If I pair them this way, like this, or if I add parentheses this way, I get 3 here, and I get minus 2 here. So I get-- the product of that is negative 6, which i probably not the maximum. In fact, I can do better, I believe, by doing the left operator last. So this, I claim, the best parenthisization, if I remembered it correctly. This is, minus 2 times minus 4 is 8, plus 7 is 15. So I got a positive number-- definitely better than the negative number I got. I claim this is the best. And the key property here is, when we take a product of two negative numbers, we get a positive number.

Sometimes, you actually want to make things small, because small might mean very negative. You take two very big negative numbers-- very small negative numbers, in other words. You take their product, you get a very big product, positively, because the signs cancel. OK. So this seems tricky. We want to work on substrings, but we don't know whether we're trying to maximize, or you might think, well, maybe I'm trying to maximize the absolute value. But that's not good. Maybe overall, on this entire expression, I get negative 1 million. And that's not what I wanted. I wanted to maximize the sum.

So I still need to solve the max evaluation that I can get, the max parenthesization, but I also need to solve the min parenthesization. If I can solve max and min, I'll know the entire range that I could get. And I really only-- I'll care about min especially when it lets me go negative. But let's just solve, in all cases, the min and the max, and then just brute force the rest. That's what I'm going to write down. So that was some motivation and why we are going to define subproblems this way.

I'm going to define x of i, comma j, comma opt to be-- opt, here, is going to be either min or max. And this is my subproblem expansion. I really just care about max at the very end, but I'm going to care about min along the way. And i, j is going to specify my substring. So this is going to be the opt value-- opt stands for "optimum" here, or "optimization." The opt value I can get for the substring a i star plus 1, a i plus 1, and so on to star j minus 1, a j minus 1. OK.

Being careful to get my indices correct here. And I want 0 less than or equal to i, less than j, less than equal to n. I claim and opt like this. OK. I'm going to get the min value and the max value separately. Those are two different subproblems. This is my expansion. This is the constraint I'm adding. And I'm only focusing on this substring from i inclusive to j exclusive. OK. So I claim those are good subproblems. Let's write down a recurrence relation. OK. Relate.

I want to write x of i, j, opt on the left. And I want to optimize-- so this will be min or max-- on a set of choices. What is my set of choices? Well, like I said, I want to guess what is the last operation evaluated. I wrote star i here, but star i is already defined, so I'm going to use star k. So I'm going to guess which of my operations between i plus 1 and j minus 1 is the last one, and I evaluate. And that decomposes everything left of k. So that would be x of i, comma k, comma something. And then we will do operator star k on the part after k, which is from k to j, something.

And I'm choosing between-- I think it's i less than k less than j. k is some operator in between, because I started i plus 1 and I ended j minus 1. So those are the possible choices for k. I tried them all. That's my local brute force. And then I take what I can get on the left, what I can get on the right, and multiply or add them according to whether the operator is plus or times. Now, should I maximize or minimize this one? Should I maximize or minimize this one? I don't know. So I'm just going to do more local brute force. Well, let's just say opt prime for the left-- or maybe I'll call it opt L for the left and opt R for the right part. And I'll just add this to my four-loop. Let's just try opt L and opt R. Just take all possible choices among min and max.

Now, you could think hard-- and for addition, for example, if you're maximizing, you really only need to maximize the two parts. And if you're minimizing, you can prove you all need to minimize the two parts. But for multiplication, it's messy. It could be, really, any of the options. Because sometimes, when you minimize, you get a negative term. Sometimes, you don't. And so it depends what you're trying to do. You have to consider all the signs. But we don't need to think hard. We can just try all options. There's only four choices for opt L and opt R among min and max. You could do min-min, min-max, max-min, and max-max.

So try-- it's just a multiplication by 4 in this four-loop. The big cost is actually this one, because there are j minus i choices for k. There's a constant number of choices for opt L and opt R. And you need to prove that this is correct. I won't do it here. But the idea is, if you're trying to minimize or maximize your sum or product, it's enough to know what ranges these could come in. And the optimal choice will always be an extreme in that range. We consider all of them here. And so we get this recurrence. Now, it needs a base case, and we need to check that it's acyclic. But topological order is just increasing j minus i.

This is the usual order for substring problems, because this is increasing length of the substring. So start with very tiny substrings. Here, we'll start with length 1 substrings. We just have an a, i there. So that's going to be our base case. And you grow up to the entire string. And it doesn't matter how we order relative to opt as long as we are increasing in j minus i, because i to k and k to j will always be strictly smaller than i to j, and so this will be acyclic. The base case is x of i, i plus 1, opt. This is always a i. Doesn't matter what opt is, because there's nothing-- there's no choice. You just have a single number in that substring, because we're exclusive on i plus 1.

And then the original problem we want to solve is x of 0, n, max. You could also solve min and see how small you can get it. So if you wanted to maximize the absolute value, you could solve the max problem and the min problem and take the largest of those two options. And how much time does this take? Well, how many subproblems are there? For substring problems, we have n squared subproblems. Now, we multiply the number of subproblems by 2, but that's still n squared. So we have n squared subproblems.

And how much work per subproblem are we doing? Well, as I mentioned, we're doing j minus i choices for k and a constant number of choices for opt L and opt R. So this is theta j minus i, which, if I'll be sloppy, that's at most big O of n. And it turns out to be the right answer anyway. So there's a linear amount of non-recursive work. In fact, it's like a triangular number, but that's still theta n cubed. Same running time as v cubed we just got. But polynomial time.

And this is pretty impressive, because we're really brute forcing all possible parenthesizations. There about 4 to the n, exponentially many, parenthesizations of an expression. But we're finding the biggest-- the one that evaluates the largest value and the one that evaluates to the smallest value in just n cubed time-- polynomial. And a key here was subproblem expansion, where we, in addition to solving the max problem, we also solved the min problem, because sometimes, you want to take two very small negative numbers and product them together to get a larger positive number. Cool. Question?

AUDIENCE: Would anything go wrong if I added minus or divide?

ERIK DEMAINE: So what if I had operators minus and divide? It's a good question. I'm certain that minus should work fine. If we do min and max, this should still evaluate the largest thing for division. I need to think about the cases. I would guess it works, but what we need to prove is that the way to maximize or minimize a division, say, given two numbers in the left and right, is that it either corresponds to maximizing or minimizing the thing on the left and then maximizing or minimizing the thing on the right. So as long as you have this kind of-- it's not exactly monotonicity.

It's just that, in order to compute max or min, it suffices to know the max and min of the two parts. It's like interval arithmetic. You know, interval arithmetic? I want to know, what are the extremes I can get on the output of a division if I'm given that a number is in some interval here and some interval here? If the answer is always, use one of the extreme endpoints here and use one of the extreme endpoints here, then this algorithm will work. Otherwise, all bets are off. Cool. So if you negate-- if you put a minus here, that will work fine, because it's negating this range. And then it's just like sum. But--

AUDIENCE: [INAUDIBLE]

ERIK DEMAINE: Oh, a divider-- if you're careful about 0, yeah. Actually, it doesn't work, because we care about how close this can get to 0 for division. It might be enough to consider those. It's like, instead of minimizing and-- instead of computing this entire interval, if this interval spans 0, maybe I need to know-- if 0 is here, I need to know how close to 0 I can get on the left side and how close to 0 I can get on the right side. Still just four quantities I need to know. I would guess, for division, that's enough. Yeah. Nice. Solved a little problem. Then, we would be multiplying the subproblem space, instead of by 2, by 4.

Hey, maybe we should put this on the final. No, just kidding. Now it's in lecture, so we can't use it. But it's a cool set of problems, right? You can do a lot with dynamic programming. You don't need to be that clever, just brute force anything that seems hard. And when it works, it works great. And this class is all about understanding when it works and when it doesn't work. Of course, we will only give you problems where it works. But it's important to understand when it doesn't work. For example, DAG shortest paths-- that algorithm on a non-DAG, very bad. Infinite time. OK.

Our last example is piano fingering. Here, we're given a sequence of notes t 0, t 1-- t for note-- up to t n minus 1. These are single notes. And all of the single notes-- all of the single notes, right? And we have fingers on our hands. This is not like two-finger algorithm. This is the five-finger algorithm. So in general, I'm going to assume an arbitrary anthropomorphic object. So this is 5 for humans-- most humans. Some humans-- I think the maximum on each hand is 7. Could be smaller. Maybe you've had an accident.

I'll solve it for arbitrary F. And what we would like to do is assign fingers to notes to tell our pianist which finger to use for each note. Normally, when you're given sheet music, it just gives you a sequence of notes you want to play. It doesn't tell you which finger you want to play it with, unless you have some nice training booklets and they have a little number on top of each. And I number them 1, 2, 3, 4 or 5, and symmetrically, 1, 2, 3, 4, 5. Here's a giant piano for my giant hands. If Jason were here, he could sing these notes. So maybe I play this with my first finger, this with my second finger.

Let's just say I'm doing a scale. So I can walk. And now, I think, the typical way to do a scale is to reach over with your first finger-- second finger, I guess, and then do something like this. No? OK. Clearly, I don't know scales or how to play a piano. But there's limits here. I can-- if I'm going from this note and I want to go to another note over here, OK, maybe I have a decent span from first finger to fifth finger, but my span for my first finger to my second finger is not as big. I can't reach as far. So if I want to play this note and then this note, I would like to start here with a very extreme figure on the left and then go to at a very extreme figure on the right.

I'm going to formalize this problem pretty abstractly, because I don't want to get into music performance. I'm going to say that there is a metric d for, if I'm at note t with finger f, and I want to go to note t prime with finger f prime, then this function, d of t, f, t prime, f prime, gives me the difficulty of doing that-- the difficulty of playing note t with finger f and then playing note t prime with finger f prime. This w is "with." So this is a transition difficulty. I'm not going to worry about the difficulty of the whole piece other than saying, well, I've got to play this note, then I've got to play this note.

And for now, just single notes. You play a single note with your right hand, then you play another single note with your right hand, then another single note with your right hand. Let's assume no pauses for now-- no rests. Great. So we have this difficulty from going from the ith note to the i plus first note. And then our goal is to minimize the sum of difficulties. Minimum sum of d ti fi ti plus 1, plus 1. And these-- f i's and f i plus 1 is what we want to compute. We don't know which fingers to use. We're only given which notes to play. This is actually a natural problem. There are lots of papers about this problem. I've read a bunch of them-- obviously not super well, but how to play scales.

But there are notes like-- there are constraints in this-- usually, people write this metric as a sum of different penalty terms, if I want to minimize difficulty. Difficulty is high if I play a note far on the left. So if I go from a low note to a high note, that's easier to do if I use a lower-numbered finger and go to a higher-numbered finger. You don't want to go, like I was doing, from a high-numbered finger to a low-numbered finger to play a note on the right. That's annoying. I would like to do an assignment, if I can, that avoids that. So I'll just have some penalty of, like, 100 if that happens, and 0 if it doesn't happen, sum up a bunch of terms like that.

Other examples are, avoid the fourth and fifth fingers-- weak fingers-- or, if I'm playing a portion of the song that is legato, then I don't want to use the same finger to play two notes right after the other. I've got to use two different fingers for that. So you have a penalty if f i equals f i plus 1 and these two notes are not the same. Then-- and we're in legato mode-- then we add a penalty term. And things like that. I would prefer-- if I'm going from a very low note to a very high note, I would like to use more extreme fingers. Things like that. But we're just going to assume this d function is given to us. It's some polynomial size. If you imagine the notes on your keyboard are end notes or m notes, then some polynomial and m size to this function.

How do we solve this problem? I'm running low on time, so let me give you the idea. And this is going to use subproblem expansion. So the subproblems are going to be x of i, comma f-- this is the minimum total difficulty to play suffix-- because I like suffixes-- t i up to t n minus 1, starting with finger f on note t i. The obvious subproblems would be without this constraint. This here is a problem constraint. And you could try to define the subproblems just as, what's the best way to play a suffix? But I claim it's important to know which finger we start with.

So we're going to multiply the number of subproblems by capital F, which is just 5. So a very small subproblem expansion. And then we're going to constrain these subproblems to say, well, what if I started with my first finger? What if I started with my second finger? What if-- up to the fifth finger. Try them all. Then, it turns out, I can write a relation, which is X of i f equals the min. What should I min over?

I'll just guess this-- I'm already told what my first finger is to use-- which finger I should use for t i. So what's the next thing that matters? Well, I guess what finger to use for the t i plus 1, the very next note. What is the next finger I use? I will call that f prime and minimize over f prime, between one and capital, F of the remaining suffix, starting with f prime, plus my difficulty function from t i, comma f, to t i plus 1, comma f prime. End of bracket. OK.

So there's kind of a lot going on here. This is a lowercase f prime. But actually, if you think about what I would like to write the recurrence on, well, I start with the suffix i, I would like to recurse on the smaller suffix, so that's X of i plus 1. So here, if I know that I'm prioritizing my finger number for i, will then I'm in order to even call this function, I need to know what finger I'm using 4 plus 1. Once you decide on these subproblems, it's really obvious you need to guess, what is the next finger, and then recurse on that finger-- recurse on the remaining suffix of that finger.

Now, why did we need to know what these fingers were? Why not just guess what the first finger is? Well, it has to do with this difficulty function. For this difficulty function, I know that I want to measure the difficulty from t i to t i plus 1. And to do that, because this function is parameterized by four things, I need to know both the finger for t i and, at the same time, the finger for t i plus 1. If I remove this f, I could add a min over one finger, but I can't really add a min over two fingers.

So what this does, by parameterizing by f here and writing down the optimal for each starting finger f, I can-- in some sense, I'm remembering, in this call, what finger I started with. Because I told it, you have to start with finger f prime. And so locally to X i, f, I know what finger f prime is being used for t i plus 1. And also, because of the definition of X i, f, I know what finger I'm using for t i. And so I get to know both of these fingers. One comes out of this min and the other is given to me as this parameter.

And then, of course, if I can solve these problems, I can solve the original problem by just one more min of 1 up to capital F of x 0 little f. I don't know which finger to start with, but that's just f choices. And so then, this recurrence gives me the overall solution. And I'll just jump to the time. We need a base case and topological order. But it is n f squared time. There are n times f subproblems here. And for each one, I'm doing an optimization over f choices, so I get n times f squared. It's a polynomial. And if f is a constant, this is actually linear time. Very fast DP.

Now, what I described here is for one hand, one note at a time. But you can generalize this to two hands, each with one note. Well, that's just ten fingers. So you could solve this separately for the right hand and left hand if you know which notes are being played with left hand and right hand. But some pieces, that's not obvious. So to make it more interesting, what if you have multiple notes at the same time? And let's say-- I think it's reasonable to say you can only play up to f notes at a time, 1 for each finger. And so you have an upper bound on the number of notes at a time that we're playing, which is good.

Oh, I have a drawing of this DP, by the way, as a subproblem DAG. This is the original problem, which is, we don't know which finger to start with. But then we just have a complete bipartite graph here, where we write on each of these edges, what is the difficulty of this transition? The y-axis here is which finger I'm using-- 1, 2, 3, 4, 5-- and the x-axis is which suffix I'm considering. Which note do I start on? And so you could solve this with shortest paths in a DAG, or you could just solve it directly as DP, either top-down or bottom-up. OK.

Jumping ahead, if you do multiple notes at a time, instead of this finger choice, which just had f choices, we have-- what do I write here? t to the f possible states, where t is the maximum number of notes that I could play at once. And usually, this is at most f, 1 for finger. But we could generalize. And this is number of fingers. So I could deal with all 10 fingers of a typical human set of arms-- hands-- and say there's at most 10. And so this is 10 to the 10. It's a big constant, but it's constant-- like 1 billion right? 10 billion? But then it's that times n. And maybe that squared times n will let me exhaustively enumerate all of the possible things I could do be doing with all of my hands.

You can apply this not only to piano fingering, but also to guitar fingering-- also to *Rock Band. Rock Band* is an easy case where you just have five buttons, and usually only four fingers that you use. And this doesn't really make any sound, so it's not that exciting. So the case where f equals 4 you can apply to optimally figure out which finger-- once you have a difficulty function of what transitions are easy and hard for *Rock Band,* then you can optimally figure out your fingering for *Rock Band* songs. With a real guitar, this is a little bit harder because there are actually multiple ways to play the same note.

For example, I can play the note of this string like this. These should both sound the same-- if my guitar were perfectly tuned, which it's not. And that's properties of strings and the way these things are played. So in addition to what finger I use, I should also decide which string to play that note on. If all I'm given is sheet music, different notes to play, another thing I could guess is which string to play that note on. So for example, maybe I want to play my favorite song here. [PLAYING MUSIC]

AUDIENCE: [APPLAUSE]

ERIK DEMAINE: *Super Mario Brothers,* my favorite song. I could keep going, but I actually can't keep going. It won't be as impressive. I don't actually know how to play guitar. But there are a lot of choices there, right? I started with playing this note down on this string. That's good. I could have also played it on this string. But that's more work for my finger, so I have a penalty function that says, well, if I play an open string, that's a lot easier.

And then I had a transition from here-- this note [PLAYING A NOTE] to this note [PLAYING A NOTE] to this note. [PLAYING A NOTE] And if you focus on my fingering here, I chose to use my index finger for the first one, because that index finger is always the easiest to use, but it also gives me lots of room to move my pinky over to here. And then I like to use my middle finger to come up here. You could also use your index finger. It's just a little bit more of a reach.

So you have to define some difficulty function. It might depend on how big your fingers are. But you can do that, and then optimize, using these algorithms, what is the best guitar fingering for a given piece-- one note a time or several notes at a time. You could even add in parameters like, oh, maybe I want to play a bar. That's not a perfect bar. But this would be great for my playing this note down here and this note here-- bad example. So you could do other things with a guitar to make it more interesting and generalize this dynamic program suitably.

But I think this gives you a flavor how, with subproblem expansion, I can capture almost any aspect of a problem that I want. As long as the number of states that I need to keep track of is small, I can just multiply the number of subproblems by that state, and I can keep track of any kind of transition from one state to another, which I could also do with taking the product of a graph. But dynamic programming gives you a kind of methodical way to think about this, by figuring out some property-- in this case, the state of how my fingers are applied to the instrument-- and then just sort of brute forcing the rest-- a very powerful framework.