[SQUEAKING]

[RUSTLING]

[CLICKING]

ERIK DEMAINE: All right welcome back to 006, Dynamic Programming. We're now in step two out of four-- going to see a bunch more examples-- three more examples of dynamic programming-- longest common subsequence, longest increasing subsequence, and kind of a made up problem from 006, alternating coin game.

And through those examples, we're going to explore a few new ideas, things we haven't seen in action before yet-- dealing with multiple sequences, instead of just one; dealing with substrings of sequences, instead of prefixes and suffixes; parent pointers so we can recover solutions, like in shortest paths. And a big one, which will mostly be fleshed out in the next lecture, is subproblem constraint and expansion.

This is all part of the SRTBOT paradigm-- remember, subproblems, relations, topological order, base case, original problem, and time. Here is subproblems and relations. I've written down both what these things are and the key lessons we got from the first dynamic programming lecture. Namely, we want to split up our problem into multiple subproblems, and if your input is a sequence-- that's the main case we've seen so far-- like the bowling problem, for example, then the natural subproblems to try are prefixes, suffixes, or substrings.

Prefixes and suffixes are nice, because there's few of them. There's only a linear number of them. In general, we want a polynomial number. Sometimes you can get away with one of these. They're usually about the same. Sometimes you need substrings. There's quadratically many of those.

Then, once you set up the subproblems, which-- it's easy to set up some problems, but hard to do it right-- to test whether you did it right is, can I write a recurrence relation that relates one subproblem solution to smaller subproblems solutions? And the general trick for doing this is to identify some feature of the solution you're looking for.

So you're trying to solve some subproblem, and you try to ask a question whose answer would let you reduce to a smaller subproblem. So if you can figure out what that question is, and that question only has a polynomial number of answers, then boom-- and you've only got polynomial number of subproblems, then you will get a polynomial running time.

So I have, I think, another way to say this. We just locally brute force all possible answers to whatever question we come up with, as long as there's polynomially many. And then each of them is going to recursively call the smaller subproblems, but because we memoize, we'll only solve each subproblem once. And so on the end, the running time will be, at most, the number of subproblems times the non-recursive work done in that relation.

For that to work, of course, the relations between subproblems must be acyclic, so we'd like to give an explicit topological order. Usually it's a couple of for loops. But this is a topological order in the subproblem DAG, which I defined somewhat informally last time. the. Vertices are subproblems, and I want to draw an edge from a smaller problem to a bigger problem, meaning that, if evaluating b in this relation calls a, then I'll draw an arrow from a to b, from the things I need to do first to the things I'll do later.

So then topological order will be ready-- by the time I try to compute b, I will have already computed a. And of course, the relation also needs base cases. And then sometimes the original problem we want to solve is just one of those subproblems. Sometimes we need to combine multiple. We'll see that today. So that's a review of this framework.

And let's dive into longest common subsequence. This is kind of a classic problem. It even has applications to things like computational biology. You have two DNA sequences. You want to measure how in common they are. One version of that-- might see other versions in recitation-- called edit distance-- this is a simplest, cleanest version, where I give you two sequences-- I have an example here.

So for example, it could be a sequence of letters. So my first sequence spells hieroglyphology-- study of hieroglyphs. And second sequence spells Michelangelo. And what I'd like is a subsequence. So remember, substring has to be some continuous range, some interval.

Subsequence-- you can take any subset of the letters in your sequence or any subset of the items in your sequence. So you can have blanks in between. You can skip over items. And so what we want is the longest sequence that is a subsequence of both the first string, the first sequence, and the second string. And if you stare at this long enough, the longest common subsequence-- I don't think it's unique, but there is a longest common sequence, which is hello hiding in there.

And that is a longest common subsequence. So given that input, the goal is to compute hello, or whatever the longest common subsequence is. So we're given-- write this down carefully-- given two sequences. Let me name them A and B. We want to find the longest sequence L that's a subsequence both A and B. So that's the problem definition, and we're going to see how to solve it using dynamic programming.

And whereas, in the bowling problem, we just had a single sequence of numbers-- the values of the bowling pins-- here we have two sequences. And so we need a new trick. Before, we said, OK, if our subproblems-- or sorry-- if our input consists of a single sequence, we'll try prefixes, suffixes, or substrings.

Now we've got two sequences, so somehow we need to combine multiple inputs together. And so here's a general trick-- subproblems for multiple inputs. It's a very simple trick. We just take the product, multiply the subproblem spaces.

OK. In the sense of cross product of sets, and in particular, from a combinatorial perspective-- so we have two inputs, the first sequence A and the second sequence B. For each of them, there's a natural choice, or there's three natural choices. We could do one of these. I will choose suffixes for A and suffixes for B. You could do some other combination, but that would be enough for here.

And then I want to multiply these spaces, meaning the number of subproblems is going to be the product of the number of suffixes here times the number of suffixes here. And in other words, every subproblem in LCS is going to be a pair of suffixes. So let me write that down.

So for LCS, our subproblems are L of i, j-- this is going to be the longest common subsequence-- of the suffix of A starting at i and the suffix of B starting at j. And just to be clear how many there are, I'll give the ranges for i and j-- not going to assume the sequences are the same length, like in the example.

So I'll write lengths of A and lengths of B. I like to include the empty suffix. So when j equals the length of B that's 0 items in it, because that makes for really easy base cases. So I'd like to include those in my problems. So that was the S in SRTBOT. Now I claim that set subproblems is enough to do a relation-- recursive relation among them.

So I'd like to solve every subproblems L i, j. Relation is actually pretty simple, but it's maybe not so obvious. So the idea is, because we're looking at suffixes, we should always think about what happens in the first letter, because if we remove that first letter, then we get a smaller suffix. If you're doing prefixes, you should always look at the last letter. Either one would work for this problem.

So we're doing suffixes. So we look at A of i and we look at B of j. That's the first letter in the suffix A starting at i and the suffix B starting at j. And there are two cases. They could be equal or different. I think the easier case to think about is when they're different.

So like in hieroglyphology and Michelangelo, if we look at the whole string, say, the first letter in the top one is H. The first letter in the second one is M. Those are different letters, so clearly, one of those letters is not in the common subsequence, because they don't match. I can't start with an H. Well, I can start with an H, I could start with an M, but I can't start with both.

One of those letters is not in the output. In this example, it's M. But I don't know which one, so I have this question. I want to identify some question. And the question is, should I-- do I know that the H is not in the answer or do I know that M is not in the answer-- the final longest common subsequence? We don't know which, so we'll just try both.

And then we're trying to maximize the length of our common subsequence, so we'll take the max of L i plus 1 j and L i, j minus 1. So the intuition here is one of-- at least one of Ai and Bj is not in the LCS. Got this wrong. j plus 1-- sorry-- thinking about substrings already.

Yeah. These are the beginning points, so I want exclude the i-th letter-- so if Ai is not in, then I want to look at the suffix starting at i plus 1. If Bj is not in, then I want to look at the suffix starting at j plus 1. So the indices are always increasing in the function calls.

And the other case is that they're equal. So this one I have a little bit harder time arguing. I'm going to write the answer, and then prove that the answer is correct. Here I claim you don't need to make any choices. There's no question you need to answer. You can actually guarantee that Ai and Bj might as well be in the longest common subsequence.

And so I get one point for that and then I recurse on all the remaining letters-- so from i plus 1 on and from j plus 1 on. Why is this OK? Well, we have A, B. We're starting at position i, and starting at position j for B. Think of some optimal solution, some longest common subsequence. So it pairs up letters in some way. This would be some non-crossing pairing between equal letters.

So first case is that maybe i and j aren't paired with anything. Well, that's silly, because if they're not paired with anything, you have some bearing on the rest of the items. You can add this pair, and that would be a longer subsequence. So that would be a contradiction. If we're taking-- imagining some hypothetical optimal solution, it has to pair one of these with something.

Maybe it pairs i with something else, though. Well, if we have a longest common subsequence that looks like that, I can just instead pair i with Bj. If I had this pairing, I'm actually not using any of these letters, so why don't I just use this letter instead? So you can argue there is the longest common subsequence that matches Ai with Bj, and so then we can guarantee by that little proof that we get one point for matching them up-- that we don't have to max this with anything.

OK, so two cases-- pretty simple formula. And then we're basically done. We just need to fill in the rest of SRTBOT. So next is topological order. So this I'll write as for loops.

Because I'm dealing with suffixes, we want to start with the empty suffixes, and then work our way to larger and larger suffixes. So this might seem backwards. If you're doing prefixes, it would be an increasing order. There's all sorts of orders. You could flip the i and j. It's very symmetric, so it doesn't really matter. But anything that's generally decreasing i and j is good.

Then we have base cases. These are when one of the sequences is empty. I don't care how many items are in B, but if A has no items, there's no common subsequence. It's empty. And same for no matter how big A is, if I have exhausted the B string-- I start from beyond the last item-- then I should get 0.

Then the original problem we want to solve is L 0, 0-- that's just the longest common sequence of the entire A and the entire B-- and time. OK, so for time, we need to know how many subproblems there are. It's A plus 1 times B plus 1. I'll just call that theta A, B. Assume these are not empty subsequences.

So this is the number of subproblems. And then what we care about is, how much time do we spend for each sub problem in evaluating this recurrence relation? So we ignore the cost to recursively call these L's, because they are smaller subproblems. They're already dealt with when I multiply by the number of subproblems.

So I just care about this max computation and this equality check, and we'll say those each cost constant time. So this is quadratic time. If the two strings are size n, this is n squared. In general, it's the product of the two string sizes. And that's longest common substring-- so pretty straightforward.

Other than this little argument understanding the case when they're equal, the easy case where they're unequal, we just try the only two things we could do. One of Ai and Bj is not in the longest common subsequence, so what I like to say is we guess which of those is the correct one that is not in the longest common subsequence.

And if we guess that it's in A, then we'll recurse on that side. If we guess that it's in j, then we'll recurse on-- by increasing j. I'd like to assume-- not really-- that we always make the correct guess, that we made the correct choice, whether it's i or j. Now, we don't actually know how to do that, so instead, we brute force. We try both of them, and then, because we're trying to maximize, we take the max-- just another way of thinking about it.

But overall, very straightforward-- the only added complication here is we had to deal with two sequences simultaneously, and we just took the product of those-- pretty easy. In general, if you have some constant number of sequences, you can afford to do this. You'll still get polynomial.

But of course, once you go to n sequences, you can't afford this. You would an n to the n behavior, so that's a limit to how far you could go. Two sequences is fine, three sequences is fine, but n sequences-- there probably is no polynomial time algorithm for this problem.

Cool-- I want to show you an example. I have an example here. I didn't want to try out hieroglyphology versus Michelangelo, so I came up with another example. Their habit is to say hi. So the longest common subsequence of there and habit is HI. And it's a giant-- well, not that giant-- it looks kind of like a grid graph.

The base cases are out here, because those correspond-- each of these nodes is a subproblem, and this corresponds to, what is the longest common subsequence between EIR and ABIT? And it should be I. It's the only letter they have in common, and that's why there's a 1 here to say that the longest common subsequence has a 1-- has size 1.

The base cases are when either their has been emptied or when habit has been emptied, so those all have 0's on the outside. And then the problem we care about is this one. It's their versus habit. Claim is the length is 2. And what I've drawn in here are what I'll call parent pointers, like we talked about with BFS, and shortest paths, and so on.

So we had this choice-- sometimes we had a choice-- on whether we recursed here or recursed here was the best thing to do. I'll draw in red the arrow from L i, j-- sorry-- to L i, j from one of these in this case. And in this case, there was no choice, but I'll still draw in red that arrow.

So these diagonal edges are exactly the cases where the letters match. Here H equals H. I equals I here, so I draw this diagonal edge. That's that first case, where the letters are equal, and so I recurse by increasing i and j. That's why I get a diagonal edge. There's also one over here, where T equals T.

So for those, we're getting a 1 plus. You see this 1 is 1 larger than the 0. This 2 is 1 larger than the 1. This 1 is one larger than the 0. And for every other vertex, we are rehearsing this way and this way. We see what those two numbers are, and we take the max. So this whole diagram is just filled in by-- for each position, where they're not equal.

We look at the guy below. We look at the guy to the right. Those are the slightly smaller substrings. We look at those values. We take the max. As long as you compute this in a generally right-to-left and bottom-up fashion, whenever you're trying to compute a guy, you will have the-- its predecessors already computed. That's the topological order of this graph.

And then, at the end, we get our answer, which is 2. And now, if we pay attention to where we came from-- for example, this vertex had to come from this direction-- 2 is the max of 2 and 1, so I highlight the 2 edge. And if I follow this path, there should be a unique path to some base case. We don't know which one.

And in this case, the diagonal edges correspond to my matching letters. So there's the H here, followed by the I here. And so HI is our longest common substring. In general, we just follow these pointers backward-- the red pointers-- and we get our answer. So not only do we compute the length of the LCS, but we actually can find the LCS using parent pointers.

And this is a concept you can use in most dynamic programs, including all the ones from today. OK, any questions about LCS? All right-- perfectly clear to everyone in the audience. Now we move onto longest increasing subsequence, which-- did I lose a page here? I just disordered them.

OK, this problem has almost the same name, but is quite different and behavior-- longest increasing subsequence-- LIS, instead of LCS-- both famous problems, examples of dynamic programming. Here we're just given one sequence, like carbohydrate. So this is a sequence of letters, and I want to find the longest subsequence of this sequence that is increasing-- strictly increasing, let's say-- so in this case, alphabetically.

I could include CR, for example, but not CB. That would be a descending subsequence. In this example, the right answer-- or a right answer is abort. There aren't very many English words that are increasing, but there are some, and I looked through all of them. As I just implemented this dynamic program we're about to write down, it took me like two minutes to write down the DP, and then more work to read the dictionary and look for cool examples.

So in general, we're given some sequence A, and we want to find the longest increasing subsequence of A-- the longest sequence that is increasing, strictly. We could use the same thing to solve not strictly increasing, but-- so here things are going to be a little trickier. It's easy, in that we just have a single sequence.

So again, we think, OK, let's look at our chart here. We could try prefixes, suffixes, or substrings. I personally prefer suffixes. Jason prefers prefixes. Whatever you prefer is fine, but always, generally, start there, because there's nothing in this problem that makes me think I need to delete things from both ends.

AUDIENCE: I have a question.

ERIK DEMAINE: Yeah-- question?

AUDIENCE: Isn't the answer to this problem always 26?

ERIK DEMAINE: Is the answer always, at most, 26? Yes, if you're dealing with English words-- so when I say sequence here, this is a sequence of arbitrary integers-- word size integers. So there you can have a ton of variety. This is just for the fun of examples, I've drawn this.

But even if the answer is 26, finding that longest common subsequence is-- the obvious algorithm would be to take all substrings of size 26, which is n to the 26. We're going to do much faster than that here. N squared time. And then, if you remove the strictly increasing, then it can be arbitrarily large.

OK, so let's try to do this. Maybe I won't be so pessimistic to write attempt here. Let's just go for it. So I want some subproblems, and I'm going to choose suffixes. So I'm going to define L of i to be the longest increasing subsequence of the suffix of A starting at i. That's the obvious thing to do.

And now I'm going to leave myself a little space, and then I'd like a relation on these. So I'd like to say what L of i is. And what do I have to work with? Well, I have live things larger than i. Those would be smaller suffixes. But let's go back to, what is a question that I could ask about this subproblem that might help me figure out what the longest increasing subsequence looks like?

So we're looking at a-- here's A from i on. Longest increasing subsequence is some subsequence. And we'd like to remove letter i. Now, when we do that, there are two choices. Maybe i is in the longest increasing subsequence, or it's not in. So the question I would like to answer is, is i in the longest increasing subsequence of A-- of A from i onwards?

This is a binary question. There are two options-- so again, just like before. And so I can brute force those two options. And then I want the longest one, so I'm going to take the max. So I'd like to take the max of something like Li plus 1. So in the case that I don't put i in the solution, that's fine. Then I just look at i plus 1 on, and recursively compute that, and that would be my answer.

And the other option is that I do put i in the longest increasing subsequence, so I do 1 plus the rest L i plus 1. If I close this brace, this would be a very strange recurrence, because this is always bigger than this one. There's something wrong here, and the something wrong is I haven't enforced increasing at all.

There's no constraints here. It's just saying, well, I'll i put in, and then I'll do whatever remains, and I'll pray that that's increasing-- probably won't be, because indeed, if i is a letter-- or is a number that is strictly greater than i plus 1, then this will be wrong. So I really can't always do this.

I could check whether i plus 1 is in the answer, but some-- but I don't. I can check whether the letter i is less than letter i plus 1. But maybe I put this in the longest increasing subsequence and then I put this in the longest increasing subsequence, and so I need to compare these two items. But I don't know when that's going to happen.

Things seem really hard. And indeed, there's no way, from this subproblem definition, to write down a relation. But there is a slight tweak to this definition that makes it work. So the trouble we have here-- and this is the idea of subproblem constraints or conditions-- the trouble we have is, when we recursively compute the longest increasing subsequence on the remainder, we don't know the first item in that answer.

Maybe it's i plus 1. Maybe it's some guy over here. If we knew who it was, then we could compare that item to item i. And so what we'd like to do is add a constraint to the subproblem that somehow lets us know where the longest increasing subsequence starts. So what I would like to say is long as increasing subsequence of that suffix that starts with A of i. So in other words, it includes A of i.

This was a separate question. OK, this is a bit of a funny constraint. It changes the problem. It's no longer what we want to solve. If you think about the original problem, before, it was solving L of 0. We just want the longest increasing subsequence of the whole thing. Now it's not necessarily L of 0. L of 0 means, what is the longest increasing sub sequence of the whole sequence A that includes the first letter of A?

And maybe we do include the first letter of A, maybe we don't we don't know where the longest increasing subsequence starts. Here, for example, it didn't. It started at the second letter. But conveniently, it's OK that we don't know, because that's just another question we could ask is, where do we start? Where might the LIS start? Could start at the first letter, second letter, third letter-- there's only n choices, and let's just take the max of all of them.

So before I get to the relation, let's solve this problem. I just want the max of L of i for all i. I guess we've been writing size of A here. OK, the maximum will be the overall longest increasing subsequence. So if I could find longest increasing sub sequence that includes the first letter, or the longest one that includes the second letter, and so on-- so it starts at the first letter, starts at the second letter-- then this max will be the longest overall.

This subproblem is not what I really wanted, but it's still good enough, because it lets me solve my original problem. And this is adding an extra constraint to my problem. And doing this is challenging. Thinking about what the right constraint that would let you solve your problem is tricky, especially in the beginning. But for now, let's just take this as a thing that works.

Why does it work? Because now I can say-- well, this term was fine, max-- so I'm trying to write longest increasing subsequence starting with the i-th letter, versus-- yeah, actually, no. This is just going to be different. OK, so now I get to the central issue, which is I know, by the definition of L of i, that I include letter i. This is going to be in my longest increasing subsequence. That's what I'm looking for, this definition.

But I don't know what the second letter is-- could be i plus 1, could be i plus 2, could be anything bigger. Whenever there's something I don't know, I'll just brute force it. I don't care that I don't know. I'll just take a max over all the possible choices. Let's say that the next letter included in the longest increasing subsequence i is j. Then I would like to look at L of j.

Now, I don't know what j is, but I'll just brute force all possible choices for j. So that's going to be i strictly less than j, because I don't want include the same letter i again. And otherwise, I would get an infinite recursive loop, if I put less than or equal to here. And maybe I don't do anything else to n.

OK, not quite done-- now-- this is the interesting part-- I can enforce increasing, because I can't just choose any letter j to the right of i. Also, the number-- or letter that's written in here has to be greater than the number that's written here. That's the strictly increasing property. So I can add as a constraint in this max to say A of i am strictly less than A of j. And if you wanted non-strictly increasing, you would add an equal there.

This is mathematical notation. In Python, you would say max open paren of this thing for j in this range, if this holds. So I'm just doing for loop, but I only do-- I only look at the possible choices for j when the strict increasing property holds. And then, when that holds, I put-- check this. Now, this set might be empty. I need to define what the max is when it's empty.

Oh, I also need a 1 plus, don't I? Let me just do 1 plus. So we're told that i is in the answer, so we always get 1. And then the remainder is this or 0. If there are no Aj's greater than Ai, then we'll default to 0 and say that i is the only item in my increasing subsequence.

OK, so there are a few details to get right, but in general, once you figure out what these recurrences look like, they're very simple. This is one line of code, and then all you need in addition to this is the original subproblem and some other things. We need the base cases, but I should do them in order.

Topological order is just the usual for loop, because I'm doing suffixes. It's going to be i equals length of A down to 0. Base case is going to be L of length of A, which is 0, because there's no letters in that suffix. And we already did O, and then time-- is a little different from the past examples.

So number of subproblems, just like usual, is linear length of A subproblems. It's only one sequence we're thinking about now, unlike the previous example. But the work we're doing in this relation now is non-trivial work. Before we were just guessing among two different choices. Now we're guessing among up to n different choices. This n here is length of A.

And so we have theta length of A, non-recursive work that we're doing in each subproblem. Or you might think of this as choices that we're considering. And for each choice, we're just spending-- I mean, we're just taking a max of those items, adding 1. So that's a constant overhead. And so we just get this product, which is A squared-- cool.

So that's longest increasing subsequence. Make sure I didn't miss anything else-- so we're using this idea of asking a question, and guessing or brute forcing the answer to that question in two places. One place is we're promising-- we're requesting that the longest increasing subsequence starts at i, so then the question is, well, what is the very second item that's in the longest increasing subsequence that starts with i?

We're calling that j, and we're brute forcing all the possible choices that j could be, which conveniently lets us check, confirm that it's actually an increasing subsequence locally from i to j. And then L of j will take care of the rest. By induction, the rest of the longest increase in subsequence starting at j will also be increasing. And so this guarantees, by induction, the whole thing will be increasing.

Then we also use this local brute force to solve the original problem. So we added this constraint of starting at i, but we didn't actually know overall where to start. But that's fine, because there's only A choices. So I should mention, in the running time analysis-- so they're solving the subproblems. That's fine, but then there's also a plus whatever it costs to solve the original problem. But that's OK. That's length of A.

All of this plus length of A is still length of A squared. But if you're doing exponential work here, that would be bad. We have to do some reasonable amount of work to solve the original problem in terms of all the subproblems.

I have an example hiding here. This is a little harder to stare at. Here I have empathy. And this example is not-- doesn't have much empathy, because the longest increasing subsequence of empathy is empty. Empty is one of the few English words that's increasing. And the hard part here is drawing the DAG. It's almost the complete graph, but we only draw edges from smaller letters to bigger letters.

So we draw from E to M, from E to P, from E not to A-- there's no edge from E to A-- from E to T, not from E to H, but yes from E to Y. And then we also draw from E to the base case, which is there's no more letters. That edge to the base case is-- corresponds to this 0, or I guess this n, where we say, oh, let's just recurse. Let's just throw away-- actually, maybe we don't need the union 0 there, in fact, because we include L of n, which is the empty substring. Then the definition of L is a little funny. What does it mean to say you start with A of n? Hm?

AUDIENCE: If we define A of n.

ERIK DEMAINE: Right, A of n is not defined, so that's not so nice. So maybe fix that. n equals case. OK, but I'm still going to draw an edge there-- conceptually say, oh, we're just done at that point. That's the base case, where we have no string left-- cool.

And when I said from to to actually, I meant the reverse. All the edges go from right to left. And then what we're doing is looking for the longest path in this DAG. Longest path is maybe a problem we've talked about in problem session, because it's a DAG-- well, longest path is the same thing as shortest path, if you just negate all the weights.

There are no weights in this picture, so if you just put negative 1 on all these edges and ask for the shortest path from the base to anywhere-- so single source shortest paths from this base-- then we would end up getting this path, which, if you look at it, is E-M-P-T-Y, empty. And so that shortest path is indeed the right answer.

What I've drawn here is the shortest path tree. So also, if you wanted the longest increasing subsequence starting at A, then it is A-T-Y, just by following the red arrows here. And how do you get that? You just draw the parent pointers, just like we did before. I didn't mention, but this example can also be solved with shortest paths. Once I construct this graph, you can do the shortest path from some base-- I don't know which one-- to here.

If you put negative 1 on all of the diagonal edges and you put weight 0 everywhere else, then that corresponds to-- the shortest path in that graph will correspond to the longest-- the path with the most diagonal edges. And that makes sense, because the diagonal is where we actually get letters in common. And so in this case, it's 2.

So both of these dynamic programs could instead-- instead of writing them as a recursive thing with memoization or writing them bottom-up as a for loop and then doing the computation, you could instead construct a graph and then run DAG shortest paths on it. But the point is these are the same thing. It's actually a lot simpler to just write the dynamic programming code, because it's just a for loop and then a recurrence. So you're just updating an array.

Because you already know what the topological order is, you don't have to write a generic depth for search algorithm, take the finishing order, reverse it, and then run this-- run DAG shortest paths with relaxation-- much simpler to just write down the recurrence, once you figured it out.

But they are the same in these examples. In Fibonacci, for example, you cannot write Fibonacci as a single source shortest paths problem, but a lot of DPs you can write as shortest paths problem-- just a connection to things we've seen.

All right, last example, last problem for today-- we'll do more next week. Alternating coin game-- this is a two player game. We're going to find the optimal strategy in this game. In general, you have a sequence of coins, and we have two players. They take turns.

So given coins of value v0 to v n minus 1-- so it's a sequence. They're given in order-- in some order-- for example, 5, 10, 100, 25-- not necessarily sorted order. And the rules of the game are we're going to take turns. I'm going to take turns with you. I'm going to use I and you to refer to the two players.

And so in each turn, either one-- whoever's turn it is, I get to-- we get to choose either the first coin or the last coin among the coins that remain. So at the beginning, I can choose 5 or 25. And I might think, oh, 25's really good. That's better than 5. I should choose that. But then, of course, you're going next, and you're going to choose 100, and you'll win the game.

You'll get more of the total value of the coins. So in this is example, a better strategy is to take the 5, because then the 100 is still in the middle. And so once I take 5, you get to choose 10 or 25. At this point, you'd probably prefer 25, because that's better than 10. But whichever you choose, I can take the 100. And so I get 105 points, and you're going to get 35 points.

OK-- good example for me. So that's easy for a simple example, but in general, there are exponentially many strategies here. At each step, either of us could go left or right-- choose is the leftmost or the rightmost. And we're going to give a dynamic programming algorithm that just solves this fast.

I didn't mention-- so this algorithm is quadratic time, but it can be made n log n time. It's a fun exercise. Using a lot of the data structure augmentation stuff we've done, you can make this n log n. This algorithm, I think, is going to be n squared time.

So I won't right the problem exactly, but I think you know the rules. Choose leftmost or rightmost coin, alternating moves. So I'd like to define some subproblems. And this is a problem that's very naturally a substring problem. If I just looked at suffixes, that would deal great with-- if I'm deleting coins from the left, but as soon as I delete-- and if I delete coins only from the right, that would give me prefixes.

But I'll tell you now, there's no dynamic programming where the answer is suffixes and prefixes. You can do suffixes or prefixes, but if you need both, you almost certainly need substring, because as soon as I delete the first coin, and then maybe you take the second coin-- that's exactly the optimal strategy here-- now you have an arbitrary substring in the middle. But substrings are enough, because we're only leading from the ends. We'll look at substrings.

So more precisely-- this is just the intuition-- we're going to define some generic x of i, j is going to be what is the maximum total value I can get from this game, if we play it on coins of value Vi to Vj. So that's a substring. So this is one way to write down the subproblems, and it's also a good way. You could write down a relation on this definition of subproblems.

But I'm low on time. There's two ways to solve this problem. This is a reasonable way, exploiting that the game is zero-sum. But I'd like to change just a little bit to give you, I think, what's a cleaner way to solve the problem, which is to add a third coordinate to my subproblems. So now it's parameterized by three things.

P here is-- only has two choices. It's me or you. And this gets at a point that's maybe not totally clear from this definition-- max total value that I can get from these-- this substring of coins. But this is not obviously what I need. So obviously, at the beginning, I want the whole string and I want to know what my maximum value is-- fine. And I go first in this game. I didn't specify, but I do. [INAUDIBLE].

But as soon as I do a move-- as soon as I take the first coin, for example-- it's now your turn. And so I don't really want to know the maximum total value that I would get if I go first. I'd like to say, if player P goes first. I'd really like to know what happens in the case where you go first.

So for some of the substrings, I want to know what happens when you go first, and for some of them, I want to know what happens when I go first, because as soon as I make a move, it's your turn. And so we're going to flip back and forth between P being me and P being you-- P-U. So you don't have to parameterize. There's a way to write the recurrence otherwise, but this is, I think, a lot more intuitive, because now we can do a very simple relation, which is as follows.

So I'm going to split into two cases. One is x of i, j, me and the other is x of i, j, you. So x of i, j, me-- so I have some substring from i to j. What could I do? I could take the first coin or I could take the second coin. Which should I do? That's my question. What is my first move? Should I take the first coin or the second coin?

So this is my question. What is the first move? There are exactly two possible answers to that question, so we can afford to just brute force them and take the max. If we're moving, we want the maximum number of points we can get-- maximum total value of the two choices. So if I take from the i side, the left side, that would be x sub i plus 1, j. Sorry. And now, crucially, we flip players, because then it's your turn.

And if I take from the j side, that will make it j minus 1. This is what I accidentally wrote at the beginning of lecture. Also flip players. So either I shrink on the i side or I shrink on the j side. Oh, I should add on here the value of the coin that I get, and add on the value the coin that I took.

This is an expression inside the max. That sum. And if I take the max those two options, that will give-- that is my local brute force the best choice of how many-- what are the total value of coins I will get out of the remainder, given that you start, plus this coin that I took right now in the first step, and for the two possible choices of what that coin is?

OK, what remains is, how do we define this x of i, j, you. This is a little bit funnier, but it's conceptually similar. I'm going to write basically the same thing here, but with me, instead of you-- because again, it flips. This is, if you go first, then the very next move will be me. So this is just the symmetric formula here.

I can even put the braces in-- so far, the same. Now, I don't put in the plus Vi and I don't put in the plus Vj here, because if you're moving, I don't get those points. So there's an asymmetry in this definition. You could define it in different ways, but this is the maximum total value that I would get if you start.

So in your first move, you get some points, but I don't get any points out of that. So there's no plus Vi. There's no plus Vj. It's just you either choose the i-th coin or you choose the j-th coin, and then the coins that remain for me shrink accordingly.

Now, you're kind of a pain in the ass. You're an adversary you're trying to minimize my score potentially because you're trying to maximize your score. This is a 0 sum game. So anything that you get I don't get. If you want to maximize your score, you're trying to minimize my score. These are symmetric things.

And so if you think for a while, the right thing to put here is min. From our perspective, we're imagining what is the worst case that could happen, no matter what you do. And we don't have control over what you do, and so we'd really like to see, what score would I get if you chose the i-th coin? What score do you get if you chose the j-th coin?

And then what we get is going to be the worst of those two possibilities. So when we get to choose, we're maximizing. And this is a general two player phenomenon that, when you choose, we end up minimizing, because that's the saddest thing that could happen to us.

OK, this is one way to write a recurrence relation. We have, of course, all of SRTBOT to do, so the topological order here is in increasing length of substance. So the T is increasing j minus i. Start with empty strings. So base case is that x of i, i, me is Vi. So here I'm inclusive in both ends in this definition.

So there is a coin I can take at the end. But if you move last and there's one coin left, then I don't get it, so it's 0. Then we have the original problem that is x i, j, me-- sorry-- x 0, n. That's the entire coin set, starting with me. That was the problem I wanted to do.

And then the running time we get is the number of subproblems-- that's theta n squared, because we're doing substrings-- times the amount of non-recursive work I do here. That's just a max of two numbers. Very simple. Constant time. So this is quadratic.

Let me show you an example. This is hard to draw, but what I've described here is called solution 2 in the notes. So here's our sequence-- 5, 10, 100, 25-- in both directions. And what we're interested in is all substrings. So over here I've written the choice for i. So we start at one of these, and if you start here, you can't end earlier than there. So that's why we're in the upper diagonal of this matrix.

And then there's two versions of each problem-- the white version and the blue version just down and to the right of it. If you can't see what blue is, this is the version where you start. This is the version where I start. And I've labeled here all of the different numbers. Please admire, because this took a long time to draw.

But in particular, we have 105 here, meaning that the maximum points I can get 105. And that's the case because, if we look over there, it is the max of these two incoming values plus the Vi that I get. So either I go to the left and I take that item or I go down and I take that item.

So the option here is I went to the left and took-- well, that's going to be tricky to do in time. The claim is that the best answer here is to go here with the 100 and take the 5, because going down corresponds to removing the last item. If I went to the left, that corresponds to-- sorry-- the first item. If I went to the left, that corresponds to removing the last item, so my options are 10 plus 25, which is 35, versus 100 plus 5.

105 wins, so that's why there's a red edge here showing that was my better choice. And in general, if you follow these parent pointers back, it gives you the optimal strategy in what you should do. First, you should take the 5 is what this is saying, because we just clipped off the 5. We used to start here, and now we start here in this subinterval.

Then our opponents, to be annoying, will take the 25-- doesn't actually matter, I think. Then we will take the 100, and then they take the 10, and it's game over. OK, all the numbers here are how many points we get-- doesn't say how many points the opponent gets. Of course, you could add that as well. It's just the total sum minus what we get.

Now, let me come back to high level here. What we're really doing is subproblem expansion, and this is an idea that we will expand on next lecture. And the idea is that sometimes you start with the obvious subproblems of prefixes, suffixes, or substrings. Here, the obvious version was substrings, because we were moving from both ends. If you don't know, probably suffixes or prefixes are enough.

So we start there, but sometimes that's still not enough subproblems. Here, as soon as we made a move, our problem almost turned upside-down, because now it's your turn, instead of my turn. And that was just annoying to deal with, and so we could-- whenever you run into a new type of problem, just build more subproblems. As long as it stays polynomial number, we'll get polynomial time.

And so here we doubled the number of subproblems to just the me case and the you case, and that made this recurrence really easy to write. In the notes, you'll see a messier way to write it, if you don't do that. In the examples we'll see next lecture, we're going to do a lot more expansion, maybe multiplying the number of problems by n or n squared.

And this will give us-- it will let us add more constraints to our subproblems, like we did in longest increasing subsequence. We added this constraint that we start with a particular item. The more subproblems we have, we can consider more constraints, because we'll just brute force all the possible constraints that could apply. Well, we'll see more of that next time. That's it for today.