[SQUEAKING]

[RUSTLING]

[CLICKING]

JASON KU: Hey, everybody. Welcome back. This is our last quiz review for the term, quiz 3, we'll be talking about, which will be the last quiz until the final. It's on dynamic programming, which you guys have been studying in lectures, and recitations, and on your problem sets 7 and 8, and lectures 15 through 18, so four lectures.

Quiz 1 and quiz 2 material on, essentially, data structures and graph algorithms aren't going to be explicitly test-- or we're not trying to explicitly test it, that material, on quiz 3. But it is fair game. The material's cumulative. And so if you have to store some stuff in the data structure, that's fair game. But we're not specifically trying to test you on that material.

OK, and really, we haven't learned all that much new material in these last four lectures in this unit. So this is scope. We've got, we're mostly handling dynamic programming on these four lectures and recitations and these two problem sets. But really, the focus is going to be on this recursive framework of solving problems with a focus on dynamic programming, specifically.

Now, the recursive framework we have, I think, in previous slides, we used this SRTBOT notation. And I think there might be a space there in previous versions. I'm concatenating them together here. But really, it's a framework for solving breaking down your problem into a set of subproblems that can then be related recursively.

And if that relationship depends on problems in a decreasing or in a smaller sense, there's a directionality to which subproblems I'm reducing to each time I make a recursive call. And that dependency graph is a cyclic. Then we can solve via dynamic programming by memoizing from bottom up or by calling things and remembering the calls that we've called before via memoization. And the basic idea here is this the recursive framework SRT BOT that we established is good for any recursive algorithm.

But in the special case where subproblems may be used more than once, may be used when computing other subproblems, then we get this really nice speed up by recognizing that we don't have to do that work more than once. And essentially, instead of looking at it as a tree of recursive calls that may call the same problems more than once, we look at it by collapsing those nodes of the same value down into one. We get a DAG. And dynamic programming is when those subproblems overlap.

OK, so let's take a look at our recursive framework here, SRT BOT. We have the S-R-T B-O-T. I remembered the space this time. Subproblem, you're going to define some subproblems. You're going to relate them recursively. You're going to specify a topological order of those subproblems that the relation satisfies.

You're going to list some base cases, basically wherever you could solve this problem in constant time without doing any recursive work. Stating how you solve the original problem, which might involve combining many subproblems, but frequently, is just finding one subproblem, and possibly remembering-- storing parent pointers to return an optimal sequence, or something like that. And then, analyze the time.

Now, the last one isn't really important for solving a problem recursively. But in this class, it's really important, because we want to tell whether the algorithms that we make are efficient. So let's dive a little deeper into each one of these things.

So when we approach a subproblem, really, what I'm asking you for is to describe-- basically, set up a set of problems. Basically, I like to use the variable x. But you can use whatever variable you want.

But basically, you're telling us what's in your memo and how big your memo is. So we usually have x as a function of some variables. And you're wanting to describe to me what the meaning of that subproblem is in terms of the parameters. Now, if you have parameters in your subproblem that don't appear in your subproblem definition, you're doing it wrong. And you're probably not going to get points for the problem.

Because I don't know what your problem means now. Even if it's a correct problem and you do the rest of it right, part of this class is about communication. And if you're not communicating to us what this thing is doing, it's really difficult for us-- for you to convince us that your algorithm is correct. So you really want to, in words, describe what the output of your subproblem is. What will the memo return to me? And how those return values depend on the inputs, the parameters of your subproblem.

So that's what, in words, describe what a subproblem means. So that's going to be a really important thing for you not to forget on a quiz. Then, when making subproblems, often, what we're doing is we're rehearsing on different values of indices in a sequence or numbers in your problem. That's kind of what we got to in the last-- in lecture 18, I guess, when we were talking about expanding subproblems based on an integer in a problem.

Now actually, an integer in our problem is the number of things in a sequence. And so, really, those indices are integers in our problem that we're looping over. Except those integers happen to be the size of our subproblem. Whereas, other integers might be larger, which is why you might get a pseudopolynomial time bound.

But in general, when I have a sequence of things that I might want to dynamic program over, common choices for subproblems are prefixes or suffixes if I can kind of locally figure out to do with one what to do with one item and then recurse. Or if I can't kind of localize it by one choice on one side, if I have to make a choice in the middle, or I have to make a choice on both ends, then you might want to use sub-- basically contiguous subsequences of your sequence. Because you might need that flexibility when reversing downward, if you need to take something from both the front and the back, for example.

And really, what's the difference between prefixes and suffixes? Not much. OK, we've been concentrating on suffixes in this class. Because in some sense, it's easier to think about. What am I doing with the first thing in my sequence, or my suffix? And then I can recurse on what happens later.

Now, in actuality, when you're doing this, say, bottom up, the actual computation that is evaluated first is where in that sequence? I may be calling, at the top level, what happens to my first element. But I'll actually deal with that first element last. Because I will recursively solve everything below me, in front of me, before I figure out what to do with this thing.

So in actuality, when I'm solving my recursion, I will start at the end, bottom up, because that's my base case. And then I'll work my way back to the front. Whereas, with prefixes, you look at it the other way.

What am I doing with my last element? If I look at what I'm doing at the last element, I recurse on a prefix, on the stuff that's before me. And then when I do bottom up, I start from the front and work my way up. it's two different sides of the same coin. And usually, these are interchangeable.

We've been doing it suffix-wise, because when starting to learn dynamic programming, it's a lot-- we read things from left to right and things like that. It's a lot easier to figure out what's happening with the first thing and move forward, conceptually. It's actually exactly the same thing. I could just flip my sequence, do the exact same thing with prefixes. It would be the exact same dynamic program.

So these things are interchangeable. It's really useful, when learning to dynamic program, to be able to switch back and forth between these things. We'll be working on suffixes today on the problems that we do. But these are interchangeable. And sometimes it's useful to be able to conceptually think about it in both directions.

So aside from dealing with subsequences of sequences, in particular, contiguous ones, we also often multiply our subsets across multiple inputs. Like if we have multiple sequences, we might take indices in each one of them to represent prefixes or suffixes. And then we might have to remember additional information by maintaining some auxiliary information, like am I trying to maximize or minimize my sum in a-- or evaluated expression in an arithmetic parenthization.

Or is it player 1's turn or player's 2 turn? Or which finger-- where was my finger when I was playing piano or something like that? Those are the kinds of things that we might expand our state on. And in particular, we might expand our state based on the numbers in our problem if we're trying to, for example, keep track of how much space is left in a knapsack or something like that.

But in general, if I'm trying to, say, pack a set of things, it's useful to know how much space I have left to pack. So that's subproblems. This is really the key part about dynamic programming is the recursive part. This is what makes it hard is choosing a set of subproblems.

And it's often you build subproblems to fit well with relations. So usually, building what these subproblems are is usually closely coupled with the next step, which is relating the subproblems recursively. And relate recursively, I-- usually what I want is an expression, a mathematical expression, relating the definition of a subproblem you had in the previous section, relating those, in math terms, to the other things.

This is-- it's really important that you write this in math, because it needs to be precise, to communicate this thing well. Now, you can write it in words. But I would suggest you write it as a mathematical expression, because it's a lot more concise for us to see what's happening in your recursion.

So relate them recursively. Basically, I'm going to write, say, that x of some set of parameters equals some function, usually a maximization, or a minimization, or a summation, or and or, or an and, or some other combinator of a bunch of choices that you might make, so-- or a bunch of subproblems that you might recurse on. Basically, you're going to depend on some other subproblems that are smaller in some sense.

Now, actually embedded in this, the idea of a smaller subproblem isn't really well defined yet. We haven't told you an ordering of these subproblems to be smaller. But that's what's going to come in the third step. So kind of a strategy for figuring out what these recursive relations might be is to identify some question about the subproblem solution.

What do I do with the first character in this string? Or which cage do I put this tiger in? To figure out what subproblem should I recurse on later? I don't know the answer to that question.

But if I knew the answer to that question, then I could recurse on a smaller subproblem, because I figured out what to do with that tiger. And so it will let me reduce to smaller subproblems. And then, what dynamic programming does is because I only have a polynomial number of subproblems, and I assumed I've already computed what those are, I've already memoized what the solutions to those problems are, then I can just locally brute force over all the possible answers to that question. And that's one way to look at dynamic programming.

OK, so then as we were talking about topological order, arguing that relation is acyclic. Essentially, just defining what smaller means when we say we're recursing on smaller subproblems. What does smaller mean? Usually, you're saying that some index or some parameter of my subproblem always decreases or increases. Sometimes, that's not always the case. Sometimes, you have to, maybe, add a couple indices and see that that always increases, because one may stay the same while the other increases or something like that.

But in general, as long as you argue that the relations are acyclic, then the subproblem graph is a DAG. And you can compute in a bottom-up manner. And you don't get infinite loops in your recursion.

OK, the last thing, the last couple of things are kind of bookkeeping. But if you don't write these on your exam, we can't give you points for them. So write these down. Base cases, if you don't tell us base cases, then your algorithm cannot be polynomial time. It can't even be finite time, because your algorithm never stops.

It just continues to recurse forever and ever and ever. And so it's hard to give us points-- I mean, we will give you some points if your subproblems and relation are correct. But really, if you write code without a base case, it's going to be wrong. So base cases are really important.

Basically, for anything at the bounds of your computation, wherever your recursive relation would essentially go outside the bounds of your memo, let's say I'm dealing with a subsequence. And at some point, I'm trying to point to a state where I have zero or negative elements in my sequence, that's probably a bad thing.

And so I want to define how to compute those things in constant time. So that my algorithm can terminate when it gets to one of those base cases. So it's really important that you cover all of those possible leaf locations where you want to be able to return in constant time. And we'll do some of that today.

State solutions for all reachable, independent subproblems where the relation breaks down. Essentially, I would be going outside the bounds of my thing. Or anything where, maybe if you've got one item left, you might say, well, I have no choice on what to do with that item. I have to pick it or something like that.

OK, then for your original problem, you show how to compute solution to the original problem from the solutions of your subproblems. So usually, this is just here's a subproblem. It's the one that used all of the things, and that's going to be my answer.

But that's not always the case. Sometimes, like in a longest increasing subsequence, we had to take a max over all of our problems that we computed, or max subarray sum, we also had to do that. But in general, the output to our subproblems wants to be some scalar value that we're trying to optimize, or a Boolean, or something like that.

It's how we maximize or minimize what we're doing. We're not storing the entire sequence of how we got there. Because there could be an exponential number of possible subsequences that got there. That's the whole point of dynamic programming. We're kind of isolating the complexity of one subproblem down to a single number.

But in a lot of problems, we might want to reconstruct, say, the placement of tigers into cages and not just how-- what's the minimum discomfort over all tigers or something like that, like you had in your problem set. So I actually want to know where to put tigers into cages. And to do that, every time I maximized a subproblem, I can remember which subproblem or subproblems I depended on, just like storing parent pointers in shortest paths. And then, using those parent pointers, I can just walk back in my subproblem graph and figure out which path to a base case led me to an optimal solution.

And then the last thing is analyzing running time. Generally, you're just summing the work done by each subproblem. Because the assumption is you're calculating all of the subproblems you described to me. But if the work per subproblem is bounded by the same value, you can just multiply it out. So that's generally a weaker bound, but usually asymptotically equivalent to the stronger notion on the left.

And that's basically how you do running time. Usually, it's enough to-- how do I determine how many subproblems I have? Well, I look at the possible values of each of my parameters. And then I multiply those numbers together. A lot of people will maybe say, oh, I add them together. No, because I'm able to choose each of these independently. And so I multiply those things together.

And then the work done by each subproblem is usually the size of the thing I maximizing over, or minimizing over, or summing in my relation. It's going to be the size of that, the branching that I have, the number of subproblems I depend on. And so the number of subproblems, you probably look at your subproblem statement definition. To find the work done by each subproblem, you look at your recursive relation.

OK, so with that, we've got this really nice framework. And we're going to use it to solve some practice problems, happy days. And these are a little bit longer in terms of description than our previous quiz 2 review. So I'm going to go ahead and read them out for you. This one's a little shorter.

Tiffany Bannen stumbles upon a lottery chart dropped by a time traveler from the future, which lists winning lottery numbers and positive integer cash payouts for the next n days. Anyone get the reference here, Tiffany Bannen? Biff Tannen from some *Back to the Future* thing. So this was actually-- I think it's the second to *Back to the Future* movie where this happens.

Anyway, Tiffany wants to use this information to make money, because she knows the future about the lottery. But is worried that if she plays winning numbers every day, lottery organizers will get suspicious and shut her down. So the idea here is maybe it's still suspicious, but decides to play the lottery infrequently, at most, twice in any seven day period. She'll win, but it's infrequent enough that maybe that's by chance, maybe not.

Describe a linear-time algorithm to determine the maximum amount of lottery winnings Tiff, Tiffany, can win in the next 10 days by playing the lottery infrequently. Now, this was a particularly difficult type of p-set, or first p-set on dynamic programming problem. But let's try to do it together. So this is problem 1. I'm going to just call it Lotto.

OK, so how can we deal with subproblems here? Well, I might want to think about what do I do on the first day. Am I going to play the lottery or not? And then recurse on the rest. That sounds good, right?

I might have something like-- well, let's say that L of i is the winnings on day i. This is kind of just like-- this doesn't define use notation on what the cash payouts are. And so I'm making a variable to do that.

And for the sake of what's written down on my sheet, I'm going to assume that this is 1 index, I don't know why. So I have days 1 to n. I know what their lottery payouts are. So I might, when I'm doing my SRT BOT stuff, I have subproblems. What I might want to do is see what happens on day one and recurse on what's later.

So I might have something like x of i is max winnings for, I guess, possible for days i to n. Anyone have a problem with this type of subproblem? Let's kind of see what this type of subproblem would lead me to. I can either, in my relate step, what are my choices? I can either play on day i or I cannot play on day i. If I'm trying to maximize this thing, maximize.

Either if I play on day i I get Li. And then I can recurse on the remainder. Or I don't plan Li and I recurse on the remainder. Anyone like this recurrence? Why don't we like this recurrence?

I'm just always going to pick this thing. These things are always positive. I think it's always positive integer payouts, yeah. And so I'm always going to pick Li and the problem here is this is not obeying or dealing with this condition that I have, which is I'm only allowed to play twice a week, or not quite twice a week. It's not a fixed week-long period. It's within any consecutive seven-day period, which is a little confusing.

How can I remember what days I'm allowed to pick later on? It seems a little daunting. In a sense, for me to know-- this is max winning possible for days i to n. But in some sense, it depends on which days I picked before. Because if I picked i minus 1, I can't pick another day for another six days.

If I have-- right, so I have-- let's do this precisely. This is i minus 1. I have seven day period, 1, 2, 3, 4, 5, 6, 7. If I played the lottery here and I played the lottery here, then I'm not allowed to play the lottery here, I'm not allowed to play here, not allowed to play here, not allowed to play here, not allowed to play here. I am allowed to play here. So this is i plus 1, plus 2, plus 3, plus 4, plus 5, plus 6, i plus 6.

So depending on what happened before me, I might not be able to play until day i plus 6. But if I haven't played until-- since way back here, I could potentially play the next guy. I don't actually know which of these I can play on next. In some sense, I need to remember which days I'm allowed to play on next. And I want to be able to-- at the beginning, I have no restrictions.

I can just play on this guy next, even if I played there. But in general, I will be restricted in some way between being able to play on this guy and being able to play on this guy. And so what I'm going to do is I'm going to generalize my problems by storing additional information so that I can rely on that information when I look into the future and recurse.

So instead of-- I'm going to, in some sense, need to remember two days. Where was the last two places I played? I'm going to simplify that a little bit by saying that this subproblem, max winnings possible for days i to n, I'm going to say that I have to play on day i, similar restriction as longest increasing subsequence, I'm definitely including this in my subsequence.

That's just going to make it easier for me to be like, oh, I definitely know I played on this day. It's going to be-- it would make it easier for my thinking. So assuming play on day i, and actually, I need to remember what's the next day I can play. So I'm going to expand this subproblem by another-- I guess this is a j.

OK, assuming I play on day i, and I'm allowed to play, I guess, and next allowable play on day i plus j for, what are my possible range of days for j? I can either play the next day, but I'm never restricted past day i plus 6.

Because the things before me [AUDIO OUT] further to the right, because I haven't dealt with them yet. So I only have to deal with this from 1 to 6. Well, this is nice, because it expanded my subproblems by a constant number. So I actually didn't lose anything asymptotically here by remembering this information. But I'm kind of-- I'm able to remember all the things that could have happened before me. I compress it into this one number.

OK so now let's rewrite our relation. I'm actually going to go ahead and use some more board space, because I think that's easier than erasing. All right, so we are looking at my relation. This is a pretty complicated relation. But what's happening now?

Now, I'm assuming I play on day i. That actually simplifies things a little bit. Because no matter what, I get the winnings on day i. Now, when I call this subproblem, I better make sure that it's OK that I played on day i. But that's in my caller. I am locally allowed to play on day i. I am playing on day i. That's the definition of my subproblem.

x of i is I'm going to maximize over some choice. Max of-- I guess I have Li no matter what. So this could actually come out of my max. And then I'm going to choose-- what am I choosing here? I'm not choosing whether I'm playing on day i. I'm choosing what my next day i play is, so that then I can recurse on that subproblem.

So what day can I play on? Well, I'm kind of restricted by this j parameter that I didn't add into my subproblem on what possible days I can play next. I'm going to split that into-- kind of compress that into one thing. So x, I can play on-- I have a choice of the next day I play, somewhere between j, which is my next allowable play, and sometime in the future.

So i plus k, this is going to be my loop that I'm looping over in terms of my max. And then, what am I restricted on in my play? It depends on how far I am from i. So if I'm here, this is i. If I choose k to be the next day, I can't play for many, many times.

So the subproblem I'm going to recurse on is-- this is i plus k. I'm going to recurse on i plus k. But I'm not able to-- j needs to be the max it can be. Because I can't play until i plus 6. So this i plus k plus 6 is going to be my thing. So let's see.

So I'm going to put-- let's see if I can unpack what I wrote down here. Max of 1, 7 minus k. Yuck. Oh, I'll put it here. 1, 7 minus k. OK, if I pick k, I'm not aggressive toward these boards as Justin is.

So if I pick a k way down here, I'm not restricted at all. And so the most permissive option I have here is 1. So I definitely can't be worse than 1. But if I pick-- and then this needs to be some number between 6 and 1. And so I can just check the other bound.

If this is the most productive, then this should be 6. So when k equals 1, this should be 6. And it decreases every time further back-- or further forward I choose this k. So that's what my subproblem is going to be. And I'm choosing over k in-- from j, sorry i plus j, sorry, j, thank you, until what? That's the question, until what?

Do I have to loop over n? If I loop over n, I'm going to get a quadratic running time, which is worse than what I'm allowed to do. The assumption is that I only have to check the constant number of these. And why might that be? Any ideas?

Let's say I am-- got my subproblem, I'm recursing. I've got i. Yeah, I don't know where j is. j is somewhere over here, One, two, three, maybe it's four, or something like that. Let's say I pick some k down here.

What is this? This is i plus-- so this is j is 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 15, way over here, two weeks later. Is it ever optimal for me to do that? Why not?

AUDIENCE: You could play the lotto in the middle and it wouldn't effect it.

JASON KU: Yeah, I could play the lotto in the middle here, right? Within-- from here to there, that's a seven day period where I only played once. And from here to here, that's a seven day period where I only played once. So it's going to be-- these are positive values. So it's going to be more optimal for me to choose something in here to play.

So how far-- I mean, I could just use 15 and that would satisfy. Because I've already argued to you that it's never optimal to. I can check this. It's not going to be optimal. It's going to be more optimal to play some time over here. But how far do I have to check?

Well, maybe I have to check up to 7. Does that work? Not quite. So let's say I played here, and I played here, and I played here, and I played here, I actually can't play here, here, here, here, here. I'm not allowed to play those. I guess these should be O's.

I played there. And I'm not allowed to play here, 1, 2, 3, 4, 5. But I am allowed to play anywhere in here. So I basically want to shrink this until these X's collide with each other. Because then it's possible that an optimal solution would require me to pick these two and then require me to pick these two way over there.

So this is 10 things in the middle. I only have to go up to, at most, 11. It's 11. Now, you can move any constant above 11 and get the same running time bound. But that's my analysis. OK, so we have our recursive relation.

And so what am I doing? I'm just looping over my choices of next day to play. I'm rehearsing on this thing where I actually do play on that day. But I'm remembering the information about what I'm allowed to play next by limiting, based on what my previous value was.

So that's the kind of key thing. I'm remembering something further in advance-- or I'm remembering what happened in the past by describing it as a restriction of something in the future. So this was a pretty difficult problem. I think it was one of our first dynamic programming problems on that term. It was probably a little ambitious.

AUDIENCE: Are you saying this recurrence goes up to 11?

JASON KU: Yes, the recurrence goes up to 11, not 10, 11. So we have our topological sort. What's a topological thought for these subproblems? Anybody? k is always going to be a positive number. Because j goes from 1 to 6. So I'm always going to be increasing in this first quantity. So for x of i, j, i always-- sorry, depends on strictly larger i.

So this i, when I call subproblems, it always calls subproblems with a larger i. Now, this is a little weird, because I wanted me to depend on smaller subproblems, smaller. Now, it is smaller, because I'm taking a smaller suffix. But its corresponding to using a larger number.

To me, that's a little confusing, but that's OK. Because we're kind of using things that are always monotonically going in some direction. So this is corresponding to a smaller subproblem in some measure. It's the number of elements that we're actually recursing on. In some sense, if we wrote this as a prefix, we would have us depending on strictly smaller i. And that would be more natural in terms of recursing on smaller subproblems. But I digress.

All right, then we have our original subproblem. I'm going to-- I can't move this board. I'll just keep going, because we have lots of boards. OK, our original subproblem, now, what could I do? I have to start somewhere.

Here, my subproblem's assuming that I'm starting at i. But I don't know where I start. I could start by taking the first element, but I might not. So I could just take the max over all i, over all i of x what? The first one, I'm not restricted on what I choose next. So what's the most permissive version of j?

One, I'm allowed to take the next j. So if I just take the max over all of these subproblems, I'll get the solution. Now, actually, this is a little bit more work than I need. This is looping over all n. It's definitely correct, because I have to start somewhere.

But will I ever start after the first, I don't know, 7? No, so I could just take this max over the first some constant number. And that would be fine. But that's OK. This is still smaller than the number of subproblems that we have.

AUDIENCE: If I were being lazy during my exam and I looped over j, would that be correct?

JASON KU: If I looped over j for?

AUDIENCE: [INAUDIBLE] over every possible thing.

JASON KU: Took the loop over this and j? Yeah, that would still be fine. Why not? It's just less-- it's more restrictive of subproblems. It will never be better to do that. But you could do that, because it wouldn't change your running time.

AUDIENCE: Could it change my running time if my j accidentally looped too far?

JASON KU: Well, j is restricted to be 1 to 6. So I'm not-- I don't think so. But in a different problem, in a different context, it could. OK, so that's the original. And in time here, what do we got?

We have a linear number of subproblems, number of subproblems. We've got-- I actually like, usually, saying exactly how many subproblems I have. Oh, we didn't do base case. I missed BOT, I missed my B. We'll do the original first. And then the base case.

OK, base case, what do we have as a base case here? Well, when I don't have anything to do. If I'm-- and actually, if I have this situation for i equals, say, n, I got my last thing. I could potentially start looping over sbuproblems that are negative in terms of my index. I'm not going to want to do that.

There's a couple of ways I can deal with that. I could set a value for all of my problems for negative i. That's one thing I could do. But then I have to kind of remember, or I have to figure out how far I go into the negative. That's one thing I could do. And I give a base case for each of those cases.

I don't have anything. So I get a-- I don't know, zero value for playing in the future, because I have negative things. I can't do anything with that. Another way of handling that, which I think I did in my solutions, was restrict that k to only be-- I guess and restrict that i plus k is less than or equal to n.

And then, I'll never go to negative problems. I'll never recurse on these things. But that means that when I call this on n, When I only have one lottery day left, this set will be empty. So what's the max over that thing? Max over an empty set? I don't know.

I mean, I could add on 0 here, that's one way I could do it. Or I could just say when I'm at n, and that thing is empty, or whenever it's empty, we can say the base case x i, j, I guess we could put this at n equals 0, or sorry, equals L of i, L of n, thank you.

Because at the last guy, I have to use Ln. So there it is. Now, in actuality, if you write this correctly, I put the Li outside, and I union this with the 0, I can actually get away with just having the relation and no base case. Because my relation actually reduces to a base case, because of the way that I wrote my relation.

But in general, you'll want to write some kind of base case here to either acknowledge that your relation handles it or be specific about what happens when I can't do any more work. And the last thing, time, we've got n subproblems exactly times constant work per subproblem. Because I'm looping over 11 possible values, actually, it's up to 11, because j could be 6. So this is order n work total.

So this is a pretty daunting first problem. But in terms of what Erik, Professor Demaine, was talking about last lecture, in terms of categorization of subproblem, or categorization of dynamic programs, what do we got? We've got a suffix subproblem where we expanded by some local information, remembering when the next time I can play. So that's kind of a categorization of these subproblems.

The recurrence relation has constant branching, but more than two branching. And I'm combining a bunch of subproblems in my original evaluation. And if I wanted to figure out what days Tiff should play on the lottery, you can store parent pointers when I'm evaluating this max.

I figure out which subproblem x I recurse-- that gave me the max. And I can walk back to see which choices I made to figure out which days I played the lottery. Does that make sense? So any questions on problem 1?

That's the most-- I wanted to have the most complicated one first. So that we could have an easier way to go. In a sense, this is the most complicated version of this kind of actually pretty simple dynamic programming setup. Why do I say simple dynamic programming setup?

It's just suffixes. And I'm just doing a constant amount of work local to me. It's just a very complicated local setup. But that's what I mean by simple. When we're designing subproblems, this is one that we could-- I mean, when we're designing problems for this dynamic programming setup, it's one of the hardest from a-- it's one of the easiest from a conceptual standpoint, but one of the hardest to actually implement.

OK, so problem 2, this one's a long one. A wealthy family, Alice, Bob, and their young son Charlie are sailing around the world. When they encounter a massive storm, Charlie is thrown overboard, presumed drowned. This is very colorful language for these problem set writers. 20 years later, a man comes to Alice and Bob, claiming to be Charlie, having maybe been marooned on an island for that long.

Alice and Bob are excited but skeptical. And they order a matching tests from the genetic testing company 46 and Thee. Given Alice and Bob-- sorry, given three length n DNA sequences, basically strings of CGTA, or something like that, from each of Alice, Bob, and Charlie, the testing center will determine three-- their ancestry as follows.

If Charlie's can be partitioned into two, not necessarily contiguous, subsequences of equal length, so basically I can take-- if I have n is length 5 or length 6, it's better be even. I need to find three characters in order. And then the other three characters must match to make some substrings-- some subsequences in Alice and Bob's DNA.

So that's a little hard to parse. So let's look at an example here. For Example. Alice's is AATT. Bob's DNA is CCGG. If Charlie's were CATG, they'd be matched, because CG is a subsequence of Charlie's DNA, and is a subsequence of Bob's DNA.

And AT is a subsequence of Charlie's DNA. And Is also a subsequence of Alice's DNA. And so we've partitioned them into two equal length subsequences. These are not necessarily consecutive subsequences, but just any subsequences, such that they appear in Alice and Bob.

But if Charlie would be found to be an impostor, if his sequence were AGTC, essentially, it's easy to realize that, because G and C are swapped in terms of their ordering. And GC, the letters GC only appear in Bob's DNA and don't appear in that order. So it's easy to see that he's an imposter with these strings. But you can imagine, with longer strings, this could be difficult to solve.

So we want an n to the 4th time algorithm to determine whether Charlie is a fraud. OK, so I actually shortened this last night. This was twice as long on the problem set. So yeah, anyway, so how do we approach this problem?

AUDIENCE: [INAUDIBLE].

JASON KU: No, they don't have to be contiguous. Like in the example, it would be matched if C and G is a subsequence, not contiguous, of CATG. Yeah, so that's an important part of this problem. I'm just-- I'm not trying to figure out if there's-- basically, there are only two contiguous subsequences of length 2n that this thing can be partitioned in. I just look in the middle.

No, we're looking for subsequences, not substrings. So they kind of interleave like this in some way. And there's actually a number of different ways I can partition that. There's actually an exponential number of ways. So that's a problem, potentially. Yes.

AUDIENCE: Is there a biological basis?

JASON KU: No, there's no biological basis to this thing that I know of. OK, all right, so how do we solve this problem? What problem does this look like? I mean, it seems like string matching.

So I might want to think it's something like longest common subsequence. But here, I have three sequences instead of two sequences. And we've got this other weird condition where we kind of need an exact partition of Charlie. I need to use all of the letters in Charlie, but I don't have to use all of the letters in Alice and Bob.

So let's get some notation here A, B, and C are n length strings. So what could I do? Let's define some subproblems. If I were to go via longest common subsequence, I might keep track of an index of a suffix or prefix of each one of these strings. That kind of makes sense.

Something like i, j, k, where we're talking about the suffixes-- sorry, that's prefixes, i, B, j, and C, k. That seems reasonable, at least. It's what we would do for longest common subsequence. What's the problem here?

I mean, I could match this guy with one of these guys, or decide to skip it, and match one of these guys, and decide to skip it. But if I do that, I might get a subsequence. But actually, I always need to match all of C. Does that make sense? I always need to match all of C.

So in a sense-- Let's see, how can I do this? I need to match all of C. But I also need to make sure I'm using exactly n over 2 characters from C in B. And exactly n over 2 characters from C in A? Does that make sense?

So how can I satisfy that condition? Now I understand why I used prefixes before. And I swapped it to suffixes here. But we'll make it work. How can I remember how many characters I assigned from Alice versus Bob? As I I'm matching characters in Alice and Bob, I need to kind of remember where they point to, or how many I've already used in Charlie. So that I can divvy up the remainder in here.

Oh, actually, this works in a different sense. There's 18 different ways we could do this. So, OK, so I need to remember how many I've already used up, so that I can be sure to allocate exactly that many characters in the future to either Alice or Bob. So how can I remember that?

I can just remember. How many do I-- I'll do it the way that I did it before, which is I can remember-- I can remember two different things here. I can remember how many things I have left to match Alice, in C, or I can remember how many things I've already matched in C to Alice.

If I talk about how many things I've already matched, then I can index this thing by the sum of those things. If I talk about how many things I have yet to match, I have to do n minus the things. So those are the different parameters that we can do. We'll do what's in my notes. And I'll try to fix it.

So what we're going to do is remember-- or figure out how many things I'm still needing to match in C to Alice and Bob. So I'm going to call this k-- sorry, ki. i is associated with A. And B is associated with j. And kj, this is going to be the-- I have to write this down.

So this is going to be what kind of output do I want to my subproblem? I just want to know if these things are-- if he's a fraud or not. So this is going to be a Boolean. So true if can match ki length subsequence of suffix A, suffix is this guy, and length kj, I guess, kj length subsequence of suffix B, ji, or Bj suffix, to all characters in.

And now what is this in? This is the hard part. Do I need a separate index for C to know where I am in C? In a sense, yes, I need to know where I am in C, how much I have to match. But if I need to match ki to kj to all of them, then there better be ki plus j things left in C.

So in a sense, I don't need to remember that information again. It's not independent to my other parameters. I can compute it. I could throw it in, but I can determine it from the other parameters. And so I want to match it with the suffix of C of length ki plus kj.

So I think this is the only part that is going to be annoying to me. So this should be suffix of all of the things minus ki minus kj minus 1. It's just this. And why is that?

If I have matched to everything, ki and kj are both 0. And I should have nothing in C, which should n colon. We're at 0 index, yes we are. Whenever I use Python notation, I'd better be 0 index. Does this make sense as a subproblem?

I mean, it's confusing. But hopefully, it makes sense. What I'm going to try to do is I'm going to match some number of characters in this suffix, which is hopefully longer than ki. Otherwise, it's-- I'm going to be in a base case where this is impossible. And some subsequence of this matched completely into this.

So that's-- those are my subproblems. I'm going to try to relate them now. We have x, i, j, ki, kj, what is this going to equal? Well, we've got Booleans. So this is and false otherwise. That's a Boolean.

So I just need some subproblem I recurse on to be true. So what's the commutator for some of a bunch of choices, Boolean choices, any one of which may be true? I want to combine a bunch of them. I just want to see if any of them are true. I'm going to or over them.

I'm going to or over four choices. What are my choices? Either the first thing in A matches with C, the first thing in B matches with C, or I don't match with either. So those are my four choices. So if I match with A, i plus 1, I recurse on a smaller suffix of A and a-- by adding-- oh, this all just works, great.

Kj, this is i, if Ai equals Ci and Ai is greater than 0. So if ki is greater than 0, I need to match an i. So this conditional doesn't even make sense unless I've evaluated this ki to be bigger than 0. Otherwise, I'm trying to access i of n. So I'm just putting this conditional on there.

Same with matching B, ij plus 1, ki, kj, plus 1 if-- sorry, this should be minus 1. I have fewer characters that I have to recurse on. So that's a typo in my notes. Bj equals C. Oh, this is not Ci. What is this? It's whatever that thing is. So I'm going to just say question mark and kj is greater than 0.

So I'll fill that in in the notes. It's going to be some complicated expression that looks like that. It's exactly that expression. Yes, that it is. So it's that thing.

OK, then we have two more choices. Either I-- if I didn't match Ai, I may match Bj in the future. So I only want to reduce Ai. So xi plus 1, I leave everything else the same. Assuming if i less than n, I don't want to move off the end of this thing, or x, i, j plus 1, ki, kj if j is less than n.

So those are my four choices. If I match the letter n, great. Otherwise, I decrease the size of my subproblem and I recurse. So fun recursion, topological sort, these subproblems only depend on what? Larger i? Not quite. Larger j? Not quite.

Changing k or-- these don't even change here. So we're going to use depend on larger, I guess, strictly-- that's kind of an important thing, i plus j. Because at least one of these two things is increasing. And then the nice thing about that is it kind of tells us when we should stop.

We should stop when either i or j get to n. We should know enough, at that point, to be able to determine if we succeeded or not, possibly. So we have our base case. What's the easy base case? When we succeeded. When have we succeeded?

If we have nothing left in A and B And we have nothing left in C. I have nothing left to match. So I have nn. And I don't need a match anything else. That's just going to be true.

All roads point to this subproblem to get to a true solution. Otherwise, we have some false base cases. If you set something up like this and you only give us a base case that's true, and you're oring over the things, your answer will always be true. So you're not having any discriminatory power at all. If you give us a true base case, you better be giving us some false base cases, or one, at least.

So in the case where the first one is n And we have some i, kj, this is going to be false if what? If we have nothing left in Ai or A, but this guy is positive, we got problems. Otherwise, this thing is 0. And we'll just try to match everything up here.

And eventually, we'll get down to this base case or something where this thing goes to 0 and we've got a problem. And the same goes for the other side as well. If we run out of things in B, when the number of things we need to match in B is greater than 0. So those are our base cases.

The original problem is what? Yeah, it's just going to be one of our subproblems, nn, and then n over 2 and n over 2, trying to match half of the things in C with half of the things.

AUDIENCE: The first two arguments should be 0.

JASON KU: 0, thank you, because we're 0 index, yes. Again, that's-- switching from prefix to suffix in the middle is fun. And it better be the case that n is 2, or else it's obviously false-- or is even, or else this is obviously false. And then, the last thing, which I'm not going to write down, is we have a constant work here. Because I'm just checking the value of four subproblems and a conditional for each.

And I have how many subproblems? i loops over n, j loops over n, k and kj loop over n over 2. So I get a quartic number of subproblems, quartic running time as designed. So I'm not going to write that down, because I'm quite a bit late. I'm probably going to do-- just do one more problem, which is sad, because the last one is about Gokemon Po, which is fun, fun problem.

Gokemon Po basically relies on I'm trying to catch a bunch of pocket monsters, just monsters, I think is in this. And you can either go to a location and catch that monster for free, but that costs money, because I have to ride share there. Or I don't have to go to that location, and I buy it on my in-app purchase, but that costs me a different amount of money.

But buying it from the in-app purchase kept me at the location I was previously, wherever I was. And so the point of that problem is I need to remember where I was last. So that I know how far I need to travel to get to my next monster.

So that's going to be the last one that I'm not going to be able to get to. Number 3 is a problem about tapas. So these all come from spring '18. The first two came from a problem set. These next two come from the final exam that year. Obert Ratkins is on a diet.

But he has a dinner at an upscale tapas bar, where he got many-- he's going to order many small plates. There are n plates of food on the menu, where each plate has a certain information, it has a volume, the number of calories in that dish. And a sweetness label, basically 0 or 1, whether it's sweet or not.

But he's on a diet. And he wants to eat no more than k calories during his meal, but wants to fill his stomach as much as possible, because he wants to feel full. So he wants to maximize the volume that he fills, even though he wants to reduce the number of calories-- restrict the number of calories.

He also wants to order exactly s sweet plates. So we've got this other condition where I need to make sure I'm eating a certain number of sweet plates. It might be useful for me to remember how many sweet plates I've already eaten. So I make sure that I eat that number, without purchasing the same dish twice.

So here's a condition that's similar to the knapsack 01 problem, versus a knapsack kind of general problem. Am I allowed to take more than one of these things or not? Here, it's a restriction that I'm not allowed to take a plate more than once. And I'm going to try to describe an order nks time algorithm to find the maximum volume of food Obert can eat given his diet.

So first thing I'm going to note here is one of the things that we talked about at the last dynamic programming lecture was is this a polynomial running time that it's asking me for. Actually, on your problem set 8, you were asked on each problem to categorize whether the running time of your algorithm was polynomial or not. And actually, you don't have to solve the problem in order to answer that question if we give you the running time.

If we give you the running time, you can just take a look at that running time and be like, oh, is that polynomial and the size of my input. And here is it? All the ones previously were. This one was order n. That was order n squared, because the n was the number of things in my input, the number of words it took to give you that input.

Here, what do I have? I have a triple of numbers for each plate. There are n of them. So n is polynomial. s polynomial because s is smaller than n and it's positive number. But k, k is just some number in my input. It's representable in, potentially, one word, that's the assumption. But it could have exponential size depending on the size of my word of my machine.

I don't know how big k is relative to n. And so this is a pseudopolynomial running time. Because k is just a number in my problem, similar to subset sum, similar to knapsack, which you guys did in lecture and recitation. And so if we ask you on an exam, which we probably will, whether certain running times are polynomial or not, that's the logic that you go about it.

How big is my input? What is my running time that I'm trying to evaluate? And can I bound each of those terms in terms of the size of my input? If not, then you say it's pseudopolynomial. All right, so let's try to tackle this problem. Already, because we've got pseudopolynomial, you're thinking maybe this is going to be knapsack-like or subset sum-like.

What do I need to-- I'm just going to go straight for subproblems here. Actually, I should probably say what my things are. Meh, this is fine. I gave notation up there, didn't I?

So we're going to have subproblems. I'm going to-- I want to maximize the number, the volume of food. So that should probably be the output of my subproblem, the max volume on some subset of dishes. I'm going to choose suffixes here. i and some other stuff is going to be max volume of food possible for plates Pi to Pn. Going to assume one index here, because why not?

But do I need to remember information along the way? Yeah, just like with subset sum or knapsack, I need-- I have this calorie limit. So it's going to be really useful for me to know how many calories I've already eaten, or how many calories I have left in my budget. So let's say j, using at most j calories from the remaining dishes.

And I need to make sure that I'm eating exactly some number of sweet plates in the future. And I need to remember, as I eat a sweet plate, the number of sweet plates I need to eat decreases. And so I want to generalize that. I'm going to put an s prime here to denote eating exactly s sweet plates.

OK, so that's my subproblem. I've got tons of board space. I'm going to go ahead and use it. Relation, we've got x, i, j, s prime equals-- OK, I'm trying to maximize volume. Probably, I want to be maximizing over something.

This combinator is kind of what I like to call it. Usually, what you're doing in dynamic programming is making some kind of choice or combin-- combining, combinating, combining some number of subproblems and choosing which one's the best. If you just list a bunch of options here and don't tell us how to combine them, that's going to be a problem.

Because we don't know what your dynamic program is doing at all. So it's really useful for you to be able to tell us how you're combining your subproblems. Here, we're doing a maximization over the different volumes possible. If we decide to eat the plate i, then we get Vi in volume, we fill our tummies with Vi in volume. But then we have to recurse on using one fewer plate, because we can't use that plate again.

And we've decreased the amount of calories in our budget. And I'm going to say s prime minus si, because si is 1 if it's sweet and 0 if it's not. So it's kind of nice that they kind of gave us this notation here. I can just subtract it off if it's there. I don't have to do this conditional or something.

And I don't ever want to go below these budgets. So I'm just going to say if Ci is less than or equal to j and si is less than or equal to s prime. So that's going to make sure that I never have these guys go negative. Otherwise, I don't eat the plate. And that's kind of the easy case, because I just go i plus 1, j, s prime. These things didn't change. I just have one fewer thing left.

So I'm maximizing over these things. This one's an always. It's not an if. So I just have two choices. And maximizing over them, topological sort order, here, I'm always recursing on a thing with larger i. Depend on larger i, so acyclicl, happy.

Base cases, what's the good case? I get to the end, I've reached the end of my menu, I can't look at any more plates, I'm stuffed. And I've already forbidden myself from going negative on the calories. So that should be all good. But what do I want on the third parameter? 0, I better have eaten exactly s plates.

So I want to get down to x n plus 1, because I'm 1 index, j, for any j, 0. That's going to be 0. I got no calories there. But it's a good thing. It's a good place. It's fine. 0 is good.

This is done, I don't know. OK, there's another base case. What's the bad base case. I get to the end. I'm always increasing i. And so I better be doing something on n plus 1. I got to the end.

j again, is going to be non-negative. Because we're always going to be in our calorie budget. But if this is anything other than s prime greater than-- if it's anything but 0, what is that going to be?

AUDIENCE: It'd be minus infinity.

JASON KU: Minus infinity. I never want to be in this situation. If I do my dynamic program and I get a minus infinity up at the top, that means there is no path to this subproblem here, where I'm happy. I'm always sad. And so I return that the maximum volume of food Obert can eat and maintain his diet is not possible.

Essentially, there aren't s dishes, sweet dishes in the thing whose calorie budget are below my limit. And that's probably an easier thing to check than in this band. So we have our original subproblems now. Solution is given by what?

Just one of our subproblems, it's just seeing what's the maximum volume. I don't have to retrace my steps to figure out my thing. I just-- I say one of the subproblems, it's using all of the things on my menu, using my entire budget k, and trying to get exactly s things. That's going to be my output to my algorithm.

And this takes what time? How many subproblems do I have? I have n plus 1 subproblems for this parameter. I have k plus 1 possible things for this parameter. And I have s plus 1 possible things for this parameter. So I get order nks subproblems, subproblems. How much work per subproblem, just a max of two things, so constant work per subproblem yields order nks time total.

So those are three nice practice problems for you, two that are polynomial, one that's pseudopolynomial. You have one more example in there, which is the Gokemon Po problem, which is a fun problem. It involves remembering additional information that's not-- not really a pseudopolynomial number in your problem.

But it's the location of where I was last, or where I was going. So take a look at that problem. It's another kind of non-trivial way of expanding subproblems. OK, and with that, good luck on your quiz 3.