[SQUEAKING] [RUSTLING] [CLICKING]

JUSTIN SOLOMON: Right, so today is going to be our first, I believe, of two problem sessions covering dynamic programming. I've learned dynamic programming is one of these interesting parts of an algorithms class where, somehow, the people that are really good at it are completely disjoint with the people that are good at all the other parts of the algorithms class. So for some of you guys, that might be promising, and for others, maybe a little bit less so.

So maybe we'll spend just a minute or two reviewing the basic ideas we're going to apply in these problems because they'll follow, more or less, the same template. Although, of course, as usual in 6.006, we like to put some interesting window dressing around it so that it's not totally obvious what you're doing. And then we'll do a bunch of sample problems.

Right. So let's talk a little bit about dynamic programming and the basic idea here. So, dynamic programming is kind of a funny outlier in 6.006 in that-- for example, in the data structures part of the course, we learned, like, what-- now I'm struggling to think of a data structure-- like a useful-- like trees and arrays or whatever. And these are actually things that you can code. if you look and see if there are-- well, plausibly, it could be an implementation of a tree in there somewhere.

And so these are useful algorithms that you can maybe even read the pseudocode. And there's a universe where you really do translate that pseudocode into something inside of your laptop. Dynamic programming is a little bit less so. This is more of a meta-- I don't know if you'd call it a meta algorithm or problem solving approach or what, but it's not like you somehow say, I'm going to apply the dynamic programming algorithm to this problem.

But rather, it's sort of this big class of things that all follow a similar template or sort of approach to thinking about problem solving, which I think sort of explains why, actually, in some sense, the last couple of lectures that you've seen-- and, I guess, if I'm getting the time sequence of our course right, the next couple of years that you will see-- and the problem sessions actually start to coincide in the sense that when Erik was teaching you guys dynamic programming, how did he do it? Well, he didn't write down-- well, he sort of wrote down some template for dynamic programming, but then we just did a bunch of sample problems. And that's exactly what we're going to do today.

So, somehow, all of these things are just going to converge in this part of our course because dynamic programming, it's really more of a way of life than any particular algorithm. And this is a pattern that I think you see a lot in advanced algorithms. Like, for example, in my universe, in numerical analysis, when you talk about the ADMM algorithm, it's actually a totally useless algorithm. What matters is applying it to a particular problem. And this is sort of, I think, a more mature or grown up way to think about a lot of things in algorithms, that pretty soon, this sort of general purpose stuff that's useful all the time, I think, it starts to disperse a little bit in favor of different patterns and mechanisms that you're used to thinking about. So there's my 10-second, sort of, philosophical introduction to what we're doing, during which I've managed to chase this table across the room.

You know, I played-- I did college on the west coast, and I thought I was going to be a music major. And there was a piano master class where we forgot to put the little clips on the wheels and there was an earthquake, and I just thought I was really nervous because the piano was literally slipping away from me. I can never think of that Chopin nocturne in quite the same way.

But in any event, in dynamic programming, Erik laid out for you guys a particular, sort of, set of steps that are a useful problem-solving approach in the dynamic programming universe. In today's problem session, I'm going to try and help you guys translate a little bit from this template to what it means to actually write code to implement a dynamic programming algorithm because I think it's a little easy to forget that here. But, on the other hand, on your homework, when you're writing out answers to algorithms problem, it's perfectly fine to follow this template even letter-- I guess-- literally letter for letter and answer each of these questions. And then the remaining glue that you need to actually write the code is not terribly exciting from an algorithms theory perspective.

So the basic idea here is that there's a lot of different problems that can be written recursively, in some sense. Certainly, we've encountered many of those in this course. In fact, I think the bias in the way that we've presented algorithms that don't have to be recursive is to write them in a recursive way. And the point here is that when you have a recursive call and you repeat something, you give the function the same input more than one time, you might as well remember what you got the last time you saw that input, and then you don't have to do that computation again. Really, in one sentence, I think that's roughly the logic behind all these dynamic programming things.

So there's no reason to be too redundant with lecture. For just the 10-second overview, I think that there's an example which is simultaneously good and misleading, which is this Fibonacci sequence. It's good in the sense that the logic of dynamic programming is really easy. It's bad in that the runtime is kind of weird to think about. But remember, though, your Fibonacci sequence looks something like f of k equals f of k minus 1 plus f of k minus 2. And if you look at your, sort of, recursive call tree here-- like, let's say that I do k equals 4. Then it's going to call-- my function f is going to have to evaluate it at 3 and 2. And then the 3 is going to evaluate at 2 and 1, and so on.

And the thing to notice is that when I call f of 4-- or, rather, f of 3 here, if there were 3 somewhere else in my tree, I get the same number, so, in particular, f of 2 and f of 2. Both of these are going to take some amount of algorithmic work, but if just the first time I see a 2 I have a little piece of scratch paper and I say, oh, any time I see k equals 2, just return this number rather than doing recursive calls, then, in effect, if there's any subtree underneath this thing, I've just pruned it from my tree. And so that's the basic logic here.

And that's basically the paradigm that's going on in this SRTBOT acronym, which is you first take your problem and divide it into subproblems. That is mysterious. Why is this board moving? Oh, there's a phone in my pocket and I bumped against the wall. I'm not used to this classroom. Right. Yeah. So the first thing you want to do is to write my problem as this sort of form. Notice just that we've done this a lot in this class, we've written things recursively. The difference here is the sort of argument that goes into recursion is typically, maybe, a little simpler than putting some giant data structure inside of there or something like that.

So, for instance, merge sort, you could write in this paradigm. I guess we covered that, but it's probably not the most natural way to think about merge sort. Then we need to relate our subproblems to each other. So, for instance, in the Fibonacci sequence problem, I just gave you the relation that-- what defines the problem. Incidentally, this is, what, a model for the reproduction of rabbits, I think, if I remember reading the history of the Fibonacci sequence.

And then, I think, to me, the most-- not necessarily unnatural-- but I think the thing that maybe is hardest to translate to an algorithm if you're thinking about writing code is this-- oh, man, this is going to be a problem-- this idea of topological order. The basic idea here is that if f of 1 depended on f of 2, and f of 2 depended on f of 1, I'd be in a lot of trouble, right, because somehow my tree would never converge, for one thing, if I made these recursive calls and I'd never be able to memoize or, kind of, remember a value when I move on. Right? And so the idea here is that there's some ordering of my subproblem so that I can build up a solution.

And there's, sort of, two dual ways to think about why that's useful. So in the memoization universe, what do I do? I just add an if statement saying if I've already evaluated f of k, return it. That's perfectly fine. The other thing I can do is if I write my problems in topological order, then I can sort of go in the reverse direction and build up my memoization table. So, for instance, for the Fibonacci sequence problem, I could do f of 1 and then f of 2 and then f of 3 and f of 4 all the way until I get to the k value that I actually wanted. And those are just duals of the same coin. They're exactly the same approach.

Although the memoization version, sometimes you can prune out subproblems that you didn't actually need to solve. So, for instance, maybe this was f of k minus 7, and so I can skip a few indices in my array. I don't think, typically, that has a big effect on runtime for the problems that we've seen, but it could, plausibly, in some universal. I'd have to think about a problem where that makes a difference.

Right. And then I think the BOT part of SRTBOT is a little easier to think about. You have to make sure that this recursion has a base case, like when is this thing going to stop. That's exactly the same. It's just any recursive algorithm. The O for original, I think, is a little bit retrofit to make SRTBOT sound nice, but I think the idea here is that you need to go back to your original problem and make sure that it corresponds to one of the function calls that you've written in all this complicated stuff. Hopefully that's a reasonable characterization.

And then, finally, the t is more-- these are for describing your algorithm. The last one is for analyzing it. And, again, the BOT part of SRTBOT almost applies to anything we've done in 6.006, like you should always analyze your run time. OK. So, in any event, that's my 10-minute version of the last couple lectures and, I think, more or less, enough to get us started with some sample problems here. Sorry, I couldn't help it. I like to teach things.

OK. So, right. So in our problem session, we have a few of the homework problems from last year to go over. If it makes you guys feel any better, I got myself all balled up on one of them last night while I was preparing for today. And I look forward to doing that in front of all of you guys now. Right. So, I'm afraid of this, so I'm going to go to the next board.

OK. So in our first problem, Sunny studies-- this was-- somehow, the cute naming conventions we have in 6.006 got really meta in this problem, because there's a problem about Tim the Beaver. But, as we all know, Tim is MIT backwards, so he happens to fit into this goofy game that Jason likes to play in writing homework problems. Anyway, but it's also the MIT mascot. Anyway, I got very excited.

Right. So what's going on in this problem? So Tim the Beaver has kind of an interesting-- you know, mathematics, I think you would call this a martingale if you flip the coin a little bit when he solve this problem. But luckily, Tim the Beaver is a deterministic kind of a guy. And he looks at the weather outside, and if it's a temperature t-- apparently, Tim the Beaver is OK with boiling. The higher the temperature, the happier Tim gets. So this is a first derivative kind of a phenomenon.

In particular, on a given day, if I have a temperature t, Tim the Beaver has two things that he can do to change his mood. Apparently, Tim the Beaver's mood never stays fixed. It always goes up and down. In particular, he can either go outside, in which case the happiness increases by t, OK, or he can stay inside, in which case his happiness decreases by t. OK.

So every day, Tim the Beaver, he wakes up he-- I really want to say that he checks for his shadow, but that's a gopher, right? i any event, he wakes up in the morning, he checks the weather, and he makes the determination does he want to go outside or not. And if he goes outside, he gets happier by an amount that's equal to the temperature. If he stays inside, he gets less happy by an amount that's equal to the temperature. By the way, I think our solution is perfectly fine if temperatures are negative here, in which case, I guess, everything would flip intuitively. But there's no reason to get too hung up on that.

But, of course, there's a twist here. So Tim, as with many of you, has n days until his final exam. And he's worried about studying. Yeah? So, in particular, he never wants to go-- he's come up with a personal resolve to never go outside more than two days in a row. Yeah? So, right. And so the question is-- right, because that way he has to stay inside and study at least one out of every three-ish days.

OK. So the question is how can Tim maximize his happiness. Incidentally, in machine learning, sometimes they call that minimizing regret, which I always found to be a very sad way to think about algorithms when there's a totally dual version. But Tim's an optimistic guy. He wants to maximize his happiness subject to this constraint that he cannot go outside more than two days in a row. Right? So if I go out on Monday and Tuesday, I have to stay inside on Wednesday. Yes?

AUDIENCE: I think there's no effect to his happiness when he stays in.

JUSTIN SOLOMON: There's no effect to his happiness when he stays in.

AUDIENCE: At least, that's [INAUDIBLE].

JUSTIN SOLOMON: No, it says with a decrease in happiness when t-- oh, when t is negative. That's not actually going to affect our problem at all.

AUDIENCE: It's not going to affect it.

JUSTIN SOLOMON: Sure, yeah, I can fix this live. This is what happens when I do the problem myself before looking at the answer and then don't check it closely. Fine. So let's change that. I like this problem better, somehow, psychologically. But that's OK. Right. So, Jason correctly points out that if you actually read the problem, what's asked there is slightly different, that when he goes outside, his happiness increases by t. If he stays inside, his happiness does nothing. Right? So it stays the same.

My apologies, so Tim the Beaver is a particularly optimistic beaver. His happiness can only increase in time, assuming he lives in a climate with positive temperatures. OK. I think I've got it right now. Cool. We'll see if I can still do this. Yeah, I think basically nothing changes. OK, that's great.

All right. We're going to do it. OK. Right, so the question is how do we solve this problem. And thankfully, I think we put the easiest problem first. And, in particular, if we're following our SRTBOT paradigm here, somehow there's a set of subproblems that are staring us in the face. That's the word I'm looking for.

In particular, well, there's sort of only one index in our problem, which is what day it is. So the obvious thing to do would be to say, can we figure out the maximum amount of happiness for days, say, i to the last day? By the way, if I do that, I'm using the prefix version of my problem-- ah, suffix version of my problem. I could also do it the opposite way and work from the end back in. Maybe if we have time all the way at end, we'll do the second one. But it doesn't really matter.

OK. So, in particular, just to add a little bit of notation, let's say that t of i is equal to the temperature on day i. OK. And now we're going to make a new thing, which is going to be the actual variable we want to compute. This is going to be x to i, which, we'll write, is the maximum happiness that you can achieve if you only consider the calendar from day i to day n, I guess inclusive. OK. Incidentally, just for convenience, we'll assume that x i is equal to 0 if I go past the end of my array, which I think is, kind of, a, typical thing to do in these type DP algorithms.

OK. So the question is can we actually come up with a recursive algorithm that computes x i using this nice, sort of, typologically acyclic way of thinking about our problem. The answer is obviously yes, or I wouldn't be here today. And so in the absence of a smarter idea, let's just do the Toucan Sam approach here and follow our nose and see if we can just write our problem in terms of other ones.

So, in general, let's say that Tim the Beaver wakes up on day i. He has, basically, two decisions that he can make, right? He can either stay inside or he can not stay inside. He can go outside, right? So let's just basically handle these three cases. So in case, one he stays inside. Well, now what happens to his happiness? Well, according to my revised version of this problem, nothing, so, in particular, what do we know?

Well, if he stays inside, then he has-- any decision he can make tomorrow, it doesn't matter. He can go inside, he can go outside, whatever, because by having stayed inside, he's earned himself two free days of going outside if he wants. Right? So, in particular, in this case, we can convince ourselves that this is true, I think. Yes, so, in other words, while he gets no utility for today, he wakes up tomorrow and he can make whatever decision he wants. OK.

The second thing he can do is go out. This is where things get a little tricky. Right? Can I just do, like, take t i and add it to x i plus 1? What goes wrong?

AUDIENCE: You go out three days--

JUSTIN SOLOMON: Maybe you go out three days in a row, right? Somehow, you have to remember that, right? And so that's where things are a little bit of a headache, that, in particular, if I go out today and tomorrow, I can't go out the day after that. And somehow, if we just dealt with this one case as t i plus x i plus 1, we wouldn't remember that. And that's a problem.

So, instead, what we can do is think of there being two subcases, right? So what we're going to assume is that not only does he go out today, but that he's free to go out tomorrow. And we're going to make that recursive assumption as we move down our array. So if we do that, well, now we have case a and case b. So in case a, he goes out today and he stays in tomorrow. Yeah. OK. So what happens in this case, well-- by the way, I'm using this kind of weird arrow notation. I don't know if this is good or not, but, essentially, the point is that I'm keeping track of cases, and then eventually I'm going to want to have to take the max overall of these things. So I don't like the equal sign because somehow that's a little misleading.

Right. So in that case, well, he gets the utility of having gone out today. Tomorrow he stays in, which means that the day after tomorrow he can do whatever the heck he wants. He has free reign. So I can write that using this recursive call. OK. Similarly-- right. I'm getting the hang of this. Sorry, this is way too entertaining for me. I can play with this board all day. OK.

So in case 2b, he goes out today and he goes out tomorrow. OK? So--

AUDIENCE: He's a party animal.

JUSTIN SOLOMON: He's a party animal. He is an animal and he's going out a lot. Right, so in that case, what happens? Well, he gets that. He gets today's utility. He gets tomorrow's utility. The day after, he has to stay in, so we might as well skip it. And then he can do whatever he wants the day after that. OK? So if we go back, I guess, technically, we should revise our definition of x a tiny bit, that it's not the maximum of happiness-- well, we can convince ourselves that it's the same thing. But really, it's not the maximum happiness for day i through n. It's the maximum happiness for day i through n under the assumption that he has permission to go out on day i. Right? And that's really what's going on in our recursive set of calls here.

OK. So does our recursion make sense here? Cool. All right. So let's see here. So if we're following our SRTBOT-- I keep reviewing papers that use the word paradigm a lot, so I feel like I should do that. So what is t? It's the topological order. Notice that x i only depends on larger i's. So in terms of our topological order, the dependence graph is really simple. It's just a line, so remember that you can think about topological order or you can think about being acyclic graph. Those are equivalent. We covered that in this course. I kind of like thinking about acyclic graphs. So x1 depends on x2 depends on x3 depends on x4. That graph has no cycles, so we're good. Right.

So next we have to come up with our base case for our recursion. Notice that the way I have chosen to solve this problem is by calling future indices, which means that my base case sits at the end of my array because that's sort of like the lowest down on the recursion train. The recursion chain is what I was going for, but I kind of like the recursion train better. In particular, on day n-- well, if he has permission to go out on day n, he can do one of two things. He can either go out or not. It doesn't matter, right?

So, in particular, we can say that that's the max of 0 or t of n. Remember, I didn't tell you that temperatures have to be positive. Maybe he's a Celsius kind of a beaver. OK. Right. And then in addition to the-- for convenience, notice that, like, there's a universe where I look beyond the end of my array in my recursive call here, so I should probably think about a few extra x's. Obviously, the utility of going out on a day that doesn't exist is 0. So we can say that x n plus 1 equals x n plus 2 equals 0. OK. I've managed to use way too much space for one simple algorithms problem. OK. Yeah? I get credit for that? OK.

Right. So now we need to do the o and the t. So what's our original problem? Well, remember that he wants to maximize his happiness starting on day one, so our original problem is just x of 1, or is it? So, remember that Tim the Beaver-- your instructor is very sloppy when it comes to actually reading the problems, as you saw at the beginning. A second mistake for which I would have personally lost points were I to solve this problem on my homework is that it didn't ask for just the maximum amount of happiness that Tim could achieve-- that's not very practical for your everyday beaver-- but rather, he wants to know the actual plan. He wants to know what days he can go out and what days he can't. Yeah? And I haven't actually told you how to do that, right? I've only told you how to compute x, which is just the maximum amount of happiness.

If I were you guys, I think this is a perfectly reasonable simplification that's like a warm up problem to solve. In fact, I would argue it's less of a warm up and more the crux of the problem-- and then going back and making sure you can convince yourself that you could actually reconstruct the solution. My way of solving this was ever so slightly different from the one in the problem, but they're equivalent, which is to say I can make a second array-- I won't write it down, because I'm slow at writing-- that just says on every day, whether I took option 1, option 2a, or option 2b. And now I can reconstruct my plan very easily, right?

So I look at x1, if I took option 1, then I stay in and I look at day two. If I took option 2a, then I can label today, tomorrow, and the day after. Oh, wait-- yeah, that's right. I can label today's choice, tomorrow's choice, the day's after choice, and then look three days later and recurse that way. Option b is kind of similar. So a reasonable way to reconstruct the actual set of what days you go out and what days you go in is just to remember, as you do your memoization or whatever, whether you did option 1, 2a, or 2b. And then it's pretty easy to reconstruct from there. Maybe I'll let you guys convince yourselves of that at home or in the last 8 seconds if you happen to be the two audience members that I have.

And then, finally, we need to do our time thing. And most of the time arguments here follow more or less the same pattern, which is you count the number of subproblems and the time per subproblem, you multiply those two things together, and you get your runtime. We're going to see in one problem on this problem set that's not quite right because we have to account for some precomputation. But in this case, it is. Right, so let's see, what are our subproblems here? Well, essentially-- I guess I didn't actually say it, but you have to take the max of these three values.

This is the max of three expressions which have a constant number of plus signs and lookups and memory and all that good stuff. So each subproblem takes order one time. How many problems are there? Well, there's, I guess, n plus 2 max, if you want to be conservative about it. So, in particular, there's order n subproblems, right? So all I have to do is multiply these two things together, and my algorithm takes order n time. And that's our solution to problem number one. Any questions so far? Yes? Uh-oh.

AUDIENCE: When I was thinking about the problem beforehand, I was wondering could you use base cases-- right now, we have two different kinds of base case, a base case for x of n and a base case for things after. Can I remove the first one and add an x of n plus 3 equals 0 as well? What would that do?

JUSTIN SOLOMON: Could I remove the first one and add an x of n plus 3? Yeah, I guess that's fine. I'm sorry, that's not a particularly helpful answer for the people watching on video. My answer to this question you can't hear is yes. So the question, to repeat, was this base case was somehow kind of complicated-looking. To be fair, it's the one that I was given in [INAUDIBLE] assignment, but that's OK. But the question was is this truly necessary. In particular, can I get rid of the x n case and instead add a third day past the end of time, which also has value 0?

And if you, kind of, look at that plus case b, I think-- or rather, case a-- think you can convince-- well, case a and b, for that matter-- you can convince yourselves that these are equivalent, right? That's absolutely right. So I could add a third day after the end of this thing, which also has value 0. Or, by the way, I could just say in my code if n is-- if i is bigger than n, return 0. That's the same thing. Yeah, and then I guess I don't have to worry about that x n case. Yeah, these are the same. To each their own. Fabulous question. Any others that I can answer while we're at it? Cool. All right. So that's problem one.

Writing too big-- I don't like this big chalk, you know. OK. So problem two is the one that got me all hot and bothered yesterday. So let's see if we do any better in front of people, because that's usually the best way to improve a problem solving skill. Right. So in problem two, which, annoyingly, is also probably the most practical problem on this problem set. Essentially, you have a-- I suppose I should write some stuff down.

So in problem-- I used the wrong side-- two, you have an operating system Menix-- whatever-- which is-- apparently, it's very simple.

AUDIENCE: Menix, Unix.

JUSTIN SOLOMON: Oh, I get it. [LAUGHS] It doesn't mean I have to like it. Right. [LAUGHS] So, in Menix, apparently, the only thing that my operating system can do is compute edit distance between files. And it wants to do so efficiently. So we have that a file is a sequence of strings. And I believe we say their length is less than or equal to k. That's going to come into play a little bit later. And the strings are basically just lines of the different files.

So there's three different ways that we can change a file. So here are the changes we can do. Change number 1 is to add a line. Change number 2 is to remove a line. And change number 3 is to swap. But a caveat for this interesting model of what's cheap and what's not is that, apparently, swapping two lines is cheap because they exist in memory. Like, maybe I'm, I don't know, using a linked list or something to store files, and so swapping two pointers isn't so bad. But inserting and removing a line is hard because, I don't know, memory allocation is expensive, like Menix is actually operating on clay tablets.

And I can chop my clay tablets into different slices and just pick them up and swap them, and that's fine, but if I want to add a line in my file, I have to go to the Tigris and Euphrates and pull out the-- or whatever it was, the Eugris and the Tiphrates-- and pull out a stone. It's a lot of work to make a new line or to dispose. So these are expensive and this is cheap.

And so the question that I'm trying to sa-- that I'm trying to say and that the problem is trying to ask is that you are given files A and B with n lines each. We want to know what the minimum number of non-swap operations it'll take, and so, in other words, the minimum number of time for you to add and remove lines to transform A into B, essentially, with low cost.

And, in fact, just to be nice-- I think it's actually kind of a critical hint in this problem-- we give you the runtime. And I'm going to, kind of, ignore it in my answer, notice that I did something wrong, and then go back and fix it. This is different from the way the answer is written, where god came in and said, like, oh, we observe that we're probably going to need this thing, so we're going to go ahead and do it here. I think that's, maybe, not representative of the logic here. Right, so the runtime here is k n plus n squared.

The first thing to note is there is a k here. Yeah, and so someday we're going to have to compare strings, because that's what k is. And I think that's the hint that's implicit in this problem. It's easy to miss. And so, indeed, what we're going to notice is we're going to look at our solution and say, well, wait a second, if we didn't incur a factor of k, we must have done something wrong. And, indeed, that's going to be the case, but it's only a minor fix to change it.

AUDIENCE: There's another important distinction on this [INAUDIBLE].

JUSTIN SOLOMON: Oh, I'm sorry. Yeah, right, when I swap things, they have to be adjacent. I can't write at the bottom of the board. That's supposed to be a, d, j, for those watching at home. But they have to be-- you can only swap lines that are adjacent, as they appear in their original file. I'll say it out loud rather than try and write it because you it's going to take the rest of the lecture to do that. OK, any other things I've forgotten? There's a high likelihood. I'm bad at this. OK.

So this one was annoying. And it's not actually annoying. It's actually a relatively easy instance of a very well-known dynamic program plus a tiny bit of additional stuff, which is called edit distance. In fact, I think if you guys are looking for intuition on this problem, you might google that one first as, sort of, a-- what was that?

AUDIENCE: [INAUDIBLE]

JUSTIN SOLOMON: Oh, in fact you're doing a recitation. Oh, that's why it's not totally unreasonable to come up with the answer here, even better. But even if you hadn't, you know, this is just another dynamic programming problem that's just a little bit more annoying than your average dynamic programming problem. Now, the solution written out in the course notes works from, sort of, the last line of the file downward, in some sense-- upward, whatever. I, like, literally lost two hours of my life trying to think about editing files from the end up and just getting myself all upset and confused. So here, I'm going to attempt to do it in the other direction and probably introduce a bunch of mistakes in the process.

So what do we do in dynamic programming if we don't know what else to do? We do sort stuff, SRTBOT. And so let's do that here. So, in particular, what are our subproblems? This is a little bit funky. So actually, even before we do the S of SRTBOT, let's think about our problem a little bit. Let's think about what it actually means to edit a file because this is what helped me think about the right answer here, which is to say, you know-- so what's going on?

I have, like, two documents. This is document A. This is document B. Each one of them is composed of a bunch of lines. And I'm basically trying to turn A into B. And the only thing I can do is scroll out of line, insert, just hit the Enter key, or do a third thing where I kind of like swap two things that are adjacent to one another. That is the only thing I can do. And the way I like to think about this problem-- there's kind of an annoyance here, which I think is a typical annoyance in dynamic programming problems, which is that the order of operations suggests that this problem is a lot, combinatorially, more difficult than it is, because, like-- OK, let's think about how I actually edit documents-- like, I spend 2/3 of my day editing bad grad student writing-- is like I'm jumping all over the place between different lines. Like, first I delete this line, then maybe I go to the bottom of my document and delete some other one.

That would be a big problem from a dynamic programming perspective. I can't jump all over my document, because keeping track of that whole edit history is going to be somehow combinatorially ginormous. Right? I'm not the Track Changes button in Microsoft Word. I want the minimum number of changes. And if I have to recurse over all possible edits to every single line in any order, that's an awful lot of factorials and 2 to the n's floating around that I don't want to have. Right?

And so that's the sort of crux of the challenge here, is to organize my approach to editing these files in a way that doesn't require me to have to do this sort of combinatorial jumping all over the place. And I think it's also the one where there's a sort of-- like I know Jerry Caine at Stanford talks a lot about the recursive leap of faith. Like, somehow dividing your problem into organized subproblems, that's really where the challenge lives here.

So if I were a more organized PhD advisor, the way that I would edit a file, or a clay tablet, I guess, in this case, would be linearly, that I might as well do whatever the heck I'm going to do to line one before I move on the line two. And at the end of the day, even if I did stuff in a different order, you can convince yourself that I could always order it in such a way that all the edits that I do to the first line, kind of, happen before lines later in the document, with the possible exception of this swap thing. But we'll see that somehow doesn't matter.

And, moreover, if I do an edit, I might as well do the edit to make things better, right? There's no reason to start willy-nilly inserting and removing lines. I might as well always do an operation that improves stuff. And so thinking about that sort of logic leads me-- ta-da-- to a particular way that I might write down my S, my some problems here, which is to say that I'm going to think about editing my document line by line. So, in other words, once I've dealt with line one, meaning that I found some way to mess with it and make it match line one of the other guy, I'm just going to think about removing it and then think about the rest of the document. You start saying, aha, that sentence sounds like recursion. And that's right. That's how we're going to solve this problem. OK?

So, in particular, here's going to be our thing. I'm going to do a slightly different one in the solution-- so you guys should all be vigilant-- which is I'm going to write x ij to be the min work to convert. I'm not a Python programmer, but hopefully I got this right. i colon is going to be everything from i to the end of the file. So in other words, this is the suffix version of our problem-- and into B j colon, like that. OK.

So, in other words, I have a little-- it's kind of like a video-- like, think about Tetris. Once you get that full line of blocks, you can just throw that line of blocks away and the whole video game moves down. There's somehow something very similar going on here, which is the second I've managed to get a match for line one of document into line one of the next document, I'm just going to throw it away and pretend like I have two documents with one less line in them.

Now, the thing that got me all hung up last night, my original problem assumes that both of my documents have the same length. But here, I'm not making that assumption, right? And, essentially, what we're going to figure out is that that actually doesn't matter a whole lot, that if I end up with one document of length k-- well, I shouldn't use k-- one document of length l and another document of length 0, what's the amount of work that I should do to convert? Well, l, because my only choice is to insert a bunch of lines in one document, by the way, or delete a bunch of lines from the other. Those are dual to one another. They're exactly the same.

I'm philosophizing a lot because I'm also convincing myself that my answer is OK in the process. OK. So this is going to be our set subproblems. And now we have to do the r, right? We have to relate, something we struggle with in the math department sometimes. And, essentially, the way that I went about this is to just do a billion different cases of all the possible edits that I could do to line i and line j. And that's perfectly fine in this problem. I think the problem is a little slick. And the way that they're written the solution, they've convinced themselves that some things are equivalent to others and removed them. But you don't have to. As long as there's a constant number of cases, your golden Ponyboy.

So, in particular, let's think about some cases. So first of all, if line i matches line j of my document-- remember that it's not really line j. It's like making a document that just happens to start at line j. It's like taking scissors. Well, then I can match them with zero-cost because the beginnings are in the same place. And I can move my Tetris came down one, and that's perfectly fine.

So case one, I think, is the easiest one, which is if A i equals B j, then I can just remove that line from both documents and move forward, in which case-- I'll use my same goofy notation-- I'm going to get that x ij. Well, I'm going to just increment i and j and keep going, like that. Cool? So what's something else I could do? I could delete a line. Yeah, so what happens-- OK.

So case two is delete A i, right? That's a different thing I can do the line i. Well, now what do I have to do? I have a document on the left-hand side, which is 1 line shorter. And on the right-hand side, nothing changed. But deleting a line cost me a dollar. So, in particular, I have that x ij. Well, what happens? Well, I got rid of one line, but I had to pay. OK. Let's think about some other things.

You could delete B j. This case actually isn't in the solution because it turns out to be unnecessary.

AUDIENCE: Well, we're only allowed to edit A.

JUSTIN SOLOMON: Oh, I'm only allowed to edit A? Oh, in that case, I don't have to delete B j. I really didn't read these problems very closely. That's my bad. This would have made it much easier. I really should read these things. Cool, so that eliminates half of the cases on my notes. Fabulous. Incidentally, you could do these things on the other direction and it really wouldn't change this problem a whole lot.

Sorry, you know, I have this bad habit when I'm reading research papers of reading the research paper I wanted to be there instead of the one that's actually on the paper. And, somehow, it's very similar phenomenon here. OK. Right. So, great. So I can only edit document A, which makes this probably easier than what I was worried about. Fabulous.

In that case-- ah, bananas. With our third case here, well, let's see, I could also insert a line. Let's see. So what ends up happening there? So I can only edit document A? So that makes my cases different than the ones I wrote down on my notes. Sorry. OK.

OK. So if I insert-- let's do this live. Yeah, OK, so if I insert a line at line i, I might as well make it match B j. There's no reason not to. Right? I might as well kill off one element of B while I'm at it. Yeah? So if I do that, what ends up happening? Well, I still have to match line i. I've just, kind of, moved it lower in my file. But I've, in essence, killed one line in file B by making it match this new line that I inserted.

In my notes, because I thought I could edit B, I said, OK, I can just delete the line in B instead. And somehow, logically, that's a little easier to think about. But these are exactly dual to one another. So, in that case, I have x ij. Well, I still have to deal with A i. I haven't gotten rid of it. But I've matched line j. So I paid $1 for inserting a line. And now I have that because I've gotten rid of a line in the other file.

If I stopped here, by the way, I would have it at a distance. But, unfortunately for me, I have one additional case, which is mild irritant, as they say, which is that I can swap. First of all, can I always swap? I mean, I can, but if I swap two lines and they still don't match the lines on the right-hand side, I'm kind of hosed, right, because you can convince yourself that in the next step, I'm going to have to delete something anyway. Swapping was free. If I swap and delete, that's the same thing as just deleting, so it doesn't really matter.

So, in particular, what that means is I might as well only check the swap if it actually helps me. Yeah? So, in other words, if I have A-- now you have to be a little bit careful because I'm swapping. So if the next guy in A equals B, the current guy in B and the current guy in A equals the next guy in B. Well, now I can swap this guy and kill off two lines in my files while I'm at it, right?

So, in this case, I get that X ij. Well, swapping doesn't cost me anything, and I killed off two things. So that's the recursion. So if I were to write this out on my homework, what should I do? Well, I shouldn't-- I mean, probably if you use this error notation, I don't think it would be a big deal. But really, you should add a line at the bottom saying that I can choose to do any of these things. So really, my recursive call is x ij gets the min of all of these 1, 2, 3, 4 expressions that I've written here. OK.

AUDIENCE: What if you have the first condition but not the second one?

JUSTIN SOLOMON: What if I have the first condition but not the second one? Ah, so that's a great question. Yeah, so the question was like, OK, well, what if I can match the next line but not the current one. Well, there's two different things you could do. You could either make another case for that. That's perfectly fine. In fact, you could do that. You could do that I matched the second condition, not the first one, whatever. You can just enumerate as many things you want as long as they're all true and there's a constant number.

Alternatively, you can convince yourself that actually is unnecessary here because a different-- so that's like swapping. But then one of those two lines is still a mismatch, so you're going to have to delete something in the next step. So you might as well just delete first, rather than swap and then delete. And so that's why that case isn't necessary. Yeah?

AUDIENCE: [INAUDIBLE] in the first case.

JUSTIN SOLOMON: Exactly. Exactly. So if you swapped and you killed a line, then, in effect, I think it's a combination of case 1 and case 2 here, if you, kind of, expand your recursion out. But if you're having trouble convincing yourself of that, that's fine, just add a case here. Yeah. Any other questions? I'm going to ask quickly because this problem makes me nervous.

AUDIENCE: [INAUDIBLE]

JUSTIN SOLOMON: Sure. OK. In the worst case, if we've done something wrong, you can certainly add another case here. I'll think about it at home. OK. So since I've managed to pontificate too long, let's keep moving here.

AUDIENCE: Can we swap them if they [INAUDIBLE] match?

JUSTIN SOLOMON: Oh, you know, the problem is it might have been--

AUDIENCE: --used in the final file. So if you can make swap one match one of them and not the other, then that's not OK.

JUSTIN SOLOMON: Yeah, because at the end of the day, the files have to agree. Like, you have to match B to A.

AUDIENCE: Swap and delete is cheaper than doing two deletes and two inserts.

AUDIENCE: No, no, no, but the swap and delete is illegal because you have to use both ones. That's a condition in the pocket.

JUSTIN SOLOMON: Oh, I'm sorry. That's a better answer. So Jason points out that if I swap, then I can't delete it because the way the problem is written. So that effectively removes this case. Otherwise, I think-- I guess Erik is--

AUDIENCE: Actually, there are two important conditions--

JUSTIN SOLOMON: Oh, sorry. I've managed to totally botch this, which is totally unsurprising. Yeah. So I think the problem also states that if you swap, the swap has to be useful. And that's why this additional case that Erik is asking about where you swap and then you match one line but not the other is unnecessary. You might be able to relax that by just adding a case here, but since the problem doesn't ask it, I'm not going to think about. OK. Right. So, under all the assumptions of this problem that I didn't read but are very important to solving this problem correctly, I believe we really have written down all of our cases here.

OK. So let's continue with our SRTBOT paradigm. So now we have all of our recursion. The topological order here is a little bit trickier than normal because now you've got a two-dimensional array, but it follows a pattern that's pretty typical here, which is that x ij only depends on other x ij's with higher i plus j. So I think about my graph of subproblems. If I wrote this in a 2D matrix, it always, kind of, points down and to the right, maybe, which is what's making it acyclic. This is a very typical pattern in these sort of two-dimensional dynamic programming problem.

All right. So let's see here, SRTBOT. So we need our base case. This isn't too bad because, essentially, when you have boring documents, they're very easy to match to one another. So, in particular, for any i, if I'm at line n plus 1-- in other words, I have a blank document that I'm matching to document i-- well, how much work do I have to do?

You have to be a little bit careful. This is where the suffix version of this problem is a little bit more annoying than the prefix one-- or have I managed to swap those backward again-- that, in particular, the remaining number of lines looks like n plus 1 minus i, which is different than in the problem. It's just i because they're working in the other direction-- in the solution, rather. And similarly, you need a second case for those two here, right? So you have x n plus 1 j is going to be n plus 1 minus j. Cool. OK.

So we're going to continue with SRTBOT here. So what is our original case? Kind of by definition it's x 1, 1 or 0 0, depending on how you index. And then, finally, what's our runtime? Well, let's see, there's n plus 1 squared subproblems, and, of course, that's equal to order n squared. The subproblems are just a constant amount of work, so they're each with constant work. So our entire runtime is order n squared. And hopefully by watching me be confused in front of you and think through this problem, you too will see how the problem solving procedure can happen in your own disorganized brains. OK. So that concludes our treatment of this problem here.

That, I think, is the hardest one. So the other two, thankfully, are much easier to think about, I thought. But I never liked edit distance. I remember seeing that and undergrad algorithms getting confused. OK. So the next problem, problem 3 here, deals with Saggy Mimsin. And she has a bunch of block, and she wants to stack them on top of each other, as one does. And as a young structural engineer, she has a few criteria on her problem.

Let me go to the right page in my notes here. Right. So this is problem 3. So we have that block bi has size that looks like width wi by height hi by length li. I remember getting confused in elementary school about the difference between width and length all the time. To me, those always sounded the same. But it doesn't really matter, because she's happy to rotate her cubes any way that she pleases.

There's a key detail which I did remember to actually read in this problem, which is that she has at least three of each type, where type here means that I can permute these three numbers any way that I want because that's the same as just rotating a block. But any time she has one block that's like 1 by 2 by 3, she has at least two more in her bag somewhere. OK.

APPLE WATCH: It's 6.

JUSTIN SOLOMON: Oh, 1 times 2 times 3 is equal to 6. Thank you, Apple Watch. OK. So that's odd. So she can orient her block any way that she wants, meaning she can rotate it in any fashion that she'd like. And so what we're trying to do, what we want is the max height where she's stacking her n blocks. I suppose I should say they're n blocks.

So she wants the max height that she can achieve. But just to be kind of annoying, or because, again, she's very concerned with structural stability-- she lives in an earthquake zone-- she would like with the condition that each block is strictly supported on the block beneath it. Right? So in other words, if this is the base of one block, then the next block that's stacked on top of it has to be strictly contained within the block below it. Right? So does the problem make sense? Have I omitted any critical details? I don't think I have this time. This one's a little easier.

AUDIENCE: [INAUDIBLE]

JUSTIN SOLOMON: Oh, yeah. And she can't do anything crazy. She can't do a weird, like, balance it on this edge kind of thing, which-- Erik is absolutely right-- could actually give her a taller tower than you could get if you're only allowed to rotate blocks 90 degrees. I don't think the problem states that explicitly, but this isn't a trigonometry class, so I think we're in good shape.

OK. Right. So that's our basic problem here. And this is one of these problems that is going to be a dynamic programming problem, but, again, similar to many of the things that we saw in lecture, is not totally obvious how, because somehow she has this big, disorganized bag of blocks. You could imagine a universe where there's 2 to the n different things she could do, right? For every single block, she could decide whether or not to put it in her stack, and then she has to do a bunch of other work to verify whether she can stack them while supporting the strict support condition or not. So, initially, that seems kind of annoying.

So what we have to do, which, again, is pretty common to a lot of these problems, is place some order on it. I mean that both in the entropy sense, and also, like, literally, were going to order stuff. And, in particular, we'll see that this problem has a lot in common with that longest common subsequence problem that we saw in lecture-- increasing subsequence, sorry.

Right. So here's some observations about our problem which are going to help. First of all, when we stack our blocks, we might as well always align the shorter side of the block on top to the shorter side of the block underneath it. Right? Let me draw a picture of what I mean.

So let's say I have a really-- a block whose base kind of looks like that and then another block which is also rectangular that I sit on top of it like that. Then notice I could-- so in this case, the shorter edge of one block is aligned to the longer edge of the other. Notice I can rotate it 90 degrees and it still supports on one another. So there's never a case-- you convince yourself with just a few inequalities-- where I don't, kind of, always put the long side parallel to the long side of the guy underneath and the short side parallel to the short guy underneath it Does that makes sense? Cool. So that's observation one.

Observation two, can they ever-- like, let's say that Maggie actually-- sorry, Saggy actually had not just three blocks of a type, but like 25. So she's just has hella blocks. My question is does it matter. The answer is no because this word here is really critical, which is that there's strict support. So your block only has so many faces.

And, in fact, by observation one, really all that matters is which of the three types of faces is sitting on top because we can just always rotate it. So there's three configurations of every block, so, at most, can any one configuration appear more than one time? No, because of the strict support condition. Right? Otherwise, the rectangles would match up, and that's against the rules.

So, in particular-- oops, the number after two is-- the number after 1 is 2, which looks like that. OK. Right. So, in particular, there are only three orientations. This is just which of the three edges of the block is the one that's going away from the floor, the normal to the ground. And moreover, each can appear less than or equal to 1 time. That's good because it limits the size of our problem.

And, finally-- oops, I collapsed two of the cases in my notes into one case here. But that's OK. And, in fact, notice that the problem tells us that she has at least three of each type. So, in a sense, if the problem-- if you observe one of a block you might as well just throw away the rest because you know that you can use it at most three times. And she has three of that block. We can't use it more than three times, so in a sense, that's just superfluous information. OK. Right.

So this allows us to put a little bit of order here, because notice that when I look at the stack of blocks here, what do we know? If I look at the length of the long side and the length of the short side in the plane of the ground, those numbers have to decrease on every level of my block. They can never increase. That's what the strict support conditions says, combined with observation one, actually, even without observation one, which is good news, right? So this is what's going to allow us to impose order on our problem, namely, that we can sort by the edge lengths because we know that we have this support condition.

OK. So let's fill in some details of our algorithm. OK. So originally-- already we can see that our list of blocks is kind of useless because the width, length, and values are sorted in ways that don't matter. Moreover, if we have more than three of a given block, that's somehow not super useful. So, instead of that, without loss of generality, let's assume-- so WLOG here-- we can always take our block and assume-- I'm going to do this slightly different from my notes-- that the width is less than or equal to the height is less than or equal to the length. OK.

So every block, if this isn't the case, I could go down my array of blocks and sort. And sorting a list of three numbers is constant time. OK? Right. So what does this allow me to do? Well, I'm going to say that a block type actually is an ordered set where the third number is going to be the axis that points up. And the reason to do that is that we know that we can never use that more than once for any type of a block. Yeah?

So now I'm going to make a new list of blocks with a capital B because I like blocks. And it's going to look like the following. So if width-- so if w is less than h is less than l, then I'm going to take every block and duplicate it three times. Notice that I might end up with a list with, like, nine times of every block, but we'll fix that later. Right.

And it's going to look like the following, which is that, OK, I'm going to have wi, hi, li. This is like describing a way to stack my block because it's saying this is the short side, this is the long side, this is the vertical side. And there are three cases where any one of these guys can be the vertical side. So there's one.

Let's say that the h is the vertical side, then w has to go before l. So it would be wi, li, hi, and a third one where the third guy is w. H is less than l, so it would be hi, li, wi. And those are all the different ways that I can, sort of, orient these blocks in my stacking, assuming that I impose condition one for convenience here. OK.

I'm going to make a new list of blocks where I take every block in my original set and I just duplicate it three times this way after I sort its coordinates. And now, well, what do I need to do? For one thing, this thing may have too many blocks. I might have a block that's repeated more than one time and I can't do that. And moreover, it's going to be convenient to have this sorted because I've got to stack these guys eventually. Yeah?

So I'm going to sort that list. And I want to do it-- I can never say this word-- lexicographically, meaning that I'm going sort out the first coordinate, and then the second and the third lexicographically. Notice this length is 3 n if I had n blocks to start with. So this entire thing takes order n log n time, which is important to account for. And then I can remove duplicates. I'll let you guys convince yourself you can do this in order n time. An easier way would've been making second array and just kind of move-- and only add stuff when you didn't see the same thing before. OK.

And, finally, now these are ordered in a really nice way because I can stack my blocks, but only ever looking to the right in my sorted list, assuming that I'm stacking from the top of my tower down, which is, I think, sort of what's going on in this thing. OK. So now, finally, we can do our SRTBOT. And I might do S and R and T and then allow you guys to think about the rest because, as usual, I'm talking too much. OK.

So now this is starting to look like a subsequence problem because, essentially, when I stuck my blocks, if I use this block here-- again, if I'm stacking from my tower from the top down-- all the blocks that can sit underneath this one have to be further to the right in my array because of the way that I sorted. Now, that doesn't mean that I can put anything on the right underneath this guy, but it does mean that I know nothing to the left can go underneath this guy. That's the way to think about it. OK.

So it is going to be SRTBOT. So S, what I'm going to say is that x i here is equal to the maximum height of my tower, and I'm going to-- taking a little bit of inspiration from our subsequence problem that we've already seen, I'm going to force myself to use block i. We'll see if that's convenient. i and possibly-- just for fun, maybe we'll do the prefix version of this problem this time. So now I can use any of the previous blocks. So I can use the first i blocks to make a tower, but I'm forced to use block i.

By the way, from now on, when I use indices, it's into this sorted array. OK. So this is a problem. Obviously, if I could solve for x, I would be done, because I could get the maximum height by just iterating over all the x's and choosing the biggest possible value here. And the question is, how do I do this recursively?

OK. So here's our recursive step. So let's say that I use block i, well, because we know we have to. Right? So, in particular, we have-- now I'm seeing why they didn't use this notation in their answer, but that's OK. Let's use another letter to refer to the third coordinate.

AUDIENCE: v i for vertical [INAUDIBLE].

JUSTIN SOLOMON: Yeah, let's say v i is always the third coordinate. We've already used w, h, and l, and I'm afraid if I reuse them after sorting, it's going to confuse people. So v i is the third coordinate of the i-th element of my sorted array. That's fine, OK. Right, so what is my height if I use x i here? Well, I get some height from vi. And in addition to that, I get whatever I stack underneath that guy.

So, in particular, I get that xi. Well, I get the height of the block that I just decided to use. And now, what are all my cases? Well, I could decide to do nothing else, like, just not use any other blocks. That gives me a height of 0. Or, well, let's see here. I could use the x's, but I have to be careful that I can actually stack them. So, in particular, well, I need-- yeah, I can take an x j value, but I've got to be careful that I can stick it underneath.

So, in particular, what do we know? Well, I can do anything from 1 to i minus 1 because that's sort of the definition of xi. But, in particular, I can stack it on top. So one easy way to do this is I just array-- I iterate through the first i minus 1 elements of my array, and I just check my stacking condition for every single one of them relative to block j, so, in other words, that the width and the height-- or rather, the first and the second coordinate satisfy the strict inequalities that I need. I'm phrasing this sentence neutrally because I forget whether this is increasing or decreasing.

But in any event-- so what do I do? I check all of the blocks that I could possibly stack from the index of the array perspective. I make sure that I could actually stack them, thanks to the size of the current block that I just decided to add to my stack. And I move recursively. OK. Right. So this is great because now we're in exactly the recursive scenario we wanted to be in, because x i only depends on x j, where j is smaller than i. And that is exactly our topological ordering that we need.

If you do that on your homework, you get a minus n for large n. OK. Similarly, what's our base case? Well, obviously, if I only have one block, I might as well use it. So in that case, we have x1 is equal to v(1) our notation here like that. Our original one, we have to be a little bit careful because of the way that I've defined x, because x assumes that I've used a particular block, so I have to say, well, I might not have actually chosen the very last block as the one I want to keep.

So I have to iterate. I can say that, really, my original is the max over i of x i. So one of these blocks has to be the block on top. I'm just going to iterate over all the possible ones and find it. And then our final thing to do is the runtime t. This one is mildly trickier than the previous runtimes that we've done so far in our example problem. In particular, how many subproblems are there? Well, there's n subproblems-- or I'll say order n because I'm always off by 1-- corresponding to each block in my stack here.

But how much time does each subproblem take, at least the way that I've written it here? Well, what I have to do? I have to loop over all of the possible blocks and find the one that I can stack on top of and then take the max. So there's a loop here from 1 to i. i is upper bounded by n. So this is order n subproblems times order n work per subproblem. So at the end of the day, my algorithm is going to be order n squared time.

And, of course, again-- I guess I promised it and then I didn't actually do it-- to actually implement this algorithm, there's sort of two different ways to do it. I could write a recursive call plus a table. The tables maybe initialized to a bunch of NaNs. And then I implement this function recursively. But before I do that, I say, if the table does not equal NaN, just return the value in the table and otherwise call this recursion. Or I can just have a four loop from 1 to n and build up the table one element at a time. And both of those are exactly the same from a runtime perspective.

OK. So I think I've managed to watch that much more than my notes or the written solution, but the problem itself is actually pretty straightforward. So if you guys read through the answer plus some of-- I think, actually, the hard parts of this problem were not the dynamic programming, it was all the observations you need to get there. So that's why I spent a little more time there.

OK. So, as usual, I haven't left myself enough time for the last problem. But we have a few minutes and that'll be sufficient to set up the parts. I actually found the last problem to be easier even though it technically is, sort of, two dynamic programs in one. So I think the logic is a little easier. OK. So

AUDIENCE: Use the backboard [INAUDIBLE].

JUSTIN SOLOMON: I think this is the backboard. Yeah, I was just realizing that this room doesn't work the same way as the other one. Yeah, this is embarrassing. You know, I spent all day thinking about topology, and this is like a classic kind of problem in that universe. OK. Well, we'll just erase one board at a time, and I'll try not to write three feet wide this time. Oh, this is probably the one board I shouldn't use. I don't think I like this classroom.

OK. So, right, in our final problem, we're given an n-by-n grid. And on our n-by-n grid, Princess Apple-- Banana--

AUDIENCE: Plum.

JUSTIN SOLOMON: --Plum. Princess Plum. Right, so here's our basic setup. There's a big grid of stuff, or maybe a small grid because I don't feel like drawing. And every grid square can have one of three things. We can have a mushroom. We can have a tree. Or it can have nothing at all. And our princess starts here, and she goes-- she wants to go there. And moreover, there's a couple of things that are worth noting here.

So, first of all, her path is quick, meaning that she can only traverse 2n minus 1 grid squares to get from one corner to the other. And, apparently, she's very into mushrooms, and she'd like to accumulate as many as possible along her path. That's the basic setup here.

So she wants to get from the upper left to lower right. And, in order to do so, she wants to take a quick path. Her main priority is to be efficient. But among the different quick paths, she wants to pick up a lot of mushrooms. It's understandable.

AUDIENCE: And not walk through trees.

JUSTIN SOLOMON: And not only through trees, thank you. So maybe there's some grid squares that are marked with a tree, meaning that you just can't go there. That's a tree. OK. Right, so that's our basic setup here. But the problem does-- it takes a bit of a twist, right? It's not saying just could be the shortest path, which would be very much like the last kind of unit in 6.006. But rather, the question is, sort of, what is the number of paths that she can take from one side to the other and what is the maximum number of mushrooms, is roughly the question asking, or at least what I remember from reading it last night.

AUDIENCE: The number of paths that maximizes the same number.

JUSTIN SOLOMON: That's right. So she has to take the most number of mushrooms she can, but there may be more than one path that gets you there that is quick, that satisfies this condition, in which case, she wants the count of the total number of ways you could get from one corner to the other. Why, you might ask-- why not?

OK. So, right. So, for instance, maybe there's a mushroom here. Now there's a quick path that gets her there and collects one mushroom. So there's exactly one. But maybe if there's a mushroom there, well, initially, it feels like maybe she could get two mushrooms. She could go there, go pick up the second mushroom, and get back. But we're going to see that this quick condition actually allows-- it doesn't allow you to do that. OK?

So, in fact, it'll turn out that quick paths can only collect one mushroom in this 3-by-3 case, so there's at least two different paths. Well, there's 1, 2, 3 different ways that she could collect one mushroom and have a quick path. OK. So the first thing to notice is the instructions are a little bit sneaky by defining quick paths, basically, by giving her no slack at all. Right?

And here's a basic observation. Notice that in order to give from the upper left to the bottom right, she's going to have to go down and to the right. Plausibly, she could also go up. She could try and go around a tree, but only plausibly. And, in particular, the question is how many times does she have to go down and how many times did she have to go to the right.

Well, she has to get to the bottom of the grid. So she's on grid square number 1. She has to go down, in this case, at least two more times, so, in general, n minus 1 times. She has to go to the right n minus 1 times. So what does that mean? She has to make 2n minus 2 moves. And that's a lower bound, right? So if she goes up, she's going to have to go down again, so it's only going to make it bigger. Right?

So at the very least, she has to do 2n minus 2 moves down and right to get from the upper left to the lower right. How many squares does she touch when she's doing that? This is a fence post problem. So she made 2n minus 2 moves, and she had a place where she started. That implies that just by moving down and to the right, she makes 2n minus 1 squares-- she touches, rather. So can she ever move up? No. Can she ever move to the left? No. And that, basically, is all you need to solve this problem. The rest of it's actually pretty easy.

So the basic observation here is she can only move down and to the right because if she moved up or to the left, her path would no longer be called quick, and that would be a problem. Moreover, every path that moves down and to the right is a quick path, assuming she reaches their target and doesn't hit a tree. OK? So that's the basic observation.

And notice that that already basically suggests-- it's, like, screaming out to you-- how to do dynamic programming because, literally, you have a table looking at you on the blackboard right now, and you have an ordering down and to the right that is acyclic. Yeah? OK. Have I slammed on the board enough times? The first time I taught at Stanford, I got negative course feedback that I had too much coffee and was slamming on the board a lot, apparently. I watched the video later, and, indeed, that was not wrong. OK.

So, right. So, we're going to call k-- this is going to be the max mushrooms she can get going on the entire path from the upper left of the lower right. So we want to know the number of quick paths that can achieve this number k. OK. So let's do SRTBOT really fast because I've got four minutes-- actually, a tiny bit more than that because we started late.

OK. Now, the kind of annoyance here is that there's two different numbers that we don't know. One of them is k, and the other is the number of paths. The problem didn't tell you how many mushrooms she can pick up. It does tell you that there's some path to get from the upper left to the bottom right, that there's not just like a row of trees somewhere, which I feel in my commute sometimes. But it doesn't tell you the number that she has to accomplish. And, initially, that's kind of annoying.

So maybe the first thing that we do is just compute k, like, the maximum number of mushrooms that she can collect on any quick path. And then we go back and compute that other count. That would be one problem-solving approach that we could think about a little bit. So, in particular, let's define our k ij to be equal to-- well, we can generalize our problem slightly, and say what is the number of mushrooms that I can get on any kind of rectangle embedded inside of my full problem, right? So, in other words, this is the max number of mushrooms, or m's for short, on a quick path to ij. So, in other words, she always starts in the upper left, but now she stops at any other grid cell. OK?

Because I'm running low on time-- no, I'm going to do this the way I want to do this. No, so we're just going to think about k. Yeah? So the question is could we compute just as value k, which certainly seems convenient. Princess Plum, she might as well know her enemy. She might as well know the number of mushrooms she's targeting, if she can get it.

So how could we do this recursively? Well, she has to get to position ij. And from our argument up there, she has to get there by either coming from up or to the left, the way we've chosen to write down this problem. So what are our different cases? Well, first of all, if there's a tree, you can't do a damn thing. She shouldn't even be able to get there. And for convenience, we're going to find you can argue that it's 0. We're going to mark this with a special number, and we'll see that that makes our notation a little convenient.

So one is if there exists a tree, then we're going to say k ij is minus infinity. Again, there's a philosophical question there. Does she get minus infinity mushrooms if she is standing on top of a tree? I don't know, because she shouldn't stand on top of a tree. But at least it's going to let us know that something went wrong in this grid square in the other parts of our recursion. OK?

And otherwise, well, what are our different cases here? Well, she always picks up a mushroom if it's there. She might as well. She's maximizing. In fact, the problem even says that she's really into mushrooms. She collects them automatically, right? So what do we get? I use unnecessarily fancy notation. This is an indicator of whether there exists a mushroom at position ij, meaning this is a 1 if there is, and there's a 0 if there's not. Sometimes this is indicated with a 1 with a little subscript, but whatever.

In addition to that, she might have picked up mushrooms along paths. And we know that her path to position ij either came from up or to the left. So, in other words, we know that she could have gotten the max from any path ending above her or any path to the left of her. So that's k i minus 1j, which, I guess, is to the left, and k ij minus 1, like that. And this can be used to fill in our whole table of k values.

In fact, since I'm low on time, I'll let you do the TBOT for the remainder of this problem. Essentially, I think the key observation is this one. Obviously, when she starts in the upper left, she gets 0 mushrooms because she's not standing on top of one. The problem says that. And this allows us to fill in our whole table k. So, in particular, this gives us our enemy now. We now know how many mushrooms she should have at every step of her journey.

In fact, it tells us a tiny bit more than that because it says if I'm at this grid sq-- at this particular grid square during my path, I should have this many mushrooms. If I didn't, then something went wrong. Yeah? So the way that the solution is written, they do two pieces of the recursion at once. You actually could have just solved for this k matrix first and then gone back and done the second half of this problem. And those are exactly the same.

And when I was writing my solution, this is the way I thought about it, because, somehow, I kind of felt like she might as well know how many mushrooms she wants to collect before she starts counting paths. That's like a secondary question, you know. And so this is one way to do it.

So in our remaining negative 2 minutes, let's think about the recursion for computing. Remember that we want to know the number of paths needed to collect that many mushrooms, the maximum number of mushrooms. Please let there not be a whole lot of stuff on this board. Ah, there's not any stuff on this board. That's great. OK.

So, in particular, now I'm going to define a second thing that I also I'm going to do dynamic programming on. I'm going to say that x ij is equal to-- and I'm going to make it kind of a sneaky definition here, which is the number of quick paths that end at ij with-- now, let's anticipate our problem a little bit. So at the end of the day, we're going to do x of n comma n because she wants to end up all the way down and to the right. And how many mushrooms does she want to have, now that we know k? She wants to have k of n comma n mushrooms.

Along the way, it would be kind of ambitious if she wanted to have k of n comma n mushrooms the entire path. But it would be slightly less ambitious to have k of ij mushrooms because, somehow, that's exactly what we just constructed in the previous thing where it was paths. Well, the last guy looks like a max. Now we're going to expect to see some plus signs here because we're adding up how many paths we have. OK?

And now let's come up with a recursive rule for the array x, and then we'll call it a day. So, in particular, [INAUDIBLE] one, if there's a tree, how many paths are there? There are no paths, because I can't get there. So then ij equals 0. OK? Otherwise, there ain't a tree. And now I have to be a little careful, right? So I need to write this like a little piece of code. You could have written this as a giant max, instead, or a bunch of cases and whatever.

So let's think about it like a piece of code. So, initially, I think there are no paths that get me k ij mushrooms. That's perfectly fine. And remember, we're going to keep applying the same piece of logic, which is that a path can only come from to the left and up. And let's think about those two cases. And by the way, we're going to use chi to equal this chi of stuff that we had in the previous expression. So chi is 1 if there's a mushroom at this place, and 0 if there is not. OK.

So my path can come from the left or up. I know that it can't come from up if the number of mushrooms that I got from up plus, potentially, the one that I got here doesn't align with the number of mushrooms that I should have by the k ij standard that I have set for myself. So if I want to write that out in code, the way I do that is I would say, if k of-- so let's say I look to the left first. This is like, you know, look to your left, look to your right, one of you will pass this exam kind of scenario.

And I potentially add a mushroom at my current position if there is one. If that is equal to k ij, well, what does that mean? That means that paths that went to the left were able to collect the number of mushrooms I need to get to the position I am now. So now I can add all of the different ways. Maybe I'll do a plus equals x of i minus 1j because any path that got to the previous guy and collected the right number of mushrooms can now reach me and get the right number of mushrooms.

And similarly, I can look up and do exactly the same logic. So if k of ij minus 1 plus this number is equal to k ij, then x ij gets an additional number of paths, like that. And now I do think it's worth spending 8 seconds thinking about our base cases here because, initially, when I first saw this, I panicked a little bit because it kind of looks like this would just end up getting a bunch of 0's because I'm just adding values of x's to themselves. I don't have, like, a 1 plus anything anywhere, which is kind of weird if you think about it. So all of the reason why the positive numbers appear in this problem is from the base case, which is kind of cool, I think.

This is like, I think, one of these things where if you anticipated a problem, then it's cool, and if you didn't anticipate the problem to begin with and you just wrote down these formulas, you probably wouldn't even think that it's interesting. But in any event, what is our base case? So, we'll do the B in SRTBOT. So, first of all, what is k of 1, 1? Remember, that's the number of mushrooms she can collect by starting at the left square and going nowhere. And that's 0 because the problem says there's no mushrooms in the upper left.

What's x of 1, 1? Well, this is the number of paths from 1, 1 to itself that collects 0 mushrooms, so that's 1. OK. And I think that the rest of the SRTBOT table here isn't terribly difficult to fill in. So I noticed that in kind of a funny way, all of these recursive steps are just adding 1 to itself a bunch of times. But, of course, the way you do that, the reason why you get a number that is interesting, is because of all these if statements and the fact that you can add two different pluses coming from two different sources. OK.

So I actually do encourage you guys to look at the code in the problem solution because I think it's a nice example of taking this recursive formula and then unrolling it into, like, iterating over a table. And that's a useful skill that I intended to do today and then didn't actually do very carefully. But with that, as usual, we've gone over time here. So we'll call it for the day. And I will see you guys when I see you guys. All right.