[SQUEAKING]

[RUSTLING]

[CLICKING]

JUSTIN SOLOMON: So today, we're going to continue in our discussion of dynamic programming. I actually found this set of problem session problems to be easier than the previous one. There's a funny thing, which is we learned in class about pseudo polynomial time style dynamic programs. Somehow, that language is a little bit liberating, in the sense that you're using parameters that you really shouldn't, when it comes to the runtime of your algorithm. Well, I suppose we should, in the sense that it's allowed, if you call your algorithm pseudo polynomial time.

But it somehow makes it a little easier to formulate your dynamic programming algorithm, because all the numbers are staring you right in the face. You don't have to be so careful about what's fair game and what's not, when you post your algorithm so long as it's efficient in the values that you care about. And so today's problem session has five problems squeezed instead of the usual four.

We'll see how far we get, seeing that I usually talk too much anyway. But I'll try to stay on schedule here, and we'll see how well we do. Any questions from our students about dynamic programming before we get started here?

AUDIENCE: You can cut one, if you want.

JUSTIN SOLOMON: We can cut one? Oh, I will gladly cut one if I run out of time. Yeah. OK, so without further ado, let's get started with coin-crafting, which is problem 9-1 here. So I suppose it should be Ceal Naffrey if I were to work through my splinterism properly here. In any event, we have a thief who's in desperate need of money, as with many thieves, or else, of course, they wouldn't resort to the world of prime. And Ceal Naffrey here has n identical coins.

I've been very self-conscious about the way that I write the letter I, ever since Eric pointed it out, and it's only gotten worse. Now it's inconsistent and hard to read. But in any event, so we have n identical coins. And of course, these coins have very distinctive markings. And so we can't possibly run away with them as is, because if you take them to your standard jeweler, they'll immediately recognize that these markings are bad and stolen. And that's not so good.

So instead, we can melt these coins into other objects. So our sneaky thief here has identified a potential buyer. And the buyer has a few criteria here. So we have a buyer, and the buyer has a very strange value system thing, that apparently, it's easy to take coins and make them into other things.

But in any event, the buyer, what they care about is not the fact that the coins are made of gold, but rather that they like particular objects better than others made out of said gold. So in particular, they have a different rate for each object, a different price they're willing to pay. And so they have a list of n objects that they're interested in. And each one is associated with two things of n objects.

They have a price-- so the amount that the buyer is willing to pay for that object, which, again, is not just the weight in gold, for some reason. There's a value added tax in this universe. And it takes a different number of coins to manufacture. Right?

So maybe I can make a golden chocolate fountain, and that takes 10 coins. But I don't want two of those. So if I make that, then the next thing I have to do is a figurine of Eric and Jason to put alongside it. And having more than one of those would also be creepy and weird. So I can only make one of each object.

And of course, my goal here is I have n coins. By the way, the fact that there's n coins and n objects for the buyer doesn't really matter. I mean, it will for the runtime. But you could imagine this being n and m. So I wouldn't be too hung up on that.

And what you're trying to do is, of course, maximize your revenue, subject to the constraints that you have n coins, and you can't make two objects that are the same. Hopefully, I've captured the essence of our problem OK. OK, fabulous.

So as with all of our dynamic programming problems in 6006, we have a paradigm for how to approach them, which has a cute acronym, which is SRTBOT And I think SRTBOT is a totally relevant and straightforward approach to this problem here. Yeah, so in particular, let's give ourselves a bit of notation. So we're going to number our objects between 1 and n, just for convenience.

So we'll say that Pi is the price of object i. OK. And we'll say that Ki is what the problem calls the melting number. In other words, if I want to make object i, this is the amount of coins that I'd have to melt to make that object. OK? So just a tiny bit of notation.

And in general, so OK, what do we do when we formulate our dynamic programming problems? Some problems we could solve that are smaller. And when we compose them all together, we get the final solution to our problem. And this particular problem, involving manufacturing objects out of coins, I think is a really classic one, when it comes to dynamic programming. This is the sort of thing where someday, when you're trying to pay for your tuition by doing these hacker contests online, this is the sort of thing that comes up all the time in that universe, right?

So in particular, the two variables here, when I make a new object, is what object did I make? And how many coins did I spend when I did that? So those are the two natural parameters to use, when I solve my dynamic programming problem. Yeah.

And of course, it's going to be recursive, in the sense that I can either choose to make object i or not. And it doesn't matter what order I make my objects in or similarly, what order I spend my coins in, which is usually a nice property to have in a dynamic programming universe. So in particular-- oh, this thing is my nemesis. This is front. I think this class is particularly tough for short people, because it's moving up and down all the time.

OK, so given our observation here, if we're doing SRTBOT, so what's our S again? Let me ask myself that-- Sub-problems. Thank you, Professor Demaine. Then essentially, what we want to do, based on what I argued verbally, is maybe define our variable x i, j the thing we're going to compute, to be the revenue from using i coins and objects 1 to j. OK.

So in other words, I have i coins left in the bank, and I'm only allowed to use the first j objects. And we can already see that this is kind of a sensible way to approach this dynamic programming problem, in the sense that there's an obvious recursion here. I choose to make an object. I have fewer coins. And I can sort of imagine there being a topological order, in the sense that I could first decide about object 1, and then object 2, and object 3, and so on, or vice versa, depending on whether you're a prefix or suffix kind of guy-- which I still get backward.

But the good news is that it doesn't really matter. What matters is formulating the equation. OK, any questions about our definition of the thing that we're going to chase after here? Fabulous. Ah-- OK, right.

So let's continue SRTBOT. So our next piece of our puzzle here is the R. I believe R stands for recursion. This is a new acronym for me, too. Ope, no, Relate. But it might as well be Recursion for most of these problems. Right, so the basic relationship here is that, of course, I can either use object j, or I can not use object j.

And in both of those cases, If j is the very last guy that I'm going to consider, that'll be a totally reasonable recursive rule, right? So in particular, I have that xi, j essentially can take one of two values. That's a parenthesis, in case you're wondering. And of course, you're trying to maximize your revenue. So let's do that, OK?

So we have two potential options. But we have to be a bit careful. Is there a case where I can't make object j? Yes, class, there is, which is the case where I don't have enough coins left in the bank, right? So I want to make that really expensive fountain, but I only have one coin. Then I'm out of luck, yeah?

So let's do these two cases. So first, if-- oops, I got my cases backward. That's OK. Right. So let's say that I choose not to make object j. OK, so what does that mean? So did I spend any coins? No.

And moreover, what is my maximum profit? Well, it's going to be the same as the maximum profit, using objects 1 through j minus 1, because I didn't use object j. Yeah? So in particular, that would be x. Let's see, I got it backward in my notes, so we're going to do it live, like that. OK?

And otherwise, let's say that I did choose to make object j. Well, what happened? So I did get some revenue now, right? Man, what on earth did I write on these notes? So I get the price of objects j here as my revenue. But I spent some coins in the process. So I have i minus K sub i, which is the number of coins. What's that?

AUDIENCE: K sub j.

JUSTIN SOLOMON: Oh, thank you-- sorry, K sub j. That's why we should be consistent with our indices when we write these things down. So I spent K sub j coins making this thing, object j. And moreover, I can still choose to make any of the previous objects. So it's still just j minus 1.

But I have to be careful, because I can't always do this. In particular, this had better be a positive number for n, or at least a non-negative number. Because if I end up with a negative number of coins, well, that that's not a physical universe that I choose to be in. So in particular, what we need, in some sense, is i minus k minus j, and k and j to be greater than or equal to 0. Or equivalently, i is greater than or equal to K sub j. I think you guys could all do that one at home.

OK, and this is our recursion. Hopefully, I've gotten it right, because it disagrees with the crazy thing I wrote in my notes at 1:00 AM yesterday. But I think it's pretty straightforward. Essentially, either I can choose to use the last object, or I choose not to. And either one of those, of course, decrements j, because that's the index of the object I'm considering. And I either account for the price, but have to pay in gold, or I don't account for the price. So implicitly, there's a 0, and I don't have to pay in gold. OK, good.

All right, so let's continue with SRTBOT. We might, later in the session, relax going through every one of these steps, because a lot of the arguments are similar. But for now, we'll do one or two carefully. So T, I believe, stands for Topological order. And here, it's staring us in the phase. Because notice that xi, j only depends on x question mark, comma, j minus 1. [LAUGHING] Right?

So of course, on your problem set, you should write things more carefully. But the basic point here is that there's a clear topological order, just by looking at that second index. Because if you think of all your x's as variables in a graph, which we've actually drawn in lecture-- so maybe these are all the i's and then the j-- so i goes down, and j goes to the right. Then essentially, this argument is saying that all the arrows point left in this graph.

I suppose the way I've drawn my arrows isn't quite accurate. But it actually doesn't matter. The only thing that matters is that it goes from right to left. But I'm going to erase this, so you don't remember it. OK.

So in general, just when you want to make your topological order argument, I think the totally sensible one is looking at the indices of recursion and then just trying to find some number that decreases. Incidentally, if you take a differential equation course, that's roughly how you prove that a lot of those things converge, too. So there's a generic math check that we use a lot. What do you call that in ODE where have some number that decreases?

AUDIENCE: I would call that potential function.

JUSTIN SOLOMON: Potential function-- that's a perfectly sensible one. It's not Lipschitz. It's some other mathematician. Anyway, OK, so let's continue SRTBOT. So next we need B, which is our Base case. So in this case, it's pretty straightforward. If I don't have any coins, I can't make any money. I have a t-shirt that says that at home.

And moreover, if I can't sell anything, I can't make any money. So those are pretty straightforward cases. So we have that 0 equals x of 0, comma j. Remember, the first index is the number of coins you have. So this is saying, I can't make anything. I don't have any coins.

And similarly, equals x i, comma 0 for all i, j. So this is the coins and the objects. OK. So let's see. I keep writing these problem sessions too big and then spending half of it erasing. So let's try and fit this on one board here.

So we're going to do SRTBOT. Then the second-- no, the first O, because there's no O in SRT-- is the original problem that you want to solve. So of course, you start out with n objects and n coins. So the original problem we want to solve is equivalent to computing x income n. And then finally, we've got to do our runtime. T stands for runtime, or Time, I suppose.

So first of all, how many problems are there? Well, there's x i, j. Both I can go from 0 to n. This is a great way to be off by 1. So there's n plus 1 squared sub problems.

And how much work does each sub-problem do? Well, it does boring work. It's just a formula. Right? So there's order 1 work per sub-problem. So the overall algorithm takes n squared time.

So I promised to do something in our last problem session. Then I didn't actually do it. So I did think I would spend just a minute here translating what this SRTBOT thing would mean in terms of code. Because I think that it's a little bit implicit here. In particular, I think this step here, I mean, you will see that it really clearly is going to give you an algorithm. But I think it's kind of easy to, again, just to forget what your code actually looks like.

And actually, the coding problems on these problem sessions are almost too interesting and can obscure it a little bit. So I thought we'd do a boring problem and show you that it's really not so hard to do this. And in fact, we covered two different strategies in class for how to take SRTBOT and convert it into a piece of code. Although, they might have zipped past you in this 2x speed thing that you can do now.

So here are two options. One of them is called memoization. And the other, I don't know, bottom up, I guess, is a reasonable phrase to describe. And so I thought we'd do them both, because they're both easy for this particular problem. OK. Right. So let's do that.

So is this necessary on your homework? Strictly speaking, no. If you've gone through SRTBOT, then essentially, everything that happens after that is boilerplate, in terms of converting these steps into a piece of code or an algorithm. But I do think, just for understanding why SRTBOT makes sense, it's worth thinking about for a minute.

So option A here is memoization. Ironically, I've taught and TA'd algorithms a few times, and I never actually knew what memoization meant. So I learned something from Eric's lecture the other day. Which remember, memoization is the key thing in what, apparently, is a made-up word, is memo-- if you're back in the day, and you had a steno pad, and you were writing down stuff. Because that's how you solved problems, with your slide rule.

Then essentially, the idea is that, if I compute x i, j for some i, j pair, I shouldn't compute it again. I should just write it down on my memo pad. Stenoization sounds better to me. But I suppose we'd have to go back to the 1940s and fix it. OK, so let's actually write down. I'm going to write down pseudocode, that will probably look more like Matlab, because I'm that kind of guy.

So let's say that I wanted to make a function, which, I guess, is revenue of i, j, like that. Bah. And this is the thing I wanted to compute. That's actually going to be a problem, because we're not going to be able to see. OK, right.

So in addition to this, I'm going to pass in an array x, which is going to be like my memo pad. This is going to be terrible coding practice, but easy board coding practice. So if I were in C++, maybe I'm passing by reference, so that when I edit-- that's a treble clef, but whatever-- when I edit x, it actually persists when I recurse.

This is terrible coding practice, and you shouldn't do it. OK? But it's just going to be because I don't want to write too many lines on the board here. And maybe we initialize. We have some helper function to-- let's see, we want our revenue to be big. Well, actually no. We'll just initialize it to not a number, so that we know that we haven't computed it yet. How about that?

OK, so what should we do? Well, if we're going to memoize, the first thing we should do, any time that I call my revenue function on an i, j pair is check that I've already computed it. Yeah? So in my goofy, bad board-coding style here, what could I do? I'd say, well, if I've already computed it, then this thing won't equal NaN anymore and won't be not a number.

So I can say, OK, if x is not equal to not a number-- so in other words, it is a number-- return. And this, I think that this little line of code here, it gets a little lost. But this, we should put sparkles around it. This is the magic of dynamic programming, because I just killed recursive calls.

Even if i and j are 17 and 23, if I already computed it, I'm done, right? I don't have to call my recursion again. OK. And otherwise, what am I going to do? Well, otherwise, I'll maybe call-- do I want to write it all down? I don't want to write it all down.

So otherwise, I'm going to evaluate, R, where R is this formula over here. Notice that this will require recursive calls, right? And I'm going to store it in x i, j and then return x i, j. OK?

So basically, the only difference between what we've seen in the first 2/3 of 6006 and now is this beautiful line of code, saying, if I already computed this thing, return it. And this is the memoized version of our algorithm. I think this is the easiest one to maybe think about.

But actually, from a runtime analysis, it's a little bit annoying. It's not in the sense that we convinced ourselves that SRTBOT is OK. But of course, if you're thinking about your recursion tree, what's happening is that you're maybe convincing yourself that this piece can be lopped off in your function calls. So you have to do your counting carefully.

There's a different way to implement the same thing. So this would be option B. This is maybe more efficient, maybe less efficient, depending on your problem. But these are all within constant factors of each other, for the most part-- not always, but for the most part. This would be bottom up.

And this is the idea of, rather than just taking our recursive algorithm that we already know, and then just checking a table to say, like, OK, did I already do this? And in that case, return it, in the bottom-up version, I'm going to build up my array x i, j because so there's no recursion at all. So what would that look like?

So in the bottom-up case, notice that, in some sense, memoization is a top-down strategy. I would call it on n, comma n. Here, we're going to start from 0 and work our way toward n, right? So we'll start with x 0, j equals x i, 0 equals 0 for all i, j. Obviously, I can do this with a for loop.

And now, well, remember, if we think about our topological order, x i, j only depends on previous j's, right? So it makes sense to have an outer loop, which is over j. And now, inside of this outer loop, I can compute anything that I want in that j column of x. And I'm in good shape, because I'm building it up one column at a time, right?

So in particular, now we can do our loop over i and then just have x j and now evaluate our R step in our SRTBOT paradigm. And notice that that's perfectly fine. Because by the time I get to computing x i, j, I've already filled in x i, j minus 1, which is all I needed to evaluate that formula.

So what are the advantages and disadvantages here? So notice that here, our runtime is staring you in the phase. Right? We have n squared sub-problems, order 1 work, and you're done.

On the other hand, there's some possibility. If you were an old-school AI person, you might be able to do some pruning on your left-hand side that I can't do over here. Right? So here, I'm literally evaluating every entry of x i, j. It's not the case for this particular problem, but maybe x i, j only depends on x i, j minus 5. This strategy is going to still build up that whole table. This one, maybe you can skip over some entries. So in practice, it could help.

On the other hand, here, I've got a bunch of recursive calls I've put on the stack of my computer, that here, I don't have. So I think, actually, because of the overhead of recursion, typically, the strategy on the right is preferred; also, for clarity. But that's a blanket statement that I shouldn't make.

OK, so anyway, I think I've done this problem to death. Are there any questions here? I just thought I'd fill in for something I promised last time and didn't actually do. OK, fabulous. So we'll go on to problem 9-2.

So this is continuing from last time in the saga of Tim the Beaver here. So I forget what Tim the Beaver was doing in our last problem session. But today, Tim the Beaver is going to the career fair. And as we all know, the only real purpose of going to a career fair is to pick up free stuff. We joke.

Actually, you guys should all go to the career fair. I got my first job out of college by going to a career fair and haranguing somebody that picks our booth, until they let me in. But in any event, so Tim the Beaver is not interested in getting a job, but rather just wants swag, just wants free stuff out of booths at the career fair. OK?

So in this particular problem, there's n booths-- is it booths? It's certainly not boots. I don't know. There are n booths, each of which has a swag. And in particular, each swag has a value associated to it, which is Ci, which is the coolness of object i. It additionally has Wi. This is the weight of object i.

And just to make this problem verbally difficult to communicate, there's a W-A-I-T associated with each object, which is the time it takes to wait in line and pick up object i. OK? And Tim the Beaver, if there's some ridiculously cool object with a little wait time, he might just keep getting in line and getting more of that object. So unlike Ceal Naffrey on our first problem, Tim the Beaver is perfectly happy to have more than one of the same thing. OK.

In addition to this, just to make this problem, in my opinion, slightly more annoying and point-losing, Tim the Beaver also takes one minute to get in line at any booth. So we're just going to have to remember that when we account for our time ti. OK. So let's continue adding some more constants to our problem.

So each booth has an object which has coolness, Ci, weight Wi, time ti. Tim is carrying a bag. The bag can hold a particular weight. b. So this is the max weight that Tim can hold in his bag at any given time. And finally, Tim is a greedy beaver, but he also can go home, or go back to his dam, I suppose, and empty his bag. Right?

And so h is the amount of time to go home and back and empty his bag, in the meantime. And again, just to be annoying, don't forget he incurs plus 1 to get in the next line. This is what made my answer wrong, and I'm bitter. So I'm going to keep complaining about it. OK.

So of course, what he wants is the max. What would economists call this? I don't know. But for Tim the Beaver, he wants the max total coolness, or MTC, which is, of course, a number that we're all trying to optimize, in k minutes. You might remember those old TV shows, where you get one minute in a grocery store to empty the shells into your cart kind of thing.

So he wants to do this, and the computation time that he's reserved for this is an order nbk. And I know it's a big setup. I tried to document all the different constants that are in this problem. I don't think I missed any here. OK, fabulous. So incidentally, continuing on--

AUDIENCE: What's k?

JUSTIN SOLOMON: k-- k is the total amount of time that Tim the Beaver has allotted to do his job fair scavenging. Fabulous. Any other? Cool. OK. Notice that this is going to be an example of a problem that was not kosher in last week's problem session, in the sense that k is included in our runtime. Right? But what is k? It's just a number. k doesn't scale in the size of your problem in a linear way. So it's not going to matter for how we solve our problem But it is just a feature that's worth pointing out. OK.

So how do we solve dynamic programming problems? We use SRTBOT, or a Señor BST, if you are watching previous iterations of this course. OK, so right, so let's do that. See, I'm trying to conserve space. This is not going to end up succeeding.

So again, what are the different sub-problems here? Well, what are the different things that are limiting Tim the Beaver? What are his constraints? Well, he only has so much time.

And this is going to sound more philosophical in intent, but time always moves forward for Tim the Beaver. So this is a pretty good candidate in terms of dynamic programming. Because there's not going to be some cyclical dependency. Remember that in dynamic programming, we're all about trying to identify topological orderings in our sub-problems. And when you see something like time, not only does it also begin with t, but it's useful in the sense that time moves forward.

There's never a case where Tim the Beaver purchases a weird Warp Speed airplane and somehow goes back in time. That doesn't happen in this particular homework problem and for Tim the Beaver's for sake, I hope in no homework problems. So this is a long-winded way of saying that time is a pretty reasonable constraint to put in our problem, not a constraint as much as an index, I guess.

Moreover, there's another thing which is limiting Tim the Beaver, which is the capacity of his bag. Remember, he can only hold weight b. But this one should give you the heebie-jeebies a little bit. Because this problem, as a twist, has allowed the bag to empty itself out. So it's not true that somehow, you can come up with sub-problems where Tim the Beaver is just monotonically decreasing the weight of his bag.

However, if he does choose to decrease the weight of his bag, he has to spend time doing it. So time continues to move forward for Tim the Beaver. And that's what's going to give us our topological order. Yeah? This is a little too philosophical, I guess.

But in some sense, this is a long-winded way of saying, for our SRTBOT paradigm here, a totally reasonable thing to do would be to have x i, j-- I don't think, in this class, we do this definition notation. So just x i, j equals the max coolness where he has i minutes, and he has j weight-- let me glance at my answer to see if it's left in his bag or weight that he's carrying. Left in his bag-- either one would make a reasonable problem. I'm just bad at looking at my notes and seeing them disagree with the board. So I'm going to try and stay consistent.

OK, so x i, j is the MTC, but in i minutes with j weight left, rather than in k minutes with b weight, which is going to be our base problem. OK. Oh no, I did it out of order. Disregard that. We'll get to the b in a minute. OK.

OK, so that's our sub-problems. So now let's do the R in SRTBOT. [LAUGHING] I hate this classroom so much. I'm sorry. So right, so let's say that Tim the Beaver has i minutes left on the clock. And he has j weight in his bag. He's got a number of actions that he can take. Yeah? And let's think about what all those actions are.

By the way, there's one thing that plausibly, Tim the Beaver could do. But he doesn't really need to, is do nothing now, but then in 10 seconds, do something. You could account for that in this problem, but there's not a reason, right? He might as well stack up all his actions and then leave all of his leftover time at the end. You convince yourself that that's sort of OK. Tim prefers a compressed schedule. Right.

So he's got a lot of energy, this Beaver. That's what they say. They're nature's construction workers, something? OK. All right, so let's think about all of our options.

So for one, Tim the Beaver could not do a damn thing. He could just sit around for the rest of time, and that would be perfectly fine. So a different way of putting that is that he could give up. How much coolness would Tim get from giving up? 0. That's right, kids. Giving up makes you zero cool.

So OK, so Tim the Beaver is trying to maximize. He has a lot of different options. One of them is 0, meaning he gave up. Right? Notice he could recurse. Like, x-- I guess what would be-- he could give up for one minute, and have that be x i minus 1j, or something. I forget if I'm going to do i plus 1 or i minus 1. i minus 1j.

But there's no reason. He can just give up and stop. That would just be extra recursive cost for no good reason. OK. The next thing that he could do is he could get in line for a booth. Right?

So first, let's work out what happens then. So for one, he gets coolness. Let's say that he gets into booth k. So he gets coolness Ck when he does that. And now he can recurse.

So first, you have to account for the time. So it takes tk-- that's a t-- time. Or does it? No. Because it takes an extra minute for him to get in the next line, or to get in this line. I guess this is better.

And moreover, he needs to account for j minus Wk, like that. Can he always do this? No, he needs to have this much time remaining, and he needs this much time in his bag-- or this much weight remaining in his bag. You guys can work out the inequality. But since I only have a foot here, I'll just say, if applicable.

This is a great way to lose points on your homework. But for board-writing, it's OK. So when I say applicable, I mean these two numbers had better be greater than or equal to 0. OK? And you can do this for all k, right? So in other words, he can choose to get into booth k.

And Tim has the third option, which is he can go home. OK? So what happens if he goes home? Does Tim the Beaver get cooler when he goes home? No, I'm afraid to say. But he does spend time. So how much time does he spend? h--

AUDIENCE: He should have lost coolness.

JUSTIN SOLOMON: What was that?

AUDIENCE: He should have lost coolness.

JUSTIN SOLOMON: He should have lost coolness by going home. No, home is cool, guys, especially this semester. Stay home. Stay indoors. Right. So he loses home time h and 1 to get in the next line. OK?

But he's got a devil's bargain, of sorts. He loses time, but he gains bag, bagginess. Weight is the word that I'm looking for, b, like that. OK? Again, in this case, remember that he still needs, if i is greater than h. Otherwise, he's in trouble.

By the way, I wrote this in an annoying way. A different way of saying that is i minus h minus 1 is greater than or equal to 0, which is really what's applicable in every recursive call. But this is strict greater than [INAUDIBLE]. There's another little thing that caught me up, when I was reading the answer here.

OK, and those are all the options for Tim the Beaver, yeah? Notice that every one of our options either gives up completely or decreases time. So we have our topological order, which is, again, the arrow of time always continues to move forward. I'm going to prove that rigorously by putting a check mark next to the letter T. Again, on your homework, you should write out your answers.

What is our base case for Tim? Well, how much coolness do you get, if you have no time? 0 coolness. That's how much. So we have this expression, x0j equals 0 for all j.

Incidentally, although it's perfectly fine to have this be your base case, actually, in some sense, I didn't need it, because Tim the Beaver always had the option of giving up. So you could, I guess, in this problem, have no base case, if you really wanted. It would be OK, but kind of weird. OK.

And what's our original problem? Well, he starts out with time k and weight capacity b in his bag. So it's x k, b. And then finally, we need to do our runtime analysis.

So how many sub-problems are there? Well, again, a sub-problem is basically just the number of indexes for most of our dynamic programming problems. Right? So the first index is time. The second one is bag size. This is always between 0 and b. This is always between 0 and k. So there's order kb sub-problems.

How much time does each sub-problem take? Well, notice that I have to loop over all of my different options k here. So I incur-- oh, I'm noticing k is abused in our answer to this problem. We should use k only once.

OK, so here's where I made a mistake. And I believe it's in the written solution, but I'm not going to take now. There's k, which is the total time that Tim the Beaver has, and there's the k that I use as an index here. And those are not the same. I guess I can make this k bar really fast. There you go, problem solved.

And I just noticed that, because I was doing my runtime. And it's not order k. It's the loop over all the k bars. How many k bars are there? Well, these are all the different booths, and those are n of those. So this is order n time per sub-problem, which gives me a total runtime of order kbn, which, I believe-- oh-oh, our desired runtime was order nbk. But I think we can convince ourselves that indeed, those are the same thing.

OK, so my apologies for a slight overloaded character here. But honestly, it's one of those things, if you read the answer, you probably wouldn't even notice. But now that I'm saying it out loud, I am. OK. And that solves Tim the Beaver's maximization problem. He's a very cool Beaver. Any questions about this one?

Notice that both of these problems are very similar in nature. I basically just wrote sub-problems indexed by every possible thing and then enumerated every possible solution. I think this is totally sensible. Again, I remember I had a math Professor in college that always would use this phrase-- it's important not to think here. And I think this is absolutely true for these dynamic programming problems, that somehow they look a lot more complicated than they are. Fabulous.

So problem 3, protein parsing. Ah, yeah, so this one also got me tripped up for a minute. Because the runtime they want is not the runtime of the obvious solution, but it kind of, sort of is, after a little bit of fixing. OK.

So Professor Leric Ander has a laboratory. And that laboratory processes DNA. Ta-da. OK, so let me go to my notes here, because I think they're easier to read. So a strand of DNA, as we all know, because we're MIT students, is equal to basically a strand of characters that are ACTG.

If you ask me to name what those stood for, I could make a stab at one or two of them. But I'm a failure of-- I did not have the GIR, because I went to Stanford. And this is why I'm apparently a poorly educated person, according to a person in a faculty meeting.

But in any event, so we have a strand of DNA. It's basically a long string of characters that are one of four options. I'm told that there's sometimes a fifth and a sixth option, but not too often. In this problem, there's not.

And moreover, so a strand can be cut up. So I have this big, long strand, and I'm looking for certain markers. In particular, I have a list P of markers, which are really a sequence of less than or equal to k nucleotides. By the way, this really is something that people do. String searching really is applicable to processing these DNA strands.

Obviously, . I think, in practice, these techniques have to be a lot more resilient to error. But really, a lot of these algorithms we're covering are not all that far off from how people process these giant data sets, which is pretty cool, I think. OK, so what are we going to do?

We have a string. And then we're going to make a division. So we'll call our string S and our division D, which kind of looks like d1 to dm, which are substrings that concatenate to make the full guy. So if S is our input string, then a division D is just like chopping up S into little substrings, each of which we can give a name little d here. So big D is the full division. Little d is all the little pieces.

There's that old song about going through the big D in [INAUDIBLE], Dallas. I think it's for divorce. Right, OK, so the value of a division is the number of strands. So strands are these little d's here, that are in our list P. OK?

So given S and P, what we want is the max value. And the runtime that we're budgeted is kind of a weird runtime. And this should make us a little suspicious. So the max value is big O of-- I think I wrote it slightly differently in the problem, but whatever-- distributed to k, like that. So it's k mod P plus k squared. mod S. So we have two terms here, which somehow smells funny in dynamic programming. OK.

So what are we to do? Well, what I did is I ignored our desired runtime, came up with a dynamic program, and noticed that it was a little wrong, and then fixed it. By wrong, I mean it was correct. It just wasn't fast enough. OK, so let's do version 1 here-- 1.0, that makes it more accurate, or more precise. I always confuse those two.

So here's a thing that is going to be a little bit funny. Because it's going to look like we're going to have an easy computational problem. But then it's going to turn out that it's actually too slow.

So in particular, in our S in SRTBOT, what we could do is say, xi is going to be the max value of a suffix. If you're wondering, I don't know the difference between prefix and suffix. But I wrote the word suffix in my notes and checked it at home. Si, comma, colon-- see, it's suffix, because it starts at i, and it goes to the end.

AUDIENCE: [INAUDIBLE]

JUSTIN SOLOMON: Huh?

AUDIENCE: No comma.

JUSTIN SOLOMON: In Matlab, it would be a comma. And that's a colon, and that's an i. OK. [LAUGHING] Thanks. I'm inventing my own programming language on the board as we go today. All right, so and notice this is a reasonable set of sub-problems, right? Because obviously, if I lop off some piece of my string, then the max value I can get is just the value of whatever remains.

So spiritually, this somehow feels like the right dynamic programming. And indeed, we'll see that it is. But it just requires a little bit of a list that could be to run on the right time. OK?

So let's do a recursive call. And this is actually straightforward. At least, the recursive call that you want to make is straightforward. And then we'll see that there is an equivalent formula, which is the one you'll see in the solution, which looks more complicated. But is the same thing.

So I'm going to write it as pseudocode for our recursive call, rather than as one giant formula, because I think it's easier to follow. Not pseudocode-- I know that's frowned upon in this class, a description of a set of steps for obtaining a recursive call, rather than a formula.

So in particular, we're going to initialize xi to be 0. We want to do a maximization problem. So initializing a variable to 0 is a sensible thing to do. And remember, what can we do here? So we're looking at a suffix. I could go down my list P of all the different markers, see if any of the matches the first couple of characters of my string. If it does, I get some value, and then I hop onto the next thing. Does that make sense?

Oops, I'm realizing I have a slight mistake here. Rather than initializing to 0, [LAUGHING] I actually have one additional option that I forgot to account for. Because I could just not use this character. I could put it in its own little snippet and get no value from it.

So maybe initially, I make a recursive call like that. It would be OK to initialize it to 0 and then do this-- but whatever. Notice we're already seeing the t in our SRTBOT start to stick out at us. We're going to only depend on bigger indices i. OK.

But in addition to this, this isn't enough, right? This would obviously just recurse the end of our list and do nothing. We get value if we can find a substring that's in our list. So what we could do is, for each marker in P-- remember P is the list of things that we're looking for-- OK, what could I do? If the marker matches S starting at i-- I'm going to just not even attempt to do Python-- starting at i and ending at the length of the marker, well, what could I do?

I could get $1 by matching that object. And then I have to hop forward the length of the string in my recursive call, right? So well, I could do that, or I could not do that. And I want to maximize, right? So I can keep my old value, or I could get one point by using this as my match, plus xi plus the length of the marker. OK?

This is actually, in my mind, the simplest possible dynamic program you could come up with. This is actually a totally fine dynamic program. We'll just see that the runtime isn't good enough. OK.

So does everybody agree this is a way I could solve this problem, and it would give me a correct answer? I'll do the TBOT.

AUDIENCE: [INAUDIBLE]

JUSTIN SOLOMON: Yeah, that's what we're going to do. But we're going to maybe skip some parts of SRTBOT. So in particular, what's the topological order? Notice that I always look to the right, when I make a recursive call.

What's my base case? Well, in this case, it's just a x of 0, I guess, because I'm looking forward. Oh, I'm sorry. My base case is x of the whole length of the string, which is going to return 0, right? Because if I have no string, I can't get any value out of it.

The original is x of 0, meaning that I want the whole string. And let's actually do the runtime analysis, as Jason suggests, because that's, of course, the relevant computation here. So this is T2, because it's the second T in SRTBOT.

OK, so how many sub-problems are there? Well, I mean, it almost looks like we should have a fast algorithm. Because our sub-problems are only indexed by i, right? So there's one sub-problem for basically each character in the string. So there's mod S sub-problems total.

But how much time does it take for each sub-problem? , Well let's be careful. So there's a for loop over all the markers in P. So I know that I incur at least mod P work in my sub-problem. Is that all the work that I incur? It kind of looks like it, because there's only one for loop, and there's an if.

But how do I check if strings are equal? Well, I have to iterate over the length of the string. So I incur a second cost, which is checking if two strings are equal. We know that our strings are at most-- well, I didn't actually write it down. But the problem tells us that our strings are at most length k.

So I incur another factor of k in every one of my sub-problems. And that implies that our whole algorithm, that I've outlined for you all above, takes mod S k squared-- oh, I'm sorry. That was a lie. If I actually multiply these things together, what I get is-- sorry. [LAUGHING] I need more sleep-- S P k. And that is against the rules. So that's frowny face, yeah. Because in particular, I took two big numbers, in some sense, and multiplied them together, and that's not good.

So what is a person to do? I tried to solve my problem. I came up with-- by the way, from a partial credit perspective, I think you'd be doing OK if you got to this point. OK, not great, but OK. But of course, the problem is asking you to solve this is this funny runtime, which is kP, plus k squared S.

When I see a sum like this-- and remember that there are two problems to solve. There are two strategies for solving problems in algorithms class. There's one which is useful in your everyday lives, which is to devise algorithms. There's a second, which is to psychologically diagnose your instructors. And I think that second strategy is actually pretty effective here.

I see two terms. Most of our dynamic programming things involve filling in a table, where you would expect there to be a product. So in general, I would squint at this and think, like, hm, maybe I have to do some pre-computation. Yeah? In particular, we've got to do a lot of string-matching in our problem, and maybe we can make that more efficient. Yeah? That's sort of the main question here.

So this thing is too slow, and we're trying to fix it. The way I'm going to try and fix it is to say, like, OK, well, I have mod S sub-problems. If I look at these two terms, how much work can I actually do in my sub-problem, something that looks like k squared, maybe plus this amount of pre-computation. See what I did there? OK. So let's do that.

So in particular, here are the types of queries that I'm going to have to make. There's a bunch of times in my code when-- here's a number that-- [HICCUPS] yikes, I'm falling apart. That's what I get for sprinting across campus to get here. I'm going to find a number m i, j. And it's going to be 1 if the substring S i-- oh, man-- to j is ir my list of markers P, and 0 otherwise.

By the way, why is this enough? Notice that I'm not answering, which marker? But the problem doesn't really care, right? This problem just checks if there is a marker. And if so, then I use that, yeah. So if I have this thing, that's going to somehow make this for loop a heck of a lot easier, because now I don't have to do string-matching. We'll return to that in a minute. OK.

So my question is, how can I compute this thing? And by the way, notice that I know how to compute this when the difference between i and j is bigger than k, because I know that all of my markers have length k or less. That's going to be important, because even though m is doubly indexed, I don't actually need to do that. In fact, I could even store it using less memory than that, if I wanted to, by just storing that diagonal block. OK, right.

So the other thing, which I think we saw in a previous problem session, as well, is that when we do a lot of string-matching, it often pays to put our strings into a hash table, so that they're easier to look up later. Does this string exist in this thing or not? Rather than matching every character every single time.

So maybe we do that, just for fun. And in general, when you see a string-matching problem-- and you have a list of strings, I'd suggest thinking about hash tables, just for fun and profit. So in step one here, maybe I put all the strings in P into a hash. OK?

How much time does that take? Well, I have to process every string, find its code, which is going to take more of k time. There's mod P of them, so this is k mod P time. Notice that that conveniently agrees with our first term here. So we feel like, aha, we're in good shape. We're making progress here.

And now, maybe I want to fill in this m, i, j objects here. How could I do that? Well, for one, I'm certainly going to iterate over all possible i's. OK? So let's do that. So we're going to do 2.

By the way, I'm using 1's and 2's and a's and b's and the whatever. These are just ways to denote steps of things. [LAUGHING] OK, so let's say that I just want to fill in m using a brain-dead algorithm. So I could go from 1 to the size of my strings for i.

Careful. Now I can't incur an S squared. But I know that my strings are always at mostly k. So I could do for j equals i plus 1 to i plus k. So this for loop actually incurs k time, not mod S time.

And now I can do two things. I can find the hash of the string from i to j. This is going to take order k time. And I can search for it-- ah, no-- to see if it's actually in our hash table of P. And if it is, then I set m equal to 1. Otherwise, I set m equal to 0. And these are constant time operations, at least in expectation.

So how much cost to it? And so this, you can convince yourself fills in that array m. How much time does it take? Well, I have a loop to the size of S. I have a loop of size k. I have a second loop of size k here to compute the hash.

So this whole thing is going to take order k squared mod S time, like that. Now, this conveniently-- there's chalk on the floor. And this is the second term in our runtime. So this is kosher. We can fill in m. And that's a convenient object to have around.

So the only thing that remains is to revise our R from above, to make use of the m that we have. And that's pretty straightforward. So the trick is to not lean against this thing and to actually hit the stop button. I'm learning. So now I suppose we had T2 for the second T. So for revised R, we should have R prime. [LAUGHING]

AUDIENCE: [INAUDIBLE]

JUSTIN SOLOMON: What was that?

AUDIENCE: The deriviative.

JUSTIN SOLOMON: That's right. The derivative-- yeah, we could do Christoffel symbols, like i, j prime, and semicolon k. Take 6838 if you want to learn what Christoffel symbols are. Right, so now, what is my recursive call for xi? Well, I want to maximize.

Well, what can I do? I can check every possible length of a string that could be in P, check if it is, using my array m, and get that amount of profit. So in particular, I get m. By the way, I keep using the word profit here. I am essentially using that to mean increment in every single one of our problems here. I like to think of our problems as maximizing profit, because I'm a greedy professor.

So this is m, i j, which would be 1 if I found a string there and 0 if I'm not, plus xi plus j, to account for the length here, where j is in 1 to-- well, either the length of the string, or I get to the end of-- either the maximum length of a string in P, or I get to the end of my array, like that. OK.

And this is our new recursive call. The one thing we should double-check is, what is the runtime for actually filling in x now? Well, there's still the mod S sub-problems. And now how long does it take? Well, now I just have one loop over k things. This is mod S times k. It's actually less than any of the terms that's in our runtime.

And so this is fine. This actually is kind of a funny example, where the dynamic programming part of our algorithm, once we've done all this cute pre-computation, is actually insignificant, compared to all the pre-computation that we had to do in our final runtime-- sneaky. All right, any questions about protein folding, or whatever it is that we just did? OK. So as usual, I'm talking too much.

AUDIENCE: [INAUDIBLE]?

JUSTIN SOLOMON: Yeah, which one would you prefer to cut?

AUDIENCE: [INAUDIBLE]

JUSTIN SOLOMON: I have very few preferences. OK, so one of your problems-- I would take a vote, but with our audience, there's is a high probability of a split on jury here. Right. So there's two remaining problems on the problem session. As usual, your instructor--

AUDIENCE: [INAUDIBLE]

JUSTIN SOLOMON: You can leave it there. This is another problem to learn about. As usual, I've talked too much and haven't got to the end. I get the impression that in 6006, this egg drop thing is a bit of a tradition anyway. So maybe we'll cover that problem really fast. Do they do that in section, some variation?

AUDIENCE: No.

JUSTIN SOLOMON: Not this time-- even better. OK, so yes, so maybe we'll do this egg-drop thing, mostly because the other one, I think, takes a lot of verbal setup. The other one is-- I would say, from a dynamic programming perspective, maybe not super exciting. But from an interesting problem perspective, it's kind of cool to think about. So I'd encourage you to leave it in there, and you guys can read it at home.

From a coding perspective, it's also kind of fun. I notice the solution didn't do what I would do, which would be to use the bits in the-- assume that something wasn't too tall, use the bits in an integer to store your binary variables. But that's an old hack. That's like this old hack for computing the square root of a number, that's apparently in the code for the *Doom* video game, which involves bit-shifting, and it happens to agree with square root, for some magic reason, that numerical analysts really hate.

OK, so let's do lazy egg drop instead. So that's problem 4. OK. So we're in a building. Our building has n floors and k eggs. I guess it's debatable whether the building has eggs or the residents-- but in any event, have some set of eggs. And maybe I'm in this data center or some other weird building. So I don't have heights of floors that are isotropic, but rather, each floor has a different height, which could vary.

So it's height of floor, or i. I really want to write flour, but I digress. And we're going to assume that our list is already sorted. So in other words, the fifth floor is taller than the fourth floor, and we don't have to spend n log-in time doing that. OK. Right.

So apparently, in our problem, we have an egg with a mysterious mechanical property that we are trying to recover. And all eggs, as we know, are identical. So the only difference between eggs is chicken, versus goose, versus-- I'm struggling to think of a third category of poultry-- turkey. Thank you.

But assuming that I got all of my eggs at the same Stop-N-Shop, and they all come from the same species, then they have roughly the same mechanical properties. Actually, probably the better setup is that floors are very far apart relative to the size of an egg. And if I get high enough, my egg, when I drop them on the ground, like that, breaks. Didn't break. But it could have broken.

And of course, if I drop it from an even higher height, my egg still is going to break. However, if I have a very low floor-- apparently, a very low floor. Maybe this is a house for mice. And I drop my egg, it actually stays intact. And the question, as all good scientists want to know, is, what is the highest floor in my building from which I can drop an egg and have it remain intact?

And the question is kind of a funny one. It's sort of like experimental design, in some sense. It's not asking, given this and a list of experiments, try and figure, infer, something about the eggs. But rather, it's saying, if I carefully design a sequence of floors to drop my eggs from, from which upon I drop my eggs, then what is the maximum number of experiments I need to do to triangulate in on that floor, that critical floor, above which my eggs break?

So what I'm given are the heights of the floors and a bunch of eggs. In some sense, the budget of eggs doesn't matter more than just putting a cap on the size of our problem, in some sense. What really matters is I'd like to use fewer than k eggs to determine that. Because of course, the remaining ones I'm going to use to make an omelet.

But notice that I can be a little sneaky in my experimental design, that what happens if I drop my egg from a really low floor in my building? Well, it remains intact. So I can schlep down the stairs. I can pick up my egg, and I can use it for my next experiment. And I have not paid an egg. Yeah?

So the first question you might ask is, like, well, why the heck wouldn't I just start on the first floor, drop the egg. If it's not broken, go on to the next one and then drop the egg, and so on? That would be the most egg-efficient plan. And indeed that is the case, because you'll only break at most one egg.

But you're schlepping up and down the stairs a bunch of times when you solve that, right? Every single time, you've got to go retrieve that unbroken egg. You've got to run down the stairs, pick the thing up, and then run up to the next floor.

And maybe in your optimization problem, rather than trying to minimize the number of eggs that you break, you're trying to minimize the expense on your quads. And so instead, you've skipped your leg day, or whatever, and the thing that you're trying to minimize is the sum over the heights of the drops in your experiments. So you're trying to determine the mechanical property of your egg by designing an experiment, sort of a procedure, that minimizes the number of times that you need to drop eggs. Because every time you do, you've got to run all the way back down the stairs, and go look at the pavement, and see if the egg broke or not. That's a lot of work. OK.

This is different from the classic egg-drop 6006 problem, which I encourage you guys to go seek out, in previous iterations of this course. Let me see. And so the question is, what is the minimum number of egg drops you need to do to ascertain that for any type of egg-- so I give you a mystery basket of eggs, and you have to design the experimental procedure and bound the number of this particular value here, given a budget of k eggs. OK.

And the amount of time that we have to do that is order n cubed k. Apparently, our building, we have lots of eggs and not very many floors. OK. Does our setup make some sense here? We're just trying to avoid running up and down the stairs. That's the main takeaway. OK.

So what are we going to do? SRTBOT, because that's all we know how to do, yeah? And in particular, we're going to make one observation, which is kind of handy. If I drop a floor-- ooh, if I drop an egg from a floor, in this deterministic universe, where egg mechanics are very predictable, there's only one of two things that can happen. Either the egg broke or it didn't, while I run into the board again.

So let's think about our experiment. Remember, at the end of the day, we're trying to figure out the tallest floor in my building from which I can safely drop an egg. So if I think about bracketing that height of that floor, for one thing, do I ever need a bracket that's not a continuous or a connected set of numbers? The answer is no, right?

It should never be the case that, like, oh, I think that my eggs could be on floors one-- only the prime number of floors in my building, or something. That really makes no sense, right? Because if I convince myself my egg breaks at four or five, then obviously, floors six through n, my egg also breaks. And so I always can just keep narrowing down some interval. Right?

So in particular, here's a clever S in my SRTBOT, which is to say that I'm going to say that x, i, j, e is equal to the minimum-- by the way, I'm writing this as minimum total height. So this is the minimum total times I've got to run down the stairs and check my eggs, or total height that I run down the stair-- the number of stairs I run down, assuming my stairs are 1 foot tall-- where I have e eggs left.

Notice the way that we've written the problem this time, I might as well use all of my k eggs. That doesn't cost me anything. What costs me is running up and down the stairs. And that I have floors i through j inclusive to check. So in other words, if I'm on a floor below floor i, I've convinced myself my egg won't break. But if I'm on a a floor above floor j, I'm convinced my egg will break. OK?

So what do I do? Well, remember that this is an experimental design problem. I can drop my egg from any floor f, which is in the range i to j. And of course, there's never a reason for me to drop an egg from a floor below i or above j, because we already know what happens in that case. OK?

So what happens when we do that? Well, if I drop it from floor f, I have to pay, in terms of my cost function, right? Because to pay the height of f, I've got run down the stairs. OK? But in exchange for that, I learn a little bit about my egg problem. I either get an upper or a lower bound of f, depending on whether the egg broke. OK?

So let's formalize that mathematically. So in particular, we have x i, j, e. Well, what do I get to control in my life? And what do I have to deal with? Well, what I have to deal with is the fact that I don't know what's going to happen to the egg. It might break. It might not. Right?

And the egg might be an adversarial egg-- it wants you to run up and down the stairs. And I have to account for that. But i and the egg's adversary need to choose what floor I drop it from. So remember, we saw an example. I forget what, from class, where there was a game. One guy was trying to minimize. The other was trying to maximize.

In some sense, the egg is trying to maximize the amount of work you have to do, running up and down the stairs to do your experiment. A better way to put it is that we're trying to upper bound the amount of work in your experimental procedure. And I'm trying to design a procedure that minimizes my work.

So let's say that I'm the player. So I want to minimize. And the decision that I get to make, the control that I have, is what? Well, it's what floor I choose. So let's say I choose floor f. Well, I have to go down the stairs. So that takes me hf. This was going up the stairs, is probably what incurs the hf going down is nothing. But I digress.

But now I still am not done. I've narrowed it down into one of two cases, right? Either f is my new lower bound or my new upper bound. And I have to account for both of those in my recursion and actually, the max of those two, in the sense that I need, in every possible case, that my egg drop experiment narrows down my floor to a width of 0.

So in particular, this is a mini-max problem. There's a max inside of of a min here. So either the egg broke in my experiment or it didn't. Right? So if it did broke, then, well, what happened? Well, let's see here. If the egg broke, then I got an upper bound for my floor. So my lower bound remains the same. It's i. My upper bound is f minus 1, because it broke on floor f.

Well, what happens to eggs when they break? I can't drop them from floors again. So I lost an egg. OK? So this is my egg broke. OK? Or my egg didn't break. So in that case, well, if my egg did break, now I have a lower bound. So I'm only unclear about floors f plus 1. But the upper bound didn't change. It's still j.

And how many eggs do I have? Well, my egg didn't break, so I can run down the stairs, which is going to be tiring, I've accounted for that here. But at least I can re-use my egg. So I didn't lose anything. OK?

And I get to choose. So notice-- oops-- so do you guys see why there's a max here? Essentially, I have to account for every possible scenario when I'm designing my experimental procedure. But I get to minimize, in the sense that I can choose what floor at every step. So in particular, my f here is in the range i to j, like that.

OK, and now we have our recursive formula for our egg drop that minimizes total height. So now let's finish off SRTBOT in four minutes. It's actually not too hard. So I think we'll actually make it for once by removing 20% of the problems I was supposed to cover. All right.

So first of all, what's our topological order? So this one seems kind of annoying. Because I think usually, we think of spending stuff in a lot of these dynamic programming problems. But do we actually spend an egg? Not necessarily, right? Because in this recursive call, the number of eggs I had remained the same.

So maybe that's not actually a great way to establish a topological order. But instead, what do we know? What is in science the purpose of an experiment? It's to improve our understanding of the world. In this case, our world consists only of eggs and floors of buildings.

And in particular, once I drop that egg, I learn something about my building. And I narrowed down the range of floors that are uncertain for me. So in particular, I know that x i, j, e only depends on x, I guess, i prime, j prime, e prime, with what? Well, my sub-problems, I always have a smaller range of flaws than I did before.

So in particular, j prime minus i prime is going to be smaller strictly than j minus i. And that'll give me my topological order. Cool. And so that's actually, I think, the sort of annoying part, other than working out this mini-max expression here. The remaining things are not so hard.

So what are our base cases? Well, let's say I have 0 eggs left. But I still have a set of uncertain floors. That's bad news. Yeah? Yeah, so that should be infinity. And the reason is, of course, I'm going to take the min here. Right? And so obviously, I should never choose infinity as a min.

So in other words, I should never choose an option for a floor that could possibly lead me to an uncertain scenario, when I run out of eggs, yeah? In addition to that, there's another base case here. This is I've got no eggs left, but some floors to check, is the second one, i minus 1e. So in this case, I've got e eggs left, and I'm done, right? I've narrowed it down to the bounds.

Incidentally, the way I wrote it in terms of inclusive, versus exclusive, might be a little fishy. Over here, you guys shouldn't be off by 1, like your instructor often is. But in any event, here, you're 0, right? There's no more floors left to check. You've narrowed it down to a range of 1. Again, something that I'm out of time so I'm not going to check carefully, is if my bounds are inclusive, should that be i, j minus 1-- i minus 1 or just i, i? But I think you guys are all smart enough to work that out at home,.

Which would my original case be, well, now I have all the floors to check, and all my eggs in my metaphorical basket here. So I have floors 1 to n here. And the problem tells me I have k eggs when I start. And then finally, I need to do my sub-problems here.

I think you can actually simplify the argument that's written down a tiny bit. And just again, look at your sub-problems. They're indexed by three numbers, I'm going to do a really conservative estimate. I think the problem actually works out a better estimate, but then asymptotically, it's the same.

What's our bound on the first and second index? Well, they're both just the index of floors, which go between 0 and n. Yeah? Obviously, you could do better than that, because the lower floor is always less than the upper floor, which is what the problem accounts for. But if I'm being lazy, then, well, there's n squared sub-problems to account for the two floors.

And the third index is your eggs, which you have at most k of. OK, how much work do we have per sub-problem? Well, let's see here. There's a for loop over f. f Is over floors. Again, if I'm going to be really conservative, well, there's at most n floors total in my building. And so that leads us to a runtime and n cubed k, which is what we wanted, at the end of the day. OK?

This should be a big O here, because I think technically, this is n plus 1, to account for floor zero. OK. And that solves our egg-drop experiment. I think this is a nice one. And I think, in my mind, actually, in terms of dynamic programming, this is one of the harder things to get right, which are these mini-max games.

I'd have to think about it, which in my negative two minutes, I'm not going to have time to do. I think in lecture, the way that we solve minimize problem was we separated out the min and the max, and we thought of there being two dynamic programming problems that are interacting with each other. You could probably write this one in that form, as well, I guess, just by pulling this term out and thinking of it as a different array.

But this form is perfectly fine, too. Either one's all right. But in my mind, these are the hardest things to get right in dynamic programming. So I would choose whichever one jives in your brain.

So in your thing, should we choose to leave it alone? There is a fifth problem here, which, as usual, I haven't managed to get to, where you're building walls by placing tiles. This is an interesting one, because your runtime is exponential. But the problem tells you that that's allowed.

But there's some exponential things which are OK and some that are not. Essentially, what you don't want is the product of two giant exponentials. You'd like to just get it down to one.

AUDIENCE: It's basically saying it's going to be polynomial [INAUDIBLE] small [INAUDIBLE].

JUSTIN SOLOMON: That's right. Or it's polynomial in everything except for the things it's exponential in. And moreover, the things it's exponential in are small. And the problem says that. So I encourage you guys to take a look. Because it really does take some time to logic through it. But the setup for that problem is, I think, longer than my glacially slow board-writing can handle. But with that, we'll call it for the day.