PROFESSOR: All right. So Lecture 7 was about many things-- universal foldings by box pleating, maze folding, and then lots of NP hardness stuff. So we're going to go through things in that order. I thought I'd start by showing some old history of box pleating, where it comes from.

The first design is by this guy Raymond McLain, 1967, design called the Mooser's train. And you can see all the creases are horizontal, vertical, and diagonal, 45 degrees. This is the original handwritten thing, and this is apparently before the time of origami books, for the most part.

So people would just photocopy this, and hand it around, just kind distribute it out through the origami community. And kind of spawned a revolution origami design, because it was the first model to make sort of a very complicated multi-object piece out of one piece of paper. Happened to be a rectangle of paper, but a pretty cool design . And it really got people thinking about.

And the basic, original principal in box pleating was to make boxes, and then find ways to attach them together. And it's fairly powerful and believed to be universal by origamists for a long time. And we proved it with that cube gadget. Now, the designs that come out of the cube gadget are not especially efficient.

I've got some ways to optimize it, as you saw, but at least it proves everything is possible that you could make out of cubes, you can make out of box pleating. Another kind of influential design is this black force cuckoo clock. For a long time, one of the most complicated origamis out there. It was designed by Robert Lang in the '80s, and there's a few of them in existence.

It has lot of detail-- it has a clock, that's correct twice a day, and it's he modeled it after a real cuckoo clock he saw in the Black Forest. It's got a zillion creases, especially in this long rectangle of paper. And there's diagrams for it in *Origami Design Secrets*, if you want to make your own. And it's also based on this box pleating idea, and sort of in the early days of the tree theory, and so on.

So really, before that kicked off, box pleating was around. And people still do basic box pleating, although now there's the fusion of the tree method with box pleating. There's a lot of designs based on that, because they're easy to fold from angular perspective, easy to find where the creases are, anyway, that's box pleating.

So next, we go on to-- I think it's an open problem. Yeah, this is a cool open problem. For some reason, never thought about it. I'm sure briefly thought about it, but never worked on it.

In the same way we get universal hinge patterns, box pleating can make any poly cube, what if you want to make, out of the simple cube gadget, can we design an analogous cube gadget for tetrahedra, octahedra, might be nice, regular octahedra tile space, out of something like a triangular grid, maybe with 30 degree lines.

That's a natural question-- open. Maybe we'll work on it sometime, but I think it'd be neat. You just need a nice gadget, and then we know how to compose them by induction just like in lecture seven. So it could be a fun thing to attack.

Next, we go to maze folding. So I thought I'd show-- we've seen this maze fold-- the 6.849. So this is folded by Jenny and Eli based on the algorithm that you saw, and you might play with it on your problem set three. Design your own maze. You can click any pattern you want. Make whatever letters you want, or any other cool orthogonal pattern.

Marty and I have made a bunch of different print designs based around this idea of taking not just the crease pattern you get for folding-- here we're folding the word yes in three dimensions-- but we've shaded the original piece of paper in this pattern. So that it looks like nothing on here, but when you fold it, the shading comes together to spell no. So it's kind of ambiguity, or the shadow of the yes is no, and so on.

And Jenny and Eli folded that as well, and here it is. The first folding. But our idea is not necessarily to fold them. I mean, it's cool that they can be folded. That's neat conceptually, but we also like to design ones that really should not be folded. Next

one here is science and art, so you fold the science in three dimensions and the art is in the background lurking there. I imagine this will never be folded. It'd be rather painful.

Here's another very complicated one. Here we're playing around with putting little shaded regions in these squares. These are just the squares that end up mapping to the flat parts over here. And so you get-- we took an image and spliced it up into thousands of pieces.

So you get-- this is a photograph of Martin Gardner, who died a couple years ago sadly. We never met him, but he was the father of recreational mathematics among other things. Also magician-- lots of cool things. So this is in tribute to him.

And this is our latest design in this spirit. We're trying to be a little more artistic and just be about the crease pattern, not necessarily the folding. So the 3-D thing that this folds into is not shown here, and in theory all these lines go off to infinity. But this spells the word glass. We made it when we were at the Pilchuck Glass School this summer.

And it's now on t-shirts of many glass blowers around the world. How many of them know exactly what it folds into, I don't know. But that's pretty cool. So we were working on glass folding, so that seemed like a natural thing to do. So that is maze folding and see some designs. How you can use it for cool things.

The next topic is NP-hardness, and that will be what we spend most of today on. And so, first kind of general question is what does it all mean? What does NP-hardness really tell you? Can you give me like a specific instance that's hard, and-- this is more for people who haven't seen MP-hardness before-- the answer is no.

NP-hardness is kind of a conceptual, philosophical thing. No specific problem is ever NP-hard. It's all about whole families of problems being hard. And really it's about measuring the growth of problem complexity with problem size.

So in this case, it's how much running time do you need to solve your problem as a function of the problem size. And we're interested in whether that grows only

polynomially or it grows exponentially, and NP-hardness probably implies exponential growth.

A nice example of this is chess. So eight-by-eight chess-- people have spent many years trying to solve it and do it perfectly, and to be the best player. But from a theoretical, computer science perspective, chess in its original form, is trivial because it just has a constant number of states. You could just enumerate them all in constant time. You know who wins in chess-- probably white.

But it's not very satisfying, but the more interesting thing to say is to study chess as it grows, and the natural way to make it grow is with an n-by-n board. So with an n-by-n board, if I give you an arrangement of chess pieces and I want to know who wins, that you can prove is something even stronger than NP-hardness called X-hardness. So there you can actually prove you need exponential time to solve n-by-n chess.

And that gives you a sense the chess really is hard. There's not going to be a good algorithm that magically will solve it for eight-by-eight, because in general as board size increases, you know it has to grow exponentially, and probably eight-by-eight is big enough. That there's no good algorithm to do it.

You can't prove that there's no good algorithm to do eight-by-eight, but you can prove there's no good algorithms to do n-by-n. So that's the limitation of theoretical computer science and of NP-hardness. So you take any particular crease pattern. You want to know whether it folds flat. You might be able to do that by exhaustive search and exponential time might be OK, but you know that as the crease patterns get big, you're totally screwed.

So, next question about hardness is can we do it with a little less hand waving? So in a little bit more detail, because I covered a lot of proofs. So I'm going to look at two of them. The simple full hardness-- which is from partition-- and the crease pattern flat foldability which is from not all equal SAT-- 3SAT-- and I'll give some more details.

So let's start by going through this proof. Remember, in general with an NP-hardness reduction, we want to take some known hard problem and convert it into our problem, because that proves that our problem is at least as hard as the original one. So in this case, the known hard problem is I give you n integers-- a1, a2, up to an. And I want to know, can I split them into two groups so that the two groups have equal sum? And here we're mapping that into the simple fold problem, which was given a crease pattern can you fold it. Can you follow all the creases flat by simple folds?

We're mapping it into these lengths-- these lengths are the AIs between the different legs of the staircase. And then there's this extra stuff. This is a super long length-- length l-- and this is a doubly super long length-- 2l. And then there's this frame, which has height 2l, and this part is aligned right at the midway.

And so the idea was if there's a partition-- if there's a partition of the AIs into two groups-- call them Group A and Group B-- then I'm going to fold some of the creases. The ones between any two guys of different groups. Because folding here corresponds to switching directions.

So whenever I fold, I'm kind of switching which group I'm in, and then Group A will be all the up directions and Group B will be all the down directions. So maybe both of these guys are in Group B so they both go down, and this is a Group A guy, so we go up, and then down, and then up. And if we get them-- A and B to be balanced-- then we'll end up right where we started. So we'll end up right at the middle point here.

And, in that case, there's these two more folds-- this vertical fold and that vertical fold. And then if you-- sorry. Yeah, I don't have it drawn here. When you do one more fold here, this will fall into the frame, and if you got this to stay-- if you got this point basically to be aligned with here, then going up L, and then back down to L will not collide with this box. It just barely fits, and so you fold that in. Then you fold this vertical line and you fold it back out.

So the goal was just to get these two folds made. Once they're made and it's back

out here, you can finish all folds that you didn't fold before. And what we're checking here is that during all this motion, you don't get collision because simple folds are not allowed to collide during the motion. So if this, for example, went too far down or too far up and it collided with the frame, then you wouldn't be able to make the second vertical fold here because you'd have to bring the frame with you, and the frame's not supposed to move.

So in particular, there's this question. Seems like we're not making all the folds, and that's because I didn't say the last part. After you fold it in here and avoid collision, you fold it back out with the second of these two folds. Then you can finish all the folds. Indeed the goal is to fold everything, but in order to avoid collision, when I fold this, this thing has to be nice and compact and fit within this 2L boundary. Then it goes through.

So what we really need to show though at the top level is given any set of AIs, we can convert it into an equivalent simple fold problem. There's actually two things we need to check for equivalence. We need to check that if there's a partition, then there's a way to fold it. We also need to check the reverse, which is if there's a way to fold it, then there's a partition.

But both of these kind of follow from this argument. In order to-- when we make this fold, this guy's got to-- or whichever of these two is first. They're kind of equivalent. If we collide, we won't be able to finish the folding. That's the claim.

You might have done all the folds over here, but they'll be the second of these two, and so when you do the first, you'll either get stuck or you'll be able to proceed. If you were able to proceed, you can read off from the ups and downs here what the partition was. So you can convert in either direction using this reduction. Any more questions about the simple folds one?

So, of course, one of the easier proofs, the crease pattern flatfold ability is definitely on the more complicated side, and this was the overview of the proof. Basically there were lots of little local gadgets. There's the wire, which communicates truthiness or falsity as we like to say. Or things like-- the main gadget is the one at

the top and not all equal clause, which forces among the three wires coming in.

They can't all have the same value. So there could be two true, one false. Or two false, one true. But not all true and not all false.

And then we needed some auxiliary gadgets for turning, duplicating, and crossing over. Those were fairly straightforward. The heart is a wire, and a not all equal clause. And so the idea was we were given a formula-- in this case, it's a bunch of triples, and each of the triples should be not all equal. And there's like n variables-- there's a lot of different clauses on those variables. A lot of not all equal constraints on those clauses. We want to represent that by the flatfold ability of the crease pattern.

So we basically make all the variables off the left edge of the paper here. We make lots of copies of them by zigzagging, and then we make lots of clauses at the top-- only drawn two up there-- and then in our input we're given a bunch of triples of variables that should belong to a common clause, and we just route these signals to go to that clause. And then the clauses constrain the values of those variables to be not all equal in the appropriate triples.

And the splitters down here enforce all the different copies of that variable to be the same. And so any flat folding will require these guys are consistently assigned one truth value throughout the construction, and those clauses will force not all equal things to be seen in pairs. So if there's a flat folding of this, you can read off on the mountain valley assignment here which one based on whether it's left valley, right mountain, or left mountain, right valley. You can read of what the truth assignment is that satisfies all the clauses.

That was one direction from flat foldability to satisfiability of the not all equal triples in the reverse direction. If there's a not all equal assignment of triples, you need to verify that this thing actually does fold flat. I didn't detail that, but basically to do that, you have to prove that each of the gadgets works exactly as desired. So you could really fold it flat if these have equal value, for example, and this has a negated value. You could actually always fold the not all equal cause when it's satisfied.

And then once you know that each of these gadgets-- because the gadgets are very small-- once you know that each of them folds correctly, and they have a sort of comparable interface because these things are just extending through, you could basically paste glue together all of those folded states. So as long as you verify the gadgets work, the whole thing will work provided there is a satisfying assignment.

So that's-- yeah, question?

AUDIENCE: Can you actually go in the other direction properly in the same cell? Can you map any flat foldability [INAUDIBLE]?

PROFESSOR: OK, can you map any flat foldability of a crease pattern problem to SAT? That doesn't follow from this proof, but I think it should be true. Uh, let's see-- I think so. So if I give you a crease pattern, I want to know whether it folds flat. As mentioned in lecture, you can determine where all the facets lie in 2D. And challenges the stacking order, which implies the mountain valley assignment.

So I think you could make a variable in a SAT problem for this piece-- this little polygon lives in the stacking level. You have to figure out how many levels you need, but at most 10 of them, I guess. And then write down the constraints between different polygons if there are no crossings. There's a paper by Jacques Justin that does that explicitly.

So that should give you a way to reduce to SAT, which gives you an algorithm-- an exponential time algorithm to solve it. In general, I'm pretty sure this problem is in NP, which implies there's at least an algorithm for it. But I think in particular, you can reduce it to SAT. So that's a good question.

So pretty much every problem we'll talk about at least with origami has a finite algorithm to solve it, but it's a very slow algorithm in general. Of course, there's a lot of good SAT solvers out there, and I don't know if people have actually tried to do that. I know there are some-- there's a software called ORIPA, which in Japan some versions try to find a valid folding-- a valid folded state by some kind of careful, exhaustive search. I don't think they use SAT solvers though. You might get more

mileage out of SAT solvers.

Other questions about this proof? There's one explicit question about the reflector gadget. This probably worth talking about because these arrows are maybe not so obvious what they mean. So the question is what is true and what is false. It's kind of philosophy, but in this case these arrows are explicitly drawn on this diagram. If you look closely, there's an arrow this way, then this way, then this way. And that's defining what we mean by the orientation of a wire, and then if it's a valley on the left relative to the arrow, that's true. If it's mountain on the left relative to the arrow, that's false.

And that's why this is drawn this way assuming I got it right. Here it's valley on the left relative to the arrow. Here it's mountain on the left relative to the arrow. So notice these are pointing in different directions.

So that can get a little confusing, but it's kind of what we want in that proof because we really want to route that thing in this direction. We don't care about what things look like relative to the center here. Because the center-- the gadget we are looking at keeps changing. So we want to route this as a single wire. And this one I already talked about. You need to prove both directions-- that flat foldability implies not all equal satisfiability, and the reverse.

OK, we move on. So now we get to some new material. So I mentioned in the lecture that there are two hardness proofs. The one we've been talking about is given a crease pattern, does it fold flat? That's strongly NP-hard.

And the other is if I give you something-- I might even tell you it's flat foldable. That's kind of not too relevant. But I give you mountains and valleys, and I want to know-- I want to find a valid folded state of that, or I want to decide whether the mountain valley pattern is valid.

These are strongly NP-hard hard, and this is a different proof. It uses completely different gadgets. So I thought I'd show you some of those gadgets. Sort of the key ones.

So the first part is the wire, and I think I made one of these. Maybe I didn't. No wire. Well, this is the end of a wire-- er, no, that's not the end of a wire. Here's an end of a wire.

So the idea here is we've got a fixed mountain valley pattern now because that's no longer free for the folder to choose. And you've got two tabs, and they could stack one way or the other. In this case, they can stacked like this, or they can stack like that.

So I've got a choice in stacking. That changes the folded state. That's the choice that the folder is going to make, but in either case the mountain valley assignment is the same.

So that's how we're going to communicate true and false, and I haven't provided orientation. Should have drawn an arrow there.

Next gadget is this tab gadget, and this is just a tool for building weird shapes, and it's kind of similar to some of the things you've seen in the checkerboard folding. So our crease pattern looks like this, and fold it flat. And there's two ways to fold it.

Actually, a few different folded states. But in particular, there's a state like this where there's this individually manipulatable tab, but there's no crease here. So it can't go backwards. The crease is in here. So it's got a fall down like that.

OK, so what this reduction does is, ahead of time basically, force a lot of these foldings. And so we start with a square paper. We end up with a smaller square paper with tabs sticking out in various directions of various places. So this is much messier and it'd be harder to draw the whole diagram of what your crease pattern looks like, but in theory you can do this.

That's also kind of like the box pleading method where you just make a cube ahead of time, and it's like you had a square originally. There just happens to be a cube sitting here, and then you do other folding.

And the angles in this crease pattern-- these are not 45 degrees. They're at a bit of

a weird angle to basically force this to happen first. It can't interact with any of the other gadgets that I show you. Because you've got to get rid of those angles in the beginning.

Then, the main gadget here is the not all equal clause, and that's what I have folded here. So it's a little crazy, but basically you've got three pleats-- three of these double pleats coming together. These guys could stack-- this one on top, this one on top. Now we're the same for all three of them, and then you've got this little twist like thing in the center, and it's really hard to draw what the folded state looks like. This is what it looks like within shadow pattern with transparency.

Here's the real thing. And also, there's these yellow tabs attached at very specific locations. And this is a little hard to imagine. I encourage you to fold one of these in order to really see what's going on.

But as I said before, you've got-- each of these guys can be independently stacked one way or the other. There's that one or this one can stack this way or that way for all three, but we want to forbid the case where they're all stacked the same way. And the arrows here are all pointing towards the gadget. So we want to forbid the case when they're all-- when it's rotationally symmetric.

Gadget is rotationally symmetric, but if you choose the layer order rotationally symmetric, you're in trouble. So let me make it rotationally symmetric. That would be like this. So when it's rotationally symmetric, you see-- basically all the panels you see-- there's three panels. This one, this one, and this one.

Each of them has a tab sticking out, and basically these tabs have to also be cyclically ordered, but it's not possible-- this is hard to do. It's like this. You see they're kind of colliding here. Because if you look at each of the tabs in projection, it collides with where the tab is attached.

So you basically-- if you went-- so someone has to be the lowest tab, and that tab will intersect the other two tabs actually where that tab is attached. So it'll penetrate and you can see they're not very happy to stack here.

But if I change the layer order in any way, our lowest tab is actually happy to go behind other layers, and then you can just stack them, and they barely fit. So it's a little tricky to get all the details here to work, but I think this particular geometry-- and the reason I drew this in this exact way is to make sure yeah, everything works.

You do not penetrate here, which would be a problem. You do not penetrate this crease, which would be a problem. So the tab just fits in between here and here, but it does cross. If you look at this tab, it's attached along this edge as drawn up there. And this tab here will penetrate this part of that edge, and it will penetrate this part of this edge which is where this tab is attached.

So they cause trouble in this cyclic situation, but when it's a cyclic like in this picture, you can put that bottom tab way down here and avoid any collision. Then you can basically break the cyclic condition, and you get to stack them in some linear order.

So it's a little hard to see without physically manipulating it. Feel free to come and play with that after. That's their proof. Now there's also splitters, and turn gadgets, and crossovers, but I'll leave it at that. Those are more similar. This is the heart of what's going on in this proof. So it's a completely separate proof from the one that we saw before.

Any questions about that? All right--

AUDIENCE: Here.

PROFESSOR: Yeah?

AUDIENCE: I don't understand why the tabs fold.

PROFESSOR: How do the tabs fold?

AUDIENCE: No, why do we need tabs if everything folds?

PROFESSOR: So the tabs are being used as a device to build gadgets. So the tabs serve no purpose by themselves. But this gadget involves first folding three tabs here, which are actually pointing-- they're pointing in that direction, I believe. Then you add this

crease pattern afterwards. So originally there are tabs here, which means there's a bunch of creases here.

After you fold them, then you lay on this crease pattern. So if you unfolded it, you get some much more complicated crease pattern. Which is when I made that, I just taped the tabs on. But in reality, first you fold the tabs so they exist there, then you fold this on top. So the fold crease pattern would be quite complicated. It's going to have tabs here, tabs here, tabs there, and then this reflected a few times.

And so then when you fold it-- the fold gadget is this thing with the tabs. The tabs are just like a stepping stone toward making this gadget. The other gadgets use more tabs.

Obvious open problem here is to make a simpler reduction. Shouldn't have to be this complicated, and I've heard Tom Hall talk about different approaches for making simpler NP-hardness proofs, but we're not there yet. Other questions?

AUDIENCE: So these proofs don't imply anything about the [INAUDIBLE]? They can all be very easily [INAUDIBLE].

PROFESSOR: So this proof implies these kinds of tessellation style crease patterns are hard to fold, which implies that in general crease patterns are hard to fold. But you could look at some other specific pattern. For example, it's the next topic. You could look at map folding where you have horizontal and vertical creases in a rectangular sheet of paper. Maybe each of these squares is-- or each of these cells is a unit square, let's say.

And I give you a mountain valley assignment on that. Some of these are mountains. Some of them are valleys. Whatever. And I want to know does this fold flat? That we don't know the complexity of. Could be solvable by polynomial time algorithm. Could be NP-hard.

And this is actually mentioned in lecture two notes, but not actually orally. And so I wanted to get back to this. This is from lecture two notes. 2D map folding, Jack

Edmonds poses this problem, can you characterize flat foldable mountain valley pattern. Is there an algorithm or is it NP-hard? And even the 2-by-n problem was open when I wrote these notes in 2010, but it has since been solved. So I thought I'd tell you a little bit about that solution.

So yeah, there can be special cases of crease patterns that are easy to solve. We just know in general they are hard. So 2-by-n is this kind of picture. So this is what I call 2, counting cells.

This is based on a class project in this class from two years ago by Eric Lew and Tom Morgan, but the paper got finished this year. So it's based on a series of reductions. On the one hand, we're going to start with something called-- OK, so we have map holding on the left.

We're going to convert map folding into a different presentation, which is NEWS labeling. We're going to convert that into something called a top edge view. We're going to convert that into a ray diagram. And the last part, which I won't talk too much about, is we convert that into something called a hidden tree problem.

Each of these steps is fairly straightforward, but there are obviously a lot of them. But in the end, you get a polynomial time algorithm to solve this, which implies you get a polynomial time algorithm to solve this. And I'll talk mostly about these reductions.

It's the same idea in NP-hardness. We use reductions from a hard problem to our problem. In this case, we're doing the reverse. We're reducing our problem to an easier problem, so that by the end we get something that's so easy we know how to solve it. The reductions are useful both for positive results and negative results.

So let me show you visually some-- oh, before we get to the proof, I wanted to mention this puzzle which you probably folded in problem set one. It's a map folding problem. It's a 3-by-3 map, and so we didn't have any good algorithms to solve them. So we used a not so good algorithm-- namely exhaustive enumeration.

We enumerated all possible folded states of a 3-by-3 map. We drew them out

graphically. There's a scroll bar here, so you don't see them all. But we put them in a giant-- I forget, there's 1,000 folded states or so, so it's like 100-by-- whatever-- 10 array. And for each of them-- so here's where you're designing your pattern.

So you could make holes. You could put in patterns. This is sort of an early design where it would spell MIT. The hole would show the I on another side, so it's equivalent to the puzzle you solve.

And this is what the top looks like. This is what the bottom looks like. So you can hover over these and change the letters that appeared, and then it would show you what every single state would look like, and it would color it in yellow if one side looked like MIT, and it would color it in red if both sides look like MIT. And the point was to verify there was exactly one solution for that puzzle.

And so we kept-- we would fold things to make it a tricky fold. Then we put into the software and label things accordingly, then put into the software and see is there unique folding? And then we could add in extra misleading clues here, and see did they accidentally make an extra solution?

So we could add in extra stuff to pack in. As much irrelevant stuff as we could, and still make it unique so it's more challenging. That's a nice use of exponential algorithms to design puzzles.

All right, but back to this proof. So I have here a much simpler map than this one. This is what we call the NEW map. N-E-W.

So the labeling here is very simple. If you look at this crease pattern, by [INAUDIBLE], there's got to be three of one type and one of the other type. So just label each vertex by which way is unique. So from here, the North direction is the uniquely label-- it's the mountain among three valleys. And then this vertex-- the East label-- is the mountain among three valleys, and then the next one is the West label among three valleys.

Of course it doesn't have to be like that. It could, for example, you can have three mountains at a vertex, but then the label of this guy's going to be W. It's going to be

the one that's unique among the other three.

So if you fold this map, it looks like this. I have one here, but it's a little hard to see. We've got lots of back and forth on the top edge. Some orienting it so that the-- we've got 2 by n maps.

So there's two critical features. There's the center line. I'll call that the spine of the map. There's the top side and the bottom side. When you fold that-- this is just a sequence of simple folds. You see at this last fold, the top side and the bottom side come together, and the spine is-- stays together.

So when I orient it like this as in the picture, the top side is where originally the top and the bottom of the map all come to here because of that spine fold, and the spine folds are all down here. You can't see them here, but there's you can see them at the bottom there.

There's some kind of connections on the bottom, but there's this nice relatively one-dimensional picture up here. So we're going to draw that. This is the top edge view. Got clipped a little bit, but there's one more segment at the top. So that is just this. See it?

Now there's also these blue lines. The blue lines correspond to what's going on at the bottom from the spine. Because if you think about it, you've got the-- in a 2-by-n map-- you've got the top panels and these edges making the top edge. And you've also got the bottom panels and these edges making the bottom edge.

They come in pairs. These guys are paired up. These guys are paired up, and so on down the line because they are joined by a spine edge. Those spine edges correspond to the things on the bottom of the folding, and they need to not cross each other. And that's what's illustrated by the blue connections.

So the very top edge is paired with the very bottom edge. These two-- this panel here and the back panel here-- are connected on the bottom. And that-- I'm sorry, that's not true. This one is paired with this one. The top one is paired with the

second one. So it's actually these two are connected on the bottom. That's this little connection here, and then these guys-- this wraps around everything. That's that connection.

But what you can't see it in this folding-- because it's hidden inside-- is that within this connection on the bottom, there's this connection on the bottom. And within that one, there's this connection. Now this is OK. Might look like they're crossing because they're overlapping.

But as long as they are nested-- as long as you don't start from inside here and go to outside. That would be a crossing. You start inside, you should end inside like in all these pictures. If you start outside, you should end outside.

So that's why-- these guys are disjoint, so that's OK. This is nested in that, so it's OK. Kind of like balance parentheses. And so that's what it makes. A top valid top edge view. What's nice about this is it's a one-dimensional diagram.

What's not nice is it has all these crossings because there's the blue stuff on top of the black stuff. So it's a little hard to find-- we're trying to find a valid folding state. We're trying to find one of these.

We're not restricting ourselves to simple folds here yet. Simple map folding we already solved in lecture two. So this is all-- I should've said non-simple folds. That's what makes it hard, and that's what's still open for 3-by-n maps if you don't know how to do it.

But for 2-by-n maps, we've made two steps here. We have another view, which is the ray diagram, and this is really specific to 2-by-n. And it lets us reduce to a tree problem.

So before we get there, let's talk in general about the top edge view. You can actually convert from the North, East, West, South labeling to a top edge view in the following way. Whenever-- and you can kind of see what's going on. So first we have an n.

So we've got these two guys, which are just paired together happily. That's the beginning of the map. Then we turn left, and we've got this guy paired with this guy. And so, in general, whenever you have a North, you turn left with your two guys. So that's what happens.

Then we hit an East in the map, and what happens here is this guy turns right, but this one turns left. This is what you might call an inside turn. It's like an inside reverse fold in origami. And in general, whenever you have an East, you turn inside like that. So this guy's paired with that one, and now this guy's paired with that one. They both turn in. And then it's symmetric.

Next one is a w, which is the opposite of an e, which means you turn out here. And in general, whenever you have a West, you turn out. And a South, you turn right. So those are the four possible things you could imagine when you have a pair of things turning, and those exactly correspond to N, E, W, and S.

So it's clear what you need to do in this situation, but there's flexibility, right? For example, this guy-- when you turn out-- here I turned out to the very next layer, but this could have turned out to go up here. Would that be OK? Ah, no.

If this one turned out to go up here, that edge up there would be paired with this one down here. So there'd be a blue line going like that, and that would be a crossing situation because that blue line would cross this one. Because it starts inside and it goes to outside.

So it's a little tricky. This definitely gives you some of the layer constraints, but it doesn't check for crossings. Which is the big challenge in this problem.

So we're going to simplify this diagram. So just think about top edge views. Forget about map folding now. You can forget about the previous layers, and we are now focusing on this reduction.

So we've got top edge view converting to ray diagrams. You can see it's much simpler. This is the ray diagram of that picture. And essentially-- there's two ways to think about what we're doing. Instead of having two edges, and we track two edges

turning, we just want one edge. That's one thing we're doing.

You can think of that as tracking the spine instead of the top and the bottom sides. But it's a little-- ray diagrams are going to get confusing. I'll just tell you that right now. Every time I look at it, I understand it a little bit more, but it's not going to be obviously that this is equivalent to this.

I'll just tell you-- here's an alternate way to think about it. Basically, the vertical-- the y direction here is going to be nesting depth. So whenever we go in like this, we go deeper. So for whatever reason, I flipped the labeling, but it is indeed East. When you go in, you go down.

I should have just moved these two figures so they correspond, but when you go East you go down a layer because you're nesting deeper. And when you go West, you're going out, so you go up a layer in this new weird ray diagram view. And North and South turn out to not turnaround in this view. We're just going to fire a ray downward, and there's either a North ray-- a blue one-- or a South ray-- a red one.

And so if you follow this diagram, it's N-E-W. So first we have an N, which means we shoot a ray down. Then we have an E, which means we turn downwards. And then we have a W, which means we turn to go up a layer. And I haven't told you what the constraints are here, but I claim this is a valid folding just like that one.

And it's obviously a lot easier to draw. It's a much slower complexity. But the cool thing is you no longer have these crossing parts. So let me show you-- first, let me tell you what the constraints are.

So in general, you're going to have lots of downward rays. This is a North. This is a South. A North, a South. So if you had like N, S, N, S, in your pattern, it's just going to be a straight segment with downward rays of different colors.

First constraint is whenever you have a an N ray-- a North ray-- it has to hit another North ray or just go off to infinity. And the same-- South rays can only hit South rays or go off to infinity. So they line up in this way, but you can stretch the x-coordinate to do whatever you want.

This corresponds to basically jumping over a bunch of layers. When you turn in your folded state, you can skip a bunch of layers and just go up here. So that corresponds to jumping over a bunch of folds here.

But in particular we get these things called constrained segments. Constrained segments are portions of this black spine between two rays that below it only see black. OK, so for example, this one is unconstrained, because below it, you can see off to infinity. So that's unconstrained. We don't care about that one.

But among unconstrained segments-- these are two constrained segments-- we want that all of the rays that they see below them-- so here I see one red ray, one blue ray-- I want the number of reds and blues to be equal. So this is valid because there's one red, one blue. This one is invalid because there's two blue and one red.

This turns out to be the right constraint. Essentially this is saying however much you spiral, you should spiral. That's the intuition. So here you're spiraling twice, unspiraling once. So you're trapped inside, and so these folds cannot actually come together. The rays coming together correspond to a nice nesting between two North folds.

So it's not obvious, but this turns out to be the only constraints you have. The black should not cross itself. Constraint segments should be valid in this way, and blue should hit blue and red should hit red. So that's not obvious, but it's true, and then you can take very complicated diagrams like this and simplify them into these nice pictures.

So these are two different foldings of the same pattern, which I guess is N-E-N-- I can see that easily here. You could have also extracted it from here. There's two foldings because these two layers could go here in the middle. Or they could be pushed up to fit in here, and then we get this picture.

And in the ray diagram-- you can see this pretty easily-- either you have these n rays just both go off to infinity, and this is an unconstrained segment so we don't

care about that there's only one North ray below. Or, you can extend the x, because I can stretch the x parts however I want and make these blue segments align with each other. And that corresponds to this folding.

So really, all folded states of the original thing are represented by the ray diagram. Of course, if I had any W, this would be valid, but this would not be that. N-E-S-- blast from the past. If I had N-E-S, then this would be blue, this would be red, and so this would be valid. But this would be invalid, and so on.

So it's obviously a lot easier conceptually, but you can also-- and here's a more complicated algorithm. You take a really crazy map. This notation, by the way, was introduced by Jacques Justin like late 1980s. You may remember his name from Kawasaki-Justin theorem.

So at the same time he was looking at map folding and he came up with this notation, the ray diagram was introduced in the open problem session in this class two years ago-- or actually five years ago if I recall correctly-- by David Charlton and Yoyo Zhou. And then it was picked up again two years ago, and finally we took this ray diagram and came up with an algorithm.

So how does the algorithm work? Here's a ray diagram-- the black, and the red, and the blue. We observe that the blank space here between the spine stuff-- between all the black-- is kind of a tree structure. You've got-- and that's what's drawn in cyan with the red dots. So if you draw a node every time you turn around on the outside.

Then you say, well, I can get from this node to this one, so I'm going to draw a segment there. And I can get from this node to this one without crossing any black, so I'll draw a segment there. That's sort of it on that side. There's nothing else down here. That sort of doesn't count.

But here's there's another dot, and I can get from this one to there by going out here without crossing black. Here to here, here to over here without crossing black, and so on. So that cyan structure is a tree-- has no cycles. Which in general is good

news for algorithms. Trees are relatively easy to find algorithms for using dynamic programming.

If you've seen dynamic programming for trees, this is not like that. It's a little different because we don't know what the tree is. We have to figure out what the tree is. But basically, we can guess the tree recursively.

So there's some first node-- I don't know, maybe it's this one-- just guess that. Not randomly, but guess means try all the options. So you guess some node, and then OK, we've got-- maybe we've got two subtrees. Maybe we've got three. I think that's about all there could be.

And so, just guessed that, and then so recursively guess a subtree here and a subtree here. And each of the sub trees corresponds to a sub segment of this-- of the original map. The spine, roughly speaking. And roughly speaking, these portions do not interact with each other.

There's this issue of how the rays align, and that's sort of the challenge in getting this algorithm to work. But you can essentially locally check whether a subtree is good enough. That it will interact OK with a different subtree, and just split this problem up recursively guessing all the way. Effectively trying all trees, but in polynomial time instead of exponential time.

Forget the running time is something like n to the 10 or some really big constant, but at least finally we know 2-by-n maps can be solved in polynomial time. We give the mountain valley assignment. We still don't know about 3-by-n. That's the open question, and these techniques unfortunately don't seem to apply.

You definitely don't get a tree anymore. You might get some nice structure that's kind of vaguely tree like, but this was already super complicated. So we've gotten stuck here. Any questions? All right. That's it.