PROFESSOR: Continue today. We're still in the spirit of origami. And we're going to do some origami design and foldability again. There are two main topics here on the design side we're going to talk about-- universal hinge patterns, which are the things that make underlying this robot, which I showed in lecture one. You may recall. So it's called a box pleat pattern. It's a square grid with alternating diagonal creases. And the idea with the robot is you're constrained to only-- you have to build the creases ahead of time. You can't say, you can't build one sheet that can fold anywhere at anytime. You want to build a sheet that can fold at any of the crease lines, at the built crease lines whenever you want.

And so whereas with something like Origamizer, every design has a completely different crease pattern and it's difficult to control that, here we wanted to make one. It's not exactly a crease pattern, because you're not using all the creases. It's what we call a hinge pattern, all the possible places you could fold that thing and make lots of different shapes from that one hinge pattern. So that's the first part of the lecture.

And the second part of the lecture will be about hardness. And we'll see, I think, four different kinds of origami problems which are all NP-hard. And I'll tell you what NP-hard means. And we'll prove all that. So there are lots of problems that are computationally intractable. And we're going to cluster them all together into one lecture. Because it's kind of fun to see them together. Mostly we've been talking about positive results so far.

All right, so let's do universal hinge patterns. This is pretty recent work. It just appeared at the big origami math conference this summer. And its work by Nadia, and me, and Marty, and Aviv Ovadya, who I think actually came out of this class three years ago, if I recall. It just took us awhile to write it all up. And it's why the robot has a box pleat pattern. So the idea is to require that the crease pattern of whatever you want to fold must be a subset of some fixed hinge pattern.

And the goal is to make one hinge pattern to rule them all, one hinge pattern that you can make anything. I mean you can't make literally anything, I don't think. I was wondering about this yesterday. But we're not going to try to make everything out of one hinge pattern. But we want to make lots of different things somehow from one hinge pattern. And here's the theorem that formalizes this idea of lots of things.

We're going to take the box pleat pattern from an n by n. Grid so that's you take a square grid, n by n. And then you fill in alternating diagonals. Then that thing that hinge pattern can make by using a subset of those creases, you can make any polycube of n cubes. I'm sorry, order n cubes. And what else? You can even do it seamless.

So a polycube made of n cubes is just you take n cubes. And you start gluing them together face to face until you have one connected monster. And that's a polycube. So you can make, for example, I don't know, all the Tetris pieces. Tetris pieces would be four cubes. Normally they're squares, but you can 3-dimensionalize them. That's a four cube polycube. And you get the general idea.

So this is cool because there are exponentially many polycubes with n cubes. And here is one pattern that can make all of them. And if you imagine some crazy 3D shape, you can, of course, approximate it by cubes, sort of voxelization is the usual term, the same way that we pixelize images. And then you can make basically anything you want up to the resolution provided by your sheet. So that's the sense in which this is universal.

So let's prove that. It's actually not too hard to do in at least one way. The first idea is to build something called a cube gadget. We're going to use the idea of a gadget a lot, especially in this lecture. But in general, it's a useful algorithmic tool. A gadget is just something that you reuse many times. It's like a tool. And in this case, we're going to use this folding many times. At a high level, it's just a way to fold a single cube. But it has lots of nice properties.

So this is a crease pattern. Red lines are mountains. Blue dashed lines are valleys. It folds into this thing, which you can see is a cube. Here it's some semi-transparent material. And there's some pleats coming out in the four directions. But it's basically a cube on a plane. So we started with a plane, a rectangle. We fold that pattern. And you get a rectangle plus a cube sticking out. OK, so in particular, I can make a one cube polycube out of this if I just made this, got rid of the rectangular part. But by using this gadget n times, I claim I can make an n cube polycube.

And this kind of a crazy idea that, so for example, suppose the starting sheet was not just a rectangle. But suppose it was a rectangle with a cube sticking out right here. I could still do this folding. Because this folding didn't touch The gray lines are not creases. They're just hinges. If I had a cube sticking out here. and I folded this thing, it would now be a rectangle with a cube in the center and a cube sticking out of this corner. Because that corner just folds to right there. In fact, I could have a cube sticking out right here also. And then there'd be a cube sticking out here in the finished product.

And the idea is to just keep using this gadget. And make your sheet bumpier and bumpier with more and more cubes sticking out. And this is the sort of thing if you wanted to make-- so there I had cubes that were separated from each other. If you want to have cubes that are attached to each other, you can do that too. Because all right, say here, if you apply the cube gadget at this center square, so you want to pull a cube out from there. Maybe I should show you in the previous diagram.

So you have these four squares around the center square make up the four sides of the cube, other than the top side. And so if you already had a cube on that side face, when you fold this thing, you end up with a cube that has a cube attached on its right. So if I wave my hands enough, I believe that it's possible to make anything in this way.

Let me convince you a little more formally. Let's see. Here's a real example. It gives you an idea of how you can make even overhanging cubes. All I did was initially I made this first cube. Then I made this cube with this one attached to the side, just like the previous picture. And then I just made another cube right underneath that one, so this cube was already attached to the center square. And it just got raised up by another cube. So now you've got this L overhang. And this is not quite the crease pattern. This is there's some lines that are not drawn here. But it's a rough sketch of the crease pattern.

Let's see. I'll worry about the rest next. Let me give you a little bit of an argument why this works. So a cube gadget let's say transforms a constant number of rows and columns of the grid into a cube that's sticking out of your sheet. And the key property is it works even if there are bumps on your sheet elsewhere. So when I say elsewhere, let me show you the cube gadget again. There's actually two columns sort of getting used up here. And there's one row getting used up there. Sorry, actually one row here. So there's one row and two columns in this corner. And there's actually one row and two columns used up in each corner. And that material disappears in some sense from the folding up here.

But all the other stuff, this column, everything out in this corner, and those four squares around the center and the center square itself, those can all have bumps. You better not have bumps here where I need to fold. But everywhere else can have bumps. And it's OK-- so I just need to set up my bump pattern so that when I finish making that cube, I have the bumps in the right place. So it's sort of a working backwards process. And the idea is to make a tree of cubes. So if you want to make something that has cycles in it, like a big n by n by n cube array, you can just cut it up, subdivide it, so that it's just a tree of objects.

So for example, suppose I wanted to build this shape, which is not really a tree. I can just pretend that there's a slice here. So these guys are connected in a path. But they're not connected here. In general, you can just keep slicing until your thing is connected like a tree. Then once you have a tree, you may know this fact, every tree has at least two leaves except in winter. But in this context, we're thinking of a tree. Here the tree is just this path. And a leaf is a vertex that has degree one. And every leaf has at least, every tree has at least two leaves is a fun fact about trees.

And so what we do to make this thing is make a leaf. So what I mean is we start with a blank sheet. And we'll say OK, here's a leaf. I'm going to build that cube first. So you apply cube gadget. You build a cube. Then you effectively remove that leaf. Just pretend it was never there. This is sort of a conceptual thing. You're not really removing it, but and then you repeat. It's actually a super simple algorithm. The details are a little bit tricky, because we have to make sure that this works. But once I remove that cube, now my graph is like this. So I have a choice. I could either fold this leaf next, or I could fold this one. I just keep going.

In this case, I could just go linearly along the path, like I did for making the overhanging L. And you will make all the cubes. And the property you'll have is that whenever I build a cube, all the things that were hanging off of it have already been built. Because I'm always working from the leaves up the tree. This is, for those who know trees, this is called a post-order traversal. It just means whenever I touch a node, whenever I create a node, all of its descendants, all of the leaves below it, have already been built.

And that's exactly how this thing needs to work. Because you can have existing bumps which are the things that were attached to that cube and sort of are deeper in the tree. Those are harder. You can't make those later. As long as they're already built, you just sort of keep working up the tree. And the stuff you've already built hangs off. In the end, you'll have your entire tree. And you'll have your polycube. That's it. So there are obviously details here that I'm skipping. But I think this is a fun essence of it.

Another fact, this is essentially optimal. So I'm using an n by n grid to make around n cubes. You might hope in some cases you can do better. In some cases, you can make n squared cubes out of an n by n grid. That's the best case. The worst case really is n cubes. Because if you want to make a super long polycube, a one by one by n grid, then the diameter of this thing is about n. And the diameter of my square paper is like maybe root 2 times n. So maybe I could save a root 2 factor. But just to get this diameter n, I'm going to need about an n by n sheet. So in the worst case, this is the best you can hope for from our usual diameter argument like last class.

But sometimes you can do better. So for example, here is an MIT made by Aviv

Ovadya. And this is much more efficient, in some sense because everything here is height one. You can share a lot of those pleats. You don't have to waste rows for every single gadget. You just, you can share those wasted rows along all the guys who are aligned. And general picture, this is actually Aviv's master's thesis. The way I described it, if you wanted to make two cubes right next to each other, you fold two separate cube gadgets that each use up their rows and columns.

You can be a little bit more clever like I was saying and share those used up rows between the two guys. Because they are horizontally aligned. And it's a little more efficient. And that's essentially what's done in that example. But there's still some sort of wastage here. You don't really need to do that. You really want to build the two by one thing. And with some fancier version of the algorithm, you can do that. If you want to check it out, you can see Aviv's [INAUDIBLE] thesis, which was just completed last month. And yeah, we don't have a formal sense in which in how much better this is. But it's sort of opportunistic, tries to be as good as possible. And that is one version of box pleating.

But in the same spirit, I want to talk about another situation where we can do very well. And here we can prove that we can do very well. So again, trying to be more efficient and this sort of square wasted. We have n squared of material. We only make n things. Be really nice if we could make n squared things out of an n by n grid. And oh, here's another fun example which uses all those optimizations building a car. This one there's a real version of, but I don't have it here.

And where we can be particularly efficient is maze folding. So suppose you take a graph on an n by n grid. Here it happens not to be square. And then you just sort of extrude that graph out from the sheet. That would give you a bunch of walls in an orthogonal, 3D pattern. Let's say I extrude by one unit, one unit of this square. And I claim I can fold this 3D shape from a square of paper that is just 3n by 3n. So it's just a constant factor shrinkage. And this is essentially the best you could hope for. Yeah, should I try to argue it.

So if you look at where did this material come from, well you've got to go up this wall, down this wall, over the side, up the wall, down the wall, over, over, over, over, up, down. And in general, you have to go up and down and along the floor. So that's three for every square that you have here. So that's maze folding. And this is work with Jason Ku and Marty, also from the big origami conference this summer.

So you could call it a maze. You could call it a graph. But it's orthogonal. It's on the grid. And it's extruded from an n by n square let's say. And that thing can be folded from an order n by order n square of paper. And if you're extruding by one unit, this big O at that three. And to me this is really exciting. Because one of the big mysteries to me in origami design is in practical origami, you usually start with a sheet. And you make something that's like two or three times smaller and never much more. And it would be really nice to capture theoretically what things can you make by only shrinking by a constant factor.

Like checkerboards, we know, or we think, you have to shrink a lot. For an n by n checkerboard, you have to shrink by like factor of n over 2, n over 4, whatever the best bound is. But it seems the more complicated you want, the more you have to fold. Here, I can make a super complicated maze. It could be a million by million. And I'm still only shrinking the sheet by a factor of three. So this is one of the few results we know where you can get a large class, and yet you're only shrinking by small factor.

The proof of this is also pretty easy, in fact even easier than the previous one. Again, we're going to use gadgets. And we're going to combine them in different ways. But the idea is we just make a gadget for each possible vertex in this graph. So a vertex in the graph could have no edges incident to it. So we call this degree 0. Or it could have one incident edge and the rest are absent. That's degree 1.

Or it could have two incident edges. And there's two ways it could be like that. It could be a turn or it could be straight. You could have three incident edges. Or you could have four incident edges. So that's all the possible vertices in our orthogonal graph.

And we're just going to make a crease pattern, a folding for each one of these and then just combine them together. Now it takes a lot of care in designing those patterns that they actually fit together. But all you need to know is that it can be done. And they're not trivial. But once you have them, it's easy. So you've got degree 0 on the top, degree 2 straight, degree 4, degree 3, degree 1, degree 2 turned. And they're at, I guess, an increasing order of difficulty.

And we need to know that these things exist. One way is to in some sense just to draw the picture of the folded state. But there's so many layers here it's a little hard to see, and so in order to really prove that these things exist, Jason drew diagrams of how these things could actually be folded in isolation. In some sense, we only care about this final 3D picture in knowing that it works and is non self-intersecting and all that. But to show that these exist, one way is to actually build them or show the sequence that got there.

But in reality, you wouldn't fold it this way. Because what you do is you take these crease patterns on the left and just start pasting them together. It's like a big cut and paste job. So you say, oh well maybe like if I made the square of turns, I would just take one of these, copy, rotate, put it here, copy, rotate, put it here, copy, rotate, put it here. And that would make a square root of turns.

And what you need for that to work is that the interfaces here are compatible. You can think of it in the crease pattern, or you can think of it in the 3D state. The interface in all of these pictures is that when you have an actual edge that's raised, it's very simple. It just goes over, up, down, over. And when you have an edge that doesn't exist, you go over and then you have a pleat underneath. And then you go over. And it's important that those are the same total length. Because you need to be able to choose whether it's a raised edge or a non-raised edge.

And because all of those interfaces are the same, all of the down edges are this kind of double pleat. And all of the up edges are just the ridge. These things fit together. And you can see that in the crease pattern also. Here's what the pleat looks like. And it can match with this pleat or this one. You can rotate. Here is the ridge. It just goes up and back down. And you just paste those together. And you get your desired crease pattern. So here's a simple example. The graph has almost all the vertices, everything except 0 I think, and straights. All right, not quite all of them.

But I've color coded here. So like you take the graph. You just embed it on a slightly larger grid. And you replace each of those vertices with the crease pattern that makes that thing. You just have to rotate it for it to work out. And then you see that all of these creases just meet up correctly. And then that's your crease pattern, which will fold this. So that once you have the gadgets, the algorithm is super simple, just a bunch of cutting and pasting. And you can do more complicated examples. Here's an actual maze. Get some pretty complicated crease patterns. Not so easy to fold. But it can be done. I didn't bring the physical model of this, because it actually looks better in photograph. It's a challenge. This was folded by an undergrad here, Chris Chin.

And in fact, it's such an easy an algorithm, I implemented it as a web application. It runs in JavaScript. And you can go play with it. It's [INAUDIBLE]. And you can say, OK give me a random maze. And you get a 3D representation. And you get your crease pattern. You hit print. It will print out this part in vector forms, nice high resolution, and all that. You can make more mazes. If you really want to make a particular maze, you can fool around with that.

And you can also write important messages like something like that. And then you get this 3D representation, which you can fold by this simple crease pattern. If you make that, one please send it to me. But that may take several hours and use a big sheet of paper. I mean, you're only shrinking by a constant factor. How hard could it be? The answer is quite hard, because the gadgets interact. It's tricky.

All right, that's maze folding. Any questions about maze folding or this stuff? I think that's the end of positive results for today. And we move into NP-hardness. Good? All right, well every origamist knows, and if you've been working on a problem set, you should know by now origami is hard. And we'd like to prove that formally. And because we're computer scientists, we like to know what we can't know essentially.

There are a lot of problems where there's no efficient algorithm. And instead of just giving up, we like to prove that no one can find an efficient algorithm. Because then we know we're kind of done. That's comforting.

And NP-hardness, let me change this, I don't want to formally define it. Because it's a little bit technical. But a working definition, there are lots of working definitions that are super easy to tell you. So the informal version is that NP-hard problems are computationally intractable problems, meaning there's no tractable way, no efficient way, to solve it on a computer. But what it really means, or what it really implies, is if a problem is NP-hard, then there's no efficient algorithm. I wish we could just say that. But there's a slight catch, unless P=NP.

How many people know about NP-hardness? Well, how many people don't? Just a few. All right, I'm going to go relatively quickly then.

Those who haven't heard NP-hardness and haven't heard of P=NP? Good. So you've all heard about this famous problem. It's almost certainly the case that P does not equal NP. Pretty much everyone believes that, unless you watch my April Foo;s video and search for P equals NP in YouTube. And what this means intuitively is that there's no cheating. There's no trick to make lucky guesses in life.

If you've got two choices and you don't know which is the right choice, and you're computer, you can't-- computers aren't lucky. The best they can do is try both options. That's what P does not equal NP means basically. There are no lucky, there's no way to simulate luckiness. That's the technical version. Those who know NP should agree with me. That is real. It's not how most people explain it. But it's how I like to think about it. And from that perspective, it's kind of obvious. But it's very annoying. It's unlikely anyone will prove this in the near future, says Scott Aaronson, who bet $200,000 that that was the case. Right.

So you don't need to know the definition of NP-hardness except that it probably means there's no efficient algorithm. For what we need here is this idea of reduction, that we can take some hard problem, known NP-hard problems. Why did I write ness there? NP-hard problems show that a problem that we care about, like origami design, is even harder than those problems. Therefore, it's also NP-hard. That's the usual way for NP-hardness. It's always showing that your problem is harder than another. Or showing that one of these problems is easier than your problem. And therefore, yours is harder.

So I'm going to need three problems today. I'll start just by defining two of them, which you've probably seen before. One is called partition. And I give you n numbers. I want to know, can I split them into equal halves? Be a little more precise. Two halves of equal sum. So suppose you're playing video game *Team Deathmatch.* You've got two teams. You've got a ranking for every player. You'd like to divide your players so that the sum of the rankings on the red side is the same as the sum of the rankings on the blue side, so it's an even game, more fun, whatever. This problem, sadly, is NP-hard. There's no way to do that, even approximately. The only catch is this problem is hard only when these integers are super big, like exponentially large in n. This problem is called weakly NP-hard. So as long as your player rankings are nice and small, there actually is an efficient way to solve that. Problem but when the integers are big, this is NP-hard.

An even nastier problem is satisfiability, or SAT. Here you're given some Boolean formula. So it has a bunch variables, like x and y. You can do and. You can do not. And you can do or, let's say. That's all you need, x and not y or z. And you want to know, can I make that formula true? Is there some setting to the variables x, y, and z, or in general there's n variables, so that the formula is true?

So we'd say the formula's satisfied. And that's NP-hard. And here there's not even any numbers to make it hard. So we call this problem strongly NP-hard. And this is really the prototypical hard problem, NP-hard problem, is the very first one is proved NP-hard. It's the only one really that we usually prove without a reduction, you could say.

So what we're going to do it for four different origami problems is show those problems are easier than our problems. Therefore, our problems are NP-hard also. How do we show that, say a partition is easier than some problem? Well, we just take a parti-- we show that partition is a special case of our problem. It's a special case. Clearly it's easier. So to show that, we take one of these partition problems, like n integers. We want to know whether you can split them in equal halves. I'm going to convert that into my problem, so that that problem has a yes answer if and only if the partition problem has a yes answer. Therefore, really this problem becomes a special case of the one I care about. Therefore, that problem is harder, and therefore also NP-hard.

So I'm going to start out with a super simple example which we did in the problem session on Monday. Someone pose this to me after class last week, I think. I think two lectures ago. So here's a problem. I give you a single vertex hinge pattern. Someone built some crazy robot with some crazy pattern of hinges all around a single vertex. You want to know, can I make anything out of it? Does some subset of those hinges, if I take that as a crease pattern, fold flat? Who posed this problem? Anybody remember? All right. Maybe he's not here. I forgot, unfortunately.

He asked me, so what about this problem? I said, yeah it's obviously NP-hard. And we thought about it for five minutes. And then after five minutes, it's obviously NP-hard. So let me show you the obvious proof once we have it. We're going to show that this problem is harder than partition. So we take n integers, want to know how to divide them. And it's really simple. So suppose someone gives us n integers. For integers, we're going to scale them all by the same scale factor, so that their sum equals 360 degrees.

And now, lo and behold, those integers are angles in a crease pattern, in a hinge pattern. So you just take those numbers. You turn them. You put them on a wheel. OK. Now presumably, it's not a flat, foldable, single vertex crease pattern. Otherwise, the answer would be yes. We want to know, can I remove some the creases to make it flat foldable? Now if you think about Kawasaki's theorem, the sum, the alternating sum of angles equals 0. Remember how that proof worked? We said, OK you follow one angle for a while. Then you turn. Then you go back. And somehow, you have to end up back where you started. That's the equals zero part. And it's the alternating sum part.

Now in this case, we have a choice at every crease. We could include it or not. If we include the crease, we turn around. If we remove the crease, we keep going straight. So now the problem is I give you all these integers. And I can go right. And now I have a choice. Do I go right or left by the next integer, theta 2. There's theta 1, theta 2. Each one, I have a choice. Do I go right or left? In the end, the sum must equal zero. So this problem is the same problem really as given a bunch of numbers, can I assign signs of plus or minus so that they add up to 0?

But assigning signs of plus or minus so they add up to is the same as saying all the plus guys equal all the minus guys. So really, this problem becomes assigning pluses and minuses, so that the sum with the appropriate pluses or minuses of theta i is equal 0. That's the Kawasaki version. But that's the same thing as saying the sum of the pluses equals the sum of the minuses.

And if that's the case, then really you've partitioned your numbers into two halves of equal sum. So this is going to be possible exactly when there's a partition. And so you've converted partition into this problem, which means this problem is harder. Because it has other situations maybe. But you take any partition, and are actually pretty much identical. But partition becomes a special case with this problem. Therefore, this problem is harder, and therefore NP-hard. It's only weakly NP-hard. But that's a minor detail. Clear? All right.

Now we move on to the more, I don't know, that's a brand new result, and a very simple one. Now we move on to the sort of more established, well-studied problems. The NP-hardness proofs are quite a bit more complicated. But they're still, they all follow the same spirit. And they're a lot of fun, because they involve gadgets. Here there aren't really any gadgets. We represented an integer by an angle. It's pretty direct. But in some sense, that's the gadget. And we just used n of them. All of the other proofs are going to use tons of gadgets. And they're kind of fun. I love NP-hard proofs. They're one of my favorite things.

All right, one of the first things we talked about in this class was simple folds. And we showed that in one dimension, if you had a one dimensional piece of paper, simple folds were universal. You could make any flat, foldable crease pattern. And if you remember, at the very end, we talked about map folding, which is where you have a bunch of orthogonal creases in a rectangular paper, and maybe also with mountain valley assignment. And we showed that this problem really is a bunch of one dimensional problems. And so if we wanted to solve it was simple folds, where you only fold along a single line at a time by 180 degrees, then really this turned into a one dimensional problem in one dimension, then a one dimensional problem in the other dimension. You just kept doing that until either you got stuck, in which case the thing was not flat foldable, or you flat folded it. So this is an example where simple folding is easy.

But in general, deciding whether you can fold a crease pattern flat by simple folds is NP-hard. So for this special case, it's easy. For another situation, which is when the polygons are not just rectangles, but are a little more general, then it becomes NP-hard. So in general, the problem is given a crease pattern, possibly with mountain valley assignments, doesn't really matter, can it be folded flat by simple folds, by a sequence of simple folds? Initially we thought maybe this problem was polynomially solvable, because simple folds are so damn simple. We were wishful, but it's not true. This turns out to be NP-hard.

If we take this situation, the map situation, and we add 45 degree folds, creases, so I just add some things like that. It looks kind of crazy. Got to do some sub division. I just add 45 degree folds. Then this problem becomes NP-hard. So with orthogonal creases in a rectangle, it's easy. But you had one more direction, it's hard. Another version that's hard. If I keep all the creases horizontal and vertical but I make an orthogonal polygon instead of just a rectangle, then it's also NP-hard.

I'm going to show this one because it's easier. But this actually just converts. You can set up 45 degree folds so that you are forced to make a particular orthogonal polygon to get started. And then it's the same proof. So I think I have the proof here. So again, we're going to reduce from partition. So we're given n integers. We want to know whether we can divide them into equal summing halves. And we're going to represent those integers by these lengths.

So here's an, then a3, and a2, and a1. So the lengths in the top part of the staircase are integers. We want to somehow divide those into equal halves. And when they have two halves, and I suppose at their sum is L, capital L. I can just add them all up. I should get 2L. I divide by 2, I could get what the target sum is. So I know ahead of time without solving the partition problem what L ought to be, and what twice L ought to be.

And then I build this frame over on the right whose height is exactly twice L. And so there's these creases in the horizontal creases down the staircase. All these creases have to get folded eventually, and by simple folds. And then there's also these two vertical creases bounding the frame. So the idea is, well you make some of the horizontal creases. Then you fold one of those horizontal creases. And then eventually you have to fold-- I'm sorry, one of those vertical creases. I always get horizontal and vertical confused, which causes me great difficulty when trying to sleep.

But so when I fold the first vertical crease, whatever's over here comes over here. If it hits the frame, I'm in big trouble. Because then how am I going to fold the other horizontal, other vertical crease without colliding with the frame? If I want to avoid collision with the frame by simple folds, and I fold the vertical crease, I really should not be touching the frame. So you try to fold through it. They're both valleys in this case. It doesn't matter too much.

So what I really need to do is fold this thing compactly like this, so that it just fits inside the frame. And the only way to do that is for each of those vertical segments of the staircase to decide should it go up, or should it go down? And you do that in actually pretty much the same way this proof works. For each of these segments here, we were deciding should I keep going straight and go, or should I turn around?

Over here, it's the same deal. I either keep going straight, whichever direction I was going, up or down, or I turn around. But I always have a choice at every crease. I'll just fold it or not to make it go up or down how I want. The up guys are going to be one of the halves. And the down guys are going to be the other halves. Here, it's the other side of the paper, the dark blue versus the light blue. As long as those numbers add up to exactly L, I've got this twice. Then I start here at the middle of the frame. If I can get them to balance out, I will end over here, also at the middle. And then I go up by L. And then I go down by 2L. And that will just fit inside the frame. But only if I stay in the middle will that 2L fit inside the frame. And so the only way for these two creases to be foldable and not collide with the frame is if I can solve the partition problem.

Therefore, so finding simple fold sequences is actually way harder than partition in some sense. Because this is just a very specific kind of map, specific kind of crease pattern you might want to fold. And folding that is exactly partition. So the general simple foldability problem is going to be NP-hard, because it includes partition as a special case. Clear? Good. That's our easiest proof among the next three. Going to get increasingly difficult, I guess. But I'll just getting increasingly sketchy, so it will be easy for me. I mean, the more complicated a proof gets, somehow I feel like the number of interesting details in a proof remains constant. If the proof gets more complicated, then I throw away more of the messy details.

All right, the next theorem is also about flat foldability. But now I don't just care about simple folds, I want to look at regular origami folds, which are folded states. So we talked a couple lectures ago about local foldability, which was can we assign mountains and valleys to some crease pattern so that each vertex, if you cut it out into a little disk, would by itself fold flat. And that was easy, polynomially solvable, actually linear time. And that was a result by Bern and Hayes. Another result in the same paper by Bern and Hayes, which is actually sort of the bigger result that everyone knows about, is that if I give you an arbitrary crease pattern, I want to know just does it fold flat, that's NP-hard.

They proved actually two NP-hard hardness results. So this is way back in '96. This is one of the oldest results in computational origami. So I give you a crease pattern. I just want to know, is it flat foldable in the global, in the regular sense? This is strongly NP-hard. The proof I just gave is actually weakly NP-hard. It is not known whether that problem is strongly or weakly hard, but at least weekly. The other thing they proved is that if you're given a flat foldable, even I tell you it's flat foldable, and I even give you the mountains and valleys that make it work, still flat folding thing is NP-hard.

So if I give you a flat foldable mountain valley pattern, all that's left is to decide I can fold each vertex. And then there's this issue of layering. If I have two layers of paper that are overlapping, they could be like this. Or they could be like this. And if I have, for example, two crimps, I could decide how the layers go. Figuring out what the right layer ordering is is really the heart of the problem. This is what makes it NP-hard. Because we know finding a mountain valley assignment-- locally things work-- is easy. But getting that layering to work is hard. That's what all these proofs say.

So I'm going to talk about the proof of the first result though, because the second one is pretty complicated. It's the same spirit, just a lot more details. Again, we're going to do a reduction. And we have these two nice problems, partition and SAT. I'm not going to use either of them, though in theory you could use SAT. I'm going to reduce from one of my favorite problems, one of my favorite NP-hard problems, I should say. Wait, no. That's the next proof. This is not my favorite. It's a pretty good one, though. I do like it. Is just a little more technical.

All positive, not all equal 3SAT, has anyone heard of this problem before? Nadia's heard of it, because she TA'd this class before. It's no surprise. So you've read the book. This is, there aren't a lot of proofs that use this one. But not all equal 3SAT is actually fairly common. All positive is just makes a little more convenient. I think actually the original proof didn't use all positive. But our book does. Because it simplifies things. So let me tell you what this problem is. Most people don't know it. So don't worry.

You could technically, it's a version of SAT. But instead of giving a Boolean formula, I'm going to think of that it's really a Boolean formula. But I'm going to think of it in a simpler way, which is I give you a bunch of triples if variables, so like xi, xj xk. And I want to know is there a Boolean assignment to those variables? I want to set each of them to true or false. Say there's n variables. So that no triple is all equal, no triple is all true or all false. Maybe we could call it the all state problem. I've been watching too many ads.

All right, so if not all equal 3SAT, that's this version. Actually, it's also all positive, meaning I don't have any nots in here. So ignore that. Ignore the technical term. This is the right definition. So I have a bunch of triples of variables. I just don't want them all to be true or all to be false. So two of them could be true and one false, or two of them could be false, and one true. That's all. It turns out this is basically equivalent to SAT. But the proof of that is kind of messy. So I don't want to do it here. And so that problem is NP-hard. Just take that on faith.

And now I want to show that global flat foldability includes all positive not all will equal 3SAT as a special case. And therefore, it's also NP-hard. So I'll give you a preview. Start with a high level, what we need in terms of gadgets. And then I'll show you the gadgets, and then show you how they fit together. So this is a general picture, in fact, of what a SAT kind of NP-hardness proof looks like. If you haven't done many of them, now you'll know. If you have done many of them, you will recognize this pattern, which is to represent Boolean-ness, we need something that represents true and false. And that's usually called a wire. We think of digital signals like chips. And then we've got to be able to connect those wires together to do interesting things.

In this case, the interesting thing we need to do is tell if I have a triple of them, are they all true or all false, and somehow force them to not be all true or all false. And in this case, that will be a not all equal clause. It'll be a gadget the folds flat, exactly when those wires that come together, three wires come together. And if they're all truth, they won't fold flat. If they're all false, it won't fold flat. In all of the cases, it will fold flat.

So if we could build that, that'll constrain the variables that I connect together with a not all equal clause. But how do I actually move the wires around to make them connect together at these clauses? Well, I need something called a turn gadget. If I have a wire going straight, I'd like to be able to turn it to some other angle. I could just sort of move them around. It's harder than it sounds. And I'll also need a split.

Because in this, it's maybe not obvious from this formulation, but I have these n variables. I might have many more than n clauses. Each variable, like x1, might appear in 100 different triples. And so I actually need 100 copies of x1. And that's what a split gadget does. And we're going to build one gadget called a reflector, which actually does both of those in one fell swoop.

And the last thing we need is a crossover. Because you make all these connections between variables and clauses or triples. And they might have to cross each other. And we want them to cross each other, but not affect each other. And because we're in a sheet of paper, we've got to deal with that. It's going to be these wires to go right through each other. We got to make it still flat foldable without affecting either wire. That's all. That's pretty standard for this kind of proof. These are the proofs that I like the most, actually. Pretty fun.

So in our case, a wire is going to be super easy. It's just pleat, so two very nearby parallel creases. And because they're nearby, I mean, if you look locally, it's like a one dimensional problem. They can't both be valley or both be mountain. Because then these two big panels would intersect each other. So one of them is valley, one of them is mountain. There's exactly two choices.

I'm going to always have an arrow on my wire so I have a sense of orientation, which way the signal is going. And once I have an orientation, the left side, if the left side is valley, that's true. If the left side is mountain, that's false. That's just I'm going to decide it that way. It doesn't actually matter, because in this problem, true and false are symmetric. But I just need to be consistent about which is which. OK, so that was the wire gadget.

The next one is the not all equal gadget. And I'm guessing this is where Bern and Hayes started. Because it's sort of the heart of the proof. Then there's all these details to connect up with the wires, and split them, and whatnot. But this is something called a triangular twist. You may have folded one before. It's kind of classic. So the crease pattern's in the top left. And the idea is I have three wires coming together. I want this thing to fold flat if, and only if, the wires are not all the same. And I've drawn here sort of three of the possible patterns you could have.

Here I have one truth and two false. That folds flat. Here I have three false. And here I have two true and one false. This also folds flat. In all cases, this is what the folded state looks like. I maybe never mentioned this. But this is an important concept. If you take a crease pattern and you say I want to fold it flat, you can tell where all the stuff is going to go in terms of geometry. What you can't tell is the layer ordering. That depends on the mountain valley assignment. That depends on how you decide the layers to stack.

But you can tell already where everything goes in the plane. Because you pick, let's see, did anything stay fixed here. Yeah, the triangle stayed fix. So you pick some face to stay fixed, like that center triangle. You put it there. And then you say, OK. Well, where is this flap? Where is this face of the crease pattern going to go? Well, it gets reflected through that horizontal line. So it goes here. And you can just keep playing this reflection game. Because you know every crease geometrically, it's a reflection. It could be a reflection this way or this way. But it just, as soon as you cross a crease, you end up reflecting your material.

And so you can draw this picture without knowing anything about how it's folded. Just if there's a flat folding, it's got to look like this. And the annoying thing about this set up, because that angle up there is 35 degrees, these guys are going to overlap in this common center. And in this situation where it's all false or all true, you get an intersection there. It's a little tricky to prove. And you've just got to fiddle with one. I should have brought one, but that is true. And so in the all false and symmetrically the all true case, this thing does not fold flat. In all the other situations, it folds fine. Because the layers get out of the way.

OK, also over here, and there's a bunch of these in the book, are little analyses of which of the creases have to have the same mountain valley assignment and which have to have different. That's actually how these patterns are drawn. This is from our old local analysis of a single vertex crease pattern, right. You've got these four creases. Check it out. The only one that could be crimped is this one. Because it's globally smallest. All the other guys have a bigger neighbor or are in trouble. So this guy has to be crimped. So these two are not equal. And therefore, these two must be equal. And that's symmetrical all the way around.

So in fact, you know that this crease is different from this one, is equal to this one. You already knew that these two are different. So that's OK. So you can figure out these crease patterns. There's a little bit of flexi-- once you have the mountains and valleys coming in, you know how the mountains and valleys have to be in the center. Just makes life easier. There's only one mountain valley assignment you need to consider. And the symmetric one turns out to be bad.

So that's two gadgets. We got the wire, the not all equal clause. Of course, if we're lucky, all the wires will just meet at the right points. But I'm going to need many copies of them. I've got to move them around to reach all the different clauses, all the different triples. So next up is reflector. This is actually in some sense really easy. So we have, here's our input. What the reflector is going to do is it's going to make two copies, one down here which is negated, and one up here, which is the same value. And it also effectively turns the signal two different directions.

So to see why it works is actually pretty easy. It's just a local analysis again. You look at this vertex. This is the only increase they could be crimped. So we know that these two guys have opposite assignment. Therefore, these two wires will have opposite value. And actually, they have the same value if they're pointing in the same direction. But they'll have opposite value because I decided this one's pointing down. So if this one's a mountain, this one has to be a valley. And so if this is true because it's valley on the left, this one's false because it's mountain on the left.

OK, and then you can do the same thing. Then you also know these two guys are equal. And because again, this is the only crimpable pair here, these two creases are not equal. And so you know these two guys are equal and these two guys are equal. And so you know that this valley is propagated up there. And therefore, these two wires have the same value. So you've split the signal. You've made a positive copy and a negated copy. And if you just do this again, you'll get-- this will make another negated copy and a positive copy.

So I get two positive copies and a negated one. You can just keep doing this and you'll get tons of positive copies, tons of negative copies. So it doesn't even matter if it's all positive. But if you take all positive ones. Let the others go off to infinity. And now you want to move around so that they hit those not all equal clauses. How do I move them around? I just use more reflectors. If I come in at some angle, I can now turn by however much that is. Or I can turn by however much that is. I'm not going to try to figure out what those angles are. But it turns out, this is enough to make everything work.

The last thing you need are a bunch of crossover gadgets. This is one of them. There's a second one. I'll just wave my hands and say, if you take two pleats in the obvious way, they really don't care about each other. Because the way to fold this locally, is to fold this diagonal pleat and then fold the vertical pleat. And it will work whether one pleat is true or false. It doesn't matter which side is valley, which side is mountain. It always works. So crossovers aren't actually that big a deal. Once you have that, you take those gadgets. And you put them together into a monstrosity of a crease pattern, which looks something like this. And this took forever to draw, I remember.

So we have on the left side our variables. These are just wires. They're pleats. And each one could be folded true or false, left over right or right over left. And then I do a whole bunch of reflectors, just reflect, reflect, reflect, reflect, reflect, just to make a whole bunch of copies. And then at the top there, way at that little yellow triangle, is a not all equal clause. And is a not all equal clause between x3, where we make a copy. Wow, this is crazy. I end up making a copy, negate it downwards, and I flip it around. And it goes up straight. And then I turn it at an angle to hit the triangle dead on, the way it's supposed to. Then I also take x1 up the top, just get a copy straight off the top there. And x2, I take a copy here. For some reason I feel like negating it, and turning it around, and spitting it up there.

And then I turn it back down. It hits the yellow triangle at just the right angle. And therefore, if this thing's going to fold flat, it must be that x1, x2, and x3 are not all equal. And you just keep doing that, one for every triple. Remember, we're given as input one of these not all equal SAT problems. I give you a bunch of triples. I just put the wires together according to those triples. This thing we flat foldable if and only if this formula is satisfiable. You can set the pleats so that the desired triples are not equal. So that's pretty crazy. This is in some sense one of the hardest proofs that we will see. But in the end, you get NP-hardness of flat foldability. Questions?

OK, I want to do one more proof sketch. This is yet another paper that was at this year's Origami in Science Math and Education conference. This is, it's fun to teach this class. Because it changes so much over a period of three years. Lots of new results. This is a result with a guy named Sandor Fekete from Germany. He does a lot of optimization, and Robert Lang, and myself. So disk packing is something we talked about last class in the context of the tree method of origami design. We said, in particular we were thinking of this situation where we wanted to make them a Margulis napkin counter example.

So this was equivalent. If we wanted to build this uniaxial base and sides, was the same thing as packing n disks into a square. That's what the tree method shows. So in some sense, I'm talking about this problem. But equivalently, I'm talking about this problem of packing disks into a square. Now if I give you n unit disks all the same size, I want to pack them in a square, that problem cannot be NP-hard. That's annoying. But the input is only n. It's not very interesting.

So to make it harder, be able to show that this problem is computationally intractable, I'm going to consider generalization, which is I still have this kind of star tree, still a very simple kind of uniaxial base I might want to build. But now all the limbs are different lengths. So what that corresponds to is I have disks of various sizes. It's kind of fun. It's like bubbles. They all have to fit. They can't overlap each other. All of the centers of the disks are inside the square. And I want to know, can I-- how big a square do I need to pack them?

OK, I'm going to formulate it as a decision question, yes or no. Can you place n given disks, I give you the sizes of them. I want them to be non-overlapping. They can touch on the boundary but the interiors can't overlap. And I need the centers to lie in a given square. So I want to know, can I make this uniaxial base from this square paper? And I claim that as NP-hard. So good luck solving it perfectly. And what tree maker does is it solves it approximately with a heuristic.

For those who know approximation algorithims, this is a problem you can find a constant factor approximation. There's one in the same paper. But it's still not, still unresolved how close to optimal you can get. But I want to focus here on the NP-hardness. So I should say this problem is NP-hard. So we're going to prove that. And now I get to reduce for my favorite problem, favorite NP-hard. It's my favorite partly because not many people know it unless you've done NP-hardness proofs with me before. And it's very powerful.

Whenever you have a problem that involves numbers, and this problem involves numbers. It's the radii of the disk's. Three partition is the problem you should know. Partition is all right. The three partition is like 50% better. Because partition is like you've partitioned it to two parts. So it's like two partition. Three partition, you partition into? Does not one have the right answer? I hear three parts and five parts. Any other guesses?

AUDIENCE: Seven

PROFESSOR: Seven. Keep going. n over three parts is the answer. So it's not so obvious. Maybe it should be called n over 3 partition. So instead of the number of parts being three, the size of each part is three. So n should be divisible by 3. So I'm going to partition it to n over 3 triples of numbers of equal sum. This problem is cool for this technical reason that it is strongly NP-hard. So even when the numbers are really small, like about n, this problem is NP-hard.

With partition, the numbers had to be exponential and n for the problem to be hard. And that's kind of artificial here. So this is actually strongly NP-hard. So this problem is hard even when the disk sizes are not that different. There's like a range between one and n, let's say. If I reduce from partition, I'd need some disks to be microscopic and some to be ginormous, exponential difference. I'm going to reduce from three partition because I get a better result. I get a stronger result the says even when the disks are not so different in size, this problem is NP-hard. That's my problem that I've got to start from.

So I give you n integers. I want to somehow triple them up using a disk packing. So somehow solving a disk packing problem is going to solve this three partition problem. And it's kind of crazy. I'll give you the high level picture. It's n over three identical pockets. So in this case, there is kind of like the proof where we had the simple folds and we a frame and some stuff. We're going to have some infrastructure, which in that case was the frame. That was sort of a basic thing that always exists.

In this case, we're going to have some infrastructure which is a whole bunch of disks that just sort of set, instead of having this very open problem, open playing field with a square, and like you could put disks anywhere, I want to partition my square into lots of little pockets, all the same size, all the same shape. And all other pockets are going to be much smaller. So it's a little hard to draw this. Because while I said that the disks aren't that different size, they're only a factor of n different in size, if you try to draw these pictures, it gets tiny very quickly.

So here's the high level picture. You have a square. And I'm going to put down these disks. There's 20 disks here, I think, 4 times 4 is 16 plus 5. I'm sorry, 21. So these 21 disks, they have a unique package, this one right here. You can see that because these four disks have to be in the corners, can't fit one in the center and have room for the others. Then this guy has to go right there. And then these four disks are unique if you set it up right. It's a little bit, it's hard to draw. But these are little bit wedged over to the right.

So you set up these disks. That's infrastructure. It's the only way to put them. And then there's these little pockets here. They're nice. They're symmetric. They're like a triangle, an equilateral triangle so to speak, except they have curved edges. They're all the same size. That gives me four pockets. But I want n over 3 pockets. So I'm going to need a lot more. So I take each of these pockets. That's these three disks. And this is the triangular pocket in the center. And I put down these 17 disks. It's almost the same. And I can see it's not quite drawn perfectly. This is supposed to touch here.

This guy is floating a little bit. And what we end up, this guy's uniquely placed. And then these guys have got to go in the three remaining pockets. What we end up getting is another equilateral whole pocket right there. And these three will have the same size and be identical. If I do that in all four of these, now I'll have 16 pockets. And each time I do this, side I quadruple the number of pockets. So after I do it, whatever log n times, or I apply that gadget around n times, I get around n over 3 pockets. They're all identical. The ones I don't want, I'll just throw a disk in there to destroy it. That's the infrastructure.

Now I've got n over three identical pockets. There are these other pockets. Just throw in disks in there. I mean, I can't force them to go there. But if you set disks to the right size, they really have to go there. And so all of the other pockets will get even tinier. It will destroy them. That's the infrastructure. It's already pretty crazy. But then the last part is actually kind, is very cool and elegant.

So I've got these n over 3 pockets. And this was the central idea we started with. It actually took us awhile to find this way to force a bunch of identical pockets. It's easy if you start from a triangle of paper or a rectangle of paper. But we really wanted to start from a square. So that's the case we care about in origami. And so we came up with that proof. This is hard to draw perfectly. Here's an equilateral triangle, so to speak, of equal radius disks pairwise kissing. A little slightly wrong aspect ratio. But you get the idea.

I'm going to put a disk here that has a little bit of slack. I'm going to put a disk here that-- actually, I'm going to draw them without slack first. And then I'm going to say how the slack is. Because that'll be clearer what I mean. So suppose this guy was actually kissing all three. And this guy was kissing three and it's like a bunch of threesomes here. All right, now I want to make each of these disks a little bit bigger, which will make this impossible. But then I'm going to make this guy a little bit smaller. So I'm going to make this one ai bigger. I'm going to make this one aj bigger. I'm going to make this one ak bigger.

And this one, what did I call it, L, smaller. So L is the target sum for a triple, meaning I take all the integers that I'm given. And I add them all up. And then I divide by n over 3. That's what every triple should sum to. Because they're all supposed to be the same. So I call that L. This is the sum divided by n over 3. So I'm going to make this one that much smaller. This is all slightly approximate. Bear with me.

So that gives it a little bit of slack, which is good because these guys are little bit bigger. And if they are bigger, in total if the sum of these two values is L, this will barely fit. If it's bigger than L, it won't fit. If it's less than L, it will fit. But because L has to be, L is always the average sum of the triples, if they're all going to-- if you have a bunch of triples, all of whose sum is less than or equal to L, in fact they all have to be exactly equal L. Because there's no slack.

So the only way for these guys to pack, now I said this is ai, aj, ak, in fact I'm just giving you a heap, a bag of disks. I don't say which ones go where. You have to choose. So clearly, these are all the same size. And they're just going to go in the center. And they'll wiggle around a little. These guys, you get to choose how I triple them up and how I put them into these n over 3 pockets. So that's your flexibility in disk packing. It's your only flexibility in disk packing, is how you triple them up. And the only way for it to work is if you can triple them up so that their sums, the sum of the amount by which they are bigger, is exactly L. Because that's exactly the slack of this disk. And it will just fit. Question?

AUDIENCE: So the fact that you have an increase in the ai, j, k [INAUDIBLE] geometry work out with some tangents and stuff?

PROFESSOR: Yeah, all right. So you raise a good point, which is I said this is ai bigger. I didn't really mean that the radius is ai bigger, although it's actually close to that. ai is an integer. If I made it that much bigger, it might be huge. But what I really mean is I take ai, I multiply it by a very small number greater than zero called epsilon. And all of these are actually by epsilon. And that is actually how much you change the radius. Maybe there's a second order term. But to the first order, yeah you think, oh, there's this trig. And I've got to do tangents and all this funny stuff.

It turns out to the first order, actually things work really simply. If I shrink this disk by an additive amount, or sorry, if I grow these disks by and additive amount, I shrink this by the same amount, this will still work up to the first order, so up to the first derivative. So you might have to do a little bit of fudging. I can just subtract off an epsilon squared or something to give me just a little bit of freedom. And then this is actually how big the disks are.

But you raise a good point. There's details I'm hiding here. It's actually pretty clean. You work out the tangents and things just are pretty, surprisingly. We thought this would be messy. But it actually works pretty well. Other the questions? All right. Now you're experts at NP-hardness of origami.

So here's the no equal set clause gadget. And if we fold it here with all of the incoming signals the same direction, in the center they collide. You can't go all the way to flat. This thing is not yet folded. And it's stuck in the center. But if I flip one of them, I'll flip this guy, then it very happily folds flat. Because you don't get that collision because the center ends up going off to the side. And you can check that for all three, it's actually symmetric. But no matter how these guys are set, if they're all equal, you get collision in the center. If they're not all equal, it folds flat.