PROFESSOR: All right, are you guys ready for some more hardness proofs? Let's do it. So today, we start the universe of 3 partition. And in general, we're going to see today 2 plus problems, which are useful for dealing with numbers for reducing to number problems. Numbers here are going to be integers for the most part. Usually in complexity, it's a lot easier to think of all your numbers being integers. If they're not integers, use fixed point or rationals, and then they're basically integers.

So we're going to talk. My favorite is 3 partition. And pretty much all the proofs we'll do in the number context will be reduction from 3 partition. This is all about NP-hardness, I should say. But before we get to e partition, I'm going to talk about another problem, which is called 2 partition.

First, I'll just have to define all these problems, which will take a little while. Then we'll get to all the fun reductions. So in 2 partition, your given n integers. We'll call the set of them capital A and we'll call the integers themselves little a sub i-- i from 1 to n, let's say. And what we'd like to do is partition them into two groups. Let's say A1 and A2 equals A of equal sum. We want the sum of the integers in A1 to equal the sum of the integers in A2. And therefore, they will equal one half the total sum.

OK, so we just want to find our choice for each number. Should it go in A1 or A2? So this problem is NP-hard. And it's one of the original Karp problems. There's this famous paper by Karp in 1972. Before that, there was a paper by Cook that proved SAT is NP-complete. And then Karp wrote a paper with a zillion-- not literally, but dozens of NP-hardness proofs from SAT to things like 2 partition. I will not cover the proof here. If you're interested in it, read Garey and Johnson. There's some nice coverage of this problem and all the problems we'll be talking about today.

OK. As a sideline, I'll mention there is a generalization of this problem. So it's also NP-hard. Any generalization is going to be even harder. But it's useful to know about. It's called subset sum. Occasionally it's easier to think about subset sum instead of partition, although it's a generalization. So if you can reduce from subset sum, you can also reduce from partition.

And here the set up is similar. You're given n integers again, just like before. And you're given a target sum. This is one other integer. We'll call the target sum t. I'll also call this t. And what we'd like to do is find a subset S of A whose sum is t. OK, this is a strict generalization here. You're not given a number t, but you're essentially choosing a subset A1, whose sum equals the sum of A over 2.

So this is really just a particular choice of t. This is the general case where you're given t. So this is harder than that. But it's worth knowing about. In particular, there's a problem set problem, the problem set will be out in a couple of hours. And it's a little easier to think about subset sum, we think. You don't have to.

All right, so let's go to 3 partition, my favorite NP-complete problem. So start is the same, given integers A1 up to An. We're not given a target sum. What we'd like to do is partition them-- partition that set A into how many pieces?

AUDIENCE: 3.

PROFESSOR: 3. Nope. n over 3. Kind of like 3. Not quite. Let's say sets A1, n over 3 equals A. I'm using union with a dot to indicate it's disjoint union. So that's what a partition means. That's redundant. This is old fashioned notation. You probably haven't seen it, but when I was growing up, that was the notation of the day.

Then we have n over 3 sets. Cool, yeah, assume n is divisible by 3. Otherwise the answer is no. And then this is the easy case of equal sum. So we want the sum of each Ai to be equal to t, which is to the same thing. They're just going to be the sum of the A's divided by n over 3.

OK, if this problem were about partitioning to 3 sets, it would not be very interesting. It would be basically identical to 2 partition. With n over 3 sets, things are quite a bit

different. And we will get to that difference in a moment. But first, let me tell you about some fun facts about 3 partition.

So we're partitioning into n over 3 sets, so you would expect, on average, most of these sets will have size 3. In fact, you could make all the sets have size exactly 3, because if you look at the reduction for this problem, you can assume that each Ai is strictly between t over 4 and t over 2. OK, if you're going to have a set of numbers that's summed to exactly t and each of them is strictly bigger than t over 4, then that means you can't have 4 of them because you would sum to too much. And they're strictly less than t over 2. And you can't just have 2 of them and hope to get to t. So you have to have exactly 3 in every set.

You don't have to assume that. In some situations, it's useful to just say, well, the Ai's are sum sets that sum to the right things. Sometimes it useful to assume that they're triples. It's up to you whether you consider that part of the problem statement, to ask for the Ai's to have size 3 or not. And it's useful to have both.

In fact, once you have this, you can make all the Ai's very, very close to t over 3 using a trick, which we'll see a bunch of times today. So let's say you could add-- I'll call it infinity-- to each Ai. In that case, of course, any solution that you had before is still a solution. Just think of infinity is a really large number, something like n to the 100 times maximum item in A. Something like that. That's really maybe to the 100th power, if you want, but a really big number.

This will preserve all solutions, because all solutions, every set has size exactly 3. So if you had 3 times infinity, they'll still have the same sums. And when you do that, then all the Ai's are basically equal to each other, which means they're roughly equal to t over 3. The leading term is the same for everybody. And then there's this little noise, which was the original integer that you're adding onto, that infinity. So then all the adds are roughly t over 3. Cool.

Let me tell you about a couple related problems to 3 partition. And then I'll tell you why we're even bothering to talk about two problems instead of just one. So next problem has a funny name, especially when you don't know what 3 dimensional

matching is, but we will get to that second. Don't worry about the name for the moment. This is a closely related problem. It's really a specialization of 3 partition.

In this case, we're going to be given 3 sets of integers. And what we'd like to do is-- it's basically the same problem. Just think of taking the union of those sets, but each of these sets that we're choosing, the Ai's, should consist of one element from A, one element from B, one element from c. So how did I formalize that? Let's say a partition. This is a disjoint union into triples, Si, and cross product ABC. So you're choosing one from each of these guys in each Si, and they should be of equal sum.

And so the sum will be sum of A plus the sum of B plus sum of C divided by n , I think, because there's 3 of them. Cool. OK, so almost the same problem, I just changed what n was by a factor of 3. That's why there's n over 3 here and then here. But here, we're specializing in a sense to say that we're giving different types of numbers-- the red integers, the green integers, and the blue integers. You have to choose one of each color.

OK, for fun, let me show you why this problem is, in some sense, simpler. This also gives you a useful technique for playing with these number problems. So I want to reduce from this numerical 3 dimensional matching problem, which you might also call the ABC version of 3 partition. I will reduce it to 3 partition. So I'm given an instance, these numbers. So the basic idea is to take these numbers just throw them into one big set, A prime. But I want to force this ABC property that I choose one from each. Maybe any suggestions how? Yeah.

AUDIENCE: Well, I just have a question. You define these all as sets. You can't have duplicates of an integer?

PROFESSOR: I didn't say the word set. I actually meant multiset. Yeah, so this is a multiset. Good question. All of them are multisets. So it makes the notion of disunion a little bit weird. But you know what I mean. Yeah.

AUDIENCE: Could you add a really big number to all the B's and twice that really big number to all the C's?

PROFESSOR: OK, add a-- hadn't thought about this. So add a really big number to all the B's, and then twice that big number to all the C's. And then the target sum would be three times the big number, plus--

AUDIENCE: I'm sorry, so make it three instead, so you can't do three things from each side.

PROFESSOR: Yeah, right. So you don't want take two B's and then that looks like a C. So maybe you want three times a big number here and nine times a big number here. Something like that. I think actually one times a big number and three times a big number would do it. Yeah, cool.

So the way I did it in the notes was adding a small number, but either way, it works. Let's do it with the big number, because that's fun. So we will construct, I'll call it A prime to consist of a overall little a and big A. And then we'll say a plus-- usually big number I write with a capital B, but that's not very good because we already have a couple of B's. So I'll call it X. Or I should call it infinity for consistency.

So for each b and B, I will add infinity to it. And then c is going to be is 3 times-- let's do 4 times infinity. It makes me feel a little safer somehow, because we're constructing a set of size 3. So if you take 3 of these, I worry that looks like a C. It's not enough, but we'll be safe. Why not? We can choose pretty big numbers here. So happy? No?

AUDIENCE: You might be able to do a couple A's in a single set. The set should be small, because they're not-- [INAUDIBLE].

PROFESSOR: OK. a is our problem. You could maybe do more than 3 total. So let's do 1 infinity, then get a proper eraser. It's a good thing we have Sarah here to correct our proofs. 3 infinity 9.

AUDIENCE: I think it's enough, because then we need--

PROFESSOR: 6 would be bad, because then 2 of these would be 1 of those.

AUDIENCE: 13 infinities on [INAUDIBLE].

PROFESSOR: Right, so this means the new target sum should be the old target sum, which was the sum of these divided by n, plus 13 infinity.

AUDIENCE: Isn't infinity a function of the n's?

PROFESSOR: Infinity I should define. Let's make it, I think, just like 10 times max of A ought to do it. You could be it as a function of n. I don't think that's necessary here, because all of our sets are size 3. But if you're doing something like this for 2 partition, you'd want to multiply by n or n squared. This is a common trick to do this kind of scaling or offsets or things like that. Instead of infinity here, you can write epsilon, where epsilon is much smaller than 1. I think that would also work. And that's what's done in this.

AUDIENCE: So we just make it the max of the A as opposed to A prime?

PROFESSOR: Oops, this should probably be max of ABC. Yeah, you can't make it max of A prime. That would be cyclic. That infinity would actually be infinite. Good. So I think that works. There are things to check here, obviously. We need to check that whenever we have a solution to the original instance, we have a solution to this instance. That's pretty clear. You just take the same choices of the Ai's, Bi's, and Ci's.

And the infinities will always add up to 13. The tricky part is to show that if you have a solution to this 3 partition instance, you can convert that back into a solution to the numerical 3 dimensional matching instance. And that's where you essentially want to show that the infinities can be treated algebraically. Because they're so huge, you could never construct an infinity in another way.

But I'll leave that as an exercise to actually proof, because we're not really going to use this problem too much. You're welcome to, maybe on problem sets, maybe on open problems it would be good. But most the time, I would say we use 3 partition. I want you to know about this in particular, because it's related to another problem, which is called 3 dimensional matching. Yes?

AUDIENCE: Is the assumption that the Ai is all a cardinality of 3 built into the problem

specification here?

PROFESSOR: You get a choice. You can either make this part of the problem specification or not. It won't matter.

AUDIENCE: OK, well I think in this case, if you don't know the problem specs, can you take 12 A's for something? And then you get 12 infinities and then they add up to another infinity?

PROFESSOR: Yes, we definitely want to take this as part of the specification for this reduction. Yeah. Good point. OK, so here's a problem. You've probably heard of the matching problem. Let's say the perfect matching problem is you're given a graph and you want to find a set of edges which are disjoint, meaning no two edges share a vertex, that covers all the vertices. So if you have n vertices, you'd like to find n over 2 edges that are disjoint from each other. That's perfect matching. You might call it 2 dimensional matching, because an edge has 2 ends.

3 dimensional matching takes place in a hyper graph. And in this case, it's going to be a tripartite hyper graph. You may not have seen hyper graphs before. They're just like graphs, but edges can have more than 2 endpoints. In this case, they will have exactly 3. So let's say the vertices are A disjoining in B disjoining in C. And let's say they all have like size n. And the hyper edge is E.

OK, maybe it's worth drawing a picture. So you have this set A, set B-- they're all the same size-- set C. And an edge might look like this or like this or so on. OK, you take one from each. And then your goal is to find n disjoint edges. If we find that many and they're disjoint, that means they will cover all of the vertices. Every vertex will be hit by exactly one hyper edge. OK, so this is the 3 dimensional version of bipartite matching, I guess. There's also the non-bipartite version.

For extra completeness, there's something called exact cover by 3 sets, 3 meaning the cardinality. This is called X3C. You do see 3DM and X3C frequently in the literature and especially in Garey and Johnson. So they're good problems to know about. Here is sort of the generalized version, where you're given non-tripartite, just

a 3 uniform hyper graph. 3 uniform just means every edge has cardinality 3. And your goal is to find-- let's say it has n vertices-- n over 3 disjoin hyper edges.

OK, many times the almost same problem. How do these relate? Well, certainly if you take a numerical 3 dimensional matching problem, that is a 3 dimensional matching problem. You can convert it into a graph like this. You're just forgetting the fact that there's numbers. But then when you draw the edge, you draw one of these hyper edges exactly when Ai plus Bi plus Ci equals the targets sum.

So numerical 3 dimensional matching is a special case of 3 dimensional matching. Occasionally it's useful to start from 3 dimensional matching. Not for number problems though. If you have a number problem, you like the numerical version. Personally I prefer 3 partition, but these two are pretty close. We saw one connection between them.

Exact cover by 3 sets is, in some sense, even more general than 3 dimensional matching. Yeah, it is more general, because tripartite hyper graph like this is 3 uniform. So this is a strict generalization of that. But they all appear somewhat, so I thought it'd good for you to know about them. Again, 3 partition is where the action is.

All right, let me go back to this issue of 2 partition versus 3 partition. These are the two main problems I want to talk about. And they relate to an important distinction in NP-hardness, which is weak versus strong NP-hardness. So there are two types of NP-hardness for number problems. If your input is a graph, these two notions are the same, so you don't need to worry about it. But when your problem has integers as input, then there are two types.

Weakly NP-hard is, in some sense, the type you already know, if you think about it in the right way, in the intended way. We haven't been super formal about how numbers getting encoded, but usually we think of NP-hardness in terms of how hard your problem is in terms of the encoding size, which we usually call n, the encoding of your input.

OK, so in the usual form, what this means is that you allow numbers to have exponential value, because even when their value is exponential, the encoding of that value is polynomial. Coding length of such a number is, if you read it in binary, that's log base 2 of the number if the number is 2 to the n to the c, which is our definition of exponential. That's going to be n to the c. And that's polynomial.

OK, for here, I'm going to let n be the number of numbers to avoid circular definitions or something. So I'm saying there are n different numbers. If each of them is at most 2 to the n to the c in value and they're integers, then I can encode them in this many bits. And that's a polynomial number of bits. So I consider that a reasonable encoding.

And sometimes that's just fine. But there's a stronger notion, which is really useful, especially for a lot of problems we will see today and next class, which is called strongly NP-hard. And this is NP-hard even when your problem is restricted to numbers of polynomial value. So polynomial here is with respect to n and is the number of integers-- number of numbers.

OK, so strongly NP-hard is stronger than weakly NP-hard. If you're NP-hard even when your numbers are only polynomial in size, then of course your NP-hard when you allow the numbers to be as large as exponential. This is a weaker constraint on the problem.

So to cut to the chase, 2 partition up here and subset sum are weakly NP-hard. And if you believe that the P does not equal NP, then it's not strongly NP-hard. That's the best we can hope for, whereas 3 partition in every other problem we talked about is strongly NP-hard.

AUDIENCE: I have a question. What do you mean exactly by numbers having exponential value?

PROFESSOR: So again, exponential is like 2 to the n to the c. n is the number of numbers in your input.

AUDIENCE: OK.

PROFESSOR: Yeah. So that's what I allow. I mean, this is what would be allowed if you didn't say anything, if you interpret complexity theory in a way that I haven't necessarily told you, but generally you want to assume that all your inputs are reasonably encoded in an efficient way that most possible input strings are valid inputs. And so if you're encoding, then in particular, you should encode your numbers in binary or ternary or quaternary or hexadecimal-- anything bigger than one is good. OK? And you should know if you're writing that number, don't use unary.

But today we're going to use unary a lot. So this is the same thing as saying encode in unary. Strong NP-hardness is like saying, well, my problem is so hard that even if I encode my numbers in unary, it's still hard in the resulting input size. The input size would be if you write down every number in unary and add up all the numbers. That's your input size.

3-partition is that hard. 2-partition is not. OK? So if you can prove strong NP-hardness, you should. It's better. If you can only prove weak NP-hardness, it's OK. It's still hard.

Now you should know, there are corresponding notions on the algorithm side. So let's talk about that a little bit.

So corresponding algorithms are-- there are actually three of them. The main ones are pseudopolynomial and weakly polynomial. Sorry. The weaklies are unrelated to each other, but that's life. There's also a notion called strongly polynomial-- unrelated to strongly NP-hard. Well, there you go. OK.

So weakly polynomial is the usual notion of polynomial I should mention. When I say I have a polynomial time algorithm, what you mean is weakly, meaning that, let's say, I'll write it as polynomial in n-- the number of integers-- and let's say, log of the largest integer-- largest number in the input. OK? That's a reasonable encoding size. You write them in binary. You add them all up. So you'd multiply that by n or whatever. And the polynomial in that value is weakly polynomial.

Pseudopolynomial you omit the log. OK. So I want polynomial on n and the largest

number.

And strongly polynomial, you omit that term altogether. So you just polynomial n the number of numbers. OK? So there's a different world of algorithms which we're not going to talk about here, which is weakly polynomial algorithms versus strongly polynomial algorithms. I'm not aware of any lower bounds about that, but there are famous open questions like, can you solve linear programming in strongly polynomial time? We don't know. There's no hardness notion there. So we're not going to talk about it any more than that.

What we care about more is this distinction between pseudopolynomial. This is polynomial when the numbers are written in unary, and this polynomial when the numbers are written in binary. So these two notions correspond to these two notions. So in particular, if you believe p does not equal NP, then weakly NP-hard means there is no weakly polynomial algorithm. But there might be a pseudopolynomial algorithm.

Strongly NP-hard means there's no M polynomial algorithm, and so therefore there's no weakly polynomial algorithm.

Adam.

AUDIENCE: So the difference between weakly polynomial and strongly polynomial is that we don't even necessarily give ourselves time to read all the numbers and put it in binary?

PROFESSOR: I didn't want to get too much into models of computation, but the model of computation here is that you're allowed to read an entire number in one time step. So if you've taken 6851 Advanced Data Structures, this is like a word RAM. So that's the only reason this would be reasonable. Otherwise you couldn't read the input. That would be pretty hard.

So the model is you could take a constant number of your numbers, do something with them, output another number of the same type roughly in constant time. And

you want to do a polynomial number of such steps, usually arithmetic operations.

Here you don't have to be so careful about the model because any reasonable encoding, you could read it and do whatever you want with it. If you don't care about the exponent in the polynomial, like all models are equivalent. If you believe the Strong Church [INAUDIBLE] Thesis-- anyway. This one you have to be a little bit more careful what the model is.

Good. So let me draw my favorite diagram with the difficulty axis. But now instead of p, I'm going to distinguish between weakly polynomial, pseudopolynomial, and strongly polynomial. So these things are pseudopoly. These things are weakly poly. And these things are strongly poly. This is on the algorithm side, upper bounds-- be on the left of the particular position. And then on the hardness side we have-- let me use red. It's more dramatic. This thing is weakly NP-hard. This is if p does not equal NP then you're strictly to the right of pseudopoly. So there's no pseudopoly problem. There's also weakly NP-hard.

And then strongly NP-hard excludes even a weakly polynomial algorithm, so strictly to the right of that position. So obviously, strongly NP-hard is a better result because it's smaller than this class.

But for example, 2-partition, which is only weakly NP-hard, has a pseudopolynomial algorithm. You've probably even seen one in your algorithms class if you've ever done anything like Knapsack as a dynamic program. That is a dynamic program probably originally done for a 2-partition.

OK. So that problem is somewhere right in here. Weakly NP-hard-- whoops.

AUDIENCE: I think the diagram that's is backward, but I don't--

AUDIENCE: Yeah. Diagram is--

AUDIENCE: If a problem is strongly polynomial it ends [INAUDIBLE] a weakly polynomial.

PROFESSOR: So the middle one's right. Thank you. That seemed funny. But strongly polynomial is this tightest class. That's the best result. And pseudopoly is the worst result. Did I

get my arrows in the right place?

AUDIENCE: You've got to shift over.

PROFESSOR: Shift over one. Strongly NP-hard. Whoops. So weakly NP-hard is here. Strongly NP-hard is there. OK. Cool.

So let's check. So 2-partition should be pseudopoly, so left of this line, but weakly NP-hard still right of that line. That looks good. So this is 2-partition. 3-partition is out here. I mean, I'd draw this is a region, but it's more like a point in my old diagram because 3-partitions are actually E and NP. So it's strongly NP-complete. So it's like right at the edge there.

Anyway, there's no pseudopolynomial algorithm for 3-partition, unlike 2-partition. That's the point of all that.

Let's do some. Shall we? NP-hardcore time. So that'll be over here. OK. The first reduction is going to be pretty--

AUDIENCE: May I ask a question before we move on?

PROFESSOR: Yeah.

AUDIENCE: So why is there a no corresponding notion for problems are not numeric? Like why can't for a graph problem we say, well, we're going to examine this graph problem assuming we're using an inefficient encoding of our graph to see how hard it is?

PROFESSOR: I guess you could try to define this in terms for graphs, but I don't think there is such an obvious inefficient notion of graphs. I mean, the obvious inefficient notion is like use a matrix instead of an adjacency list. But that's polynomially equivalent. So I mean there certainly are other notions you could imagine, but there aren't any in the literature.

All right. So just to warn you, the first reduction we do is going to be a little bit trivial. So don't get too excited yet. But this is where 3-partition comes from. Then we'll do lots of cool ones.

So multiprocessor scheduling. Let's say I have n jobs, and they have completion times. And that's it. You can start them whenever you want. And then on the other hand, I have p processors to run those jobs. Each processor is sequential and identical. So the model is you assign jobs to processors. Job can only be on one processor, and it runs to completion. Then it can run another job, and so on.

My goal-- I'll state the decision version-- is to finish all jobs in time less than or equal to t. OK? So I claim this problem is NP-hard of some variety. Let's start with a reduction from partition. Sorry. Partition is a synonym for 2-partition. Usually we omit the two. So reduction from partition. This problem over here.

So now I'm given some integers. I want to split them in half basically. Any suggestions on what I could do to turn that into a multiprocessor scheduling problem?

AUDIENCE: Two processors.

PROFESSOR: Two processors. Yeah. p equals 2. That is the reduction. I guess, technically, I also have to specify what t is, but it's over there. So you set t to be the sum of the A's over 2. And you don't change A. You just leave it alone. So you just feed in the same A i's into this problem. And that is 2-partition.

OK. Let's do a reduction from-- so now we know the problem is at least weakly NP-hard, but in fact, it is strongly NP-hard, because we can reduce from 3-partition. What do you do?

AUDIENCE: P equals n over 3.

PROFESSOR: P equals n over 3. That's it. You set t accordingly. Sum of the A's over p. Done. OK. So this problem comes from the paper that introduces 3-partition, which is by Gary and Johnson, the same people who wrote the book. So, I mean, this basically is 3-partition. That's why they cared about it. But here's a practical motivation for it.

Cool. Let's do something more interesting. And I kind of want a projector. Any

questions about-- I think you've asked enough questions about pseudopoly, weakly poly, strongly poly, as I erase them. Don't forget them.

But I think for the rest of today we're just going to be doing strong polynomial reductions from 3-partition. I think next class we might do some weak ones, but today we're going to really use the fact that it-- use the strongness. And you'll see why, because we really want to encode some numbers in unary.

OK. Let's start with a problem-- rectangle packing. And this is going to lead us to a whole bunch of other types of packing problems. So this is what you might call packing rectangles into a rectangle. So I'm given n rectangles, and I'm given a target rectangle. So when I say I'm given a rectangle, what I really mean is I'm given the dimensions-- the width and the height-- as integers. We are usually going to assume all our numbers are integers, as I said. Do they fit?

So what I'd like, in a solution, is to rotate and translate the given rectangles so that they're disjointed from each other, they can't overlap, and they all fit inside the target rectangle. All right. So this might be the target rectangle. And then, I don't know, maybe this is my set of rectangles appropriately packed. So here maybe the answer is yes. If they don't fit inside the target rectangle, the answer is no.

This problem is strongly NP-hard, but I'm pretty sure we don't know whether it's E and NP. So there are lots of open problems. I don't mean to indicate everything is known. I should double check that this is definitely not known, but I'm pretty sure it's open. Because of this rotation issue there are kind of a lot of things to worry about. So it's not clear how efficiently you can encode a solution. Maybe you need a ton of bits of precision to say how things are rotated. Here they're all rotated at the same angle. They may not be. Who knows.

So we will consider a special case, which is exact packing. This also makes for a stronger hardness result. So exact packing means there are no gaps. None of this stuff. That's forbidden. So if you have no gaps and you look at a corner of the rectangle, that must mean there is a rectangle right there filling the corner, which means there's no rotation. Well, you could rotate by multiples of 90. But just know

there are four rotations. Well, I guess two for rectangles-- integer times 90 degrees.

So then the problem is an NP, because also then you can show at that point-- because you have integer rectangles packing into an integer rectangle-- by induction, the translations are integers. Therefore, there are succinct encodings of them.

OK. We don't have to do this, but just to point out, this is one way to make the problem in E and NP. Now they are assisting solutions.

So we're going to prove both of these problems are strongly NP-hard. I think I'll skip the weak NP-hardness proof because it's not really any harder-- no pun unintended. So let's go there.

So the basic idea again, we're reducing from 3-partition. So that means we're given, on any instance of 3-partition, n integers. We want to represent those integers as rectangles. So kind of the obvious way to do that is we take each a i and we convert it into a rectangle. Let's say it's a i by 1. That seems pretty obvious. This is what you might call the a i gadget, since we're going to map this feature of our given instance to this feature of our constructed instance.

All that would be left at this point is what is the target rectangle, which I didn't give a name. So let me call the target rectangle B. I think that's what I used in my notes. Yep. So let's say B is going to be a rectangle. What is the height of the rectangle?

AUDIENCE: n over 3.

PROFESSOR: n over 3. Yes. Right. And then the width?

AUDIENCE: The sum divided by n over 3.

PROFESSOR: The target sum, t. Cool. So the intent is that the rectangles will look something like this. And always, however, if I fit three in here-- or maybe I could do more than three, but we know that won't happen with the actual 3-partition instances we get, because we can assume there are always three of them-- but anyway, we're not constraining there are only three. You put in however many rectangles. Add up to

exactly the target sum. Boom. You've got the target sum. And there's just enough space for all of them if you pack them this way.

Question?

AUDIENCE: Shouldn't we specify that the a i rectangle is a i times n over 3 plus 1 to make sure that we can't accidentally rotate them?

PROFESSOR: Yeah. So part of my confusion in drawing this-- and I worry-- is you could take one of these rectangles and rotate it. And then this reduction is wrong. So this reduction is wrong. So I will cheat and make this epsilon, and then multiply this by epsilon. This is one way to fix it. Of course, now I have to scale everything up by 1 over epsilon. And I have to tell you what epsilon is. You were suggesting some multiple of n, which is a good suggestion.

I think as long as we make this much smaller than 1, then it's not possible to rotate, because these guys are length at least 1, because they're integers. So if I tried to rotate, it has to be a multiple 90 degrees. So I think as long as this is strictly less than 1, we are OK. I didn't change the intended solution. I just made everything narrower. And so I just need to set epsilon less than 3 over n. It would be that inequality.

But I want integers, so I should scale everything up by n over 3. So just multiply all the numbers by n over 3, and you're done.

In fact, you can be a little simpler. And I drew this diagram long ago, so I might as well show it to you. You can just add n over 3 to all your integers. That will also work, under the assumption that you can only take three numbers and add them up. Then you will always be adding 3 times n over 3 horizontally. So they a i's here are the yellow part of these rectangles, and then we're just adding this fixed n over 3 to all of them. So this is a 4, a 4, a 4, a 4, a 5, 5, 5, 6, 7. And that happens to be a positive instance of 3-partition. There is a solution. This is one way to do it.

So we're just adding n overall here, because we have 3 times n over 3. And the

length of these is always smaller. Rotation should not be possible, because each of these guys is going to be length strictly larger than n over 3 because we add an integer to n over 3-- a positive integer-- and the height here is n over 3. So that just barely works. Same thing, more or less, just this is scaling by a multiplicative factor here. We're doing an additive thing, because we have this luxury of knowing there are three items in every triple. Cool. Rectangle packet.

Question.

AUDIENCE: This is a question about 3-partition. Is it assuming that they are positive integers?

PROFESSOR: Yes. All the integers are positive. You don't have to, but we can. So we will. Makes our life a lot easier. Other questions? OK. Time for some puzzles. So I think I'll show you some real life puzzles first. There's this notion of an edge-matching puzzle, which goes back to the 1890s. I believe this one is from the late 1890s, early 1900s. So it's one of the older ones. At least when I was growing up, we had these puzzle all over the place.

So what's going on here is you have physical triangles, and these half frogs are drawn on the edges of the triangles. And your goal is to pack all of these triangles into the big triangle. That's easy. It's a regular triangulation. But you need to half frogs to match up. I don't know. The hind end of the white-- I'm sorry-- yellow-speckled frog matches the front end of the yellow-black-speckled frog, as drawn here. This is a valid solution, but if I rotated this tile 120 degrees, it's not a valid solution even though it's all hinds matching to fronts and so on. OK? Rules clear?

So these are edge-matching puzzles. This is formulated in a different way. This is not a valid solution. The goal is to get these colors to all match up. So in general, you can think of there being a color on an edge. And here it happens to be composed of these two colors. And it would have to match both of the colors on the other side. So it's essentially like having a color on each edge.

Here, we also have a color on each edge, but we also have a sign, which is the front or the hind part of the frog. There's the yellow frog, the red frog, the green frog, the

orange frog. So in general, we're going to formulate this. I'll get over here.

OK. There are many types of these puzzles I'll try to capture. It's not exactly either one of those. If you've ever studied something called Wang tiles, this is exactly that model. So I'll just throw in the word there. But what we have are n unit squares. Those are called the tiles. Each one has four colors-- a, b, c, d. So one per edge.

And you are also given a target rectangle. It's going to be integer, because these are unit squares. And you want to pack-- probably exactly pack so there are not going be any gaps-- the squares into the rectangle with colors matching. So whenever I have two squares that share an edge, the color that's written here better be the same color that's written there. Those are the rules. If you're on the boundary, no restriction. But if you're an interior edge, you restrict in that way. OK?

This actually closely matches what is currently maybe the most famous edge-matching puzzle called Eternity II. This one is still unsolved. Well, the creator has a solution. And if you ever find it, or one of the many other solutions-- who knows-- you win $2 million US. So time to get rich.

AUDIENCE: [INAUDIBLE].

PROFESSOR: Well, yeah. I guess that's the way to make the most money is first solve p equals NP. And then solve Eternity II, I guess. But you could easily solve this puzzle without p equals NP.

There are actually a lot of almost solutions out there. I believe this is one of the leaders. This guy, Louis Verhaad, got-- so this is a complete solution with all the pieces, but not all the edges match. So it's hard to find them, but here's one, for example, that mismatches. So I didn't explain, but everything is a square tile here, and there's a different shape and color. But it's basically a color from a mathematical perspective. Have to match them all up. Here he got 467 matches out of the 480 edges. And that won him already $10,000. And I believe it's found with a computer-aided search.

AUDIENCE: Is that--

AUDIENCE: [INTERPOSING VOICES].

AUDIENCE: That's not the person who--

PROFESSOR: I don't know for sure. I would guess that it's Louis's daughter. Here's a different almost solution where there are some missing tiles but everything matches. And then at the bottom of his web page, you should check it out. It's funny. It says, "And here is a solution to Eternity II." I looked at this for a long time, and I was like I don't see any edge matches, but you guys are smart, so you probably know there's one other cheat you could possibly do in this puzzle.

AUDIENCE: Add Different tiles?

PROFESSOR: Add different tiles? Maybe. Although these are nicely-printed tiles. So I would guess he bought two sets and just mixed some parts. I wrote a program to check yes, indeed, some tiles are repeated here. I forget exactly how many. 10 or 20 or something. It's still pretty cool. It looks like it's done.

AUDIENCE: [INAUDIBLE].

PROFESSOR: But there's no complete actual solution using each piece exactly once. It's probably clear here, but you only get one version of each square.

OK. So there's your motivation for solving edge-matching. So you might wonder, oh, can I solve it in polynomial time? And the answer is no unless p equals NP. And here we're going to use essentially rectangle packing and this reduction. So how could I-- again, we're going to reduce from 3-partition. How could I convert a number a i into something like a rectangle using edge-matching? I should probably copy over the one that we had in the end, which was a i goes to n over 3 plus a i by 1, because we want integers.

Yeah.

AUDIENCE: So you make it so that there is sort of one color, and that's for [INAUDIBLE] most things. And that's [INAUDIBLE] kind of on the outside of each rectangle. And then

put together each rectangle. If you need like-- if the rectangle is like a i by 1, you're going to need a i squares, such that there's like one color for that rectangle that is only used to join the other squares of that rectangle.

PROFESSOR: Good. So we're going to convert this something by one rectangle into something different squares. And we're just going to sub-divide like that. And we're going to surround all these edges with the same color. That will work.

AUDIENCE: And the interior edges [INAUDIBLE].

PROFESSOR: The interior edges-- all right. So we're going to take a i, and we're going to convert it into-- OK. Don't make me draw all of them-- the width here, let's say, will be n over 3 plus a i. And we're going to use a special color-- I'll call it black-- on the outside. And this color I'm going to call-- I need some red so it's slightly visible-- I'm going to call i. This is going to be a different color for each a i.

This will require a little bit more work. You could also use a different-- you could use i1, i2, i3-- but I would really force this to go together in this way. If I'm trying to be conservative on colors-- only use n of them-- then I will put those all i. And then as long as none of these matched to the boundary, I'm OK. I guess even if I used the i1, i2, i3, it's possible some of those match the boundary. And then weird stuff could happen. But as long as all these i colors match up with each other, you have to build this rectangle. And then we don't care about how the rectangles are assembled with respect to each other. So we're basically simulating this reduction rectangle packing.

OK. The one I have drawn here is slightly different. I don't remember why we did it this way. This is [INAUDIBLE] work with Marty Dumaine, the cameramen. And here, just I guess to be as simple as possible, we made the width of each of the rectangles just a i. That has a problem that you could rotate. But to prevent rotation, we use a different color horizontal and vertical. So the top edges and the bottom edges all have the colored percent sign. I think this is a limitation of the drawing program I was using at the time actually. And dollar signs on the left and right side.

And then we're going to build frame. This is what you might call infrastructure for the proof. This is one gadget happens. You just execute it once. This we execute for each of the a i's. We're just going to have one copy of this big frame. The goal the frame is to hide the boundary so that none of these tiles could match with the boundary, because when they match the boundary then you're breaking a rectangle in half and bad things could happen.

So instead, what we're going to do is all of these capital letters on the outside-- and I ran out of letters and used a couple of symbols-- appear exactly once in the reduction. So they can't match anyone, which means they have to be on the boundary. So those guys have to be on the boundary. And then all these other colors appear exactly twice, so you have to join these tiles together in exactly this way. And now the boundary is gone. And we have set up dollar signs on the left side and the right side, and percent signs on the top and the bottom sides. In reality, these would be whatever t wide and n over 3 tall. This is just some different notation. And that's it. Now you're simulating rectangle packing.

Question?

AUDIENCE: Why do you need two rows at the bottom?

PROFESSOR: So this two rows is to indicate, for whatever reason, because in edge-matching puzzles, usually the target shape is not a rectangle, but a square. And we wanted the stronger result at packing unit squares into a square with edge-matching results E as NP-hard. So the point here is you just add however many rows or columns you need to make it a square. Yep. Good question.

Jason?

AUDIENCE: So your different symbols, I believe, are also to let you build this into the rectangle later on. Otherwise, you'll run into issues with rotating things.

PROFESSOR: We'll get there. Yeah. So I have different symbols here and here in order to prevent rotation. That's right. Instead of making them longer. I could've, instead, made these longer like we had with the rectangle thing, but I guess, at the time, I wasn't

thinking of teaching a class about this. So I didn't reuse my previous reduction for this one. But oh, well. Yeah.

But the reason there are dollar signs in there like red or some actual color is because probably black and white printing was more common then. OK. Cool. That was one puzzle.

And now key point. What if we did this reduction from partition? So with partition, instead of having n over three groups here, we just have two groups? And same thing. Why not do that?

AUDIENCE: So that would be a weakly NP-hard.

PROFESSOR: That would be a weak NP-hard reduction. Now, this problem here-- edge-matching puzzle, n unit squares of four colors-- are there any numbers in this problem?

AUDIENCE: 4

PROFESSOR: Other than 4 and n? Are there any numbers that are inputs?

AUDIENCE: I guess the colors have to be represented as numbers.

PROFESSOR: The colors do you have to be represented as numbers, but this is sort of a sideline. Usually, when we're talking about colors, all we care about is whether the colors are equal or not. So you don't normally think of those as numbers. You can think of them as vertices in a graph or something. I mean, they're more like combinatorial objects. You're never adding two colors together.

Yes?

AUDIENCE: The other point is that the number of different colors that you can get is at most 4 times n because [INAUDIBLE].

PROFESSOR: The number of colors is in most 4 times n. Yeah, you could write down a color as 2 to 2 to the n, but why? It's supposed to be an efficient encoding. So there are really no numbers here. So it doesn't even make sense to say this problem is strongly NP-

hard.

But even worse, if I started with 2-partition, and I did this reduction exactly as drawn but not as tall, I could still make it a square, but everything would break. Because for 2-partition to be hard, those numbers-- the a i's-- have to be exponential in value. If I have an exponential value a i, when I convert it into a rectangle, or actually I convert it into a bunch of tiles, I will have exponentially many tiles. This construction will be of exponential size. So I will get exponentially many different colors on the boundary. So that's very bad.

Even if you're clever and you say, oh, well, these tiles are all the same, so maybe I'll encode them efficiently and say there's an exponentially many of this one tile. The ones in the boundary are all different. So it's not a valid reduction. You should always take a problem that's supposed to run in polynomial time and produce a result of polynomial size.

So if I start with something where the input is polynomial size but it has exponentially large a i's, I'd get an exponential size puzzle, which is boring to solve. I don't know. It's not a valid NP-hardness reduction. This is why we want to start from 3-partition. Because then I know even if what I'm doing here is representing the a i's in unary. I take the a i and said, OK, I'm going to have that many square tiles. But that is exactly unary representation. And we know with 3-partition that will have polynomial size. And so we're good. This is why we care about strong NP-hardness so much, because a lot of puzzles you can only represent numbers in unary. Then you're forced to work with 3-partition.

OK. Next puzzle is signed edge-matching puzzles like the lizards. Suppose same set up, but now the colors aren't supposed to be equal. They're supposed to be opposites. So I'm just going to modify this diagram, and I will use capital letters to denote tails of lizards, and lowercase letters to represent heads of lizards. And little b only matches capital B. It does not match little b. You can't put two heads together or two tails together.

Well, you can reduce these things, which are called unsigned edge-matching

puzzles, to these things, which are called signed edge-matching puzzles, with this easy gadget. If you have a tile with, let's just say, colors ABCD-- some of them might be equal-- I'm going to replace that one tile with these four tiles and scale everything up by a factor of 2-- scale the target shape by a factor of 2. These colors appear exactly once with each sign, and so again, because there's a frame and all the boundary is eaten up, you're forced to match these guys to each other. And therefore you're forced to make this 2 by 2 tile.

And so that 2 by 2 tile is supposed to represent this 1 by 1 tile. And the claim is signs no longer matter, because we put little a, big A, little b, big B, little c big C, and so on. So if I took another tile like over here, and it has something like capital D, lowercase d, that's going to match with this lowercase b lowercase b, if and only if b equals d.

So if the original colors are equal, the resulting colors will be equal and opposite. So it'll be the same letter of the alphabet, but with opposite signs, and so everything will match up, because this capital letter match here, and this lowercase letter will match there. Is that clear? So it really becomes the same puzzle, and this one's going to be solvable only if this one is. So we get strong-- well, just NP-hardness of signed edge-matching puzzles. Cool.

Next puzzle. These were all on the same paper. It was a fun paper. This is even older. Jigsaw puzzles. This is not the oldest jigsaw puzzles. It's one of the newest. It's our course poster turned into a jigsaw puzzle. If you want to make your own jigsaw puzzles, go there. They go back to the 1760s when they were made with jigsaws, hence the name. But basically, we have square pieces, and each side of the-- I'll assume that's a special kind of jigsaw puzzle, most common-- each edge can either be straight. That means it's on the boundary. It can be a pocked or a tab. And each pocket and tab is slightly differently shaped, although maybe not on this picture, usually they are.

I'm going to allow that some of the pockets and the tabs have the same shape. That's what you might call ambiguous mates. Most jigsaw puzzles sold in the world

do not have ambiguous mates, but let's say they do. And to make it NP-hardcore, there is no image on the pieces. It's just a white puzzle. Now those are sold. You can buy all white or are all single color puzzles. But usually they don't have ambiguous mates. That's like the ultimate combo. If you don't assume either of those things, then the problem is easy. So we don't care about that. But our paper is about how to solve jigsaw puzzles in the practical case.

But in our situation, I mean, that basically is edge-matching puzzles with science. This is actually probably why I care about signed puzzles the most. I have lowercase letters. Those are going to correspond to pockets. I have uppercase letters. Those are going to correspond to tabs. And of course, a tab must meet a pocket, same as a lowercase letter must be an uppercase letter. And so that is the same thing.

I would like my target shape to be a square because that's clean. So whenever I have a unique color, which is one that I know has to be on the boundary, I'm going to make it a flat thing. And then I know in this series of reductions, I should draw this series of reductions, because it's kind of fun.

So far I kind of started with rectangle packing. I guess technically we started with 3-partition. I'll draw it like this. We reduce that to edge-matching puzzles. Then we reduce that to signed edge-matching puzzles. Then we reduced that to jigsaw puzzles. And if you follow this series of reductions and see what you get, we know that the boundary shape will be a square because that frame gadget is a square. And if we make all the unique things flat, then the shape will actually be square.

There's a reason I'm drawing it this way, because we have another one. So the next puzzle-- you might ask well, if Eternity II was an edge-matching puzzle, what was Eternity I? Well, it's something called a polyform packing puzzle. These are-- if you take an equilateral triangle and cut in half like that, that's a shape. If you take k copies of that shape, join them edge to edge along matching edge lengths, you can get all these shapes. It's like Tetris, but instead of the squares you start with that shape and you join a bunch more of them together. I don't know exactly how many.

They may not even be all the same area.

And your goal is to fit inside this big dodecagon or something. This was solved. One million pounds were acquired by two people. I think they were mathematicians, Alex and Oliver, in 2000. So actually quite soon after the puzzle was posed, I think within less than a year. So there were articles about how the insurance company was not happy. That's why you get insured against a puzzle being solved, I guess. But in Eternity II, as a consequence, it is much harder. So no one had solved it yet, but it will happen someday I'm sure. So this gives you a clear sense of what the shapes look like.

So I want to prove these puzzles are NP-hard also by reduction from jigsaw puzzles. So let's just bring up the picture. So we just did this reduction. It actually works in both ways. They're really the same puzzle. This does not work so easily in both ways. But I'm going to take a jigsaw piece. I'm going to turn it into roughly a square made up of little squares. I'm not going to work on the half triangles, although I think it wouldn't be any harder. It's a lot easier to think about square puzzles more like Tetris. These are polyominos.

So polyomino packing is this type of puzzle. They also exist, but no one has won one million pounds from them. So that's why I showed Eternity. So almost a square, but then in the middle of the edge we're going to write in binary what this shape was here. I mean, if there was a color over here and a shape here, all we care about is whether they're the same or different. So there's only 4n different colors out there. So we're going to take log of 4n bits and write them in this way.

And if it was indented, whenever we have a 1, we're going to indent here. And if it's a tab instead of a pocket, wherever we have a 1, we're going to outdent. OK. So these are all different codes. And so none of them fit together, for example. And let's see. Does anything match up?

AUDIENCE: Yes.

PROFESSOR: This one matches with this one, because if I rotate it around, stick it on, it will match

up. So you have to make sure you get the orientations to the codes all right, but you can do this. And two things will fit together and make a nice clean seem, if and only if the original colors matched. And so this represents that puzzle where we've blown everything up by like a logarithmic factor in each dimension. But the number of pieces is staying the same, in a sense.

So that was this reduction to polyomino packing. For fun, we close the loop, and reduce from polyomino packing to edge-matching in this way. So we're going to take our polyomino, carve it up into little squares, turn them into edge-matching puzzle. And here I actually want to allow rotation, so I'm going to change these colors to all percent signs. And now it's just the same puzzle again. So we have to use the frame again.

And so if you want, you could just keep following these reductions over and over and over. I think every time you do it, you increase the scale factor by like a log factor. But it's kind of fun. I mean, it really shows these puzzles are very close to identical in that we didn't have to blow up the puzzle size by very much to go from one puzzle to the next. So usually we're just thinking about polynomial blow up here. We have a very small notion of blowup, which is some tight sense in which these puzzles are almost the same.

OK. End of paper.

A fun open problem is this reduction used log n by log n-- order log n by log n pieces. What if your pieces only have area log n? So maybe they're root log n by root log n or something that. I'm sure it should still be in NP-hard, but you never know. For example, they're constant size. Actually, well, let's say, if they're size 2, its easy. Leave it at that even if the shape is very messy. But if there are also 2 by 2 blocks it's easy. So clearly small shapes are too easy. But probably logarithmic area is where it starts to get interesting. I don't know. What about log, log area? That's probably neither NP-hard nor polynomial, but oh, well. Cool.

So I want to talk about one more problem-- one more hardness proof in this similar vein. It's about packing. I already erased packing rectangles into rectangles or

rectangles into a square. What about squares into a square? I don't know if you can tell if that's a square and not a rectangle.

OK. This is a nice paper. We've used it for other hardness proofs, which we'll probably talk about next class. So I'm given a bunch of squares, and I'm given a target square. Do these squares fit in there? And again, we will probably use exact-- well, initially we won't be exact packing. But let's say that the squares can only rotate by a multiple of 90. In other words, they can't rotate. We will fix that by making exact packing.

Why don't I jump to the reduction? The heart of the reduction is this idea-- we're just going to add a big number here. I am going to use capital B for the big number. That's what's in the paper. This is a paper from 1990. Motivation here is you have a big grid supercomputer, which were popular in the '80s and '90s. And you have a job, which was a square grid of processors, let's say. And you want to execute them all in parallel, but they're not allowed to overlap, because that would slow your supercomputing down. But you are allowed gaps, let's say.

So I'm just going to take each of my a i numbers, add this huge number to it. So they're all roughly the same size. This is the trick we did at the beginning of class with the 3-partition. All right. All these numbers-- all these sizes are almost the same. I've drawn them quite different, but imagine they're just like tiny fluctuations.

And then I make the height of this rectangle. For whatever reason I transposed my image. That's probably why I was confused earlier. And I just make the height three times the big number-- because I want three pieces to come together-- plus t, my target sum of the a i's. So this is a huge thing plus a super tiny thing. But what it's saying is well, you could have slop. They're almost the same size, but there's that little bit of extra. And that little bit of extra has to add up to exactly t. So this is one triple with the desired target sum. And then I want to get n over three of them.

Now here's where things get a little dicey. You say, OK, let's just say I take the largest one, or actually I'll go all the way up to B plus t. I know that every number I have is smaller than t, because I just saw the sum of three of them is t. So I'm

always going to have a gap here. And let's say I just packed the next three there and then the next three there. And so I'll leave enough room that even with the slop I could fit everything in.

So each one is B plus t, and the number of them is n over 3. So that'll be the width of my rectangle. Looks like a lot of slop, but that's because I drew this with the squares very different in size. If we compute the amount of slop, the maximum, if you look at this distance, it's at most t, certainly. Even if a i were 0, which is not allowed, then this distance would be t. So it's going to be even smaller than that. But the gaps here are always at most t.

So we can take-- so this is total slop is t times the height. So that was like the width of one sliver of slop, and then the height is 3B plus t. And then times n over 3, because there's n over 3 of those things. And hopefully this works. I want to say this is less than B squared if B is bigger than t times n. 3s cancel. So if B is, I think this term won't matter, because B is going to be bigger than t. So maybe put a 2 there for extra safety. Then you're totally dwarfed. So just dividing both sides by B we're getting 3 times t times n over part 3. That's 2 times n. OK.

Cool. So the point is the total area of slop that I have here is less than a single square, because a single square is even bigger than this. So you're not going to be able to pack anymore squares, and that's a vague argument, but enough hand-waving later, and you've proved that, in fact, any packing of these squares will give you 3-partition. That requires more work, but this is some intuition why it is true.

OK. So this was actually packing squares into a rectangle. But that wasn't the problem I started with. I wanted to pack squares into a square. So for that you need an infrastructure gadget like the frame that we made before. And this is kind of acute number theory-ish trick. I'm going to take some integer x. We'll choose it later. It's just going to be big enough-- something like n. And we have one giant square, which is x times x plus 1 minus 1. But they're all squares, so by the same amount.

And then we're going to take these medium-sized squares of size x plus 1 and the small squares-- so much smaller-- of size x. And because this is just one smaller

than what it's supposed to be-- I mean, if it was x times x plus 1 it would sort of come to here. So when you pack these guys up you go just one too far there. But there's exactly enough room for one more. If I put x plus one extra, this is x. That's the one, and so on. So this leaves, as a gap, a rectangle.

Now this is not the only solution these guys could slide down, but you can show that will only make your life harder. And the paper does that. That's sort of the easy part of the proof. And so now you're left with a rectangle, which is 1 by this thing. You multiply everything by the 3B plus t, because that was what we wanted one of our dimensions to be-- the smaller one. So that scaled up the rectangle.

And then the horizontal-- the width may not match exactly. But if there's too much space here, and let me just set x to be large enough, then there will be extra space. And just throw in a bunch of B by B blocks there to fill up the extra space. So you can figure out how much extra space there is and fill it in. So they you go-- packing squares into a square.

And this is strong NP-hardness of this. Even when the squares are only of polynomial area, this problem is hard, which is actually the case you care about when each of these 1 by 1 squares is a processor. Now this is not exact packing, and so you might wonder oh, what if I rotate the squares, and then crazy things could happen. Probably not, but you could also just compute how much extra slop there is, because you know the area of the input squares, and you know the area of the target square. And just add a whole bunch of 1 by 1 little chips, and that will fill in exactly the space. Then you have exact packing. Then you don't even have to worry about rotation. So that makes proof even cleaner. I think that wasn't in the original paper, but it's a nice little addition. And that is our packing through 3-partition.

Next class we'll do more 3-partition for different types of problems. So in case you lose lots of pieces in your jigsaw puzzle, we have the one-piece jigsaw puzzle. This is all the pieces are in one giant connected mess. And still it is not so trivial to solve. Wow.

AUDIENCE: [INTERPOSING VOICES]

PROFESSOR: All right. This is the one-piece jigsaw puzzle in solve state. It's the Simpson's addition. It's-- you get it.

AUDIENCE: [INTERPOSING VOICES]

AUDIENCE: And then you put that in. And then you put it in your slop.

AUDIENCE: Yep. [INAUDIBLE].

PROFESSOR: Ta-da.