PROFESSOR: All right, today we are going to learn to count. One, two, three-- In a algorithmic sense of course, and prove hardness of counting style problems. Or more generally, any problem where the output is an integer. Like approximation algorithms, we need to define a slightly stronger version of our NP style problem. It's not going to be a decision problem.

The remaining type is a search problem. Search problem you're trying to find a solution. This is just like optimization problem, but with no objective function.

Today all solutions are created equal. So we just want to know how many there are. So we're given an instance, the problem. We want to produce a solution. Whereas in the decision problem we want to know whether a solution exists.

With a search problem we want to find a solution. Almost the same thing.

So that would be a search problem in general. For an NP search problem you can recognize what is an instance. And you could recognize solutions to instances in polynomial time. Given an instance in the solution you can say, yes that's a valid solution.

OK for such a search problem of course, is the corresponding decision problem. Which is, does there exist a solution? If you can solve this, then you can solve that. You could solve the decision problem in NP by guessing a solution and so on. So this is intricately related to the notion of a certificate for an NP problem. The idea solutions are certificates.

But when we say problem is in NP, we say there is some way to define certificate so that this kind of problem can be set up. And the goal here-- the point here is to

solidify a specific notion of certificate. We can't just use any one, because we're going to count them the-- If you formulate the certificates in different ways you'll get different counts. But in general, every NP problem can be converted into an NP search problem in at least one way. But each notional certificate gives you a notion of the search problem.

OK, in some complexity context these are called NP relations, the way you specify what a certificate is. But I think this is the more algorithmic perspective. All right, so given such a search problem we turn it into a counting problem.

So that's a search problem, is called A. Then counting problem will be sharp A, not hashtag A. Some people call it number A. The problem is, count the number of solutions for a given instance.

OK so in particular, you detect whether the number is zero, or not zero. So this is strictly harder in some sense than the decision problem, does there exist a solution. And we will see some problems where the search problem is polynomial, but the corresponding counting problem is actually hard. I can't say NP hard, there's going to be a new notion of hardness.

So some examples. Pretty much every problem we've defined as a decision problem had a search problem in mind. So something like SAT, you need to satisfy all of the things. So there's no objective function here, but you want to know how many different ways, how many different variable assignments are there that satisfy the given formula?

Or take your favorite pencil and paper puzzle, we'll be looking at Shakashaka today, again. How many different solutions are there? You'd like, when designing a puzzle, usually you want to know that it's unique. So it'd be nice if you could count the number of solutions and show that it's one. These problems are going to turn out to be very hard of course.

So let's define a notion of hardness. Sharp P is going to be the class of all of these counting problems. This is the sort of certificate. Yeah, question?

AUDIENCE: Just for the puzzle application, is it going to turn out that counting if there's one solution versus one-to-one solution is as hard as just counting total numbers?

PROFESSOR: It's not quite as hard, but we will show that distinguishing one for more than one is very hard, is NP complete actually. That's a decision problem that could show that's NP complete. So normally we think of zero versus one, but it turns out one versus two is not-- or one versus more than one is about the same difficulty. Counting is even harder I would say. But it's bad news all around. There's different notions of bad.

Cool, so this is the sort of certificate perspective. With NP we-- I had given you two different definitions of certificate perspective, and a non-deterministic computation perspective. You can do the same computational perspective here. You could say sharp P is the set of problems, solved by polynomial time. Call it a non-deterministic counting algorithm. We don't need Turing machines for this definition, although that was of course the original definition.

So take your favorite non-deterministic algorithm as usual for NP, it makes guesses at various points, multiple branches. With the NP algorithm the way we'd execute it on an NP style machine is that we would see whether there's any path that led to a yes. Again, it's going to output yes or no at the end.

In this case, what the computer does, the sharp P style computer, is it-- conceptually it runs all the branches. It counts the number of yeses and returns that number. So even though the algorithm is designed to return yes or no, when it executes it actually outputs a number. The original paper says magically. It's just as magic as an NP machine, but a little-- even a little more magical.

OK so you-- I mean if you're not comfortable with that we just use this definition, same thing. This is all work done by Les Valiant in the late '70s. These notions.

So it's pretty clear, it's pretty easy to show all these problems are in sharp P, because they were-- the corresponding decision problems were in NP. We convert them into sharp P algorithms. Now let's think about hardness with respect to sharp

P.

As usual we want it to mean as hard as all problems in that class. Meaning that we can reduce all those problems to our problem. And the question is, by what kind of reductions? And here we're going to allow very powerful reductions. We've talked about these reductions but never actually been allowed to use them.

Multicall, Cook-style reductions. I think in general, this is a common approach for FNP, which is Functions NP style functions. Which you can also think of this as kind of-- the counting version is, the output is a value. So you have a function instead of just a decision question. When the output is some thing, some number, in this case. You might have to manipulate that number at the end. And so at the very least, you need to make a call and do some stuff before you return your modified answer in your reduction.

But in fact we're going to allow full tilt, you could make multiple calls to some hypothetical solution to your problem in order to solve all problems in sharp P. And we'll actually use multicall a bunch of times. We won't always need multicall. Often we'll be able to get away with a much simpler kind of reduction. Let me tell you that kind now. But in general we allow arbitrary multicall. Yeah?

AUDIENCE: Your not limited in the number of multicalls?

PROFESSOR: Right. You could do polynomial number of multicalls. As before, reduction should be polynomial time. But, so you're basically given an algorithm. That's usually called an Oracle that solves your problem, solves your problem B. And you want to solve-- yeah, you want to solve A by multiple calls to B. So what, we'll see a bunch of examples of that.

Here's a more familiar style of reduction. And often, for a lot of problems we can get away with this. But especially a lot of the early proofs needed the multicall. And as you'll see you can do lots of cool tricks with multicall using number theory.

So parsimonious reduction. This is for NP search problems. This is going to be a lot like an NP reduction, regular style. So again we convert an instance x of a, five

function f, into an instance x prime of b. And that function should be computable in poly time. So far just like an NP reduction. And usually we would say, for a search problem, we would say there exists a solution for x, if and only if, there exists a solution for x prime. That would be the direct analog of an NP style reduction.

But we're going to ask for a stronger condition. Which is that the number of solutions to problem A to instance x, equals the number of solutions of type B to instance x prime. OK so in particular, this one's going to equal to one, this one will be greater than equal to one, and vice versa. So this is stronger than an NP style reduction for the corresponding decision problem.

Yeah, so I even wrote that down. This implies that the decision problems have the same answer. So in particular, this implies that we have an NP reduction. So in particular if A, the decision version of A is NP hard, then the decision version of B is NP hard. But more interesting is that, if A is sharp P hard, then B is sharp P hard. But also this holds for NP.

For the decision versions of A and B. Sorry, sharp A and sharp B. OK this is a subtle distinction. For sharp P hardness we're talking about the counting problems, and we're talking about making calls to other counting solutions. Then doing things with those numbers and who knows what, making many calls. With parsimonious reduction we're thinking about the non-counting version. Just the search problem. And so we're not worried about counting solutions directly,

I mean it-- what's nice about parsimonious reductions is they look just like NP reductions for the regular old problems. We just need this extra property, parsimony that the number of solutions to the unit is preserved through the transformation. And a lot of the proofs that we've covered follow-- have this property and will be good for us. If we can get our source problems hard, then we'll get a lot of target problems hard as well.

Well let me tell you about one more version which I made up. C-monious reductions. This is my attempt at understanding the entomology of parsimonious. Which is something like little money, being very thrifty. So this is having a little bit

more money. You have C dollars. But you have to be very consistent about it. I should probably add some word that means uniform in the middle there.

But I want the number of solutions of x prime to equal c times the number of solutions to x. C a fixed constant. And it has to be the same for every x. This would be just as good from a sharp P perspective. Because if I could solve B and I wanted to solve A, I would convert A to B, run the thing, then divide by C. There will never be a remainder and then I have my answer to A. As long as C's not zero. C should be an integer here. We will see a bunch of C-monious reductions. I guess, yeah it doesn't have to be totally independent of x. It can depend on things like n. Something that we can compute easily I guess. Shouldn't be too dependent on x.

All right, let's do-- let's look at some examples. So, going to make a list here of sharp P complete problems. And we'll start with versions of SAT because we like SAT. So I'm just going to tell you that sharp 3SAT is hard. First sharp SAT is hard, and the usual proof of SAT hardness shows sharp P completeness for sharp SAT. And if you're careful about the conversion from SAT to 3CNF you can get sharp three satisfied. It's not terribly interesting and tedious. So I will skip that one.

So what about Planar 3SAT? I stared at this diagram many times for a while. This is lecture seven for replacing a crossing in a 3SAT thing with this picture. And all of this argument and this table in particular, was concluding that the variables are forced in this scenario.

If you know what A and B are-- so once you choose what A and B are, these two have to be copies of A and B and then it ended up that A-- We proved that A1 equaled A2 and B1 equaled B2, and then these variables were all determined by these formula. And so once you know A and B all the variable settings are forced. Which means you preserve the number of solutions. So planar sharp 3SAT is sharp P complete.

I'd like to pretend that there's some debate within the sharp P community about whether the sharp P are here or here. I kind of prefer it here, but I've seen it over here, so you know. I don't think I've even seen that mentioned but I'm sure it's out

there somewhere.

So let's flip through our other planar hardness proofs. This is planar monotone rectilinear 3SAT. Mainly the monotone aspect. We wanted there to be all the variables to be positive or negative in every clause. And so we had this trick for forcing this thing to be not equal to this thing. Basically copying the variable with a flip in its truths. But again everything here is forced. The variables we make are guaranteed to be copies or indications of other variables. So we preserve number of solutions. So this is hard too.

OK, what else? I think we can-- I just made this up --but we can add the dash three as usual by replacing each variable with little cycle. What about planar 1-in-3 SAT? So we had, we actually had two ways to prove this. This is one of the reductions and we check that this set of SAT clauses, these are the SAT clauses implemented this 1-in-3 SAT clause. And I stared at this for a while and its kind of hard to tell.

So I just wrote a program to enumerate all cases and found there's exactly one case where this is not parsimonious. And that's when this is false and these two are true. And because of these negations you can either solve the internal things like this, or you could flip all of the internal nodes and that will also be satisfied.

Now this is bad news, because all the other cases there is a unique solution over here. But in this case there are exactly two solutions. If it was two everywhere we'd be happy, that would be twomonious. If it was one everywhere we'd be happier, happy that would be parsimonious. But because it's a mixture of one and two we have approximately preserved the counts with a factor of two. But that's not good enough for sharp P, we need exact preservation. So this is no good.

Luckily we had another proof which was actually a stronger result, Planar Positive Rectilinear 1-in-3SAT. This was a version with no negation, and this one does work. There's first of all this, the not equal gadget and the equal gadget. I don't want to go through them, but again A forced to be zero, C was forced to be one, which forces B and D to be zero in this picture. So all is good. And again parsimonious there.

And then this one was too complicated to think about, so I again wrote a program to check-- try all the cases and every choice of XYZ over here that's satisfied, this is a reduction from 3SAT. So if at least one of these is true, there will be exactly one solution over here. And just as before after-- if zero of them are true, then they'll be no solution here. That we already knew. So good news. I should check whether that's mentioned in their paper but it proves planar positive rectilinear 1-in-3SAT is sharp P complete.

Sharp go here, here. OK, so lots of fun results. We get a lot of results just from-- by looking at old proofs. Now they're not all going to work, but I have one more example that does work. Shakashaka, remember the puzzle? It's a Nikoli puzzle. Every square, every white squared can be filled in with a black thing, but adjacent to it too, there should be exactly two of those. And you want all of the resulting white regions to be rectangles possibly rotated.

And we had this reduction from planar 3SAT, and this basically, this type of wire. And there's exactly two ways to solve a wire. One for true, one for false. So once you know what the variable is, you're forced what to do, there is also a parity-shifting gadget and splits and turns. But again, exactly two ways to solve everything. So parsimonious, and then the clause everything was basically forced just-- you're forced whether to have these square diamonds.

And you just eliminated the one case where the clause is not satisfied. So there's really no flexibility here, one way to solve it. And so it's a parsimonious reduction and indeed in the paper we mentioned this implies sharp P completeness of counting Shakashaka solutions.

Cool. Here's an example that doesn't work. A little different, Hamiltonicity or I guess I want to count the number of Hamiltonian cycles. The natural counting version of Hamiltonicity.

So we had two proofs for this, neither of them work in a sharp P sense. This one, remember the idea was that you would traverse back and forth one way or the other to get all of these nodes. That was a variable, that's fine. There're exactly two ways

to do that. But then the clause, the clause had to be satisfied by at least one of the three variables. And if it's satisfied for example, by all three variables, then it could be this node is picked up like this, or the node could be picked up this way, or the node could be picked up this way.

So there are three different solutions even for one fixed variable assignment. So that's bad, we're not allowed to do that. It'd be fine if every one was three, but some will be one, some will be two, some will be three. That's going to be some weird product of those over the clauses. So that doesn't work.

We had this other proof. This was a notation for this gadget which forced either this directed edge or this directed edge to be used, but not both. It's and x or. So that's, remember what these things meant and that we have the variable true or false. And then we connected them to the clauses, then separately we had a crossover.

But the trouble is in the clauses, because again, the idea was that the variable chose this guy, this one was forbidden. That's actually the good case I think. If the variable chose this, chose this one, then this one must be included. That's bad news, if you followed this, and then this, and then this, then you cut off this part of the graph and you don't get one Hamiltonian cycle. You want at least one variable to allow you to go left here and then you can go and grab all this stuff and come back.

But again if multiple variables satisfy this thing, any one of them could grab the left rectangle. And so they get multiple solutions, not parsimonious. But parts of this proof are useful and they are used to make a parsimonious proof.

So the part that was useful is this x or gadget, and the way to implement crossovers. So just remember that you can build x ors, and that you can cross them over using a bunch of x ors. So only x or connections, these notations, can be crossed in this view.

We're going to build more gadgets and this is a proof by Sato in 2002. It was a bachelor's thesis actually in Japan. And so here you see redrawn the x or gadget.

Here it's going to be for undirected graph, same structure works. And this is do the crossover using x or. So he's denoting x or's as this big X connected to two things.

Now given that we can build an or gadget, which says that either we use this edge or we use this edge, or both. Here we're not using an x or, but this is the graph. And the key is this has to be done uniquely. That's in particular the point of these dots. This looks asymmetric, it's kind of weird. For example, if they're both in, then this guy can do this, and this guy can do that.

But it's not symmetric. You couldn't flip-- this guy can't grab these points, that would be a second solution which would be bad, but we missed these points. So this guy has to stay down here if he's in at all. And then this guy's the only one who can grab those extra points. Or if just the top guy's in then you do this. And if just the bottom guy's in I think it's symmetric. That is symmetric, and it's unique. Good.

If there's an implication-- so this says if this edge is in, then that edge must be in the Hamiltonian cycle and this is essentially by copying. And we just have to grab an extra edge and add this little extra thing just for copying value. So if this one is in, then this edge must not be used, which means this edge if it's used must go straight. In particular, this is not used. And we have an or that means that this one must be forced by a property of or.

On the other hand, if this is not set, this one must be used. So I guess this must be an edge that was already going to be used for something. So that edge is just going to get diverted down here, and then the or doesn't constrain us at all. Because zero or one, this one is one so the or is happy. So that's an implication. It's going to be a little more subtle how we combine these.

This is the tricky gadget. I sort of understand it, but it has a lot of details to check, especially on the uniqueness front. But this is a three-way or which we're using for clause, SAT clause, 3SAT clause. We want at least one of these three edges to be in the Hamiltonian cycle. And so here we use x ors, ors, implications, and more x ors. I'll show you the intended solution, assuming I can remember it.

So let's say for example, this edge is in the Hamiltonian cycle. Then we're going to do something like go over here, come back around like this. And then

AUDIENCE: Did you just violate the x or already?

PROFESSOR: This x or?

AUDIENCE: Yeah.

PROFESSOR: No, I went here.

AUDIENCE: And then you went up.

PROFESSOR: Oh, then I went up, good. So actually I have to do this and around. Yes, so all these constraints are thrown in, basically to force it to be unique. Without them you could-- still it would still work, but it wouldn't be parsimonious.

OK, let's see, this--

AUDIENCE: In the middle.

PROFESSOR: --doesn't constrain me. This one?

AUDIENCE: Yeah.

PROFESSOR: Good, go up there. So I did exactly one of those and then I need to grab these. OK, so that's one picture. I think it's not totally symmetric. Again, so you have to check all three of them. And you have to check for example, It's not so hard.

Like if this guy was also in, I could have just gone here and this guy would pick up those nodes. So as long as, in general, as long as at least one of them is on you're OK. And furthermore, if they're all on, there's still a unique way to solve it. And I'm not going to go through that. But it's thanks to all of these constraints they're cutting out multiple solutions.

OK, so now we just had to put these together, this is pretty easy. It's pretty much like the old proof. Again, we have-- we represent variables by having these double

edges in a Hamiltonian cycle. You're going to choose one or the other. And then we have this exclusive or, forcing y and y bar to choose opposite choices. And then there are these exclusive words to say, if you chose that one, then you can't choose it for this clause. And then the clauses are just represented by those three-way ors.

So this overall structure is the same. The crossovers are done as before and it's really these gadgets that have changed. And they're complicated, but it's parsimonious. So with a little more work. We get the number of Hamiltonian cycles in planar max degree three graphs is sharp P complete.

So that's nice. And from that, we get Slitherlink. So this is-- I've sort of been hiding these facts from you. When we originally covered these proofs, these papers actually talk about-- This one doesn't quite talk about sharp P, but it is also sharp P complete. Counting the number of solutions to Slitherlink is sharp P complete.

This was the puzzle. Again you have that number of adjacent edges-- each number. And the proof was a reduction from planar max degree three Hamiltonian cycle. And at the time I said, oh you could just assume it's a grid graph. And then you just need the required gadget, which is the b part. Just need this gadget. This was a gadget because it had these ones. It meant it had to be traversed like a regular vertex in Hamiltonian cycle and it turns out there was a way to traverse it straight or with returns.

And then we could block off edges wherever there wasn't supposed to be an edge. And so if you're reducing from Hamiltonicity and grid graphs that was the whole proof and we were happy. Now we don't know whether Hamiltonicity and grid graphs is sharp P complete. To prove that we would need to be able to put bipartite in here, and I don't know of a proof of that. Good open problem.

So this was the reason that they have this other gadget, which was to make Hamiltonian-- to make these white vertices that don't have to be traversed. They're just implementing an edge basically. So just the black vertices have to be traversed. We needed that for drawing things on the grid. But if you just don't put in the ones and add in these zeros again you can traverse it or not all.

And this one, as you can read here, says this would be bad to have happen. In fact you can rule it out, it will never happen in the situations we want. Because the white vertices only have two neighbors that are traversable.

Cool, and then furthermore, all of these solutions are unique. There is exactly one way to go straight. There's exactly one way to turn right, exactly one way to turn left. That's the subtle thing that was not revealed before. But if you stare at it long enough you will be convinced.

So this is a parsimonious reduction from planar max degree three Hamiltonian cycle to Slitherlink. So counting solutions in Slitherlink is hard. That was the fully worked out example. Any questions? Yeah.

AUDIENCE: So are there examples of problems in P who's counting versions are sharp P complete?

PROFESSOR: Yes. And that will be the next topic. Well it's going to be a little while til we get there. But I'm going to proof things and then we will get to an answer, but the answer is yes.

AUDIENCE: Question.

PROFESSOR: Yeah.

AUDIENCE: So, for if we wanted-- if we're like thinking about a problem that we're trying to prove something P-hard and we start thinking maybe stop, we would just show it's in P by finding algorithm. Is there a nice way to show that our problem is not P hard?

PROFESSOR: Well you usually would say that sharp, that problem is NP. You find a polynomial counting algorithm. And they're lot's of examples of polynomial counting algorithms especially on like trees. Typical thing is your dynamic program.

So like maybe you want to know-- let's say you have a rooted binary tree and for each node you could flip it this way or not. How many different ways are there to do that? And maybe have some constraints on how that's done. Then you just try it

flipped, and try it not. You do dynamic programming and then you multiply the two solution sizes together, and you get the overall solution size.

So you basically do combinatorics and if there's independent choices you multiply, if they're opposing choices you add, that kind of thing. And from that you get polynomial time counting algorithms. In tree-like things that often works inbound treewidth.

AUDIENCE: Do you know that NP hard problems who's counting problems are not sharp P hard? I guess this technique wouldn't work.

PROFESSOR: I would say generally most problems that are hard to decide are hard to count. And where NP hard implies sharp P hard. I don't think there's a hard theorem in that there's nothing that really says-- meta-theorem that says that, but that's the feeling. It'd be nice, then we wouldn't have to do all the parsimonious work.

All right so it's time for a little bit of linear algebra. Let me remind you, I guess linear algebra's not a pre-req for this class, but probably you've seen the determinant of a matrix and use, if it's zero then it's non-invertible blah, blah, blah.

Let me remind you of a definition. And we rarely use matrix notation. So let me remind you of the usual one. N by n, square matrix. This is a polynomial time problem. It is-- but I'm going to define it in an exponential way. But you probably know a polynomial time algorithm. This is not an algorithms class so you don't need to know it. But it's based on Gaussian elimination, the usual one.

So you look at all permutation matrices, all n by n permutation matrices which you can think of as a permutation pi on the numbers 1 through n. And you look at i comma pi event, that defines the permutation matrix. You take the product of the matrix values-- if you superimpose the permutation matrix on that, on the given matrix A.

You take that product you possibly negate it if the sign of your permutation was negative, if it does an even number-- an odd number of transpositions then this will be negative. Otherwise, it'd be positive and you add those up.

So of course, an exponential number permutations you wouldn't want to do this as an algorithm but turns out it can be done in polynomial time. The reason for talking about this is by analogy I want the notion of permanent, of an n by n matrix. The same thing, but with this removed. The determinant of A is the sum over all permutations pi, of the product from i equals one to n of ai, pi of i.

Now this may not look like a counting problem. It turns out it is a counting problem, sort of, a weighted counting problem. We will get back to counting problems in a moment. This Is related to the number of perfect matchings in a graph. But at this point it's just-- it's a quantity we want to compute. This is a function of a matrix. And computing this function is sharp P complete. Yeah.

AUDIENCE: Could it just be sine minus 1 2e sine of pi?

PROFESSOR: I don't know, it depends. If you call the number of transpositions mod two so then it's zero one. You know what I mean.

All right. So the claim is, permanent is sharp P complete. We're going to prove this. This was the original problem proved sharp P complete. Well other than sharp 3SAT I guess. Same paper.

Great so let me give you a little intuition of what the permanent is. We'd like a definition that's not so algebraic. At least I would like one more graph-theoretic would be nice. So here's what we're going to do. We're going to convert A, our matrix, into a weighted graph. And then let me go to the other board.

How do we convert into a weighted graph, weighted directed graph? Well the weight from, vertex I to vertex J is AIJ. The obvious transformation if it's zero then there's not going to be an edge, although it doesn't matter.

You could leave the edge in with weight zero, it will be the same. Because what we're interested in, in the claim is the permanent of the matrix equals the sum, of the product of edge weights, over all cycle covers of the graph. OK, this is really just the same thing, but a little bit easier to think about.

So a cycle cover it's kind of like a Hamiltonian cycle but there could be multiple cycles. So at every vertex you should enter and leave. And so you have sort of in degree one and out degree one. So you end up decomposing the graph into vertex destroyed cycles which hit every vertex. That's a cycle cover. Important note here, we do have loops in the graph. We can't have loops if aii is not zero. Then you have a loop.

So the idea is to look at every cycle cover and just take the product of the edge weights that are edges in the cycles and then add that up overall cycle covers. So it's the same thing if you stare at it long enough because you're going from I. I mean this is basically the cycle decomposition of the permutation if you know permutation theory. So if you don't, don't worry this is your definition of permanent.

So we're going to prove this problem, is sharp P complete? And we're going to prove it by a C-monious reduction from sharp 3SAT.

AUDIENCE: You said this is just the original thing introducing that?

PROFESSOR: Yes.

AUDIENCE: Did they not call it-- you made up the term C-monious. What did they call it?

PROFESSOR: I think they just called it a reduction. I think pretty sure, they just called it reduction. And for them-- and they said at the beginning, --reduction means multicall reduction. So they're thinking about that. But it turns out to be a C-monious reduction. C will not be literally constant, but it will be a function of the problem sets. And this is the reduction.

So as usual, we have a clause gadget, and a variable gadget, and then there's this shaded thing which is this matrix, which you can think of as a graph. But in this case will be easier to just think of as a black box matrix.

OK all of the edges in these pictures have weight one. And then these edges are special, and here you have some loops in the center. No one else has a loop. So the high-level idea is if you're thinking of a cycle cover in a vertex because you--

sorry, in a variable because you've got a vertex here and a vertex here you have to cover them somehow. And the intent is that you either cover this one this way, or you cover this one that way, those would be the true and the false.

And then from the clouds perspective we need to understand, so then these things are connected. This thing would go here and this thing would go here, and generally connect variables to clauses in the obvious way. And in general, for every occurrence of the positive form and the negative form, sorry positive or negative form you'll have one of these blobs that connects to the clause. So overall architecture should be clear.

What does this gadget do? It has some nifty properties let me write them down. So this matrix is called x in the paper. So first of all, permanent of x equals zero. I'm just going to state these, I mean you could check them by doing the computation.

So we're interested in cycle covers whose products are not zero. Otherwise they don't contribute to the sum. So I could also add in non-zero. Meaning the product is non-zero. OK so if we had a cycle cover that just-- where the cycle cover just locally solved this thing by traversing these four vertices all by themselves. Then that cycle would have permanent zero. And then the permanent of the whole cycle covers the product of those things. And so the overall thing would be zero.

So if you look at a nonzero cycle cover you can't just leave these isolated, you have to visit them. You have to enter them and leave them. Now the question is, where could you enter and leave them? This is maybe not totally clear from the drawing but, the intent is that the first vertex in-- which corresponds to row one, column one here, is the left side. And the column four, row four is the right side.

I claim that you have to enter at one of those and leave at the other. Why is that true? For a couple reasons. One is that the permanent of x with row and column one removed is zero, and so is the permanent of x with row and column four removed.

OK vertex one and four out of this x matrix are the only places you could enter and

leave. But it's possible you enter and then immediately leave. So you just touch the thing and leave. That would correspond to leaving behind a three by three sub-matrix. Either by deleting this row and column, or by deleting this row and column if you just visit this vertex and leave. Those also have permanent zero. So if you're looking at a nonzero cycle cover you can't do that. So together those mean that you enter one of them and leave at the other.

And furthermore, if you look at the permanent of x with rows and columns one and four removed, both removed, that's also zero. So the permanent of this sub-matrix is zero and therefore you can't just enter here, jump here, and leave. Which means finally you have to traverse all four vertices. You enter at one of them, traverse everything, and leave at the other.

So basically this is a forced edge. If you touch here you have to then traverse and leave there, in any cycle cover. So we're used to seeing forced edges in Hamiltonian cycle. This is sort of a stronger form of it. That's cool now one catch. So if you do that, if you enter let's say vertex one and leave at vertex four, you will end up-- your contribution to the cycle will end up being the permanent of x minus row one and column four. Or symmetrically with four and one. You'd like this to be one, but it's four.

So there are four ways to traverse the forced edge. But because it's always four, or zero, and then it doesn't contribute at all. It's always going to be four so this will be a nice uniform blow up in our number of solutions. C is not going to be four, but it's going to be four to some power. It's going to be four to the power, the number of those gadgets.

So because you can predict that, I mean in the reduction you know exactly how many there it's not depend on the solution, it's only dependent on how you built this thing. So at the end, we're going to divide by four to the power-- it's I think five times the number of clauses. So C is going to be four to the power of five times number of clauses because there are five of these gadgets per clause. So at the end we'll just divide by that and be done.

AUDIENCE: Would it be 10 times because you got it on the variable side too?

PROFESSOR: Yes 10 times, thank you.

AUDIENCE: Eight-- two of those don't actually connect to variables.

PROFESSOR: Two of them do not connect to variables, yeah eight. Eight times the number of clauses. All right so now it's just a matter of-- now that you understand what this is now you could sort of see how information is communicated.

Because if the variable they choose is the true setting it must visit these edges. And once it touches here it has to leave here and this is a edge going the wrong way. So you can't try to traverse-- from here you cannot touch the clauses down below. Once you touch here to you have to go here and then you must leave here and so on.

But you leave this one behind and it must be traversed by the clause and vice versa. If I choose this one then these must be visited by the clauses. So from the clause perspective as he said there are five of these gadgets, but only three of them are connected. So these guys are forced, and there's a bunch of edges here and it's a case analysis. So it'd be the short version. Let's just do a couple of examples.

If none of these have been traversed by the variable then the-- pretty much you have to go straight through. But then you're in trouble. Because there's no pointer back to the beginning. You can only go back this far. So if none of-- if you have to traverse all these things it's not possible with the cycle cover.

But if any one of them has been traversed. So for example, if this one has been traversed then we'll jump over it, visit these guys, jump back here, visit this guy, and jump back there. And if you check that's the only way to do it, it's unique.

And similarly any one of them has been covered, or if all three of them have been covered, or if two of them have been covered in all cases this is a unique way from the clause perspective to connect all the things. Now in reality, there's four ways to traverse each of these things. So the whole thing grows by this product but it

doesn't matter which cycle they appear in. We'll always scale the number of solutions by factor of four. So it's nice uniform scaling C-monious. And that's permanent. Pretty cool. Pretty cool proof.

Now one not so nice thing about this proof is it involves numbers negative one, zero, one, two, and three. For reasons we will see in a moment it'd be really to just use zero and one. And it turns out that's possible, but it's kind of annoying. It's nifty though. I think this will demonstrate a bunch of fun number theory things you could do.

So next goal is zero one permanent. Now the matrix is just zero's and one's. This is sharp P complete. I'll tell you the reason that is particularly interesting. As it gets rid of the weights it makes it more clearly-- makes it more clearly a counting problem.

So first of all is the number of cycle covers in the corresponding graph. No more weights it's just-- if it's a zero there's not an edge. Since you can't traverse there if it's a one you can traverse there and then every cycle cover will have product one. So it's just counting them. But it also happens to be the number of perfect matchings in a bipartite graph.

Which bipartite graph? The bipartite graph where one side of the bi-partition is the rows and the other side is the columns of the matrix. And you have an edge I j. If and only if a i sorry j is not a good not the best terminology for I-- where I here's is a vertex of v one and j is a vertex of v two. Whenever I j equals 1, if it's zero there's no edge.

It's a little confusing because the matrix is not symmetric so you might say well, how does this make an undirected graph? Because you could have and edge from i to j, but not to j to I. That's OK because I is interpreted in the left space and j is interpreted in the right space.

So the asymmetric case would be the case that there's an edge here to here, but not an edge from here to here. That's perfectly OK. This is one, two, three, for a three by three matrix we get a six vertex graph, not three vertex. That's what allows

you to be asymmetric on a loop, what we were normally thinking is a loop just means a horizontal edge.

OK so it turns out if you look at this graph, which is a different graph from what we started with, the number of perfect matchings in that graph is equal to the number of cycle covers in the other graph, in the directed graph. So this one's undirected and bipartite. This one was directed and not bipartite.

And the rough intuition is that we're just kind of pulling the vertices apart into two versions, the left version and the right version. If you imagine there being a connection from one right to one left and similarly from two right to two left directed. And three right to three left, hey it looks like cycles again. You went here, then you went around, then you went here, then you went there.

That was a cycle and similarly this is a cycle. So they're the same thing. This is cool because perfect matchings are interesting in particular perfect matchings are easy to find in polynomial time. But counting them as Sharp P complete. Getting back to that question. So that's why we care about zero one permanence or one reason to care about it. Let's prove that it's hard.

And here we'll use a number theory, pretty basic number theory. So first claim is that computing the permanent of a matrix, general matrix nonzero one modulo given integer r is hard. r has to be an input for this problem. It's not like computing at Mod three necessarily as hard, but computing at modular anything is hard. And here we're going to finally use some multi-color reduction power.

So the idea is, suppose you could solve this problem then I claim I can solve permanent. Anyone know how?

AUDIENCE: Chinese remainder theorem?

PROFESSOR: Chinese remainder theorem OK if you don't know the Chinese remainder theorem but you should know it. It's good. So the idea is we're going to set r to be all primes up to how big? Or we can stop when the product of all the primes that we've considered is bigger than the potential permanent.

And the permanent is at most m to the n times n factorial, where m is the largest absolute value in the matrix. That's one upper bound, doesn't really matter. But the point is, it is computable and if you take the logarithm it's not so big. All these numbers are at least two. All primes are at least two. So the number of primes we'll have to consider is at most log base 2 of that.

So this means the number of primes, so that's log of m to the n times n factorial, and this is roughly-- I'll put a total here. This is like m log n that's exact plus n log n that's approximate but it's minus order n so it won't hurt us. So that's good, that's a polynomial. Log m is a reasonable thing, m was given us as a number so log n is part of this input.

So as our input and n is obviously good as the dimension of the matrix. So this is a polynomial number of calls. We're going to compute the permanent mod r for all these things and our Chinese remainder theorem tells you if you know a number which is the permanent module all primes whose products--

Well if you know a number of modulo a bunch of primes then you can figure out the number of modulo, the product of primes. And if we set the product to be bigger than the number could be, then we know [? any ?] modulo that is actually knowing the number itself. So then we can reconstruct the permanent.

So here we're really using multicall. I think that's the first time I've used Chinese remainder theorem in a class that I've taught. Good stuff.

AUDIENCE: Do you know why it's called the Chinese remainder theorem?

PROFESSOR: I assume it was proved by a Chinese mathematician but I-- anyone know?

AUDIENCE: Sun Tzu.

PROFESSOR: Sun Tzu. OK I've heard of him.

AUDIENCE: Not the guy who did *Art of War.*

PROFESSOR: Oh, OK. I don't know him then. Another Sun Tzu. Cool, yeah so permanent mod r is hard. Next claim is that the zero one permanent mod r is hard. The whole reason we're going to mod r-- I mean it's interesting to know that computing permanent mod r is just as hard as permanent but it's kind of standard thing.

The reason I wanted to go to mod r was to get rid of these negative numbers. Negative numbers are annoying. Once I go to mod any r negative numbers flip around and I can figure everything is positive or non-negative. Now I can go back to gadget land.

So suppose I have an edge of weight five, this will work for any number greater than one. If It's one I don't touch it. If it's greater than one I'm going to make this thing. And the claim is there are exactly five ways to use this edge. Either you don't use it and you don't use it. But if you do use it, exactly five ways to use it. If you make this traversal, there are five ways to do it. Remember this has to be a cycle cover so we have to visit everything.

If you come up here, can't go that way gotta go straight. Can't go that way, got to go straight. Hmm, OK. So in fact, the rest of this cycle cover must be entirely within this picture. And so for example, you could say that's a cycle and then that's a cycle, these are loops. And then that's a cycle and then that's a cycle and then that's a cycle and then that's a cycle. And whoops I'm left with one vertex. So in fact I have to choose exactly one of the loops, put that in and the rest can be covered.

For example, if I choose this guy then this will be a cycle, this'll be a cycle, this will be a cycle. Perfect parity this will be a cycle, is will be a cycle, and this will be a cycle. Can't choose two loops. If I chose like these two groups then this guy would be isolated.

So you have to choose exactly one loop and there are exactly five of them. In general you make-- there'd be k of them if you want weight k, and you can simulate. If you don't use this edge then there's actually only one way to do it. You have to go all the way around like this. So that's cool. If you don't use the edge, didn't mess with anything. If you use the edge you get a weight of five. Multiplicative, because

this is independent of all the other such choices.

So you can simulate a high-weight edge, module r. In general you can simulate a non-negative weight edge like this. Just these are all weight one obviously and absent edges are zero. So we can convert in the module r setting there are no negative numbers essentially. So we can just do that simulation, everything will work mod r. So use gadget.

Now finally we can prove zero one permanent is hard. Why? Because if we could compute the zero one permanent we can compute at mod r. Just compute the permanent, take the answer mod r and you solve zero one permanent mod r. So this is what I might call a one-call reduction. It's not our usual notion of one-call reduction because we're doing stuff at the end. But I'm going to use one-call in this setting.

Just call it-- in fact we don't even have to change the input. Then we get the output when we computed mod r. Question.

AUDIENCE: So in that the weight you can have really high weight. But then your [INAUDIBLE] nodes so then when you use that wouldn't n?

PROFESSOR: Oh I see. If the weights were exponentially large this would be bad news. But they're not exponentially large because we know they're all at most three. They're between negative one and three. Oh but negative one-- Yes, OK good I need the prime number theorem. Thank you.

So we have numbers negative one then we're mapping them modulo some prime. Now negative one is actually the prime minus one. So indeed, we do get weights that are as large as r. So this is-- r here we're assuming is encoded in unary. It turns out we can afford to encode the primes in unary because the number of primes that we use is this polynomial thing, a weakly polynomial thing. And by the prime number theorem, the prime with that index is roughly that big. It's the log factor larger.

So the primes are going to-- the actual value of the prime is going to be something like n log m, times log base e, and log m. By the prime number theorem and that's

again weakly polynomial. And so we can assume ours encoded in unary. Wow, we get to use the prime number theorem that's fun.

AUDIENCE: Is that the first time you had to use prime numbers?

PROFESSOR: Probably. Pretty sure I've used prime number theorem if and only if I've used Chinese remainder theorem. They go hand in hand. Clear?

So this was kind of roundabout way, but we ended up getting rid of the negative numbers. Luckily it still worked. And now it's zero one permanence hard, therefore counting the number of perfect matchings in a given bipartite graph isn't going to be hard. These problems were equivalent, like they're identical. So you could-- this was a reduction in one way but that's equally reducible both ways. So you can reduce this one, reduce this one to perfect matchings or vice versa.

All right. Here's some more fun things we can do. Guess I can add to my list here. But in particular we have zero one permanent. So there are other ways you might be interested in counting matchings.

So, so far we know it's hard to count perfect matchings. So in a balanced bipartite graph we had n vertices on the left, and n vertices on the right. We're going to use that. Then the hope is that there's matching a size n. But in general, you could just count maybe those don't exist at all. Maybe we just want to count maximal matchings. These are not maximum matchings. Maximal, meaning you can't add any more edges to them. They're locally maximum.

So that's going to be a bigger number, it's going to be always bigger than zero. Because the empty matching, well you could start with the empty matching and add things you'll get at least one maximal matching. But this is also sharp P complete, and you can prove it using some tricks from bipartite perfect matching, counting.

Don't have a ton of time so I'll go through this relatively quick. We're going to take each vertex and convert it, basically make n copies of it. And when we have an edge that's going to turn into a biclique between them. Why did I address such a big one? This was supposed to be n and n. OK so just blow up every edge. My intent is

to make matchings become more plentiful.

So in general, if I used to have a matching of size i, I end up converting it into n factorial to the i-th power, distinct matchings of size n times i. OK because if I use an edge here, now I get to put in a arbitrary perfect matching in this biclique and they're n factorial of them. Cool, why did I do that?

Because now I suppose that I knew how to count the number of maximum actions. So they're going to be matchings of various sizes. From that I want to extract the number of perfect matchings. Sounds impossible, but when you make things giant like this they kind of-- all the matchings kind of separate. It's like biology or something. Chemistry.

So let's see, it shows how much I know. Number of maximal matchings is going to be sum i equals zero to n over 2. That's the possible sizes of the matchings. Of the old matchings I should say. Number of original maximal matchings in the input graph of size i, times n factorial to the i. OK this is just rewriting. I'm just this thing, but I'm summing over all i. So I have however many matchings I used to have a size i, but then I multiply them by n factorial to the i, because these are all independent.

And this thing, the original number of matchings in the worst case, I mean the largest it could be is when everything is a biclique. And then we have n over two factorial maximal matchings originally. And this is smaller than that. And therefore we can pull this apart as a number modulo n factorial. And each digit is the number of maximum actions of each size. And we look at the last digit, we get all the number of maximum matchings of size n over two also known as the number of perfect matchings. Question?

AUDIENCE: So that second max is maximal?

PROFESSOR: Yes this is maximal. Other questions? So again I kind of numbered [? through ?] trick by blowing up-- well by making each of these get multiplied by a different huge number we can pull them apart, separate them-- In logarithm these numbers are not huge so it is actually plausible you could compute this thing. And a tree graph

you definitely could do it by various multiplications at each level. All right. So that's nice, let's do another one of that flavor.

AUDIENCE: So I was actually wondering can you find all those primes in polynomial time?

PROFESSOR: You could use the Sieve of Eratosthenes.

AUDIENCE: Is that like going to be--

PROFESSOR: We can handle pseudo poly here because as we argued, the primes are not that big. So we could just spend-- we could just do Sieve of Eratosthenes, we can afford to spend a linear time on the largest prime. Or quadratic in that even. You could do the naive algorithm. We're not doing real Hogan's [? mid-number ?] theory here. I could tell where you have to be more careful. All right. .

So here's another question. What if I just want to count the number of matchings? No condition? This is going to use almost the reverse trick. So no maximal constraint just all matchings, including the empty matchings. Count them all. We'll see why in a moment, this is quite natural.

So I claim I can do a multicall reduction to bipartite number of maximal matchings. And here are my calls for every graph g, I'm going to make a graph g prime. Where if I have a vertex I'm going to add k extra nodes connected just to that vertex. So these are leaves and so if this vertex was unmatched over here, then I have k different ways to match it over here.

And my intent is to measure this number of matchings of size n over two minus k. That would be the hope. So if I have m r matchings of size r over here, these will get mapped to Mr times k plus one to the r, matchings of size no, not of size. Matchings over here. Because for each one I can either leave it unmatched or I could add this edge, or I could add this edge, or I could add this edge and that's true for every unmatched vertex.

Sorry, this is not size r, this is size n over two minus r. R is going to be the number of leftover guys. Then they kind of pops out over here. So I'm going to run this

algorithm-- I'm going to compute the number of matchings in Gk for all k up to like n over two plus one.

So I'm going to do this and what I end up computing is number of matchings in each Gk, which is going to be the sum over all r. r is zero to n over two of Mr the original number of matchings the size n over two minus r, times k plus one to the r.

Now this is a polynomial in k plus one. And if I evaluate a polynomial at its degree plus one different points I can reconstruct the coefficients of the polynomial. And therefore get all of these and in particular get m zero, which is the number of perfect matchings. Ta da. Having too much fun. Enough matchings.

We have all these versions of SAT, which are hard. But in fact there are even funnier versions like positive 2SAT. Positive 2SAT is really easy to decide, but it turns out if I add a sharp it is sharp P complete. This is sort of like max 2SAT, but here we have to satisfy all the constraints but you want to count how many different ways there are to solve it.

This is the same thing as vertex cover remember. Vertex cover's the same as positive 2SAT. So also sharp vertex cover's hard and therefore also sharp clique is hard. So this is actually a parsimonious reduction from bipartite matching. That's why I wanted to get there.

So for every edge we're going to make a variable, x of e, which is true if the edge is not in the perfect matching. And so then whenever I have two incident edges in f, we're going to have a clause which is either I don't use e, or I don't use f. That's 2SAT, done.

Satisfying? It's parsimonious. So satisfying assignments will be one to one with bipartite matchings. So this is why you should care. And if we instead reduce from-- did I erase it? Bipartite maximal matchings up here, then I get the counting the number of maximally true assignments that satisfy the 2SAT clause is also hard. For what it's worth, a different way of counting that.

(BREAK IN VIDEO)

Maxwell means you can't set any more variables true. Yeah.

AUDIENCE: Since the edges on your positive 2SAT reduction are the variables there are true when the edge is not in a matching. I think it's maximally false. And not maximally true. Also maximally true, you just said everything was true.

PROFESSOR: Yes. Right, right, right, sorry. I see, you're right. So it's minimal solutions for a 2SAT, thank you. OK in fact, it's known that three regular bipartite planar sharp vertex cover. I won't prove this one. But in particular you can make this planar, although it doesn't look like we have positive here. And you can also make it bipartite, which doesn't mean a lot in 2SAT land. But it means-- makes a lot of sense in vertex cover land. In my zero remaining minutes I want to mention one more concept.

To go back to the original question of uniqueness for puzzles. And you'll see this in the literature if you look at it. Lot of people won't even mention sharp P, but they'll mention ASP. ASP is slightly weaker but for most intents and purposes essentially identical notion to sharp P, from a hardness perspective.

The goal for-- in general, if I have a problem a search problem like we started with. The ASP version of that search problem is, I give you a solution and I want to know is there another one?

This is a little different from everything we've seen because I actually give you a starting point. There's some problems where this helps a lot. For example, if the solution is never unique like coloring, I give you a k coloring, I can give you another one I'll just swap colors one and two. Or more subtle if I give you a Hamiltonian cycle in a three regular graph. There's always a way to switch it and make another one.

So ASP is sometimes easy. But you would basically use the same reductions. If you can find a parsimonious reduction that also has the property-- so parsimonious reduction there is a one to one bijection between solutions to x and solutions to x prime. If you could also compute that bijection, if you can convert a solution from one to the other. Which we could do in every single proof we've seen and even the

ones we haven't seen.

You can always compute that bijection between solutions of A to solutions of B. And so what that means is, if I can solve B, consider the ASP version of A where I give you a solution. If I can convert that solution into a solution for my B problem, and then with the parsimonious reduction I also get a B instance, then I can solve the ASP problem for B. That's a decision problem. Is there another solution? If yes, then A will also have another solution, because it's parsimonious the numbers will be the same.

We need a weaker version of parsimony we just need the one and more than one are kept the same. If this one was unique then this one should be unique. If this one wasn't unique then this one should be not unique. If I can solve B then I can solve A. Or if I can solve ASP B then I can solve ASP A.

A lot of these proofs are done-- so-called ASP reduction is usually done by a parsimonious reduction. And so in particular this was introduced, this concept was introduced in the likes of like Slitherlink, I think, was one of the early ASP completeness proofs.

And they were interested in this because the idea is if you're designing a puzzle usually you design a puzzle with a solution in mind. But then you need to make sure there's no other solution. So you have exactly this set up and if you want a formal definition of ASP completeness it's in the notes. But it's not that difficult.

The key point here is if you're going to prove sharp P or ASP completeness you might as well prove the other one as well. Get twice the results for basically the same reduction. All the versions of 3SAT and 1-in-3SAT, and all those things, those were all parsimonious. And all of those are ASP complete as well. But once you get into C-monious you're no longer preserving one versus more than one.

So while you preserved-- from a counting perspective in multicall reductions where you can divide that off that's fine. But from an ASP completeness perspective you're not fine. So in fact all the stuff that we just did with the permanent matchings and

sharp 2SAT, their sharp P complete. They may not be ASP complete.

But you'll notice the puzzles that we reduce from like, whatever, Shakashaka is one of them. That was from a version of 3SAT, and the other one we had was from a version of Hamiltonicity. This guy. These are all-- which was also from 3SAT. So these are all ASP complete also.

But if you use the weirder things like 2SAT you don't get that. So usually in one proof you can get NP hardness, ASP hardness, and sharp P hardness. But if you're going to go from these weirder problems with C-monious reductions then you only get sharp P hardness, because they're not even NP-hard. Cool, all right. Thank you.