PROFESSOR: Today, we begin a new topic, which is parameterized complexity. This is one of the most recent areas of hardness that has been invented that we will cover so far. It started around 1999, and then since the 2000s, it's been getting really big, originally by Downey and Fellows.

And the general idea is we're going to take problems, which normally we think of as being parameterized by the problem size, n, and thinking about polynomial versus exponential growth of n. And we're going to add another parameter. So you could say this is a two parameter analysis in some sense. Usually, we call problem size n and then we call the parameter k.

In general, a parameter is just some way of measuring instances. So parameter, k, is a function from instances to non-negative integers, 0, 1, 2. And usually, this number k is just part of the instance, for example-- well, let me go over here to examples. Let's say problem that's called k-vertex cover-- and usually, I'll just call it vertex cover for simplicity.

You know the problem vertices to cover edges. And usually, the decision version of that is are there k vertices that cover all edges?

So k is an input to this problem, and it appears in the statement of the problem. But in particular, it's an input. And so this function, k, of a vertex cover instance just pulls out that one number and throws away the graph. That's the typical parameter. But in general, a parameter could be some hard to compute thing. Maybe k-- you think of the parameter-- you take vertex cover as your problem, but you define your parameter to be the smallest independent set-- or largest independent set. Smallest is pretty small, 0.

Or, pick your favorite. You could, for example, think about vertex cover with respect to-- this is how we'll write the parameter-- crossing number.

So the idea here is-- sorry I forgot to covet-- we always measure our instances with respect to the problem size, which is the size of the graph for vertex cover instance. But we're also going to parameterize by some other quantity here. Maybe it's the minimum crossing number for your graph, so 0 if it's planar and so on. We prove that that's NP-complete to compute. So this parameter may not be easy to compute in general. But a lot of the times it's just part of the problem. And in general, this is called the natural parameterization.

If you have an optimization problem, like vertex cover is-- minimum vertex cover-- and you convert the optimization problem into a decision problem, you're asking is opt less than or equal to k. That k is the natural parameter for an optimization problem. So most the time, we'll be thinking about that. But things like this also arise.

So just to be a little more precise a parameterized problem is a decision problem plus the parameter.

So when I said vertex cover with respect to crossing number, I meant my decision problem is vertex cover, and I'm going to choose to use this particular parameter function to parameterize those instances. And in general, we'll call them parameter k and problem size n. And our goal is to get a very good dependence on n at the cost of a bad dependence on k.

So let me start with a so-so dependents, XP, would be the set of all prioritized problems solvable in n to the f of k time for some function k-- for any function k, any function f. And FPT is the set of parameterized problem solvable in f of k times n to the order 1.

So this is considered a good running time. This is considered a bad running time. F is presumably exponential. Assuming your problem is NP-hard, you have to have exponential in something. And the goal is to get the exponential away from n

because n is hard to control. We like to solve big problems, but maybe we can somehow characterize that the problems we care about have some small measure, k-- for some interesting measure k. If such an algorithm is possible and, in practice, your k's are small, then you're golden.

These give really good algorithms for solving lots of problems. Any problem in FPT, if k is somewhat reasonable, even when n is huge, we can solve the problem.

This running time is also polynomial for fixed k, but the polynomial changes depending on k. So here, we typically get linear time algorithms for any fixed k. Here, as k increases, the polynomial gets bigger and bigger. And in practice, you probably can't handle more than an n squared algorithm for a large n. So this is considered not useful even for small k. This is considered very useful for small k, depending on your notion of small, of course.

And parameterized complexity, the main name of the game is distinguishing between FPT. And XP is actually rather large, but distinguishing between when this is possible and when it is impossible.

Cool. So for example, vertex cover is FPT. Vertex cover with respect to crossing number, I think, is FPT. I have to think about that a little bit more.

AUDIENCE: Can you say what XP and FPT roughly stand for?

PROFESSOR: Oh, right. FPT is fixed parameter tractable. That's the good word. intuitive-- etymological that means when you fix the parameter k, you get a tractable problem although it's a strong sense of tractable. You can say it is n squared for any fixed k. Or, it's n to the fifth for any fixed k. This constant does not depend on k. But the lead constant does. So that's FPT.

XP, experience points. Good question.

One other good thing to know. You might wonder why f of k times polynomial n. Maybe I could hope for better like f of k plus polynomial n. That actually is the same class. So FPT also equals the set of all parameterized problems solvable in f of k

plus n to order 1. That's a nice general theorem. I won't bother proving it here. I think the f ends up becoming square of the original f. It's a very easy case analysis based on whether which of these two things is bigger.

I will leave that as an exercise. It would be a good problem set problem. Cool, so that answers one plausible question.

Let me give you an example of an FPT algorithm, let's say, for vertex cover. So back over here to vertex cover. How about an XP algorithm for vertex cover. So that's really easy, not very interesting. You could just guess a vertex cover, try all possible vertex covers of size k. Remember, we want to know whether there's a vertex cover size k. k is given to us, so we could try all vertex covers-- or, I shouldn't say that-- all vertex sets of size k.

There's only n to the k of them. And for each of them, we'll spend linear time to check whether it's a vertex cover. So this proves that the problem is in XP. That's trivial. Most problems you'll find are in XP with reasonable parameterization. On some, it's not obvious. And now, let's prove that it's in FPT. This is a little more interesting. There are lots of FPT algorithms for vertex cover. It's an active area. Every year there's a better algorithm pretty much. Question?

AUDIENCE: So in the inputs of the problem, the k is an input?

PROFESSOR: For vertex cover, k is an input.

AUDIENCE: Oh, because you can also think of vertex cover where k is not an input, but you can analyze the run time.

PROFESSOR: Yes. Right. So you could also think about vertex cover with respect to vertex cover. So here, let's say-- this is a little bit funny, but here the idea is k, the parameter, is the minimum vertex cover. Let me write minimum vertex cover. And here, you're given some other value. Let's call it j. And you want to decide whether vertex cover is at most j, but you're not told what k is. You're not told what the optimal value is.

So this is a little bit subtle. It's essentially the same problem. You're just not told the

critical value. But if you can solve k vertex cover when k is given, you can just run that algorithm for k equals 1, 2, 4, and you'll end up solving this problem. So even though-- so in this problem, you can say, well, if I discover there's no vertex cover of size k, I can just return no.

Here, it's a little more annoying, but just by iterating and stopping when you get the first vertex cover, your algorithm will end up having a good running time with respect to k even though you didn't know it. So it's a subtlety. In general, this is messy to work with, so we usually think about k being an input. But that reduction shows they're more or less the same. But the complexity theory for this is annoying.

In the same way that for NP, we restricted decision problems, here we're going to usually restrict to the natural parameter. So good question.

Let me prove vertex cover is in FPT, so a much better running time than n to the k. I think I will get something like 2 to the k times n. Look at an edge of the graph, any edge. Pick an edge, any edge. And we know for this to be a vertex cover, one of the two endpoints has to be in there. Which one? I don't know. Guess. So I'm going to guess either the left guy or the right vertex is in the vertex cover, and then repeat.

So when I put somebody in the vertex cover, I delete all of the incident edges. That takes linear time. Over here, I would delete whatever edges are incident to this guy and the vertex itself, of course. And so I get a smaller graph. I do this k times, so in general, my execution tree will have two branches, every time I pick an edge and say, do I put the left guy or the right guy in.

But I only need to worry about a tree of height k because after I make k decisions, I'm supposed to get a vertex cover of size k. After I make k decisions, if there's anything left to the graph that means that wasn't a vertex cover of size k and I should backtrack. So I just explore this tree, look at all the different leaves. If there's any vertex cover of size k, I will find it. And so I only spend 2 to the k to explore this tree.

2 to the k is the size of the tree times n because I maybe delete n things at every

node. And that's one of the simplest FPT algorithms for vertex cover. There are much cooler, fancier ones. But this is now going to be about hardness, so I will stop with algorithms.

Also related is a stronger notion of PTAS. So I mentioned before, with PTAS, you could have a running time something like n to the 2 to the 1 over epsilon-- which is actually fairly common in a lot of PTASes, not so great. If epsilon is anything good, then this is going to be a really huge polynomial. Well, the corresponding notion in FPT land is a running time of f of epsilon-- or, I'll write 1 over epsilon. It doesn't matter-- times n to the order 1.

So this is really saying that the approximation scheme is FPT with respect to 1 over epsilon. This is to get a 1 plus epsilon approximation. So this is relating the two worlds. If you want a FPT algorithm for approximating a problem and getting within a factor of 1 over epsilon, if you get this kind of running time verses this kind of running time, I consider this good, this bad.

And in general, these are called Efficient PTASes, or EPTAS. And in general, if you have an EPTAS, you also have an FPT algorithm with respect to the natural parameter, which is to basically compute what the optimal value is, because you can let epsilon be something like 1 over 2k-- that ought to do it-- something less than 1 over k.

You're trying to decide whether you have a solution of less than or equal to k or not, whether optimal is strictly greater than k. And I said k is an integer, so this is an integer valued problem you're trying to optimize. And so the difference became k and k plus 1, multiplicatively, is about 1 plus 1 over k factor. It's exactly 1 plus 1 over k. So if we let epsilon be smaller than 1 over k, then to get within a multiplicative factor of 1 plus 1/2 1 over k, you actually have to be within an additive plus 1 of the right answer at k.

So this will decide whether opt is less than or equal to k based on whether that approximation algorithm will give you something of value at most k or strictly bigger. Strictly bigger would be k plus 1.

So this is particularly interesting because it means if a problem is not in FPT, then it does not have an Efficient PTAS. So we can use-- before, in previous lectures, PTAS was considered gold. Now, we can distinguish between EPTAS and PTAS. And there are problems that have PTASs that do not have FPT algorithms assuming FPT does not equal w1-- an assumption we will get to in a moment.

So if you can, in general, establish something is probably not in FPT, then we get it probably does not have an Efficient PTAS. So this is useful for lower bounds about approximation. Even if you don't care about fixed parameter tractability, this will give us cool things about approximability.

AUDIENCE: Can you give a few examples of EPTAS?

PROFESSOR: A few examples of EPTAS. For example, most of the planar results I've talked about, planar independence set, planar vertex cover, those things all have Efficient PTASs. But that gets us into the world of bounded treewidth algorithms. But in general, if you have like a-- a lot of bounded treewidth algorithms are FPT. There will be some exponential in treewidth times polynomial n

So a lot of times, you can use those to get EPTASs for planar and h-minor free problems. So that's one set of examples. There's no short example I can give you, but that's a big class. So for either reason, either you want to solve problems for small k or you want to figure out whether there's an EPTAS, how can we prove that problems are hard in this world using reductions like usual?

So in this context, luckily, there's only one type of reduction we need to learn unlike approximation where there were a lot. And I'll call it parameterized reduction although throughout this lecture and the next one I'll just say reduction usually because we're always talking about parameterized problems.

So in general, we have some problem A-- decision problem A and parameter k. And we want to convert into some decision problem B with parameter k prime. And of course, as usual, the set up is we're given an instance x. This is going to look almost identical to NP Karp-style reductions, but then we're going to have one extra

condition.

So instance x of A gets mapped to by a function f to an instance x prime of B. X prime is f o x as usual. This needs to be a polynomial time function just like for NP reductions, which means, in particular, x prime has polynomials size reduced back to x. It should be answer preserving. So x is yes instance for A if and only x prime is a yes instance for B.

So, so far, exactly NP reductions. And then when we need one extra thing which is parameter preserving.

This is there's some function g, which I'll call the parameter blow up-- or, I guess you call it parameter growth for g-- such that the new parameter value for the converted instance is, at most, that function of the original parameter value of the original instance. Question?

AUDIENCE: Are there any limits on the amount of time that g can take to compute?

PROFESSOR: g should be a computable function. I think that's all we need. Probably polynomial time is also a fine, but usually this is going to be like linear or polynomial or exponential. It's rarely some insanely large thing, but computable would be nice. Cool.

So that is our notion of parameterized reduction. And the consequence, if this exists and B is fixed parameter tractable, then A is because we can take an instance of A converting it to B. If the original parameter was founded by some k, this new parameter will be bounded by g of k. New instance will be bounded of g of k. So we run the FPT algorithm for B, and that gives us the answer to the original instance of A.

So if we don't care about what this function is, we are basically composing functions. So there's some f dependence on k in this algorithm, and we're taking that function of g of k is our new function. And we get a new dependence on k and the running time over FPT. So what that means is if we believe it A does not have an FPT, then B does not have an FPT if we can do these reductions.

So same style, we're going to reduce from a problem we know is hard, A, into a problem that we don't know about, and that proves B is hard as well. Yeah?

AUDIENCE: What is the relationship between FPT and XP? Is there like a difference?

PROFESSOR: Yeah

AUDIENCE: If there's overlap.

PROFESSOR: Well, FPT is contained in XP. And they are different if you believe the exponential time hypothesis. If you believe SAT does not have two to the little of n algorithms-- if that's not possible for SAT, then XP and FPT are different, and much, much more things are different. But we'll talk about that more next class-- relating to exponential time hypothesis.

So at this moment, we don't really care what g is. But if you assume exponential time hypothesis, then g matters, and you can get very explicit lower bounds about how good an algorithm you could hope for. So we won't just prove you're probably not an-- we won't just prove you're not an FPT, but we will give an actual lower bound and how much running time you need, something like n to the little k.

But the running time we get will depend on this blow up function. So next class, we'll care about what g is. So I do try to minimize it, but so far we don't care. Any g is fine. XP will not turn out to be the class that we think about very often. There are things in between that are a lot easier to work with, which we will get to. In particular, W1 is the most common.

So let me do two examples. One of which is a correct example and the other is not. And you can help me figure out which is which. Independent set to vertex cover, these are reductions we've seen before. We have tons of reductions, but usually we weren't thinking about parameters. And independent set to clique, these are basically identical problems left and right. And I want the natural parameter. Which of these is a parameter preserving reduction?

I need some quiz show music.

AUDIENCE: The second.

PROFESSOR: The second, yeah. Because if we take a graph and a parameter k, what we convert it to is the complement graph with the same parameter. So that's obviously parameter preserving. Whereas independence set, these are complementary in a different sense that if you take everything that's not in the independence set is a vertex cover. So it's actually the same graph but with n minus k as the new parameter.

So this is not a parameterized reduction. This is. And in fact, vertex cover, we just showed is FPT. Independence set is not, if you believe w1 does not equal FPT or if you believe exponential time hypothesis. And that would contradict this statement. So this is definitely not a valid reduction. But those are some pretty trivial reductions. Let's do some more interesting things. And I want to start to introduce the notion of w1.

I won't define w1 yet. I'd like to wait a little bit because the definition is not super intuitive, but here is a fairly intuitive hard problem that you should be fairly convinced is not FPT. So I'll call this k-step non-deterministic Turing machine. The one downside is I've never mentioned Turing machines in this class. I thought I could get away without it, but I'm going to have mention them. How many people know what they are? Anyone does not? OK, a couple. Here's a Turing machine.

I will give you, very briefly, a very non-standard definition of a Turing machine. Let's think of-- so you have this infinite tape, infinite memory, and for our purposes, these are not just binary symbols, you can write up to n different symbols on each of the squares. So this is basically your memory, but it's in the style of old-fashioned tape drives in that you-- In order to get to position k away from you, you have to spin k time. So you can only step one unit at a time.

And this is basically computers that we are used to. So let's say it has a sequence of instructions, and it has an instruction pointer. And in general, I want to the machine

to have order n stateS-- yeah, let's say order n lines of code and order n options per cell of the tape. This will look actually a little weird if you're used to Turing machines, but I want to use this definition.

So it's basically a regular algorithm, but you're very limited the number-- in the amount of internal state. These are basically registers, but you can have only a constant number of registers that vary from one to n. One of them will be what line of code are you on. And then you have those lines of code are instructions like jump here, compare these things, write a particular symbol to the current square of the tape, move the tape left, move the tape right. So regular types of instructions, let's say, except you have this weird tape thing.

Now, I mentioned these parameters. And usually we think of Turing machines of having constant size, but here I need an n, and I need a k. Question?

AUDIENCE: I was going to ask what n was.

PROFESSOR: OK. So n is the input. I mean basically the Turing machine is specified by these things. You have order n instructions, and we're guaranteed there's only order n possible states, let's say. And now what we-- and this is a non-deterministic machine. Now non-determinism, we have talked about this in the context of NP. A non-deterministic Turing machine has a funny instruction which says non-deterministically branch to one of n different locations in my memory. Or, let's say choose a symbol from my-- I have this alphabet that I'm using in my cell-- choose a symbol non-deterministically.

So I have order n choices made non-deterministically. In the usual sense of NP, that if there's any way for the Turing machine to output, yes-- one of the instructions is output, yes-- then, I will find it. These are guesses, and they're always lucky guesses, so I always end up finding the return, yes, if there is such a path. Otherwise, all paths return, no, and then the machine returns, no. Yeah?

AUDIENCE: What does it mean for the number of states to change as the input changes?

PROFESSOR: I mean the states are also given to you as part of the Turing machine. There's not

one turning machine-- well, there is actually one Turing machine to rule them all, but that's not the point here. I give you a machine that has-- think of this as firmware built into the machine, and the number of states in the machine includes which instruction you're currently executing. So I mean this is just saying you're given an arbitrary machine of size-- of complexity n. And usual Turing machine land, there are n states because I give you a state diagram with size n.

So you're given everything about the Turing machine. So that is the input to this problem. And the question is, can I find a return, yes, solution that only is k steps long. So I'm given this huge machine, and yet I want a very short execution of the machine. I want the running time to only be k. k, again, is parameter. n is big. k is small.

So I want to know is there a yes path from the initial state of length k. So this is basically-- Yeah, question?

AUDIENCE: Are you given the input to the Turing machine?

PROFESSOR: Yeah, let's say it has no input. Input is all embedded in the code. So the tape is initially blank. Good. There's a reason I need to do Turing machines here instead of usual algorithms, but if you define-- usually NP is defined in terms of these things. So by analogy to NP, we expect there are no lucky algorithms. And so we expect that when you have non-deterministic branches, the best thing you can do is to try all the branches.

And so if I have an execution time of k, and at each step, I can potentially make a non-deterministic choice among n different options, then you would expect the best algorithm is n to the k. Try all the branches, that's just like our vertex cover algorithm, but in the bad case where I have branching factor n instead of branching factor 2. So presumably, there's no way to replace a guess among n options with a guess among two options.

That's an assumption. And this problem is w1 complete. I will to define w1 a little later, but for now, just take this as given. You could define w1 complete to mean

problems that are reducible via parameterized reductions to this problem. And then we'll get lots of examples of problems that are as hard as this. So a modulo the annoyance of having to define Turing machines, I think this is a pretty natural assumption.

It's stronger than p does not equal NP, so we would imply that. So let's do some simple reductions-- some reductions, maybe not simple.

So I mentioned this problem independent set. That's also w1 complete. And to prove, first of all, that it's w1 hard, I'm going to reduce this problem to independent set. So I'm given a Turing machine and a number k. I want to convert it into a graph, and a number k such that if there's an independent set of size k in that graph, at most k in that graph if and only if there's an accept path of length at most k in the Turing machine.

So I'm going to skip a lot of details because I've been a little vague about how Turing machines work, but the idea is nice.

So my graph is actually going to consist mostly of k squared cliques. The cliques are actually quite large because this graph has size n, total size, but there's going to be k squared clusters, which are cliques. Plus cliques, not very many different cliques, only k squared of them. And my target independent set is size k squared, which means if there's going to be an independent set, I must choose exactly one vertex from each clique. So one vertex per clique.

Independence that has to have no edges among them. So if I chose two from a clique, there would be an edge among them, that's disallowed.

And so the cliques I'm going to parameterize by two parameters i and j between one and k. And the idea is that ij represents memory cell i at time j plus the state of the machine.

So what I mean is there are order n states in the machine, in general, that completely characterize what the machine is about to do and what it's thinking about, all of its internal state. And so I'm just going to take those n states and plop

them in. And each node in this clique represents one of those states. But also I want to keep track of what symbol is written on that square of the tape.

And so, in general, my running time is bounded by k-- also the number of squares I write to the tape is bounded by k. Space is at most time-- so I only have to worry about k different cells for times 1 up to k. So there are only order n states for this, only order n states for this. So the size of each clique is order n squared polynomial. That's cool. It's a messy clique. I won't try to draw what it really looks like.

And then the general approach is I'm going to add some edges between pairs of vertices to say, well look, if this cell was this thing at this time, and the state of the machine was move right, then you better not change what was on the cell at that time because nothing changed. You weren't here. So we can just forbid that by saying, well, here is one state. These two states should be mutually exclusive.

So if I choose this vertex in an independent set, I can't choose this vertex. So you just draw all these connections of forbidden things. If something is true time i, times j, then something else should not be true at time j plus 1. You draw in all those connections, and it works. I'll leave it at that because the details are messy, but it gives you some idea.

At this point, I can plug a book on this topic that is coming out next year, a little bit far in the future but it is-- I didn't write it down here of course. Sorry. It is by these guys again, Fomin, Kowalik, Lockshtanov, Marx, Pilipczuck brothers, and Saurabh. And it does go through the details, the gory details. If you're interested in that, I could share with you that section.

But unfortunately, the book is not released yet. Cool. So that was a reduction to independent set, which shows the independence set is at least as hard as this problem, which we think is hard, w1 hard. So that proves w1 hardness. But there's another question of is independent set in w1. As you might guess from the name w1, there's more than just w1. There's w2 and w3 and w4 and even more.

And there's no one notion of hardness like we had with NP here, so we get a bit of

complexity in that hierarchy. But in fact, independent set is w1 hard-- sorry-- w1 complete. So we can do a reduction from independence set to Turing machine. Let's say k-step non-deterministic Turing machine.

And the idea is pretty simple.

We have k-steps, so let's guess all the vertices. You can see here this is where we need the ability to guess among n different options in each guess step. So we're given-- we want to know whether there's an independent set of size k, so guess all those vertices of size k and then check it. So this was our XP algorithm, but now phrased as a non-deterministic Turing machine algorithm. And it's a little bit subtle what it means to check.

Of course, what we want to do is make sure there's no edge between any pair of the chosen vertices. So what we're going to do is when we guess these k vertices, we're going to write them one at a time on the tape squares. So we'll use k tape squares to store the k vertices that we've chosen. Then, we want to check each pair of them. So this is like a doubly nested loop for each vertex among the k, for each vertex among the k.

So what that means is you're going to have to-- sorry, that's a little bit annoying-- loop through the vertices on one tape. And then the fun part is the graph is encoded inside the machine. You could think of there being a data section within the code that says where all the edges are. The graph is size n, and so you can put the entire graph into the machine. And then given two vertices, you could check that there is indeed no edge between them basically by using the code as a look-up table.

Say hey, is there a vertex from i to j. If yes, then we're in trouble, abort this option, try the other guesses. If you look up in the table and it says there's no edge, then you keep going. So you test all the pairs, make sure there's no edge by-- the code, as it is, is a constant size algorithm which looks like this plus an order n size thing, which is the look-up table for what edges are in the graph. So that part's a little bit weird. Yeah?

AUDIENCE: So in case of the k in k vertices aren't the same k?

PROFESSOR: That's right. I'm sorry. This is k prime. Yes. So probably k prime here is exactly order k squared. Thank you. It makes me a little happier. Because I'm doing this doubly nested loop, I'm going to have to loop over this thing several times. I would look at the first vertex, memorize it by reloading it into a register, and then loop over all the other guys for each of them check that they're bad. So I'm going to have to go over the tape k squared steps.

And also n prime is whatever. We get a machine that's size-- basically the size of the graph V plus E. So the point of doing that was to show the independence set is also w1 complete. So if you were not happy with Turing machines, now you can completely forget about them. They are mainly used here as motivation for why you should expect this problem to be hard.

But it's equally difficult from a parameterization perspective. If you ignore how big the blow up is, it's just as hard as an independence set. So independence set is a problem we think there's no good algorithm. And by this reduction, and also the reverse reduction, clique is also w1 complete.

So a bunch more reductions.

In general, most w1 hardness results start from clique. So that's a good problem to know about, or independent sets, I think. And there are some simpler versions of clique that are also hard. So first one is clique in regular graphs. So it's helpful to assume all the vertices have exactly the same degree. Is it hard in three regular graphs? No. Because three regular is only a clique of size three. So that's pretty easy.

But in s regular graphs, the degree is going to be huge, going to be some function of n. Clique is hard, and in general, let's say capital delta is the maximum degree in your given graph, and we want to convert it into a delta regular graph. So we're going to increase all the lower degrees up to delta. What we're going to do is take delta copies of the graph. So for every old vertex, we get delta copies of it.

And then we do this reduction. So the blue is the original-- is the graph that's been duplicated delta times. Here, delta is 5. But suppose, in the graph, this vertex only had degree 3, some number less than 5. Then, what we're going to do is create delta minus d new vertices and then add a biclique here, bipartite clique, connecting all of those delta minus d things to all of these delta things.

They're delta copies of the original vertex. This is v over here, v1 through v delta. And so what that means is these vertices will have entered degree delta because there were delta things to connect to over there. And now these things will have degree delta because they used to have degree d, and now they have an additional degree delta minus d on the left. So cool, everything's now delta regular.

And I claim that this reduction preserve cliques because if you look at these added vertices, they do not belong to any triangles. This thing here is an induced bipartite graph, and so there are no triangles, no things of size-- no cycles of size 3 because all cycles are even here, which means if you're in a clique-- if you have three guys in a clique, then they better have triangles everywhere.

So if you put one of these vertices in, you'll have a very small clique, namely size 2. And there are always cliques of size 2 in a graph. There's at least one edge. So if there was at least one edge before, afterwards, we do not increase any of the clique sizes. So clique size is preserved. Everything's cool.

So in this situation k prime equals k when you blow up anything, except the graph size, of course. That got a little bigger. And now everything's delta regular.

Le me give you an example of why this is useful. Of course, every time I say something is true of clique, it's also true of independence set. You can just flip things. Here, we have to be a little careful, but when you complement a graph, if you started with a regular graph, your new graph will also be regular. Every vertex would have degree n minus 1 minus delta or n minus delta, something like that.

So here's another problem, eerily similar to vertex cover, called partial vertex cover. This is also w1 complete. We want to know, can I choose k vertices that cover l

edges?

So usually vertex cover, you need to cover all the edges, and that let us do some crazy things because whenever we looked at an edge, we knew one of the two things was in the vertex cover. Now, we don't know that anymore. Now, it's a matter of which vertices get the most bang for your buck. And this is easy to reduce from delta regular independence set because-- k independence set if we want to be explicit.

We want to know is there an independent set of size k. I just give that to partial vertex cover with k prime equal to delta times k. Sorry, that's l prime is delta times k. So the idea is here I want to choose k vertices that are independent. And if I can do that, I will be able to choose k vertices that cover exactly delta times k edges because if they're independent, none of the edges will be shared among my independence set. And this is if and only if.

If I try to choose k vertices that cover exactly delta k edges, then they can't be adjacent. The vertices can be adjacent. So these problems become the same under this mapping, so I didn't even blow up. My parameter here is k. I forgot to mention. I mean you could guess from the letter, but parameterized by k, this problem is w1 complete. Parameterized by l, this problem is FPT. So be a little careful.

But here's the reduction for k. There is no reduction for-- there's only good reductions for l. So there we are clearly using that the graph was regular. Otherwise, it would be at most k-- at most delta for everybody, and then it's hard to get the actual independence. Yeah.

AUDIENCE: Sorry, I'm still a bit confused about the relationship between w1 and XP and FPT. I don't know what w1 is.

PROFESSOR: Yeah, well we'll get there. For now, you can think w1 is all problems they can be parameterized, reduced to k-step non-deterministic Turing machine. That's a fine definition. Some people use that. And in general, FPT is contained in w1. It's contained in other things, which we will get to. w2 and so on is all contained in XP.

And these are strict if you believe exponential time hypothesis.

So if you believe there are no sub exponential algorithms for SAT, then this problem has no FPT algorithm with respect to k and all the w1 complete things. I mean from a complexity theory standpoint, it will be fun to look at these larger classes. From is there an FPT algorithm standpoint, all you care about is it's not here. And any hardness, w1 or worse, will imply there's no FPT algorithm if you assume ETH.

AUDIENCE: Do you know if any of these inclusions are strict?

PROFESSOR: Like I said, they're all strict if you assume exponential time hypothesis. If you prove XP is different from FPT, then you prove P does not equal NP, I think, pretty sure. So we're not going to non-categorically say these things are strict, because these are all stronger versions of P does not equal NP. But if you believe exponential time hypothesis, then they're all strict. So that's one standard assumption that gives us everything. Cool, let me give you another version of clique that's hard.

Mufti-colored clique if you remember way back to three partition, we had a variation on three partition called numerical three-dimensional matching where you had to choose your triples from three different sets. So this is the analog of that, or the analog of three-dimensional matching where you had three different sets of vertices. For clique here, set the vertices partition into k clusters.

And the question is, is there k clique with one vertex per cluster? And in fact, so you think of these things as being colored, color 1, and these are color k and so on. And in fact, we can assume this is a proper coloring because if you're not allowed to choose two vertices from one class, then there's no reason to have edges between vertices of the same color. So this is in fact a k-coloring of the graph.

So we are given a k-color graph. And we want to know, does it have a k clique? And if it has a d clique and it's k-colored, then in particular, you will have exactly one vertex per color class.

We can prove this is hard by a reduction from clique.

Namely, if we have a vertex v, we're going to make k copies of it. You'll see this is a common trick.

And we will color them 1 to up to k. So good, now we have vertices in each color class. And then if we have an edge (v,w) convert it into edges (vi, wj) for all i not equal to j. So pretty much the obvious reduction. Once you said make k copies, we'll also duplicate the edges in all versions except, because it's supposed to be a k-color, we're not allowed to connect (vi,wi). But otherwise, we'll just throw in all of those edges. And the point is this, again, doesn't really blow up your cliques because if you have some clique in this structure, you can just forget about the i indices.

Because you know it's a clique, you will never choose two verses from the same color class, two vertices with the same index, and so you can take any clique here and collapse it to a clique here. Conversely, if you have a clique here, you can just assign those vertices arbitrary numbers as long as they're all distinct any permutation of one through k, and you'll get a clique down here. So it's the same problem. Question?

AUDIENCE: That v is different from the v above it, right?

PROFESSOR: Yeah, this is for all v and for all vw. If there's an edge in the graph, then we do that. And if there's a vertex in the graph, we do that. Cool. So k prime here equals k, no expansion.

This may seem trivial, but it's actually a fairly recent innovation to think about multicolor clique and, in general, it simplifies proofs. I have heard of proofs simplifying from tens of pages to one page. We'll probably get to some more-- to such sophisticated example soon. But I can give you one simple example.

Before I get to a simple example, I want to show you a fun example. Not simple, but let me tell you what the problem is. I won't cover that proof.

The problem is shortest common super sequence. This is a problem that comes up in computational biology. Your given k strings. Let's say alphabet sigma. And yes,

you're given a number, which I'm going to write l. You want to find a string of length l that's a super sequence of all input strings.

So maybe you're given the DNA sequence of human and DNA sequence of mouse, and you want to know what is the shortest DNA sequence that contains all of the letters of those strings in the correct order. So this is often called an alignment. If you have ACGG-- this is probably not valid, but whatever-- And we have some other guy like CAGGAT.

So I tried to draw those aligned. So then the common super string here is ACAGGACT. This is a super string, meaning I can drop letters from down here and get this. Or, I can drop letters from down here and get this. That was for k equals 2. In general, this problem does have an n to the k dynamic program. But the question is whether you could get FPT in k. Could you get a small dependence on the-- better dependence on the number of strings?

And the answer is no because this is w1 complete. And this is a sketch of the proof. it's a reduction for a multicolored clique. I think this the paper that introduced multicolored clique in fact. It's the earliest one I could find, and this is 2003. It became more popular since 2009.

So you're basically encoding the edges by whether there's a 0 here in between a huge set of ones. And then this is intuitively rep-- and then for every vertex you're looking at, all of the other vertices and encoding whether they have an edge to them. And so if you have a clique, that will be a pattern that you see repeated among multiple vertices. And so you'll end up being able to shrink your longest-- your shortest common super sequence.

But the details are messy. It's a few pages to prove it. I will skip that and tell you why I wanted to tell you about this problem other than its computational biology and useful. Because it's used to prove that flood-it is hard. Time for some fun. So here is flood-it.

You have a grid of colored squares. And in the top left, we have colors. And this is

the special square. What I can do is control the color of that square. So for example, I could set it to red and then anything that was in that connected group becomes red. And now I've got a bigger connected group. So now I can make it blue, and then pink, and then red, and then blue, and then red. Let's see how well I can do. And the trouble is I have a limited number of moves.

So this is a model for Ebola virus spreading or zombie infection or pick your favorite. I mean you imagine these are different species-- green. Thank you. A lot easier to play when I have an oracle. Yellow, that's better. I don't know. I'm actually doing pretty well here. I think now I just have to do them all. I made it! 25 moves. Wow! Felicitations!

That's the first time I've won. It's a good demo. Only played a few times, but-- this apparently became quite popular in 2006, a company called LabPixies, since bought by Google. But there's tons of free versions out there you can play. It's w1 complete with respect to number of colors and number of leaves. So I don't know if this is a generalization or specialization, but it's a variation of the problem that's been studied.

It's the only one I could can that talked about parametrized complexity of flood-it. And so this is flood-it on trees. So same set up, but instead of being a square grid graph, I'm on a tree graph. And the root of the tree is the one I can control, and my graph is going to look like-- my tree is going to look like this.

These are going to be my strings. And if I want to be able to solve this in some number of moves, l moves, then I need to-- the sequence of colors I do is a longest common subsequence, super sequence, of these strings because they all have to get to the bottom.

They'll sit there and wait, and so you can essentially drop letters. And that's cool. Happy? This is not literally true because if I have, let's say, a zero and a zero here, both the same color-- So I'm representing letters by colors, then when I play-- if I eventually play here and play zero, I actually get advanced two spaces instead of one. So for that, you need to first map every letter a.

So let's say in string i, we map every letter to a followed by a special character for that string. And so this blows things up. In particular, it blows up l. l grows by a factor of the total length of all strings because of all these things that we add. These are not really compressible, but it means that we alternate between regular characters and special characters. And therefore, we never have two characters in a row that are the same.

So first you take this problem, then you reduce it to the version of the problem that has no repeated characters. And then you can reduce that to flood-it because that's what we care about.

AUDIENCE: Sorry, what's the parameter on the shortest common super--

PROFESSOR: Right. Parameter is-- they're actually two parameters, so this is a fun thing, k and sigma. So two parameters, you can think of that as being parameterized by the sum or just by the vector k, sigma. Basically, we get to assume both of them are relatively small. We could be exponential in both k and sigma. If you put l in there, it would not be hard. But if you put k and sigma and it's still w1 complete.

And so over here it's the number of leaves in the tree is one parameter and the number of colors is the other parameter. So we're fixed parameter tractable with respect to that joint parameter. Yeah?

AUDIENCE: Why is it obvious that you can reduce flood-it on trees to not flood-it?

PROFESSOR: It's not obvious that the grid problem is related to trees. There are hardness results for like 2 by n flood-it. But they're just NP hardness. I didn't see a w1 hardness for 2 by n flood-it. So as far as I know, 2 by n colored is hard-- is open from a parameterized complexity standpoint. Cool. I want to do one more reduction, and then I'll finally define w1.

Dominating Set, this problem is actually w2 complete, so this is even harder. But before we worry about w2, let's prove that it's w1 hard. So we're going to reduce from multicolored clique to dominating set. I have a nice figure. This is a preview of

this cool parameterized algorithms book. So I'm going to represent each-- got it right. Yeah, funny.

I'm going to represent each vertex in the multicolored independence set problem by a vertex in the dominating set problem, so vertices mapped to vertices. This is the joy of multicolored cliques. We did all this k duplication stuff once so that we don't have to duplicate our graphic anymore. We just take this graph. We poured it over. Now, what that tells us is that the vi's form color classes.

And what I'm going to do is connect each color class. So this is the set of all things of color one, set of all things of color two. First of all, I'm going to add 2 dummy vertices to each color class just to hang out there. And then I'm going to connect everything in the color class by a clique. Before we only had edges between color classes, but now I want to color--

I'm going to also change those edges. But the vertices mapped to vertices, I add two vertices of each color, and then this circle represents that there's a clique in here, clique in here, clique in here. Now, my goal is to find a dominating set. And so the intention is that you need to choose one vertex from each color class in the dominant set. And that's actually what these dummy vertices are doing.

They are only going to be connect to things in here. They have to be covered by somebody, so that means you have to choose someone in this clique. You have to choose someone in this clique. When you do, it covers everybody in there. And there's going to be no point of choosing these dummy vertices because they're only connected to other things in the clique. You might as well choose these things.

Now, how do I represent an edge in my independence set graph. If I have an edge between say u and v, that's in this color class and this color class. We know they're in different color classes by multicolored clique property-- multicolored independent set property. Sorry, same thing. So I want to not choose both u and v. So I'm going to represent that by adding a vertex here, which must be dominated, and connect it to everyone except u over here and everyone except v over here, the red patches.

So there are edges from this vertex to every red vertex here and here. So what that means is if I don't choose u, I will cover this added vertex. If I don't choose v, I will cover this added vertex. So I will cover it as long as I don't choose both u and v. So it's like I should choose at most one of these two guys, then this will be covered for free. And it has to be covered for free because we just-- if I set k prime to be the number of color classes, which is k, then I don't have any flexibility. I can't choose any of these vertices to dominate.

And so that simulates independence using dominating set, pretty simple and clean. So that gives you a flavor of nice reductions you can do with multicolored clique or independence set.

AUDIENCE: What's two extra vertices?

PROFESSOR: Why two?

AUDIENCE: The two you added, it looks like are not adjacent so--

AUDIENCE: Oh, so we derived them.

PROFESSOR: Right We want them to be there in order to force choosing somebody down here, but we don't actually want to choose one of them. And so we emit that edge so if you choose one, you don't cover the other. Thanks. I think it simplifies the argument. It's probably not necessary because I think you could argue by replacement argument that you won't choose them, but anyway.

Time for some definitions. I wanted to do some fun problems before we got to the somewhat weird definitions. The time has come.

By the way, dominate set is w2 complete, so is set cover. And most w2 hardness reductions start from one of these two. We already know how to reduce dominating set to set cover. You make a set for every neighborhood set in the graph. So this reduction is easy, and that preserves a parameter so just from the same reduction we did two classes ago, I think.

Here is a problem which will look familiar, but under a slightly different name. This is

what I might call circuit ones. One's meaning you want to set some number of the inputs to be one, just like our optimization version. In this universe, it is called weighted circuit set. I'm not a fan of weighted because there aren't actual weights here. The goal is to get minimum hamming weight, meaning the minimum number of ones in your input.

So I give you a circuit-- I'll draw a very simple circuit, and it has one output. It has some number of inputs. My goal is to set k ones and get a one out. Did I do it? I did it! ! wasn't even looking. So that's an example of an input and an output to weighted circuit set. The parameter here, k, is the number of ones, one inputs. So the question is can you satisfy a circuit using only k ones. That's weighted circuit set.

This problem defines the class called WP. This is the original definition of WP is all problems that reduce to weighted circuit set in a parameterized reduction sense. It's all parameterized problems that reduce to weighted circuit set.

I lost my hierarchy. But this is even bigger than all the things. So we have FPT is contained in w1. It's contained in w2, et cetera. And then we have WP, and then we have XP. So it's not bigger than everything. It's not bigger than XP. I won't prove this here, but it is true.

Now, I want to specialize this problem. I'm going to simplify it using a notion called weft. So first the depth of the circuit, this is the longest path. That's a normal notion. Then, the weft of a circuit is the maximum number of big gates on an input to output path.

Big gates means let's say more than two inputs. In general, it's more than some constant number of inputs, but two is fine because if you use a bunch of two input gates, you can build a ten input gate. But there's a distinction between constant input gates and super constant input gates. So if you count how many super constant input gates you need in a constant depth circuit, then we get the w classes, w for weft.

w(t) is the set of parameterized problems that reduce to constant depth weft-t

weighted circuit set. t is not the parameter. K is still the parameter. k is the number of ones. t is the thing. Let's do some examples.

AUDIENCE: Depth has to be larger than weft numbers?

PROFESSOR: Yes, depth is always larger than weft. So we allow-- so t is a fixed constant. Depth could be 100 times t or something.

AUDIENCE: Then, w in WP is a different w than w(t)?

PROFESSOR: They both have to do with weft, but that's the P is a polynomial. It corresponds to polynomial weft, which is basically unbounded weft. w does not stand for weighted. In both cases, it stands for weft. What is weft? Weft is the opposite of warp. When you're weaving, you've got the warp threads and then you've got this weft thread that goes back and forth and ties the whole circuit together.

So that's sideways, so if you imagine that constant input gates do not need a weft to hold them in, but super constant ones do, then it's how many layers of wefting do you have to do to cover all those things? That's the larges the longest path in terms of counting the-- that's the term, OK? But it's actually fairly useful to think about. So for example, independence set, inputs are up here. Output is down here. I need to choose, for each vertex, whether it's in the independent set. And if I negate them, then I need various constraints.

This is the graph that's being represented. I want to say either I don't choose this or I don't choose this. And then all of those clauses must be true. That's the and of those things. This is big. These are not big. They have two inputs. This has one input. So these are all free. And then we just do one in terms of the depth here where there's one level of big gates. So independent set is in w1. That's a proof that it's in w1. So that proves all the things we've been talking about, except dominating set, is in w1.

Now dominating set, you need two levels. So we have the a constraint. So this is saying that a should be covered. a is adjacent to a, b, and c. I mean if I choose a, b, or c, then a will be dominated by the definition of dominating set because a is

adjacent just to b and c. But also if I choose a, then it's dominated. So this is an or of those three things. And a should be dominated, b should be dominated, c should be dominated. So it's an and of all those things.

But if we don't have bounded degree, which we can't because this problem is easy for bounded degree, then we have two levels of big gates. So this problem is in w2. Now, funnily enough, w1 and w2 are the most common things you see. I've never seen a paper about w3. Some statement, I think, about problems that we tend to care about, ands of ors are common. This, of course, CNF SAT is in w2. I should say weighted CNF SAT is in w2. It's actually w2 complete. Weighted three set is w1 complete because things of size 3 are OK.

We can split those into two things of size 2. So in general, order one SAT, you have five SAT. That's w1 complete. And CNF SAT is w2 complete. And I think most of the problems we care about can be expressed as-- this should be weighted. I'm going tow rite w, but I mean weighted. But this is a capital W, different. That's weft. So fine that's-- and most of the problems we care about can be represented as CNF SAT.

I think that's why we rarely get outside of w2. But there are other things there. You could do an or of ands of ors or an and of ors of ands. Now, we know such formulas can be converted into CNF form and only get a polynomial blow up. What's going on?

Well, when you do that inefficient form, you add extra variables. When you add extra variables, you are no longer preserving the weight. It would not be a parameterized reduction. So if you're just caring about satisfiability, that's fine. But if you're considering weighted satisfiability, you want to minimize the number of ones, it totally changes when you convert to CNF. So it matters.

One more fun result, if you look at 2-tape non-deterministic Turing machines-- the reason I had to define 1-tape non-deterministic Turing machines is because they're different from 2-tape non-deterministic Turing machines. These are w2 complete. The k-step version of 2-tape. 2-tapes is you have 2-tapes that can advance independently. Or, you can think of there are two fingers you can independently

move on one tape. Those are equivalent, and they give you w2. Three tapes, gives w2.

Any constant number of tapes, you still get w2. And this is another, I think, natural reason why w2 comes up a lot. I have seen problems that are hard for what's called w-star. This w-star is wt for all fixed t. You can think of that as w order a. So it's the same thing. So I've seen problems that don't depend on what t is. They're hard for all of them. But I've never seen a w3 complete problem. Maybe we can think of one.

I've also seen WP come up, and there's also a problem called W-SAT which is if you have weighted formula sat instead of weighted circuits sat, you get w-sat instead WP. I think it's a little weaker because in circuit SAT you can reuse things and in formula set, you can't. And it turns out to give slightly different classes. So things are messier here. But if you just care about whether a problem is fixed parameter tractable, these are all bad.

But the point of knowing about them is you want to know if your problem-- first you should check where it fits in the w hierarchy by just thinking about what's a trivial way to write it down in the CNF style thing. Figure out whether it's w1 or w2. Then, you'll know which problems you should start from. If it's w1, probably want to start from a version of clique. If it's w2, you probably want to start from a version of dominating set. That's why we talked about them here. And usually it's w1 or w2.

Cool. Questions? We'll do more next time. This is a fun area. And next time we'll talk about, in particular, how the exponential time hypothesis relates to all this.