[SQUEAKING]

[RUSTLING]

[CLICKING]

YAEL KALAI: OK, so let's start. So first of all, wow. Thank you all for coming back. I'm happy I didn't push you away. So before we start with today's lectures, just a few logistics.

So for some reasons, you guys are taking a holiday next week, unfortunately, so we won't have class. The week after, on the 29th, unfortunately, I'm away that week at Germany, but we're going to have our amazing Rachel in the back give a fantastic lecture. So you'll enjoy that.

And then the week after, which I believe is October 6, I would have loved to give you a great lecture, and I'm here. However, you're going to get something even better because there is a really, really amazing work, a new work by Oded Regev, who does a new construct, a new factoring algorithm, quantum factoring algorithm, so kind of improves Shor's algorithm in some way. It's a really, really exciting piece of work.

Oded is in Paris, but that's the one day he was able to come here and present his work, and it's exactly on this class. So I think what we should do, we should all-- this class will move on October 6, and we'll all go to Oded Regev's talk. It's going to be this class, it's going to be a colloquium, and it's going to be a theory reading group. It's going to be all in one.

It's going to have a very nice format where the first hour or so is going to be kind of a very general theory audience talk, and then we're going to have the next two or less hours, depending on how much he'll have the energy to speak to us, we're going to talk about more details about his algorithm and his work. So that's October 6. And then I'm going to be back. So you're going to have a bit of a break from me now, which I'm kind of sad about, but you may be happy.

So let's first kind of recap where we were and continue from there. So last time, we defined this new class of proofs, kind of going away from the classical proofs to the regime of interactive proofs. We showed the power of these interactive proofs by showing one protocol in particular, the Sumcheck protocol. I wrote the Sumcheck protocol kind of very loosely in that board because we're going to refer to it today again.

So the Sumcheck protocol, just to recap, it's a way to prove that the sum of a multivariate polynomial over, let's say, a small set h-- you can think of h as 0,1 as an example, over a Boolean cube or just a bigger cube, the sum is some value over some finite field. And of course, to compute the sum takes a very long time. It's, like, h to the m. You need to just sum everything up.

But it turns out that you can verify it using an interactive proof very, very efficiently. Assuming you have oracle access to this function, you can compute it efficiently on any random element in the field. So this is what we covered last time, the Sumcheck protocol.

As I told you, this object protocol is really the bread and butter of proof systems. Almost every proof system uses this protocol. It's a beautiful protocol, in my opinion, and we'll use it throughout this course. So if anyone has any questions about the Sumcheck protocol, please ask.

And if it doesn't sit with you yet, I really encourage you to review the notes and-- in the website, there's also notes to Justin Thaler's survey which contains a very nice exposition. And also we have now Scribe for that, so you have various resources to look at the protocol.

And then we start to look at applications for the protocol. The first application we got is to show interactive proof for Sharp-SAT, and we showed this interactive. The idea was to it was via this technique of arithmetization converting kind of a Boolean circuit into an arithmetic circuit. We'll use this technique today as well.

And then we departed and talked about this idea of a doubly efficient interactive proof, where in a doubly efficient interactive proofs, we really care about the runtime of the prover in addition to the runtime of the verifier. So in the classical interactive proof setting, all the focus was on efficient verification. Nobody cared about the runtime of the prover. For all practical purposes, we thought of him as an all powerful.

But doubly efficient, interactive proof, now, yeah, the proof, of course, is more powerful. Otherwise the verifier would do the work on his own. But we don't have all-powerful things in this world, unfortunately, so we also care about the efficiency of the prover. So he's just more efficient, but we don't want his work to be crazy expensive. So that was the doubly efficient interactive proofs.

And we just started, and we'll continue today to show how we get a doubly efficient interactive proofs for counting triangles, for counting triangles in a graph, and it was via technique called the low-degree extension. So now the plan for this class today is to look at the low-degree extension. Again, this low-degree extension, it's a theorem. It's written here. I'll explain it.

It's, again, the bread and butter that actually-- this low-degree extension theorem in some sense is what makes the subject protocol so popular, so kind of important. This theorem is what gives the Sumcheck protocol its power in a sense. So really together, the low-degree extension theorem together with the Sumcheck is like everything follows from it, via corollaries almost. So that's what we'll see this lecture and the next lecture.

OK, so what is the low-degree extension theorem? What it says is that take any function from H to the m to 0,1. H is just any set. It can be 0,1 to the m, can be Boolean hypercube, but it can be a bigger hypercube. If you're more comfortable, if it's easier for you to think of H as 0,1, just think of it as 0,1.

So take any function from H to the m to 0,1. So when I say any function, it's not a polynomial, it's just a set of elements. So when I say any function from H of them to 0,1, you should think of it as just an arbitrary sequence of H to the m bits, and then just ordered them, like in a hypercube.

So I have this sequence of H to the m bits defined by this function. And the theorem says that if you take any finite field that contains this set H, so if H, let's say, is Boolean 0,1, you can take any finite field. Every finite field contains 0,1, but more generally, any finite field, there exists a unique polynomial. So now we move from an arbitrary set of bits to a polynomial f from f tilde.

This is the extension, so we call it from f to f tilde. From f to the m, not H to the m, but f to the m, so I extended it to f to the m, to f such that if you look at the extended polynomial and the small hypercube H to the m, it really is exactly f. And this f tilde has small degree, so it's degree in each variable is the size of H minus 1.

So if you want to visualize it, here's what you should think. I have a bunch of bits sitting on some hypercube. It can be Boolean or it can be more elements here. But I have arbitrary sequence of bits sitting here in this hypercube.

And the theorem says you can extend this to a function over a big field, over f to the m in a unique way. I don't know if I specified, it's a unique way in a way that it extends. So the values are exactly as before. And the degree of this polynomial in each variable is exactly H minus 1. This is the theorem.

By the way, in the case where H is 0,1, then the degree in each variable is 1 because H is 0,1, so the size of it is two degree 1. This is often called multilinear extension because the resulting f tilde is of degree 1 in each variable, in which case it's called multilinear. Any questions about what this theorem says, the meaning of this theorem.

So we're going to prove this theorem next. And then we're going to go back to this counting triangles, which was just kind of a motivating example. But the focus is really this theorem.

And before I go into the proof, I want to say why this thing is so powerful is that it allows you to go from an arbitrary sequence of bits to a polynomial, which is of low degree. And a polynomial of low degree has a lot of structure, and that kind of really helps us. For example, once you have polynomials of low degree with structures of low degree, we can apply Sumcheck protocols.

So that's kind of the way we often use Sumcheck protocol. We're first going to go to look. We have an arbitrary sequence. We're going to take the low-degree extension, and now we can argue we can then use the Sumcheck protocol. Any questions before we-- yeah.

STUDENT: Is another way to get these [INAUDIBLE] maybe not for arbitrary constants. So you take the arithmetized--

YAEL KALAI: Arithmetize?

STUDENT: f.

YAEL KALAI: Yeah, right.

STUDENT: That should give you a low-ish degree, lower degree, if the depth of f is small on the Boolean hypercube [INAUDIBLE].

YAEL KALAI: So what you're saying is good. Let me just repeat what you're saying. What you're saying is if f has already-- is not arbitrary, but actually is kind of a circuit, an arithmetic circuit with and, and or, with addition and multiplication, and it's very low depth, like log depth, let's say, or let me call it depth D, then just kind of think of it as over a big field. Especially if it's 0,1, you can think of over a big field. And then the degree of this thing will be 2 to the depth. You're right.

But 2 to the depth, eh. So you're right. And so actually for Sharp-SAT set, that's what we did. How did we use the Sumcheck? That's exactly what we did.

But then you need to assume a lot of things. You need to assume that-- so you're right. If you start with an f that has structure, that it's very, very low depth, or the degree is very small, then you're done. You already have it. But what this is saying for any sequence, so any sequence of bits can be arbitrary. You can put it in this way.

STUDENT: So I was just dropping to the uniqueness claim. So for these kind of low-depth f's, you might have a different--

YAEL KALAI: Different.

STUDENT: It's probably-- but I guess the point is that degree will prevent you, right?

YAEL KALAI: Exactly. So what you're saying for the functions that are kind of natively small degree, you can immediately get an extension, but the degree will not be the minimal, minimal degree. So to get the minimal degree, that's how you do it. Yeah, good. Thanks.

Any other questions? OK, I just want to remind you, questions are great. I hope you just witnessed it because it's good for all of you. So please ask questions.

OK. Should we dive in? OK.

So the first thing to notice is, actually, you should be familiar with this theorem for the case where m equals 1. So first, just notice, for m equals 1, this theorem is simply known as the Lagrange interpolation theorem. This is really just Lagrange.

What does Lagrange interpolation theorem says? Just to recap, it says that if you have arbitrary-- if you have an f exactly the same, I'll just write it from H to 0,1, then there is a unique way of extending it to f to be of degree H minus 1. So there is a unique f tilde from f to f of degree H minus 1.

And Moreover, I'll tell you what it is. Let me just tell you what this f tilde is. The F tilde is simply sum over all the h's in h-- sorry, f of h-- times some polynomial chi h of x, where this polynomial is 0 on all the h elements, except for on this specific one. So this polynomial where-- so this is the f where chi h of x is what? Is 1, where if for every h prime in H, it's 1 if h prime is equal to h, and 0 otherwise. Yes?

STUDENT: Shouldn't that be the inverse [INAUDIBLE]?

YAEL KALAI: Shouldn't it be the inverse?

STUDENT: [INAUDIBLE]

YAEL KALAI: I want the output to be-- so OK, let's see. What I want that if I'm somewhere in H, so I want to say that if I'm somewhere in H prime, I want to get f of H. I want f tilde of-- so for any H star, I want f tilde of H star to be f of H star.

STUDENT: Either 0 or 1, right?

YAEL KALAI: Which is either 0 or 1. Yeah? Yeah.

STUDENT: So that's what I'm saying.

YAEL KALAI: Oh, sorry.

STUDENT: Instead of this, it should be the inverse so that when you apply on the H star, it cancels.

YAEL KALAI: What do you mean, f of H should be the inverse? I'm not sure-- you're saying here, I should have inverse?

STUDENT: Yeah.

YAEL KALAI: Why?

STUDENT: So the next H will give you something that's non-zero.

YAEL KALAI: OK, so let me try to argue. Maybe let's try to do this together. So if this is chi-- So I first define chi of H that I'm going to define it in a second. Actually, I didn't finish defining it, but what's important to me that on every h prime in H, it's 1 if h prime is equal to h and 0 otherwise.

Suppose I defined it. Now, let's see what I get here. I have f tilde of H star is sum fH. Chi h of H star, but this is always 0. Except it's 1 when h equals H star. So it's always 0, except for when h equals H star and when h-- oh, you're saying when h equals H star, then I have f of H star.

So this does extend. Yeah? So we're good? OK, great.

OK, but let me just mention what is-- I said, this is the chi. But let me actually be explicit. So let me erase this.

The chi in the Lagrange interpolation theorem, it satisfies this. But let me actually tell you exactly what it is. It's pretty simple. It's just sum over h prime in H, and then it's, I think, H prime minus x divided by h. Oh, sorry, multiplication. h prime in H minus except for h, and its h prime minus h.

So first, note, this is of degree h minus 1. And when you look at it for any element-- oh, yeah, for any element h prime, it's going to be-- this is going to cancel out unless you happen to be h.

Wait, did I do it wrong? Unless you're h, and then you're 1. Good. OK, so this is just a recap for Lagrange interpolation.

Now, let me tell you here. So this theorem says the same thing, but for m bits, for m variables. And it turns out, you know how this was f tilde? Here f tilde, of Z-- Z now is in f to the m-- is very similar. It's also sum f of h1 up to hm, so for every point times this chi h1hm of z.

And what is this? This is 1 if z-- and the small hypercube, this is 1 if the Z is h1 up to hm and is 0. Otherwise, and it extends. So but actually, let me be more specific.

It's really where chi h1 up to hm of-- let me call it x1 up to xm is just the multiplication of chi hi xi. i goes from 1 to m. OK, so let me move this board.

Don't let me erase this board. OK? We'll need it.

So again, what is the low-degree extension f tilde? It's similar to the univariate case. You sum over the entire hypercube, and you multiply by the degree h minus 1 in each variable. It's just the multiplication, all of these chi i's that were defined here.

OK. So let's see if-- let me try to prove that this works and that it's unique. So first, what do we need to argue? So we have this f tilde. Now I need to argue that it extends.

So the fact-- OK, so proof. So let's first see that for every h1 star up to hm star in H to the m, let's look at f tilde of h, hm. What is it? It's just sum f of h1 up to hm times chi of-- so let's look at f tilde. f tilde, on any point in the hypercube, I defined it by star because it's a specific point.

And I just substituted. I defined this f tilde. I just substituted Z because this is my Z point.

And now what do I have? I know that this is equal to 1, if and only if hi star is equal to hi for every i, because this is how the chi is defined. This is just the multiplication of these kinds.

So therefore, that's it. This is just f of h1 star up to hm star, because the rest of the terms cancel. Done. Yeah? So it extends.

How about the degree? What's the degree of f tilde? It's just the degree of these chis. So let's look at degree i of variable i. It's the degree i of chi of h1hm.

Which is what? It's just the degree of a single chi hi on xi, because this just factors to the multiplication chi i and xi. So that's just the degree of this, which is, by what we saw there, simply h minus 1. Done.

So it is indeed. It extends f. The degree is correct, h minus 1 in each variable. The only thing that I need to argue is that it's unique. Questions?

OK. So the uniqueness will follow from the uniqueness in the univariate case. So we already know from Lagrange interpolation theorem that in the univariate case, f tilde is unique. Now we're going to rely on that to argue that in the multivariate case, f tilde is unique. OK? Questions?

OK, so let's do it. So uniqueness. So suppose for contradiction that it's not unique. So suppose for contradiction that it's not unique. So there exists two ways to extend that agree on the hypercube.

This means that if you subtract these two, you'll get a degree h minus 1 polynomial in each variable that in the hypercube is 0. But it's not 0 because there are two different polynomials. So suppose for contradiction-- let me just erase this because it's in my way. Suppose for contradiction that there exists a g from f to the m to f, non-zero, but g in the hypercube is 0-- that's the contradiction-- and of degree h minus 1 in each variable. Yes, that's a contradiction.

Say, suppose there exist two polynomials that agree on the hypercube of this degree. We're going to subtract them. We're going to get a single polynomial that just 0 on the hypercube because the two agreed. But it's not 0 because they were not the same polynomial.

What does it mean, non-zero? What does this mean, of degree m minus 1 in each variable? The fact that it's non-zero means that there exists some t1 up to tm in f to the m such that g and t1 up to tm is not 0. Yeah? So we're starting to think, suppose we have g. It's not 0, but it's 0 on the hypercube.

I'm going to argue, no, it can't be. I'm going to find an element in the hypercube for which it's not 0. OK, how do I do that?

What I'm going to do, I'm going to go into the hypercube coordinate by coordinate, kind of by induction. And here's what I'm going to do. I'm going to say, OK, let's look at g mt t2 up to tm. This is non-zero because on t1 it's not 0. So this is non-zero of degree H minus 1.

But from the Lagrange interpolation theorem, it says that if it's not 0, there must exist small h in H, for which it's not 0, because if it was 0 in all h, there's a unique way to extend the 0 polynomial. So because the uniqueness of the Lagrange, it means that-- so from Lagrange's theorem, this is just univariate. It means that there must exist h1 such that g of h1t2 up to tm is non-zero.

Again, why? So, the t's are fixed. This is a univariate polynomial. If it was 0 on H, then you could extend it to the all zeros. But there's another way to extend, because the g's not all zeros. That would be a contradiction.

So therefore, there must exist H1 such that this is 0. And now we're going to slowly continue in this manner. OK, now let's look at gh1t3 up to tm. So h1 is fixed in the small cube. These are fixed in f.

It's non-zero because on t2, it's not 0. That's what we established. So this is non-zero of degree h minus 1, and therefore somewhere in this small hypercube-- somewhere on h, sorry, it should be non-zero from Lagrange interpolation theorem.

So therefore from Lagrange interpolation theorem, there exists h2 such that gh1h2 t3 up to tm is non-zero. And you just continue one by one until you get that there exists gh1h2 up to hm that's non-zero, and that's a contradiction because we assumed it is 0 everywhere on the small hypercube. Yeah?

OK. So in the notes, I have the reduction. There's a proof that, kind of by induction, very formal, but I think this is good enough for class. Otherwise, it's going to be a bit boring. Questions. Yes?

STUDENT: Can you also just use a counting argument? Like, you can just count how many polynomials of degree [INAUDIBLE] when there are? You break it up into all the terms, and that's just equal to the number of functions?

YAEL KALAI: OK, good. So you're saying maybe you can also prove it using just a counting algorithm and say, if there is too many, then you'll have too many polynomials, and that's a contradiction because the number of polynomials, like the number of coefficients, probably you can do something like that as well. Yeah. Yeah?

STUDENT: Can I try to make the case for the arithmetized--

YAEL KALAI: Yes, please. Good, good.

STUDENT: So the arithmetization, we can compute it on any element [INAUDIBLE] easily, I guess, without looking at the truth table of the original function.

YAEL KALAI: Good.

STUDENT: If I can just repeat it.

YAEL KALAI: Yeah.

STUDENT: If you have to compute f on any point, you need to look at the entire [INAUDIBLE].

YAEL KALAI: Good. Very good. OK, so what you're saying is you're saying, look, once you force me-- look at me. I don't assume anything about f. You assume all degree. Look at me. I assume nothing, and look what I get.

Now you're saying, well, you assume nothing, but you don't get that much either, because this f tilde is so inefficient. Note, to compute this f tilde-- where is it-- it takes time h to the m. You need to sum over all the hypercube. So computing f tilde is really inefficient, whereas your f tilde is so efficient and nice. 100%, yes, you're right.

I'll give it to you Vinod, yeah? No, but this will be useful in settings that the arithmetized version will not be. But that's a good point. Thanks.

STUDENT: Is there an easy way to bound the number of monomials? Is there some unique f that has at most so many monomials?

YAEL KALAI: OK, so the question is, can we bound, essentially asking, well, can we make this f tilde a little more efficient maybe, kind of? So in general, we cannot because in general-- it depends what you want to assume about the f. So I'm saying, I don't want to make any assumptions about the f. And as you will see, it's not just because I want to just be as generous as general as possible for the sake of generality.

Actually, in the way we use it, it really has no structure. The f we use is very arbitrary, so we can't assume that it comes from a low degree, whatever, nothing. And in that case, if you want to allow any f, then you have to have all the monomials because just by a counting argument, if the number of f's is, like, all the possible h to the m bits. So you have to have that many monomials, otherwise where would you store that information?

So in general, you can't do with less monomials. However, in many cases, you can do. And actually, even, we'll use it, the fact that in some cases, you can do with less monomials. So we'll see that. But that's a great, great point. Any other questions?

So let me just tell you, this is a very good theorem to stop and ask questions because it really will get back to this so much. And even if not in this class, you'll see it in-- it will come at you. It's a really, really important and fundamental theorem.

So I guess, next, I'm going to try to convince you why it is such an important and fundamental theorem. And I'm going to start with just looking back at the example of doubly efficient interactive proofs for counting triangles, just as a motivating example. We'll do it quick.

And then we'll go into I think the more interesting and general way of using this low-degree extension together with the Sumcheck to get doubly efficient proofs for any low-depth computation, so kind of arbitrary computation. So we'll see that next. But let's first, kind of as a warm-up, look at the example for counting triangles. Why this low-degree extension with Sumcheck, how can we use these together to count triangles in a graph?

So now, note, what's the goal? We have a graph. I want to know how many triangles there are in the graph. Now, it's not that hard. It takes time, n to the third. I go over every three pairs of vertices and check if there's a triangle. So I have an n to the n cubed algorithm.

But let's say the verifier can't run in time n cubed. The poor guy, he gets the graph, so size n squared. All he can do is run time n squared, essentially, with some polylog overhead, but only n squared. So he can't count. So now what does he do?

He needs help. Now I'll show you how the prover can help him in a way that now, yes, he only needs to run in time n squared or tilde of n squared. OK, so let's look at that example. So let me erase this.

So here is an example where this is useful. We're given a graph. And the goal, we want a protocol, a proof where the prover will convince to the verifier that the number of triangles in g is, let's say, some number beta. I want to prove for this fact.

How will I-- so I'm going to prove to you. You guys are time of n squared. You can only read the graph. I'm going to help you, prove to you that the number of triangles is exactly beta while you just run time n squared. How do I do that?

Let's look at the function f from-- so suppose v is n. So I look at the function from 0,1 to log n to 0,1, and this just tells me whether-- so it's like 0,1 to the log. It's you can think of this as being kind of n by n, so it takes log n bits that describe one vertex log n bit to describe another vertex, and it outputs 1 if and only if there's an edge. So you can think of each-- so f takes two vertices i and j, and outputs 1 if there is an edge and 0 otherwise.

So f is just the adjacency matrix. The adjacency matrix is n squared bits. This is n squared, so I just put it in, define it this way. So again, f takes two vertices. It outputs 1 if there is an edge between these vertices.

Now, so what do we need? So now, what do we know? We know that this f has a low-degree extension. So this is of degree. Essentially this is multilinear because I started with 0,1. So f tilde is multilinear, so degree 1 in each variable.

Now, why am I looking at these adjacency matrix? What do I want? What do I want to prove? How does this relate to the number of triangles?

So the number of triangles-- we started discussing this last time-- is just the sum over all kind of triangles, i, j, k, or sorry, all kind of three nodes, i, j, k, and v. And it's a triangle if what? If f, i, j is one and f, j, k is and f, i, k is one. So I want to know how many triplets is this thing one? If an edge is missing in one of them, then this is going to be 0 because I multiplied. So I want to know how many of them do I have one?

And that's almost the number of triangles, except that I need to divide by 6 because I counted each triangle, each three vertices, all the permutations. So that's what the prover wants to prove to the verifier. So let me-- I said v, but let me call it 0,1 to the log n, which is the same.

Now I'm done. Why am I done? Because I'm saying, you know what? So how do I prove this? I'm going to use the Sumcheck.

But you're saying, wait, but this is not a polynomial. No worries. Here you go. Now it's a polynomial.

Now, I didn't change anything because it's only over the small cube. So the fact that I extended doesn't change the sum because of the extension. So the prover needs to prove that the sum of this polynomial-- so before, it was multilinear. Now it's degree 2 because each variable, we multiply 3 polynomials, each one appears in two. Fine, degree 2 in each variable. And we need to prove that the sum is beta.

This is exactly what Sumcheck is for, precisely. Note, what does the prover-- so what are we going to do? We're going to do Sumcheck protocol.

What is the runtime of the verifier in the Sumcheck protocol? So they interact. We do, every time you get rid of one variable, one by one by one. At the end, the verifier needs to compute the final polynomial, which in our case is this entire polynomial, and a single point in the extension. Namely, he needs to compute f tilde on a random point in f in, I guess, f squared three times.

But that he can do very, very easily. Or what do you mean, in time n squared. Because what is f tilde? f tilde, as Vinod said, you need to enumerate over all the small cube. It's a lot. But here are the small cube is n squared. So it really takes time n squared, so order of n , I think it is, three times. He needs to compute this, but this is polylog.

So by the way, throughout this-- I mentioned it last time, but I'll mention it again. Throughout this course, the polylog factors, I'm not even-- these are all you can put things in tildes. I don't want to pay attention too much to polylog factors.

So really what he worked, the runtime of the verifier is just n squared. The communication complexity is nothing. m number of variables is order log n. d, the degree is constant. Log f, well, don't take f to be too long, too big. Take f so the log f is not small.

And note, even the prover doesn't run in too long. What's the runtime of the prover in the Sumcheck protocol? It's m, the number of variables, log n, times h to the m.

Now, h to the m seems big, but again, it's n squared, times the time it takes to compute f once, which is, again, n squared. So it is polynomial. And now the verifier runs in time of n squared. So that's just to illustrate. Yeah.

STUDENT: [INAUDIBLE] runtime has to be [INAUDIBLE] cubed [INAUDIBLE].

YAEL KALAI: Yeah, right, and he is. He is actually n to the fourth in this example, because he needs to compute this-- OK, let's see. He's more, actually.

He needs to compute this thing. So he needs to do it for every i, j, k in-- not just the compute, to do the entire Sumcheck. So in the Sumcheck, he needs to h to the m, which is m here is going to be 3 log n in this. So it's already m cubed, and then he needs to compute t of f, the polynomial, which is n squared.

So he's n to the fifth. Because computing this once is n squared without the sum. But he also needs to run over all the sum and then also log factors. Yeah, he's running more. But it's considered doubly efficient because he's still polynomial overhead. Yeah?

STUDENT: Is there a way to get into the omega or something [INAUDIBLE] runtime [INAUDIBLE] that's the actual runtime?

STUDENT: Even n cubed.

YAEL KALAI: Yeah, even n cubed, you're saying. Yeah, currently. So n cubed, you can get it using recent kind of really nice works, with just optimizing a little bit.

That's the work of Dan Sung and friends, they get a linear time. So they can get an n cubed. I don't know how-- but it may be that for this specific protocol it's possible. Actually, I don't know. It's a good question.

But let me, actually, before you get too excited about it-- so yeah.

STUDENT: [INAUDIBLE] the number of rounds is log n, right?

YAEL KALAI: The number of rounds is log n.

STUDENT: [INAUDIBLE] actually get a--

YAEL KALAI: Great. Great. Question of two messages. So when I finished-- where's Gabe? So you came to me last time, right?

So when I started talking about this last lecture, Gabe came to me, rightly so, and said, well-- we just started talking about it, and he's like, well, OK, but you can do it so much easier. And it's true. This was kind of as an example to show.

But actually, going since you asked the question also, actually, there's very much simpler protocol that Gabe noticed. So let me just write it here. The idea is that the prover will only send two matrices over. So it's a bit communication complexity. Here, the communication complexity is polylog.

But if we just care about verifier runtime and we don't care about the communication complexity, then the prover can just send the verifier the-- so let A be the adjacency matrix. Let me now instead of f write it in the more classical notation of a matrix A, the adjacency matrix, where there's a 1, if and only if there's an edge, and 0 otherwise. The prover will send the matrix A 2, which is A squared-- he can also send A, but the verifier knows A-- and A3, which is A cubed.

Now, the number of triangles is just the trace of A cubed. So it's very easy to compute. You just compute.

Now, how do you know that this is A squared and this is A cubed? Well, we saw last time a protocol, a randomized protocol for checking equality between matrices via this Reed-Solomon kind of decoding idea. So you do that. So there you go.

STUDENT: That's really, really nice, but I'm not happy.

YAEL KALAI: But then blame Gabe, of course.

STUDENT: [INAUDIBLE]

YAEL KALAI: Exactly. You're right.

STUDENT: [INAUDIBLE]

YAEL KALAI: You can get-- OK. So there's a really beautiful work of R, R, R, and they get constant round complexity. And so using their particle, probably you can get very good. But I need to think about the parameters. I think you can get in constant round probably really good parameters, but I need to think about the details. Yeah. OK.

So this was just a motivating example to show you that you can take something also with no structure whatsoever, like adjacency matrix. It's a no structure. It's arbitrary. But using the technique of a low-degree extension, you can add structure to it, and then you can use the Sumcheck.

So more generally, I want to say, I think what's really the power of this is in error correcting codes, what's nice about error correcting codes is if you cheat, you need to cheat everywhere in the sense. Things kind of percolate.

In proof system, this kind of thing is very, very useful because we want to-- proof systems about catching cheaters. When we construct a constructive proof, our goal is to catch a prover, a cheating prover. And so you want some encoding. You want to say, if you cheat somewhere, he has to cheat everywhere.

So this idea of having-- in some sense, you want kind of-- if you want the proof to-- it's very natural. Some encoding comes into proof systems, and we'll see that next when we look at the GKR protocol for low-depth computation. So that's coming next.

Now, before I start this protocol, I have a question for you. So last time-- this is a long class. It's a three hour class. It's tiring. And it's technical, which is even more tiring.

So now I want to ask you two options. Option 1, we'll take two breaks, one after one hour, one after the second hour. It will give you-- or we take one in the middle, which is maybe a little longer. How many people are for two kind of short breaks to decompose? Let's take a break now for five minutes and then we'll do GKR.

OK. So so far, the applications, we saw for the Sumcheck and even if you want the low-degree extension, were both kind of counting. We said, let's count the Sharp-SAT, let's count the number of triangles. So you may think, OK, well, Sumcheck is about counting, so yeah, I can see how you can do counting problems by either they're already kind of low degree and you just apply Sumcheck automatically, or they're not low degree and then you do a low-degree extension, and now you have the Sumcheck. So maybe you can do counting things, but that's it.

So now what I want to show you next is that, no, actually you can do a ton with that. So just the low-degree extension and Sumcheck together, that's all you need. And what I'm going to show you now is how to do a doubly efficient interactive proof for all low-depth circuits. So let me explain.

So suppose you have here some circuit C. It's really, really big, but it's shallow. OK, so think of this circuit size S, which is really big. The depth is d, which is very small. Now what I'm going to show you is a doubly efficient, interactive proof where the verifier runs in time only d, or linear in d or d and poly log in S.

And the prover, of course, he needs to compute the circuit. He runs in time S. He has to run the computation. And only poly overhead, so poly of S. So I'm going to show you a doubly efficient interactive proof for any bounded depth. Yes.

STUDENT: Just to clarify here, circuit size is just the number of gates?

YAEL KALAI: Good. The circuit size is just number of gates, yeah.

STUDENT: Thank you.

YAEL KALAI: Yeah. Now, note, this already seems weird because-- OK, so the prover, I said, the goal is the prover is going to convince the verifier that, let's say, so the goal is to convince the verifier that C of x equals, let's say, 1. Think of C as any Boolean circuit of small depth. And the verifier should be very efficient.

But now you're saying, just reading C is not efficient. If you have an arbitrary circuit, just reading the circuit is not efficient. You're right.

So for this to be meaningful, C needs to have a succinct representation. So it needs to be kind of a uniform circuit that you can specify it succinctly. And indeed we will assume that C is what's called logspace uniform, which essentially means that there's a logspace machine, a log S space machine, a uniform Turing machine that generates the circuit.

But actually, for now, don't think about. Just suppose it's just kind of there's some uniform condition. So the verifier has succinct representation of the C, because it will only come much, much later when you'll see this-- you'll need to deal with this issue.

So the goal is for the prover to prove to the verifier that C of x equals 1. And the idea, the way we're going to do this essentially is, the high-level idea, is we're going to-- so I want to catch the prover. He's cheating. How am I going to catch him? I want to make sure he's correct.

So the prover comes to me and says, look, C of x equal 1. The value here is 1. I want to reduce the claim on the output value to a claim about something in one layer below, by the small cell protocol. And then I want to reduce the claim about something about layer below to layer below and layer below and layer below and layer below and so on, until I get a claim about the input, which the verifier can compute on its own. That's kind of the idea.

So for this, the first thing we assume, and this is without loss of generality, that C is layered. What do I mean by layered? There's an output gate. There's kind of that let's call it layer 0.

And then you go to gate, and then there's kind of wires to one layer below. We'll call it layer 1. And then there are gates to one layer below. We'll call the layer 2.

So there's no kind of long gate that goes from the input straight to-- it's really layer by layer. So that's the first assumption. So we can talk about layers. So what it means is that gate in layer i is connected only to gates in layer i plus 1, where we think of i equals 0 is output and i equal d is input layer.

So this is just for the sake of simplicity. You can always add dummy gates to make sure that each gate is only connected to one layer below. If before it was connected to someone closer to the input, you just add dummy gates to go up, up, up.

So again, let me just make sure everyone's on board. When I say a gate is in layer i, essentially what I mean, the depth from there to the input, the number, the path, the length of the path is i. So any gate that is kind of i away from the input, it's connected to gates that are only i minus 1 away from the input.

So that's an assumption I'm going to make. And the way I can realize it is by just if the original circuit was not in this way, I'm just going to add dummy gates in the middle. So if a circuit at layer 3 was connected immediately to the input, I'm going to just copy the input in 1, 2, and then I'm going to connect it. So things are going to be layered. Questions about this assumption?

OK. So now the circuit is layered, and now what I want to do, I want to reduce. So now I'm going to have an interactive protocol. So the idea in GKR is to have an interactive protocol that consists of d sub protocol, and each sub protocol essentially reduces a claim about some layer i to a claim about layer i plus 1, so a claim about the layer that's one closer to the input. So these are how these sub protocols will go.

I'm going to explain the GKR protocol formally, but this is just intuition. So let me take this, actually. So the first thing is-- here is here is kind of the idea.

So, again, this is not the protocol. I'm giving you an intuition of, helping you understand where I'm going to go. So we want to prove a cheating prover. Suppose he's cheating. He's trying to argue that C of x equals 1.

But think of it, it's not. We want to catch him. How do we catch him? If he's cheating, how do we catch him?

So one idea is to say, well, if he's cheating here, he must-- so let's say this is an AND gate. Suppose it's 0, but he's claiming it's 1. So it means that I'm going to ask him, you know what? It can't be, if I tell him, give me both of these. He also must be cheating.

So I'm going to tell him, you know what, 1 is 1. So in that case, you claim that both of these are 1. He's like, yeah. Or if it was an OR, I'm saying, this is 1, so you claim that one of them is [INAUDIBLE]. He's like, yeah.

But in reality, they're both 0. So now, what I can say, you know what? OK, fine. I'm going to choose one of them at random. Really? OK. Tell me that this is a 1.

One of these has to be a cheat. So I'm going to press him-- really? So show me. Show me this. And then I'll go down-- show me, show me, show me, show me. And then he's stuck because it's an input. So this is very easy approach.

The problem with this approach, I lose probability 1/2 in each layer because suppose if it's an AND, let's say it's a 0, then he tells me one of them is zeroes. But in reality, I know they're both 1. So I'm going to tell them, really? Both of them are 0?

Fine, I'll choose one of them. Show me this is 0. But maybe this really was the 0,1, and then he'll, of course, cheat. So I'm like, finger crossed, this was actually a 1, but only with probability 1/2.

So that idea works very well. It's just the sum is 1 over 2 to the d. So that's not good because 1 over 2 to the d is the size of the circuit. I need to-- that sucks. OK, no good.

So what do we do? So here is the basic idea. The idea is, you know what? Don't ask per gate, per gate. That's not a good idea.

Let's ask something about an encoding of the entire layer. So instead, I'm going to have to give me something kind of about the low-degree extension of this layer. So I'm not going to ask, reduce it. Tell me this. Tell me this.

No, tell me something about this. And then I'm going to argue, if he cheated here, he must also cheat here with very high probability. And again, if you cheat here, I'm going to argue that you must cheat here with very high probability, and so on and so forth. That's kind of the high-level idea.

But now you can say, well, this is a bit confusing. Before, when I said, if you cheated here, I'm going to ask him open cheat on one of these, it's very easy for me to check. If you say, this is 0, and it's an AND, you're claiming that one of these is 0. But I know one of these is 1. So it's very easy to argue.

How do I go-- So I can just say, OK, prove to me that this is a 0. That's easy. How do I go from arguing about the low-degree extension of a layer to a low-degree extension of another layer? What do I ask you even?

And it turns out that what I do here is really low-degree extension and Sumcheck. So I go from here to here. It's just a Sumcheck. So that's what we're going to see. That's kind of the high-level idea. Any questions? Let me just make sure I didn't forget to say something. OK, good.

STUDENT: What's the goal for the verifier runtime?

YAEL KALAI: Good, OK. What we're going to get-- OK, goal, as efficient as possible. What we get is the verifier's runtime and the communication complexity will be d, the depth, times polylog the circuit size. And the reason we paid this depth is because in the protocol, we're going to run d sub protocols. We're going to go from a claim on layer i to another claim of layer i. And each kind of reduction in the layers is going to be kind of a Sumcheck protocol, which will consist of log S rounds or log S-- the variable's going to be log S.

So eventually, the runtime of the verifier will be something like the depth times log the circuit size. And the prover is going to run time poly in the circuit size. But we'll go over all these parameters once we see the protocol. But that's kind of the theorem that we'll write when we get there.

Great. Thanks. Any other questions? Yes.

STUDENT: So the verifier isn't able to see the whole circuit.

YAEL KALAI: Right.

STUDENT: How does the verifier get--

YAEL KALAI: Good, good, good, good, good, good. So the verifier, as I said, he can't even see the entire circuit. All he has is some uniform Turing machine that generates the circuit.

So that's a very good question. How does he-- we'll see. The answer is, it's complicated, but we'll see. We'll see as we go along.

He'll use his kind of uniform access to help him out. It's really unclear how at this point. Great.

OK, so what do we do? So first thing, so step 1, this is just, again, for simplicity, arithmetize the circuit. Remember last time we had arithmetization? So arithmetize, all I mean here is convert an OR to-- convert an AND to multiplication, convert an OR into the sum minus multiplication. Just convert the AND/OR gates into addition and multiplication gate. So step 1, arithmetize C so that it consists of only and-- sorry, add, and multi gates.

Now, you're right. At this point, the verifier doesn't even know-- so we'll talk about how the verifier has access to these things later. But the prover, what he does at least, the prover first arithmetizes the circuit. So it's add, and now everything all the gates are multiplication gates and addition gates. Good.

Now, what he does, so I want to convince you that C of x we said 1, so I'll make it 1. What do I do? I'm the prover. I take my input x. I compute the values of all the gates in the circuit.

So what does the prover do? First, compute the value of all the gates of C. So now the prover computed all these bits, which are the values of the gates. Note, this is still over-- it's Boolean. It's over g of 2. Everything is 0,1.

OK. Next thing I do is-- the prover does, he computes the low-degree extension of each layer. So what are the parameters? How do we do it?

There's many ways to do it. Let me add, actually, another assumption just for simplicity. Let's assume that the number of gates in each layer is exactly S, like the size of the circuit. I'm just assuming that so I don't need to now have different S's. It's just going to be annoying.

Why can I assume that the number of gates in each layer is exactly S? Just add dummy gates or just 0. Nobody's going to look at them. I'm just going to-- it'll just make my notations easy.

So now, if there's less, just add zeroes, 0, 0, 0, 0. So now it's like this kind of circuit by just adding zeroes everywhere. OK, good.

Now, what does the prover do? He thinks-- so now, each layer consists of S values. That's an assumption, that we added demi gates. So now we have S values per layer. Many of them can be 0, if the layer was short, but there's S values.

So let's write these values. This is for layer i, let's write these values in a hypercube. So let's write these values as kind of a function from H to the n to 0,1, where H to the m is exactly S. So we said each layer has S elements. Write them in some hypercube where the size of the hypercube is exactly S, so you can write all these elements in the hypercube in some order that makes sense to you.

So if you want to take H to be 0,1, just take the binary representation of each gate. The gates are numbered, and Vi of this number is the value of this gate in layer i. So each gate in layer i has kind of a number, which is a number in H to the m, and this is the value of the gate.

Now, what is H and what is m? I said it needs to be of size H to the m. H to the m needs to be S so I have place to write all the values of the layers. But which H and which m do I choose? And now, actually, there's freedom.

So many times, it's easier to think like we're used to binary because we're kind of computers. So it's comfortable to think of H as being 0,1 and m to be log S. You can think of it this way. That's fine.

However, later, like probably Rachel will tell you about it, it'll be convenient, actually-- Rachel is like, oh no, what did I sign up for. It's like, I'm not going to tell you.

It'll be convenient, actually, to take the parameters, which are H being log S and m being log S-- fine. I tried to put in the same, didn't work-- m being log S over log log S. So it'll be convenient to think of these parameters.

So first, just note that log S to the power of m, which is log S over log log S, is just-- this is just 2 to the log log S times this thing. These two cancels, and therefore you get indeed S. So I'm just saying, these parameters, indeed, if you take H to the power of m, you get S back because H log S to the power of m log S over log log S, log S is like 2 to the log log S, and then the two log log S canceled, and you get two to the log S, which is S. So this is good. You get what you want.

So you can be like, why am I taking this setting of parameters? And you also may be kind of annoyed, like, why are we talking so much about parameters? This is kind of boring. And I agree.

However, this is important because these parameters are used a lot and for a good reason. So let me tell you why these parameters, why am I spending your time on these parameters, this specific choice. And the reason is that later, when we'll take extension-- remember, we take an extension over a field. We'll take a big field.

Now, the field, if you remember the Sumcheck protocol, the soundness you get is like m times d, the number of variables times the degree over f. So f needs to be of size at least m times d. So it will need to be at least like this m, which is, if you take binary, like 0,1, it'll need to be at least log S, which is the m. Or here, the number of variables is not constant usually, and it'll need to be bigger than the number of variables.

And now the problem is if you take-- so what's very desirable in later applications, in particular for PCP applications, which we'll talk about, it's really beautiful. For those who haven't heard of what PCPs are, we'll learn about it. It's really, really nice.

But for these applications, what you want is that f the size of the extension you take is poly in H. And now if you take H to be 0,1, which is binary, which is what we all like, then the field becomes constant size because it needs to be for applications. Like, to get the PCP, you need f. If f is bigger, you'll get a super polynomial object.

So for later applications, it's important that the field is not too much bigger than H, that it's polynomially related to H. And then if you take H to be small, like constant, like 0,1, then f is a constant, and then you don't get sums because m is not a constant. So you get sums bigger than 1. You got nothing.

So if you didn't follow, it doesn't matter. You will see it when we talk about PCP. But this is just to tell you why I'm choosing these weird parameters as opposed to 0,1. The point is that later we'll need these parameters when we'll talk about PCP. And so that's why we're using them now.

STUDENT: [INAUDIBLE] GKR itself?

YAEL KALAI: GKR itself you don't need.

STUDENT: Of course, you're never writing down the entire--

YAEL KALAI: Exactly. Exactly. Because in GKR, it's interactive. You're never writing down the entire thing. For GKR itself, you can take H to be 0,1 and you can take f to be big and the kind of parameters you want.

I was actually debating if to teach it with 0,1, and then to change all the parameters. In many-- if you look at Justin Thaler's notes that are in the website, I gave the exact chapter, then he uses 0,1 because it's easier. But since I'm going to change the parameters anyway, I thought I'll just start with m. And I think it's good to get accustomed to these type of parameters because they show up a lot.

So this is just kind of-- if you didn't follow exact choice, don't worry about it. It turns out this is the choice I want. And why, you'll learn about it when we talk about PCPs. But if you want to think about H as 0,1 and m as log S, you can do so. All the lecture, everything I say makes sense in that parameters, and you'll understand everything with these parameters as well for today. So good.

So now what will the prover do? The prover will-- he has for every layer, he has the values of the gates. He computes the values of the gates of that layer, which you can think about it as for each layer i, the prover computes Vi, which is the value of all the gates in layer i.

So we can think of the gates as going from H to the m to 0,1. Each gate has a label in H to the m because it's kind of the same as it is amorphous to S. So each gate has a label here. And this is just the value of the gate on input x.

So this is the proof. There's no protocol yet. I'm just telling you what the prover computes before the protocol begins.

And now the prover can compute the low-degree extension. So this-- let me actually be careful with room. So I'll write it here. This is nothing but the low-degree extension of Vi. OK, now we're in business.

Once the prover computed this, then we're in good shape. So now what? Now let's go back to the protocol. What do they do?

So as we said, the goal is to reduce. So remember, the first thing the prover gave the verifier-- you can think of the protocol started with V0, which is the output layer, and one element. Let me call it Z0, because we're going to call them Z, and Z0 in H to the m in this time, which is kind of-- actually, the output layer has just one gate. It's 0,1. We added a bunch of zeros to it because dummy gates assume that it's kind of size S.

But there's one gate. Let's call this gate Z0. And he's arguing on Z0 there's a certain value V0. In our case, we said it's 1.

So he comes with an argument. He tells it, look, if you look at the output layer-- this is the output layer. If you look at the output layer and the relevant gate and the actual single, not dummy gate, I claim the output is V0, namely, let's say 1.

Now, I say, why would I trust you? And he said, OK, let me help you. Here. I can always add tilde.

So now let's see. What is V0 tilde and any element Z? What is this value? Remember, this is the output. Yeah? So it is what?

And what I'm saying here is actually general. Even let's think of i, layer i. You can think of layer 0 if you want, but in any layer. Take any layer, and take any Zi, even in the extension, in f to the n. Here, it happened to be that the Z0 was actually kind of not in the extension even because we started with a certain gate.

But let's think of it more broadly because later we'll go to will be in the extension. So the prover claims that this is Vi. We start with i equals 0.

I don't know if that's true. I want to reduce it to a claim about layer i plus 1, one layer down. How do I do that?

What is Vi tilde and Zi? Zi in general is in the extension. In this case, it's not, but in general it's in the extension.

So first thing, let me move Zi not in the extension. So let's remember, what is Vi tilde? Vi tilde is sum. Vi, that's kind of the original Vi, and H-- let me make sure I'm consistent with my notation so I won't confuse myself-- and point P in H to the m times this chi P of Zi.

I just kind of-- so what is the low-degree extension of V? It's the Vi on every point, not in the extension, in the cube, in H to the m, times this kind of chi function, which was this-- oh, I erased it. But it's the-- yeah, this multiplication. Each one of them was this degree H minus 1 function.

But what is Vi? Vi, we can easily relate it to the layer one below. Vi is either an add gate or a multiplication gate. If it's an add gate, it's an add of two elements below. And if it's a multiplication, it's a multiplication of two elements below.

So now, to connect it to the layer below, and this goes to the question of what is the verifier know or doesn't know. What I'm going to use is the circuit has-- so the circuit is associated for every i with an add function and the multiplication function. And this function goes from 0,1 to the, I guess, 3 log S to 0,1. And it just says whether you take a gate in layer i and two gates in layer i plus 1, and it checks whether there's an ad there.

So it takes kind of i, j, and k, and it says 1 if and only if gate i-- sorry, i is taken. What notation did I use? OK. Sorry.

It takes a gate P, w1, and w2 in S, one of the gates, and it outputs 1 if and only if gate P in layer i is a add of gates w1 and w2 in layer i plus 1, and 0 otherwise. I start 0 and I go down to d. Yeah, and 0 otherwise.

So just check if there's an and there, if there's an add there. And mult is exactly the same. Mult checks if there's a multiplication gate. So mult also takes P, a point P in layer i, two points in layer i plus 1, and it outputs 1 if and only if this gate P in layer i is actually a mult gate, a multiplication gate, and it connects to gates w1w2 in layer ii plus 1.

So what is this? Now we can say this is nothing but sum of P. Let sum over all the possible w1 and w2, all the gates in the layer below, and see. Maybe it adds.

So if there's add, if add ipw1w2, if that's 1, then you have Vi. Then VP is just Viw1 plus Viw2. Again, if P is an add gate. If it's not, an add gate, you'll get a 0. So you're not adding anything.

If it happens to be an add gate, then the value of this gate is just the sum of the values of the gates below. If P is not an add gate, all this is 0. I didn't do anything, because I multiplied by add, which is 0.

OK. plus-- let me just open-- plus mult Ipw1w2 if it's a multiplication gate, and let's say it's connected to w1, w2, I'm still the sum. Then what do I do then? What's the value of VIP-- I mean, Vi in point P?

STUDENT: Vi and point w1 times [INAUDIBLE].

YAEL KALAI: Exactly, Vi and point w1 times, because it's a multiplication gate, Vi in w2. Sorry, plus 1. Thank you. Thank you.

Yes, great. Good. And all of these times this polynomial Chi P of Zi. Good.

So now where are we? We want to prove something about layer i, but we managed to write it as sum of elements in layer i plus 1. That's great. That's really what we wanted. We want one layer down.

And note, this is really similar to a Sumcheck. What do we have here? We want to prove that Vi-- so I want to tell the prover, prove to me that instead of-- he claims Vi is this value. I'm like, I don't know what this value is.

Prove to me that Vi is equal to the sum. So prove to me that Vi is equal to something about the layer below. How does he prove that something is equal to the sum? Sumcheck, right?

So do the Sumcheck protocol. And now we're saying, wait, but where is the polynomials? No problem. Everything's over h of the m. Look at the extensions.

This is already polynomial. We don't care about it. That's good. Look at the extensions.

We have a low-degree extension up here. Any polynomial can be extended. Look at the extension.

So now we want to argue that V_i equals the sum. This entire thing-- this entire thing, I argue, is a low-degree polynomial. Why is this a low-degree polynomial? Let's see.

First of all, it's a polynomial over H to the 3m because m here and two w's here, so we're talking about 3m variables. But Mm is small. This is good. m is log S over log log S, so that's good.

What about here? So let's first look at the degree. The degree of these things, it's the low-degree extension. It's H minus 1 of each one, because I'm really taking each one is H. Each add, everything is kind of-- oh, add is even better because we did 0,1, so add is even better.

You could do-- oh, sorry. Can I take it back? I also wanted to add an H. Everything is in-- I made a mistake of putting 0,1, but I need to be consistent. If I'm in, I'm in H.

So everything is in H. So everything here is degree H minus 1, so good. So it's more than H1 is 1 because I have your multiplication. w1 appears here, w1 appears here, so it's going to be degree 2H minus 1. Fine.

But H minus 1 is small. It's like log, so two logs. So the degree is small, the number of variables small. Perfect. Just do Sumcheck.

So now you're asking, what is the GKR protocol? Sumcheck. That's it.

You start with V0. You tell him, do Sumcheck. Convert this claim about the output, which the verifier has no idea. It's so far away from him. He doesn't know what-- let's do the Sumcheck protocol to reduce this claim to a claim about-- about this thing, essentially.

But now this is very confusing because what happens when we do the Sumcheck protocol? When we do the Sumcheck protocol, we reduce, if you remember-- let's look for a second at the Sumcheck here. We want to prove that the sum of something there, it's a huge thing. Add V plus mult, multiplication, all this craziness, whatever. Low degree is equal to beta, is equal to something, in that case, V0.

We run the Sumcheck protocol. At the end, the verifier needs to compute to check. In the Sumcheck protocol, we assume that the verifier has oracle access to the polynomial. But essentially, he needs to compute this function, the polynomial, at some point.

Now, maybe he has oracle access at that point. Great but maybe. But there's no oracles in this, in GKR. So what do we do now?

But here's an important observation that we'll use. Note, yes, the prover needs-- sorry, the verifier. When he verifies the Sumcheck, he needs to compute the polynomial on a single point. But this point t, t1 of the tm, is a random point that's chosen by the verifier.

What is the Sumcheck protocol? The verifier chooses a random t1. He gets a polynomial, then chooses a random t2, and so on and so forth. At the end, he needs to evaluate f and t1 of the tm that he chose. We will use this.

So now let's see. What happened here? After we did the Sumcheck, the poor verifier needs to compute-- let me again, make sure I'm consistent with my notation. Good.

The verifier now, so he does the Sumcheck. Let's denote kind of the randomness that he used in Sumcheck by a-- so we started with i. Let's say that when we reached here, let's say, he needs to compute a add tilde on Z_i plus 1, because we moved to layer i plus 1, 0 Z_i plus 1, 1, Z_i plus 1, 2 times V_i plus 1, and another-- and so that was one.

So Z-- don't like this notation. Sorry. Let me change it. It's more cumbersome on the board than in the-- OK.

Let me call Z_0, Z_1, Z_2. So suppose they do a Sumcheck protocol. In the Sumcheck, it's on three n variables. So let's denote the randomness used by the verifier and the protocol by Z_0, Z_1, Z_2. Each of them is in f to the n.

So now the verifier needs to compute this function and the points Z_0, Z_1, Z_2. What is this function at this point? It's add times V_i plus 1 tilde on Z_1 plus V_i plus 1 tilde on Z_2 plus mult Z_0, Z_1, Z_2 times V_i plus 1 Z_1 tilde times V_i plus 1 Z_2 tilde, and then all this times this chi. I'll put it here, chi p of Z of-- oh, oops.

Sorry, I messed this notation. This is the Z_i. OK, sorry. Z_i is the original point that we started with. We started a claim about V_i equals V_i tilde of Z_i. Z_i is a fixed point that we started with.

The randomness of the verifier in the subject protocol I denote by Z_0 Z_1, Z_2. These are chosen by the verifier. Yes, Vinod?

STUDENT: The V should be Z_0, right? [INAUDIBLE]

YAEL KALAI: B?

STUDENT: [INAUDIBLE]

YAEL KALAI: Yeah, exactly. But I'm in layer i. You're right. In layer 0, it's just Z_0.

STUDENT: No, no, no, no. The V, the V, you replace it with Z_0 in this notation. [INAUDIBLE]

YAEL KALAI: Oh, I replaced--

STUDENT: [INAUDIBLE] so P is Z_0, w1 is Z_1. Right?

YAEL KALAI: Yes. Yes.

STUDENT: [INAUDIBLE]

YAEL KALAI: Yes. Yes, yes, yes. Through the name you're right. Yes. Yes. Z_0, yes. Yes, you're right. You're right.

You're right, and this is still a low-degree polynomial in Z_0. Good point. It's sitting-- yeah, maybe I erased it, but this. No, but I didn't write this.

STUDENT: We need it though below the--

YAEL KALAI: Below here? No, but I need to write-- it's like this is xi minus-- I think. Yes.

So it's-- oh, maybe I need to take extension of this. Sorry, one second. This is-- hold on.

Sorry. Maybe I missed this. Let me think because I missed in the notes. So hold on.

So here the point is H. I think I need to take extension of this too, because the point here, this is not a polynomial in H. OK, so let's do that carefully. Sorry, I missed that.

So chi is a low-degree polynomial, but it's a low-degree polynomial in x. It's not a low-degree polynomial in this H. H here is in the denominator, so it's not a low-degree polynomial in H, sadly.

But you can extend, so it's OK. Don't worry. So it's not a low-degree polynomial. However, it is in-- all we care about is, if we think of it as a function of Z0 of p, then we just extend. We can-- I just want to make sure that I'm not missing something, so hold on one second.

Yeah. OK. Yeah. So the way to think about this-- sorry about that.

The way to think about it is to think of this as chi of P and Zi. It's low degree in this. It's not low degree in this.

But you can think about it-- it's in 0,1. You can think of Chi as 0-- sorry. You can think of chi as being in H to the m times f to the m to whatever to--

STUDENT: [INAUDIBLE]

YAEL KALAI: I'm not sure how to, because-- maybe, but I'm not sure. But there is, actually. The low-degree polynomial is one of them. So the answer is yes, there is. It's the low-degree-- yeah.

So it's defined only on H. You can think of it as defined only-- this chi is an extension. This chi on its own is extension.

What is this chi? This chi is a function. You can think-- OK, sorry. Let me write it differently. OK, let me fix this. Sorry about that.

Instead of this, think of chi-- I'm just changing the notation-- H1 up to Hm Z, which is Z1 up to Zm. I don't want to put it down there. They're both here.

And now-- actually, I don't. So what are we saying? We're saying, so this is the extension.

So you're saying, what is chi? I'll tell you what Chi is. Chi is the extension-- OK, let me rewrite this.

Chi H1 up to Hm, Z1 up to Zm is just multiplication of chi Hi, Zi. So it's exactly the same thing. I just don't want to put Hi on the bottom, like I did before. So let me put it as a variable. I didn't change anything.

And now this original-- what is this? This is the low-degree extension of the function. Let me first define chi and H to the 2m. So chi is such that an H to the 2m, so for every Hi or for every H and Z in H to the m, chi HZ is equal to 1 if and only if H is equal to Z.

I only define an H to the m, now extended. That's how we define chi, the unique low-degree extension. What is exactly the extension? Yeah, it's this multiplication, blah, blah, blah.

I don't want to go into it. But it's just a unique extension. So now with that-- and the reason there is a unique-- actually, I should say, this is kind of chi 1.

And the reason there is a unique extension-- this is kind of H1, Hi, Zi. And the reason there is a unique exception is from the Lagrange interpolation theorem. So it's just a multiplication of the unique kind of Lagrange interpolation extensions.

So all I did here-- so sorry about this, but all I did here is I changed the notation from the H kind of as a subscript to H as input. And now I'm treating them both symmetrically, which was important because otherwise. So thank you. Wow, you guys are great.

So now let's fix the mistake here. So this is chi of P and Zi. We're summing over P, but Zi is an input. That's the input we started with. That's the claim. Good.

But for me, chi is a low-degree polynomial already. So this is just chi of Z0 and Xi. Thank you. Yeah.

STUDENT: I guess what I still don't understand is isn't chi not a polynomial in the first variable since--

YAEL KALAI: Why?

STUDENT: So the definition for chi, doesn't it only appear in the denominator?

YAEL KALAI: OK, so let's look at the definition of chi. The definition of chi and m variables is just-- so chi takes two variables. The definition of 2m variables, just the multiplication, i goes from 1 to m of a chi 1 on Hi and Zi.

So now I need to define what is chi 1. Oh, you're saying it's two variables. You're saying it's not one variable.

STUDENT: Or rather more like in chi 1, isn't it not a polynomial in Hi?

YAEL KALAI: It's a polynomial? No, I only think of it as a polynomial in Hi and Zi. Chi-- sorry. OK, maybe I didn't understand your question.

Chi 1 is a two-variate polynomial. It has two variables. So you're right. It's two variables, not one, because I said it reduces to Lagrange.

You're saying, no, Lagrange is one. I'm thinking it is 2. That's what you said, right? Or maybe I didn't--

STUDENT: I guess my question is more emphasizing, when you write out the definition for chi 1, are we going to get that it's not a polynomial in Hi?

YAEL KALAI: No, it is. It is. So I want to--

STUDENT: Is it going to be a ratio of polynomials?

YAEL KALAI: No, no, no, no. No, no, no, no. So what is exactly chi 1? How did it-- chi is well-defined. It's just a product of chi 1s.

Now you can ask, what is chi 1? If you understand what chi 1 is, then you understand what chi is, because chi is just multiplication. What is chi 1?

Chi 1 is a bivariate polynomial. It's defined over f 2m to f. And so now you're asking, how is it defined? Let me tell you how it's defined an H first.

So if both of them-- first of all, it's a polynomial. It's a two-variate polynomial. So it's a polynomial and two variables, and both its degree H minus 1.

	Now you can say, how is it defined on H? On H, I'll tell you. It's 1 if and only if they're equal. That's how it's defined, 0 otherwise. So I defined it on H.
	Now you can ask, how do I define on f? So on f, I'm going to define it in the unique way that extends it on H. That's how it's going to be defined. So chi is a low-degree polynomial in each and every variable. Before, I wrote it only on Z. But that wasn't good because I need it also to be low degree on H. So-- yes, Vinod?
STUDENT:	Just the point, the point is that the way it was written down previously had H in the denominator. It's not a polynomial.
YAEL KALAI:	Exactly.
STUDENT:	[INAUDIBLE] it's not a polynomial.
YAEL KALAI:	Exactly.
STUDENT:	And this one is.
YAEL KALAI:	OK. So I write it-- what's important to me is that chi 1 is a polynomial. It's a two-variate polynomial. Both of them are degree H minus 1.
	So with that, if we believe-- yeah.
STUDENT:	If they're not equal, is it 0?
YAEL KALAI:	Yeah. Yeah, yeah, yeah, yeah. So if H and Z are equal, it's 1. If they're not equal, it's 0. Yes, great.
	Thank you. Thank you, guys. You're great. Fantastic.
	So now we reduce this big sum to this. Because we sum over the P, so the P becomes Z0, this. Wow, what's going on here? What is the verifier supposed to do?
	So before, he need to verify one little point in Vi tilde, and now he needs to verify this mess. But let's deconstruct. Yes.
STUDENT:	Sorry. Is the argument [INAUDIBLE] chi [INAUDIBLE] Z1 or Zi?
YAEL KALAI:	This is good. Sorry, this is Zi. This is the claim we started with. We started with a claim on layer i. You can think of i as 0 in the beginning.
	But we start with the claim that Vi is equal the low-degree extension of layer i in point Zi. This is the Zi here because-- why is it the Zi? Let's see. What did we say?
	We said that Vi, we need to argue that it's this. But this is just sum-- that's the definition of a low-degree extension, sum of the points times this. Let me change this to P Zi.
	The Zi I is fixed. This is fixed. This is from the claim. Good.
	Now we say let's sum over the P, so we sum over the P. The P is not fixed. It's summed over.

And now we say, let's instead of this, let's sum over the two w's, which are the possible kids. All the possible pair of kids-- oh, I think I didn't say. I assume that C is fan in two, that each gate has only two children. And that's, again, with all this generality, you just increase the depth by log factor. So I'm assuming fan in two, so it's only two. Each gate has either an addition of two or multiplication of two.

OK, so this is what we get. And now when we do the Sumcheck protocol, the P and the-- so we sum over P, w1 and w2. At the end of the day, we need to compute this polynomial. The verifier will need to compute this polynomial. Instead of over some fixed P w1, w2, it's over some-- I called it Z0, Z1, Z2 chosen by the verifier. That's kind of the randomness of the verifier.

So P becomes Z0. w1 becomes Z1, and this is Z2. That's kind of the after. Yeah.

STUDENT: I'm very confused because then, if i is equal to 1 [INAUDIBLE]?

YAEL KALAI: I know, I know.

STUDENT: So that's the different Z.

YAEL KALAI: Yeah, I know. I'm so sorry. These are stars.

STUDENT: OK, now, that makes a lot more sense.

STUDENT: It's a constant.

YAEL KALAI: It's a constant. Yeah, I know that. Sorry, that's a bad notation on my end. Yeah. This is a very, very different Z. It's like a totally different animal.

These are random Z's chosen by the honest verifier. This is kind of, I don't know, a Z star that was fixed ahead of time. So now, the verifier needs to verify this. Yeah.

STUDENT: [INAUDIBLE] is fixed. Can you just use the same chi [INAUDIBLE] before and put it in the subscript of the chi? So then you don't need the extension or whatever. You can just write it out [INAUDIBLE].

YAEL KALAI: The thing is that before, I assumed that the chi, when I fixed the H, it was an H to the m. This isn't the extension. It's not going to be an H to the m. So I think we should keep the two, yeah. But that's a good idea.

OK, so what is this? So it seems like we had something that looked really kind of reasonable, and we ended up with something that's horrifying. So what does the verifier do now?

Now the verifier is like, I just need to verify that this equals some value that we got out of the Sumcheck protocol. So we did the Sumcheck protocol, we got some value, whatever. I don't know what to denote. I'm now scared to denote anything. So this equals something.

Now, what is he supposed to do? He's supposed to verify that. What do you do?

So here is the idea, and then we'll take a break. So he needs to verify a lot of things. First, he needs to compute this add tilde and mult tilde. This goes back to your question now. Add tilde, these are a function of the circuit.

He doesn't have the circuit. How does he compute these add tilde and mult tilde? Turns out, actually, this is sad news. He actually can't compute. We don't know how to compute. He doesn't know how to compute add tilde and mult tilde.

Even if he gets this kind of uniform description of the Turing machine that generates the circuit, he can't compute them. Let's deal with that later. Suppose he has oracle access.

Suppose someone-- there's a trusted oracle that lets him compute these values for now. In the paper, it's called the bare bones protocol. So suppose somehow, magically he has a trusted oracle, but only to compute add tilde and mult tilde.

Note, these are only functions of the circuit. They have nothing to do with the claim of the prover. It's just information about the circuit. It's whether kind of-- not in the extension, it's whether this gate is connected by an AND to these two gates, yes or no, or by a multiplication to these two gates, yes or no.

So it's not enough for him to compute that. He needs to compute the low-degree extension of that. But this has nothing to do with the prover. This is just a function of the circuit.

So suppose first-- so first, we're going to assume this is not without loss of generality. But just for simplicity, we're going to assume that we can compute. add tilde and mult tilde. Suppose he can.

Actually, what we're going to do today is we're going to show the entire GKR protocol, assuming you can compute this. And then only next time, we're going to show how to compute this thing. And actually, there's-- we're going to show that next time. How can we help them compute it or how can we compute in a different-- how can we deal with this issue?

But for now, let's say he can compute this. So the verifier is like, OK, let's see. This I can definitely compute. That's easy. So the verifier can compute this.

He can compute this. He can compute this. And what does he's left? He's left with two things he can't compute, V tilde and i plus 1-- V tilde and i plus 1, and point Z1 and Z2.

So now, how do we continue? What we're going to say is the following. Suppose in this sum he was cheated, namely the Vi value, Vi star is not correct.

By the Sumcheck protocol, it means that this entire value is also not correct with very high probability. I mean, the probability depends on f. You'll take f large enough that it can be with very high probability.

So you're left with a false claim. Now, what the verifier is going to do, the verifier is going to tell the prover, you know what? Tell me what these two-- you gave me something I don't know what to do with it.

Tell me the values. You tell me the values of Vi plus 1 tilde and Z1 and Z2. So the verifier, the prover comes in, OK, sure. V tilde and i plus 1 and Z1 is Vi plus 1 star 1, and V tilde and Z2-- sorry, i plus 1, is V star i plus 1, 2. Here are the values.

Now I'm the verifier. I'm the verifier. What do I do? I check. I check that add times the sum of these two values you gave me plus mult times the multiplication of these two values you gave me is equal what we got.

Now, by the soundness of the Sumcheck protocol, we know what we got is false. It's not true. So the only way-- if the prover gives the correct thing, then he will fail. If you give the correct thing, I'll catch him. Because if he gives me the correct thing, I get the true value.

But I'm going to check him against the false value. Again, what do we know? He started with a false Vi star. We did sum of such Sumcheck, we're left with a false-- we said that this is something false because the sum of the Sumcheck protocol says if you started with a sum that's false, you'll get a false claim about the f.

So now I get that this thing is equal to some beta, but this beta is not true. So now I tell him, OK, I don't know how to compute this thing. Tell me what the two values in the low-degree extensions are. If you tell me this, assuming I compute this add tilde and mult tilde for now, then I compute this thing.

Now, the important thing to note, that if this value was false, he must-- if he gives me the true values, I'm going to reject them because I'm going to compute this thing and check if it's what he claimed it is after the Sumcheck protocol. And because he claimed the false value, if he give me the true things, I'm going to reject him. So the only thing he can do if he doesn't want to be rejected off the bat is to give me one of them false, or both false.

STUDENT: So he has to give two values.

YAEL KALAI: Right. Exactly. So where are we? What did we do?

After a lot of sweat, we started with a claim on layer i. We did manage to get down to layer i plus 1, as I promised you guys. Here's the price.

First, miraculously, we assume he can compute add tilde and mult tilde on its own, Assumed Why? Whatever.

Second, we went from one claim about layer i to two claims about layer i plus 1. That doesn't sound very reassuring.

STUDENT: Isn't that where we started?

STUDENT: Yeah.

YAEL KALAI: Exactly. It's where we started. This is a great time for a break.