ERIK DEMAINE: All right, today is our last lecture on the predecessor problem. And we did two lectures on upper bound, started with Van Emde Boas x-Fast, y-Fast tries. And then fusion tries was last class.

This time, we're going to show that those bounds are essentially optimal. And we're going to prove this bound. So we currently have min of log w, and log base w of n as an upper bound. That was Van Emde Boas and fusion tires.

Were going to prove an almost matching lower bound. There's a log log w factor here. And this bound holds for even a static predecessor data structure no updates, as long as the data structure has polynomial space, which if you have any hope of making something dynamic, you definitely want polynomial space.

Now, it's known that this log log factor is not real if you have, essentially, linear space or n log n space, or something like that. But that's much harder to prove. So this is where we will end our coverage of predecessor. But I'm going to start out with an overview of what's known.

And proving this lower bound is actually pretty cool, because at this point, especially seeing fusion tries, you might think bit tricks are kind of all about magic, and doing crazy things, and using these operations in bizarre ways they weren't intended. Bit tricks are necessary as an upper bound tool, given our weird historical precedent, which is computers are built to do arithmetic. And why not do bit operations too?

And so it's sort of an artifact of computers as we know them today. That's why we have the word RAM, because it's based on C, because it's based on computers that exist now. And that's legitimate in that that's the computers we have. So we might as well try to use them.

The lower bounds, on the other hand, are, in some ways, more beautiful, because they're just about how much information has to go between Alice and Bob, actually. This is the communication complexity perspective of data structures. And it's nice. We will get, in some ways, cleaner, simpler arguments because in lower bound land, you don't have to worry about, oh, can I use this operation or this operation to do?

It's just, is there any operation at all to do it? And then it's just about information. It's just about

information theory. And it's, in some ways, cleaner.

And this proof, in particular using a technique called round elimination is actually simple. And we will see in it the concepts of Van Emde Boas and fusion tries again, but without having to do any bit tricks to actually make them happen. So if you felt like all of these bit tricks are kind of weird, what is the real essence of the problem?

This, in some sense, provides an answer. And it says, well, really those were the right answers. That's what you should have been getting.

So let me start with the survey. This is more of a historical survey. So the first bound for predecessor problem, first lower bound, Asvab Atjai in 1988. And he proved that for every word size, there's a problem size such that there is an omega square root log w lower bound.

You can compare that with this one. This is log w over log log w. Now, this is a bound that works for all values of n and w, which is more interesting.

Here, we're saying that if you just want a bound in terms of w, you need at least square root log w. The original paper claimed log w. But that's not true, in general.

But the proof actually establishes square root log w. And in this proof is complicated. It was the first one. It got the field started. There's been many papers since then.

Next one was by Miltersen, who we've cited a few times. And so this a few years later. Miltersen, essentially, took the same proof, presented it in a more coherent way, and sort of really got to the essence of what the proof was showing, and could prove the same bound, but also a complementary bound, because there is a symmetry between word size and problem size, which we will see when we get to the communication complexity perspective, which will be right after the survey.

It's not completely symmetric. But Miltersen proved this lower bound that for every problem size, there is a machine, there is a word size, where you need omega cube root of log n. Now, we know how to get order square root of log n by taking them in. So this is matching up to the exponent.

And we know how to do log w by Van Emde Boas. So this is matching up to the exponent. So it's progress. But it doesn't give a complete picture of w versus n. And this is the paper that introduced this communication complexity idea, that this would be a useful concept for

predecessor lower bounds.

Then the following year, there's a bit of a breakthrough. So here, we've got the idea of using communication complexity. Then Miltersen and others had the idea of round elimination.

And so just treat that as a black box for now. We'll explain what is in a moment. But it proved exactly the same result, but in a clean way, using round elimination.

These proofs were messy. This proof is pretty simple. And it's the beginning of the proof that we will cover. But we're going to prove stronger bounds than this.

Let me go to another board. Next, we have Beame and Fich. Now is about the time when I entered the world of data structures. And so Beame and Fich came out just when I was starting, 1999.

And this proved for all of w exists an n. So log w over log log w, which is the same thing we're going to prove. Except this only proves it for certain values of n, whereas we're going to prove it for all values of n where this is smaller than that. So this is a bit more special, not covering the whole w n trade-off.

And then there's this symmetric version. There's going to be a lot of log log n's today. So get ready for log logs in general.

So for every n, there exists a w such that there's an omega root log n over log log n. So this is almost matching the min of fusion tries and Van Emde Boas, which is order square root log n. They also proved-- did a little bit on the upper bound side. And they found a static data structure achieving-- I'm just going to cheat and use some arrows.

So they achieved the min of these two. So in a certain sense, this is tight. This is not exactly what you'd want, because the data structures achieving the min, what you'd like is a lower bound of the min.

And it's not quite that. It's saying, there's a particular pair of w and n values where this is optimal. There's another pair of n and w values where this is optimal with respect to either one of these parameters. But there are other pairs of values of w and n where it may or may not be optimal. So it's kind of matching, but not quite.

The way to say it is these are the best-- this is the best pure w bound you could hope for. And

this is the best pure n bound that you could hope for. And it's the best mix of two pure bounds.

OK, then we go back in time a little bit, which is-- there's a funny chain here, which is this paper, 1995, cites this paper from 1999. This paper cites this paper of 1992. That's normal.

This one's a little bit out of order, bit of time travel. So I think this one was in draft for a long time, and took awhile to finish it. But this paper is before 1995. It's not cited by this paper.

What's interesting about it is it proves both of these lower bounds. I don't think it proves the upper bounds. But these were independently discovered by Beame and Fich in the future of this paper. This is actually a PhD thesis.

So I think it was lost, or not known about, for a long time. But it ends up establishing the same bounds. And then the real contribution here was that those were the best bounds that are purely w and n, but still not a full story.

Next up is Sen in 2003, which gave a round elimination proof of the same result. And whereas both the Beame and Fich and the Xiau proof are messy, this round elimination proof is very slick and clean. And this is the proof we're going to cover today.

And it's going to prove-- it will imply these pure w and n lower bounds. But it will, in some sense, in a stronger way, prove this bound, which is a real min thing. And so we're just off by this log log factor.

Otherwise, we were proving exactly the min of fusion tries in Van Emde Boas is optimal. Getting rid of the log log factor is messier. That's the next paper, which is the last-- it's two papers, actually.

But the final papers in this story are by Patrascu and Thorup, 2006 and 2007 are the two papers. And they give a tight n, versus w, versus space trade-off. I think I can fit it on this board. Let's try.

So it's the min of four terms. The first term is nice, log base w of n. That's our good friend fusion tries. The next one is roughly log w. So it's roughly Van Emde Boas. But it turns out you can do a little better. Now, I need to define a.

I'm going to assume here the space is n, times 2 to the a. That's the notation they use. It's a little bit weird. But it makes this bound a little bit easier to state.

So, of course, you need linear space to store the data. This is measuring words. So the question is, how much more than linear space do you have?

If you want poly log update, then this better be poly log n. This is about static data structures. So if you want poly log n, then a to be order log log n. So you can think of a being order log log n. Or you can think of a being constant, if you want linear space. Then this stuff-- I mean, then this is basically log w. OK, so that's roughly Van Emde Boas.

Next, we get two crazier terms. These ones, I can't give you intuition for, I'm afraid. You see, there's the log w.

They're, again, versions of Van Emde Boas like things. But they're improving by some factors that make you better when a is large. And, in particular-- I mean, they should match this thing at some points. So when a is order log n, that's when you have polynomial space, which would be very difficult to update.

You need polynomial time updates. But that's what this static data structure uses polynomial space. And so with polynomial space, if you plug in a equals log n, you end up improving by a log log n factor.

And so that's where we are. I guess there should be a square root in there somewhere. This is a log w over log log w. This is the part that diverges a little bit from Van Emde Boas. And this really is achievable if you have polynomial space, instead of like n or n poly log.

So for static, yes, you can get these log log n improvements. We won't cover them here. I mean, they're small improvements. And we're mostly interested in dynamic data structures.

And for dynamic data structures, these things don't turn out to help. So let me state the consequence, which is what I stated a couple of classes ago. If you have n poly log n space, so if you have polylogarithmic update, which is usually the situation we care about, then we get min of log base w of n. And I'd like to say log w, but you can actually do slightly better.

Oh God, indeed. I think I stated this two lectures ago. So log w is what we know how to do. This is a very small improvement.

Let me state another consequence of this, which is Van Emde Boas is optimal for w poly log n. OK, that's when it gets log log n performance. And so you can check in that case this term doesn't buy you anything.

That's log w would be order log log n, if you're poly log. So this disappears. And you just get log w. So that's one thing.

Fusion tries are optimal for log w order square root log n, times log log n. So this is saying w is at least 2 to the root log n, or actually log n to the root log n, if you want to include that term.

So that's a fairly large w. So for large w, fusion tires are optimal. For small w, Van Emde Boas is optimal. In between, this thing is optimal.

That's the reality of it. It's a little messy. But this is tight. So this is the right answer.

I think most of the time, the situations you care about, w is probably going to be small. So you should use Van Emde Boas. If w really large, you should use fusion tries. In between, you could consider this trade-off. But it's not it's not much better than Van Emde Boas. So take that for what it is.

So that's what's known. As I said, we're going to cover this sen proof, which will be tight up to this log log factor. And this holds as long as your space is polynomial. So it doesn't assume very much about space.

And it's going to be a lower bound for a slightly easier problem. If you can prove a lower bound on an easier problem, that implies a lower bound on the harder problem. The easier problem is called the colored predecessor problem.

In colored predecessor, each element is red or blue. I should say blue, whatever, anyway, no Team Fortress fans, I guess. So and now a query just reports the color of the predecessor.

You don't need to report the key value, just whether it is red or blue. So, again, you've got your universe. Some of the items are present.

And your query is an arbitrary universe element. And you want a predecessor kind of this way. And this guy is either red or blue. And you just want to report which one is it?

So this is, of course, easier, because you could just have a lookup table, a hash table for example, that once you determine the key, you could figure out whether it's red or blue. It's a static data structure. So you could use perfect hashing. It could be totally deterministic once you set it up.

And so, OK, just reporting the colors, of course easier than reporting the key. One interesting thing about this problem is it's very easy to get a correct answer with probability 50%. You flip a coin.

Heads is red, tails is blue. And this lower bound we proved will actually apply to randomized data structures that work with some probability. But it has to be probability greater than half of succeeding.

And this will be useful, because we're going to take our problem. And we're going to modify it a little bit, and sort of make it simpler and simpler. And it's easier to preserve this color property than to preserve actual key values. So that's the colored predecessor.

Now, we get to communication complexity perspective. So this is pretty cool. And it brings us to the idea of round elimination.

So communication-- OK, why don't I tell you the generic communication complexity picture out of Alice and Bob? And then I'll tell you how it relates to data structures. They're both simple. But I've got to do something first.

So let's say Alice is one person who knows some value x. Over here, we have Bob who knows some value y. The goal is to-- I'll move this a little bit to the left.

Their collective goal-- they're trying to cooperate. Their goal is to compute some function of x and y. The trouble is only Alice knows x, and only Bob knows y.

So how did they do it? They have to talk to each other. Ideally, Alice sends Bob x, or Bob sends Alice y. One of them could compute it, send back the answer.

That's one possibility. But maybe x and y are kind of big, and you can't just send it in one message. So here's the restriction.

Alice can send messages with less than or equal to little a bits, a for Alice. Bob can send messages with less than or equal to little b bits, b for Bob. So that's the constraint.

And, potentially, x is much larger than a, and/or y is much longer than b. And so you're going to have to spend many messages. And let's restrict to protocols of the form Alice talks to Bob, then Bob talks to Alice, and back and forth, so rounds of communication.

And you want to know, how many rounds does it take to compute f of xy? And, of course, it

depends on f. OK, that's the totally generic picture. And there's techniques which we're going to use for lower bounds on how long these protocols have to be.

How does this relate to colored predecessor? And, remember, also there's the model of computation, which I should mention, cell probe model. Cell probe model, we just count the number of-- in this case, because it's a static data structure, we don't really change it.

Let's say memory word reads. We want to know how many words of the data structure do we need to read in order to answer a colored predecessor data structure? If we can prove a lower bound on this, of course it applies lower bound on the word RAM, or pick your favorite model that works with words, transdichotomous RAM, whatever.

OK, so I want to cast that cell probe picture in terms of this picture. So here's the idea. Maybe I'll switch colors.

Alice is the-- I guess what do you want to call it, the algorithm, the query algorithm. OK, Alice is the poor soul who has to compute the color of a predecessor. And so what's x? x is the query.

We're used to that. That is the input to the predecessor. So it's a single word, which is I want another predecessor of some value.

Bob, on the other hand, is the data structure. That's the static thing. Where Bob represents the data structure, y actually is the data structure.

I guess if I want to be a little bit more prosaic or something, Bob you can think of as memory. We'll call it RAM, to be a little more space saving. So Bob is the memory which you can access in random access. And what it knows is the data structure. I mean, that's what it's storing. That's all it's storing. That's y.

Now, what are these rounds? They are memory reads. So what's a?

a is a log of the size of the data structure, because that's how many bits you need in order to specify which word you want to read. So if you have, let's say, s words of space, a is going to be log s. OK, you could make it larger. But it doesn't help you.

So we're going to make it as small as possible, because that will let us prove things. It's fine to let a be log s, because Bob is not very intelligent. Bob, you just say, look, I would like word five please. And it says, here's word five.

So it's not doing any computation. Alice can do lots of computation and whatnot. In fact, free computation, we don't count that.

The question is just, how many things do you have to read from Bob? Now, in this picture, Bob could potentially compute stuff. But we know, in reality, it won't. Lower bounds aren't going to use that fact. But that's why we can set a to just be log s, because Bob wouldn't do anything with the extra information.

How big is b? b is just w, because the response to a word read is a word. So this is the picture. Query can probe the data structure.

It says, give me word something, which is only log s bits. The response is w bits. And you repeat this process over and over. And then, somehow, Alice has to compute f of xy.

In this model, Bob doesn't need to know the answer. Of course, it's just a single bit. What is f of xy? This is colored predecessor.

x is the query. y is the data structure. And f of xy, is it red or blue? Is the predecessor of x in this data structure in the set represented by this data structure red or blue?

So it's one bit of information. Alice could then tell the bit to Bob. But actually, in this model, we just want Alice to know the answer.

So if you can prove a lower bound on how many rounds of communication you need, then you prove a lower bound on the number of memory reads. Each round corresponds to one word read from memory. Clear? So a very simple idea, but a powerful one, as we will see, because it lets us talk about an idea which makes sense when you're thinking about protocols of rounds of communication, but does not really make sense from a data structure perspective-- well, I mean, not as much sense.

Round elimination is a concept that makes sense for any communication style protocol-- not just the red one, but the generic white picture. I need to define a little bit before I can get to the claim. This will seem weird and arbitrary for a little while, until I draw the red picture, which is what it corresponds to in the predecessor problem.

But just bear with me for a minute. Imagine this weird variation on whatever problem you're starting with. So we have some problem f, which happens to be colored predecessor. We're going to make a new version of that problem, called f to the k. And here is the setup.

It's going to be a little different from this, and kind of fits the same framework. But now, Alice has k inputs, x1, x2, up to xk. Bob has y, as before. And it has an integer, i, which is between 1 and k.

Also, this is a technicality because we'll need it for colored predecessor-- we won't see that for a few minutes. But Bob happens to know all the xi's up to xi minus 1. And the goal is to compute f of x i, y. Maybe I should draw a picture of this.

So we have Alice. Alice has x1 up to xk. Bob has y and i, same communication setup. And the goal is to compute f of xi, y. Before it was x, y. And now, we're saying, well, actually, x consists of these k parts. We really just care about the i-th part.

So this function does not depend on any other of the xj's, just xi. So naturally, Alice should just communicate to Bob about xi. Trouble is, Alice doesn't know what i is. Only Bob knows what i is. So if you think about a communication protocol, where initially Alice sends a message, then Bob responds, that first message that Alice sends is probably useless.

I mean, probably the first question is, what's i? That has no information. It's just every time in the beginning, you say, what's i? Then Bob says, here's i. And then after that one round, Alice can you just think about xi, from then on.

OK, one message may seem like nothing. But it's like every time you put a penny in the jar. After you do that enough times, you have a lot of money.

So one message may seem like very few. But we just need to prove a lower bound of log w over log log w messages. So you do this a few times. Eventually, you'll get rid of all the messages.

Now, if we can get rid of all the messages, it may seem crazy. But it turns out you can iterate this process of eliminating that first message. If we get rid of all the messages, the best we could hope for is an algorithm that is correct with 50% probability.

If Alice can do nothing, then the best Alice could do is flip a coin. So we will get a contradiction if we get a zero message protocol that wins with more than 50% probability. That's what we're going to try to do.

But what does it mean to eliminate this first message? Let me formalize the round elimination

a little bit over here. So here's the round elimination. This, again, works for any function f.

So if there's a protocol for this f to the k problem, and Alice speaks first, then that first message is going to be roughly useless. So let's suppose it has error probability delta. So there's some probability it gives the wrong answer.

And let's suppose that it uses m messages. Then there exists a protocol for f where Bob speaks first, the error probability goes up slightly. And it uses one fewer message.

OK, ultimately, we care about rounds, which are pairs of messages. But we're going to count messages. And then, of course, divide by two, you get rounds. So the idea is you can eliminate the first message that Alice sent.

The difference is, before you were solving this f to the k problem. If you start with Bob, then of course you know what i is. And so then your problem just reduces to computing this f on xi, y. So you don't get a protocol for f to the k anymore. But you get a protocol for f.

And we're going to iterate this process, and eventually eliminate all the messages. That's the plan. Let me give you some intuition for why this dilemma is true.

It's a little messy for us to prove. I'm not going to give a proof here. If there's time at the end, I'll give a little bit more of a proof. But it will still use some information theory that we will not prove, and some communication complexity which we won't prove, because it's a little bit involved to prove this.

But once you have this lemma, this lower bound is actually quite easy, and intuitive, and nice. So that's where I want to get to today. But let's start with some intuition of why this should be true, why there's this extra error term. Yeah, question?

AUDIENCE: Does it matter who reports the answer? Because there seems to be some symmetry there.

ERIK DEMAINE: Right, does it matter who has the answer, Alice or Bob? Let's just say you're done when anyone knows the answer, either Alice or Bob. Yeah, that would be cleaner.

Otherwise, you'd have to send a message at the end to tell it to Alice or something. So let's make it symmetric, by saying either Alice or Bob knows it. Then the protocol can end. Good question. That way, we won't pay for an additive one every time, only pay for it at the end. Good questions.

All right, so there's an issue here. It said, oh, Alice's message is probably useless. But maybe Alice gets lucky, and sends xi. Then, that message is useful.

What's the chance that Alice sends xi? Well, 1 out of k. So there's something going on there.

Alice doesn't just have to send an entire xj, though. Alice could send some mix of these guys. Maybe it sends the x or of all the xj's. Or it could be anything.

But there's a limit. There's only a bits being sent from Alice to Bob. So the idea is if the total number of bits here is much bigger than a, then very few of the bits that you send are going to be about a particular element xi, in expectation. So this is a probabilistic thing.

So just imagine this for the moment. Basically, because we're in lower bound world, we get to, essentially, set up the input however we want. And so, in particular, we're going to prove a lower bound that, in expectation-- so we're going to have a probability distribution of data structure, or not data structures, but of sets of values that are in the set.

And the claim is that, in expectation, any data structure must do at least log w over log log w, minned with log base w of n queries, in expectation. So we get to assume that the input is random. And we'll see why in a bit.

So in this world, we can assume that i is chosen uniformly at random. And given that assumption, you would expect exactly a over k bits to be about xi. That would be the best you could hope to do. Sort of, you have a bits.

You spread them out evenly talk about all the xi's that you can. So you get to communicate a little bit of information about the particular xi that Bob will care about. You don't know which one that is. So you have to communicate about all of them. So it's going to be a over k in expectation.

So here's the idea. We want to remove that message so Alice can't communicate those a over k bits. So what's Bob going to do? Bob is going to guess them by flipping coins so he gets them uniformly randomly, a over k bits.

What's the probability that Bob is right? Well, it's going to be 1 over 2 to the a over k. That seems not so big. But if k-- how does it go?

If k is much larger than a, then this is actually a good thing. So let me-- so this was the

probability of being correct. So the probability of being incorrect is going to be 1 minus that.

And we're interested in-- we had some probability delta failing before. And now there's this new way that we can fail. So I'm using union bound here, saying, well, we could fail the old way, or fail the new way. Maybe they're correlated, maybe not.

But the worst cases is that it wants the sum of the two errors. So the increase in error is, at most, 1 minus this thing. Now, this 1 minus 1 over something to the power or something, if this something is large, there's this fun fact 1 minus 1, over e to the x is approximately x.

So this is going to be approximately a over k, if k is large enough. So this is for large x-- small x, sorry. k is large. And so a over k is small, very close to 0.

So if this were true, then the error increase would be order a over k. There's e versus 2. So there's a constant factor I'm losing there.

It's not quite that good. So this is only intuition. The real bound has a square route here. And it's necessary. And I don't have a great intuition why it's square root.

It's just a little bit worse than this intuition. I mean, the main issue is, what does it mean for bits to be about something? And can you really just guess those bits?

Actually, you have to guess the message that Bob sent. So it's a little bit more than just the bits that-- sorry that Alice sent. So you lose a little bit more. But it won't make a huge difference to us, square root.

So that's some rough intuition for round elimination. Let's see why it is so cool to have round elimination, how it lets us prove a pretty strong lower bound on colored predecessor in the cell pro model. OK, I think I missed a claimed somewhere here.

This is the lower bound we're going to prove. And it's nice and symmetric, log base a of w, log base b of n. This is kind of perfectly symmetric in Alice and Bob.

Alice, well I don't know which one represents-- and, apparently, Alice represents w. Alice has got a bits to communicate with to Bob. I mean, the input, I guess, is of size w. That's the query.

Bob, on the other hand, knows all the data. So it's, in some sense, represents the n side. It's able to communicate with b bits of information. So log base b of n, somehow enough to communicate n.

It's not a great intuition for this. But at least it's nice and symmetric. Now, let's work out what it actually corresponds to for-- this is a lower bound on colored predecessor. So for any colored, even static, colored predecessor data structure, static can be randomized. And this will be a lower bound on the expected performance.

OK, so what this implies for polynomial space, which is kind of the case we care about, for polynomial space data structure, a is going to be order log n. In fact, we only need a to be poly log n for what I'm about to say to be true. So then this becomes min log-- what n? I guess.

Is that what I wrote over there? I don't even remember. I wrote a log log w. I guess I'm going to write log log n. This is the right answer.

OK, fine, so we get a-- b is just w. So that's a log base w of n. That's just fusion tries. This one, ideally, would be log w. But we're off by this log log factor if a is log n.

That's the best we'll be able to prove today. So this is slightly less beautiful, because it's both w and n. But so was this one.

And this is not the true answer. The true answer has no log log. But that's what we get from nice symmetric bound for polynomial space.

You can also use this to prove the Beame Fich lower bounds, which I've by now erased, which are the pure-- the for all n, there exists a w, for all w, there exists an n. Why don't I briefly cover that?

Again, let's assume that a is order log n with polynomial space. Then, the lower bound will be largest when the two terms in the min are equal. So log base a of w equals log base b of n.

And that's these two guys. So this is log w over a is order log n. So this is going to be order log log n.

And I want this to be equal to log n over log w. So we can cross multiply, get log square w equals log n, log log n. And so log w is equal to square root log n, log log n.

And let's see. You can also take logs, and from this conclude that log log w equals log log n. I'm throwing away constant factors.

And when I say equal here, I mean up to constant factors for throughout this half board. So

you take logs. You get 2 log log w on the left hand side. Over here, you get log log n, plus log log log n.

So it's dominated by log log n. So in this scenario, where these two things are roughly equal, you get log log w roughly equal to log log n. And so, from this bound, or from this equality, we should get the old Beame Fich bounds, which let me write them again.

So they are-- I mean, we want to compute what was this bound. We have log w, divided by log a, or log w divided by log log n-- so log w, divided by log log n. But log log n now is log log w. So this is going to be log w over log log w, which was one of the Beam Fich bounds.

So for all w, there exists an n, such that omega this thing holds. On the other hand, we have log n, divided by log w is now square root of log n, log log n. So this is log n, divided by square root of log n, log log n.

So the square root log n cancels with this. We end up with square root of log n on the top. And we keep the square root of log log n on the bottom.

And so we get the other Beam Fich Xiau bound, for all n, there exists a w, such that omega this holds. And when you just need existence, then we get to choose the relationship between n and w however we want.

So you don't have to believe me that it's largest when they're equal. It happens to be true. But the point is, there is a choice of w and n-- namely, this choice, log w equals root log n, log log n, where you get a lower bound of this. And there's another choice for n versus w where this happens.

So this implies the Beame Fich Xiau bounds. But I kind of prefer this form, because it's clear up to this log log n factor, we understand the complete trade-off between w and n, assuming polynomial space. So we're going to prove this bound, which implies all this stuff, using round elimination lemma.

So we're proving this claim here, omega min log base of a of w, log base b of n, for any colored predecessor data structure. So let's suppose you have a colored predecessor data structure. And it can answer queries in t cell probes. which in the communication complexity perspective, that's rounds of communication, or colored predecessor.

Our goal is to do t round eliminations. Slight discrepancy in terminology here-- round

elimination lemma is really about eliminating one message, which is half a round. But you do it twice, you eliminate a round.

And so we need to do two t calls to this lemma. We will eliminate all messages. Then there's zero communication.

Then the best you could hope to do is by flipping a coin. Maybe you're worse than that. But the probability of error has to be at least 1/2.

So we'll either get a contradiction. Or we're going to set things up so the error is, at most, 1/3. And, therefore, this will prove that t has to be large. So you couldn't eliminate all the messages. Anyway, we'll see that in a moment.

So we have a setup like this. And in our case, with this picture, there is an asymmetry between Alice and Bob. I mean, yeah, the picture is nice and clean.

But in reality, this has to represent a colored predecessor problem. So in reality, Bob is a data structure and is not very smart, just does random access. Alice, we don't know, could be very smart. And Alice just has a single word.

So there's an asymmetry between Alice and Bob. So when we eliminate a message from Alice to Bob, it's going to be different from eliminating a message from Bob to Alice. This f to the k thing here is going to have to be different when we're doing an Alice to Bob message, versus doing a Bob to Alice message.

And that's where we're going to get a min. When we go from Alice to Bob, we're going to be, essentially, contributing to this term. When we go Bob to Alice, we're going to be contributing to this term, I think, or the reverse. We'll find out in a second.

So let's do first Alice to Bob, because that's the first type of message we need to eliminate. So let's suppose-- now, as we eliminate things, w and n are going to decrease. So I'm going to suppose at this point, Alice's input has w prime bits left.

Initially, w prime is w. But I want to do the generic case so I don't have to repeat anything. Here's the concept.

Remember, we're proving a lower bound. We get to set up the set of elements however we want. We're going to define a distribution on the set of elements. And we're going to do that by

breaking this input, w prime bits, into a lot of pieces.

Alice's input, you think of as x. Don't necessarily want to call it x. Well, sure, we can call it x. There it Is

What I'd like to do is break that input into chunks, x1 up to xk. This is basically what round elimination tells me I should do. If I want to view my problem as an f to the k problem, somehow my input is not just one thing. It's k things.

Well, in reality, the input is a single word. So I'm going to have to split it into sub words, in the usual way. Van Emde Boas would split it in half. We're going to split it into more pieces, because we need to guarantee that this error is small.

We need k to be large for this error to be small. I claim this is the good choice of k. And so now, this is x1, this is x 2x, this is xk. The low order bits are xk. The high order bits aer x1. Or the reverse, it doesn't actually matter.

So why is this a good choice of k? Because if we then look at the error increase we get from here, error increase is square root of a divided by k. So error increase, which is order a over k, is going to be k is now this.

So the a's cancel-- sorry, square root of a over k. A over k was the wrong analysis. The square root of a over k is the correct bound. Still, the a's cancel, because the k has an a factor.

And so we get order square root of 1 over t squared, also known as 1 over t. This is good. There's a constant here. And if I tune this constant correct, I can make this constant less than 1/3.

So if I start with a protocol, let's say, that's completely correct-- you don't have to. You could start with one that's correct with at least probability 3/4 or something. But let's just say, for simplicity, the initial data structure is completely correct. If every time I do an elimination of a message, I guess I should set it to be 1/6 time's 1 over t, and I do this 2 times t times, in the end the error will be only 1/3. So I'll be correct with 2/3 probability.

So never mind the constants. I can set this constant right so this is some epsilon times 1 over t. So after I do this 2 t times, I end up with an error that's only epsilon, or 2 epsilon, or whatever. So that would be a contradiction.

So that's why this is a good choice. I'm basically balancing the error I get from every message elimination. I want them all to be order 1 over t so it's nice and balanced. That will give me the best lower bound, it turns out.

But what does this mean? I mean, somehow I have to have an f to the k problem, meaning, really I should only care about 1xi here, that all the interesting stuff is these bits. But Alice doesn't know which is i. Only Bob knows which is i.

What does that mean? Bob knows the set of elements in the data structure. So, basically, the set of elements-- maybe they all differ in these bits.

Or maybe they all differ in these bits, or these bits, or these bits, but only one of these. So that's a distribution of inputs. There's one class of inputs where they all differ in the x1 part. But then they're identical for the rest.

For each xi, there's a set of inputs where they all differ in the xi's, and nowhere else, let's say. But Alice doesn't know which setting she is in. Alice just knows x.

And so Alice is kind of in trouble, because you can only send a few bits. You can only send a bits out of this thing. So you can't communicate very much.

Let's go over here. I need the bound. I don't need the corollary. Let me draw a picture for Alice, or for what the data looks like.

So there's some initial segment, which is shared by all elements. Then they all differ in this middle chunk. And then they all have-- I don't really care what they have after that.

So these are the elements in the data structure. And this is our usual picture of a binary tree. Except, I didn't draw it binary. I drew it with some branching factor so that the height of this tree is theta at squared.

So I set the base of my representation so that you branch whatever, w divided by at squared here. That's my branching factor. So cool, that's my picture. And this is depth i.

So Bob knows this picture. Bob knows what all the elements are so it can build this picture or whatever. Bob knows which is the relevant depth that they differ. But in the lower bound, we're going to say a is chosen uniformly at random.

So Alice has no idea which bits to send. And so probably, Alice is going to say, oh, here, I

know these bits. But Bob already knew those bits. So Bob learned nothing. And so that's why you can eliminate the round. That's the intuition, anyway.

The proof is, well, you just apply round elimination. You see that the problem now becomes to compute predecessor on this node. Just like in fusion tries, at every step of the way, the hard part was to compute predecessor at the node.

Here, if Bob is allowed to do computation, then, really, Alice just needs to say, what are the things here? Or together, Alice and Bob somehow have to figure out, what is the predecessor at this node of xi? Which way the query goes, which could be in between one of these, that is xi.

The rest of the xj's don't matter. Only xi matters. And so the problem reduces to computing predecessor here. And that matches the definition of f to the k.

So we have successfully set up an f to the k problem. The only thing you need to do to solve the overall predecessor problem is to find your predecessor in the node, because there's only one element within each subtree that's enough to figure out your overall predecessor.

OK, depth i, at squared, I guess you can think of this as y. That's the part that Bob knows. These are, in some sense, xi up to xk minus 1.

And this is why we had to say over here that Bob already knew xi up to xi minus 1, because Bob already knows that the shared prefix among all the items-- sorry, not minus 1, i minus i. And so all the content is here in this node, which is y. And so it reduces to a regular predecessor problem, which is what f will always denote, colored predecessor of xi, y.

Cool. OK, what happens here in terms of n and w? We have a smaller problem here. We threw away all this stuff. What got reduced is our word size for here.

We started with something of size w prime. And now, we end up with something of this size, which was W prime divided by at squared up to theta. So this reduction reduces-- or this round elimination reduces w prime to w prime, divided by at squared, theta that. I think that's all I need to say at this point.

The claim is that this picture is kind of like Van Emde Boas in the following sense. Van Emde Boas is, essentially, binary searching on which level matters. If it goes too low, things are empty. If it goes too high, there's too many items.

So it's binary searching for that critical point, where you basically have a node and nothing else. And that's why it took log w. So here, one level matters.

And the goal is to figure out which one. And it's not exactly binary search here. We're losing a larger factor, at squared, at each step. So we're not reducing w to w over 2, like we did with Van Emde Boas. But this is why we're losing a little bit. We reduce by a factor of at squared.

So it's kind of like the Van Emde Boas upper bound. But here, we're setting up a lower bound picture so that-- this is still an arbitrary predecessor problem. But figuring out which level matters, that's tricky. And you really can't do much better than Van Emde Boas to style a binary search. OK, maybe you can do at squared wave binary search, instead of binary search.

So that was eliminating a message from Alice to Bob. Eliminating a message from Bob to Alice is going to look like fusion trees. That's the cool thing. So let's do that. Next page.

OK, let's suppose-- so this is going to get slightly confusing notationally, because you have to reverse a and b. The picture I set up over here, and for the round elimination lamma f to the k, and round elimination lamma, are all phrased in terms of eliminating the Alice to Bob message. You've got to invert everything.

So we're going to get square root of b over k error if I don't relabel things. So you can restate this, if Bob speaks first, and then Alice speaks next, I get square root of b over k error. The F to the k problem is now that Bob has k inputs.

Maybe call them y1 to yk. Alice has an input x and the integer i. So now Alice, the querier, knows what i is. But Bob doesn't. The data structure doesn't know it. So it just reversed everything.

So let me state it over here. Bob has input. Now, in general, what Bob knows is a bunch of integers, n integers. In general, n prime integers, because n is also going to increase.

And let's say each of them is w prime bits. That's what Bob knows. That's y.

So what do we have to do to phrase an f to the k problem? Well, just like this, we've got to break that input into bt squared equal-sized chunks. So let's just think about what that means.

So it's going to be y1 up to yk. k is theta bt squared. Again, we want bt squared, because then

you plug it into this root b over k formula. And you get that the error increase is order 1 over t, which is what we need to do t round eliminations.

So before, our input was a single word. Now, our input is a whole bunch of integers. So it's natural to split it up into different integers. Here's how I'm going to do that.

I'll just draw the picture over here. So this was the Alice to Bob elimination. Now, we'll do the Bob to Alice picture.

So what I'd like to do is split up all the inputs. What's the natural way to do that? Well, have a tree at the top. Let's just make it a binary tree.

So I'm drawing the usual picture I draw when I draw a bunch of elements. And then there's some elements left over in each of these subtrees. Each of these is a predecessor problem.

I'm constraining the inputs to sort of differ a lot in the beginning, in the early bits, the most significant bits. This is the most significant, next most significant, and so on, bits. I want the number of leaves here to be theta bt squared-- or not leaves, but number of intermediate nodes here. And, therefore, there are theta bt squared subtrees.

And I want to set up the n items to be uniformly distributed. So each of these has n divided by bt squared items. So we used a few bits up top here.

How many bits was this, just log of bt squared? So if you look at one of these subproblems, we have reduced w slightly, but only by this additive amount. It basically won't matter.

The big thing we changed is we used to have n items to search among, or this is n prime, I guess.

Now, we have and divided by bt squared items. So we reduced n substantially. And this is what fusion trees do, right? They look at the root first, and say, OK, look, I can distinguish w to the epsilon, different things in constant time.

Here, it's on w to the epsilon. But it's b to bt squared. And we haven't figured out what bt are yet. I mean, b is w. So that's close to a w factor, w folds search.

And then you determine, am I in this subtree, or this subtree? This is exactly what fusion trees do. But we're setting up a lower bound instance where that is the right thing to do, where you have to know what the early bits are before you can figure out which subtree you're in.

Now, reverse roles. Alice knows what these bits are. The data structure of relevance-- let's suppose you fit here, because you are 0111 in the beginning.

This is our yi problem. Alice knows what i is. Alice knows what those leading bits are. Bob doesn't.

And Bob is speaking first. So Bob could try to summarize this entire data structure. But there's basically no hope.

Bob needs to know, what are those early bits? So the first message from Bob is going to be useless. Alice can then send the bits. And then you can restrict your problem to a regular predecessor problem in here.

And so you've eliminated this sort of root node. Your entire problem is a predecessor problem in here. So, again, we have an f to the k style problem. We reduce to a predecessor problem on a smaller-- in this case, y and x are reversed. But we end up with a smaller problem, mainly in the fact that n got smaller.

So let me go back to this board. What happens is that we reduce n prime to n prime, divided by bt squared. There's theta.

We also reduce w prime. w prime becomes w prime, minus log bt squared. OK, but in particular, this is going to be at least w prime over 2 most of the time, till the very end. And over here, we're reducing w prime by a big factor, something bigger than a constant.

So we don't really care about losing a factor of 2, because we alternate between these. And so a factor of 2 just falls into the theta. So it's going to be at least that, as long as w prime is, I don't know, at least log-- log w would be enough.

Yeah, this is log w. t squared is going to be, at most, log w. So this is going to be log w, plus log log w. Log log w goes away. What is t?

The whole point is to prove a lower bound on t. But we know from an upper bound perspective, t is, at most, log w, because Van Emde Boas exists. So we're never going to prove a lower bound bigger than log w.

So whatever t is going to be is, at most, log w. So this is just log w. So you're fine.

But at some point, we have to stop. We start with some w and n. n prime is racing down at this rate. w prime is racing down at this rate.

When w prime gets down to log w, then we're in trouble. If n prime gets down to 2, 1, 0, one of those constants, when n prime gets down to a constant, we're in trouble. You can no longer evenly divide things.

So when does that happen? This is where we're going to get a lower bound. So I guess we can go back over here.

So if you do a round elimination, you decrease w prime by this factor. And you decrease n prime by this factor, both of these in sequence. Then you repeat it.

So we have to stop when w prime hits log w, or when n prime hits 2, let's say. This is not really going to make a difference. You can also think that as 2. So, in fact, I'll just do that, just to clean the arithmetic a little bit.

Now, if we succeed in eliminating all the messages, we get a contradiction, because we set up our errors to add up to only 1/3. And we know the error has to be at least 1/2, if you eliminate all the messages, which means we didn't eliminate all the messages.

Now, we will not eliminate all the messages if t is large. So when would we eliminate all the messages? Let's compute that.

And then t has to be at least that big. t has to be greater than the point at which we would eliminate all the messages. Otherwise, we would get a contradiction. So this is where we get a min, which is kind of cool, because we have to stop when w prime gets too small, or when n prime gets too small, whichever happens first.

And so it's going to be a min of the thing relating to w, which is going to be log base at squared of w, because we're reducing w by a factor of at squared in each step. So that's when it's going to bottom out. And then there's the thing for n, which is log base bt squared of n.

That's when n will bottom out, because we're reducing by a factor of bt square at each step. So that's the bound we get. Now, it doesn't look quite like the bound that we want, which is this very simple log, base of w, log base b of n. But I claim that it's basically the same thing.

Why? I got to cheat here, speed things up a little bit. I claim that at squared is order a cubed.

Why? Because a is at least log n. Remember, a is log of space.

You have to store the data. So you need at least n words of space. a is log of that. So it's got to be at least log n.

And, furthermore, t is at most log n. Why? Because there is a predecessor or data structure that runs in log n time operation, namely balance binary search trees. There's a theta here, order, big O. But at squared is, at most, a cubed.

On the other hand, this thing, bt square, is order b cubed. Why? Because t is order log w. b is w. That's the definition of b.

T is order log w, because Van Emde Boas exists. And so this is actually, at most, w times log squared w. But, in particular, it's at most w cubed, because we don't care about that constant because it's in a log. So it just comes out.

It's a factor of 1/3 or whatever. And so this is the same thing as log base a of w, log base b of n, minned. And that's the nice, symmetric bound that you get our of round elimination, pretty cool.

Now, if you look in the notes, I'll just mentioned this briefly, there's a couple of more pages that give you a little bit of flavor of what the round elimination and that vague proof sketch, which gave the wrong answer by a square root, really means. We said, oh, Alice can only communicate, in some sense, a over k bits about xi.

And you can formalize that by talking about distributions of inputs, and talking about the expected amount of entropy of xi communicated by that first message. And so it's very simple definitions of entropy, and shared information, but just in terms of probabilistic quantities. And it at least gives you a sense of how you might prove something like this.

Whereas talking about information about something is kind of vague, talking about entropy, which is about sum of probabilities times log of probabilities is a clean, probabilistic notion. So it becomes a purely probabilistic statements about error probabilities. And that's how you argue this.

But even these notes do not give a proof. They just give a hint of how you formalize what this means, or what the proof sketch means. And it's several pages to actually prove it, so a bit beyond this class.

But once you have it-- because it's very clean, I'd say kind of beautiful lower bound, not quite the right answer, but up to log log factors, the right answer for predecessor. So that's the end of predecessor in this class.