**LING REN:** All right. Let's get started. Today's topic would be distributed algorithms. We will look at two new algorithms. But they are similar to what you have seen in the lectures. So it will also be a review of some concepts in lectures.

So our first example would be, again, [INAUDIBLE], the simplest example. But this time, the network and topology would be a ring. So in the lectures, the example we see is a click, meaning they are fully connected. Right.

Every node can talk to every other node. There, if you remember, our solution for everyone to generate a UID or a random number. And if you are the maximum, then you output, you're the leader

You can do that because, yeah, you are connected to everybody. So you immediately know what random number everybody generates. And you can compare whether you are the largest. Now, the idea is the same if we have a ring.

So you want everyone to generate a ID or random number. I'll just say ID from now on. And you want to collect everyone else ID so that you know whether your ID is the largest among all of them. OK.

**AUDIENCE:** [INAUDIBLE]

**LING REN:** Pardon?

**AUDIENCE:** [INAUDIBLE]

**LING REN:** OK. The question is where did the comparison happen? So we first need some way to pass the numbers around such that everyone has everyone else's number. Right. You have the number of everyone, then you can compare whether the largest equals yours. OK.

If the largest equals yours, then you know you are the largest. And you're going to output, I'm

the leader. OK. So the difficulty is just how to pass the numbers around so that everyone sees everyone else's number. Any ideas? Simple solution.

Well, there's not much you can do. You are just connect to your two neighbors, the left one and the right one. So how do you propagate the information? Go ahead.

**AUDIENCE:** You can take the maximum of your ID and neighbor's IDs and kind of broadcast that.

**LING REN:** OK. So what does broadcast mean here?

**AUDIENCE:** Like tell your neighbors what the largest of your value [INAUDIBLE]

**LING REN:** OK. And yes. Let me call them A, B, C, E, F. Then, yes, C can tell D or can tell B, but how does C tell E?

**AUDIENCE:** It waited around to I guess propagate the maximum.

**LING REN:** Yep. So it would be just, say, everyone talks to its right neighbor. And then B has A's IDs, C these B's ID, D has C's ID, and then continue and pass around next time. So just to make it perfect clear.

Say they generate some random IDs that's 5, 10, 20. And then in the next round, A would send its ID to B, B would send its ID to C, C would send its ID to D. And in the next round, B would pass this information along to C. And C would pass the information to D. And just continue.

And eventually everyone will have everyone else's ID. So how many rounds do we need? If they are in nodes in the system.

**AUDIENCE:** I think probably in just one direction or two neighbors.

**LING REN:** I think you can do it either way. If you propagate both ways, it probably 2x faster, yeah.

**AUDIENCE:** [INAUDIBLE]

**LING REN:** Yeah, correct. It's just O(n). But to keep it simple, let's say we just propagate in one direction. That's also fine. Still O(n). So how many messages are sent in total?

**AUDIENCE:** n squared.

**LING REN:** Pardon?

**AUDIENCE:** n squared.

**LING REN:** n squared. Yep. Is that obvious to everyone? OK. Give an explanation.

**AUDIENCE:** Yeah, at each round everyone sends a message to its neighbors.

**LING REN:** Yeah. Yeah. Every round, everyone sends a message. Or you can think of every message as propagated n times. And there are n messages in total. OK.

Yeah. That's definitely a solution. Well, you can imagine, it's probably the naive solution. And can we do better than that?

**AUDIENCE:** In this case, are we assuming they know that there are n of them?

**LING REN:** OK. Good question. So if they know there are n of them, yep, then they know how to terminate. So your question is how to terminate, right? If they don't, eventually you will receive your own ID on the other side.

If we keep sending left, eventually you will receive it on the left port. And, yeah, that's an indication of termination. Question?

**AUDIENCE:** Are we assuming the trajectory that these are unique?

**LING REN:** Yeah. We usually assume that, yeah. Either it's a unique user-- sorry, UID. What does U stand for by the way? Or if you generate random numbers in a large range, it's very unlikely that they collide.

OK. So can we do better than that? You have some idea?

**AUDIENCE:** Once something [INAUDIBLE] where you only pass it to the one that's bigger.

**LING REN:** Hm-mm. OK. I think you have two ideas. And binary search, we'll see how that works later. So you said only forward something that's larger? 0

**AUDIENCE:** Yeah, like only forward things are larger.

**LING REN:** OK, yeah, that's on the right track. So one obvious thing we can do is that so we actually don't care what all the IDs are, right. We only care whether a certain ID's the largest. So for example, in this case, when A send its ID 5 to B, B knows this 5 won't be the leader.

All right. Because 5 is too small. It's even smaller than his ID. So we can choose to drop this message. There's no point in passing that message further.

Same thing for this message. C knows that 10 is too small. And it doesn't have to pass it along.

**AUDIENCE:** [INAUDIBLE]

**LING REN:** Oh, the ID? It's just the integer each node chooses at random. See. Yeah, the only purpose of the ID is to break symmetry. So, yeah. Like we have seen the lecture, if they don't do this, if they don't have any unique identifier, they won't be able to select the leader.

And when they have a unique number, then they can select, yeah, the largest one or smallest one in some way. OK. Does this optimization make sense? We can cut [INAUDIBLE] messages that have no chance of becoming the leader.

**AUDIENCE:** What is the upper bound of it? Is it still n squared?

**LING REN:** Correct. Yeah. That's a very good question and very good answer, by the way. But how effective is this? Well in average case, we may be able to drop some messages. But there pathological case where it actually doesn't help at all.

Say this is the ID we choose. Then when there's 20, it send around. B cannot drop it. Right, because it may be the largest. And when this 10 is sent to C, C cannot drop it. And when 20 comes along, it also cannot drop it.

So no node can drop any message. Its worst case is still n square. OK.

So let's think about the binary search equivalent, yeah, or how to binary search in this case.

**AUDIENCE:** So binary is [INAUDIBLE] what?

**LING REN:** Yeah, good question. It's not very obvious binary searching to what. I Yeah, just some binary idea that will give us a n log end bound. So actually this is the better logarithm. But once you have a log n, you can probably get-- OK, go ahead.

**AUDIENCE:** How about kind of merging, like instead of finding the [INAUDIBLE], finding like log maximum, then [INAUDIBLE].

**LING REN:** Yeah. That's on the right track, yeah.

**AUDIENCE:** Like clusters of some [INAUDIBLE].

**LING REN:** Yeah. That's definitely the right idea. Let's detail it a little bit. How do you carry it out? Hm-mm.

**AUDIENCE:** Divide the two parts. Like kind of the [INAUDIBLE].

**LING REN:** And, OK, so then it's let them select their leader respectively and compare which one is larger. It's interesting thought.

**AUDIENCE:** First, for example, A, B finds the maximum between these two, the C, D finds the maximum E [INAUDIBLE] maximum. Kind of merging, considering them as one node.

**LING REN:** I think the first difficulty I see is that if you cut it by half, it's no longer a ring, right.

**AUDIENCE:** Also they can't cut themselves in half.

**LING REN:** Yeah. Yeah. Correct. Yeah, yeah.

Yeah, it's definitely not an easy problem, not a easy algorithm. And the idea is to, well, you had the right idea. That we want to-- what's the word? Let the weak candidates shut up early.

So what I mean by that is say we have several round, so this B and A and C. We will that B only propagate to A and C first. If B is the local maximum among A, B, C, then B will try to talk further. Increase its range.

If B is not the local maximum of A, B, C, then, yeah, it can be quiet. Doesn't need to send messages anymore. If B succeeds in the next round, then you go further increase its range to try to talk to more people. OK.

So how does it work in detail? Well so in round I, we will let a node send this message up to 2 raised to I hops. In this case, is 1 and in next round is 2 and then 4.

If at any point some mode like along this range decides that you are not the local maximum, then they will reply that, yeah, you no longer need to send messages anymore. If this message successfully reaches this endpoint that 2 raised to I hops, then this guy will respond.

And this guy, if it still thinks you are the local maximum, then it will respond the message while

saying, yeah, continue. OK. And if the sender receives the continue message on both sides, then they it continue into the next round. Otherwise, it will go inactive. Is the algorithm clear?

**AUDIENCE:** If the problem reaches the next maximum, like next the node each has to send a message.

**LING REN:** Say that again.

**AUDIENCE:** Like after receiving like don't sending it, like do you have to choose one node, which has to send the message?

**LING REN:** Oh, OK. In the first round, everyone send their messages. OK. Then some of them will go inactive because they learn they are not maximum. And the remaining, the surviving ones, will, yeah, continue sending messages.

And then, yeah, half of them probably will die in the next round. And the surviving ones keep sending messages. Make sense?

**AUDIENCE:** How do you return messages through I? Like if you [INAUDIBLE] its neighbor and not the node that do either.

**LING REN:** Yeah. So we will send the message of this form, say, well, some message. And then we will send the hop and the direction, either left or right. And this hop will initially be set to 2 raised to I, the number of round. Then when this guy receives the message, it will increment the hop count and pass it along.

And every node when forwarding the message will decrease the that h by 1. And finally when it reaches here, that number becomes 0. And when a node sees a message with 0 hop count, it's going to reverse it, send it in the other direction.

And, again, set it back to 2 raised to I. This message I should say ID. And at certain point, a certain node may decide this ID's too small. It doesn't have a chance. Then the I can directly send it in the opposite direction, replying a message saying, yeah, you are too small.

OK. So any more questions on the algorithm itself? If not, what's the next step?

**AUDIENCE:** Time.

**LING REN:** Yeah, time complexity. I already claimed it's n log n. Is it? This is round. This is message. This is also message complexity. OK. So why is it n log n?

**AUDIENCE:** The log n [INAUDIBLE].

**LING REN:** Hm-mm. So we have a certain number of rounds. So how many rounds do we have?

**AUDIENCE:** Two.

**LING REN:** Yeah, maybe. But, yeah, I'll just say log n. OK. Because you are increasing your hop length. And we're going to compute like how many nodes are still active each around and how many messages are sent in the round.

So, well, the number of nodes active will just be this number if we start from 0. Why? Because the first time everyone is active. In the next time, only 1/3 of them will be active. But we said we are conservative here at we put 1/2. Right.

Next round is actually 1/5. If it's local maximum, it means like these two and those two will go inactive. But we put this as a upper bound.

So this is the number of nodes that are still active. And they will send the message up to this many hops. OK. And there are two directions.

And you send a message back. And then, yeah-- sorry, send the message forward and someone will reply. But in the end, this is n log n. Yeah, and I think I got 8n log n.

So my recitation note says it's 4 log n, not entirely sure what's going on. Yeah. But you can double check whether this is correct or the recitation note is correct.

**AUDIENCE:** What is I again?

**LING REN:** I's the number of round in the I-th round. Yeah. This many nodes are still active. And each of them will send a message of this many hops.

And I didn't mention whether the network is synchronous or asynchronous. And it turns out it doesn't care. Some of them can work for both synchronous and asynchronous. Apparently, it works for synchronous networks.

If it's asynchronous, then what changes is that different nodes are in different rounds. A certain node may be far ahead than the others. But it's fine. Eventually, they will converge to the correct result. OK.

So let's look at the second problem. Well, problem's defined even simpler. We just want to

count how many nodes are out there. We want the algorithm to work both synchronously and asynchronously.

By that I just mean we have a network. Well, say you have a lot of nodes after that. Just want to count how many nodes are there in this network.

So I'll give you, say, one minutes to let's first come up with a high-level plan. Is the problem clear?

**AUDIENCE:** In the worst case, [INAUDIBLE]. The worst case.

**LING REN:** OK. I haven't defined that. Let's not worry about complexity now for now. The complexity will depend on number of nodes and the number of edges, E. Let's just get it functionally correct.

OK. Anyone share a high-level strategy? Go ahead.

**AUDIENCE:** So each node will start like IDs of the other devices [INAUDIBLE]. So basically like it's going to send propagates [INAUDIBLE] sets of each node.

**LING REN:** On what edge?

**AUDIENCE:** Furthest one.

**LING REN:** Yeah. It wasn't on both edges?

**AUDIENCE:** Itself first.

**LING REN:** OK. So then maybe send the 1 here and send the 1 here there. Then this node will think it has a children, it as a child, right, which is this one. Let's just say this is the entire network. And what message does it send to this guy?

**AUDIENCE:** Reinforced it.

**LING REN:** Well, you probably need send two here because you have one and this is possibly its child, right. But then we're double counting this node. See the problem?

**AUDIENCE:** Do nodes have their IDs, like unique IDs?

**LING REN:** Oh, yeah. They have their IDs.

**AUDIENCE:** We can't send the IDs instead of the IDs through the end node.

**LING REN:** OK. We are going to send all the IDs.

**AUDIENCE:** Yeah, neighbors. And after n steps, it's going to have the [INAUDIBLE].

**LING REN:** OK. Yeah, that's an interesting thought. And then you may still need the root. And that will have everyone's ID. And then see how many unique I's are there. OK.

Interesting. Does this algorithm work? Yeah, I don't see any problem. OK. But let me still repeat what our algorithm is because it's closer to what's in the lecture.

So we're going to find a spanning tree of this network. A spanning tree means, well, like I have to cut one of these edges. If I have a tree, then I can have every child report to its parent.

Like how many offsprings, including myself do I have. And this node will sum up all its children and report to its parent.

Does everyone get that? So first, we'll find a spanning tree. And second, we'll have child reports to parent.

**AUDIENCE:** How can we find the spanning tree?

**LING REN:** Good question. So in the lecture, we have seen algorithm that find BFS spanning tree for synchronous networks. OK. This is review of the lecture. How does it work?

Each node will, say, we need to first choose a root. And our root will just send a message to its neighbor and, yeah, saying, you are my child, you are my child, you are my child. And then every node upon receiving this message from the parent will search among its neighbors. All right.

So the neighbors that haven't got a parent will acknowledge this sender as the parent. OK. That's a little messy. So this node will search to the leaf node. But then it will also try to search for this guy.

But this guy already had a parent, then, yeah, it will say I already got a parent and blah, blah, blah. This will give us a BFS spanning tree. What does it mean?

It's a spanning tree found by BFS. Does it work for asynchronous network? Go ahead.

**AUDIENCE:** This version doesn't. But there's a different version that contains an edge for relaxation

technique.

**LING REN:** Yeah. Correct. And so why does this version not work for a synchronous network?

**AUDIENCE:** Because different nodes can be on a different round number so that a longer path could end up going to the node even.

**LING REN:** Yeah, exactly. So let me give a concrete example. What does asynchronous network mean? Is that, well, messages travel at different speed. Say this link for some reason is temporarily down.

And this node doesn't receive the message from the root. And then this message travels very fast. And this message also travels very fast. So this message may reach this node earlier.

And then this node will think of it as a child of this node. OK. And then this message comes along, finally. But this node says, yeah, I already have a parent. And I'm going to reject you.

**AUDIENCE:** So it still makes a spanning tree?

**LING REN:** Yes. Good point. This is not a BFS spanning tree but it's still a spanning tree. All right. So in our problem, we're totally fine with it. Yeah.

But you do have to know if you really want a BFS spanning tree in an asynchronous network, then you have to like record the distance and do the relaxation and so on and so forth. OK.

So we will just use this algorithm, the BFS spanning tree algorithm, just run it asynchronously. It doesn't find the BFS spanning tree but it finds some spanning tree. OK. How does this algorithm work?

We will have several variables. The first one is parent. We will initialize to this undefined single. OK.

Then every node will pass around this search message. I'll use a slightly more shorthand notation than from the lecture. OK. Let's say the code I wrote is for a process u. OK.

If we receive a message, a search message, from v to u, that means u-- sorry, v is trying to become a parent. OK. And if I do not have a parent yet, I should set the parent to v. OK.

So what's the next step? Now, we got this search message from our parent. And we have to pass it along.

**AUDIENCE:** Should we send it to the like add your child [INAUDIBLE].

**LING REN:** Oh, yeah. Great. Yeah, we first need to respond by saying, OK, I'll use a shorter notation. Send these a cue that that's the message that will be sent to v at some point.

The message we send is parent 1. Parent 2. This is a response to v saying, yeah, you are my parent. OK.

Then else, we also have to respond by saying parent 0. You are not my parent because I already got some other parent. OK. Missing a step here.

If we receive a search message. Go ahead.

**AUDIENCE:** We send messages to all the other nodes.

**LING REN:** Yeah. We need to pass it to all my potential children. So I use this comma u, meaning the neighbors of u. And then I'll send them a message.

What message? Search. OK. OK.

So, well, naturally since we have a search message and we know how to deal with it, now we are sending this parent message with better deal with it. Right. So then the next chunk of code should be if we receive this parent message, I'll say parent b, meaning this message comes with either true or false.

I received this message from some node w. OK. I'm still u here. Because I just send all the message to all the w's, to all my neighbors, and they should give me a response. OK.

If b is 1 that means this particular w take me as its parent. Make sense? OK. So I'd better have a list of my children. I want to keep track of that. How?

We will create a new variable for children. It's going to be initialized to what? Yeah, empty set.

And now if this b is 1, then I'm going to put w in this children list. OK. I'm leaving some space here because our root should be slightly different.

So every other node will send search messages when it receive a search message. We need someone to initiate. Make sense? So if u equals root, we say the root is v0.

So two things should happen. First, we should set its parent to some special value. Just say root. And then I should copy this blob of code to here. OK.

It's a little crowded but I hope it's still legible. OK. This is already almost the correct algorithm, except that we don't know how to terminate. If we wait long enough, then everyone will receive all the responses and everyone will know its parent and this child list.

But how do we terminate?

**AUDIENCE:** After n rounds.

**LING REN:** Yeah, that's one way to do, after n rounds. But the whole point is we are trying to find what n is. We don't know how many nodes are there.

**AUDIENCE:** If we receive reject for all, [INAUDIBLE].

**LING REN:** Say again.

**AUDIENCE:** For all the search, if we receive a reject.

**LING REN:** Oh, yeah. If you receive a response, either parent 0 or parent 1 from all your neighbors, then your job is pretty much done. But the others have not. All right.

So you send a message very fast. They responded to you. And they are still working very hard. OK.

So then we need to use the technique from the lecture that's called converge cast. So everyone will send a done signal when it is done and all its children are also done. OK.

To do that, I'm going to define a new variable called searched. Searched means someone has responded. OK. In this case, w has responded. I'll put it into the search the list, no matter whether it accepts me as parent or reject. OK. Then naturally I need to define this variable here. OK.

As a last step, If we search the list equals my neighbor list, that means everyone has responded and all my children are done, then I need a new variable called done. That's another list tracking who has finished and who has not. OK.

**AUDIENCE:** So what's the first one?

**LING REN:** Say again.

**AUDIENCE:** What was the first one?

**LING REN:** This one? Searched means someone has responded to the search message. OK. Done means all its children are done. I haven't write how done is defined. Give me a minute.

In this case, I'm going to send my parent a message. This is the step of converge cast. What do I send? I'm going to send them I'm done. OK.

Then whenever we create a message, we should deal with that message. If we receive a message I'm done, what do we do? OK. From w, then we're going to mark that node as done. OK.

There's this other point. So someone has to initiate the done signal. That's going to be our leaves, right. Because when this condition check, they don't have any children. Their children list is empty set.

And their done list is also empty set. But they're going to send the I'm done signal first. And then in the median nodes, we'll send a done signal when all its children are done. All right.

So this is the converged cast version. Only gives us a termination point of our spanning tree search. We haven't count the number of nodes in the network. But that's a small modification.

We're just going to include that number in the I'm done signal. So then I need to define another variable called total that's initialized to 0. This variable will attract how many nodes do I have in my subtree, including me and all my children.

Then when I send the I'm done signal, I'm going to send this-- sorry. That's not right way. When I'm sending the I'm done signal, I'm going to send my total number of offspring with it. And when I receive, one of my children reports that I do have t children.

I need to increment my total by that amount. OK. I made a mistake again. Should be total plus 1. Right.

Because I'm counting all my children and then I should include myself. So that's the complete algorithm. Yeah.

One purpose is just to create a different angle to look at distributed algorithm. Usually just

draw that network graphs. But sometimes it's helpful to think about how the code actually works. OK. That's all for today.