ERIK DEMAINE: All right, let's get started. Today, we have another cool graph algorithm or problem. Actually, we'll have two algorithms. The problem is called minimum spanning tree. You can probably guess from the title what it's trying to do.

We'll see two algorithms for doing it. Both of them are in the category of greedy algorithms, which is something we've seen a couple of times already in 6.046, starting with lecture 1. This is the definition of greedy algorithm from lecture 1, roughly. The idea is to always make greedy choices, meaning the choice is locally best. For right now, it seems like a good thing to do, but maybe in the future it will screw you over. And if you have a correct greedy algorithm, you prove that it won't screw you over.

So it's sort of like Cookie Monster here, always locally seems like a good idea to eat another cookie, but maybe it'll bite you in the future. So today we will embrace our inner Cookie Monster and eat as many-- eat the largest cookie first, would be the standard algorithm for Cookie Monster. I don't know if you learned that in Sesame Street, but-- all right.

So what's the problem? Minimum spanning tree. Can anyone tell me what a tree is? Formally, not the outside thing. In graph land. Acyclic graph, close. Connected acyclic graph, good. That's important. This is 604.2 stuff. OK, so how about a spanning tree? Sorry?

AUDIENCE: It contains all the vertices.

ERIK DEMAINE: It contains all the vertices. Yeah. So let me go over here.

Spanning means it contains all the vertices, so implicit here, I guess, is subtree or subgraph. You're given a graph. You want a spanning tree of that graph. It's going to be a tree that lives inside the graph. So we're going to take some of the edges of G, make a tree out of them, make a connected acyclic graph. And that tree should hit all the vertices in G. So this is going to be a subset of the edges, or subgraph. Those edges should form a tree. And, I'll say, hit all vertices of G.

OK, if I just said they should form a tree, then I could say, well, I'll take no edges, and here's a tree with one vertex. That's not very interesting. You want a vertex-- you want, basically, the vertex set of the tree to be the same as the vertex set of the graph. That's the spanning property. But you still want it to be a tree, so you want it to be connected and you want it to be acyclic. Now if G is disconnected, this is impossible. And for that, you could define a spanning forest to be like a maximal thing like this, but we'll focus on the case here as G is connected. That's the interesting case. And so we can get a spanning tree.

All right? So what is this minimum spanning tree problem? Minimum spanning tree. We're given a weighted graph, just like last time, with shortest paths. We have an edge weight function W giving me a real number, say, for every edge. And we want to find a spanning tree of minimum total weight. So I'm going to define the weight of a tree T to be the sum over all edges in T, because I'm viewing a spanning tree as a set of edges, of the weight of that edge. OK, so pretty much what you would expect.

Minimum weight spanning tree. It's a relatively simple problem, but it's not so easy to find an algorithm. You need to prove a lot to make sure that you really find the right tree. I guess the really naive algorithm here would be to try all spanning trees, compute the weight of each spanning tree and return the minimum. That sounds reasonable. That's correct. But it's bad, because-- n to the fourth, that would be nice. It's larger than that. Maybe not so obvious, but it can be exponential.

Here's a graph where the number of spanning trees is exponential. This is a complete bipartite graph with two vertices on one side and n vertices on the other, and so you can-- let's say we put these two edges into the spanning tree. And now, for each of these vertices, we can choose whether it connects to the left vertex or the right vertix. It can only do one, but it could do either one independently. So maybe this guy chooses the left one, this one chooses the right one. This one chooses the left one, and so on. If I have n vertices down here, I have 2 to the n different spanning trees. So there can be an exponential number. So that algorithm is not so good.

Exponential bad. Polynomial good. So today, we're going to get a polynomial algorithm. In fact, we will get an almost linear time algorithm as fast as Dijkstra's algorithm. But we can't use Dijkstra's algorithm, there's no shortest paths here. Plus, one of the algorithms will actually look pretty similar.

Two lectures ago, the dynamic programming lecture, we saw an example where we tried to do greedy, and it gave the wrong answer, and so we fell back on dynamic programming. Today, we're going to try to do dynamic programming, it's going to fail, and we're going to fall back on greedy. It's like the reverse. But the way it's going to fail is we're going to get exponential time initially, and then greedy will let us get polynomial time. This is actually a bit unusual. I would say more typically, dynamic programming can solve anything, but, you know, with n to the seventh running time, something slow. And then you apply greedy, and you get down to like n or n log n running time. So that's more common. But today, we're going to go from exponential down to polynomial. And that's pretty nice.

Cool. So let me tell you a little bit about greedy algorithm theory, so to speak. This is from the textbook. If your problem can be solved by greedy algorithm, usually you can prove two properties about that algorithm. One of them is called optimal substructure. And the other is called the greedy choice property. Optimal substructure should be familiar idea because it's essentially an encapsulation of dynamic programming. Greedy algorithms are, in some sense, a special form of dynamic programming.

So this is saying something like, if you can solve subproblems optimally, smaller subproblems, or whatever, then you can solve your original problem. And this may happen recursively, whatever. That's essentially what makes a recurrence work for dynamic programming. And with dynamic programming, for this to be possible, we need to guess some feature of the solution. For example, in minimum spanning tree, maybe you guess one of the edges that's in the right answer. And then, once you do that, you can reduce it to some other subproblems. And if you can solve those subproblems, you combine them and get an optimal solution to your original thing. So this is a familiar property. I don't usually think of it this way for dynamic programming, but that is essentially what we're doing via guessing.

But with greedy algorithms, we're not going to guess. We're just going to be greedy. Eat the largest cookie. And so that's the greedy choice property. This says that eating the largest cookie is actually a good thing to do. If we keep making locally optimal choices, will end up with a globally optimal solution. No tummy ache. This is something you wouldn't expect to be true in general, but it's going to be true for minimum spanning tree. And it's true for a handful of other problems. You'll see a bunch more in recitation tomorrow.

This is sort of general theory, but I'm actually going to have a theorem like this for minimum spanning tree and a theorem like this for minimum spanning tree. This is the prototype, but

most of today is all about minimum spanning tree. And for minimum spanning tree, neither of these is very obvious. So I'm just going to show you these theorems. They're fairly easy to prove, in fact, but finding them is probably the tricky part. Actually, I guess optimal substructure is probably the least intuitive or the least obvious greedy choice. You're probably already thinking, what are good greedy choices? Minimum weight edge seems like a good starting point, which we will get to. But there's even a stronger version of that, which we will prove.

And first, optimal substructure. So here, I'm going to think like a dynamic program. Let's suppose that we know an edge that's in our solution. Suppose we know an edge that lives in a minimum spanning tree. We could guess that. We're not going to, but we could. Either way, let's just suppose than an edge e--

I should mention, I guess I didn't say, this graph is undirected. A minimum spanning tree doesn't quite make sense with directed graphs. There are other versions of the problem but here, the graph is undirected. So probably, I should write this as a unordered set, u, v. And there are possibly many minimum spanning trees. There could be many solutions with the same weight. For example, if all of these edges have weight 1, all of these trees are actually minimum. If all the edges have weight 1, every spanning tree is minimum, because every spanning tree has exactly n minus 1 edges. But let's suppose we know an edge that's guaranteed to be in some minimum spanning tree, at least one.

What I would like to do is take this, so let me draw a picture. I have a graph. We've identified some edge in the graph, e, that lives in some minimum spanning tree. I'm going to draw some kind of tree structure here. OK. The wiggly lines are the tree. There are some other edges in here, which I don't want to draw too many of them because it's ugly. Those are other edges in the graph. Who knows where they are? They could be all sorts of things. OK? But I've highlighted the graph in a particular way.

Because the minimum spanning tree is a tree, if I delete e from the tree, then I get two components. Every edge I remove-- I'm minimally connected. So if I delete an edge, I disconnect into two parts, so I've drawn that as the left circle and the right circle. It's just a general way to think about a tree. Now there are other unused edges in this picture, who knows where they live? OK? What I would like to do is somehow simplify this graph and get a smaller problem, say a graph with fewer edges. Any suggestions on how to do that? I don't actually know where all these white edges are, but what I'd like to do is-- I'm supposing I know

where e is, and that's an edge in my minimum spanning tree. So how could I get rid of it? Yeah.

AUDIENCE: Find the minimum weight spanning tree of the two edges.

ERIK DEMAINE: I'd like to divide and conquer. Maybe find the minimum weight over here, minimum weight over here. Of course, I don't know which nodes are in what side. So that's a little trickier. But what do I do but E itself? Let's start with that. Yeah.

AUDIENCE: You remove it?

ERIK DEMAINE: You could remove it. That's a good idea. Doesn't work, but worth a Frisbee nonetheless. If I delete this edge, one problem is maybe none of these red edges exist and then my graph is disconnected. Well, maybe that's actually a good case. That probably would be a good case. Then I know how to divide and conquer. I just look at the connected components. In general, if I delete the edge, and I have these red edges, then I maybe find a minimum spanning tree on what remains. Maybe I'll end up including one of these edges. Maybe this edge ends up in the spanning tree, and then I can't put E in. So it's a little awkward. Yeah?

AUDIENCE: Can you merge the two nodes into one?

Merge the two nodes into one. Yes. Purple Frisbee. Impressive. This is what we call contracting the edge. It just means merge the endpoints. Merge u and v. So I will draw a new version of the graph. So this was u and v before. You've got to put the label inside. And now we have a new vertex here, which is uv. Or you can think it as the set u, v. We won't really need to keep track of names. And whatever edges you had over here, you're going to have over here. OK? Just collapse u and v. The edge e disappears.

And one other thing can happen. Let me-- go over here. We could end up with duplicate edges by this process. So for example, suppose we have u and v, and they have a common neighbor. Might have many common neighbors, who knows. Add some other edges, uncommon neighbors. When I merge, I'd like to just have a single edge to that vertex and a single edge to that vertex.

And what I'm going to do is, if I have some weights on these edges, let's say a and b, and c and d, I'm just going to take the minimum. Because what I'm about to do is compute a minimum spanning tree in this graph. And if I take the minimum spanning tree here, and I had

multiple edges-- one weight a, one weight b-- do you think I would choose the larger weight edge? It does-- they're exactly the same edge, but one is higher weight. There's no point in keeping the higher weight one, so I'm just going to throw away the higher weight one. Take them in.

So this is a particular form of edge contraction and graphs. And I claim it's a good thing to do, in the sense that if I can find a minimum spanning tree in this new graph-- this is usually called a G slash e, slash instead of negative, to remove e. I'm contracting e. So this is G slash e. This is G. If I can find a minimum spanning tree in G slash e, I claim I can find one in the original graph G just by adding the edge e. So I'm going to say if G prime is a minimum spanning tree, of G slash e, then T prime union e is a minimum spanning tree of G.

So overall, you can think of this as a recurrence in a dynamic program, and let me write down that dynamic program. It won't be very good dynamic program, but it's a starting point. This is conceptually what we want to do. We're trying to guess an edge e that's in a minimum spanning tree. Then we're going to contract that edge. Then we're going to recurse, find the minimum spanning tree on what remains, and then we find the minimum spanning tree. Then we want to decontract the edge, put it back, put the graph back the way it was. And then add e to the minimum spanning tree.

And what this lemma tells us, is that this is a correct algorithm. If you're lucky-- and we're going to force luckiness by trying all edges-- but if we start with an edge that is guaranteed to be in some minimum spanning tree, call it a safe edge, and we contract, and we find a minimum spanning tree on what remains, then we can put e back in at the end, and we'll get a minimum spanning tree of the original graph. So this gives us correctness of this algorithm.

Now, this algorithm's bad, again, from a complexity standpoint. The running time is going to be exponential. The number of sub problems we might have to consider here is all subsets of edges. There's no particular way-- because at every step, we're guessing an arbitrary edge in the graph, there's no structure. Like, we can't say well, it's the first k edges, or some substring of edges. It's just going to be some subset of edges. There's exponentially many subsets, 2 to the e, so this is exponential.

But we're going to make a polynomial by removing the guessing. This is actually a really good prototype for a greedy algorithm. If instead of guessing, trying all edges, if we could find a good edge to choose that's guaranteed to be in a minimum spanning tree, then we could

actually follow this procedure, and this would be like an iterative algorithm. If you-- you don't guess-- you correctly choose a good-- you take the biggest cookie, you contract it, and then you repeat that process over and over, that would be a prototype for a greedy algorithm and that's what's going to work. There's different ways to choose this greedy edge, and we're going to get two different algorithms accordingly. But that's where we're going.

First, I should prove this claim, cause, you know, where did edge contraction come from? Why does it work? It's not too hard to prove. Let's do it. Question? Oh. All right. I should be able to do this without looking. So--

Proof of optimal substructure. So we're given a lot. We're told that e belongs to a minimize spanning tree. Let's give that spanning tree a name. Say we have a minimum spanning tree T star, which contains e. So we're assuming that exists, then we contract e. And then we're given T prime, which is a minimum spanning tree of G slash e. And then we want to analyze this thing. So I want to claim that this thing is a minimum spanning tree, in other words, that the weight of that spanning tree is equal to the weight of this spanning tree, because this one is minimum. This is a minimum spanning of G. And this is also supposed to be a minimum spanning tree of G.

OK. Sounds easy, right? I'm going to cheat, sorry. I see. Right. Duh. Easy, once you know how.

So what we're going to do is think about contracting e. OK, we already know we're supposed to be thinking about contracting e in the graph. Let's look at how it changes that given minimum spanning tree. So we have T star, minimum spanning tree of the whole graph, and then I'm going to contract e. What I mean is, if that edge happens to be in the spanning tree-- it is, actually. We assumed that e is in there. So I'm basically removing, I'm just deleting that edge, maybe I should call it minus e. Then that should be a spanning tree of G slash e.

So when I contract the edge in the graph, if I throw away the edge from this spanning tree, I should still have a spanning tree, and I don't know whether it's minimum. Probably, it is, but we won't prove that right now. I claim it's still a spanning tree. What would that take? It still hits all the vertices, because if I removed the edge, things would not be connected together. But this edge was in the spanning tree, and then I fused those two vertices together, so whatever spanning-- I mean, whatever was connected before is still connected.

Contraction generally preserves connectivity. If these things were already connected directly

by an edge when I contract, I still have a connected structure, so I'm still hitting all the vertices. And also, the number of edges is still exactly right. Before, I had n minus 1 edges. Afterwards, I'll still have n minus 1 edges, because I removed one edge and I removed one vertex, in terms of the count. So that proves that it's still a spanning tree, using properties of trees.

Cool. So that means the minimum spanning tree, this thing, T prime, the minimum spanning tree of G slash e, has a smaller weight than this one. Because this is a spanning tree, the minimum is smaller than all spanning trees. So we know the weight of T prime is less than or equal to the weight of T star minus e. Cool. And now we want to know about this thing, the weight of T prime plus e. Well, that's just the weight of T prime plus the weight of e, because the weight of a tree is just the sum of the weights of the edges. So this is less than or equal to w of T star minus e plus e, which is just the weight of T star.

So we proved that the weight of our proposed spanning tree is less than or equal to the weight of the minimum spanning tree in G, and therefore, T prime union e actually is a minimum spanning tree. OK? This is really easy. It actually implies that all of these inequalities have to be equalities, because we started with something minimum. Clear? That's the easier half. The

More interesting property is going to be this greedy choice property. This is sort of where the action is for greedy algorithms, and this is usually the heart of proving greedy algorithms are correct. We don't yet have a greedy algorithm, but we're thinking about it. We need some way to intelligently choose an edge e, and I'm going to give you a whole bunch of ways to intelligently choose an edge e.

So here's a really powerful lemma, and we're going to make it even stronger in a moment. So I'm going to introduce the notion of a cut, that's going to be a similar picture to what I had before. I'm going to look at some set of vertices. S here is a subset of the vertices, and that leaves in the graph, everything else. This would be V minus S. OK, so there's some vertices over here, some vertices over here, there's some edges that are purely inside one side of the cut. And then what I'm interested in are the edges that cross the cut. OK, whatever they look like, these edges. If an edge has one vertex in V and one vertex not in V, I call that edge a crossing edge.

OK, so let's suppose that e is a least-weight edge crossing the cut. So let's say, let me be specific, if e is uv, then I want one of the endpoints, let's u, to be in S, and I want the other one to be not in S, so it's in capital V minus S. And that would be a crossing edge, and among all

the crossing edges, I want to take one of minimum weight. There might be many, but pick any one. Then I claim that edge is in a minimum spanning tree.

This is our golden ticket, right? If we can guarantee an edge is in the minimum spanning tree, then we plug that in here. Instead of guessing, we'll just take that edge-- we know it's in a minimum spanning tree-- and then we'll contract it and repeat this process. So the tricky part-- I mean, it is true that the minimum weight edge is in a minimum spanning tree, I'll give that away. But the question is, what you do then?

And I guess you contract and repeat but, that will be Kruskal's algorithm. But this is, in some sense, a more general tool that will let us identify edges that are guaranteed to be in the minimum spanning tree, even after we've already identified some edges as being in the minimum spanning tree, so it's a little more powerful. Let's prove this claim. This is where things get particularly cool. And this is where we're going to use something called a c and paste argument.

And if you are ever trying to prove a greedy algorithm correct, the first thing that should come to your mind is cut and paste. This is almost universally how you prove greedy algorithms to be correct, which is, suppose you have some optimal solution which doesn't have the property you want, like that it includes e here. And then you modify it, usually by cutting out one part of the solution and pasting in a different part, like e, and prove that you still have an optimal solution, and therefore, there is an optimal solution. There is an MST that has the property you want.

OK, so we're going to do that by starting from an arbitrary minimum spanning tree. So let T star be a minimum spanning tree of G, and if the edge e is in there, we're done. So presumably, e is not in that minimum spanning tree. We're going to modify T star to include e. So again, let me draw the cut. There's S and V minus S. We have some edge e which crosses the cut, goes from u to v, that's not in the minimum spanning tree. Let's say in blue, I draw the minimum spanning tree.

So you know, the minimum spanning tree connects everything together here. I claim it's got to have some edges that cross the cut, because if it has no edges that cross the cut, it doesn't connect vertices over here with vertices over here. So it may not use e, but some of the edges must cross the cut. So here's a possible minimum spanning tree. It happens to have sort of two components over here in S, maybe. Who knows? But there's got to be at least one edge

the crosses over.

In fact, the minimum spanning tree, T star, has to connect vertex u to vertex v, somehow. It doesn't use e, but there's got to be-- it's a tree, so in fact, there has to be a unique path from u to v in the minimum spanning tree. And now u is in S, v is not in S. So if you look at that path, for a while, you might stay in S, but eventually you have to leave S, which means there has to be an edge like this one, which I'll call it e prime, which transitions from S to V minus S.

So there must be an edge e prime in the minimum spanning tree that crosses the cut, because u and v are connected by a path and that path starts in S, ends not in S, so it's got to transition at least once. It might transition many times, but there has to be at least one such edge.

And now what I'm going to do is cut and paste. I'm going to remove e prime and add an e instead. So I'm going to look at T star minus e prime plus e. I claim that is a minimum spanning tree. First I want to claim, this is maybe the more annoying part, that it is a spanning tree. This is more of a graph theory thing. I guess one comforting thing is that you've preserved the number of edges, so it should still be if you get one property, you get the other, because I remove one edge, add in one edge, I'm still going to have n minus 1 edges.

The worry, I guess, is that things become disconnected when you do that, but that's essentially not going to happen because if I think of removing e prime, again, that disconnects the tree into two parts. And I know, by this path, that one part contains this vertex, another part contains this vertex, and I know that this vertex is connected to u and this vertex is connected to v. Maybe I should call this u prime and v prime.

I know u and u prime are connected by a path. I know v and v prime are connected by a path. But I know that by deleting e prime, u prime and v prime are not connected to each other. Therefore, u and v are not connected to each other, after removing e prime. So when I add in e, I newly connect u and v again, and so everything's connected back together. I have exactly the right number of edges. Therefore, I'm a spanning tree. So that's the graph three theory part.

Now the interesting part from a greedy algorithm is to prove to this is minimum, that the weight is not too big. So let's do that over here. So I have the weight of T star minus e plus-- minus e prime plus e. By linearity, this is just the weight of T star minus the weight e prime plus the weight of e.

And now we're going to use this property, we haven't that yet, e is a least-weight edge crossing the cut. So e prime crosses the cut, so does e, but e is the smallest possible weight you could have crossing the cut. That means that-- I'll put that over here-- the weight of e is less than or equal to the weight of e prime, because e prime is a particular edge crossing the cut, e was the smallest weight of them. So that tells us something about this. Signs are so difficult. I think that means that this is negative or zero.

So this should be less than or equal to w of T star, and that's what I want, because that says the weight of this spanning tree is less than or equal to the optimum weight, the minimum weight. So that means, actually, this must be minimum. So what I've done is I've constructed a new minimum spanning tree. It's just as good as T star, but now it includes my edge e, and that's what I wanted to prove. There is a minimum spanning tree that contains e, provided e is the minimum weight edge crossing a cut.

So that proves this greedy choice property. And I'm going to observe one extra feature of this proof, which is that-- so we cut and paste, in the sense that we removed one thing, which was e prime, and we added a different thing, e. And a useful feature is that the things that we change only are edges that cross the cut. So we only, let's say, modified edges that cross the cut. I'm going to use that later. We removed one edge that crossed the cut, and we put in the one that we wanted. OK so far?

There's a bunch of lemmas. Now we actually get to do algorithms using these lemmas. We'll start with maybe the less obvious algorithm, but it's nice because it's very much like Dijkstra. It follows very closely to the Dijkstra model. And then we'll get to the one that we've all been thinking about, which was choose a minimum weight edge, contract, and repeat. That doesn't-- well, that does work, but the obvious way is, maybe, slow. We want to do it in near linear time.

Let's start with the Dijkstra-like algorithm. This is Prim's algorithm. Maybe I'll start by writing down the algorithm. It's a little long. In general, the idea-- we want to apply this greedy choice property. To apply the greedy choice property, you need to choose a cut. With Prim, we're going to start out with an obvious cut, which is a single vertex. If we have a single vertex S, and we say that is our set capital S, then you know, there's some images coming out of it. There's basically S versus everyone else. That's a cut.

And so I could take the minimum weight edge coming out of that cut and put that in my

minimum spanning tree. So when I do that, I put it in my minimum spanning tree because I know it's in some minimum spanning tree. Now, I'm going to make capital S grow a little bit to include that vertex, and repeat. That's actually also a very natural algorithm. Start with a tiny s and just keep growing it one by one. At each stage use this lemma to guarantee the edge I'm adding is still in the minimum spanning tree.

So to make that work out, we're always going to need to choose the minimum weight edge that's coming out of the cut. And we'll do that using a priority queue, just like we do in Dijkstra. So for every vertex that's in V minus S, we're going to have that vertex in the priority queue. And the question is, what is the key value of that node stored in the priority queue?

So the invariant I'm going to have is that the key of v is the minimum of the weights of the edges that cross the cut into v. So for vertex v, I want to look at the-- I'm not going to compute this every time, I'm only going to maintain it. I want the minimum weight of an edge that starts in S and goes to v, which is not in S because v in Q-- Q only stores vertices that are not in S-- I want the key value to be that minimum weight so if I choose the overall minimum vertex, that gives me the edge of minimum weight that crosses the cut. OK?

I've sort of divided this minimum vertex by vertex. For every vertex over here, I'm going to say, what's the minimum incoming weight from somebody over here? What's the minimum incoming weight from someone over here to there? To here? Take the minimum of those things. And of course, the min of all those will be the min of all those edges. OK, that's how I'm dividing things up.

And this will be easier to maintain, but let me first initialize everything. OK, I guess we're going to actually initialize with S being the empty set, so Q will store everybody, except I'm going to get things started by setting for particular vertex little s. I'm going to set its key to zero. It doesn't matter who little s is. That's just your start vertex. Just pick one vertex and set its key to zero. That will force it to be chosen first because for everyone else, for v not equal to S, I'm going to set the key to infinity, because we haven't yet seen any edges that go in there, but we'll change that in a moment.

OK, so that was the initialization, now we're going to do a loop. We're going to keep going until the Q is empty, because when the Q is empty, that means S is everybody, and at that point, we'll have a spanning tree on the whole graph, and it better be minimum. OK, and we're going to do that by extracting the minimum from our priority Q. When we remove Q-- we remove

vertex u from the queue Q, this means that we're adding u to S. OK, by taking it out of Q, that means it enters S, by the invariant at the top.

So now we need to update this invariant, that all the key values are correct. As soon as we move a vertex into S, now there are new edges we have to consider from S to not S, and we do that just by looking at all of the neighbors of u. I haven't written this in a long time, but this is how it's usually written in Dijkstra, except in Dijkstra, these are the outgoing edges from u and v are the neighbors. Here, it's an undirected graph, so these are all of the neighbors v of u. This as an adjacency list.

OK, so we're looking at u, which has just been added to S, and we're looking at the edges. We want to look at the edge as they go to V minus S, only those ones. And then for those vertices v, we need to update their keys, because it used to just count all of these edges that went from the rest of S to v. And now we have a new edge uv that v needs to consider, because u just got added to S.

So the first thing I'm going say is if v in in Q. So we're just going to store a Boolean for every vertex about whether it's in the queue, and so when I extract it from the queue, I just set that Boolean to false. Being in the queue is the same as being not in S, this is what Q represents. So Q is over here, kind of. So if we're in the queue, same as saying v is not in S, then we're going to do a check which lets us compute the minimum. This is going to look a lot like a relaxation. Sorry.

A couple things going on because I want to compute not just the value of the minimum spanning tree, I actually want to find the minimum spanning tree, so I'm going to store parent pointers. But this is just basically taking a min. I say, if the weight of this edge is smaller than what's currently in the key, then update the key, because the key is supposed to be the min. OK, that's all we need to do to maintain this invariant, this for loop. After the for loop, this property will be restored, v dot key will be that minimum.

And furthermore, we kept track of where the minimums came from, so when you end up extracting a vertex, you've already figured out which edge you added to put that into the set. So in fact, u already had a parent, this would be u dot parent, and we want to add that edge into the minimum spanning tree when we add u to S. Overall, let me write why this is happening. At the end of the algorithm, for every vertex v, we want the v dot parent. And that will be our minimum spanning tree. Those are the edges that form the minimum spanning

tree.

Let's prove that this works. Actually, let's do an example. We've done enough proofs for a while. Let's do it over here. I need a little break. Examples are fun, though easy to make mistakes, so correct me if you see me making a mistake. And let me draw a graph. OK, weights. 14, 3, 8, 5, 6, 12, 7, 9, 15. 10. OK. Colors.

So I want to start at this vertex just because I know it does an interesting thing, or it's a nice example. Here's my weighted undirected graph. I want to compute minimum spanning tree. I'm going to start with a capital S being-- well actually, I start with capital S being nothing, and all of the weights-- all of the key values are initially infinity. So I'm going to write the key values in blue. So initially everything is infinity for every vertex, except for S the value is zero.

So all of these things are in my priority queue, and so when I extract from the queue, I of course get S. OK, that's the point of that set up. So that's when I draw the red circle containing little s. The red circle here is supposed to be capital S. So at this point, I've added capital S-- little s to capital S, and then I look at all of the neighbors v of S. And I make sure that they are outside of S. In this case, they all are. All three neighbors, these three guys, are not in S.

And then I look at the weights of the edges. Here I have a weight 7 edge. That's smaller than infinity, so I'm going to cross out infinity and write 7. And 15 is smaller than infinity, so I'm going to cross out infinity and write 15. And 10, surprise, is smaller than infinity. So I'm going to cross out infinity rate 10. So now I've updated the key values for those three nodes.

I should mention in the priority queue, to do that, that is a decrease-key operation. This thing here is a decrease-key. You need to update the priority queue to say, hey look, the key of this node changed. And so you're going to have to move it around in the heap, or whatever. Just like Dijkstra, same thing happens. OK, so I've decreased the key of those three nodes.

Now I do another iteration. I look at all of the key values stored. The smallest one is 7, because this node's no longer in there. So I'm going to add this node to capital S. So capital S is going to grow to include that node. I've extracted it from the queue. And now I look at all the neighbors of that node. So, for example, here's a neighbor. 9 is less than infinity, so I write 9. Here's a neighbor. 12 is less than infinity, so I write 12. 5 is less than infinity, so I write 5. Here's a neighbor, but s is in big S, so we're not going to touch that edge. I'm not going to touch s. OK? I will end up looking at every edge twice, so no big deal.

Right now, who's smallest? 5, I think. It's the smallest blue key. So we're going to add 5 to the set. Sorry, add this vertex to the set S, and then look at all of the outgoing edges from here. So 6 is actually less than 12, so this edge is better than that one was. Then, what's that, an 8? 8 Is less than 10. 14 is definitely less than infinity. And we look at this edge, but that edge stays inside the red set, so we forget about it.

Next smallest value is 6. So 6, we add this guy in. We look at the edges from that vertex, but actually nothing happens because all those vertices are inside capital S, so we don't care about those edges. Next one is 8, so we'll add in this vertex. And there's only one edge that leaves the cut, so that's 3, and 3 is indeed better than 14. So never mind. Stop.

So good, now I think the smallest key is 3. Notice smallest key is smaller than anything we've seen before, other than 0, but that's OK. I'll just add it in, and there's no edges leaving the cut from there. And then over here, we have 9 and 15. So first we'll add 9. There's no edges there. Then we add 15. OK, now s is everything. We're done. Q is empty.

Where's the minimal spanning tree? I forgot to draw it. Luckily, all of the edges here have different numbers as labels. So when I have a 3 here, what I mean is, include 3 in the minimum spanning tree, the edge that was labeled 3. OK, so this will be a minimum spanning tree edge. 5 will be a minimum spanning tree edge. These are actually the parent pointers. 6 will be a minimum spanning tree edge. 7, 9, 15, and 8.

Every vertex except the starting one will have a parent, which means we'll have exactly n minus 1 edges, that's a good sign. And in fact, this will be a minimum spanning tree. That's the claim, because every time we grew the circle to include a bigger thing, we were guaranteed that this edge was in the minimum spanning tree by applying this property with that cut. Let me just write that down.

OK, to prove correctness, you need to prove an invariant that this key, the key of every vertex, always remains this minimum. So this is an invariant. You should prove that by induction. I won't prove it here.

But we have another invariant, a more interesting one from an MST perspective, you know, it's just a sort of algorithm implementation detail, that the tree T sub S, within S is always contained in a minimum spanning tree of G. So over here, we have this way of computing minimum spanning tree for all vertices v, but what I'd like to do is just look at v that's currently in S. By the end, that will be the whole thing, but if I look at v in S, and I always look at the

edge from v to v dot parent, that gives me this tree TS. I claim it will be contained in a minimum spanning tree of the entire graph, proof by induction.

So by induction, let's assume-- induction hypothesis will be that, let's say there is a minimum spanning tree T star, which contains T sub S, and then what the algorithm does, is it repeatedly grows S by adding this vertex u to S. So let's suppose that it adds u to S. So I'm actually going to look at the edge that it adds. So we have S and V minus S, and we do this thing, like we just saw, of growing by one. We add one new vertex over here to S, and that vertex has a parent edge, has a parent pointer. So this edge, I'm going to call e. So we're adding some vertex u that we extract at the minimum, and we also added an edge e to this TS, because we grew S by 1.

OK, when I do that, all I do is say, look, greedy choice property guarantees there's a minimum spanning tree that contains e. Because we extracted the min from the queue, and the key values are this, as I was arguing before, that is the minimum overall edge that crosses the cut. e is a minimum weight edge that crosses the cut, and so by greedy choice property, there is some minimum spanning tree that contains e. But actually, I need that the minimum spanning tree not only contains e, but also contains all the other spanning tree edges that we had already said were in T star.

OK, so here's where I'm going to use the stronger property. I can modify T star to include e and T sub S. So we already assumed that T star includes T sub S. I just don't want to break that. And if you remember the proof of this greedy choice property, we said, well all we need to do is remove one edge that crosses the cut and replace it with e. So here what I'm saying is there's some edge, yeah, maybe there's some edge over here in T star that we had to remove, and then we put e in. And then we get a minimum spanning tree again, T star prime.

OK, this edge that I remove cannot be one of the TS edges because the TS edges are all inside S. So because I'm only removing an edge that crosses the cut, I'm not disturbing TS. TS will remain inside T star, but then I get the new property that e is inside T star, and so I prove this invariant holds. OK? I keep changing T star, but I always preserve the property that all of the spanning tree edges that are inside S are contained in some minimum spanning tree of G. Maybe I'll add in some for emphasis. Cool? So that's how we use the greedy choice property to get correctness of Prim's algorithm.

What's the running time of Prim's algorithm? Same as Dijkstra, good answer. I guess it

depends what priority queue you use, but whatever priority queue you use, it's the same as Dijkstra. And so in particular, if we use Fibonacci heaps, which, again, we're not covering, we get V log V plus E. In general, for every edge, we have to do a decrease-key. Actually, for every edge we do two decrease-key operations, potentially, if you think about it.

But this for loop over the adjacency, the cost of this stuff is constant. The cost of this is the degree of the vertex u. And so we're basically doing the sum of the degrees of the vertices, which is the number of edges times 2. That's the handshaking lemma. So for every edge, we're potentially doing one decrease-key operation, and with Fibonacci heaps, that's constant time. But we're also doing V extract mins those cost log V time, cause the size of the queue is at most V, and so that is actually the right running time. Just like Dijkstra, so easy formula to remember.

All right, let's do one more algorithm, Kruskal's algorithm. Kruskal's algorithm is a little bit weirder from the S perspective, I guess. We'll see what cuts we're using in a moment, but it's based around this idea of, well, the globally minimum weight edge is the minimum weight edge for all cuts that cross it, or for all cuts that it crosses. The globally minimum weight edge is going to be a valid choice, and so, by this theorem, you pick some S that partitions the endpoints of e, therefore e is in a minimum spanning tree. So let's choose that one first, and then repeat. Conceptually, what we want to do is that DP idea of contract the vertex, sorry, contract the edge and then find the minimum weight edge that remains. But the way I'm going to phrase it doesn't explicitly contract, although implicitly, it's doing that.

And there's a catch. The catch is suppose I've picked some edges out to be in my minimum spanning tree. Suppose this was the minimum weight and this was the next minimum, next minimum, next minimum, next minimum. Suppose that the next lar-- at this point, after contracting those edges, the minimum weight edge is this one. Do I want to put this edge in my minimum spanning tree? No. That would add a cycle. Cycles are bad. This is the tricky part of this algorithm. I have to keep track of whether I should actually add an edge, in other words, whether this vertex and this vertex have already been connected to each other. And it turns out you've already seen a data structure to do that. This is what I call union-find and the textbook calls it disjoint-set data structure.

So it's in recitation. Recitation 3. So I want to maintain for my MST so far, so I'm adding edges one at a time. And I have some tree-- well, it's actually a forest, but I'm still going to call it T, and I'm going to maintain it in a union-find structure, disjoint-set set data structure.

Remember, this had three operations, make set, union, and find set. Tell me given an item which set does it belong to? We're going to use that, the sets are going to be the connected components. So after I've added these edges, these guys, these vertices here, will form one connected component, and, you know, everybody else will just be in its own separate component.

So to get started, I'm not going to have any edges in my tree, and so every vertex is in its own connected component. So I represent that by calling make-set v for all vertices. So every vertex lives in its own singleton set. OK, now I'd like to do the minimum weight edge, and then the next minimum weight edge, and the next minimum weight edge. That's also known as sorting, so I'm going to sort E by weight, increasing weight, so I get to start with the minimum weight edge.

So now I'm going to do a for-loop over the edges, increasing order by weight. Now I want to know-- I have an edge, it's basically the minimum weight edge among the edges that remain, and so I want to know whether I should add it. I'm going to add it provided the endpoints of the edge are not in the same connected component. How can I find out whether two vertices are in the same connected component, given this setup? Yeah?

AUDIENCE: Call find-set twice and then--

ERIK DEMAINE: Call find-set twice and see whether they're equal, exactly. Good answer.

So if you find-set of u is from find-set of v, find-set just returns some identifier. We don't really care what it is, as long as it returns the same thing for the same set. So if u and v are in the same set, in other words, they're in the same connected component, then find-set will return the same thing for both. But provided they're not equal, then we can add this edge into our tree. So we add e to the set T, and then we have to represent the fact that we just merged the connected components of u and v, and we do that with a union call.

And if you're ever wondering what the heck do we use union-find for, this is the answer. The union-find data structure was invented in order to implement Kruskal's algorithm faster, OK? In fact, a lot of data structures come from graph algorithms. The reason Fibonacci heaps were invented was because there was Dijkstra's algorithm and we wanted it to run fast. So same deal here, you just saw it in the reverse order. First you saw union-find.

Now, union-find, you know you can solve v in alpha of n time, the inverse Ackermann function,

super, super tiny, slow growing function, smaller than log log log log log log log. Really small. But we have this sorting, which is kind of annoying. So the overall running time-- we'll worry about correctness in a moment. We have to sort-- to sort E by weight. So I'll just call that's sort of E. Then we have to do some unions. I guess for every edge, potentially, we do a union. I'll just write E times alpha of v. And then we have to do, well, we also have to find-sets, but same deal. So find-set and union cost alpha amortized, so the total cost for doing this for all edges is going to be the number of edges times alpha, and then there's like plus v, I guess, but that's smaller. That's a connected graph.

So other than the sorting time, this algorithm is really good. It's faster. But if you're sorting by an n log n algorithm, this is not so great. That's how it goes. I think you can reduce this to sorting just v things, instead of E things, with a little bit of effort, like doing a select operation. But when this algorithm is really good is if your weights are integers. If You have weights, let's say weight of e is 0 or 1 or, say, n to the c, for some constant c, then I can use rate x sort, linear time sorting, and then this will be linear time, and I'm only paying E times alpha. So if you have reasonably small weights, Kruskal's algorithm is better. Otherwise, I guess you prefer Prim's algorithm. But either away.

I actually used a variation of this algorithm recently. If you want to generate a random spanning tree, then you can use exactly the same algorithm. You pick a random manage that you haven't picked already, you see, can I add this edge with this test? If you can, add it and repeat. That will give you a random spanning tree. It will generate all spanning trees uniform leap likely. So that's a fun fact, useful thing for union-find.

Let me tell you briefly about correctness. Again, we proved correctness with an invariant. Claim that at all times the tree T of edges that we've picked so far is contained in some minimum spanning tree, T star. T start is going to change, but I always want the edges I've chosen to be inside a minimum spanning tree. Again, we can prove this by induction. So assume by induction that this is true so far, and then suppose that we're adding an edge here. So we're converting T into T prime, which is T union e.

By the data structural setup, I know that the endpoints of e, u, and v are in different connected components. In general, what my picture looks like, is I have some various connected components, maybe there's a single vertex, whatever. I've built a minimum spanning tree for each one. I built some tree, and I actually know that these trees are contained in one global minimum spanning tree. OK, and now we're looking at an edge that goes from some vertex u

in one connected component to some vertex v in a different connected component. This is our edge e. That's our setup.

Because the union-find data structure maintains connected components, that's another invariant to prove. We're considering adding this edge, which connects two different connected components. So I want to use the greedy choice property with some S. What should S be? I want e to cross a cut, so what's a good cut? Yeah?

AUDIENCE: The connected component of u and then everything else.

ERIK DEMAINE: Connected component of u and everything else?

AUDIENCE: Yeah.

ERIK DEMAINE: That would work, which is also the opposite of the connected component containing v. There are many choices that work. I could take basically this cut, which is the connected component of you with everything else versus the connected component of v. I could take this cut, which is the connected component of u only versus everybody else. Either of those will work. Good. Good curve, all right.

So let's say S equals the connected component of u, or connected component of v. e crosses that, all right? Because it goes from u to v, and u is on one side, v is on the other side. I wanted to include an entire connected component because when I apply the greedy choice property, I modify T star, and I don't want to modify, I don't want to delete any of these edges that are already in my connected components, that I've already put in there. But if I choose my cut to just be this, I know that the edge that I potentially remove will cross this cut, which means it goes between connected components, which means I haven't added that yet to T.

So when I apply this greedy choice property, I'm not deleting anything from T. Everything that was in T is still in T star. So that tells me that T prime is contained in T star prime. The new T star that I get when I apply the cut and paste argument, I modify T star potentially by removing one edge and putting e in. And the edge that I remove was not already in T, which means I preserve this part, but I also get that my new edge e is in the minimum spanning tree. And so that's how you prove by induction that at all times the edges that you've chosen so far are in T star.

Actually, to apply the greedy choice property, I need not only that e is cut-- sorry, that e crosses the cut, I also need that e is the minimum weight edge crossing the cut. That's a little

more argument to prove. The rough idea is that if you forget about the edges we've already dealt with, e is the globally minimum weight edge. OK, but what about the edges we've already dealt with? Some of them are in the tree. The edges that are in these-- that are in T, those obviously don't cross the cut. That's how we designed the cut. The cup was designed not to cross, not two separate any of these connected components. So all the edges that we've added to T, those are OK. They're not related to the edges that cross this cut.

But we may have already considered some lower weight edges that we didn't add to T. If we didn't add an edge to T, that means actually they were in the same set, which means also those are-- I'm going to use my other color, blue. Those are extra edges in here that are inside a connected component, have smaller weight than e, but they're inside the connected component. So again, they're not crossed. So they don't cross the cut, rather. So e is basically the first edge that we're considering that crosses this cut, because otherwise we would have added that other edge first.

So here, we have to do sort of the greedy argument again, considering edges by weight and e is going to be the first edge that crosses this particular cut, which is this connected component versus everyone else. So e has to be the minimum weight edge crossing the cut, so the greedy choice property applies. So we can put e in the minimum spanning tree, and this algorithm is correct.

OK? So we've used that lemma a zillion times by now. That's minimum spanning tree and nearly linear time.