[SQUEAKING]

[RUSTLING]

[CLICKING]

JUSTIN SOLOMON: OK. Well, welcome back to another problem session. It's great to see everybody's smiling faces again. Right, so today, we're going to continue our discussion of graph theory problems, this time focusing on, well, a little bit of this, a little bit of that, some shortest past problems, all pairs shortest path, and then modification of the Dijkstra's algorithm to finish us off for the day.

So as always, we'll go through our problem session problems in order, for no reason other than that's the way they were presented to me, and I think roughly isomorphic to how most of the homeworks in this class go anyway. But in any event, let's get started here.

So in problem session problem 7-1 here, as always, our first problem is sort of a warm-up to make sure that we understand all these sort of definitions, techniques that we use in 6.006. Today, we're going to go over Dijkstra's algorithm. So if y'all recall, Dijkstra's algorithm is a technique for computing shortest path from a single source to your rest of your graph.

There's about a million different ways to explain, understand Dijkstra's algorithm. So I'll undoubtedly revert upon the one that I remember, which is probably not the one that Jason just covered in lecture. But we'll make it work. And of course, if there are any questions, we'll address those along the way. I'm going to switch to a piece of chalk that isn't an eight of an inch and get started.

Right, so in problem 1A our homework here, we were asked to run Dijkstra's algorithm from vertex s here. By the way, just sort of a standard terminology in graph theory that I think we'll see a lot in the homework is we typically use s for the sort of starting point of a path, or sometimes the source if we're talking about network flows. But I don't think we do those in this class.

And then t is usually the destination. Why t, you might ask? Because it's the letter after s. Right, so our first task here is to compute the single-source shortest path from s to everything else in our graph. Initially, this looks painful. But it's not.

So you're going to forgive me. I'm going to write a sort of shorthand version of Dijkstra's algorithm, because I'm talking to you as I solve this problem, which of course would be much more annoying for you guys to do on paper. But that's life in the city. So there. That's my-- in the words of Britney Spears, it's my prerogative as your instructor.

OK, so in Dijkstra's algorithm, what do we do? We initially label all of our vertices as having distance infinity to the source, or-- and we insert them into our priority queue, except for one vertex, which of course is our source vertex. And he or she has distance 0 for an obvious reason, which is that if my path starts at the source, it has distance 0 to the source.

So as our shorthand-- what was that? Do I want to use red? Yes, of course I do. But see, it's this fat chalk, man. OK. 0. I can make this work. OK. So our convention for today is going to be if a vertex does not have a label on it, it is distance infinity.

So what does Dijkstra's algorithm do? Dijkstra's algorithm grabs the closest vertex that I haven't yet processed, and closest in terms of the distance value that I've stored at that vertex, and then updates all of its neighbors using sort of a triangle inequality style construction. So let's see what that looks like.

So so far, everything is distance infinity away, except for one vertex, which is vertex s, which is distance 0. So obviously, that should be our first iteration of Dijkstra's algorithm. And now what vertex S is going to do is look at all of s's neighbors and update them using the triangle inequality. And if they're closer by having a path through s to the neighbor, then I update the distance. And if it's not, then I don't.

In this case, everything is infinity. So it's pretty clear that I should route my path through s, because any distance less than infinity is smaller than infinity. So in particular, notice that there's an edge of length 8 from s to a. So now, rather than being distance infinity away, I can see that vertex a is really distance 8. Oh, this chalk gives me shivers.

Similarly here, there's an edge of length 7. And I believe that's all of the edges out of s. So we're good. And now the kind of nice thing about the way Dijkstra's algorithm works, which I guess was a little bit implicit in the construction we saw yesterday, but that's OK, is that once I visit a vertex, I never touch it again. It gets frozen in time-- in distance, I suppose.

But in any event, what that means is I'm going to put a little box around it, meaning I'm done with this guy. He is no longer in my queue. OK. Hopefully our pictorial system makes some sense here. Again, on your homework problem, you actually have to write this stuff out. And I'm sorry. That sucks. But I don't have to, because I'm talking to you all today.

OK, so remember Dijkstra's algorithm. We're going to look at our list of vertices we haven't seen yet. So it's everything except for S. Find the one that's closest. And process that one next. So in this case, that's the 7 here. OK, so let's take a look. What are the neighbors of 7? Well, I've got s. And that's-- oh, and d here. OK.

So first of all, let's take a look at s. Obviously, if I have a path that goes through c, back through this vertical edge to s, that path has length 8, right? 7 plus 1. 8 is bigger than 0. So I do not update s. But we actually already knew that, because s was frozen here. So I didn't even have to look at that edge. I could have removed it if I wanted to.

OK. But there's another edge coming out of c, which is pointing toward d, that has length 4. 7 plus 4 is, wait for it, 11. And that's less than infinity. So I update d's distance to 11. And I don't think I've managed to make a mistake yet.

OK, so now, we've looked at all the edges out of c. And c is frozen. And we move on. OK, next, let's see. On the other vertices, we have infinity, 8, and 11. So the smallest of those three numbers is 8. And we're going to update all the neighbors of the 8, which, thankfully, although this graph looks big, they had some mercy on your section instructor today. And really, there aren't that many edges. So this isn't too hard to process.

But here's the thing. There's an edge of length 0 from a to d here. I can get to the a in 8 units. So I can also get to the d in 8 units by traversing that edge. 8 is less than 11. That's good news. Do I want to erase it or scratch it out? What's going to be better? I'll scratch it out, just to be messy.

OK. So now, d has a distance 8 from vertex s. And I believe that's all the edges out of a. So a is set. OK. This fun or what? So now we look at all the edges out of-- oh, sorry. We step back, and we look at all of our vertices. We find the closest one. That's d. And now we've got to update all of d's neighbors.

So thankfully, all the remaining vertices have distance infinity. So what do we know? There's an edge of length 1 here. So we get a 9. There's an edge of length 2. So I get a 10. And I believe those are all the outgoing edges from d. And now d's set. I'm going to start moving faster, because this is hella boring.

OK. Now the next closest edge is the 9. Notice that the 9 only-- or rather, b, I suppose, is the vertex, which is currently at distance 9. It only has one neighbor that hasn't yet been processed, which is e. So we know that that's 9, 10, 11, 12 distance away.

I'm going to check my scrap paper and make sure I haven't made any mistakes yet. 7, 8. Cool. OK. So now our next closest vertex is vertex h, which is distance 10. And aha! If I traverse the edge upward from h to e, I can get a path of length 11. And 11, according to most mathematicians, is less than 12. And hence, we should update the value here.

In addition to that, there is an edge of length 2 out of h, pointing into g. 10 plus 2 is 12. And I believe that's it. Getting there. OK, so the next closest vertex is e. e has no outgoing edges. So e is all set.

After that, we've got g. g has an outgoing edge of length 1 into d. 12 plus 1 is 13, which is larger than 8. So we don't update. And similarly, 12 plus 0 is bigger than 7. So we don't update. But again, the edges that point into vertices that we've already processed, we really don't have to even consider, just by the way that Dijkstra's algorithm works.

Notice that if there were a negative weight edge, we're going to come back to, then that assumption would be problematic. So the takeaway here is that this guy is frozen in stone. And now, notice that our queue is actually empty, yeah? So depending on how we set up our-- well, it's not empty. But it only contains one vertex. And it's at distance infinity. Infinity plus 0 is still infinity. So that's this guy's distance here.

And now our queue is empty. Sorry. OK, so I believe I managed to do that right. Excellent. So the problem asked for two things. It asked for the single-source shortest path distance. It also asked for the traversal order, which I forgot to do while I was doing this problem. But you could retrace it pretty easily.

OK, so that's part A. Then in part B, they say, change the weight of the edge from g to c to minus 6. So we have g to c. And now, instead of 0, we're going to make it minus 6. And the question is, if I ran Dijkstra's algorithm, essentially, what would break?

And I think it's pretty easy to eyeball. Remember that g was essentially the less interesting vertex that we touched in our algorithm here. The only one that we looked at was f. And so this outgoing edge from g actually wouldn't even be seen by Dijkstra's algorithm, thankfully, until we've touched all these other vertices in an identical fashion to our previous traversal.

And then we finally get to g. And now what's going to happen? Well, if I had this edge of length negative 6, what's 12 minus 6? That's 6. And notice that that's less than 7, which is the label of vertex c. But that breaks our assumption, which is that as soon as I visit a vertex, I never have to touch it again, right? Because now I've identified a path through g, back to c, that has a similar distance than the path that c had when it was visited in the queue.

So somehow, spiritually, I should add c back to the queue. But that's against the rules in Dijkstra's algorithm. So for instance, if I did that, I would have to convince myself that the runtime doesn't explode and that this algorithm terminates, which could be a problem if you have a negative weight cycle. Thankfully, we have algorithms for detecting negative weight cycles. But that's a different matter.

OK, so I think this is pretty straightforward. It's just essentially asking you to walk through Dijkstra's algorithm and make sure you understand what's going on. Any questions about that one? Cool. I think this is one of the easier problem sessions. So maybe we'll finish early, except I always say that, and then I talk too much. Sorry. Not sorry.

OK. So, right. So in problem 7-2, this is an extension of a problem that we considered, I believe, two problem sessions ago. And it looks something like this. I'm going to go back into my notes. So remember, in two problem sessions ago, we defined the radius of the graph for an unweighted graph. We came up with an algorithm for computing it and approximating it, all that good stuff.

Now, in this problem, we're going to do basically the same thing. But now we're on a weighted graph, yeah? So in particular, we're going to define a quantity-- this is problem 2-- called weighted eccentricity, which is associated to a vertex in a weighted graph. And it looks like this.

So the weighted eccentricity associated to vertex u is equal to the max over all vertices in my graph of the shortest path distance from u to that vertex. Again, hopefully our audience will catch if I accidentally swap the argument to shortest path. I'm used to thinking of that as symmetric, but it's not, because our graph's directed. Actually, in this problem, I don't think the graph is directed, so-- I forget.

It's directed. OK, good. Yeah, then good. I'll try to be accurate. [LAUGHS] OK, and just like the problem we had two problem sessions ago, so the eccentricity that's associated with a vertex is sort of like the distance to the farthest-away thing in my graph. And now the radius of my graph tries to find the most central vertex, which is the minimizer of weighted eccentricity. And we call that the weighted radius.

And so the rated-- agh. Oof, that's a tough one. The weighted radius is a measurement, which is associated not with a vertex, but rather with the graph. And it's equal to the min over all possible vertices, u, of the weighted eccentricity of u.

OK, and so the problem here is that we're given a weighted directed graph with no negative weight cycle. So it may have negative weights. But they can't have a negative cycle. And the question-- oh, I touched my face. The question is, what is the-- can I find the radius of my graph in time that looks like order mod v cubed?

Now, if you recall from our previous problem session, when we considered computing the radius of a graph, what did we do? Well, we tried to come up with a clever algorithm. And then we realized that that was actually kind of unnecessary. It turned out that the kind of braindead thing, where you just look at the definitions and just make it work, was actually good enough.

And that actually turns out to be the case here, right? This is a good reminder for us all that before we go crazy-- 6.006 is a fun algorithms class. We get to learn about cute references to TV shows and all that. Before we go crazy with that, of course, if there's an obvious algorithm staring us in the phase to solve a given algorithms problem, we should try that first before we try something more clever. And indeed, in this case, that works.

So what are the sort of ingredients that we need to compute the radius? Well, the radius is the minimum eccentricity. So what would be the smartest-- or the simplest thing to do, rather, would be compute the eccentricity for every vertex and take the smallest.

How do I compute the eccentricity for every vertex? Well, I have to have the max distance away from that vertex. So what would be a simple thing to do would be to compute distances between all the possible vertices. And conveniently, in lecture, in some day or another-- I'm a little confused about the time ordering of this class, because of the way we're filming it-- we uncovered an algorithm that computes the distance between every pair of vertices. And that's called Johnson's algorithm.

So if we were to do a totally braindead version of solving this problem, maybe for convenience, the first thing we'd do is compute delta u, v for every possible u, v pair. And because our graph doesn't have a negative weight cycle, we can do that with Johnson's algorithm.

So step one is to use Johnson's algorithm for all pairs, shortest path. And I'll refer you guys to lecture for how to do that. But the important thing is the runtime of this step of our algorithm. So Johnson's algorithm, generically speaking, has ve plus v squared log v-- hopefully I got that right-- runtime.

And what do we know about our problem? Well, I believe we're given that the graph is connected. And so one thing that we can do is notice that e is upper bounded by v squared-- I guess even if it's not connected. I'm sorry. That was a dumb thing to say.

Just generically speaking, our graph is simple. So we know that e, at most, is, I guess, 2v squared, which means that this term is upper bounded by v cubed, right? So we have v cubed plus order v squared log v. And so at the end of the day, the first term wins. And we have that this is v cubed time.

Notice that we're given that budget in the statement of our problem. So this is perfectly fine. In other words, this is a long-winded way of saying, it's kosher to compute all pairs shortest paths in the constraints of our problem here. So that's convenient, because now, in step two, well, maybe now we just keep-- we do the Toucan Sam approach again. We follow our noses.

And well, now that we have our pairwise distances, we can now compute the eccentricity for every vertex, for all u, just directly. Of course, in your homework, you should write out what that means. But here, directly just means that for every u, I loop over every v, and I take whatever value is biggest.

So notice that I have two loops, 1 over u, 1 over v. So this is order mod v squared time. So already, the first term is dominating here. So that's a good thing, I guess. And then finally, we have to take the smallest eccentricity of any u, which, this requires one more for loop.

So this time, I for loop over this array. And I just take the smallest value, right? So this is just one for loop. So that takes order v time. And then we're done, right? So that's our technique for computing the radius. And notice that all I did was translate the definition into an algorithm. I didn't do anything smart in this problem at all.

And then we should really quickly double check our runtime. So step one takes v cubed time. Step two takes v squared times. Step three takes v time. So I add them all together. And of course, the v cubed wins. And that is what was given in our problem as our budget.

OK, so I think the first two problems in this problem session are fairly straightforward. Are there any questions so far? I'm talking fast. Cool. All right, so now we're going to move on to problem 3, involving Atniss Keverdeen, who is probably playing the-- what, the Gunger Hames? Unger Games. I was close. Sorry, Under-- ah, whatever. You get the point.

Yeah, so before I get carried away trying to read Jason's jokes here, what's going on in this problem? So this is problem 3. There's an underground sewer network. I suppose he also could have written this problem about MIT, right? There's all kinds of crazy underground tunnels here.

I remember when I was looking at MIT as a potential undergrad, they had us like schlepping around the tunnels. I thought they were very dirty, and I didn't get the point. So I went to Stanford. But in any event, right. So what I'm given is a map. And this thing has n bidirectional pipes.

I'm not going to write it down, but the problem tells you that they're all connected. They make-- like, you can get from your source to your target moving through the pipes. And right, and they're connected at junctions. But at every junction, there is less than or equal to four things that come together.

So just like in our last session, every time you see a phrase like that, it's like screaming there's a degree bound hiding inside of your graph. And moreover, every junction is reachable from every other junction, I believe. OK. In addition to this, we're given a positive integer length for each pipe. So it's starting to smell like a shortest path problem. But is it? That's our question.

But just to make things a little bit worse, Atniss Keverdeen here is trying to escape through the pipes. And she doesn't want to be detected. And in particular, there exist junctions with motion sensors. And apparently, Atniss's intel is pretty good here. And she knows which of the junctions in her pipe network actually have motion sensors that can detect people moving around.

OK. Now, what this problem is asking her to do is to say, maybe-- apparently she knows where the motion sensors are. But maybe she doesn't know if they're like-- what kind of brand they are. Is it a Microsoft sensor, or an Apple sensor, or something? And of course, the sensors have different ranges.

So Atniss Keverdeen here, she wants to be as conservative as possible when she traverses this pipe network. In particular, what we're looking for, what she wants is in n log n time here, find the path that maximizes the distance to the sensors.

So hopefully this problem makes sense. So you've got some grid graph. Well, not necessarily a grid graph, but a bunch of vertices of valence 4, maybe something like that. It makes sense. She's living in a city somewhere. And she has some source that she's starting at, some destination she wants to go. And then a few of these vertices are marked as having motion sensors at them.

And rather than giving you a radius or something like that, instead, what we're saying is that she wants to go from the source to the target. She's willing to walk a long distance. The length of the path doesn't matter. What matters is that she never wants to get closer to any motion sensor-- maybe there's a second one like here-- than she has to, OK?

So in this case, I guess if I were to eyeball it, it looks like you can't do better than one edge, right? So she would go like that. And of course, in an extreme scenario, it might be the case that there's a sensor in every junction, in which case she's hosed. But we'd like to let her know that before she embarks upon her journey here.

OK, so does our problem make enough sense? Excellent. OK. So, right. So unfortunately for us, this isn't-- again, it doesn't look like a shortest path problem. And the reason is because it's not. But rather, it's sort of a readability problem in disguise. And let's think about what I mean here.

So, right. So there's an obvious graph here-- we'll call it G out of a lack of creativity-- where what I'm going to do is give a vertex per junction. And I'll have an undirected edge for each pipe whose weight is the length.

OK. By the way, I think the length-- I didn't even mention it in the problem description. But I believe she wants to find the shortest path that maximizes the radius. So like, if there are multiple different paths that she could take that both have the same radius from the sensors, then she'd like to be lazy and not walk too far. So that's where this is going to come into play. But that's sort of a secondary concern, I would imagine, on Atniss's behalf here.

OK. And moreover, I'm not going to even attempt to remember the details of this problem. But rather, there's a source, which I'm sure has some cute Hunger Games name attached to it, and some other target. And these are just two nodes in the network of pipes. And she wants a path from s to t.

OK. So first of all, let's just count and make sure that we know. So how many vertices are there? Well, the problem actually gives a name to that. There's order n vertices, because that's just the number of junctions, which is what we define to be n. I guess I forgot to write that there. And because we have a degree bound, our favorite argument in this class, we know that there's order n edges as well. So that's good news.

Why can't I use this graph directly? Like, let's say that I computed the shortest path from s to t. Notice that that completely ignores the point of the problem, right? The point of the problem is that Atniss wants to avoid these starred vertices on our graph up here. But the shortest path may not do that. In other words, you may have to walk like a really indirect path to avoid being detected by the sensors. That's a problem.

So we need to be a little more clever than that. We do have to think on this problem a bit, but not too much. And so here's the basic trick. Like, let's say that-- let's have a slightly different problem first, which is, let's say that I give you a radius k, and I want to know, does there exist a path that can get me from s to t without coming more than distance k away from the sensors?

Notice that once I-- if I have a tool that can answer that like yes or no problem, I could come up with an algorithm that finds my number of sensors by looping over k or something. it may not be fast enough. But I could do that. OK, so that problem is actually not terribly difficult, because essentially, what I could do conceptually is just remove the vertices that are too close to the sensors and then solve a reachability problem.

Like, can I get from s to t without getting distance k away from any one of the sensors? Well, what do I do? I just remove any vertex that's distance k away from the sensors. And then I compute reachability. So conceptually, I think this isn't a huge leap, intuitively speaking. But there's a lot of details to fill in, yeah?

So unraveling just a little bit more, we might define a graph Gk. And Gk is going to be the subgraph of vertices whose distance-- distance bigger than k to any sensor, right? And somehow, reachability in this thing can answer, yes or no, can I get from s to t without coming distance k to a sensor?

By the way, do we know this term, subgraph, in this class? Essentially, it's pretty clear what it is just from the word, right? Like, essentially, I'm just going to remove vertices inside of our larger graph and any edges that touch those vertices. And obviously, if my original graph had order n size, then the subgraph has big O of n size. You might have less. But certainly, it's an upper bound.

OK, so we unravel a little bit more. And somehow, this seems like a convenient structure. But I haven't told you how to compute it. And in particular, the sort of annoying thing is this piece, right? I need to be able to figure out if I'm distance k from any sensor. Or more generically speaking, it might be kind of handy to compute distance from every-- from the set of sensors to every other vertex in my graph, every other junction in my pipe network, OK?

Well, we already covered a trick in our problem sessions that's going to help us do that, right? Because, OK, what would be a very simple algorithm for doing that? It would be to loop over every sensor, call Dijkstra's algorithm for each one. So that gives me the distance to sensor number 1, and the distance the sensor number 2, distance to sensor number 3.

And then I take the min over all those. And that function gives me the distance to any sensor. That's going to be problem, right? Because Dijkstra's algorithm runs in n log n time. But now I've incurred another factor, because I have to loop over all the sensors. And we didn't give you a bound on how many there are.

So more generically, this is actually a problem that shows up all the time in my everyday life, which is that we don't just want compute shortest path to a single point. Sometimes we want shortest path to a bunch of stuff. Like, in other words, I don't care which sensor is close to me. I don't want to get close to any sensor, yeah?

This shows up in geometry all the time. Like, maybe I want to know the closest-- like, I want to find the closest point on the highway. So the highway is driving past my house. So the highway is a whole bunch of points on some network. And I just want to get on the highway and start driving. I don't care about the closest path to every single point on the highway. I just want whatever is closest to me, right? So this is a pretty practical thing to think about.

So how do we solve that? So let's say this is our pipe network. I feel like I draw this network a lot. Maybe I'll spice it up with an extra edge. And maybe, for board drawing purposes, these two vertices have sensors. So I'm trying to find the shortest path distance to either one of these two vertices from every other vertex on the graph, or vice versa. It doesn't matter. It's undirected.

And I don't want to loop over all these sensors. That's the basic headache here. So one thing that I can do-- this is kind of a sneaky trick. And it's exactly the same trick that we've applied a few times in the problem sessions here-- is to add a new vertex that's kind of like a source and make that vertex distance 0 to every one of my sensors. And now what I'm going to do is do single source, shortest path from this new extra vertex that I added to all the rest of my graph.

Now, why do I do that? Well, if you think about it, well, there's an edge of length 0 to any sensor. Topologically, I can think of it like I glued all the sensors into one vertex if I wanted. But that doesn't really matter. I feel like that would be a different way to solve this problem, I guess, would be to take all the sensor vertices, glue them together into one, and then solve this problem. But I digress.

Right, so the shortest path distance from our new source vertex to all the other ones is going to be just the short path to any sensor, because notice that any path coming out of s necessarily has to pass through a starred vertex. OK, so let's write that down.

Basically, what I'm trying to do here is, in step one of my algorithm, I want to label each junction with its distance to a sensor. That's the high-level goal here. And the way that I'm going to do that is I'm going to make-- oops, in the problem solution, they called it x. So I'll be consistent.

I'm going to make a new vertex, or a new graph, rather, G prime, which is equal to my original graph, G, which is coming from the pipe network, with one extra vertex, which we're going to call x, which is connected to every motion sensor with weight 0.

And now I'm going to do Dijkstra's starting at x, which takes n log n time, because I just gave you the size of our graph up here, and gives me essentially the shortest distance to any motion sensor from all the vertices in my graph. So that's a good thing. That's sort of like a convenient piece of information.

When we're solving these kinds of algorithms problems, notice that I've done sort of a similar reasoning in both of the last two problems, which you can do. And actually, it's a pretty practical way of thinking about algorithms, where, like, this problem tells me, at the end of the day, my algorithm has to run in n log n time, right? Or like, in the previous problem, it had run in n cubed time-- I guess v cubed time.

So one thing I can do is say, what is all the information that I can gather out of my graph in n log n time? And I might as well compute it, right? So for instance, the distance to the closest sensor, I just gave you an n log n algorithm for computing it. That seems like a useful piece of information. So what the heck? I might as well compute it in step one and just have it around.

Obviously, you could do breadth-first search on all the computable numbers. And this might not be the most efficient way to solve a problem. But I think for graphs, there's only so many things that we typically want to compute. So it's worth kind of going down your checklist.

Like, similarly, here, notice that we give you a budget of v cubed time. So like, you might as well compute all pairs shortest path, because we can do it in v cubed time. And why not have that information around? It seems useful for computing radius.

OK, so in any event, now, in step one, we now know how close every junction is to every sensor. So now I can argue-- I'm numbering these like steps. But they're not really steps. These are more just like thought bubbles. So thought bubble number two is going to be, how do I actually construct Gk, right?

And notice, I have a nice piece of information here. I now know what vertices are inside of Gk and which ones aren't, right? Because I can just loop over all the vertices. If the distance is bigger than k, I keep it. And if it's not, I don't, yeah? So that gives me an algorithm for computing Gk.

So we can construct-- "construct" with a C-- construct Gk from our original graph, G. And that's really easy to do, just by looping over the vertices and remove any whose distance is too big-- or too small, rather. v. Does that make sense? Because those are the ones that are dangerous. If the radius of my sensor is k, any vertex with distance less than or equal to k, I want to throw away.

And how much time does this take? Well, there's just a loop over the vertices. I guess I need to account for the storage of my graph also. But of course, this graph takes less space than G. So overall, this takes order n time-- Tim-- time and space. But there's a catch, which is this is per k, OK? So every time I want to make a new Gk, I incur an expense of order n.

But this is already getting us pretty close to our problem, because what can we do? If I have Gk, I can say that BFS on Gk establishes reachability from s to t, which is what we care about, outside of radius k.

And how much time does BFS take? Linear in the size of the graph. I only ever ask Jason one question, which is-- always has the same answer. Right, so BFS takes time linear in the size of our graph. Our graph has size n, or kind of 2n-ish. So at the end of the day, this takes order n.

So if our problem were written slightly differently, we would be done, right? If the problem said, given a radius k and a graph, tell me, yes or no, does there exist a path that stays outside of radius k of the sensors? This is how we would do that. Hopefully we all agree. And we can do that in order n time. Oh, just kidding. Order n log n time, because I had to do Dijkstra's algorithm first. Thanks.

OK. But sadly for us, we're not quite done, because we want to find the largest possible k. We want to find the biggest radius that we can stay away from the sensors and still get successfully from s to t. So let's say that I want to find this number. So this is-- I'm going to define k star-- I've got a moving target to write on-- is the largest k where Gk is connected-- oh, that's not quite right-- from s to t.

This is a weird way to phrase it. Really, this should say, where there exists a path from s to t in Gk. I'm sorry. Yeah, where Gk-- I've just phrased this in a funny way-- has path from s to t. There we go. [LAUGHS] All right.

So our question is, how do we find that? Well, here's a dumb algorithm. I could loop over all k's, construct Gk, and then, if my answer is, yes, it's reachable, then go to the next k, increment by 1, and start over, right? So this is the dumb answer. When I say dumb, I mean the answer that your instructor wrote down on his notes and then realized it was dumb, which is loop over k until you get to k star, I guess plus 1, because once I get there, then I get a thumbs down.

Obviously, getting any bigger than that is only going to make my graph smaller. This is kind of filtration, because each graph is contained inside of a different one. Actually, a filtration would be the other way. I'll think about it for later, because this is not a topological data analysis class.

But in any event, if I did that, how much time will it take? Well, remember-- OK, so for one thing, I have n log n from Dijkstra's algorithm. I don't get around that. So I always have to account for that. But now, every time I try a new k, I incur cost n, right? That's what we argued up here.

And so at the end of the day, this algorithm takes order k star n time like that. Of course, it's a little weird to have the answer to your problem in the runtime. But k star here, it could be-- we don't have any bound here. It could be the number of vertices or anything like that, yeah?

And so if we have a budget of n log n time, this doesn't quite work. And so the question is, can we rescue this strategy here? And the answer, of course, is yes, or else I wouldn't be standing here today. One way that you might do that-- so there's the way that you would do that as a real algorithms person. And then there's the way you could do it by psychologically diagnosing your instructors. So let's talk about both of those.

Let's actually do the second one first, because I think that's the most practical if you want to get your homework done quickly, which is as follows. This problem tells you that you have an n log n budget of time in order to run the algorithm. And so what does that mean?

Well, when we loop over potential Gk's that we can try, we have a budget of log n tries before we're done, yeah? So we kind of know that any algorithm that constructs a Gk and tries it can only do it log n times. And to my knowledge, we only have one algorithm that runs in log n time in this class, which is binary search. And so we might be thinking very critically about how we could use that tool.

But more generally than that, I think this is actually a strategy that shows up a lot, both in algorithms and actually in numerical analysis a lot, which is, you have some like yes or no answer. And you want to find the point on the interface where yes flips to no. And so one way to do it is to sort of bound it on two ends and then keep dividing in half. And as long as your relationship is a bunch of yeses and then a bunch of nos, you can keep doing that by binary search.

So let's think about it this way. So we have a long interval of k values. By the way, obviously, there's an upper bound here, which is like the biggest distance to any vertex in my graph, or something like that.

AUDIENCE: The sum of all distances?

JUSTIN SOLOMON: Yeah, like the sum of all distances or some--

AUDIENCE: That would be a very large number compared to n.

JUSTIN SOLOMON: But you can afford a lot. You could-- if you took the sum of every edge-- here's a way to do it. If you took this to be the sum of every possible edge length--

AUDIENCE: Wouldn't that be bounded in n polynomially?

JUSTIN SOLOMON: Polynomially bounded in n?

AUDIENCE: Difference between u and n. [INAUDIBLE] very large numbers in the near space.

JUSTIN SOLOMON: OK. I'm not sure if that's quite right. But that's OK. In any event, let's say that we have an upper bound for k for now. Then what do we know? We know that here's the k star that I want. And to the left, my algorithm will return yes. This algorithm up here, up here, it'll say no.

So one thing I can do-- one thing I should do is put k star not right at the center of my interval for illustration purposes. But now I can binary search, right? Because I could query here. And now I'm going to get a no. And maybe I subdivide at the midpoint for some reason. Now I get a yes. And I can kind of triangulate in on what I want.

So that's our basic strategy is binary search here. But we have to figure out how to do that exactly. OK, so first of all-- OK, there is an obvious upper bound here, which is just the biggest distance from any vertex to any sensor. Right, so we could probably come up with a conservative one. We didn't feel like it.

But conveniently, in step one, remember that computing convenient numbers is always a convenient thing to do. Clearly, if Katniss wants to-- sorry, Atniss wants to go within a radius that's bigger than the distance of any vertex, any sensor, she's in trouble, because that covers the entire graph, yeah? So right. So we actually do have an upper bound here, which is the biggest distance to a sensor.

And now we want a binary search. But we have to be a little bit careful how to do it, because we want to be logarithmic in n, which is like the number of vertices in our graph. And of course, the way that I've drawn this interval here, as Jason points out, I don't, at least immediately, have a bound on this number in terms of n. Like, it could be that my edge weights are like really ginormous.

OK, so right. So how could we get around that? Well, essentially, what we want to be doing is binary search in an array that scales like the vertices. And here's the solution that I came up with, which I'm pretty sure is the same as the one in the answer-- I should really check that before teaching this thing-- which is to do the following, which is a.

Remember, again, we have a budget of n log n. And so we can do a constant number of things that take n log n time. We just might as well keep doing it, yeah? And so another kind of convenient thing we might do is sort my vertices by the distance to x, which, of course, remember, is exactly the distance to their closest sensor.

Why would you do that? Well, in some sense, as I move along that array, that's the sort of order in which I'm going to remove vertices from my graph and make the radius get bigger and bigger and bigger. Does that makes sense? Because these, the first couple ones are the ones right next to the sensor. As I move along this array, they get farther and farther away.

OK, so we're going to say-- and of course, why can we do that? Because sorting-- I think this is one that all computer science students everywhere know-- takes n log n time using whatever your favorite sorting-- well, that's not true-- whatever my favorite sorting algorithm is.

OK, and we're going to take di to be the distance-- the i-th largest-- actually, I think in my notes, I got this one-- the i-th smallest. I'm diverging from my notes. So there is a high likelihood of a sign mistake that's about to happen.

And now what I'm going to do is I'm going to binary search on i. In other words, I'm on the index into my distance array. There's a reason to do that. Is there a reason-- like, let's say that my distances are like 1, 2, 5, 7. So these are the distance of some vertex to a sensor. And let's say that I test 3. And I notice that 3 is admissible. In other words, she can get from one vertex to another, never getting within a radius of 3.

Should I try a radius of 4? Well, no, because I'm going to remove vertices only when I pass an element in this array. So it makes sense to do binary search not in distant space, but rather in array index space, because those are the sort of junctures that determine when I'm going to remove stuff.

So, right. So remember, how big is this array? It's got length n. So overall, this binary search takes log n time. And that's good, because our whole algorithm now-- what do we do? We binary search on d. Or rather, we binary search on i. And then we take di and plug it into our algorithm up there to construct our subgraph. We test yes or no. And that tells us the left or the right interval in our binary search. And we recurse.

And so overall, this takes log n time, or rather, log n steps of binary search. And of course, each one of these steps, as I argued up there, takes order n time. So overall, our algorithm takes order n log-- oh no, I made an accidental theta-- n log n time. And that was the bound we wanted to achieve.

As usual, there's a lot of detritus that we need to clean up at the end of our problem here. One of them is that we actually want to return a path. But that's not so bad, right? So now we've essentially determined that we can compute k star in n log n time.

Well, that's good. So now, if we actually want the path, we can construct our graph one more time, right at k star. I suppose we already have it around from the end of binary search. And then use whatever your favorite reachability algorithm is to go from s to t and give Katniss the path that she wants, or specifically Dijkstra's algorithm if you want the shortest path, which is courteous, because she doesn't want to walk too far if she doesn't have to.

Moreover, let's see. There are a couple of boundary cases that are probably worth mentioning in your problem. So for instance, if k star equals 0-- in other words, I do my binary search. I get all the way back to the first element of my array. And it's still saying I can't do it. What does that mean?

That means that I cannot go from my source to my target node without passing through a vertex that has a sensor at it, in which case, what do you do? Well, you can return any path, or you might as well return the shortest path so that she can run. [LAUGHS] OK. And that, I believe, concludes our problem.

Any questions about that one? Thumbs up, cool. How are we doing on time? As usual, I think I'm going to end early. We're at precisely the same time I'm always at at problem 4. OK, so problem 4, we move from one fictional universe to another here. So now we're playing like Okepon or something like that.

Sorry. If I didn't write so big, I wouldn't have to waste half of class erasing. But that's OK. Right, so cool. So now we have Ashley Gettem. And Ashley Gettem is trying to go from Twinkletown to Bluebluff. And these are both two clearings in the Tanko region. I'm sure if I played Pokemon, this would have meaning to me, or whatever this is.

But in any event, we have a bunch of maps-- we have a map of a bunch of clearings. I'm not liking the erasing on this board. So maybe we'll start up here. OK, so we have n clearings. And they're connected by two-way trails. And thankfully, there's less than or equal to 5 trails connecting in every clearing. This is like our favorite detail to add to 6.006 problems is a degree bound, yeah?

OK. Now, with every trail, we associate a length, which we're going to call l sub t for trail t. But in addition to that, we're going to have another piece of information, which is that it has this set of critters on it, c sub t. So now every trail-- our character here is walking along the path. And what does she want to do? Any time that she walks along a path, she collects a critter, or however many critters are on that path.

In fact, she feels a very similar way to how I feel at TJ Maxx, right? She's walking down the aisles, and like, she can't help herself. She's got to catch every critter that she goes past on the path. There's no option here. She can't leave it behind, because if she does, then she'll be sad.

And the reason why our character might not be able to pick up one of the critters would be if she ran out of her tools for picking up these critters, which are apparently pocket spheres. And she has k spheres. In other words, she has like a backpack. And the backpack can hold that many spheres at a time, OK?

There are a few details here. One is that every time she walks down a path, she collects all the critters. But then if she walks down that path again, she'll collect the same set of critters. Apparently they respawn. They're very prolific, these critters. And moreover, there are stores at some of the clearings. And at these stores, she can get rid of the critters she currently has, and drop them off, and pick up empty spheres instead.

So essentially, every time she does that, she empties her backpack and gets a new set of material, where she can keep picking up the critters. I have many questions about this character. Like, does she just leave them there? Does she come back for them later? What does she do with the critters? There's a lot of questions.

Like, does she have a bigger bag she can go back around to the stores and-- but in this fictional universe, we're not going to worry about these problems, OK? And so, right. So essentially, the question here is that there are two locations. I don't remember the name-- Trundletown to Bluebluff that she's trying to travel in between.

And she gets sad if she comes across a critter that she can't collect. So the question is, can you find the shortest path without being sad? And in like kind of a Dada twist in this problem, if no such path exists-- in other words, it's like, so she has to walk along some long trail with a bunch of critters-- more than k critters, I guess, in the worst case-- then sadness is unavoidable, is what your code is apparently supposed to return, which is really defeatist. I don't know who wrote this problem set.

OK. The question is, how do we solve this? It looks like a shortest path problem. But as with all problems in 6.006, there's a slight twist, right? And in this case, the twist is that it's the shortest path without becoming sad, where sadness means that you ran out of spheres to collect your critters. Whew.

Now, notice that we're given a budget, by the way, of, I believe, nk log nk time. Is that right? Like that. OK. Notice that this is a little suspicious. Somehow it makes us think that the size of our problem really is nk, rather than just n or k.

OK, so how could we do this? Let's think back to a problem that we solved yesterday, under the assumption that you guys are binge watching your 6.006 problem sessions here. And remember, in the problem yesterday, we had this dude that was walking along paths. And like, every third path, he had to drink a beer. I've tried this on my commute home, and it doesn't work terribly well.

But in any event, we have kind of a similar scenario here, where there's some number that we need to keep track of, right? In this case, it's the number of critters that our character has remaining that she can pick up. So she starts with an empty bag. As she collects them, the amount of capacity in her bag decreases until she gets to a store, and then it goes back again.

But the good news is that our runtime bound includes k in it. So it's actually OK to make an algorithm that scales in the number of critters that she can carry around. That's kind of atypical, but an interesting choice here. So remember, the term I introduced last time for this kind of universe is that it's kind of like a state machine. Like, in addition to walking along the graph, she needs to know how much capacity she has for critters.

And so here's a way to do it. So I think, actually, this problem isn't too bad, given that you saw the problem where you do every third vertex in our last problem session. Somehow it's just like a same church, different pew kind of scenario here.

So in this case, one thing we can do is we're going to make a graph. We call him G. He's got vertices v and edges E, just for fun. But what we're going to do is have k plus 1 vertices for every clearing. And the reason is that what we're going to do is we're going to walk along the graph.

And as we traverse our edges, we're not only going to keep track of the costs, like the distance that she's walking, but also the number of critters that she has remaining in her bag that she can score. And the way that we can do that is by keeping a bunch of copies of our graph and ascending every time that we collect a new critter. Does that makes sense?

OK so here's a-- let's just add a little bit more detail here. So in particular, I can define vc comma i is going to be the vertex per clearing comma critter space. And the way that I can view it is that this is sort of representing that I'm at clearing-- oops, that should be a c. And I have i pocket spheres that are empty.

So initially, I'm going to start at v s comma 0 and go from there, OK? OK. I guess it depends whether you're decreasing it or increasing it.

AUDIENCE: You have empty pocket spheres [INAUDIBLE]?

JUSTIN SOLOMON: Oh, you're right. Yeah, I'm sorry. So in that case, I guess it would be k. We'll see if I manage to do that consistently throughout my answer here. OK, so now I need to tell you how to make the edges in our graph. And so let's do that next.

So in particular, for all trails from a-- between a and b, with length l and critters c-- so she picks up c critters. She traverses distance l. She gets from a to b or vice versa. And our trails are bidirectional here.

We need to define edges. And they look like this. So we sort of have two different cases. One is where you have a store. And one is when you don't, because that's going to affect how your state changes. So, right. So the first one would be-- oh, did I get this whole thing backward? Oh, don't tell me that. No, I didn't. OK, good. Great.

So first case would be a does not have a store. So in this case, she leaves with the same number of critters she had in her bag, minus whatever critters she picks up along the way. So now I'm going to have an edge of length l.

And I'm going to go from va comma i to vb comma i minus c. So the idea is that I go from a to b. And in the process, I lose c critters. But I have to do this for all the possible i's that I could see in my state. So that goes from ct to k, right? c, associated with the trail, t, which I decided not to use.

OK, right. So the basic point here is that there are a bunch of different states where I can traverse this trail. But I have to have at least c critters in my bag if I'm allowed to traverse this trail, or else I will be sad, OK? And so I kind of copy this edge a bunch of different times to represent all the possible transitions I can make.

And similarly, let's say that a does have a store. Well, I still want to add an edge. But now I have the luxury of clearing out all the stuff in my bag before collecting the critters, yeah? So now I still want an edge of length l. But now I get to connect more vertices and reset my state in the process.

So now I'm going to go from va comma i to vb. And where am I going to end up? Well, how many critters are going to be in my bag? I have a bag of capacity k. And I just traversed this edge from a to b, which contained c critters.

So I get a v k minus c, like that. And of course, I can do that for all i from-- oh yeah. I have to be careful-- well, no. Actually just for all i is fine. Yeah. I think there might be-- there's either a mistake in how I copied it down or a mistake in the problem-- a mistake in the problem.

AUDIENCE: Do you still need [INAUDIBLE]?

JUSTIN SOLOMON: Yeah, I think this is right. Yeah, so I think there's-- we wrote for all i in some interval, but we didn't have to. OK. Right, so those are our different cases, right? So essentially, in this case, we clear out our queue here. Well, not even a queue, just our set of spheres. We reset and go big here. Notice that the second vertex doesn't even have an i in its index, because it doesn't matter.

OK. And then similarly-- so that gets me from a to b. I also have to do the symmetric thing to go from b to a. OK, so now I have a graph. And first of all, we should reason and make sure that we can actually construct this graph in a reasonable amount of time. So let's do that really fast.

Right. So the number of vertices in our graph-- well, I basically have k plus 1 copies of my graph. So it's equal to k plus 1 times n. And that's good. So that's order kn time-- order kn space, I suppose. And similarly, there's k edges that are associated with every trail. And there's order n trails, because I have a degree bound. So overall, there's order kn edges in my graph, OK?

Great. So now my problem is not so bad. I have a source, which is the place where our walker starts. I have t, which is the destination, where she wants to go. So what do I need? Well, I just need any path that starts at v s, k. In other words, she starts at s and has a full bag with k capacity, and ends up at-- to v t comma i for any i, because it doesn't matter how much capacity she has in her bag when she reaches her destination.

So this is for all i. Any path avoids sadness in the way this problem is written. So how could I do that? Well, she wants the shortest path. So I can just do Dijkstra's algorithm, from v s, k. I'm going to do a bit of cleanup afterward to check all the different i's and find this closest one.

And how much time does that take? Well, my graph takes kn space, because it has kn vertices and order kn edges. So overall, this is going to take order kn log kn time, which is a good thing, because that's exactly what the problem asks for. And of course, if the shortest path is infinity, then what do we know? You can say it with me. Sadness is unavoidable.

[LAUGHTER]

That's just a weird thing to write on a blackboard. OK, and so that's our basic problem here. So what was our strategy? If we step back 10 feet, essentially, we took our graph. And rather than just making our graph, we made k copies of it with edges that kind of point upstairs, meaning that you give away spheres in the process of traversing these different edges, except when you hit the stores, which brings you back to level 0. So that was our basic graph structure. And then we just need to do shortest path on that thing. Any questions about that? Yes?

AUDIENCE: Last time we did graph duplication, we made-- it was layered graph. And you kept making these transitions. And it was a DAG. So can't we do this in linear time?

JUSTIN SOLOMON: Can't we do this in linear time? So in other words, why is our graph not a DAG in this particular case? Let me think about that for a second. There's nothing about this problem that says that the graph has to be a DAG. In particular, I guess you could keep walking-- like, if you had a path with a store that were exactly the same as the number of critters, you could keep walking back and forth along that path. And so there's a little cycle there, which would be enough to not be able to do it in linear time.

AUDIENCE: Well, the edges that we constructed are not-- don't necessarily correspond to the edges in the original trails. That's not in the problem.

JUSTIN SOLOMON: Sure. That's absolutely right. So we constructed a directed graph out of our original one. But that directed graph also can contain cycles, right? So yeah, so I think the example I'm giving you works. So I have a graph with two vertices. Maybe there's other graph stuff going on, but whatever.

I have two vertices here. And I have some number of critters here. And I have a store on either end of my edge, just because I'm boring and conservative. And now, well, assuming that c is less than k, I can just keep going back and forth along this edge as many times as I want. And this is for free, right? Because every time I get to the end of the edge, I throw all my critters away. So that's an example of a cycle in my graph. And because there's a cycle, I'm not in the DAG case anymore.

AUDIENCE: Well, it's a cycle in the constructed graph.

JUSTIN SOLOMON: Sorry. Yeah, it is a cycle here and also a cycle in the constructed graph. Yeah, sorry. I guess that's true. All right, any other questions I can make a hash of here? Excellent. So am I out of time? Ah, shucks. No.

OK. In that case, we'll do the last problem here. Actually, the last problem, I think, is the most interesting one. So as usual, I've blessed myself with not enough time. OK, which of these boards is the tidiest? I think the answer is none of-- yeah, none of the above. So let's use the backs here.

OK, right. So our last problem is a shipping problem, not a trans-shipment problem, which happens to be my area of research, but rather, just a good old shipping thing. So here, strangely, I did not give myself my complete notes. So this will be fun.

Right. So I'm trying to ship servers from San Francisco to Cambridge by truck. And I have all these bunch of third-parity companies that all have pairs of cities that they can ship between, and only so much. These companies are kind of divas. They have a weight limit. They only have cities. They're directed edges. Like, they don't drive their trucks back, apparently, or maybe they pick up somebody else's stuff, and they're just picking your stuff up in the boring direction, or whatever.

So I have a bunch of-- I have n trucking routes. And each route, r sub i, is a tuple with si, ti, wi, ci. And what are all these things? So here, this is the source of every shipping route, every trucking route here. This is the target, right? So this is like where the truck starts. This is where the truck ends.

This is the max weight that the truck can handle. So the truck only has so much space in it. The tires are only so big. And if I put something of weight bigger than w, then my truck is going to die. So that's the most that I can put on this truck. And this is the cost. OK. And I have n routes that look like this.

And I'm trying to ship servers. And of course, some of my servers are too darn busy. By the way, we make an assumption here, which is that our trucking routes form a sort of continuous network. I can get from s to t and ship, at the very least, like a pencil eraser, like some minimal amount of weight.

OK, and so the basic endpoint of this problem is going to be to figure out the heaviest thing that I can ship. And we give you a problem on the path toward that to help prove a little bit about this. This is a very typical kind of setup. So again, I've got a bunch of trucking routes, each one of which has a max weight. And I'm going to want to know, first of all, what's the maximum amount of weight I can ship? And then what is the minimum cost I can ship that weight?

So in problem A, which is marked as useful digression, we're first going to prove kind of a handy inequality. I decided to go off book with my proof ever so slightly of this. So get excited for this to be slightly wrong in a subtle way, which is what I specialize in.

But in particular, we're going to make a definition here. So let's say that pi is a weighted path. Then the bottleneck of pi is going to be the minimum edge weight of any edge in pi.

And to make sure that we see how this is connected to the problem, if I'm trying to ship a server, and I have to put it on truck 1, and truck 2, and truck 3, and truck 4, and truck 5, obviously, of those five trucks, the one whose capacity for weight is the smallest is the only one that matters in terms of the heaviest thing that I can ship.

OK. So, right. Now, given a directed graph and two vertices, s and t, going to define a quantity called b of s comma t. And we're going to say that this is the max over any path, pi, from s to t of the bottleneck of pi.

OK, so to sanity check this quantity here, essentially, what it's saying is, at the end of the day, I have this big network of trucking routes. I just want to go from one city to another. And I don't care which series of trucks I want to use. I just want to maximize the weight that I can ship, right?

And so this is saying I'm going to look at all the different paths that I could take. Each path is sort of band limited by the one truck that has the lightest weight that it can carry. And I'm going to find the path that has the best bottleneck as measured by this quantity. I'm using the word "best" because I tend to make sign mistakes. And that's a vague term.

OK. And then, right. So I'm writing too big. Hopefully I won't run out of space. We're going to make one additional definition, which is I of t is the incoming neighbors of t. And then the problem is asking you to prove a particular inequality here, claim, which is that b of s, t is bigger than or equal to the min of two values, b of s, v, or w of v, t, for all v in I of t.

So let's see what this is saying. So remember that this is like the capacity that I can ship from s to t using any route, and that that upper bound's the minimum of two things, right? Either the capacity of shipping to one of my neighbors with an incoming edge or the weight of basically shipping from that neighbor to me. Notice, this is kind of similar to a triangle inequality, which is why we're going to kind of know how to solve part B of this problem pretty easily.

OK. And moreover, with equality for some v star in i of t, OK? Guess what that v star is going to be. It's going to be the vertex that's sort of the previous one on the path of trucks that I actually take to ship stuff, right?

OK. So how could we prove this? So the intuition here is that the path that's actually giving you this bottleneck, this b s comma t, has to include one of my incoming edges. And I'm just trying to find it, right? That's roughly what's going on in this inequality.

And by the way, whenever a problem asks you to prove an inequality, just step back. Like, I feel like I spend like 85% of my day with research students sort of saying, like, OK, but like, what is this really telling me about life? And this is a good example.

OK, so to that end, I'm going to make a definition, which I found to be convenient when I was solving this problem, which is the following. I'm going to say that pi s comma t is going to be the actual path that gives me the bottleneck. So this is kind of like the arg max over all pi-- I'm going to use really bad notation-- pis that connect s through t. That's how we're going to think of the bottleneck.

So in other words, this is the path that actually realizes this quantity, b, right? So in other words, b of s, t is equal to bottleneck-- pi only has two lines-- s, t. This is convenient for me, because the reasoning about that path makes some sense.

OK. So now, we're going to prove the inequality first. And then we may or may not prove the equality case, depending on how I feel. OK, so right. So let's section this off and try and use my board a little more conservatively here.

OK, so right. So now let's take a vertex, v, from the incoming edges here of t, because that's what we need to prove this. And we're going to prove the inequality directly here. In particular, we can define a path that goes from s to t as follows.

I could take the pi s, v, and then concatenate onto the end of this guy, t, because we know that there's a directed edge from v to t, by definition of what this i is, OK? Right. And so this is kind of convenient, because this is like the bottleneck path for one of my neighbors. And now I'm just kind of sticking an edge on there.

And we know that, of course, this is a candidate path to get from s to t. But it may not be the one that actually achieves the bottleneck. OK? Right, so yeah. And let's-- oops, let's define that to be pi twiddle, just for fun. I like to give things names.

OK. So what do we know? We know that b of s comma t-- well, this, by definition, is the max bottleneck of any incoming path going from s to t. This is a path from s to t, yeah? So because this is the max, it is bigger than or equal to bottleneck of pi twiddle, right? That's the nice thing about maxes, is they tend to be bigger than other stuff.

OK. Well, what is the bottleneck of pi twiddle? Well, remember our definition of bottleneck here, right? It's the min edge weight over my entire path. Well, I have two options. Either that edge weight is the edge from v to t, or it's the edge weight that's associated to the rest of this stuff, yeah?

So either this is the min of the bottleneck of the first segment of our path, pi s, v-- by the way, that may be empty. And that's OK. That's just like a one-edge path. We can dispense with that case pretty easily. Or the weight of v, t. Cool?

Well, by definition, bottleneck of pi s, v is exactly this quantity, because that's what I chose it to be over here, yeah? So this is exactly min of b s comma v-- that was supposed to be a v, but I wrote a t-- or w of v comma t. And that is exactly what we wanted to prove, yeah? So that takes care of our inequality case.

Do I want to do the equality case? I don't think I do, yeah. So it's not too hard to check the equality case. Essentially, you can make a pretty easy contradiction, right? Because if it's strictly larger, then what does that mean? That means that every single incoming edge, none of them can be the edge where your bottleneck path comes in. And obviously, there's something wrong with that. OK, so I'll refer you guys to the notes for that one.

And finally, the problem says, assuming that-- so now we have kind of a funny constant in here, just to make your life a little bit annoying. We have at most 3 square root n cities that we care about. Again, why 3? Why not? I think.

And what we want is the weight of the single largest server and the minimum cost to ship that thing. So we want, one, the largest weight we can ship. We'll call that w star to ship. And two is the smallest cost to ship that weight, w star, OK?

So let's do two first, because it's easy. So let's say that I was able to compute w star. So now what can I do? Well, I can construct a graph where I only keep around the trucks that can ship at least this amount of weight, because the other ships are-- I really want them to be boats in this problem. But the other trucks are useless, right? If they can't cover w star, then they're not pulling their weight, yeah?

So to do part two here, what can I do? I can make a new graph, G sub c, with a vertex for every city and-- right-- and an edge-- a directed edge, sorry, per shipping route with-- this is going to be an unfortunate clash in terminology.

I'm going to say edge weight to refer to the edge weight in my graph. And that's going to be equal to the cost of the edge. And only keep around-- but only for routes that can carry at least w star amount of weight.

And now what do we do? Well, it's just shortest path, right? At this point, shortest path, which is really the minimum cost in this case, because that's what we're associating to the edges, is going to give me the cost of shipping w star in my network. I think that's pretty obvious from this construction.

So that's Dijkstra, D-I-J-K-S-T-R-A. And now the only question is, how much time does it take? So let's say that I do the braindead version of Dijkstra, where I just use a direct access array to do my priority queue. So if you recall, that'll take big O of mod v cubed here-- squared? Squared. Yep, you're right. Sorry. I was thinking Johnson's algorithm.

OK. Well, in this case, how big is v? Well, it's square root of n, 3 times square root of n, at most. So of course, v squared is really big O of n. And that part is solved, yeah? So really, our only remaining problem here is to do part one. And I don't have nearly enough time. In fact, I'm completely out of time. So maybe we'll just talk through it really briefly. Sorry about that.

Essentially, the basic observation here is that the inequality that we proved in part A of our problem is kind of like the triangle inequality of the bottleneck world, in some sense. Essentially, what it's giving you is some update formula that looks kind of like the update formula that we would apply in Dijkstra's algorithm for shortest path, the basic assumption here being that, well, my shortest path has to come from somewhere, right? So if I look at all my neighbors, and then I kind of add in my closest edge, or rather, the sum of the path length to the previous vertex plus the edge to me, that gives the shortest path to me.

Here, instead of that, we're saying the biggest bottleneck path. So what do I do? I look at all of my incoming neighbors. I find the one with the biggest bottleneck, which is band limited by their bottleneck, plus the bottleneck of the incoming edge. And then I update myself. So the only thing remaining here is to basically do exactly Dijkstra's algorithm. But rather than updating by summing edge lengths, we update by using this formula. And essentially, everything else remains the same.

So conveniently, I've run out of time. So I won't even have to jumble this one on the board. I'll let you guys do that one at home. But I actually think the explanation in the written material is pretty self-evident for this problem. So that's probably just as well.