PROFESSOR: Let's get started. Thanks for coming to lecture. Know there's a quiz coming up. There will be a tangible benefit of attending this lecture. And it's not Frisbees. OK? We'll figure it out soon.

So, two lectures on approximation algorithms. One today and one, I guess a week and two days from today on Thursday after the break. Eric will give that one. So this is more of an introductory lecture. Eric talked about NP complete problems and NP hard problems. He talked about how you could show that problems are NP complete or NP hard. So what happens when you discover that a problem is NP complete or NP hard?

Well, there's a variety of strategies. You could just kind of give up, and say this is intractable. I want a different job. You could say that I'm just going to do the best I can without any theoretical guarantees. I'm going to use a heuristic. I'm going to think of the simplest, greedy heuristic. I'm going to code it up and I'm going to move on.

Or you could do approximation algorithms. You could say, I'm going to think up an interesting, greedy heuristic. But I'm going to prove that this greedy heuristic, in every conceivable situation with respect to the inputs, is going to be within some factor of optimal. Right?

And that's what we're going to do today. We're going to take a bunch of NP complete problems, and we're going to essentially create simple heuristics with these problems, simple strategies that are polynomial time, to quote, "solve these problems." And what does it mean to solve these problems? Well, you know that if it's polynomial time, you're not guaranteed to get the optimum answer every time. But you'll call it a solution-- an approximate solution-- because you're within a factor of two for every possible input. That's one example.

Or, you have a more complicated approximation factor that we'll get to in a second, where it's not quite a factor of two. It might be a factor of two for small size problems. Might be a factor of 10 for larger problems. And so on and so forth.

And then the last thing is, it'd be great if you could spend more time and get better solutions. And those are approximation schemes. And we'll look at approximation schemes as well. So,

just dive in each of these problems, depending on whether it's decision or optimization is NP complete or NP hard. I'll define these problems as we go along.

But basically, the name of the game here is, grab a problem, define it, think of an interesting heuristic, do a proof. OK? And that's essentially what we're going to do three times. The good news is the proofs aren't tortuous. They're not 30-minute proofs. And they should be pretty intuitive. And we'll see if we can extract them out of you. A painful extraction process. I went to the dentist yesterday. Not for extraction.

I should've said, I went to the dentist yesterday and I'm now going to take it out on you, right? OK, so what's an approximation algorithm? An algorithm, for a problem of size n, and so it's going to be parameterized. And the approximation factor may also be parameterized. It'd be nice if it weren't, if it were a constant factor approximation. But sometimes you can't do that. And in fact, sometimes you can prove that constant factor approximation algorithms don't exist. And if they do, then p equals NP. All right? So it's gets very interesting.

So, in this case, approximation algorithms or schemes exist for these three problems, which is why we are looking at them today. But you got a problem of size n. And we're going to define an approximation ratio, row of n, for any input-- if for any input-- excuse me. The algorithm produces a solution with cost C that satisfies this little property, which says that the max of C divided by Copt divided by-- oh, sorry-- and Copt divided by C is less than or equal to row n.

And the only reason you have two terms in here is because you haven't said whether it's a minimization problem or a maximization problem. Right, so of it' a minimization problem, you don't want to be too much greater than the minimum. If it's a maximization problem, you don't want to be too much smaller than the maximum. And so you just stick those two things in there. And you don't worry about whether it's min or max in terms of the objective function, and you want it to be a particular ratio. OK?

Now I did say row of n there. So this could be a constant or it could be a function of no. If it's a function of n, it's going to be an increasing function of n. OK? Otherwise you could just bound it, and have a constant, obviously.

So, you might have something like-- and we'll see one of these-- log n approximation scheme, which says that you're going to be within logarithmic of the answer-- the minimum or maximum. But if it's a million, then if you do log of base two, then you're within a factor of 20,

which isn't that great. But let's just say if you're happy with it, and if it goes to a billion, it's a factor 30, and so on and so forth. Actually, it could grow.

So that's an algorithm. If these terms are used interchangeably, we'll try and differentiate. But we do have something that we call an approximation scheme. And the big difference between approximation algorithms an approximation schemes is that I'm going to have a little knob in an approximation scheme that's going to let me do more work to get something better. OK?

And that's essentially what a scheme is, where we're going to take an input-- an additional input-- epsilon, strictly greater than zero. And for any fixed epsilon, the scheme-- it's an approximation scheme as opposed to an algorithm-- is a 1 plus epsilon approximation algorithm.

And so here we just say that this is a row n approximation algorithm if it satisfies this property. And here we have a family of algorithms that are parameterized by n in terms of run time, as well as epsilon.

And so you might have a situation where you have order n raised to q divided by epsilon running time for an approximation algorithm. And what this means is that if you're within 10% of optimal, then you're going to put 0.1 down here. And this is going to be an n raised to 20 algorithm. Polynomial time! Wonderful! Solve the world's problems. Not really. I mean, n raised to 20 is pretty bad. But it's not exponential. Right?

So you see that there is a growth here of polynomial degree with respect to epsilon. And it's a pretty fast growth. If you want to go to epsilon equals 0.01, I mean, even n raised to 20 is probably untenable. But certainly n raised to 200 is completely untenable. Right?

So this is what's called a PTAS, which is probabilistic time approximation scheme. And don't worry too much about the probabilistic. It's a function of epsilon. That's the way you want to think about it. The run time. And we'll look at a particular scheme later in the lecture.

But clearly this is polynomial in n, OK? But it's not polynomial in epsilon. All right? So a PTAS is going to be poly in n, but not necessarily in epsilon. And I say not necessarily because we still call it a PTAS. We just say, fully polynomial time approximation scheme, FPTAS, if it's polynomial in both n and 1 over epsilon. Right? So fully PTAS is poly in n and 1 over epsilon.

So a fully PTAS scheme would be something like n divided by epsilon square. OK? So as epsilon shrinks, obviously the run time is going to grow because you've got 1 over epsilon

square. But it's not anywhere as bad as the n raised to 2 divided by epsilon in terms of its growth rate. OK?

So there's lots of NP complete problems that have PTAS's and that some of them have FPTAS's as well. And so on and so forth. Question?

AUDIENCE: [INAUDIBLE]

PROFESSOR: So we won't get into that today. But you can think of it as the probability over the space of possible solutions that you have. You can distribution of inputs. The bottom line is that we actually won't cover that today. So let's shelve for the next lecture, OK?

So just worry about the fact that it's polynomial in n and not in epsilon for the first part. And it's polynomial in n and epsilon for the second part. OK?

AUDIENCE: [INAUDIBLE]

PROFESSOR: Oh I see. So, it's a good point. So for the purposes of this lecture, thank you so much. I'm glad I didn't have to get into that. Polynomial time. Good. Better answer. Either way, we're not going to cover that. All right? Good. So, it's polynomial. I mean, you could have probabilistic algorithms that have this kind of behavior, of course. But we're not going to cover that in today's lecture. But thanks for pointing that out Eric.

So, that's our set up. We essentially have a situation where it'd be great if we could tackle NP complete problems using this hammer, right? Any questions so far? All right.

Vertex cover. So let's dive right in. Let's talk about a particular problem, very simple problem. What you have is an undirected graph, G(V,E). And all we want is a set of vertices that cover all of the edges. So a set of vertices that cover all edges. What does it mean to cover? It's the obvious thing. If I have something like this. As long as I have a vertex in my set of vertices that I'm calling a cover that touches one endpoint of an edge, we're going to call that edge covered, OK?

So, in this case it's pretty clear that the vertex cover is simply that. Because that vertex touches all of the edges at least at one of the endpoints. A vertex is going to touch one endpoint of an edge. But this vertex cover that I've shaded touches every edge. So that's a vertex cover.

If, in fact, I had an extra edge here, then I now have to pick one-- or this one of that one in order to complete my cover. OK? That's it. That's vertex cover. Decision problem, NP complete to figure out if there's a certain number that is below a certain value that do the covering. You obviously have an optimization problem associated with that. And so on and so forth. So that's our simple set up for our first hard problem. All right?

And so, just to write that out, find a subset V prime, which is a subset of capital V, such that if (U,V) is an edge of G-- belongs to E-- then we have either U belonging to V prime, or V belonging to V prime, or both. And it's quite possible that your vertex cover is such that, for a given edge, you have two vertices that touch each of the endpoints of the edge. And the optimization problem, which is what we'd like to do here, is find a V prime so the cardinality is minimum. OK? So that's it.

So, we don't know of a polynomial time algorithm to solve this problem. So we resort to heuristics. What is an intuitive heuristic for this problem? Suppose I wanted to implement a poly time, greedy algorithm for this problem. What would be the first thing that you'd think of? Yeah, go ahead.

AUDIENCE: [INAUDIBLE]

PROFESSOR: Find the max degree. I love that answer. It's the wrong answer for this problem. But I love it because it sets me up for-- ta-DA! All this work I did before lecture, OK? All right.

So, it turns out, that it's not an incorrect answer. It's really not the best answer in terms of the heuristic you apply to get an approximation algorithm. So we're still in the context of an approximation algorithm, not an approximation scheme. And what we have here is a perfectly fine heuristic that, who knows? It might actually work better in practice than this other approximation algorithm that I'm going to talk about and prove.

But the fact of the matter is that this approximation algorithm that has, as a heuristic, picking the maximum degree continually. And completing your vertex cover by picking the maximum degree continually is a log n approximation algorithm. And what that means is that I can construct-- and that example is right up there-- an example where, regardless of what n is, this particular heuristic-- the maximum degree heuristic-- might be log n off from optimal. OK?

Whereas, this other scheme that we're going to talk about is going to be within a factor of two of optimal regardless of the input that you apply. Right? So, you have a domination here with

respect to the two approximation algorithms. You've got one that is log n. Row n is log n, as I've defined over there. And on the other side, you've got two, right? So if you're a theoretician, you know what you're going to pick. You're going to pick two. It turns out, if you're a practitioner, you might actually pick this one, right? But this is a theory course.

So what is going on here? Well, this is a concocted example that shows you that a maximum degree heuristic could be as far off as log n, right? And so if you look at what's going on here, you end up with something where you have a bunch of vertices up on top, OK? And you end up with case k factorial vertices up on top.

So k equals three in this case. I have six vertices up there. I got two down here because this is 6 divided by 3, because k is 3. And then I got 6 divided by 3 here, so that's 2 and 6 divided by 1 here. And so that's 6. OK? And so these edges are set up in such a way that it's a pathological example. And I misspoke in terms of the approximation algorithm. I will correct myself in just a second, in terms of log n. It does grow with the size of the graph. Well, I'll precisely tell you what this approximation algorithm is in terms of the row n factor in just a minute.

But let's just take a look at this problem here and see what happens when you apply this maximum degree heuristic, right? And we have to take into account the fact that, if you have ties, in terms of maximum degree, you may end up doing the wrong thing. Because you haven't defined what the tiebreak is when you have two nodes that have the same degree. You could do the wrong thing and pick the bad node for this particular problem, right? You have to do a worst case analysis.

So in the worst case, when you create a vertex cover using maximum degree, what is the worst case in terms of the number of vertices that we picked for this particular example? Someone? What is the worst case in terms of the number of vertices? Yeah, back there.

AUDIENCE: Eleven?

PROFESSOR: Eleven. And where did you get that from?

AUDIENCE: [INAUDIBLE]

PROFESSOR: You grab all of the ones on the bottom. Fantastic. All right, there you go. Could you stand up? Whoa. All right. It was the dentist yesterday.

So, that's exactly right. That's exactly right. So what could happen is you could pick this, because that's degree 3. Notice that the maximum degree here is 3, of any node. Right? So if I pick something of degree three, I'm good. I'm in keeping with my heuristic. I could pick all the ones at the top, right? And then I'm done, right? That's a good-- that's a good does solution. That's a good trajectory. But all I've said is, the heuristic is maximum degree. So there's nothing that's stopping me from picking this. And then once I pick that, I could pick this one. And then I'm down to, once I've taken away these two, remember that now the maximum degree in the entire graph is two. Right? Because each of these things is losing the degree-- losing one from its degree-- as I go along.

So then I could pick this one, this one, this one, et cetera. And so I could end up with 11. OK? So if you go do the math really quickly-- and this is where I'll correct what I said before-- the algo could pick all the bottom vertices. And so the solution and the top vertices are optimal. Top optimal. So that's k factorial, right? According to my parameterized graph. That's k factorial in terms of the optimal solution for this graph. But if I pick the ones that are the bottom, then it's k factorial divided by k, plus 1 over k minus 1, plus da da da da, plus 1. Which is our harmonic number. And that's approximately k factorial log k, OK? And this is where I misspoke. I kept saying log n, log n.

But that's not completely correct. Because if I think of n as being the size of the input, k factorial is n, right? And so if you see that I have log k here, then remember that this is log k where k factorial equals n. So this is another log factor, roughly speaking. Right? So think of it approximately as log log n approximation, OK? Which is pretty good. But it does grow with n, right? The point is this does grow with n.

So it's not the best approximation scheme that you can think of. Because the approximation factor grows with the size of your problem. So it'd be great if you could come up with a constant factor approximation scheme that would beat this one, certainly from a theoretical standpoint, right? But this one, maximum degree, chances are, if you're a practitioner, this is what you'd code. Not the one I'm going to describe. OK?

But we're going to analyze the one I've described. I've just shown you that there is an example where you have this log k factor. We haven't done a proof of the fact that there's no worse example than this one. OK? So I'm just claiming, at this point, that this is at best, a log k approximation algorithm. We haven't actually shown that it is, in fact, a log k approximation algorithm. At best, it's that. OK? Any questions?

All right. So what's another heuristic? What's another heuristic for doing vertex cover? We did this picking the maximum degree. Nice and simple. But it didn't quite work out for us. Any other ideas? So, I picked vertices. What else could I pick? I could pick edges, right? So, I could pick random edges. It turns out that actually works better from a theoretical standpoint.

So, what we're going to do here is simply set the cover to be null. Go ahead and set all of the edges to be E prime. And then we're going to iterate over these edges. I'm not even specifying the way I'm going to select these edges. And I still will be able to do a proof of 2 approximation. OK? Oh! I forgot the best part. This is on your quiz. That was the tangible benefit of attending the lecture. So copy that down.

So this is very simple. It's not a complicated problem. This is not simple heuristics that are going to be particularly complicated. You just do some selections, and then you iterate over the graph. And you take away stuff from the graph. Typically, you take away vertices as well as edges. And you keep going until you got nothing left, right? And then you look at your cover, and you say, what is the size of my cover? And here we return C, all right? So I won't spend any time on that. You can read it. It's simple iterative algorithm that fixed edges randomly and keeps going.

So, now comes the fun part, which is we need to show that that little algorithm is always going to be within a factor of 2 of optimal, OK? And you can play around with this example. In fact, in this case, you have 6 and 11. So that's a factor of 2, of course. So even this algorithm is better than a factor of 2. But it won't be if I expanded the graph and increased k. But that algorithm that I have up there is always going to be within a factor of 2. And we want to prove that, all right?

So how do we go about proving that? We want to prove, in particular, that a prox vertex cover is a 2 approximation algorithm. So, any ideas? How would I prove something like this? Where do you think this factor of 2 comes from? Someone who hasn't answered yet. You answered a lot of questions in the past, all right? No? Someone? All right.

AUDIENCE: [INAUDIBLE]

PROFESSOR: I'm sorry?

AUDIENCE: [INAUDIBLE]

PROFESSOR: That's an excellent observation. The observation is that the edges we pick do not intersect each other. So, I gave you a Frisbee for the wrong answer. So for this correct one, I won't give you one? That's fair, right? No. That's unfair. Right? There you go. Sorry.

So, the key observation was in this algorithm, I'm going to be picking edges. And the edges will not share vertices, right? Because I delete the vertices once I've picked an edge, correct? So there's no way that the edges will share vertices. So what does that mean?

Well, that means that, I'm getting-- let's say, I get A edges. So let A denote the edges that are picked. So I'm going to get edges that look like that. OK? And I got continuality of A edges. I know that in my vertex cover, that, obviously, I have to pick vertices that cover all the edges. Now I'm picking edges, and what's happening, of course, is that I'm picking 2 A vertices.

So my C-- and remember, my cost was C. And I had Copt, that corresponds to the optimum cost. And so the cost that this algorithm produces is 2 times A, right? Make sense? Because I'm picking vertices, we are picking edges. There's no overlap. And therefore, the cost is 2 times A, right? So as long as I can now say that Copt, which is the optimum, is less than or equal to A, right?

I have my factor of 2 approximation algorithm. So that's it. It's a simple argument that says now show that Copt is at least A. Copt, I'm minimizing. Copt should be at least A. Right? I hope I said that right before. But I wrote that down here correctly. So if I say that Copt is at least A, then I got my proof here of 2 approximation. Because I'm getting 2 A back, right? So if you go look at C divided by-- so this means, of course, that C is less than or equal to 2 Copt, if I can show-- make that statement.

And it turns out that's a fairly easy statement to argue simply because of the definition of vertex cover. Remember that I'm going to have to cover every edge, correct? So I'm going to cover-- need to cover every edge, including all edges in A. A is a subset of edges. I have to cover all the edges. Clearly, I have to cover the A edges, which are a subset of all of the edges, right? How am I going to cover all of the A edges that happened to all be disjoint in terms of their vertices? I'm going to have to pick one vertex for each of these edges, right? I mean, I could pick this one or that one. But I have to pick one of them. I could pick this one or that one. And so on and so forth, right?

So it is clear that, given this disjoint collection of edges corresponding to A, that Copt is greater than or equal to A, OK? And that's it. So I had to cover all. This requires, since no two edges

share an endpoint, this means that I need to pick a different vertex from each edge in A. And that implies that Copt is greater than or equal to A. All right? Any questions about that? We all good here? Yup? Understood the proof?

So that's our first approximation algorithm where we actually had a proof. And so, this is kind of cool. It obviously a pretty simple algorithm. You're guaranteed to be within a factor of 2. It doesn't mean that that's the best heuristic you can come up with. It doesn't mean that this is what you'd code. But this is the best approximation algorithm that you're going to cover for vertex cover, OK?

So, what about other problems? What's the state of the world with respect to approximation? There's lots of NP complete and NP hard problems for which we know approximation schemes. And we like to move towards approximation schemes slowly. But I'd like to look at a problem that perhaps is a little more compelling than vertex cover before we get to approximation schemes. And that's what's called set cover.

So set cover tries to cover a set with subsets. And it's very useful in optimization where you have overlapping sets, maybe it's schedules, it's tasks, it's people getting invited to dinner, et cetera, and you want to make sure everybody gets invited. You want to invite families, and there's overlapping families, because people have relationships. And you want to eventually minimize the number of dinners you actually have to have. And that's, I don't know, hopefully a motivating example. If it wasn't, too bad.

So you do have a family of possibly overlapping subsets. S1, S2, Sm, subset of equal to x. So that's that big set that we have. Such that I want to cover all of the elements. So that's what this little equation corresponds to. The union of all the selected SI's should equal x. I need to cover it all. And I do want to minimize. It's called a C. Find C subset 1, 2, m. So I'm selecting a bunch of these things. So C is simply-- capital C here is some subset of the indices. And the only reason I do that is to say that I want to do this while minimizing. I wanted to I equals 1 through m while minimizing C.

OK? Let me get this right. So this is what I have. Find C, subset of these, such that-- I'm sorry. There's one more. Union of I belonging to C, SI equals x. OK? Sorry for the mess here. But this last line there-- so this is simply a specification of the problem. I'm going to be given x, and I'm going to be given a large collection of subsets, such that the union of all of those subsets are going to cover x. And now I'm saying, I want to look inside and I want to select all of them.

I want to select a bunch of these things. You know, C, which is some subset of these indices, so 1 may not be in it. 2 may be in it. 4 may be in it, et cetera. And such that those subsets that are in this capital C set add up to x. OK?

So pictorially, may make things clearer. You have, let's say, a grid here corresponding to x. So each of these dots that I'm drawing here are elements that need to be covered. So that's my x. And I might have S1 corresponding to that. This is S3, right? And S2 is this thing in the middle. That's S2. S5 is this one here. And I got a little S4 over here. And finally, I got-- let's see-- S6, yup. Which is kind of this funky thing that goes like that.

So this thing here is S6. All right? OK. What's the optimum? You got 30 seconds. What's the optimum cover? Yeah, go ahead. You had your hand up.

AUDIENCE: [INAUDIBLE]

PROFESSOR: Yep. That's exactly right. So the optimum S3, S4, S5. All right.

AUDIENCE: [INAUDIBLE]

PROFESSOR: Oh. S3. Yeah, S4 is this one here. S6, S5, OK good. S3. And then-- oh! You know what? You're right. Let's make you right. Don't erase that. Here you go.

So that's 3, right? So C would be-- cardinality of C would be 3. So it's a nontrivial problem. It's not clear how you're going to do this. I've got to use a heuristic. Hard in terms of optimization. Optimal requires exponential time, as far as we know. And we're just going to go off and say, hey let's design an approximation algorithm, right? So let's think of a heuristic. What's a good heuristic? What's a good heuristic for this problem? I hope I haven't scared you. What's a good heuristic for this problem? What's the obvious heuristic? Yeah?

AUDIENCE: [INAUDIBLE]

PROFESSOR: The largest subset. And in this particular case, it's actually the best also in terms of theory. So, approximate set cover-- at least the best that we're concerned about in this lecture. And what is approximates set cover? It's pick largest SI. Remove all elements in SI from x, and other Sj. So you're constantly shrinking. And then keep doing that. So you'll have a new problem. And you're going to specify that new problem on every iteration, just like we did for vertex cover and we've done many a time.

If you do that over here, notice that what you end up with is picking S1, because S1 is the big boy here, in the sense that it's got six elements in it. Right? Up on top. And so you'd pick approx or heuristic algo would pick S1, S4, S5, and S3 in that order. I won't go over it. It's not that important.

The point is it doesn't get you the optimum for this problem. And in general, you could always concoct examples where any heuristic fails, of course, right? Because this problem is hard. But it's four as opposed to three. And the big question, again, as always, is, what's the bound? What's the bound if you applied this heuristic? And what can you show with respect to the approximation algorithm? What is row n here?

So that's what we have to do here, in terms of what the bound is. And we're actually going to do an analysis here that is pretty straightforward. It's got a little bit of algebra in it. But if you go look at that this, it's covered in CLRS, the textbook. But the analysis in there uses harmonic numbers, and is substantially more complicated for, in my mind, no reason.

And so we have a simpler analysis here that is simply going to be a matter of counting. We are picking the maximum number of elements every time. The best we can do. It's a greedy heuristic. We're trying to shrink our problem as much as possible. Initially we have x. And then we're going to get a new problem, let's call it x0 first, for the initial problem. You're going to get a new problem, x1. And we're maximally shrinking x1 in relation to x0, in the sense that we're going to remove as many elements as we can. Because that is precisely our heuristic.

So the big question is, as we go from the biggest problem that we have, the original problem to smaller and smaller problems, when do we end up with nothing? When we end up with nothing, that's when the number of iterations that corresponds to the number of SI's that we picked is going to be the collection of SI's in our solution. And the cardinality of that is our cost, right? So that's all pretty straightforward, hopefully.

So what we need to do, of course, is show a proof. And the way we're going to do this is by a fairly straightforward counting argument. Assume there's a cover Copt, since that Copt equals t. OK? So the cardinality of Copt equals t. So I'm just assuming that this t subset's in my optimum cover, OK? t subset's in my optimum cover.

Now let x of k be the set of elements in iteration k. And let's assume that x0 equals x. So initially, I'm at 0. And I want to subscript this because I want to point to each of the problems that I'm going to have as I shrink this set down to nothing.

Now I know that for all k, including of course x0, xk can be covered by t sets. OK? I mean, that's kind of a vacuous statement because I assumed that x0 could be covered by t sets. And x0 is only shrinking to x1, to x2, et cetera. And I'm just saying, all of these things could be covered-- each of those intermediate problems as well, can be covered-- by t. In fact, in the solution that we have, the optimal solution, if these x0's are coming from my heuristic. But if they were coming from an optimum solution, then x0 would be covered by t. x1 would be covered by t minus 1. And t minus 2, and so on and so forth.

But I don't have an optimum algorithm here. I just have my heuristic algorithm. And I'm just making a fairly vacuous statement, as I said, based on my definition, that the original problem can be covered using t. And therefore, a smaller problem, when I remove elements from it, could also be covered using t. OK? So far, so good?

So now, one of them covers what? What can I say about one of them? What principle can I use to make a claim about the number of elements that are covered by one of these t sets? Remember a principle from way back? 6042? My favorite principle? Flapping. Think flapping. Pigeon hole. Pigeon hole principle, right? See, you have to remember all of the material that you learned at MIT for the rest of your life. OK? You never know when it's going to be useful.

OK, so here you go, pigeon hole. My favorite principle. It's such a trivial principle. So one of them covers at least xk divided by t. I mean, why is this even a principle? Right? Elements. OK, so that's it, right? That's the observation. And, now that implies that an algorithm-- because it's going to pick the maximum of these, right? The algorithm is going to pick the maximum of these. So, it's going to pick a set of current size greater than or equal to xk divided by t. Otherwise, the algorithm would be incorrect. It's not doing what you told it to do. Right? It's got to pick the maximum.

So keep chugging here. And one real observation, and then the rest of it is algebra. So, what I can say is that for all k, xk plus 1, which is shrinking, is less than or equal to 1 minus 1 over t, xk. OK? That's the way I'm shrinking. Right? This is again, a fairly conservative statement. Because the fact of the matter is that I'm putting t in as a constant here, right? But t is actually, in effect, changing, obviously, halfway through the algorithm. I don't need t sets to cover the x-- whatever it is-- xk over 2 or whatever it is that I have. I need the t for x0.

Maybe I did a bad selection with my heuristic, and I still would need t, which is optimum, remember, for x1. But halfway through the algorithm after I picked a bunch of sets, I'm still

saying I need t, OK? Because I just need that for my proof. That's all I need for the proof. In CLRS, this actually varies and it turns into a harmonic series. We won't go there. OK? You can do the natural logarithm of n plus 1, a proof, just doing the simpler analysis. OK?

So you see what's happening here. That's my shrinkage. That's the recurrence, if you will, that tells me how my problem size is shrinking. And when do I end? What is my stopping point? What's my stopping point? Mathematically? When do we end this lecture? When you give me the answer. No, not quite. So I end when? I got nothing to cover, right? So when one of these things gets down to xk equals 0, right? When xk equals 0, I'm done.

I'm constantly taking stuff away. When xk equals 0, I'm going to be done. OK? And I'm going to move a little bit between discrete and continuous here. It's all going to be fine. But what I have is, if I just take that, I can turn this. This is a recurrence. I want to turn that into a series.

So I can say something like 1 minus 1 over t, raised to k, times n. And this is the cardinality of x, which is the cardinality of x0. So that's what I have up there. And that's essentially what happens. I constantly shrink as I go along. And I have a constant rate of shrinkage here. Which is the conservative part of it. So keep that in mind. But it doesn't matter from an analysis standpoint. OK?

So if you look at that, and you say, what happens here? Well, I can just say that this is less than or equal to e raised to minus-- you knew you were going to get an e, because you saw a natural algorithm here, right? And so, that's what we got.

And basically, that's it. You can do a little bit of algebra. I'll just write this out for you. But I won't really explain it. You're going to have Xk equals 0. You're done. The cost, of course, is k. Right? The cost is k, because you've selected k subsets. All right, so that's your cost, all right? So, when you get to the point, you're done. And the cost is k. So what you need is, you need to say that e raised to minus kt divided by n is strictly less than 1. Because that is effectively when you have strictly less than 1 element. It's discrete. So that means you have zero elements left to cover. That means you're done OK?

So that's your condition for stopping. So this done means that e raised to minus kt times n is strictly less than 1. And if you go do that, you'll get k over t, just about greater than natural logarithm of m(n). The algorithm terminates if we just do a little manipulation. And that implies that k over t less than or equal to natural logarithm of n plus 1 is valid. Right?

k over t is going to be less than or equal to natural logarithm of n plus 1. Because the instant it becomes greater, the algorithm terminates. Right? And that's how you got your proof over here. Because this is exactly what we want. k is our C from way back where it's the cost of our heuristic or the cost of our approximation. t is the optimum cost. That's what I defined it as. And this is a bound on k over t. All right? Cool. Any questions on this?

OK. So this approximation ratio gets worse for larger problems, just like this other approximation's algorithm that we didn't actually prove from the very first problem. That also got worse. We had a log k factor for that. And as your problem size increased, obviously the approximation factor increased. This is a little clearer as to what the increase looks like in relation to the original size of n. So it's just natural logarithm of n plus 1.

So, so far, we've done approximation algorithms, a couple of different varieties. We had a constant factor one, and then we had a row of n that actually had a dependence on n. Now let's move, and we'll do one last example on partition, which it turns out has a trivial constant factor approximation scheme. And this obvious thing, and we'll get to that in just a second.

But what is nice about partition is you can actually get a PTAS, right? Polynomial time approximation scheme, and FPTAS, fully polynomial time approximation scheme, that essentially give you with higher and higher run times. They're going to give you solutions that are closer and closer to optimal, right? If you want to do the FPTAS, we'll do the PTAS.

So partition is a trivial little problem to define. It's just you have a set, and you want to partition it into two sets such that they're not unbalanced. So your cost is the imbalance between the two sets. And you want to minimize that cost. You'd love to have two sets that are exactly the same weight. But if one of them is extremely unbalanced with respect to the other, then it's bad. Either way. So, here we go with partition.

Set S of n items with weights S1 through Sn, assume S1 greater than S2, Sn, without loss of generality. So this is just an ordering thing. I mean, obviously, there's some order. I'm not even claiming uniqueness here. I'm just saying just assume that this is the order. The analysis is much better if you do this-- if you make this assumption. And I want to partition into A and B to minimize max of sigma I belongs to A. Si sigma I belongs to B Si. And this is the weight of A. And this is the weight of B.

And so there's only so much you can do. If you have 2L equals sigma I equals 1 through n Si--

so I'm just calling that 2L-- the sum of all the weights. Then my optimum solution is what? What is the lower bound on the optimum solution? If it's 2L? What's the trivial lower bound? Just L, right? So I could have L here and L there. And if I had 2L here and 0 here, then oh! That's right. I want to minimize. Remember, don't-- maybe that's what threw you off. It threw me off right here. I see a max here, and I got a little worried.

But I want to minimize the maximum of these two quantities, OK? And so the best I could do is to keep them equal. And if I get L for both of them, that would minimize the maximum of those two quantities. If I had 2L and 0, then the maximum of those two quantities is 2L, and obviously I haven't done any minimization. So now you see why there's a trivial optimum solution is greater than or equal to L. Right?

And now you see why there's a trivial two approximation algorithm. Because the worst I could do is 2L, right? I could dump all of them on one side, constant time, and the other one is 0. And my cost is 2L. So I'm within a factor of 2 in this problem trivially. So, unfortunately that's not the end of the lecture, OK?

So, we've got to do better. I mean, clearly, there's more here. You'd like to get much closer. This is a different kind of problem. And it would be wonderful if we could get within epsilon. Right? I'm within 1%. How long does it take me? I'm within 0.01%, guaranteed. How long does it take me? Right? That's what an approximation scheme is, as opposed to just a plain algorithm.

So if you're actually going to talk about epsilon here, and we're just doing a PTAS. So we're going to see something that is not polynomial in 1 over epsilon. It's polynomial in n. But not polynomial in 1 over epsilon. But there's an FPTAS with this problem that you're not responsible for.

So this is going to be an interesting algorithm simply because we now have to do something with epsilon. It's going to have an extra input. It's not going to be the simple heuristic, where I'm going to do maximum degree or maximum number of elements or anything like that. I want to take this epsilon and actually do something with it. All right? So how does this work?

Well, basically what happens in PTAS's, or in a bunch of them, is that you essentially do an exponential amount of work given a particular epsilon to get a partial optimum solution. So you can think of epsilon as essentially being 1 divided by m plus 1, where m is some quantity. And as m grows, the complexity of your algorithm is going to grow. But obviously, as m grows,

you're getting a tighter and tighter epsilon. You're getting guaranteed closer and closer to your optimum.

And so we're going to have two phases here in this particular approximation scheme. The first phase is find an optimal partition, A prime, B prime, of S1 through Sm. And we're just going to assume that this exhaustive search, which looks at all possible subsets, and picks the best one. OK? And how many subsets are there for a set of size m? It's 2 raised to m.

So this is going to be an exponential order, 2 raised to m algorithm. OK? I'm just going to find the optimum partition through exhaustive search for m. Right? m is less than n. So I'm picking something that's a smaller problem. I'm going to seed this. So, the way this scheme works is, I'm seeding my actual algorithm-- my actual heuristic-- with an initial partial solution. And depending on how much work I do to create the seed, I'm going to end up having higher complexity. And obviously that's a function of small m, or 1 over epsilon.

And so this takes-- this optimal takes order to raise to m time. And you can think of that as order 2 raised to 1 over epsilon. And so that's why it's a PTAS, and not an FPTAS. OK? So this is PTAS.

What else do we need to do? Well, I don't actually have a solution yet. Because if m is really small-- and by the way, m can be 0 as well. Right? Epsilon would then be at 2 approximation, 1 divided by-- this would be one half. And so 1 over epsilon is 2. So then you've got your trivial algorithm that we had, the 2 approximation scheme. So that makes sense?

So the second phase is you're going to start with your seed corresponding to A and B. You're going to set them to A prime and B prime. And what I'm going to do is, for I equals m plus 1 to n, if w(a) less than or equal to w(b), A equals A union I-- running out of room-- else B equals B union I. OK?

So it's not that hard to see, hopefully. All I'm doing here is, I'm just going in a very greedy way. I got my initial A prime and B prime. I set them to A and B. And I say, oh, I got this element here. Which one is bigger? This one is bigger? I'm going to put the element over here. And then I look at it again. I got another element. Which one is bigger? And I go this way, that way. That's pretty much it.

So, again, all of these algorithms are really straightforward. The interesting part's-- the fun part's-- in showing the approximation guarantee. So we're good here? Yup? All right.

So back to analysis. One last time. So, let's see. We want to show that a prox partition-- ah, you know how to spell partition-- is PTAS. I guess I don't, but you do. Let's assume that w(a) is greater than or equal to w(b) to end with. Right? So I'm just saying, at the end here, I'm just marking the one that was larger that came out of the max as A. Right? Without loss of generality. Just to make things easier, I don't want to keep interchanging things. So our approximation ratio is w(a) divided by L, right? w(a) could at best be L if I got a perfect partition, perfectly balanced. But it could be a little bit more. And that's my approximation ratio, OK?

So I need to now figure out how the approximation ratio reflects on the run time, and is related to m and, therefore, epsilon. All right? So what I'm going to do here is, I'm going to look at a point in time where I have A and B-- and remember that w(a) is defined to be greater than w(b). But here, I'm looking at some point in time, which is not necessarily at the very end here.

It could be. But you can think of this as being, I'm just going to say B or intermediate B. And this won't matter too much. But I'm just being a little bit of a stickler here. I'll explain why I said that in just a minute. But the point is, I have a situation where I know that w(a) is greater than w(b) or greater than or equal to w(b) because I assumed it. That's how I marked it. And I'm going to look at k is the last element added to A. It's been added.

Now this could have been added in the first phase or the second phase. It's quite possible that for a given m, that if it's large, for example, that A prime that you end up with is your A to begin with here, and that you never execute this statement OK? It's quite possible, right? You got your initial seed and never added to it. And that was it. Because your m was large, for example, right?

So what I'm saying here is, k is the last element added to A. OK? So there's clearly a last element. I'm just marking that. And we know that A is greater than or equal to B. Now it may be true that if I'm looking at the snapshot when just after I add the k-th element to A, I may not be done yet, in terms of I still have a few elements. And I may be adding elements to B. But regardless, given my definition, I know that the rate of B is less than the rate of A. Because even though the last element overall may be added to B, w(b) is less than w(a), and I'm only looking at the last element added to A here. OK?

So why all of this skulduggery? Well, there's a method here to the madness. We're going to analyze what possibly happens in the first phase and the second phase, and get our

approximation ratio. Shouldn't take too long.

So there's two cases here that we need to analyze. The first one is easy. The second one is a little more involved. I'm going to now assume that k was the last element and was added in the first phase, OK? If k is added to A, what can I say? If k was added in the first phase to A, and that's the last element added throughout the algorithm, by the time you get a partition, what can you say? What can you say about the solution? What strong statement can you make about the solution? What's the only interesting strong statement that you can make about a solution?

So I got to A. Remember what the first phase is. What's the first phase? Well, it's optimal, right? So, after that, what happened to A? Well it didn't change. Right? So, what you got is optimum. Right? Because w(a) was defined to be greater than or equal to w(b). w(a) was optimum for the smaller problem, whatever m was. You never added anything else to it. So you're done. It's optimum.

So in this case, your approximation ratio is 1 because you got the optimum solution, right? So if k is added to A in the first place, this means A equals A prime. We have an optimal partition. Since we can't do better than w(a) prime when we have n greater than m items. And we know w(a) prime is optimal for the m items. OK?

So that's cool. That's good. So we got an approximation ratio of 1 there. And remember that, this is not taking overall exponential time necessarily. It's just a case where I've picked some arbitrary m, and it happens to be the case that A equals A prime at the end of the algorithm. So I am taking exponential time in m-- m as in Mary-- but I'm not taking exponential time in n-- right? n as in Nancy.

So the second part is where the approximation ratio comes in. k is added to A in the second phase. So here, what we're going to do is, we're going to say we know w(a) minus Sk is less than or equal to w(b). This is the second phase we're talking about here. The only reason Sk was added to A was because you decided that A was the side that was smaller, right, or perhaps equal. So that's the reason you added into it. So you know that w(a) minus Sk is less than or equal to w(b) at that time.

So think of A and B here as being variables that are obviously changing, right? But what I'm saying here is, even if you look-- this is the algorithm where A and B, you constantly look at them and decide which way to go. But if you look at the last step, then you look at the final

values. Then you could certainly make the statement for those final values, that the w(a), which is the resultant value, minus Sk, should have been less than or equal to w(b). And you had a smaller w(a) to begin with. And you added Sk to it. And that happened in the second phase. OK? So this is why k was added. This is why k was added to A.

I want to be a little careful here given that we're overloading A and B without trying to point out what each of these statements actually means. And ask questions if you're confused. I know that w(a) minus Sk is less than or equal to 2L minus w(a). That's just a substitution. Because w(a) plus w(b) equals 2L.

And then, last little trick, again it's algebra. Nothing profound here. We have our assumption that we ordered these things. So you had S1 through-- Sn, excuse me. The whole thing was ordered. And so we can say that S1, S2, all the way to Sm, is greater than or equal to Sk. We are actually doing this in order. We are taking the bigger elements and then deciding where they go. So we sorted those things initially. And so what we end up with when we look at Sk, we have taken care of values prior to Sk that are all greater than or equal to Sk. Right? That's, again, using our initial assumption.

So what that means is 2L is greater than or equal to m plus 1 Sk since k is greater than m. Again, this is not particularly tight. Because m could be really pretty small in relation to n. But I do know that I can make the statement that since the values are decreasing, that 2L, which is the sum of all of those, is greater than or equal to m plus 1 times Sk, regardless of what m is. Right? And so, that's pretty much it. Once you do that, you have your approximation ratio. Let's leave that up there because that's the algorithm. Finish this off with a little algebra.

So w(a) divided by L is less than or equal to L plus Sk divided by 2, divided by L. I'm basically substituting. I have this and I have that. And I'm playing around with it. And I got 1 plus Sk divided by-- 1 plus Sk divided by 2L, which is-- now this I could say is equal. That's simply equal. This, I have a less than or equal to. And then I can go less than or equal to 1 plus Sk divided by m plus 1 Sk. And then I got 1 plus 1 divided by m plus 1, which is, of course, 1 plus epsilon.

So all I did here was use this fact and essentially relate 2L to Sk. And once I could relate 2L to Sk, substituting for Sk in here, I ended up with the approximation ratio that I want, w(a) over L, is L plus Sk divided by 2. That simply comes from here. And plug that in, divided by L. And then I have Sk divided by 2L. And then this part here, 2L is going to get substituted by m plus

1 Sk and voila. I'm over here. OK?

So the story behind this particular problem was it was in the quiz. And Eric, I guess took the quiz. Did you actually take the quiz? Or edit the quiz? And he says, this problem is impossible. It was a problem. He said, this problem is impossible. I had to Google it to find out the answer. Or something like that. And I said, well, I'm going to give the answer in lecture, right? So there you go. So remember this. Write it down. Four points for coming to lecture today. See you guys next time.