SRINIVAS DEVADAS: So welcome to 6046. My name is Srinivas Devadas. I'm a professor of computer science. This is my 27th year at MIT.

I'm teaching this class with great course staff, with co-lecturers, Eric Demaine over here and Nancy Lynch, who's over there, and a whole bunch of TAs, who you will meet through the term. We just signed up our last TA yesterday, so at this point, even I don't know their names. But we hope to have a great semester. I'm very excited to be teaching this class with Eric and Nancy.

I recognize some of you folks from 006 from a year ago, so hello again, and from other classes. And so let's get started. I mentioned 006. 006 is a prerequisite for this class, so if by chance you've skipped a class-- MIT or EECS has allowed you to skip that-- make sure you check in with us to see that you are ready for 6046 because we will assume that you know the 6006 material.

And by that, I mean basic material on that data structures, classical algorithms like sorting, algorithms for dynamic programming, or algorithms that use dynamic programming I should say, algorithms for shortest paths, et cetera. 6046 itself, we're going to run this course pretty much off the Stellar website in the sense that that'll be our one-stop shop for getting everything including lecture handouts, problem sets-- turning in your problem sets, et cetera. And I should mention that this course is being taped for OpenCourseWare, and while it'll take a little bit of time for the videos to be put online, we hope to do that perhaps in clumps before the quizzes that you will have as we have to have in our class.

So let me just say a couple more things about logistics, and then we get started with technical content. As I mentioned, we're going to be running this course off Stellar. Please sign up for recitations section by going to the stellar website and choosing a section that works for your schedule. Sections go from 10:00 AM all the way to 3:00 I think, and we've placed a limit on the number of students per section. We wanted the sections to be manageable in size, but there's plenty of room for everybody, and the schedule flexibility should allows you to choose a section pretty easily.

We have a course information document and an objectives document on the website. That has a lot of details on the grading policy, the collaboration policy, et cetera. Please read it very carefully from the first page all the way to the end.

And I will mention one thing that you should be careful about, which is that while problem sets are only 30% of the grade, we do require you to attempt the problems. And there's actually a penalty associated with not attempting problems and not tuning problem sets in that is way more than 30%, so keep that in mind, and please read the collaboration policy as well as the grading policy, carefully. And feel free to ask us questions. You can ask us questions anonymously through Piazza, or you can certainly send us email. All the information is on Stellar.

So that's all I really had to say about course logistics. Let me tell you a little bit about how the content of this course is structured. We have several modules, and Eric, Nancy, and I will be in charge of each of these different modules as the term goes.

Our very first module is going to start really next time. Today is really an overview lecture. But it's a module on divide and conquer, and you learned about this divide and conquer paradigm in 006 or equivalent classes. It's breaking of a problem into smaller problems and getting efficiency that way. Merge sort is a classic algorithm that follows the divide and conquer paradigm.

If you're going to take it to a new level. And I guess that's sort of the team of 046. Take the material in 006 and raise the stakes a little bit-- raise the level of sophistication-- and you'll see things like fast Fourier transform. Finding an algorithm for a convex hull, we'll do that next time. That uses the divide and conquer paradigm.

We're going to do a ton of optimization. Divide and conquer can obviously be used for search and also for optimization. In particular, we'll look at strategies corresponding to greedy algorithms, Dijkstra, which hopefully you remember the shortest path algorithm from 006 is an example of a greedy algorithm. We'll see a bunch of other examples, and we'll look at one today.

And dynamic programming, it's a wonderful algorithmic hammer that you can apply to a wide variety of problems, certainly to shortest paths as well. We'll look at it in many different contexts.

And then really quickly network flow, which is a problem that's associated with-- here's a network. This capacity is associated with the network. The capacities could respond to the width of the roads in a highway system or the number of lanes, the amount of traffic that can go through. How do I maximize the set of commodities, or the amount of commodities that I can push through the network? That it turns out is, again, a problem that has many different applications, so it's really a collection of problems.

You're going to spend some time, a little bit today, but a little more than in 6006, talking about intractability. So a lot of algorithms that we're going to talk about are efficient in the sense that they're polynomial time solvable. And first, polynomial time solvable doesn't imply efficiency in the practical sense, so if you have an n raised to 8 algorithm, it's polynomial time. But really, it's not something that you can use on real world problems where n is relatively large, but generally in a theoretical computer science class, we'll think about tractable problems as being those that have polynomial time algorithms that can solve them exactly or optimally. But intractability then corresponds to problems that, at the moment, we don't know of a polynomial time algorithm to solve them, and the best algorithms we have take worst case exponential time.

And so the question is, what happens with those problems? And we'll look at things like approximation algorithms that can get us, in the case of optimization problems, get us to within a certain fraction of optimal, guaranteed, and run in polynomial time. So you can't get the absolute best, but you can get within 10% or we can get within a factor of 2. That may be enough for a particular instance of a problem or a set of instances of a problem.

And what we do a bunch of advanced topics. I think we have distributed algorithms plan. Nancy works in that area, and we'll also talk about cryptography. There's a deep connection between number theory algorithms and cryptography that towards end of the lecture, or, I should say, towards the end of the course, I will look at a little more closely.

So much for overview, let's get started with today's lecture for real. And here's the theme of today's lecture. I talked a bit about tractability and intractability. And what is fascinating about algorithms is that you might see a problem that has a fairly obvious polynomial time solution or a linear time solution, then you change it ever so slightly, and the linear time algorithm doesn't work. Maybe you can find a cubic algorithm.

And then you change it a little more, and you end up with something that you can't find a polynomial time algorithm for. You can't prove that the polynomial time algorithm or polynomial monomial time algorithm gives you the optimal solution in all cases.

And then you go off into complexity theory. You maybe discover that, or show that this problem is NP-complete, and now you're in the intractability domain. So very small changes in problem statements can end up with very different situations from a standpoint of algorithm complexity. And so that's really what I want to point out to you in some detail with a concrete example.

So I want to get a little bit pedantic here with respect to intractability and tractability. You've seen, I think, these terms before in the one lecture in 006, but we'll go over this in some detail today and more later on in the semester. But for now, let's recall some basic terminology associated with tractability and intractability or complexity theory, broadly speaking.

Capital P is a class of problems solvable in polynomial time. And think of that as big O, n raised to k for some constant k. Now you can have long factors in there, but once you put a big O in there, you're good. You can always say, order n, even if it's a logarithmic problem, and big O lets you be sloppy like that.

And there are many examples of polynomial time algorithms, of course, for interesting problems like shortest paths. So the shortest path problem is order V square, where V is the number of vertices in the graph. There's algorithms for that. You can do a little bit better if you use fancier data structure, but that's an example.

NP is another class of problems that's very interesting. This is the class of problems that whose solution is verifiable in polynomial time. So an example of a problem in NP that is not known to be NP is the Hamiltonian cycle problem. And the Hamiltonian cycle problem corresponds to given a directed graph, find a simple cycle. So you can repeat vertices, but you need the simple cycle to contain each vertex in V.

And determining whether a given cycle is a Hamiltonian cycle or not is simple. You just traverse the cycle. Make sure that you've touched all the vertices exactly once, and you're done. Clearly doable in polynomial time.

So therefore, Hamiltonian cycle is an NP, but determining whether a graph has a Hamiltonian cycle or not is a hard problem. And in particular, the notion of NP completeness is something that defines the level of intractability for NP. The NP complete problems are the hardest problems in NP, and Hamiltonian cycle is one of them.

If you can solve any NP complete problem in polynomial time, you can solve all problems in NP in polynomial time. So that's what I meant by saying that NP complete problems are, in some sense, the hardest problems an NP because solving one of them gives you everything. So the definition of NP completeness is that the problem is in NP and is as hard-- an informal definition-- as any problem in NP. And so Hamiltonian cycle is an NP complete problem. Satisfiability is an NP complete problem, and there's a whole bunch of them.

So going back to our theme here, what I want to show you is how for an interval scheduling problem, that I'll define in a couple of minutes, how we move from linear time, therefore P, to something that's still in P. But it's a little more complicated if I change the constraints of a problem a little bit. And finally, if I add more constraints to the problem, generalize it-- and you can think of it as adding constraints or generalizing the problem-- you get small changes to something that becomes NP complete.

So this is something that algorithm designers have to keep in mind because before you go off and try to design an algorithm for a problem you like to know where in the spectrum your problem resides. And in order to do that, you need to understand algorithm paradigms obviously and be able to apply them, but you also have to understand reductions where you can try and translate one problem to another. And if you can do that, and the first problem is known to be hard, then you can make arguments about the hardness of your problem. So these are the kinds of things that we'll touch upon today, the analysis of an algorithm, the design of an algorithm, and also the complexity analysis of an algorithm, which may not just be an asymptotic-- well, this is order n cubed or order n square but more in the realm of NP completeness as well.

So so much for context, let's dive into our interval scheduling problem, which is something that you can imagine doing for classes, tasks, a particular schedule during a day, life in general. And in the general setting, we have resources and requests, and we're going to have a single resource for our first version of the problem. And our requests are going to be 1 through n, and we can think of these requests as requiring time corresponding to the resource.

So the request is for the resource, and you want time on the resource. Maybe it's computation time. Maybe it's your time. It could be anything.

Each of these requests responds to an interval of time, and that's where the name comes from. si is start time time. fi is the finish time, and we're going to say si is strictly less than fi. So I didn't put less than or equal to there because I want these requests to be non-null, non-zero, so otherwise they're uninteresting. And we're going to have a start time, and we're going to have an end time, and they're not equal.

So that's the first part of the specification of the problem and then the second part, which is intuitive is that two requests-- we have a single resource here remember-- i and j are considered to be compatible, which means you can satisfy both of these requests. They're compatible. Incompatible requests, you can't satisfy with your single resource simultaneously-- Provided they don't overlap.

And an overlapping condition might be that fi is less than or equal to sg, or fj less than or equal to si. So again, I put a less than or equal to here, which is important to spend a minute on. What I'm saying here in this context is that I really have open-ended intervals on the right-hand side corresponding to the fi's.

So pictorially, you could look at it this way. Let's say I have intervals like this. So this is interval number 1. That's interval number 2. Right here I have s of 1, f of 1 out here, s of 2 out here, and f of 2 out here.

So this is f of 1 for that and s of 2 for this. I'm allowing s of 2 and f of 1 to be exactly equal, and I still agree that these two are compatible requests. So this is-- I guess it's terminology. It's our definition of compatibility.

So you can imagine now an optimization problem that is associated with interval scheduling where, in a different example, I have this interval corresponding to s1 and f1. I might have a different interval here corresponding to 2, then corresponding to 3. And then maybe I've got 4 here, 5, and 6. So those are my six intervals corresponding to my input.

I have a single resource. I'm just drawn out in a two-dimensional form. There's six different requests that I have, the six different intervals. Intervals and requests are synonyms.

And my goal here-- and it's kind of obvious in this example-- is to select a compatible subset of requests, or intervals, that is of maximum size. And I'd like to do this efficiently. So we'll always consider efficiency here, but in terms of the specification of the problem as opposed to a requirement on the complexity of the algorithm, I want maximum size for this subset. So as I showed you, or I mentioned earlier, in this case, it is clear from the drawing that I put up there that the maximum size for that six requests example that I have is three. So that's the set up.

Now we're going to spend the next few minutes talking about a greedy strategy for solving this particular problem. If you don't know of it, the greedy strategy is going to always produce the maximum size or not. In fact, it depends on the particular greedy heuristic, the selection heuristic that a greedy algorithm uses. So that's going to be important, and we'll take a look-- and hopefully you can suggest some-- at a few different greedy heuristics.

But my claim, overall claim, that I'm going to have to spend a bunch of time here justifying and eventually proving is that we can solve this problem using a greedy algorithm. Now what is a greedy algorithm? You've seen some examples.

As the name implies, it's something that's myopic. It doesn't look ahead. It looks to maximize the very first thing that you couldn't maximize.

It says-- traffic is a good example-- don't let anybody cut in front of you. You've got some room up there. Get up there. Generally, people are greedy when it comes to getting to work, trying to minimize the time and, in this case, on the time that they spend on the road.

But we've had other examples. For example, when you look at interval scheduling, you might say, I'm going to pick the smallest request. And I'm going to pick the smallest request first, and I'm going to try and collect together as many requests as possible. And if the requests are small in the sense that si and fi, for the two requests, are close to each other, then maybe that's the best strategy. So that's an example of a greedy strategy for our particular example.

But just to give you a slightly better definition of greedy than what I've said so far, a greedy algorithm is a myopic algorithm that does two things. It processes the input one piece at a time with no apparent look ahead. So what happens is that greedy algorithms are typically quite efficient. What you end up doing is looking at a small part of the problem instance and deciding what to do. Once you've done that, then you're in a situation where the problem has gotten a little bit simpler because you've already solved part of it, and then you move on.

So what would a template for a greedy algorithm look like for our interval scheduling problem? Here's a template that probably puts it all together and gives you a good sense of what I mean by greedy, at least in this context. So before we even get into particulars of selection strategies, let me give you a template for greedy interval scheduling.

So step 1, use a simple rule to select a request. And once you do that, if you selected a particular request-- let's say you selected 1. What happens now once you've selected 1? Well, you're done.

You can't select 2. You can't select 3. You can't select 4. You can't select 5. You can't select 6.

So if you have selected 1 in this case, you're done, but we have to codify that in a step here. And what that means is that we have to reject all requests that are incompatible with i. And at this point, because we've rejected a bunch of requests, our problem got smaller. And so you now have a smaller problem, and you just repeat-- go back to step 1-- until all requests are processed.

All right, so that's a classical template for a greedy algorithm. You just go through these really simple steps. And the reason this is a template is because I haven't specified a particular rule, and so it's not quite an algorithm that you can code yet because we need a rule.

So with all of that context, let me ask you. What is a rule that you think would work well for an interval scheduling problem?

Yeah, go ahead.

AUDIENCE: Select one with the earliest finish time.

SRINIVAS DEVADAS: Select one with the earliest finish time. All right, well, I did not want that answer.

[LAUGHTER]

But now that you've given me the answer, I have to do something about this. So I want a different answer, so we'll go to a different person. But before I do that, let me reward you for that answer I did not want with a limited edition 6046 Frisbee, OK?

[APPLAUSE]

You need to stand up because I don't want to take people's heads off.

[LAUGHTER]

Yeah sorry. All right, so here you go. All right? Good.

[APPLAUSE]

So people do cookies and candy. I think Eric, Nancy and I are cooler.

[LAUGHTER]

So we do Frisbees. All right, good, so the fact of the matter was that this class was scheduled for 9:30 to 11:00 on Tuesdays and Thursdays. That's when we decided to do Frisbees. And then it got shifted over to 11:00 to 12:30, but then we bought all these Frisbees, so we said, whatever. It's not like we could use all of them

All right, good. So I don't like that answer, and I want a different one. Give me another one. Yeah, go ahead.

AUDIENCE: Just carry it through in numerical order.

SRINIVAS DEVADAS: I'm sorry?

AUDIENCE: Just carry it through in numerical order.

SRINIVAS DEVADAS: Carry it through in numerical order. Is that going to work? And what's an example that it didn't work? The one right there, right?

Should I get her a Frisbee? We should. I'm going to be generous at the beginning. You can just--

But that's an answer I liked. Yeah, there you go. So entering through a numeric order isn't going to work. This is a great example right there. Give me another one.

[LAUGHTER]

There are no Frisbees right here. Over there, yeah?

AUDIENCE: Try the one with the shortest time.

SRINIVAS DEVADAS: Ah, try the one with the shortest time. OK, so the shortest time in this case might be this one. The shortest time might be this one, and, hey, that might work in this case because you pick this one, which is the shortest, or maybe it's five, which is the shortest. Either way, you could get 2, 5, and 6, looking at this picture, seems to work. Maybe 4, 5, and 6 if you pick 5 first, et cetera, right?

I'll give you a Frisbee if you can take that same algorithm and give me a counter example.

AUDIENCE: Let's say you have two requests which don't overlap, and then there's--

SRINIVAS DEVADAS: --there's one right in the middle, exactly right. Yep, so let's see. What do I do? Oh, here.

So pictorially, a really you can look at this, and you can actually figure out whether your heuristic works or not. But this, I think, what you were thinking about. There you go, right? So you get one.

So that clearly doesn't work. So this one was smallest, doesn't work. The suggestion here was a numeric. It doesn't work.

Here's one that might actually work. For each request, find the number of incompatible requests. So you've got a request. You can always intersect the other requests with it and decide whether the second request is compatible or not, and you do this for every other request. And you can collect together numbers associated with how many incompatible requests a particular request has, and you say, well, let me use that as a heuristic.

So each request, find number of incompatible requests and select the one with the minimum number of incompatibles. So just to be clear, in this case, you would not select 1 because clearly 1 is incompatible with every other request, so that clearly is not numeric order. In this case, you would not select this one because it's incompatible with this one and that one. So you'd select that one which has the minimum number of incompatibles. So you think this is going to produce the correct answer, the maximum answer, in every possible case?

AUDIENCE: No.

SRINIVAS DEVADAS: No, who said, no? Well, anybody who said, no, should give me a counter example. Yeah, go for it.

AUDIENCE: If the one that it selects has mutually incompatible collection of intervals with which it's compatible.

SRINIVAS DEVADAS: Right, so that's a good thought. We'll have to [INAUDIBLE] that. And I think this particular example, that's exactly what you said, which just instantiates your notion of mutual incompatibility.

So here's an example where I have something. It's a little more complicated. As you can see, this is a pretty good heuristic. It's not perfect as you can see from this example, where I have something like this.

So if you look at this, what I have here is I have just a bunch of requests which have-- this is incompatible with this and that and these two, so clearly a lot of incompatibilities for these, a lot incompatibilities for these. Which is the minimum? The one in here, but what happens if you select that? Well, clearly you don't get this solution, which is optimal. The one on top, so this is a bad selection. And so this doesn't work either, OK? There you go.

So as it turns out, the reason I didn't like that first answer was it was correct.

[LAUGHTER]

It's actually a beautiful heuristic. Earliest finish time is a heuristic that is-- well, it's not really a heuristic in the sense that if you use that selection rule, then it works in every case. In every case, it's going to get to you the maximum number, OK?

Earliest finished time so what does that mean? Well, it just means that I'm going to just scan the f of i's associated with the list of requests that I have, and I'm going to pick the one that is minimum. Minimum f of i means earliest finish time.

Now you can just step back, and I'm not going to do this for every diagram that I have up here, but look at every example that I've put up. Apply the selection rule associated with earliest finish time, and you'll see that it works and gets you the maximum number. For example, over here, this has the earliest finish time. Not this, not this, it's over here.

So you pick that, and then you use the greedy algorithm step 2 to eliminate all of the intervals that are incompatible, so these go away. Once this goes away, this one has the earliest finish time and so on and so forth. So this is something that you can prove through examples. That's not really a good notion when you can prove to yourself using examples.

And this is where I guess is the essence of 6046, to some extent 006 comes into play. We will have to prove beyond a shadow of a doubt using mathematical rigor that the earliest finish time selection rule always gives us the maximum number of requests, and we're going to do that. It's going to take us a little bit of time, but that's the kind of thing you will be expected to do and you'll see a lot of in 046. OK? So everyone buy earliest finish time? Yep, go ahead.

AUDIENCE: So what if we consider the simple path example of there's one request for the whole block, and there's one small request that it mentioned earlier.

SRINIVAS DEVADAS: Well, you'll get one for-- if there's any two requests, your maximum number is 1. So you pick-- it doesn't matter-- it's not like you want efficiency of your resource. In this particular case, we will look at cases where you might have an extra consideration associated with your problem which changes the problem that says, I want my resource to be maximally utilized. If you do that, then this doesn't work.

And that's exactly-- it's a great question you asked. But I did say that we were going to look at the team here, which I don't have anymore, but of how problems change algorithms. And so that's a problem change. You've got a question.

AUDIENCE: I have a counter example. You have three intervals that don't conflict with one another. You have one interval that conflicts with the first two and ends earlier than the first one.

SRINIVAS DEVADAS: OK, so are you claiming that there's going to be a counter example to earliest finish time?

AUDIENCE: Yes.

SRINIVAS DEVADAS: All right, I would write it down on a sheet of paper. And get me a concrete example, and you can just slide it by. And if you get that before I finished my proof, you win, OK?

[LAUGHTER]

So I would write it down. Just write it down, so good. All right, so this is a contest now.

[LAUGHTER]

All right, so we are going to try and prove this. So there's many ways you could prove things, and I mean prove things properly. And I don't know if you've read the old 6042 proof techniques that are invalid, which is things like prove by intimidation, proof because the lecturer said so, you know, things like that.

This is going to be a classical proof technique. It's going to be a proof by induction. We're going to go into it in some detail.

Later on in the term we are going to put out sketches of proofs. We are going to be skipping steps in lecture that are obvious or maybe not so obvious, but if you paid attention, then you can infer the middle step, for example. And so will be doing proof sketches, and proof sketches are not sketchy proofs.

[LAUGHTER]

So keep that in mind. But this particular proof that we're going to do, I'm going to put in all the steps because it's our first one. And so what we're going to do here is prove a claim, and the claim is simply that-- whoops, this is not writing very well. What is going on here? OK.

[LAUGHTER]

Back to the white. Given a list of intervals l, our greedy algorithm with earliest finish time produces k star intervals where k star is minimal. So that's what we like to prove.

AUDIENCE: [INAUDIBLE].

SRINIVAS DEVADAS: Sorry, what happened?

AUDIENCE: [INAUDIBLE]

SRINIVAS DEVADAS: Oh, right. Good point. Maximum.

What we're going to do is prove this by induction, and it's going to be induction on k star. And so the base case is almost always with induction proofs trivial, and it's similar here as well. And in the base case, if you have a single interval in your list, then obviously that's a trivial example. But what I'm saying here for the base is slightly different. It says that the optimal solution has a single interval, right?

And so now if your problem has one interval or two intervals or three intervals, you can always pick one, and it's clearly going to be a valid schedule because you don't have to check compatibility. And so the base case is trivial even in the case where you're not talking just of intervals that have cardinality 1, but the optimal schedule has cardinality 1. So that's a trivial case. So the hard work, of course, in the induction proofs is assuming the hypothesis and proving the n-plus-1, or in this case, the k-star-plus-1 case. And that's what we'll have to work on.

So let's say that the claim holds for k star, and we are given a list of intervals who's optimal schedule is k star plus 1. It has k-star-plus-1 intervals in the optimal schedule, so L may be some large number, capital L, maybe in the hundreds. And k star, there may be 10 of what have you. They're different. I want to point that out.

So our optimal schedule, we're going to write out as this, s star. So usually if you use star for optimal in 046 and it's got k-star-plus-1 entries, and those entries look like sf pairs-- so I'm going to using the subscript j1 through j k star plus 1 to denote these intervals. So the first one is sj1, fj1. That's an interval that's been selected and is part of our optimal solution.

And then you keep going and we have sj k star plus 1 comma fj k star plus 1. So no getting away from subscripts here in 046 So that's what we have in terms of this is what the optimal schedule is. It's got size k star.

Of course, what we have to show is that the greedy algorithm with the earliest finish time is going to produce something that is k star plus one in size. And so that's the hard part. We can assume the inductive hypothesis, and we'll have to do that. But there's a couple of steps in between.

So let's say that what we have is s1 through k is what the greedy algorithm produces with the earliest finish time. So I'm going to write that down sik fik, so notice I have k here, and k and k star, at this point, are not comparable. I'm just making a statement that I took this particular problem that has k star plus 1 in terms of its optimal solution size, and for that problem, I have k intervals that are produced by the earliest finish time greedy heuristic. And so that's why the subscripts here are different. I have i1 here and ik, and then over here I have the j's, and so these intervals are different.

If I look at f of i plus f of i1, and if I look f of j1, what can I say about f of i1 and f of j1? Is there a relationship between f of i1 and f of j1? They're equal? Do they have to be equal? Yeah?

AUDIENCE: Less or equal to.

SRINIVAS DEVADAS: Less than equal to, exactly right, so they're less than equal to. It's possible that you might end up with a different optimal solution that doesn't use the earliest finish time. We think earliest finish time is optimal at this point. We haven't proven it yet, but it's quite possible that you may have other solutions that are optimal that aren't necessarily the ones that earliest finish time gives you. So that's really why the less than or equal to is important here.

Now what I'm going to do is create a schedule, s star star, that essentially is going to be taking s star and pulling out the first interval from s star and substituting it with the first interval from my greedy algorithms schedule. So I'm just going to replace that, and so s star star is si1 fj1. And then I'm going to be going back to sj2 fj2 because I'm going back to s star and all the other ones are coming from s star. So they're going to be sj k star plus 1 comma fj k star plus 1. So I just did a little substitution there associated with the optimal solution, and I stuck in part of the greedy algorithm solution, in fact, the very first schedule.

AUDIENCE: So the 1 should be i1.

SRINIVAS DEVADAS: Oh, this should be-- i1,

AUDIENCE: Right?

SRINIVAS DEVADAS: i1, thank you. Yep, good. So we've got a couple of things to do, a couple of observations to make, and we're going to be able do prove some relationship between k and k star that is going to give us the proof for our claim.

So clearly, s star is also optimal. All I've done is taken one interval out and replaced it with another one. It hasn't changed the size.

It goes up to k star plus 1, so s double star is also optimal. s star is optimal. s double star is optimal.

Now I'm going to define L prime as the set of intervals with s of i greater than or equal to f of i1. So what is L prime? Well, L prime is what happens in the second step of the greedy algorithm, where in the second step of the greedy algorithm, once I've selected this particular interval and I've pull it in, I have to reject all of the other intervals that are incompatible with this one. So I'm going to have to take only those intervals for which s of i is greater than or equal to f of i1 because those are the ones that are compatible. So that's what L prime is.

And I'm going to be able to say that since s double star is optimal for L, s double star 2 to k star plus 1 is optimal for L prime. So I'm making a statement about this optimal solution. I know that's optimal, and basically what I'm saying is subsets of the optimal solution are going to have to be optimal because if that's not the case, I could always substitute something better and shrink the size of the k star plus 1 optimal solution, which obviously would be a contradiction. So s double star is optimal for L, and therefore s double star 2 through k star plus 1 is optimal for L prime.

Everybody buy that? Yep? Good. And so what this means, of course, is that the optimal schedule for L prime has k star size.

And I'm starting with 2. I've taken away 1. So now I have L prime, which is a smaller problem. Now you see where the proof is headed, if you didn't already.

I have a smaller problem, which is L prime. Clearly, it's got fewer requests, and I have constructed an optimal schedule for that problem by pulling it out of the original optimal schedule I was given. And that size of that optimal schedule is k star. And now I get to invoke my inductive hypothesis because my inductive hypothesis says that this claim that I have up there holds for any set of problems that have an optimal schedule of size k star. That's what the inductive hypothesis gives me.

And so by the inductive hypothesis, when I run the greedy algorithm on L prime, I'm going to get sk schedule of size k star. Now can you tell me, based on what you see on the board, by construction, when I run the greedy algorithm, what am I getting on L star? By construction, when I run the greedy algorithm on L prime-- there's too many superscripts here-- when I run the greedy algorithm on L prime, what do I get? Someone? Yeah?

AUDIENCE: We get s of i sub 2, s of i sub 2 interval.

SRINIVAS DEVADAS: Exactly right, I get everything from the second thing here all the way to the end because that's exactly what the greedy algorithm does. Remember, the greedy algorithm picked si1 fi1, and then rejected all requests that are incompatible and then move on. When you rejected all requests that are incompatible here, you got exactly L prime. And by construction, the greedy algorithm should have given me all the way from si2 too sik. Thank you.

So by construction, the greedy on L prime gives s2 to k, right? And what is the size of this? 2 to k gives me a size of k minus 1. This is k minus 1.

So if I put these two things together, what is the next step? I have the inductive hypothesis giving me a fact. I have the construction giving me something. Now I can relate k and k star. What's the relationship?

k star is equal to k minus 1, right? Do people see that? So size k star or just k minus 1. So what that means is given that s2k is a size k star, it means that s1k is of size k star plus 1, which is exactly what I want. That's optimal because I said in the beginning that we had k star plus 1 in our inductive hypothesis this case as being the optimal solution.

So this last step here is all you need to argue now that s of 1k, going back up here, this is optimal because k equals k star plus 1. There you go, so that's the kind of argument that you have to make in order to prove something like this in 046. And what you'll see in your problem sets, including the one that's going to come out on Thursday, is that different problem that you have to have proof for a greedy algorithm for. I forget exactly what technique you'll have used there, perhaps induction, perhaps contradiction. And these are the kinds of things that get you to the point where you've analyzed the correctness of algorithms, not just the fact that you're getting a valid schedule, but you're getting a valid maximum schedule in terms of the maximum number of requests.

Any questions about this? Do people buy the proof? Yep. Good.

So that was greedy for a particular problem. I told you that the team of our lecture here was changing the problem and getting different algorithms that had different complexities. So let's go ahead and do that. So the rest of this lecture, we'll just take a look at different kinds of problems and talk a little more superficially about what the problem complexities are.

And so one thing that might come to mind is that you'd like to do weighted interval scheduling. And what happens here is each request has weight wi, and what we want to do is schedule a subset of requests with maximum weight. So previously, it was just all weights were 1, so maximum cardinality was what we wanted. But now we want to schedule a subset of requests with maximum weight.

Someone give me an argument as to whether the greedy algorithm earliest finish time first is optimal for this weighted case, or give me a counter example. Yep, go ahead.

AUDIENCE: Oh, well, you know like your first example you have your first weight of the first interval, it took the whole time, [INAUDIBLE] would have three smaller ones? Well, if the weight of the first one was 20 and then--

SRINIVAS DEVADAS: Exactly, exactly right. All right, I owe you one too. So here you go.

So it's a fairly trivial example. All you do is w equals 1, w equals 1, w equals 3, so there you go. So clearly, the earliest finish time would pick this one and then this one, which is fine. You get two of these, but this was important. This is, I don't know, sleep party, 6046.

[LAUGHTER]

So there you go. So the weight it is, we should make that infinity. Most important thing in the world at least for the next six months.

So how does this work now? So it turns out that the greedy strategy, the template that I had, fails. There's nothing that exists on this planet that, at least I know of, where you can have a simple rule and use that template to get the optimum solution, in this case, maximum weight solution, for every problem instance, so that template just fails. What other programming paradigm do you think would be useful here? Yeah, go ahead.

AUDIENCE: DP.

SRINIVAS DEVADAS: DP, right. So do you want to take a stab at a potential DP solution here?

AUDIENCE: Yeah, so either include it in your [INAUDIBLE] or discard it and then continue with set of other intervals.

SRINIVAS DEVADAS: Yeah, that's a perfect divide and conquer. And then when you include it, what do you have to do?

AUDIENCE: Eliminate all conflicting intervals.

SRINIVAS DEVADAS: Right, how many subproblems do you think there are. I want to make you own your Frisbee, right?

[LAUGHTER]

AUDIENCE: 2 to the power of the number of intervals you have because--

SRINIVAS DEVADAS: Well, that's a number of subsets that you have. So you have n intervals, then you have two [INAUDIBLE] subsets.

AUDIENCE: Yeah.

SRINIVAS DEVADAS: But remember, you want to go-- you want to be smarter than that, right? You want to be a little bit smarter than that. So here, you get a Frisbee anyway.

[LAUGHTER]

No, not anyway, here you go. Right. So anybody else?

So what I want to use is dynamic programming. We've established that. I want to use dynamic programming. And the dynamic programming-- you have some experience with that in 006-- the name of the game is to figure out what the subproblems are.

The subproblems are kind of going to look like a collection of requests. I mean, there's no two things about it. They're going to be a collection of requests, and so the challenge here is not to go to the 2 raised to n, because 2 raised to n is bad if you want efficiency. So we have to have a polynomial number of subproblems. So someone who hasn't answered yet, go ahead.

AUDIENCE: [INAUDIBLE] so [INAUDIBLE] subset [INAUDIBLE] So from interval i to interval j [INAUDIBLE].

SRINIVAS DEVADAS: So you're looking at every pair of i's and j's, and, well, not all of them are going to be valid. There won't be intervals associated with that, but that's a reasonable start. Someone else, someone who hasn't answered? Yeah, back there.

AUDIENCE: You could go the best term to start to some even point, and so there'd n of those.

SRINIVAS DEVADAS: Ah, best from the start to any given point. All right, well, you got close, Michael. There you go. You need to stand up.

Ew, bad throw. That's a bad throw. I've got to practice.

OK, so as you can see with dynamic programming, the challenge is to figure out what the subproblems are. The fact of the matter is that there's going to be many different possible algorithms that are all DP for this weighted problem. There's at least two interesting ones. We're going to do a simple one, which is based on the answer that the gentleman here just gave. But it turns out you can be a little smarter than that, and most likely you'll hear the smarter way in the section, but let's do the simple one because that's all I have time here for.

And the key is to define the subproblems, and then once you do that, the actual recursion ends up being a fairly straightforward and intuitive step. So let's look at dynamic programming, one particular way of solving this problem, using the DP paradigm. So what I'm going to do is define subproblems R star, so R is the total number of requests that we have, and the subproblems are going to correspond to-- I'm going to request j belonging to R such that-- oh, I'm sorry. This is R of x-- such that sj is greater than or equal to x. So what I'm doing here is, given a particular x, I can always shrink the number of requests that I have based on this rule.

And then you might ask, what is x? And now you can apply the same subsetting property by choosing the x's to be the finishing times of all of the other requests. All right, so x equals f of i. So what this means is-- then I put f of i over here-- it means all of the requests that come after the i-th request finished our part of R of fi. So R of fi would simply be requests later than f of i.

And there's something subtle here that I want to point out, which is R of fi is not the set of requests that are compatible with the i-th request. It's not exactly that. It's the set of requests that are later than f of i.

So keep that in mind because what happens here is we're going to solve this problem step by step. We're going to construct the dynamic programming solution essentially by picking a request, just like in the greedy case, and then taking the request that comes after that. So we're going to pick an early request, and then we're going to subset the solution, pick the next one just like we did with the greedy. And so the subproblems that we will actually solve potentially bottom up if we are doing recursion are going to correspond to a set of requests that come later than the particular subset that we're looking at, which is defined by a particular interval. So requests that are later than f of i, not necessarily all of the requests that are compatible with the i-th request.

And so if you do that, then the number of subproblems here are small n, where n is the number of requests. So if n is the number of requests in the original problem, the number of sub problems equals n because all I do is plug-in an appropriate i, find f of i for it, and generate the R of f of i for each of those. So there's going to be n of those subproblems.

And we're going to solve each subproblem once and then memoize. And so the work that we have to do is the basic rule corresponding to the complexity of a DP, which is number of subproblems times the time to solve each subproblem, or a single subproblem, and this assumes order 1 for lookups. So you can think of the recursive calls as being order 1 because your assuming you're doing memoization. So I haven't really told you anything here that you haven't seen in 006 and likely applied a bunch of times. Over here, we've just defined what our subproblems are for our particular DP, and we argued that the number of subproblems that are associated with this particular choice of subproblems corresponds to n if you have n requests in the original problem instance that you've given.

So the last thing that we have to do here to solve our DP is, of course, to write our recursion and to convince ourselves that this actually all works out, and let's do that. And so what we have here is our DP guessing. And we're going to try each request i as a plausible first request, and so that's where this works.

You might be thinking, boy, I mean, this R of fi looks a little strange. Why doesn't it include all of the requests that are compatible with the i-th request? I mean, I'm somehow shrinking my subsequent problem size if I'm ignoring some requests that are earlier that really should be part of-- or are part of the compatible set, but they're not part of the R of fi set. And so some of you may be thinking that, well, the reason this is going to work out is because we are going to construct our solution, as I said before, from the beginning to the end.

So we're going to try each request as a plausible first request. So even though this request might be in our chart all the way to the right, it might have a huge weight, and so I'm going to have to try that out as my first selection. And when I try that out as my first selection, then the definition of my subproblem says that this will work. I only have to look at the request that comes later than that because the ones that came the earlier, I've tried them out too. So that's something that you need to keep in mind in order to argue correctness of this recursion that I'm going to write out now.

And so the recursion, and I have opt R, what is the first thing that I'm going to have on the right-hand side of this recursive formulation? What mathematical construct am I going to have to do here? And you see something like guessing and seeing something like try each request as a possible first, what mathematical construct am I going to have to put up here?

AUDIENCE: Max.

SRINIVAS DEVADAS: Max, who said max? No one wants to take credit for max? It's max, right? So I'm going to have max 1 less than equal to i less than or equal to n.

And I'm going to-- does someone want to tell me what the rest of this looks like? Someone else? A couple Frisbees left, guys.

[LAUGHTER]

What does the rest of this look like? Yep?

AUDIENCE: 1 plus the optimal R f of--

SRINIVAS DEVADAS: Not 1, just what kind of problem do we have here? It's not 1 anymore.

AUDIENCE: Oh--

SRINIVAS DEVADAS: The weight.

AUDIENCE: Right.

SRINIVAS DEVADAS: The weight, yep, so Wi plus the optimal R fi. OK, so we got Wi plus optimum of R of fi. And you said "1," close enough. If it was 1, you'd use greedy. And so that's why we were in that Wi mode, and we end up getting this here.

So that's it. You try every request as a possible first. Obviously, you pick that request so it's part of your weight in terms of the weight for your solution.

When you do that, because it was the first request, you get to prune the set of requests that come later corresponding to R of fi that you see here. And then you go ahead and simply find the optimum for a smaller problem, clearly has fewer requests. And as long as you maximize over the set of guesses that you've taken, and there's n guesses up at the top level. Obviously in the lower levels, you're going to have fewer requests in your R of fi's, and you'll have fewer durations of the max, but it's n at the top level.

So one last question, what is the complexity of what we see here?

AUDIENCE: n square.

SRINIVAS DEVADAS: n square, and the reason it's n square is you simply use-- you can be really mechanical about this-- you say, if this was order 1, I'm doing a max over n items. And therefore, that's order n time to solve one subproblem. And since I have n subproblems, I get n times order in, which is order n squared.

So the last thing I'll do-- and I just have one more minute-- is give you a sense of a small change to interval scheduling that puts us in that NP complete domain. So so far, we've just done two problems. There's many others.

We did interval scheduling. There was greedy linear time. Weighted interval scheduling is order n squared according to this particular DP formulation. It turns out there's a smarter DP formulation that runs an order n log n time that you'll hear about in section on Friday, but it's still polynomial time.

Let's make one reasonable change to this, which is to say that we may have multiple resources, and they may be non identical. So it turns out everything that we've done kind of extrapolates very well to identical machines, even though there's many identical machines. But if you have non-identical machines, what that means is you have resources or machines that have different types. So maybe your machines are T1 to Tm. And it's essentially a situation where you say, this particular task can only be run on this machine or this other machines, some subset of machines.

So you can still have a weight of 1 for all requests, but you have something like A of i belonging subset of T is a set of machines that i runs on. OK, that's it. That's the change we make. Q of i is going to be specified for each of the i's.

So you could even have two machines. And you could say, here's a set of requests that could run on both machines. Here's a set that only runs on the first machine, and here's another set that runs on the second machine. That's a simple example of this generalization.

If you do this, this problem has been shown to be NP complete. And by that I mean, NP complete problems are decision problems. And so you say, can some specific number k less than and requests be scheduled. This decision problem is NP complete.

And so what happens when you have NP complete problems? Well, we're going to have a little module in the class that deals with intractability. We're going to look at cases where we could apply approximation algorithms, and maybe in the case of the optimization problem, if the optimum for this is k star, I will say that we can get within 10% of k star. The other way is to just deal with intractability by hoping that your exponential time algorithm runs in a reasonable amount of time for common cases.

So in the worst case, you might end up taking a long time. But you just sort of back off after an hour and take what you get from the operative algorithm. But in many cases, the algorithm might actually complete, and they give you the optimum solution.

So done here. Make sure to sign up for a recitation section. And see you guys next time.