LING REN: All right, welcome everyone. This is 6046 presentation. Just make sure you're in the right place. My name is Ling Ren. I'm one of the 10 TAs for this class. We do have a second TA for this section. I think he is not here right now, but, basically, [INAUDIBLE] and I will be switching every other week.

I want to remind you, just a heads up, this section is recorded for OCW purpose. But I think he's only recording us, the TAs. You're not in a camera. All right, so, the only purpose we are here is to help you learn this very interesting, and also very useful, class.

So don't hesitate to ask any questions or give us any feedback, like whether I'm going too fast or too slow, whether you want us to cover something that's not in the posted schedule, or just anything we can help. All right, so let's get started.

The two lectures in this week, in the first week, focus on divide and conquer. It is a class of algorithm that usually involves recursion in the algorithm description. And Professor Devadas worked through several algorithms, including weighted interval scheduling and a bunch of others, and he left several open problems.

So we will answer those open questions in this section, and we'll also show you a new algorithm, and analyze a bunch of other algorithms. So just to remind everyone what weighted interval scheduling is. In this problem, we are given a bunch of requests, each with a start time and a finish time. And our goal is to find a subset of them that are compatible, meaning they do not overlap, and that have a largest combined weight. OK, are we clear? Everyone clear about that?

So an easier case is when the problem is unweighted, meaning that every task has the same weight. In that case, we can just solve it using our [? Grady ?] algorithm. But when the problem becomes weighted, we have to use dynamic programming, or recursion, and Srini introduced a simple one, a basic version, in a class. Can someone remind us how that algorithm works? Any volunteers?

AUDIENCE: [INAUDIBLE]

LING REN: Can you speak louder?

AUDIENCE: [INAUDIBLE] so we find that it's not conflicting [INAUDIBLE].

LING REN: So what's your name?

AUDIENCE: [INAUDIBLE]

LING REN: Yeah, I think the version you described is for the unweighted case. In unweighted case, we just schedule the 1 with the earliest finish time, and, then, we remove all the incompatible ones, and we keep going, right? That solved the unweighted version. If it's the weighted version, we need to use recursion. And remember, we break the problem into many sub-problems, and each one can potentially be an optimal solution. Does anyone remember that? Care to give it a try?

AUDIENCE: Sure. So we break it into subproblems. We took the best solution from a certain point, and then we calculate that subproblem starting at all the different finish times. [INAUDIBLE].

LING REN: Great. What is your name?

AUDIENCE: [? Amin. ?]

LING REN: [? Amin? ?] OK. [? Amin ?] said, let's just try every one as our potential first request. So if we request the j as our first, we get its weight. And then, we're going to solve a subproblem. So let me call the original problem. We did interval scheduling with all the incoming requests.

Now, we choose j, request as our first. Now, we are left with a subproblem that starts after request to j finishes. So I'll write that as Rj, Where I define Rj to be the set of requests where their stop time is later than the finish time of the j-th request.

OK, so just to repeat, we choose a request, has the potential to be the first request, and then, we look at all the requests that start after it, and solve a subproblem of that case. Then, we take a max of all the candidate we have, and that's going to give us the optimal solution. Any question about this algorithm?

So this algorithm runs in n square time. Now, we're going to try to optimize that, and come up with a better algorithm. So in order to improve anything, we first want to identify the inefficiency in this algorithm. So which part in the algorithm do you think is inefficient, or silly, unnecessary? Go ahead.

AUDIENCE: So it's inefficient to look through every previous subproblem, when we're trying to find maximum [INAUDIBLE].

LING REN: I was saying that we don't need to go through every of this case.

AUDIENCE: Yeah, we shouldn't Well, we should be able to efficiently query for the right one.

LING REN: OK, I think you are definitely correct. So let me just go through what this algorithm does, and it will be more clear. So what this means is, I'll choose the first request. I'll request 1 as my first request. Then, I'm going to consider only the requests that start after 1 finishes, right? That only leaves request a 5. Potentially some other's already been drawn there. That's my first candidate in that max.

My second candidate is actually request 2 as my first request. Then, I have to remove request 1, because it starts too early, and then, I'm left with all the remaining requests. I'll solve that subproblem. That's candidate 2.

Candidate 3, I chose request 3 as my first request, and then I have to remove 1 and 2, because they start too early. Oh, 4 as well. It also starts too early, right? Before 3 finishes. Everyone following that? So we're left with the remaining request. Is it more clear now? Go ahead.

AUDIENCE: So candidate, if you start with 3, that's actually a subproblem of 2. [INAUDIBLE].

LING REN: Great point. What's your name?

AUDIENCE: Andrew.

LING REN: Andrew said, we can potentially be solving many repeated subproblems, like this. We definitely don't want to do that. And that's actually the core idea, the one crisp idea of dynamic programming. Andrew, can you tell me what's the definition of dynamic programming? You remember? Anyone remember that? Go ahead.

AUDIENCE: Just memorize subproblems, then look them up.

LING REN: Exactly. So dynamic programming says, we will break a problem into subproblems, and subproblems into even more subproblems, but whenever we solve one, we should memorize, or just remember its result and store it somewhere. And if you need it again, we'll just retrieve it, without resolving the problem. That's definitely a great point.

So we can analyze the complexity of this algorithm later, because I want to touch on this more efficient algorithm first. And we'll see that, even after Andrew's optimization, its runtime is n square. So without that observation, if we are solving repeated subproblems, these will be a lot worse than that. Yeah, go ahead.

AUDIENCE: You can also trim ones that get to the same place, so you don't need to explore two paths.

LING REN: Sorry, say that again?

AUDIENCE: You don't need to explore two paths that [INAUDIBLE] both exploring 3. Once they're both looking at 3, then you only need to do the ones more efficient to that point.

LING REN: OK, cool. Go ahead. Oh, same thing?

AUDIENCE: Well, I have another one as well. So I think to get better than n squared, we need to make the observation that it's always fine to start a subproblem later. So if you've decided you're taking a certain sequence of intervals from your first interval, and then you want to see how to compute from there, it's always valid to start later than, maybe, you had to. So that means that if we can efficiently starting at any particular point, query for the maximum of any of the subproblems starting after that point, then we can [INAUDIBLE].

LING REN: OK, great. I think we are on the same page. So when I describe the steps of this algorithm, remember this third candidate. I choose this as my first, right? It makes zero sense, because, if I do that, I might as well put in 2, as well. Doesn't hurt, right? Everyone get that?

So the idea is that we shouldn't try every possible W request as my first. Some requests are just better, more suited to be the first request. And how we're going to do that? So, apparently, 1 can potentially be the first request. 2 can also be. But it doesn't make any sense for any requests come after that, because there's and earlier request.

So the efficient algorithm, let's first sort them by their start time. We're going to consider the request that comes early first. So I have my entire problem, here. Now, I'm going to ask a question. Should I include request 1 in my solution or not? That's only two cases.

So if I do not select 1 in my solution, what subproblem am I left with? Any idea? If I decide I will not include 1 in my solution.

AUDIENCE: [INAUDIBLE].

LING REN: Yeah, exactly. There's no conflicts anywhere. I'll solve a subproblem from 2 to n. If I do decide to put my request 1 in a solution, I get this weight. So what's the subproblem I'm left with? Yeah, in this example, it's five. Correct. But more generally--

AUDIENCE: [INAUDIBLE] So every request that starts afterward.

LING REN: Exactly, every request that starts after 1 finishes. Now, suddenly, we are not breaking the original problem into n subproblems. We only have two subproblems. So let me draw a recursion tree, which is a powerful tool in analyzing these sort of things.

So we start with our original problem, from 1 to n, and we have two subproblems, 2 to n and R1 to n. This one, we'll also break them into subproblems. So what is this one? OK, I have my sub problem 2 to n, and I need to further reapply the same trick

AUDIENCE: [INAUDIBLE]. The max of Wi has to be--

LING REN: Oh, sorry. These are the two cases. Either I do not schedule my first, or I schedule it, and this is my first request. Now, here, I'm left with this problem. Either I do not schedule it, or I schedule it as my first request. So what is this one?

AUDIENCE: Probably be 3 to the n.

LING REN: 3 to n, because now I'm asking the same question for 2. And, here, I'll have R2 to n, so on and so forth. Now, let me point out the big difference for this version, and for the basic version. So I'm starting with the request that starts first. If I do not do that, say, if here, I am asking the question for 5, do I schedule 5 as my first or not?

Then, what happens? Then, these two branches do not cover all the cases, because I can potentially schedule it, but not as my first, in the optimal solution, if I'm asking a question for a random request. However, if I start with the first request, first meaning it starts earliest, it will either be not scheduled, or it will be scheduled, and as the first. Because it cannot be the second request in my solution.

Any questions about this algorithm? OK, now, this is the algorithm. Let's analyze its complexity. So what's the overall complexity. when we go all the way down, solve the entire original problem? Any guesses? What do you think? What do you think? Go ahead.

AUDIENCE: It should be log n, because [INAUDIBLE] sort the intervals, and we're going to need to, for every interval, find the next interval that starts after it, and do that for [INAUDIBLE].

LING REN: Cool. So the first step is sorting, which is n log n. Now, after I sort everything, the question I need to answer is, how many unique subproblems are there, in this entire recursion tree? So I'm not going to solve the same problem twice. How many unique problems exist, here? n of them, right? Just this left branch. All the others will be one of these.

So I can start from the bottom of the tree and work my way up, and when I want to take this step, I'll look up the result of this in one of the subproblems I've already solved. So, actually, the recursion itself is only O of n.

AUDIENCE: I missed when you said, what is the sorting referring to?

LING REN: OK, so we need to start with the request that starts first, right? We need to decide whether we schedule it or not. And, then, we need to do the same thing for this request 2. It's the start earliest request in this subset. We always need to do that for all the subproblems. So the overall complexity is n log n.

But if we only focus on this recursion step, our improvement is actually larger than that, because it went from n square to O of n. So why is the original algorithm n squared? I think it also has only n unique subproblems, right? So do you agree that the original algorithm is n squared, or do you think it's also O of n? Just focus on the recursion step.

AUDIENCE: Didn't the original algorithm solve some subproblems up to n times? So that's why it's still of n squared, because it solved the same thing?

LING REN: But assuming I do not do that, assuming, whenever I solve it once, I store the result somewhere, and I directly get it. Assuming I do that, what's the complexity? Go ahead.

AUDIENCE: O of n.

LING REN: You think it's O of n? So anyone think it's n squared? Because I think Srini said it's n squared. Go ahead.

AUDIENCE: I think the original algorithm isn't [INAUDIBLE] squared, because we're still doing [INAUDIBLE] for every subproblem [INAUDIBLE].

LING REN: Exactly right. So, here, whenever we go up one step, I'm doing constant number of work. Just comparing two numbers and taking the max. However, in the original algorithm, which is here, whenever I want to go one step up, there are n branches of the tree.

So my total amount of work is 1 plus 2 plus 3 plus n, right? Every step becomes harder when I go up. And this is n squared. Any questions about that? OK, this is so much for our weighted interval scheduling. Now, I'm going to transition into the next topic. So any question on the side, in general, for the scheduling problem? Do you have a question? No?

OK, now, let's turn to a second topic of this section, which is Strassen algorithm. Strassen algorithm is an efficient algorithm for matrix multiplication, and matrix multiplication is a really useful primitive. It has applications in almost every area I can think of. Circuit simulation, climate simulation, and physics, basically everything.

Now, I actually had some experience with matrix multiplication, because my undergrad research was improving matrix algorithms. And, actually, many matrix algorithms, including inversion, solving equations, they all use multiplication as a primitive. So it actually comes down to improving matrix multiplication.

And I tried very hard to just optimize this basic matrix multiplication. We'll take a row, you take a column, and then you get your answer for this spot. Everyone's familiar with that, right? I tried very hard, but it's still 100 x slower than the best algorithm out there.

So I finally look it up, and I was completely mind-blown when I know that matrix algorithm complexity is not n cubic. It's actually smaller than that. Is this a surprise to you? Anyone expect that before? And the technique the more efficient algorithm uses is, exactly, Strassen algorithm.

Now that we're talking about divide and conquer, you can guess it must be a divide and conquer algorithm. And, so, does anyone have an idea how to divide the original problem? Anyone want to give it a try? So are you familiar with tiled matrix multiplication, or blocked matrix multiplication? OK, can you tell us what that is?

AUDIENCE: Yeah. You can break it into [INAUDIBLE].

LING REN: OK, so say this is our A and B, and we want C. We can break each matrix into four parts. I'll call this A11, A12, A21, A22. B11, B12, B21, B22. Same thing for C.

Now, I would like someone to tell me, what is C11, in this case? Yes.

AUDIENCE: A11 [INAUDIBLE].

LING REN: A12, B21, yep. And C12 is-- Just speak up, don't be shy. This is also my first section ever, in my life, teaching a recitation. I'm more nervous than you guys. What is C12?

AUDIENCE: A11, B12?

LING REN: A11, B12. A12, B22, right? So the rule is the same as before. Matrix multiplication. Compute this, we take this row, this column, and it gives us this.

AUDIENCE: So the first one [INAUDIBLE].

LING REN: Right, thank you. A12, B21. OK. And, same thing, C21, we're going to take this row and this column. A21 B11 plus A22 B21. C11 is A21 B12 plus A22 B22. OK. And everyone understands this? OK, great.

So, now, we've broken up the original problem into several subproblems. We only need to do matrix multiplication here eight times. And each of this matrix is half in size. If the original algorithm is n cubic, now each sub problem is half n cubic. Then we have eight of them, so the complexity is still n cubic. No improvement at all.

So, actually, to be more precise, we can further break up these matrices into smaller blocks. So, to be precise, its complexity should be given by a recursion. Eight subproblems, each one half the size, plus-- can anyone tell me? What's the merging complexity, once I get all this? [INAUDIBLE]. Go ahead.

AUDIENCE: Is it just constant because you're adding?

LING REN: Yeah, because I'm adding. But is it constant? I'm adding these two matrices.

AUDIENCE: [INAUDIBLE].

LING REN: Pardon?

AUDIENCE: For each level it can base case?

LING REN: Yeah, for base case, it's, of course, constant. OK, so maybe I'm not clear about this. What's the subproblem, in this case? It's this multiplication operation, OK? And, so, these are the eight subproblems I will solve. After solving them, I need to add them together. And adding two matrices, and each is size ha;f of n. The complexity is--

AUDIENCE: n squared 2 over 2?

LING REN: Yeah, it's n squared. Basically, n squared. So, to get a precise complexity, we should solve this recursion, but it will end up being the same thing as this intuition, n cubic. OK, so now, this is the magic.

So Strassen, in 1969, came up with this algorithm.

AUDIENCE: [INAUDIBLE].

LING REN: Each of these is a half n by half n matrix.

AUDIENCE: [INAUDIBLE].

LING REN: So Strassen came up with this algorithm. He somehow defined M1 through M7, seven matrices, in this way. I can't provide any intuition, because I didn't come up with this. And, somehow, with those seven matrices, he can reconstruct, he can compute all the four submatrices in C.

And it's not very just interesting to check it, because the algorithm is definitely correct. But let's just do one of them. OK, how about this one. So C21 is M2 plus M4. So M2 plus M4. So M2 will have A12 B11, A22 B11. So M4, there's A22 minus B11, so that cancels out. So we're left with A21 B11 plus A22 B21. That's the correct answer.

So this, I guess, is a very clever algorithm. You have to work in that area for 10 years to come up with this so that's not our concern. Our goal is to analyze this algorithm. What's the complexity of it? So does anyone understand this recursion? Can someone tell me, what's the recursion for this part, for this Strassen algorithm?

We have the original problem, and we have some-- go ahead.

AUDIENCE: So since each of the M 1 through 7 only require one multiplication, you'll need to solve seven subproblems, so 7T n over 2 plus O n squared.

LING REN: That's absolutely correct. Everyone gets this? So what Strassen did, is he came up with the seven matrices. Each one requires only one multiplication. So we have seven subproblems, instead of eight, and that's going to give us a benefit, an improvement.

So the question now becomes, how do I solve this recursion? Given this recursion, how do I know its complexity? And same question there. Anyone want to give it a try? So that's going to be covered in a third topic, which is Master Theorem.

So Master Theorem does exactly that. All it does is, given the recursion, a and T of n over b, plus some work for merging, where a and b are constants, it directly tell you what T of n is, in some cases. So I'll first write the formula.

Master Theorem actually has three cases. The first case, fn, is order n raised to c, where c is less than log b of a. Then, Master Theorem says, it's complexity is theta log b of a.

Second case, fn is theta nc log K, where c is, log b is equal to b of a, then it's complexity is n raised to c log K plus 1 n. You don't necessarily have to copy them, because you just find them anywhere.

And the third case, you can imagine, it's the only remaining case, which is, fn is large, it's omega n raised to c, where c is greater than log b of a. Then, Master Theorem says, its complexity, the complexity of Tn, is theta fn.

So, intuitively, if fn is not too much work, then it's basically this recursion, what recursion gives you. If fn dominates, fn is the biggest component, then Tn is, roughly, on the order of fn, and there's a case in the middle. Now, let's see why that is the case. I'll only cover one case, here.

So, again, we are going to draw a recursion tree, because that is very useful in all the recursion problems. So we start with a problem of size n, and we break them into problem size n over b.

So on and so forth. What's the size of this subproblem? So that recursion represents a class of recursive algorithm. Every time it breaks the problem, it reduced the problem size by a factor of b, so what do I have here? Go ahead.

AUDIENCE: [INAUDIBLE]

LING REN: n over b squared. So on and so forth. So what is a, in this graph?

AUDIENCE: 3.

LING REN: 3, think so? 3. a is just a branching factor of this tree. I keep going. Finally, I will reach my base case. So my next question is, after how many levels of recursion will I reach a base case of size 1? Go ahead.

AUDIENCE: Log b of [INAUDIBLE]?

LING REN: Log b of-- OK, so, here is n over b, n over b square, next one n over b cubic, so on and so forth. So, say, this last level it Tth level, than the problems size is n over-- we raise to T, and we want that to be a constant, so what is T?

AUDIENCE: Log b on n.

LING REN: Log b of n. Now, this is the recursion tree, and we have that fn among the emerging work to do. So, here, we have to do fn work, to merge these a results to the solution of our problem, n. Then, we have f- what's the emerging work for this level, for this part of the tree? This is my problem size.

AUDIENCE: [INAUDIBLE]

LING REN: n over b, right. And we have a of them. OK, so on and so forth. So, then, we know what is Tn. Let's just enumerate all the work we have to do. So on the first level, we have to do fn. OK, on second level, af n of b-- sorry, n over b.

And what's the next level? We have how many subproblems? Speak louder.

AUDIENCE: a squared.

LING REN: a square subproblems, and each of them is n over b squared. And, finally, I reach my last level. They are all base cases, so I have a raised to T of them, because I defined T to be my depth of the tree. And each of them is T of 1.

OK, so that's Tn. I'm not entirely happy with this formula, because I have this beautiful pattern, here, except for that last guy. It's add one a and divide one b, blah, blah, blah. So I'm going to change this T into f. Can I do that?

Because it's [INAUDIBLE] is the same, right? T1 is a constant, f1 is also constant. Then, I get my beautiful form, where it's a sum from i equals 0 to T. What's in the sum? Go ahead.

AUDIENCE: [INAUDIBLE].

LING REN: a raised to i, f of n over b1. Everyone gets that? Now, you can roughly see why we have three cases. So let me deal with the first case. The first case says, fn is order n raised to c.

What does that mean? It means this guy here is sigma a raised to n, then this is what's in my f raised to c. There should be order, here, but everything has order before it, so I just omit that. So it actually should be this, OK?

So this, because n raised to c is actually independent of the sum, I can pull it out. And what am I left with? Is that correct? Now, this is a sum of geometric sequence.

We know how to solve that, but we need to check whether this ratio is greater or larger than 1, or if it's equal to 1. And what is this ratio? The case tells us. So c is less than log b of a. That means b raised to c is less than b raised to this guy, which is a, right? So we know our denominator is smaller than or numerator.

So this is an increasing sequence, right? So what we have is n raised to c, then that thing raised to t minus 1 divided by this thing minus 1. But they are all constants. Are everyone familiar with this formula. of geometric sequence? OK, so that's what we have.

Next, we have t equals log b of n. That means b raised to t is n, correct? So, then, b raised to t is n, then we have raised to c, they cancel out. What do we have? I want to make sure everyone's following.

AUDIENCE: is it a over t?

LING REN: A raised to t. No questions?

AUDIENCE: Can you do that stuff again?

LING REN: OK, let me do that again. It's actually a raised to t, and then n over bt raised to c, right?

AUDIENCE: How did you get from the line above that?

LING REN: This one to here?

AUDIENCE: Yeah.

LING REN: Oh, it's the sum of geometric sequence, so if I have 1 plus q plus q squared, all the way to qt, it's qt plus 1, I guess-- or t, I don't remember very well-- minus 1, then q minus 1. I guess this should be t plus 1.

So this is what we're doing. So this is our q minus 1, and divided by q minus 1, they are all constants, so I don't care about them. So a raised plus this thing raised to c, but we said that b raised to t is equal to n, so we're just left with a raised to t.

And what is t? t is log b of n. I can write it as log a of n over log a of b. Everyone familiar with that? That means log a of n, log b of a. This is when I flip them. So this, what is that?

AUDIENCE: n.

LING REN: That is n. OK, we're done. Are we? Not exactly, because I have an order here, right? So everything is ordered. If you only care about order, big O, then it's fine. But that theorem says theta, so you have to prove it the other way, that it's no less than that.

I'm not going to do that. It's not very hard. Next, I'm going to apply this theorem to the two problems we left here. So let's apply Master Theorem to this recurrence. I think you are still looking at that side.

So what is the a, b, c for this? a is 8, b is 2, right? And c is 2. Log b of a is 3, so that's which case of the Master Theorem? So the theorem says it should be n raised to log b of a, which is 3, OK?

Now, what we have here, can I remind someone to do that for me? Want to give it a try? Go ahead.

AUDIENCE: OK, so we have a equals 7, b equals 2, and c equals 2, like before. And, now, we want to see whether c equals 2 is less than log b of a, which is log 2 of 7. Pretty sure that's still the case. So we should just get n to the log 2 of 7.

LING REN: Yeah, exactly. So, yeah. Yeah, thank you. let's give him a round of applause. So log 2 of 7 is definitely greater than 2. Why? Because log 2 of 4 is 2, right? So this happens to be n raised to 2.80, and many other digits, but it's less than n cubic.

And, just for knowledge purpose, this is no longer the best. It was the best, when it was proposed, and, well, researchers in that area have got it down to n raised to 2.35. I think 2.37, first, then 2.35. I'm not following the literature very closely, so maybe it's 2.34 now.

So I should have one more thing to cover, but I think we are running out of time. Sorry about that. I can post it. So the last thing we should do is, remember, we have this medium finding algorithm, where we have a recurrence, which is, I think, 10 over 5, some ceiling, and then plus this [INAUDIBLE] n plus theta n.

And we want to solve this recursion, but we cannot apply Master Theorem. Apparently, it's not the right form. So when Master Theorem doesn't apply, we have to study it case by case. Let me see if I have time to do that. OK, I think I probably do.

To solve that case, first, can someone tell me, what's the definition of theta? We have to go back to the definition to solve that. What does theta even mean? So if I say fn is theta n, what do I really mean? Go ahead.

AUDIENCE: It's tightly bounded, so you can move the [INAUDIBLE] to either side.

LING REN: So it means I can find some K1 and K2 such that this holds when n gets sufficiently large. OK, so, now, we're going to do induction. Assuming, for all the small n less than capital N, my Tn is bounded by this K2 and K1 thing.

Then, my next step is, T of this capital N would be bounded. I'll do the right side first. It will be bounded by K2 n5 ceiling, plus K2. [INAUDIBLE] in a second term, n plus another theta n.

So, we know, that means it's bounded by some other number. I'll say A2, then. That's the definition of theta n. Then, I want this to be-- sorry, all of them should be capital N. Capital N, Capital N. I want this to be smaller than K2 capital N.

So let me redo this first step. This is roughly 5 of K2 plus 7 over 10 K2, plus A2 of n, plus a bunch of constants that I don't care. I want it to be smaller than K2 of n.

Can I reach that? Of course I can, right? If I select a K2 to be greater than all we have here, what is this? This is 9 over 10 K2 plus A2, right? So if I select the K2 to be greater than 10 times A2-- is everyone following that? When n is sufficiently large, Tn should be bounded by A2 n. That's the induction.

I am assuming, when n is smaller than capital N, I have solved them. So I can use these two, and I solve the next step. So there's the other side, which is very similar. I'm not going to go through that.

All right, so that's all for today. And just to quickly recap, we went through the weighted interval scheduling, and the Strassen algorithm, Master Theorem and applying Master Theorem, and that case study of a new recursion. OK, thanks, everyone, for coming.