AMARTYA SHANKHA BISWAS: Feel free to populate the front row. I'm not that scary. So today, we're going to look at more greedy algorithms. So I think you went over Kruskal's algorithm and how you do the sorting in the lecture.

So going back to make change from last recitation, so this is sort of a variant on that. So instead of discrete coins, we now have continuous coins, in the sense so the analogy here is, let's say, you have N metals, and each of the metals has some value given by Ci dollars per kilogram, or whatever units you prefer. And you want to achieve some value T. You want to give someone T dollars worth of metal.

And you want to do this while minimizing-- oh, so I should mention this. ki is the weight of every metal that you will give to the person. So you're taking ki of metal i, and you are going to-- and you have to ensure, so basically, you have to ensure that some summation of ki Ci over all i is equal to T. And in doing so you want to minimize the summation over all ki. So does that make sense?

So you have a bunch of metals. Some of them are more expensive than others. And you want to measure them out and give someone a certain fixed value. So anyone have any ideas how to do this? Should be-- should be the first thing that comes to mind.

So you have much of metals-- some of them with certain costs. And you're trying to create a value T. So which metal would you want to pick? So it should seem intuitive that if you want to minimize the weight of the metal, you would want to pick the-- have the most expensive one for weight.

So let's start by sort by Ci. And we want to sort it in decreasing order. Does make sense? So if you have the most expensive metal, you want to use as much of that as you can, so that your weight is minimized.

So once you sort by Ci, so let's say, you have your costs right now are-- let's call this one C1, C2, up to Cn. And these are in sorted order. So it's increasing this way. So you now take your

value T, and you look at T by C1. And that is the amount of weight you would need to generate C.

So you look at how much you have here. So the amount of metal-- so a constraint I forgot to mention, you are given a limited amount of every metal. OK, that's-- it's not that trivial. So you have-- let's mention that. So you have-- is that used? No, it's not-- amount.

Does that make more sense? So you look at T over Ci. And if T over Ci is greater than W of i, then you just use the amount you need to construct Wi, and you're done. Otherwise, you use all of Ci.

So if it's less than Wi, in that case, you-- sorry, other way around. If it's greater than Wi, you use all of it. And then you move on to the next one, the next one, and so on. So that seems pretty intuitive.

Let's actually do a formal proof of that. So how you go about proving this is that-- so let's say-- so it's what we call the current base method. So basically what you have is, let's say you're not using the most expensive metal you have at this point. So let's say your most expensive metal has cost Ci, but instead, you decide to use Cj.

So let's say you decide to use some kj amount of Cj. So the value you're getting from this is Cj kj. And instead, if you use Ci, how much metal would you need to get the same value? You would need Cj kj over Ci. Does that make sense?

So this is the value you would obtain by using kj kilograms of this metal. So if you instead used this one, you'd get this value. And so this is the most expensive one. And Ci is greater than Cj, this value, so this value is less than kj.

So by using this metal instead of that one, you are decreasing the amount-- the weight you would need, so your minimization goes down. Make sense? So that's like a very simple greedy algorithm. And it's-- the algorithm is exactly what you'd expect, and the proof isn't very hard.

So let's move on to a slightly interesting one. So this is process scheduling. So let's say you have a computer, and you're running end processes. And each of the process has a time-- t1 through tn, again processes. And you want to order them in some way. So first, you will do process p1. Then you'll process p2, and so on, and so forth.

Then you'll define a completion time. So completion time is simply when does process i end.

So when does process i end? You just p1 plus-- it's like the time for p1 plus time for p2 up to pn.

So basically, you have all your processes. So let's says this is p1, this is p2, and so on. And the completion time for a certain process in the middle is just the sum of all times before it. That's completion time.

And now what you want to do is you want to minimize the average completion time, which is summation over all the completion times over n. So any ideas what an algorithm for this would look like? Essentially, you want to minimize the sum of all these times. So all these times, you want to minimize the average of these.

So what do you want to do? Do you want to shift the slower-- the processes which take more time, do you want to keep them at the end, or do you want to keep them at the beginning? So if you have a bunch of small processes, what do you do with them at the end? What do you do at the beginning?

Completion time is when does-- So let's say this is process pi. And completion time for process pi is like this distance. It's like, when does pi get completed? So it's summation of all the times-- so the time taken for p1, p2, up to the end, see?

So you want to basically minimize the average of these values. So where do you put the smaller processes-- would you put the shorter processes at the end or the beginning? Which one would decrease your average? The beginning? Makes sense, right?

So yeah, that makes sense. So you basically want to like scrunch these lines towards the beginning, so your average is smaller. Note that this total length is always the constant. It's like summation over all ti.

So let's go about-- so OK, this is strategy. Again, sort by ti, and this is increasing order, and that's it basically. So this is your algorithm, sorted by tn, but use the process in that order.

So let's try to prove this. So the way you prove this is a pretty generic method. It is often used to prove greedy algorithms. So let's say that this is not the optimal. Let's say someone comes up to you and tells you, OK, I have a better sequence. I have a sequence, let's say, called-- let's say I have a sequence of p1 to pn. And that sequence does better than a sorted order.

So you're like, OK, so if this is not sorted, then you have some elements in the middle. Let's

say you call them pi is greater than pj, with i is less than j. So there's some pi here, and there's some pj here, such that this is greater than that. So it's not in sorted order. So you can always find a pair like that.

So now I'm going to claim that if you swap these two values-- so you swap pi and pj-- that'll actually decrease whatever current average completion time you have. So initially, you had something like this. So-- no, let's not draw a line there. So let's say you had something like you had this process, so pi-- actually, this is the bigger process, so this is pi, and this is pj.

And now I'm saying that-- and you have some stuff in the middle. And my claim is that, no, this is not optimal. You'd do much better if you moved the pj over here. So you want to go from this to this, and big process.

So let's see what changes when you go from there to there. So first of all, observe that the completion times of everything behind this is the same. They all have the same completion time; nothing is affected. And you're only changing these two things. Everything after this is also the same-- has the same completion time.

So the only things that are changing are this one, this one, and all the ones up to this one. Even this one has the same completion time. Make sense? So how much is this changing by? So let's define this. Delta is equal to t of pi minus t of pj-- so the difference between these two processes.

So the original completion time of pi was this. And now the corresponding process down here, the completion time is decreased by delta. So completion time for us goes down like minus delta. This is a summation of completion time. This divided by n is a constant. So you just want to minimize this. So first it goes-- so this one goes down minus delta.

So let's look at the next process. The next process is something like this. So again, these do not change. You're only swapping these two. So this completion time also down a minus delta, and so on, and so forth. So you just get a bunch of minus deltas, which is equal to however many processes you have.

But that's not even important. What is important is that just by swapping, you're going to get at least one minus delta. And delta is positive, because assumption-- oops, sorry, t-- because assumption was that t pi minus t pj is positive. So just by swapping, you're going to always decrease it. So the claim that that sequence was an optimal solution is wrong.

So you can always do better by swapping two inversions. So that out of sorted order is called an inversion. So if you solve an inversion, you always get a better result. Does that proof make sense?

So that's a slightly more interesting recent algorithm. So let's move on to the third one we have here. The third one is event overlap. So this is how it works.

So you wake up in the morning, and you look at your calendar. And being an MIT student, your calendar looks pretty full. So let's say this is what it looks like.

So these are your events. Let's use some colors, make it a little clearer possibly. And let's say you have another event over here. You have something here. You have something here. You have something here. And you have something here. So OK, let's move this down.

So the problem is that you have this bunch of events planned out. Now clearly, they're overlapping, so you can't attend all of them. So the idea is you make a bunch of clones of yourself.

And so in this case, look at the matching colors. So if you create clone number 1 goes here, and clone number 2 goes to red, and clone number 3 goes to blue.

So then clone number 1 does this. Clone number 3 does the blue one. Clone number 2 does red. I guess, we should move the red back a little or forward a little bit just to make it clear. Yeah, there we go.

And now, you could easily see that this is optimal. So you can do this with three clones and no less. So you make three clones, and then you can go off to spring break. And your schedule is fine.

So now, how would you approach this problem? So what is a greedy strategy to, given a number of intervals, how do you find the minimum number of clones you need to cover your day? Any ideas? What is a naive thing you could do?

AUDIENCE: When you say to cover your day, then it's like the number--

AMARTYA SHANKHA BISWAS: So you want to do every event. But like so this clone can't-- so clone number 1 does this event. Then he can't do this event or this event.

AUDIENCE: Sort of maximizing your events?

AMARTYA SHANKHA BISWAS: You want to do all the events? You want to minimize the number of clones.

AUDIENCE: [INAUDIBLE]

AMARTYA SHANKHA BISWAS: So it's like interval scheduling. But you want to do all the intervals, but you're allowed to use multiple people to do all the intervals. Yes? Yeah?

AUDIENCE: You could sort by end time.

AMARTYA SHANKHA BISWAS: By end time, OK. What do you do after you sort by end time?

AUDIENCE: And then iterate over all the intervals once they're sorted and just count how many intervals there are between the [INAUDIBLE]. That would get complicated.

AMARTYA SHANKHA BISWAS: So you're close. So you do begin by sorting. But you can actually do it by sorting by end time. It's easier to visualize if you sort by start time. So leading from that, anyone want to top in?

AUDIENCE: It's when your sorting starts, and every time you get a class, you get a new clone.

AMARTYA SHANKHA BISWAS: So yeah, essentially, every time you can't add it to one of your current clones, you just create a new one. You could also do it by end time, because it's symmetrical, right? So if you sort by end time, then you start with the smallest-- last end time and go backwards, exactly the same thing.

So let's write it down. So sort by start time, and so actually, let's work out this example. So in this case, you would go-- OK, actually, if once you sort-- so first you have 1, then you have 2, 3, 4, 5, 6. So that's sorted by start time.

And then you have-- so first you go for this one. Then you go for 2, and 2 intersects with 1. So you put 2 into it. So this is clone number 1. And then you have to create a new clone for 2, so

you create the new clone. And there we go.

So then you go to 3. 3 clashes with both 1 and 2, so you have to create a new clone again. So in that case, you go forth and create 3. Then you go to 4. Now, 4, you see, it's starts with 2 and 3, but it is good with 1. So you just put 4 over here.

And if you continue like this, you essentially get this and this. Make sense? So that's how you schedule it. So does that algorithm make sense? Let's try to prove its correctness.

So let's look at the instance where you're inserting the m-th clone-- so m-th clone. So when the m-th is created, you already have some values in here. So you have 1, 2, all the way up to m minus 1. So now, you bring in your interval, and you see that it collides with all of these values.

So let's just draw the final interval for all these guys. So let's say the final interval for this guy was out here. Let's say the final interval for this guy was out here, and so on, and blah, blah, blah, blah, blah. And so when you create the m-th clone, you look at the start time.

So what happens is that the start time is somewhere, let's say, here. And now, you know that because of this-- so you're only adding a new clone when you don't have an available slot. So that means that there is some interval here, which intersects with this guy.

So how do you show that this is one interval? Well, it's like consider any level. But say there is no interval that intersects with it. So that means that there is either-- So if there were a gap here-- so let's say, at this location, this interval wasn't here. Let's say if you extrapolate this line outward-- so this is your current starting value. And let's say you look at this line.

And in this segment, you can't have something which starts after this, because this is the current highest sorting starting time. So there's no interval that starts after this. So the only interval that's going to exist have already ended here. And if they're already ended here, that means you could evaluate here. Does that make sense?

So basically, then you can show that, OK, so at every existing-- if you're adding a new clone, that means at every existing level, you have something which intersects. So what that means is that you have a single point of time where there are m minus 1 plus 1 intervals. That means that you absolutely need m intervals regardless of what your strategy is. So adding the m-th clone is necessary.

So if you go on, continue the argument-- let's say your total number of clones was m-- so you can just do this argument for m. There you will show that, oh, if I followed all these rules correctly, I can show that the start time for m intersects with m minus 1 other intervals. So there's no way I can create a scheduling with less than m clones. Did that argument make sense, or should I go over it again?

So that's somewhat hand wavy, but that shouldn't be-- OK. In any case, well, that's the three problems. So I guess we can go back to this one and sort of give the motivation for this. So this could, for example, be used in scheduling processes for servers, for instance.

So let's say your server gets a request to run n processes, and they have times like that. So this is like shortest time first. So you take all the short-- the smallest jobs, and you execute them in the beginning. And you wait for other jobs. And this can also be done online.

So you can have an online version of this. So if you take this algorithm and you do it online-- so let's say your server is running jobs, and you get a new request. So you get a new request, so you already have some set of t1 to tn. And let's say, at the current moment, ti is your smallest job. And you're running it, and you're currently at this point.

And then in the middle of running it, you can get new requests for jobs. So how would you modify this algorithm to handle that? So you still want to maintain this lowest average completion time thing. So how would you handle this situation. So let's say you're in the middle of a job and you get a bunch of new requests.

So current set is all these existing jobs plus some other things you get in here. So would you consider switching to a different job here, or would you keep doing this? Let's say one of the new jobs you get is really small. So what you would do in that case is that instead of continuing with this, you would switch to current smallest job.

So you would look at the remaining time, so that's important. So you could forget about the amount of time you already spent on this. You know what the remaining time is, and that is all that is relevant. So you can just consider this problem in a different framework. It's the exact same question.

You just look at remaining time, instead of total time. So if you're in the middle of a job, and a new one comes in which is smaller, you just switch to that, complete that, and then look at the remaining times for everything. So at some point of time, you might have a lot of half

completed jobs just lying around. And for all of them, you'll update their ti values to remaining time rather than start time. And that gives you a nice way to decide which processes to do online. And that gives you--

So this is assuming that all of your tasks have equal weights. So all of them have equal reward. So obviously, that's not always the case. You might be pushing back a very long job forever, because smaller things keep coming in and that might get important.

But everything is equally weighted, then this is the optimal thing you can do. And it's a very simple strategy that works. So those are the three problems I wanted to discuss. Do you guys have any other questions or comments or anything? Good? OK. We finished pretty early, so I guess, have a great spring break.