PROFESSOR: All right. Good morning, everyone. Let's get started. So we're going to start 6.046 in earnest today. We're going to start with our first module on divide and conquer.

You've all seen divide and conquer algorithms before. Merge sort is a classic divide and conquer algorithm. I'm going to spend just a couple minutes talking about the paradigm, give you a slightly more general setting than merge sort. And then we'll get into two really cool divide and conquer problems in the sense that these are problems for which divide and conquer works very well-- mainly, convex hall and median finding.

So before I get started on the material, let me remind you that you should be signing up for a recitation section on Stellar. And please do that even if you don't plan on attending sections. Because we need that so we can assign your problem sets to be graded, OK?

So that's our way of partitioning problem sets as well. And then the other thing is problem set one is going to go out today. And that it's a one week problem set. All problem sets are going to be a week in duration. Please read these problem sets the day that they come out. Spend 5, 10 minutes reading them.

Some things are going to look like they're magic, that they're-- how could I possibly prove this? If you think about it for a bit, it'll become obvious. We promise you that. But get started early. Don't get started at 7:00 PM when we have 11:59 PM deadline on Thursday, all right? That four hours or five hours of time may not be enough to go from magical to obvious, OK?

So let's get started with the paradigm associated with divide and conquer. It's just a beautiful notion that you can break up the problem into smaller parts and somehow compose the solutions to the smaller parts. And of course, the details are going to be what's important when we take a particular problem instance.

But let's say we're given a problem of size n. We're going to divide it into a sub problems-- I'll put that in quotes so you know it's a symbol-- a sub problems of size n over b. And here, a is an integer. And a is going to be greater than or equal to 1. It could be two. It could be three. It could be four.

This is the generalization I alluded to. And b does not have to be two or even an integer. But it has to be strictly greater than one. Otherwise, there's no notion of divide and conquer. You're not breaking things up into smaller problems. So b should be strictly greater than one. So that's the general setting. And then you'll solve each sub problem recursively. And the idea here is that once the sub problems become really small, they become constant size, it's relatively easy to solve them. You can just do exhaustive search.

If you have 10 elements and you're doing effectively a cubic search, well, 10 cubed is 1,000. That's a constant. You're in great shape as long as the constants are small enough. And so you're going to recurse until these problems get small. And then typically-- this is not true for all divide and conquer approaches. But for most of them, and certainly the ones we're going to cover today, the smarts is going to be in the combination step-- when you combine these problems, the solutions of these sub problems, into the overall solution. And so that's the story.

Typically, what happens in terms of efficiency is that you can write a recurrence that's associated with this divide and conquer algorithm. And you say t of n, which is a running time, for a problem of size n is going to be a times tfn over b-- and this is a recurrence-- plus the work that you need to do for the merge operational or the combine. This is the same as merge. And so you get a recurrence. And you're not quite done yet in terms of the analysis. Because once you have the recurrence, you do have to solve the recurrence. And it's usually not that hard and certainly it's not going to be particularly difficult for the divide and conquer examples that we're going to look, at least today.

But we also have this theorem that's called the master theorem that is essentially something where you can fairly mechanically plug in the a's and the b's and whatever you have there-- maybe it's theta n, maybe it's theta n square-- and get the solution to the recurrence.

I'm actually not going to do that today. But you'll hear once again about the massive theorem tomorrow in section. And it's a fairly straightforward template that you can use for most of the divide and conquer examples we're going to look at in 046 with one exception that we'll look at in median finding today that will simply give you the solution to the recurrence, OK?

So you've see most of these things before. That's a little bit of setup. And so let's dive right in into convex hull, which is my favorite problem when it comes to using divide and conquer. So convex hull, I got a little prop here which will save me from writing on the board and hopefully be more understandable. But the idea here is that in this case, we have a two dimensional problem with a bunch of points in a two dimensional plane. You can certainly do convex hull for three dimensions, many dimensions. And convexity is something that is a fundamental notion in optimization. And maybe we'll get to that in 6046 in advanced topics, maybe not. But in the context of today's lecture, what we're interested in doing is essentially finding an envelope or a hull associated with a collection of points on a two dimensional plane. And this hull obviously is going to be something, as you can guess, that encloses all of these points, OK?

So what I have here, if I make this string taut enough-- this is not working so well, but I think you get the picture. All right, so that's not a convex hull. This is not a convex hull for the reason that I have a bunch of points outside of the hull. All right, so let me just-- that is a convex hull. And now if I start stretching like that or like this or like that, that's still a convex hull, OK? So that's the game.

We have to find an algorithm. And we look at a couple of different ones that will find all of these segments that are associated with this convex hull, OK? So this is a segment that's part of the convex hull. That's a segment that's part of the convex hull. If, in fact, I had something like this-- and this was stretched out-- because I have those two points outside the convex hull, this may still be a segment that's part of the electronics hall but this one is not, right? So that's-- the game here is to find these segments. So if you're going to working with segments or tangents-- they're going to be used synonymously-- all of the tangents or segments associated with the entirety of the convex hull and we have to discover them. And only input that we have is the set of pointx-- xiy coordinates.

And there's just a variety of algorithms that you can use to do this. The one that I wish I had time to explain but I'll just mention is what's called a gift wrapping algorithm. You might not have done this, but I guarantee you I said you probably have taken a misshapen gift, right, and tried to wrap it in gift wrapping paper. And when you're doing that, you're essentially-- if you're doing it right you're essentially trying to find the convex hull of this three dimensional structure. You're trying to tighten it up. You're trying to find the minimum amount of gift wrapping paper.

I'm not sure if you've ever thought about minimizing gift wrapping paper, but you should have.

And that's the convex hull of this three dimensional shape. But we'll stick to two dimensions because we'll have to draw things on the board. So let me just spec this out a bit. I've been given endpoints in a plane. And those set of points are s, xi, yi such that i equals 1, 2 to n. And we're just going to assume here, just to make things easy because we don't want to have segments that are null or segments that are a little bit different because they're discontinuous. But we're going to assume that no two have the same x-coordinate.

This is just a matter of convenience. And no two have the same y-coordinate. And then finally, no three in a line. Because we want to be able to look at pairs of points and find these segments. And it just gets kind of inconvenient. You have to do special cases if there of them are on a line. And so the convex hull itself is the smallest polygon containing all points in s. And we're going to call that ch of s-- convex hull of s.

STUDENT: Smallest convex polygon.

PROFESSOR: The smallest convex polygon-- thank you. And so just as an example on the board, when you have something like this, you're going to have your convex hull being that. This one is inside of it. These two points are inside of it. And all the other ones form the hull. And so we might have p, q, r, s, t, u. And v and x are inside of the hull. They're not part of the specification of ch of s, which I haven't quite told you how we're going to specify that.

But the way you're going to specify that is simply by representing it as a sequence of points that are on the boundary on the hull in clockwise order. And you can think of this as being a doubly linked list in terms of the data structure that you'd use if you coded this up. So in this case, it would be p to q to r to s. You're going to start with t in this case. It's a doubly linked list. So you could conceivably start with anything. But that's the representation of the convex hull. And we're going to use clockwise just because we want to be clear on as to what order we're enumerating these points. It's going to become important when we do the divide and conquer algorithm. So let's say that we didn't care about divide and conquer just for the heck of it and I gave you a bunch of points over here.

Can you think of a simple-- forget efficiency for just a couple of minutes. Can you think of a simple algorithm that would generate the segments of the convex hull? For example, I do not want to generate this segment vx.

If I think of a segment as being something that is defined by two points, then I don't want to generate the segment vx because clearly the segment is not part of the convex hull. But whereas the segment pq, qr, rs, et cetera, they're all part of the convex hull, right? So what is the obvious brute force algorithm, forgetting efficiency, that given this set of points will generate one by one the segments of the convex hull? Anybody? Did you have your head up? No? Go ahead. Yep.

STUDENT: Draw the line and check how many other lines intersect with it.

PROFESSOR: Draw the line and check how many lines it intersects with.

STUDENT: Yeah. PROFESSOR: Is there-- I think you got-- you draw the line. That's good, right?

STUDENT: [LAUGHS] AUDIENCE: [LAUGHING]

PROFESSOR: Well-- but you want to do a little more. Yeah, go ahead.

STUDENT: For every pair of points you see, make a half-plane and see where they complete all of their other points. [INAUDIBLE]

PROFESSOR: Ah, so that's good. That's good. That's good. All right, so the first person who breaks the ice here always gets a Frisbee. Sorry man. At least I only hit the lecturer-- no liability considerations here. OK, now I'm getting scared. Right, so I think there's a certain amount of when I throw this, am I going to choke or not, right? But it's going to get higher when one of you guys in the back answers a question. So you're exactly right. And you draw a line. And then you just look at it. And you look at the half plane. And if all the points are to one side, it is a segment of the convex hull. If they're not, it's not a segment-- beautiful. All right, are we done? Can we go and enjoy the good weather outside? No, we've got ways to go here. So this is not the segment whereas one-- let me draw that. I should draw these in a dotted way.

This is not a segment. This is not a segment. This is a segment. And I violated my rule of these three not being in a straight line. So I'll move this over here. And then that's a segment and so on and so forth, OK? Right?

STUDENT: It's no longer a side with the ones below it.

PROFESSOR: I'm sorry?

STUDENT: It would have to go directly to the bottom one from the left one.

PROFESSOR: Oh, you're right. That's a good point. That's an excellent point. SO what happened here was when I moved that out-- exactly right. Thank you. This is good. So when I moved this out here, what happened was-- and I drew this-- well, this one here, my convex hull, changed. The problem specification changed on me. It was my fault. But then what would happen, of course, is as I move this, that would become the segment that was part of the convex hull, OK?

So sorry to confuse people. But what we have here in terms of an algorithm, if I leave the points the same, works perfectly well. So let me just leave the points the same and just quickly recap, which is, I'm going to take a pair of points. And I'm going to draw-- and let me just draw this in a dotted fashion first. And I'm going to say that's the segment. And I'm going to take a look at that line and say this breaks up the plane into two half planes. Are all about points on one side? And if the answer is yes, I'm going to go ahead and, boom, say that is a segment of my convex hull. If the answers is no, like in this case, I'm going to drop that segment,

OK? So now let's talk about complexity. Let's say that there are n points here. And how many segments do I have? I have O n square theta n square segments. And what is the complexity of the test? What is the complexity of the test that's associated with, once I've drawn the segments, deciding whether the segment is going to be a tangent which is part of the convex hull or not? What is the complexity?

STUDENT: O n.

PROFESSOR: O n-- exactly right. So on test complexity-- and so we got over theta n cubed complexity, OK? So it makes sense to do divide and conquer if you can do better than this.

Because this is a really simple algorithm. The good news is we will be able to do better than that. And now that we have a particular algorithm-- I'm not quite ready to show you that yet. Now that we have a particular algorithm, we can think about how we can improve things. And of course we're going to use divide and conquer. So let's go ahead and do that. And so generally, the divide and conquer, as I mentioned before, in most cases, the division is pretty straightforward. And that's the case here as well. All the fun is going to be in the merge step. Right, so what we're going to do, as you can imagine, is we're going to take these points. And we're going to break them up. And the way we're going to break them up is by dividing them into half lengths. We're going to just draw a line. And we're going to say everything to the left of the line is one sub problem, everything to the right of the line is another sub problem, go off and find the convex hull for each of the sub problems. If you have two points, you're done, obviously. It's trivial.

And at some point, you can say I'm just going to deal with brute force. If we can go down to order n cubed, if n is small, I can just apply that algorithm. So it doesn't even have to be the base case of n equals 1 or n equals 2. That's a perfectly fine thing to do. But you could certainly go with n equals 10, as I mentioned before, and run this brute force algorithm. And so at that point, you know that you can get down to small enough size sub problems for which you can find the convex hull efficiently. And then you've got these two convex hulls which are clearly on two different half planes because that's the way you defined them. And now you've got to merge them. And that's where all the fun is, OK? So let's just write this out again.

You're going to sort the points by x-coordinates. And we're going to do this once and for all. We don't have to keep sorting here because we're just going to be partitioning based on x-coordinates. And we can keep splitting based on x-coordinates because we want to generate these half-lengths, right? So if we can do those once and for all-- and for the input set S, we're going to divide into the left half A and right half B by the x-coordinates. And then we're going to compute CH of A and CH of B recursively. And then we're going to combine. So the only difference here from what we had before is the specification of the division. It looked pretty generic.

It's similar to the paradigm that I wrote before. But I've specified exactly how I'm going to break this up. So let's start with the merge operation. We're going to spend most of our time specing that. And again, there's many ways you could do the merge. And we want the most efficient way. That's obviously going to determine complexity. So, big question-- how to merge.

So what I have here, if I look at the merge step, is I've created my two sub problems corresponding to these two half planes. And what I have here is-- let's say I've generated, at this point, a convex hull associated with each of these sub problems. So what I have here is a1, a2. I'm going to go clockwise to specify the convex hull. And the other thing that I'm going to do is in the sub problem case, my starting point is going to be for the left sub problem, the coordinate that has the highest x value, OK? So that's a1 in this case-- the highest x value going over. x is increasing to the right. And for the right half of the problem, it's going to be the coordinate that has the lowest x value. And I'm going to go clockwise in both of these cases. So when you see an ordering associated with the subscripts for these points, start with a1 or b1 and then go clockwise. And that's how we number this-- so just notational, nothing profound here.

So I got these two convex hulls-- these sub hulls, if you will. And what I need to do now is merge them together. And you can obviously look at this and it's kind of obvious what the overall convex hull is, right?

But the key thing is, I'm going to have to look at each of the pairs of points that are associated with this and that and try to generate the tangents, the new tangents, that are not part of the sub hulls, but they're part of the overall hull, right? So in this case, you can imagine an algorithm that is going to kind of do what this brute force algorithm does except that it's looking at a point from here and a point from here.

So you could imagine that I'm going to do a pairwise generation of segments. And then I'm going to check to see whether these segments are actually tangents that are part of the overall convex hull or not. So what I would do here is I'd look at this. And is that going to be part of the overall hull? No, and precisely why not? Someone tell me why this segment a1 b1 is not part of the overall hull? Yeah, go ahead.

STUDENT: If we were to draw a line through the whole thing there would be one on both sides.

PROFESSOR: Exactly right-- that's exactly right. So here you go. So that's not part of it. Now, if I look at this-- well, same reason that's not part of it. In this case-- and this is a fairly obvious example. I'm going to do something that's slightly less obvious in case you get your hopes up that we have this trivial algorithm,

OK? This is looking good, right? That's supposed to be a straight line, by the way. So a4 b2-- I mean, that's looking good, right? Because all the points are on one side. So a4 b2 is our upper tangent. Right, so our upper tangent is something that we're going to define as-- if I look at each of these things, I'm going to say they have a yij.

OK, what is yij? yij is the y-coordinate. of the segment that I'm looking at, the ij segment. So this yij is for ai and bj. So what I have here is y42 out here. And this is-- for the upper tangent, yij is going to be maximum, right? Because that's essentially something which would ensure me that there are no points higher than that, right? So if I go up all the way and I find this that has the maximum yij, that is going to be my upper tangent. Because only for that will I have no points ahead of that, OK? So yij is upper tangent. This is going to be maximum. And I'm not going to write this down, but it makes sense that the lower tangent is going to have the lowest yij. Are we all good here? Yeah, question.

STUDENT: So I am just wondering, I couldn't hear what she said why we moved out a1 b1.

PROFESSOR: OK, so good. Let me-- that reason we moved out a1 b1 is because if I just drew a1 b1 like this-- and I'm extrapolating this. This is again supposed to be a straight line. Then you clearly see that there are points on either side of the a1 b1 segment when you look at the overall problem, correct? You see that on a1 b1, b2 is on this side, b3 is on this side if I just extend this line all the way to infinity in both directions. And that violates the requirement that the segment be part of the overall hull, OK? That make sense? Good. So, everybody with me?

So clearly, there's a trivial merge algorithm here. And the trivial merge algorithm is to look at not every pair of points-- every ab pair, right? Every aibj pair. And so what is the complexity of doing that? If I have n total points, the complexity would be-- would be in square, right? Because maybe I'd have half here and half here, ignore constants. And you could say, well, it's going to be n squared divided by 4, but that's theta n squared. So there's an obvious merge algorithm that is theta n square looking at all pairs of points. And when I mean all pairs of points, I mean like an a and a b. Because I want to pick a pair when I go left of that dividing line and then right of the dividing line. But either way, it's theta n square, OK? So now you look at that and you go, huh. Can I do a better?

What if I just went for the highest a point and the highest b point and I just, no, that's it? I'm done-- constant time. Wouldn't that be wonderful? Yeah, wonderful, but incorrect, OK?

Right, so what is an example. And so this is something that I spent a little bit of time last night concocting. So I'm like you guys too. I do my problem set the night before. Well, don't do as I do. Do as I say. But I've done this before. So that's the difference.

But this particular example is new. So what I have here is I'm going to show you why there's not a trivial algorithm, OK, that-- I got to get these angles right-- that you can't just pick the highest points and keep going, right? And then that would be constant time. So that's my a over here. And let's assume that I have my dividing line like that. And then what I'm going to do here-- and I hope I get this right-- is I'm going to have something like this, like that. And then I'm going to have b1 here clockwise-- so b2, b3, and b4. So as you can see here, if I look at a4-- a little adjustment necessary. OK, so if I look at that, a4 to b1 versus-- I mean, just eyeball it. A3 to b1-- right, is a4 to b1 going to be the upper tangent? No, right? So now a3 is lower than a4. You guys see that?

And b1 is lower than b2, right? So it's clear that if I just took a4 to b2 that it will not be an upper tangent. Everybody see that? Yep, all right, good. So we can't have a constant time algorithm.

We have theta and square in the back. So it is there something-- maybe theta n?

How would we do this merge and find the upper tangent by being a little smarter about searching for pairs of points that give us this maximum yij? I mean, the goal here is simple. At some level, if you looked at the brute force, I would generate each of these things. I would find the yj intercepts associated with this line. And I just pick the maximum. And the constant time algorithm doesn't work. The theta n squared algorithm definitely works. But we don't like it. So there has to be something in between. So, any ideas? Yeah, back there.

STUDENT: So... I had a question. [INAUDIBLE]

PROFESSOR: No, you're just finding-- no, you're maximizing the yij. So for once you have this segment-- so the question was, isn't the obvious merge algorithm theta n cubed, right? And my answer is no, because the theta n extra factor came from the fact that you had to check every point, every endpoint, to see on which side of the plane it was. Whereas here, what I'm doing is I've got this one line here that is basically y equals 0, if you like, or y equals some-- I'm sorry, x equals 0 or x equals some value. And I just need to, once I have the equation for the line associated with a4 b1 or a4 b2, I just have to find the intercept of it, which is constant time, right? And then once I find the intercept of it, I just maximize that intercept to get my yij. So I'm good, OK?

So it's only theta n squared, right? Good question. So this is actually quite-- very, very, very clever. This particular algorithm is called the two finger algorithm. And I do have multiple fingers, but it's going to work a lot better if I borrow Eric's finger. And we're going to demonstrate to you the two finger algorithm for merging these two convex hulls. And then we'll talk about the complexity of it. And my innovation again last night was to turn this from a two-finger algorithm. Not only did I have the bright idea of using Eric-- I decided it was going to become the two finger an string algorithm. So this is wild.

This is my contribution to 046 lore-- come on. So the way the two finger algorithm works-- this pseudo code should be incomprehensible. If you just look at it and you go, what, right? But this demo is going to clear everything up. Right so here's what you do. So now we're going to do a demo of the merge algorithm that is a clever merge algorithm than the one that uses order n square time. And it's correct. It's going to get you the correct upper tangent and what we are starting at here is with Erikâ€™s left finger on A1, which is defined to be the point that's closest to the vertical line that you see here, the one that has the highest x-coordinate. And my finger is on B1, which is the point that has the smallest X-coordinate on the right hand side sub-hull. And what we do is we compute, for the segment A1 B1, we compute by Yij, in this case Y11, which is the intercept on the vertical line that you see here that Erik just marked with a red dot. And you can look at the pseudocode over on, to my right if I face the board. And what happens now is I'm going to move clockwise, and I'm going to go from B1 to B4. And what happened here?

Did the Yij increase or decrease? Well, as you can see it decreased. And so I'm going to go back to B1. And we're not quite done with this step here. Erikâ€™s going to go counterclockwise over to A4. And we're going to check again, yeah, keep the string taught, check again whether Yij increased or decreased and as is clear from here Yij increased. So now we move to this point. And as of this moment we think that A4 B1 has the highest

Yij. But we have a while loop. Weâ€™re going to have to continue with this while loop, and now what happens is, Iâ€™m going to go from B1 clockwise again to B4. And when this happens, did Yij increase or decrease? Well it decreased. So I'm going to go back to B1 and Erik now is going to go counterclockwise to A3. And as you can see Y31 increased a little bit, so we're going to now stop this iteration of the algorithm and we're at A3 B1, which we think at this point is our upper tangent, but let's check that. Start over again on my side B1 to B4, what happened? Well Yij decreased. So I'm going to go back to B1. And then Erikâ€™s going to try. Heâ€™s going conterclockwise, he's going to go A3 to A2 and, well, big decrease in Yij. Now Erik goes back to A3. At this point we've tried both moves, my clockwise move and Erikâ€™s counterclockwise move. My move from B1 to B4 and Erikâ€™s move from A3 to A2. So we've converged, we're out of the while loop, A3 B1 for this example is our upper tangent. All right. You can have your finger back Erik.

So the reason this works is because we have a convex hull here and a convex hull here. We are starting with the points that are closest to each other in terms of A1 being the closest to this vertical line, B1 being the closest to this vertical line, and we are moving upward in both directions because I went clockwise and Erik went counterclockwise. And that's the intuition of why this algorithm works. We're not going to do a formal proof of this algorithm, but the monotonicity property corresponding to the convexity of this subhull and the convexity of the subhull essentially can give you a formal proof of correctness of this algorithm, but as I said we won't cover that in 046. So all that remains now is to look at our pseudocode which matches the execution that you just saw and talk about the complexity of the pseudocode. So what is the complexity of this algorithm? It's order n, right? So what has happening here, if you look at this while loop, is that while I have two counters, I'm essentially looking at two operations per loop.

And either one of those counters is guaranteed to increment through the loop. And so since I have in this case p points, in one case p plus q equals n-- so let's say I had p points here and I have q points here. And got p plus q equals n. And I got a theta n merge simply because I'm going to be running through and incrementing-- as long as I'm in the loop, I'm going to be incrementing either the i or the j. And the maximum they can go to are p and q before I bounce out of the loop or before they rotate around.

And so that's why this is theta n. And so you put it all together in terms of what the merge corresponds to in terms of complexity and put that together with the overall divide and conquer. We have a case where this is looking like a recurrence that you've seen many a time t of n. I've broken it up into two sub problems. So I have 2. And I could certainly choose this l over here that's my line l to be such that I have a good partition between the two sets of points.

Now, if I choose l to be all the way on the right hand side, then I have this large sub problem-- makes no sense whatsoever. So what I can do-- there's nothing that's stopping me when I've sorted these points by the x-coordinates to do the division such that there's exactly the same number, assuming an even number of points n, exactly the same number on the left hand side or the right hand side. But I can get that right roughly certainly within one very easily. So that's where the n over 2 comes from, OK? In the next problem that we'll look at, the median finding problem, we'll find that trying to get the sub problems to be of roughly equal size is actually a little difficult, OK? But I want to point out that in this particular case, it's easy to get sub problems that are half the size because you've done the sorting.

And then you just choose the line, the vertical line such that you've got a bunch of points that are on either side. And then in terms of the merge operation, we have 2t n over 2 plus theta n. People recognize this recurrence? It's the old merge sort recurrence. So we did all of this in-- well, it's not merge sort. Clearly the algorithm is not merge sort. We got the same recurrence. And so this is theta n log n-- so a lot better than theta nq. And there's no convex hull algorithm that's in the general case better than this. Even the gift wrapping algorithm that I mentioned to you, with the right data structures, it gets down to that in terms of theta n log n, but no better.

OK, so good. That's pretty much what I had here. Again, like I said, happy to answer questions about the correctness of this loop algorithm for merge later. Any other questions associated with this?

STUDENT: Question. Yeah, back there.

STUDENT: If the input is recorded by x coordinates, can you do better than [INAUDIBLE]?

PROFESSOR: No, you can't, because-- I mean, the n log n for the pre-sorting, I mean, there's another theta n log n for the sorting at the top level. And we didn't actually use that, right? So the question was, can we do better if the input was pre sorted?

And I actually did not even use the complexity of the sort. We just matched it in this case. So theta n log n-- and then you can imagine maybe that you could do a theta n sort if these points were small enough and you rounded them up and you could use a bucket sort or a counting sort and lower that.

So this theta n log n is kind of fundamental to the divide and conquer algorithm. The only way you can improve that is by making a merge process that's even faster. And we obviously tried to cook up a theta one merge process. But that didn't work out, OK?

STUDENT: But are there algorithms that [INAUDIBLE] ?

PROFESSOR: First-- if you assume certain things about the input, you're absolutely, right? So one thing you'll discover in algorithms in 6046 as well is that we're never satisfied.

OK, so I just said, oh, you can't do better than theta n log n. But that's in the general case. And I think I mentioned that. You're on the right track. If the input is pre sorted, you can take that away-- no, it doesn't help in that particular instance if you have general settings. But if you-- the two dimensional case-- if the hull, all the segments have a certain characteristic-- not quite planar, but something that's a little more stringent than that-- you could imagine that you can do improvements. I don't know if any compelling special case input for convex hull from which you can do better than theta n log n.

But that's a fine exercise for you, which is in what cases, given some structure on the points, can I do better than theta n log n? So that's something that keeps coming up in the algorithm literature, if you can use that, OK? Yeah, back there-- question.

STUDENT: Where's your [INAUDIBLE] step? You also have to figure out which lines to remove from each of your two...

PROFESSOR: Ah, good point. And you're exactly, absolutely right. And I just realized that I skipped that step, right?

Thank you so much. So the question was, how do I remove the lines? And it's actually fairly straightforward. Let's keep this up here. And we don't need this incomprehensible pseudo code, right? So let's erase that. And thank you for asking that question. So it's a little simple cut and paste approach where let's say that I find the upper tangent ai bj. And I find the lower tangent.

Let's call it ak bm. And in this particular instance, what do I have? I have a1, a2, a3, a4 as being one of my sub hulls. And then I have b1, b2, b3, b4 as the other one. Now, what did we determine to be the upper tangent? Was it a3 b1? Right, a3 b1? So a3 b1 was my upper tangent. And I guess it was a1-- a1 b4? A1 b4 was my lower tangent.

So the big question is, now that I've found these two, how do I generate the collect representation of the overall convex hull? And so it turns out that you have to do this-- and then the complexity of this is important as well. And you need to do what's called a cut and paste that's associated with this where we're going to just look at this and that. So if we're going to have these two things, then we've got to generate a list of points. Now, clearly a4 is not going to be part of that, right? A4 is not going to be part of the overall hull. What is it that we want? We want something like a1, a2, a3, b1, b2, b3, b4, right? But there's a point that we have to discard here. Agree?

And so the way we do this is very mechanical. That's the good news here. I mean, you don't have to look at it pictorially. I just made that up looking at-- eyeballing it. Clearly, a computer doesn't have eyeballs, right? And so what we're going to do is we're going to say the first link-- in general, the first link is ai to bj. Because that's my upper tangent, OK? And in this case, it's going to be a3 d1, OK? And then I'm going to go down the b list until you see bm, which is the lower tangent. You're on the b list. So you're looking for the lower tangent point. And then you're going to jump until you see bm. You link it to ak, OK? You link it to ak and continue until you return to ai. And then you have your circular list, OK? So what you see here is you have a3 here. So I'm going to go ahead and write out the execution of what I just wrote here.

So I have a3. And I'm going to go jump over to b1. So I'm going to write down b1. Then I'm going to along the b's until I get to b4. In this case, I'm going to include all of the b's. So I got b1, b2, b3, b4. And then I'm going to jump from b4 to a1 because that's part of my lower tangent. And I got a1 here, a2. And then I'm back to a3, which is great. Because then I'm done, OK? And so exactly what I said happened, thank goodness, which is we dropped a4 but we kept all the other points. Does that answer your question? Good.

What is the complexity of cut and paste? It's order n. I'm just walking through these lists. So there's no hidden complexity here, OK? Good, good-- thank you. You definitely deserve a Frisbee. In fact, you deserve two, right? Where are you? I-- oh, could you stand up? Yeah, right-- two colors. All right. Oh, so he-- well, you can give it to him if you like. So good, thank you.

So are we done? Are we done with convex hull? OK, good. So let's go on and do median finding. Very different-- very different set of issues here. Still on divide and conquer, but a very different set of issues. The specification here is, of course, straightforward. You can think of it as I just want a better algorithm than sorting and looking for the median at the particular position-- in over two position, for example. Let's say n is odd. And it's floor of n over 2. You can find that median. Right, so it's pretty easy if you can do sorting. But we're never satisfied with using a standard algorithm. If we think that we can do better than that. So the whole game here is going to be I'm going to find the median. And I want to do it in better than theta n log n time. OK, so that's what median finding is all about. You're going to use divide and conquer for this.

And so in general, we're going to define, given a set of n numbers, define rank of x as the numbers in the set that are greater than-- I'm sorry, less than or equal to x. I mean, you could have defined it differently. We're going to go with less than or equal to. So in general, the rank, of course, is something that could be used very easily to find the median. So if you want to find the element of rank n plus 1 divided by 2 floor, that's what we call the lower median. And n plus 1 divided by 2 ceiling is the upper median. And they may be the same if n is odd. But that's what we want. So you can think of it as it's not median finding, but finding elements with a certain rank. And we want to do this in linear time, OK?

So we're going to apply divide and conquer here. And as always, the template can be instantiated. And the devil is in the details of either division or merge. And we had most of our fun with convex hull on the merge operation. It turns out most of the fun here with respect to median finding is in the divide, OK? So what I want is the definition of a select routine that takes a set of numbers s. And this is the rank. So I want a rank i. And that i might be n over 2-- well, floor of n plus 1 over 2, whatever?

And so what does the divide and conquer look like? Well, the first thing you need to do is divide. And as of now, we're just going to say you're going to pick some element x belonging to s. And this choice is going to be crucial. But at this point, I'm not ready to specify this choice yet, OK? So we're going to have to do this cleverly. And then what we're going to do is we're going to compute on k, which is the rank of x, and generate two sub arrays such that I want to find the fifth highest element. I want to find the median element. I want to find the 10th highest element. So I have to keep track of what happens in the sub problems. Because the sub problems are going to determine, depending on how many elements are inside those sub problems, which I can only determine after I've solved those sub problems. I'm going to have to collect that information and put it together in the merge operation.

So if I want to find the 10th highest element and I've broken it up relatively arbitrarily, it's quite possible that the 10th highest element is going to be discovered in the left one or the right one. And I have to show that it's the 10th highest. And it might be that there's four elements in the left and five on the right that are-- let's see. If I defined the rank as less than or equal to x, there's four on the left and five on the right that are smaller. And that's why this is the 10th highest element. And that's essentially what we have to look at. So b and c are going to correspond to the sub arrays that you can clearly eliminate one of them. You can count the number of elements in b, count the number of elements in c. And you can eliminate one of them in this recursion as you're discovering this element with the correct rank-- in this case, i. So let me write the rest of this out and make sure we're all on the same page.

What I have here pictorially is I've generated b here and c. So this is all of b and that's all of c. I have k minus 1 elements here in b. And let's say I have n minus k elements in c. And I'm going to do-- essentially take-- once I've selected a particular element, I'm going to look at all of the elements that are less than it and put it into the array b. I'm going to look at all the elements that are better than it. Let's assume all elements are unique. I'm going to put all of them into c. And I'm going to recur on b and c. Those two are my sub problems. But what I have to do is once I recur and I discover the ranks of the sub problems, I have to put them together. So what I have here is if k equals i-- so I computed the rank and I realized that if k equals-- equals i, I should say-- if k equals i, then I'm going to just return x. I'm done at this point. I got lucky. I picked an element x and it magically ended up having the correct rank, OK?

Not always going to happen. And so in other case, if k is greater than i, then going to return select bi.

So what I've done here is if k is greater than i, then I'm going to say, oh, so now I'm going to have to find the element in b. I know that it's going to be in b because k is greater than i. And I've got to find the exact position depending on what i is over here. But it's going to be somewhere between 1 and k minus 1. And then the last case is if k is less than i, then this is a little more tricky. I'm going to turn on c of i minus k, OK? So what happens here is that my k is-- the rank for the x that I looked at over here is less than i.

So I know that I'm going to find this element that I'm looking for in c. But if I just look at c, I don't want to look at c and look for an element of rank i within c, right? That doesn't make sense because I'm looking for an element of rank i in the overall array that was given to me. So I have to subtract out the k elements that correspond to x and all of the k minus 1 elements that are in b to go figure out exactly what position or rank I'm looking for in the sub array corresponding to c, OK? So, people buy that. So that's just a small, little thing that you have to keep in mind as you do this. So that's pretty straightforward, looking pretty good. And you say, well, am I done here? And as you can imagine, the answer is no, because we haven't specified this value.

Now, can someone tell me, at least from an efficiency standpoint, what might happen, what we're looking for here? As you can imagine, we want to improve on theta n log n. And so you could you say, well, I'm happy with theta n. That theta n complexity algorithm is better than a theta n log n complexity algorithm, which is kind of in the bag. Because we know how to sort and we know how to index. So we want a theta n algorithm. Now, if you take this and if I just picked, let's say, the biggest element-- I kept picking x to be n or n minus 1 or just picked a constant value. I picked x to be in the middle.

I picked the index. I can always pick an element based on its index. I can always go for the middle one. So what is the worst case complexity of this algorithm? If I don't specify or I give you this arbitrary selection corresponding to x belonging to s, what is the worst case complexity of this algorithm? Yeah, go ahead.

STUDENT: N squared.

PROFESSOR: N squared-- why is that?

STUDENT: Because if you [INAUDIBLE] take like the least element.

PROFESSOR: Yep. STUDENT: How do you compare like N o against the other analysis?

PROFESSOR: Exactly right. That's exactly right. So what happens is that you're doing a bunch of work here with this theta n work. Right here, this is theta n work, OK? So given that you're doing theta n work here, you have to be really careful as to how you pick the x element. So what might happen is that you end up picking the x over here. And given the particular rank you're looking for, you have to now-- you're left with a large array that has n minus 1 elements in the worst case. You started with n. You did not go to n over 2 and n over 2, which is what divide and conquer is all about-- even n over b, OK?

You went to n minus 1. And then you go to n minus 2. And you go to n minus 3 because you're constantly picking-- this is worst case analysis. You're constantly picking these sub arrays to be extremely unbalanced. So when the sub arrays are extremely unbalanced, you end up doing theta n work in each level of the recursion. And those theta n's, because you're going down all the way from n to one, are going to be theta n square when you keep doing that, OK? So thanks for that analysis. And so this is theta n squared if you have a batch selection. So we won't talk about randomized algorithms, but the problem with randomized algorithms is that the analysis will be given a probability distribution. And it'll be expected time.

What we want here is a deterministic algorithm that is guaranteed to run in worst case theta n. So we want a deterministic way of picking x belonging to s such that all of this works out and when we get our recurrence and we solve it, somehow magically we're getting fully balanced partitions-- firmly balanced sub problems in the sense that it's not n minus 1 and 1. It's something like-- it could even be n over 10 and 9n over 10. But as long as you guarantee that, you're shaking things down geometrically. And the asymptotics is going to work out. but the determinism is what we need. And so we're going to pick x cleverly. And we don't want the rank x to be extreme. So this is not the only way you could do it, but this is really very clever.

There's a deterministic way. And you're going to see some arbitrary constants here. And we'll talk about them once I've described it. But what we're going to do is we're going to arrange s into columns of size 5, right? We're going to take this single array. And we're going to make it a two dimensional array where the number of rows is five and the number of columns that you have is n over 5-- the ceiling in this case. And then we're going to sort it each column, big elements on top. And we're going to do this in linear time. And you might say, how did that happen? Well, there's only five elements. So it's linear. You could do whatever you wanted. You could do n raised to four.

But it's five raised to four and it's constants. Don't you love theory? So then we're going to find what we're going to call the median of medians. So I'm going to explain this. This works for arbitrary rank, but it's a little easier to focus in on the median to just explain the particular example. Because as you can see, there's an intricacy here associated with the break up. And so here we go. I'm going to draw out a picture. And we're going to try and argue that this deterministic strategy that I'll specify gives you fairly balanced partitions in all cases, OK?

So what we see here is we see-- pictorially, you see columns of length five. Each of these dots corresponds to a number. This one dimensional array got turned into a two dimensional right. So I got four full columns. And it's suddenly possible, given n, that my fifth column is not full, right? So that's certainly possible. So that's why I have that up here. It so what I've here is I'm going to lay them out this way. And I'm going to look at that. I'm going to look at the middle elements of each of these n over five columns. That's exactly what I'm going to look at. Now, if I look at what I want, what I want over here is this x. If I want to find-- I'm going to find the median of medians. So is x. Now, it is true the first that these columns-- I'm just putting that up here imagining that that's x.

That's not guaranteed to be x because the columns themselves aren't-- well, these columns are sorted. And what I'm going to have to guarantee, of course, is that when I go find this median of medians is that it ends up being something that gives me balanced partitions. So maybe say a little bit more before I explain what's going on.

Each of these columns is sorted. And s is arranged into columns of size 5 like I just said here. These are the medians, OK? If I look at determining the medians and I say that once I've determined this x, which I've discovered that it's the median, then this is right there in the middle. There's going to be a bunch of columns to the left of it, a bunch of elements to the left of it, and a bunch of elements to the right of it. And in this case, I have five columns. I could have had more. It happens to be the third one.

So the idea is that once I find this median of medians, which corresponds to this x number, I can say that all of the columns-- these all correspond to columns that have their median element greater than x. These correspond to columns that have their median element less than x, OK? So what I have here in this picture is that these elements here are going to be greater than x. And these elements here are going to be less than x. So let me clear. What's happened here is we've not only sorted all of the columns such that you have large elements up here.

Each of these five columns have been sorted that way. On top of that, I've discovered the particular column that corresponds to the medians of medians. And this is my x over here. And it may be the case that these columns aren't sorted. This one may be larger than that or vice versa-- same thing over there. I have no idea. But it's guaranteed that once I find this median that I do know all of the columns that have elements in this position that are less than this x. And I know columns that in this position have elements that are greater than x, OK? Yep.

STUDENT: Shouldn't the two elements below x also be computed [INAUDIBLE] less than x.

PROFESSOR: You're exactly right. I would have probably been able to get the same asymptotic complexity if I dropped those because I had a constant number. But you're absolutely exactly right.

So the point that-- the question was-- I just redrew it. These two are clearly less than x as well because they're part of the sorting. And that's essentially I have here. Now, my goal here-- and you can kind of see from here as to where we're headed. What I've down here by this process of sorting each column and finding the median of medians is that I found this median of medians such that there's a bunch of columns on the left. And roughly half of those elements in those columns are less than x. And there are a bunch of columns on the right. And roughly half of those columns have elements that are greater than x. So what I now have to do is to do a little bit of math to show you exactly what the recurrence is. And let me do that over here.

So that's the last thing that we have to do. I probably won't solve the recurrence, but that can wait until tomorrow. The recurrence will be something that's not particularly difficult to solve. So I want to now make a more quantitative argument that the variable being n as to how many elements are guaranteed to be greater than x. And essentially what I'm saying, which is I'm writing out what I have on that picture there, half of the n over 5 groups contribute at least three elements greater than x except for one group with possibly less than five elements, which is the one that I have all the way to the right, and one group that contains x. So for all the other columns, I'm going to get three elements that are greater than x. And so if you write that out, this says there are at least three n over 10, because I have half of all of those groups, minus 2. And I'm not counting perfectly accurately here, but I have an at least. So this should all be fine. 3n over 1d-- 3 times n over 10 minus 2 elements are strictly greater than x. And that comes from that picture.

I'm going to be able to say the same thing for less than x as well. I can't count the one. Depending on how things go, maybe I could have played around and subtracted 1 instead of a 2 in the latter case. But I'm just being conservative here. It is clear that I'm going to have a bunch of columns that are full columns, that are going to be contributing three elements that are greater than x. And in this case, I have, well, two of them here for the less than x. And I got one for the greater than x. So that's all that I'm seeing over here with respect to the balance of the partitions. And it turns out that's enough. It turns out all I have to do with this observation is to go off and run the recurrence. And we're going to get an efficient algorithm. Yep.

STUDENT: Should it not be like greater than or equal to, because there's... [INAUDIBLE]

PROFESSOR: No, there's nothing that's equal.

STUDENT: So you are saying, that's all you need.

PROFESSOR: Yeah. Yeah, I assume that-- so, convenience, yeah. There's always a little bit of convenience thrown in here. We will assume that the a has unique elements. So there's nothing that's x, OK? Good.

So the recurrence, once you do that, is t of n equals-- we're going to just say it's order one for n less than or equal to 140. Where did that come from? Well, like 140. It's just a large number. It came from the fact that you're going to see 10 minus 3, which is 7. And then you want to multiply that by 2. So some reasonably large number-- we're going to go off and we're going to assume that's a constant. So you could sort those 140 numbers and find the median or whatever rank. It's all constant time once you get down to the base case. So you just want it to be large enough such that you could break it up and you have something interesting going on with respect to the number of columns. So don't worry much about that number. The key thing here is the recurrence, all right?

And this is what we have spent the rest of our time on. And I'll just write this out and explain where these numbers came from. So that's our recurrence for n less than or equal to 140. And else, you're going to do this. So what is going on here? What are all of these components corresponding to this recurrence?

Really quickly, this is simply something that says I'm finding the median of medians. I'm finding some element that has a certain rank. So this median of medians is going to be running on n over 5 columns. So I've got this-- there are n over 5 columns here. And I'm going to be calling this algorithm recursively, the median finding algorithm, to do that-- finding the median of medians. This thing over here is-- I'm going to be discarding at least regardless of what I do. Because I have these two statements here, I take the overall n. And I'm going to discard. In my paradigm over here, I'm either going to go with b or I'm either going to go with c depending on what I'm looking for. And given that b and c are not completely unbalanced,

I'm going to be discarding 3n over 10 minus 6 elements, which simply corresponds to me ignoring the ceiling here and multiplying the 3 out. So that's 3n over 10 minus 6. So then I have 7n over 10 plus 6. That's the maximum size partition that I'm going to recur on. It's only going to be exactly one of them, as you can see from that. It's either else. It's not recurring on both of them. It's recurring on one of them. So that's where the 7n over 10 plus 6 comes from. And then you ask where does this theta n come from. Well, the theta n comes from the fact that I do have to do some sorting. It's constant time sorting for every column, OK? Because it's only five elements.

So I'm going to do constant time sorting. But there's order n columns. Because it's-- then it's n over 5 columns. So this is the sorting of all of the columns, all right? So that's it. And I'll just leave you with-- you cannot apply the master theorem for solving this particular recurrence. But if you make the observation-- and you'll see this in section. You make the observation that n over 5 plus 7n over 10 is actually less than n. So you get 0.2n here and 0.7n there. That's actually less than n. This thing runs in linear time. And you'll see that in section tomorrow. So this whole thing is theta n time. See you next time.