The following content is provided under a Creative Commons license. Your support will help MIT OpenCourseWare continue to offer high quality educational resources for free. To make a donation or view additional materials from hundreds of MIT courses, visit MIT OpenCourseWare at ocw.mit.edu.

PROFESSOR: Hi. I'm Srini Devadas. I'm a professor of electrical engineering and computer science. I'm going to be co-lecturing 6.006-- Introduction to Algorithms-- this term with professor Erik Domane. Eric, say hi.

ERIK DOMANE: Hi.

[LAUGHTER]

PROFESSOR: And we hope you're going to have a fun time in 6.006 learning a variety of algorithms. What I want to do today is spend literally a minute or so on administrative details, maybe even less.

What I'd like to do is to tell you to go to the website that's listed up there and read it. And you'll get all information you need about what this class is about from a standpoint of syllabus; what's expected of you; the problem set schedule; the quiz schedule; and so on and so forth. I want to dive right in and tell you about interesting things, like algorithms and complexity of algorithms. I want to spend some time giving you an overview of the course content.

And then we're going to dive right in and look at a particular problem of peak finding-- both the one dimensional version and a two dimensional version-- and talk about algorithms to solve this peak finding problem-- both varieties of it. And you'll find that there's really a difference between these various algorithms that we'll look at in terms of their complexity. And what I mean by that is you're going to have different run times of these algorithms depending on input size, based on how efficient these algorithms are.

And a prerequisite for this class is 6.042. And in 6.042 you learned about asymptotic

complexity. And you'll see that in this lecture we'll analyze relatively simple algorithms today in terms of their asymptotic complexity. And you'll be able to compare and say that this algorithm is fasten this other one-- assuming that you have large inputs-- because it's asymptotically less complex.

So let's dive right in and talk about the class.

So the one sentence summary of this class is that this is about efficient procedures for solving problems on large inputs. And when I say large inputs, I mean things like the US highway system, a map of all of the highways in the United States; the human genome, which has a billion letters in its alphabet; a social network responding to Facebook, that I guess has 500 million nodes or so. So these are large inputs.

Now our definition of large has really changed with the times. And so really the 21st century definition of large is, I guess, a trillion. Right? Back when I was your age large was like 1,000.

[LAUGHTER]

I guess I'm dating myself here. Back when Eric was your age, it was a million. Right?

[LAUGHTER]

But what's happening really the world is moving faster, things are getting bigger. We have the capability of computing on large inputs, but that doesn't mean that efficiency isn't of paramount concern. The fact of matter is that you can, maybe, scan a billion elements in a matter of seconds.

But if you had an algorithm that required cubic complexity, suddenly you're not talking about 10 raised to 9, you're talking about 10 raised to 27. And even current computers can't really handle those kinds of numbers, so efficiency is a concern.

And as inputs get larger, it becomes more of a concern. All right? So we're

concerned about--

--efficient procedures-- for solving large scale problems in this class.

And we're concerned about scalability, because-- just as, you know, 1,000 was a big number a couple of decades ago, and now it's kind of a small number-- it's quite possible that by the time you guys are professors teaching this class in some university that a trillion is going to be a small number.

And we're going to be talking about-- I don't know-- 10 raised to 18 as being something that we're concerned with from a standpoint of a common case input for an algorithm. So scalability is important. And we want to be able to track how our algorithms are going to do as inputs get larger and larger.

You going to learn a bunch of different data structures. We'll call them classic data structures, like binary search trees, hash tables-- that are called dictionaries in Python-- and data structures-- such as balanced binary search trees-- that are more efficient than just the regular binary search trees.

And these are all data structures that were invented many decades ago. But they've stood the test of time, and they continue to be useful. We're going to augment these data structures in various ways to make them more efficient for certain kinds of problems. And while you're not going to be doing a whole lot of algorithm design in this class, you will be doing some design and a whole lot of analysis.

The class following this one, 6.046 Designing Analysis of Algorithms, is a class that you should take if you like this one. And you can do a whole lot more design of algorithms in 6.046. But you will look at classic data structures and classical algorithms for these data structures, including things like sorting and matching, and so on.

And one of the nice things about this class is that you'll be doing real implementations of these data structures and algorithms in Python. And in particular are each of the problem sets in this class are going to have both a theory part to

them, and a programming part to them. So hopefully it'll all tie together.

The kinds of things we're going to be talking about in lectures and recitations are going to be directly connected to the theory parts of the problem sets. And you'll be programming the algorithms that we talk about in lecture, or augmenting them, running them. Figuring out whether they work well on large inputs or not.

So let me talk a little bit about the modules in this class and the problem sets. And we hope that these problem sets are going to be fun for you. And by fun I don't mean easy. I mean challenging and worthwhile, so at the end of it you feel like you've learned something, and you had some fun along the way. All right?

So content wise--

--we have eight modules in the class. Each of which, roughly speaking, has a problem set associated with it. The first of these is what we call algorithmic thinking. And we'll kick start that one today. We'll look at a particular problem, as I mentioned, of peak finding. And as part of this, you're going to have a problem set that's going to go out today as well. And you'll find that in this problem set some of these algorithms I talk about today will be coded in Python and given to.

A couple of them are going to have bugs in them. You'll have to analyze the complexity of these algorithms; figure out which ones are correct and efficient; and write a proof for one of them. All right? So that's sort of an example problem set. And you can expect that most of the problem sets are going to follow that sort of template. All right. So you'll get a better sense of this by the end of the day today for sure. Or a concrete sense of this, because we'll be done with lecture and you'll see your first problem set.

We're going to be doing a module on sorting and trees. Sorting you now about, sorting a bunch of numbers. Imagine if you had a trillion numbers and you wanted to sort them. What kind of algorithm could use for that?

Trees are a wonderful data structure. There's different varieties, the most common

being binary trees. And there's ways of doing all sorts of things, like scheduling, and sorting, using various kinds of trees, including binary trees. And we have a problem set on simulating a logic network using a particular kind of sorting algorithm in a data structure. That is going to be your second problem set.

And more quickly, we're going to have modules on hashing, where we do things like genome comparison. In past terms we compared a human genome to a rat genome, and discovered they were pretty similar. 99% similar, which is kind of amazing.

But again, these things are so large that you have to have efficiency in the comparison methods that you use. And you'll find that if you don't get the complexity low enough, you just won't be able to complete-- your program won't be able to finish running within the time that your problem set is do. Right? Which is a bit of a problem.

So that's something to keep in mind as you test your code. The fact is that you will get large inputs to run your code. And you want to keep complexity in mind as you're coding and thinking about the pseudocode, if you will, of your algorithm itself.

We will talk about numerics. A lot of the time we talk about such large numbers that 32 bits isn't enough. Or 64 bits isn't enough to represent these numbers. These numbers have thousands of bits.

A good example is RSA encryption, that is used in SSL, for example. And when you go-- use https on websites, RSA is used at the back end. And typically you work with prime numbers that are thousands of bits long in RSA.

So how do you handle that? How does Python handle that? How do you write algorithms that can deal with what are called infinite precision numbers? So we have a module on numerics in the middle of the term that talks about that.

Graphs, really a fundamental data structure in all of computer science. You might have heard of the famous Rubik's cube assignment from . 006 a 2 by 2 by 2 Rubik's cube. What's the minimum number of moves necessary to go from a given starting

configuration to the final end configuration, where all of the faces-- each of the faces has uniform color? And that can be posed as a graph problem.

We'll probably do that one this term. In previous terms we've done other things like the 15 puzzle. And so some of these are tentative. We definitely know what the first problem set is like, but the rest of them are, at this moment, tentative.

And to finish up shortest paths. Again in terms past we've asked you to write code using a particular algorithm that finds the shortest path from Caltech to MIT. This time we may do things a little bit differently. We were thinking maybe we'll give you a street map of Boston and go figure out if Paul Revere used the shortest path to get to where he was going, or things like that. We'll try and make it fun.

Dynamic programming is an important algorithm design technique that's used in many, many problems. And it can be used to do a variety of things, including image compression. How do you compress an image so the number of pixels reduces, but it still looks like the image that you started out with, that had many more pixels? All right? So you could use dynamic programming for that.

And finally, advanced topics, complexity theory, research and algorithms. Hopefully by now-- by this time in the course, you have been sold on algorithms. And most, if not all of you, would want to pursue a carrier in algorithms. And we'll give you a sense of what else is there. We're just scratching the surface in this class, and there's many, many classes that you can possibly take if you want to continue in-- to learn about algorithms, or to pursue a career in algorithms. All right?

So that's the story of the class, or the synopsis of the class. And I encourage you to go spend a few minutes on the website. In particular please read the collaboration policy, and get a sense of what is expected of you. What the rules are in terms of doing the problem sets. And the course grading break down, the grading policies are all listed on the website as well. All right.

OK. So let's get started. I want to talk about a specific problem. And talk about algorithms for a specific problem. We picked this problem, because it's so easy to

understand. And they're fairly straightforward algorithms that are not particularly efficient to solve this problem. And so this is a, kind of, a toy problem. But like a lot of toy problems, it's very evocative in that it points out the issues involved in designing efficient algorithms.

So we'll start with a one dimensional version of what we call peak finding. And a peak finder is something in the one dimensional case. Runs on an array of numbers. And I'm just putting--

--symbols for each of these numbers here. And the numbers are positive, negative. We'll just assume they're all positive, it doesn't really matter. The algorithms we describe will work.

And so we have this one dimensional array that has nine different positions. And a through i are numbers. And we want to find a peak. And so we have to define what we mean by a peak. And so, in particular, as an example, position 2 is a peak if, and only if, b greater than or equal to a, and b greater than or equal to c.

So it's really a very local property corresponding to a peak. In the one dimensional case, it's trivial. Look to your left. Look to your right. If you are equal or greater than both of the elements that you see on the left and the right, you're a peak. OK?

And in the case of the edges, you only have to look to one side. So position 9 is a peak if i greater than or equal to h. So you just have to look to your left there, because you're all the way on the right hand side. All right? So that's it.

And the statement of the problem, the one dimensional version, is find the peak if it exists. All right? That's all there is to it.

I'm going to give you a straightforward algorithm. And then we'll see if we can improve it. All right? You can imagine that the straightforward algorithm is something that just, you know, walks across the array. But we need that as a starting point for building something more sophisticated.

So let's say we start from left and all we have is one traversal, really.

So let's say we have 1, 2, and then we have n over 2 over here corresponding to the middle of this n element array. And then we have n minus 1, and n. What I'm interested in doing is, not only coming up with a straightforward algorithm, but also precisely characterizing what its complexity is in relation to n, which is the number of inputs.

Yeah? Question?

AUDIENCE: Why do you say if it exists when the criteria in the [INAUDIBLE] guarantees [INAUDIBLE]?

PROFESSOR: That's exactly right. I was going to get to that. So if you look at the definition of the peak, then what I have here is greater than or equal to. OK? And so this-- That's a great question that was asked. Why is there "if it exists" in this problem?

Now in the case where I have greater than or equal to, then-- this is a homework question for you, and for the rest of you-- argue that any array will always have a peak. OK? Now if you didn't have the greater than or equal to, and you had a greater than, then can you make that argument? No, you can't. Right? So great question.

In this case it's just a question-- You would want to modify this problem statement to find the peak. But if I had a different definition of a peak-- and this is part of algorithmic thinking. You want to be able to create algorithms that are general, so if the problem definition changes on you, you still have a starting point to go attack the second version of the problem. OK?

So you could eliminate this in the case of the greater than or equal to definition. The "if it exists", because a peak will always exist. But you probably want to argue that when you want to show the correctness of your algorithm. And if in fact you had a different definition, well you would have to create an algorithm that tells you for sure that a peak doesn't exist, or find a peak if it exists. All right? So that's really the

general case.

Many a time it's possible that you're asked to do something, and you can't actually give an answer to the question, or find something that satisfies all the constraints required. And in that case, you want to be able to put up your hand and say, you know what? I searched long and hard. I searched exhaustively. Here's my argument that I searched exhaustively, and I couldn't find it. Right?

If you do that, you get to keep your job. Right? Otherwise there's always the case that you didn't search hard enough. So it's nice to have that argument. All right? Great. Thanks for the question. Feel free to interrupt. Raise your hand, and I'm watching you guys, and I'm happy to answer questions at any time.

So let's talk about the straightforward algorithm. The straightforward algorithm is something that starts from the left and just walks across. And you might have something that looks like that. All right? By that-- By this I mean the numbers are increasing as you start from the left, the peak is somewhere in the middle, and then things start decreasing. Right?

So in this case, you know, this might be the peak.

You also may have a situation where the peak is all the way on the right, you started from the left. And it's 1, 2, 3, 4, 5, 6, literally in terms of the numbers. And you're going to look at n elements going all the way to the right in order to find the peak.

So in the case of the middle you'd look at n over 2 elements. If it was right in the middle. And the complexity, worst case complexity--

--is what we call theta n. And it's theta n, because in the worst case, you may have to look at all n elements. And that would be the case where you started from the left and you had to go all the way to the right.

Now remember theta n is essentially something that's says of the order of n. So it gives you both the lower bound and an upper bound. Big [? O ?] of n is just upper

bound. And what we're saying here is, we're saying this algorithm that starts from the left is going to, essentially, require in the worst case something that's a constant times n. OK?

And you know that constant could be 1. You could certainly set things up that way. Or if you had a different kind of algorithm, maybe you could work on the constant. But bottom line, we're only concerned, at this moment, about as asymptotic complexity. And the asymptotic complexity of this algorithm is linear. All right? That make sense?

OK. So someone help me do better. How can we do better? How can we lower the asymptotic complexity of a one dimensional peak finder? Anybody want to take a stab at that? Yeah? Back there.

AUDIENCE: Do a binary search subset. You look at the middle, and whatever is higher-- whichever side is higher, then cut that in half, because you know there's a peak.

PROFESSOR: On--

AUDIENCE: For example if you're in the middle on the right side-- there's a higher number on the right side-- then you would just look at that, because you know that your peak's somewhere in there. And you continue cutting in half.

PROFESSOR: Excellent! Excellent! That's exactly right. So you can-- You can do something different, which is essentially try and break up this problem. Use a divide and conquer strategy, and recursively break up this one dimensional array into smaller arrays. And try and get this complexity down. Yeah?

AUDIENCE: Are we assuming that there's only one peak?

PROFESSOR: No, we're not.

AUDIENCE: OK.

PROFESSOR: It's find a peak if it exists. And in this case it's, "find a peak", because of the definition. We don't really need this as it was discussed. All right? OK. So--

So that was a great answer, and-- You know this class after while is going to get boring. Right? Every class gets boring. So we, you know, try and break the monotony here a bit. And so-- And then the other thing that we realized was that these seats you're sitting on-- this is a nice classroom-- but the seats you're sitting on are kind of hard. Right?

So what Eric and I did was we decided we'll help you guys out, especially the ones who are-- who are interacting with us. And we have these--

[LAUGHTER]

--cushions that are 6.006 cushions. And, you know, that's a 2 by 2 by 2 Rubik's cube here. And since you answered the first question, you get a cushion. This is kind of like a Frisbee, but not really. So--

[LAUGHTER]

I'm not sure-- I'm not sure I'm going to get it to you. But the other thing I want to say is this is not a baseball game. Right? Where you just grab the ball as it comes by. This is meant for him, my friend in the red shirt. So here you go. Ah, too bad. All right. It is soft. So, you know, it won't-- it won't hurt you if hits you.

[LAUGHTER]

All right. So we got a bunch of these. And raise your hands, you know, going to ask-- There's going to be-- I think-- There's some trivial questions that we're going to ask just to make sure you're awake. So an answer to that doesn't get you a cushion. But an answer like-- What's your name?

AUDIENCE: Chase.

PROFESSOR: Chase. An answer like Chase just gave is-- that's a good answer to a nontrivial question. That gets you a cushion. OK? All right, great.

So let's put up by Chase's algorithm up here. I'm going to write it out for the 1D version.

So what we have here is a recursive algorithm.

So the picture you want to keep in your head is this picture that I put up there. And this is a divide and conquer algorithm. You're going to see this over and over-- this paradigm-- over and over in 6.006.

We're going to look at the n over 2 position.

And we're going to look to the left, and we're going to look to the right. And we're going to do that in sequence. So--

--if a n over 2 is less than a n over 2 minus 1, then--

--only look at the left half.

1 through n over 2 minus 1 to look for peak-- for a peak.

All right? So that's step one. And you know I could put it on the right hand side or the left hand side, doesn't really matter. I chose to do the left hand side first, the left half.

And so what I've done is, through that one step, if in fact you have that condition-- a n over 2 is less than a n over 2 minus 1-- then you move to your left and you work on one half of the problem.

But if that's not the case, then if n over-- n over 2 is less than a over n over-- n by 2 plus 1, then only look at n over 2 plus 1 through n for a peak. So I haven't bothered writing out all the words. They're exactly the same as the left hand side. You just look to the right hand side.

Otherwise if both of these conditions don't fire, you're actually done. OK? That's actually the best case in terms of finishing early, at least in this recursive step.

Because now the n over 2 position is a peak.

Because what you found is that the n over 2 position is greater than or equal to both of its adjacent positions, and that's exactly the definition of a peak. So you're done. OK? So all of this is good.

You want to write an argument that this algorithm is correct. And I'm not going to bother with that. I just wave my hands a bit, and you all nodded, so we're done with that. But the point being you will see in your problem set a precise argument for a more complicated algorithm, the 2D version of this.

And that should be a template for you to go write a proof, or an argument, a formal argument, that a particular algorithm is correct. That it does what it claims to do. And in this case it's two, three lines of careful reasoning that essentially say, given the definition of the peak, that this is going to find a peak in the array that you're given. All right?

So we all believe that this algorithm is correct. Let's talk now about the complexity of this algorithm. Because the whole point of this algorithm was because we didn't like this theta n complexity corresponding to the straightforward algorithm. So it'd like to do better.

So what I'd like to do is ask one of you to give me a recurrence relation of the kind, you know, T of n equals blah, blah, blah. That would correspond to this recursive algorithm, this divide and conquer algorithm. And then using that, I'd like to get to the actual complexity in terms of what the theta of complexity corresponds to. Yeah? Back there?

AUDIENCE: So the worst case scenario if T of n is going to be some constant amount of time--

PROFESSOR: Yep.

AUDIENCE: --it takes to investigate whether a certain element is [INAUDIBLE], plus--

[COUGH]

--T of n over 2?

PROFESSOR: Great. Exactly right. That's exactly right. So if you look at this algorithm and you say, from a computation standpoint, can I write an equation corresponding to the execution of this algorithm? And you say, T of n is the work that this algorithm does on-- as input of size n. OK?

Then I can write this equation. And this theta 1 corresponds to the two comparisons that you do looking at-- potentially the two comparisons that you do-- looking at the left hand side and the right hand side. So that's-- 2 is a constant, so that's why we put theta 1. All right?

So you get a cushion, too. Watch out guys. Whoa! Oh actually that wasn't so bad. Good. Veers left, Eric. Veers left.

So if you take this and you start expanding it, eventually you're going to get to the base case, which is T of 1 is theta 1. Right? Because you have a one element array you just for that array it's just going to return that as a peak.

And so if you do that, and you expand it all the way out, then you can write T of n equals theta 1 plus theta 1. And you're going to do this log to the base 2 of n times. And adding these all up, gives you a complexity theta log 2 of n. Right?

So now you compare this with that. And there's really a huge difference. There's an exponential difference. If you coded up this algorithm in Python-- and I did-- both these algorithms for the 1D version-- and if you run it on n being 10 million or so, then this algorithm takes 13 seconds. OK? The-- The theta 10 algorithm takes 13 seconds. And this one takes 0.001 seconds. OK? Huge difference.

So there is a big difference between theta n and theta log n. It's literally the difference between 2 raised to n, and n. It makes sense to try and reduce complexity as you can see, especially if you're talking about large inputs. All right? And you'll see that more clearly as we go to a 2D version of this problem. All right?

So you can't really do better for the 1D. The 1D is a straightforward problem. It gets a little more interesting-- the problems get a little-- excuse me, the algorithms get a little more sophisticated when we look at a 2D version of peak finding. So let's talk about the 2D version.

So as you can imagine in the 2D version you have a matrix, or a two dimensional array. And we'll say this thing has n rows and m columns.

And now we have to define what a peak is. And it's a hill. It's the obvious definition of a peak. So if you had a in here, c, b, d, e. Then as you can guess, a is a 2D peak if, and only if, a greater than or equal to b; a greater than or equal to d, c and e. All right? So it's a little hill up there.

All right? And again I've used the greater than or equal to here, so that's similar to the 1D in the case that you'll always find a peak in any 2D matrix.

Now again I'll give you the straightforward algorithm, and we'll call it the Greedy Ascent algorithm.

And the Greedy Ascent algorithm essentially picks a direction and, you know, tries to follow that direction in order to find a peak. So for example, if I had this particular--

--matrix; 14, 13, 12, 15, 9, 11, 17--

Then what might happen is if I started at some arbitrary midpoint-- So the Greedy Ascent algorithm has to make choices as to where to start. Just like we had different cases here, you have to make a choice as to where to start.

You might want to start in the middle, and you might want to work your way left first. Or you're going to all-- You just keep going left, our keep going right. And if you hit an edge, you go down. So you make some choices as to what the default traversal

directions are.

And so if you say you want to start with 12, you are going to go look for something to left. And if it's greater than, you're going to follow that direction. If it's not, if it's less, then you're going to go in the other direction, in this case, for example. So in this case you'll go to 12, 13 , 14, 15, 16, 17, 19, and 20. And you'd find-- You 'd find this peak.

Now I haven't given you the specific details of a Greedy Ascent algorithm. But I think if you look at the worst case possibilities here, with respect to a given matrix, and for any given starting point, and for any given strategy-- in terms of choosing left first, versus right first, or down first versus up first-- you will have a situation where-- just like we had in the 1D case-- you may end up touching a large fraction of the elements in this 2D array. OK?

So in this case, we ended up, you know, touching a bunch of different elements. And it's quite possible that you could end up touching-- starting from the midpoint-- you could up touching half the elements, and in some cases, touching all the elements. So if you do a worst case analysis of this algorithm-- a particular algorithm with particular choices in terms of the starting point and the direction of search-- a Greedy Ascent algorithm would have theta n m complexity. All right? And in the case where n equals m, or m equals n, you'd have theta n squared complexity. OK?

I won't spend very much time on this, because I want to talk to you about the divide and conquer versions of this algorithm for the 2D peak. But hopefully you're all with me with respect to what the worst case complexity is. All right? People buy that? Yeah. Question back there.

AUDIENCE: Can you-- Is that an approximation? Or can you actually get to n times m traversals?

PROFESSOR: So there are specific Greedy Ascent algorithms, and specific matrices where, if I give you the code for the algorithm, and I give you a specific matrix, that I could

make you touch all of these elements. That's correct. So we're talking about worst case. You're being very paranoid when you talk about worst case complexity. And so I'm-- hand waving a bit here, simply because I haven't given you the specifics of the algorithm yet. Right?

This is really a set of algorithms, because I haven't given you the code, I haven't told you where it starts, and which direction it goes. But you go, do that, fix it, and I would be the person who tries to find the worst case complexity. Suddenly it's very easy to get to theta n m in terms of having some constant multiplying n times m. But you can definitely get to that constant being very close to 1. OK? If not 1.

All right. So let's talk about divide and conquer. And let's say that I did something like this, where I just tried to jam the binary search algorithm into the 2D version. All right?

So what I'm going to do is--

--I'm going to pick the middle column, j equals m over 2. And I'm going to find a 1D peak using whatever algorithm I want. And I'll probably end up using the more efficient algorithm, the binary search version that's gone all the way to the left of the board there. And let's say I find a binary peak at (i, j). Because I've picked a column, and I'm just finding a 1D peak.

So this is j equals m over 2. That's i. Now I use (i,j). In particular row i as a start--

--to find a 1D peak on row i.

And I stand up here, I'm really happy. OK? Because I say, wow. I picked a middle column, I found a 1D peak, that is theta m complexity to find a 1D peak as we argued. And one side-- the theta m--

AUDIENCE: Log n.

PROFESSOR: Oh, I'm sorry. You're right. The log n complexity, that's what this was. So I do have that here. Yeah. Log n complexity. Thanks, Eric.

And then once I do that, I can find a 1D peak on row i. In this case row i would be m wide, so it would be log m complexity. If n equals m, then I have a couple of steps of log n, and I'm done. All right?

Am I done? No. Can someone tell me why I'm not done? Precisely? Yep.

AUDIENCE: Because when you do the second part to find the peak in row i, you might not have that column peak-- There might not be a peak on the column anymore.

PROFESSOR: That's exactly correct. So this algorithm is incorrect. OK? It doesn't work. It's efficient, but incorrect. OK? It's-- You want to be correct. You know being correcting and inefficient is definitely better than being inefficient-- I'm sorry. Being incorrect and efficient. So this is an efficient algorithm, in the sense that it will only take log n time, but it doesn't work.

And I'll give you a simple example here where it doesn't work.

The problem is--

--a 2D peak--

--may not exist--

--on row i. And here's an example of that.

Actually this is-- This is exactly the example of that. Let's say that I started with this row. Since it's-- I'm starting with the middle row, and I could start with this one or that one. Let's say I started with that one. I end up finding a peak. And if this were 10 up here, I'd choose 12 as a peak. And it's quite possible that I return 12 as a peak. Even though 19 is bigger, because 12 is a peak given 10 and 11 up here.

And then when I choose this particular row, and I find a peak on this row, it would be 14. That is a 1D peak on this row. But 14 is not a 2D peak. OK? So this particular example, 14 would return 14. And 14 is not a 2D peak. All right?

You can collect your cushion after the class.

So not so good. Look like an efficient algorithm, but doesn't work. All right? So how can we get to something that actually works?

So the last algorithm that I'm going to show you-- And you'll see four different algorithms in your problem set--

--that you'll have to analyze the complexity for and decide if they're efficient, and if they're correct. But here's a-- a recursive version that is better than, in terms of complexity, than the Greedy Ascent algorithm. And this one works.

So what I'm going to do is pick a middle column. j equals m over 2 as before. I'm going to find the global maximum on column j. And that's going to be at (i, j).

I'm going to compare (i comma j minus 1), (i comma j), and (i,j plus 1). Which means that once I've found the maximum in this row, all I'm going to look to the left and the right, and compare.

I'm going to pick the left columns. If (i comma j minus 1) is greater than (i comma j)-- and similarly for the right.

And if in fact I-- either of these two conditions don't fire, and what I have is (i comma j) is greater than or equal to (i comma j minus 1) and (i comma j plus 1), then I'm done. Just like I had for the 1D version. If (i comma j) is greater than or equal to (i comma j minus 1), and (i comma j plus 1), that implies (i, j) is a 2D peak. OK?

And the reason that is the case, is because (i comma j) was the maximum element in that column. So you know that you've compared it to all of the adjacent elements, looking up and looking down, that's the maximum element. Now you've look at the

left and the right, and in fact it's greater than or equal to the elements on the left and the right. And so therefore it's a 2D peak. OK?

So in this case, when you pick the left or the right columns-- you'll pick one of them-- you're going to solve the new problem with half the number of columns.

All right? And again, you have to go through an analysis, or an argument, to make sure that this algorithm is correct. But its intuitively correct, simply because it matches the 1D version much more closely. And you also have your condition where you break away right here, where you have a 2D peak, just like the 1D version. And what you've done is break this matrix up into half the size. And that's essentially why this algorithm works.

When you have a single column--

--find the global maximum and you're done. All right? So that's the base case. So let me end with just writing out what the recurrence relation for the complexity of this is, and argue what the overall complexity of this algorithm is. And then I'll give you the bad news.

All right. So overall what you have is, you have something like T of (n, m) equals T of (n, m over 2) plus theta n. Why is that? Well n is the number of rows, m is the number of columns. In one case you'll be breaking things down into half the number of columns, which is m over 2. And in order to find the global maximum, you'll be doing theta n work, because you're finding the global maximum. Right? You just have to scan it-- this-- That's the way-- That's what it's going to take.

And so if you do that, and you go run it through-- and you know that T of (n, 1) is theta n-- which is this last part over here-- that's your base case. You get T of (n, m) is theta of n added to theta of n, log of m times-- log 2 of m times. Which is theta of n-- log 2 of m. All right?

So you're not done with peak finding. What you'll see is at four algorithms coded in

Python-- I'm not going to give away what those algorithms are, but you'll have to recognize them. You will have seen versions of those algorithms already in lecture. And your job is going to be to analyze the algorithms, as I said before, prove that one of them is correct, and find counter-examples for the ones that aren't correct.

The course staff will stick around here to answer questions-- logistical questions-- or questions about lecture. And I owe that gentleman a cushion.