PROFESSOR: It's a bit better, my heart's back in place. I'm glad that not everyone's gone.

AUDIENCE: Well you mean passed out and died from the test?

PROFESSOR: Yeah.

AUDIENCE: OK, no there were a few students left.

PROFESSOR: That's good, that's good. I think once we release the results everyone will calm down, and we'll realize that the mistake was on our part.

AUDIENCE: The mistake?

PROFESSOR: I mean the results are lower than what we thought. And it's because we haven't covered something that I will cover today. So we think-- well we talked about it yesterday, and we think we haven't done enough algorithm design with you guys. So today I have problems, and we're going to come up with solutions.

AUDIENCE: Are you not going to tell us what [INAUDIBLE] number is?

PROFESSOR: It's in the lecture notes. It's actually quite boring. I promise it's really boring.

AUDIENCE: OK.

PROFESSOR: And if you want to do that next time when we'll actually talk about numerics. The thing is numerics are straightforward. Once you learn the algorithms you're not going to come up with a new one. I'm pretty sure about that. Like you're not going to come up with a revolutionary way of adding two numbers.

AUDIENCE: I don't know. You never know.

PROFESSOR: Well you can tell me why. Think about it and then you can tell me why you are not going to.

AUDIENCE: OK, got it.

PROFESSOR: OK so let's start with a problem. We know what sorted arrays look like, right? 1, 3, 5, 6, 7, 9, 12. This is a sorted array.

If we're given a sorted array we know how to find the number in it, right? What's the running time for that.

AUDIENCE: For any given number?

PROFESSOR: Yeah.

AUDIENCE: [INAUDIBLE]

PROFESSOR: Well what's the best way that we know?

AUDIENCE: Logarithm.

PROFESSOR: Log and binary search, right? Well instead of this we're given a shifted array. And shifted means that-- say you're shifting the array by K elements. We're taking these guys. So the array is shifted to the left.

So these K elements end up on the right. These N minus K elements end up on the left. So 7, 9, 12, 1, 3, 5, 6. So still N elements they are shifted by some number K. And I want to find one number, e, in the array.

I don't know K by the way. We just know that it's a shifted array. This is 12. So the first thing you do is figure out how much time you have for this, right?

If you're on a test, you roughly know how much time you have for a problem. If you're on an interview you have to figure out how much time the interviewer is willing to give you for problem. And spend the first, I don't, a third of the time thinking maybe. Come up with the best solution that you can. And then stop there, and start talking.

So the first thing you do, you want to make sure that when you run out of time you have something to say. The most awful thing you can say is, dude I'm going home now. Or leave the answer blank. If you leave the answer blank we're not going to give you points, right? So not good. So what's the worst answer you could give us?

OK worst is a bad term. What's the brute force solution to this? The solution where we don't care about the running time, but we want the correct answer.

AUDIENCE: Look everywhere.

PROFESSOR: Yep. So do a linear search. Pretend we don't know anything about this array. Lose the information that it's a shifted array. Linear search running time order N.

OK so this is something, at least now when your time runs out you have something. You're not going to leave empty handed. Let's start thinking now. Next step.

AUDIENCE: [INAUDIBLE] can we shift it again?

PROFESSOR: So the fact that it's shifted means that-- so originally it was sorted, right? But now instead of it being completely sorted, you have all the elements are shifted to the left. And then so there's a rotation thing going on here. So these elements got out of the array and then they are put back in from the other side.

AUDIENCE: Why do we do that?

PROFESSOR: That's how the info looks like. We don't do it. It was done to us.

AUDIENCE: Oh OK. So now what do we want?

PROFESSOR: We want to find an element--

AUDIENCE: Oh, I see.

PROFESSOR: --despite the fact that the array looks like this.

AUDIENCE: Do we know that the list that we have is shifted already?

PROFESSOR: Yes. So we're promised that this is a shifted array. So it will look like this. But we don't know what K is. If we knew what K is, could we do something fast? What would we do?

AUDIENCE: Yeah, you'd just re-shift [INAUDIBLE].

PROFESSOR: OK so if you re-shifted what's the running time?

AUDIENCE: For order K [INAUDIBLE].

PROFESSOR: OK so if we actually shifted and then do a binary search it's order K plus log N. So for big K's that's not better.

AUDIENCE: Why is it not order N plus log [INAUDIBLE].

PROFESSOR: You can say that since we don't have any promise on K it's N. It's order K if you can shift things out of both N's. With Python this would be order N, just popping out one element is order N in Python.

So this is assuming a smart array. Otherwise if it's Python, good point. It's straight up order N. So now another good point. You have this solution, and you have the brute force solution. They have the same running time.

You run out of time. Which one are you going to code up? Which one are you going to show?

AUDIENCE: The simpler one.

PROFESSOR: The simpler one, excellent. So the reason is, if you're on a test it's probably give the pseudocode, then analyze it. If you're in an interview the guy will ask you OK, what's the running time? Code it up on the board in C, Java, whatever he knows.

So you want to code the simplest solution, because that reduces the chance that you'll have bugs. So that gives you the most points. So this solution shows more insight, but it doesn't have a better run time. Stick to the simple solution.

However if you have this then you have some insight on the problem. So you can

keep going and hope you can come up with a better answer. So if we knew K, one thing we could do is reduce the array to an unshifted array. What's another thing we can do? So I claim that if you know K you can come up with a reasonably easy log N method.

AUDIENCE: If you're doing binary search, like if you just pretend like the array is all together, but if you know K. So let's say you're looking for 6. Then you'd say oh well I'm going to split the array half, but you're actually going to start at K and then split it in half. So it's like you pretend that--

PROFESSOR: So what you want to say is you have a pretend array in your mind, right?

AUDIENCE: Yes. It's [INAUDIBLE] by K.

PROFESSOR: And you want to access the middle element to see if what you're looking for is bigger or smaller. Instead of looking at the middle element here, you look at the middle plus K, right?

AUDIENCE: Yes. Oh, there you go. Plus K.

PROFESSOR: This is one way of doing it, good running time. The problem is it's hard. You'll have to rewrite binary search and hope it works. What I would do, given that I've had a bit of time to think about it, is this is sorted. This is sorted.

So two binary searched are also going to be log N time. Two binary searches, two lines of pseudocode. The running time analysis is pretty simple. Correctness is also pretty simple. And also this gives me some insight on the rest of the problem I claim.

OK so if we have K we can do log N. What if we don't have K? What do we do? Yes?

AUDIENCE: Figure out what K is.

PROFESSOR: All right let's try to find K. We know how to do it if we have K. So let's try to find K. What-- if I want to arrive to a solution that's log N, how much time can I spend on finding K? OK so let's find K in log N time.

AUDIENCE: Binary search for minimum?

PROFESSOR: Binary search-- so I like binary search, because binary search is an algorithm that runs on an array. And that runs in log N time. So if I'm able to make it work I know everything's going to be right in terms of time. So what do you run a binary search for?

AUDIENCE: The smallest number possible? I guess that's kind of going through all of them, though. It doesn't really help.

PROFESSOR: So if you have the min. Sorry, you can speak in one second.

AUDIENCE: Oh we have the min!

PROFESSOR: So no, if you can't have the min. I think it's good insight. So if you knew where the min is, you know this is K, right?

AUDIENCE: Yes.

PROFESSOR: So this is the minimum, that's K. OK, what were you going to say?

AUDIENCE: Oh for just binary search it would not [INAUDIBLE] minimum. I was thinking that if we start at 1 we will see to our right and left. And the point where [INAUDIBLE] are ending is where we have something larger to our right and something smaller to our left.

PROFESSOR: OK so there's a discontinuity here, that's what you're saying, right? So this is sorted. But then at this point this breaks.

AUDIENCE: Yes, we are kind of finding that point where something to the right that's greater than [INAUDIBLE] and something to the left is also greater than [INAUDIBLE].

PROFESSOR: OK so let's see if we can do that. So for binary search you have to go somewhere. So in our case we're trying to get K, right? And we know that it's somewhere between 1 and 10. And what binary search does is it makes a guess.

It says hey, I think it's in the middle of the array. So it will probably guess n over 2. And it makes a guess and you have to tell it was the guess too small, or was the guess too large?

Because this is what allows you to recurse on either the left interval or on the right interval. The problem with a discontinuity is, if I guess here, and if I guess here, I still don't see the discontinuity. So it's good inside, but it's not enough. I need a little bit more. Yes, 2, 3 hands oh wow you guys don't got it?

AUDIENCE: So I think we can arbitrarily take the halfway point instead of subtracting from the first element. And then if it's a negative number, then discontinuity will be in this half. If it's [INAUDIBLE] it will be in the other half. And then you recurse on that.

PROFESSOR: OK. So let's draw this up. So in a sorted array the numbers look like this. In a shifted array we splice it here, and this guy goes to the right. So it's like this and then like this.

So this picture shows me the insight that I had before, that this part is sorted and this part is sorted. The missing part, which I just heard now, is that since the whole array was originally sorted, this guy is smaller than this guy. So if I draw a horizontal line here, I can draw a horizontal line somewhere, and this and this will not cross it.

So this whole thing is taller than this. So by the way, K was where the discontinuity was, right? You said discontinuity. This is K, it's somewhere here. So this is a better.

So if I make my guess and I land somewhere here, I can know that my guess is too big, because it's below the line. If I make my guess and it's somewhere here, I know my guess is too small, because the number that I see here is above the line. Who sets the line? The first element here.

So this is how you look at it graphically. If you don't want to look at it graphically, this was a sorted array. If this is the Kth element, then everything here is smaller than it. So all these guys are smaller than the first element.

OK so honestly who understands the solution? 3, 4, OK. Oh, OK pretty good. Do we

want to code this up, or do we want to look at another problem? OK who wants to look at another problem?

Clear majority, all right. Usually I have to do both choices, because not enough people are paying attention to get this. So I am happy.

AUDIENCE: [INAUDIBLE].

PROFESSOR: Yes. All right so before I start another problem, one thing I want to say. Not only do I have a solution for this problem, but I have a process that allowed me to go from nothing to a few partial solutions. And while I was doing that, I was getting insight and I was making sure that if I run out of time before I have the final solution, I don't walk out of the room empty handed.

So I don't just want to show you the final solution, I want to show you the process. You can look at the notes and see the final solution. That's not everything I want you to get out of this.

OK, problem 2 has a heap. And this is a minimum heap, so it looks like this. So this is a minimum heap, N elements. And I want to extract the kth smallest element in the heap.

So if K equals 3, this is the third smallest element, right? K equals 4, it's this guy. 5, and 6, 1 and 2 are here. OK the good running time that we want, because this is a hard problem so we give you the running time, is K log K. However before we do that I want to hear some brute force solutions.

AUDIENCE: All of them.

PROFESSOR: And? OK you need to sort them first. So this heap is actually an array, right? 2, 5, 7, 6, 8, oh it's 6, 9, 8, sorry. So you're saying sort the array, then K--

AUDIENCE: [INAUDIBLE].

PROFESSOR: OK what's the running time for this?

AUDIENCE: Log N.

PROFESSOR: All right we have a solution. We're not going to leave empty handed. OK let's try to go a bit better. What's another way of going it that'll give me a better running time?

AUDIENCE: You could pop 5 of the K elements off of the--

PROFESSOR: All right so this is a mean heap, right? So it has find min. And find min runs in order log N. So if I call it K times I'm going to get the K smallest elements. By the way heap sort says pop and times, and you'll have all the elements in sorted order.

So we're doing a heap sort, except we stop when we lose interest after K elements. So we're down from N log N to K log N. So I would be interested in hearing a solution that's worse, because it would look like N log K. But shows me more insight.

So by the way this is good. You're already K log N. So K log N, the correct answer is K log K. Small difference, right? It's a logarithm factor.

At least it's not an N factor. If you code this up chances are we're not going to be able to distinguish between this. So you'll never see this on a PSet. So you're almost there.

And this is just applying straight up knowledge that we had before. Let's look at this solution, if anyone sees it. Before we attempt K log K.

AUDIENCE: In another case would we just pop off first K elements, why would that be in log N? Because it's actually an array, so I would think that we'd just take the K time.

PROFESSOR: So this is a heap. If you don't maintain the heaping variant after you do the first pop you're not going to be able to do the second one. OK, cool. So let me give you a hint. How would we find-- if this is an array-- how do I find the minimum?

2, 5, 7, did I forget something? No. Let's pretend this array doesn't start with 2, because it's boring if it starts with 2. How do I find the minimum? I keep one variable that says the best I've seen so far, right? Let's see, N-- oh this is still boring.

Let's start here. So we start with best seen equals 7. Then when we go to 6 we see, is 6 better than best seen? If so, replace best seen with 6. If not keep going.

Then I get to 9. Is 9 better than best seen? Nope, keep going. Is 8 better than best seen? Nope, keep going. So I compare every element with the best seen, and then whenever the element is better I do a replacement. And then at the end, best seen will have the smallest element.

So this algorithm works for k equals 1, which isn't very useful. So can we generalize this somehow to-- so we have a running time here. That might give you a hint about how we want to generalize it, and I want to generalize it for all values of K.

AUDIENCE: If you go to the power of 2 then it's less than-- the nearest power of 2 less than K--

PROFESSOR: OK.

AUDIENCE: --that element, and iterate forward with your best seen. Does that make sense? If you want the kth, if you want the tenth smallest element, then it has to be after the 8th row because it's the next level in the tree. Does that make sense? That doesn't make sense.

PROFESSOR: It makes sense, but I don't think it's right. So you're thinking that the tenth smallest element has to be somewhere below, right?

AUDIENCE: Yeah.

PROFESSOR: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, and then pretend that there are numbers here. 11, 12, 13, 14, 15, do you see what I'm saying? So this is a heap. So I can keep filling it with bigger elements, and 10 is here.

However you can do something else to limit the size of the heap. It will give us a different running time, but you can do something. You can think about it. How can you chop up some of the heap?

For example if I have a heap that's ten deep and I look at the fourth element, what can I do? You can think about that. And let's try to get to this. So I'll accept an

answer for either how do we limit the heap in that case, or how do we generalize this algorithm. Yes?

AUDIENCE: You want us to remember the smallest K elements, you'd make a max heap [INAUDIBLE] K [INAUDIBLE] [INAUDIBLE].

PROFESSOR: OK so I want to have-- I'll break down your solution into parts . So you want to have a bag of the smallest K elements, right? So instead of the best seen, you want to have the K best seen.

And once you have a bag you want to go through your elements. And then if you have something that's better than what you have in the bag, you want to put that in the bag. Suppose I have, suppose K equals 3, and I have 2, 5, and 7 in the bag. And I see 6.

Who do I want to compare it with? The biggest thing in the bag, right? So if I want the K smallest elements, if this guy is smaller than anything, these aren't the K smallest elements anymore. So I want to take the maximum in the bag, compare it with what I'm seeing right now, and if what I'm seeing is smaller I want to replace it.

AUDIENCE: [INAUDIBLE].

PROFESSOR: Because I keep doing this maximum, I keep asking this maximum question, this has to be a max heap. That's why he said max heap. So you did all these steps at once and then gave me the final answer. But this is how you do it step by step.

So it looks like finding the minimum element, except you have a bag. And that bag has to be a maximum heap. And the original heap is a minimum heap. So the fact that you have to use a maximum heap is a bit nontrivial. Good answer.

All right so we have K log N, and we have N log K, so choose what you want to have in your log. We have a solution for you. How about this.

How are we doing here? So suppose I'm looking for the fourth element, and my heap has 10 levels. How can I chop it-- how can I reduce the number of things I'm looking at?

AUDIENCE: You can reduce it down to 4 levels.

PROFESSOR: I can reduce it down to 4 levels, exactly. So this heap has log N levels. And my K is smaller than log N. I can reduce the heap down to K levels and discard everything below.

And the reason for that is we have a mean heap, right? So if we go down from, on any path from the roots to a leaf, the values have to increase, right? Otherwise it's not a mean keep. Otherwise there's an invariant violation somewhere there.

So as I go down on any path my numbers are going up. So these are all the paths of length 4. All of them have to go through here. All the paths of length 4 will stop here.

So I know that everything here has to be bigger than the first 4 elements. So if I reduce this to K and I discard everything else, what's the running time? So if I use my find my extract min algorithm before, what was the running time?

AUDIENCE: It was [INAUDIBLE].

PROFESSOR: So it's not the--

AUDIENCE: Oh sorry it's the--

PROFESSOR: So what's one operation in a heap? If I have the height of a heap, what's an operation? How much time does it take to do one operation as a function of the height of the heap?

So if my heap has h levels, in this case h happens to be log N, it's order h. So if I reduce it-- I'm not reducing it from N to K. I wish I could. I'm reducing it from log N to K.

So for really tiny K's, this becomes order K. And my total running time is K squared. So I'm going to do K operations, K extract mins.

OK now the reason I wanted to entertain this is I claim it's going to be useful to help

us find the answer. So everything that we have here gives us some insight into what the correct answer is. Well what our correct answer is. There might be others. So let's think for a bit, and see if we can do better.

Am I covering something? I hope not. So by the way, when you have problems on your own, say you are looking at CLRS or at old exams, you want to give yourselves half an hour, an hour to think. And just this process alone is going to help you do better on a test.

Because while you're thinking you're going through everything you know. And you're rearranging stuff in your brain in a way that will be easier to access it later. So now you're going to think, what do I know about heaps? What do I know that takes log N time?

What do I know that takes N log N time? And your brain will be better at answering these kinds of questions later. Now we're not going to give you 30 minutes, because that would make us run out of time.

AUDIENCE: You want to reduce it down to K elements.

PROFESSOR: I want to only have to look at K elements, that's good.

AUDIENCE: Otherwise you can't plug K into the search.

PROFESSOR: Yep, OK so that's good.

AUDIENCE: Which is interesting, because it's K log K, and that kind of suggests that you'd have K elements in the tree. But then you're searching for each one in the tree.

PROFESSOR: So maybe I'm not going to be able to cut this heap into K elements, right? I'll have to do a bit more.

AUDIENCE: Can you cut this heap into K elements and use that heap to do our [INAUDIBLE]?

PROFESSOR: Let's see how we'd cut this heap. First off let's see how this keep would look like if it's cut. How do we find the first K elements here? How do we find the first element?

AUDIENCE: [INAUDIBLE].

PROFESSOR: It's the root. Second element.

AUDIENCE: [INAUDIBLE].

PROFESSOR: What do I look at? If I want to select the second element in a heap, how many elements do I have to look at? Two, 5 and 7, because everything below will be bigger, right? OK I look at them, I compare them, I know 5 is the smallest one.

Now suppose I want to find the third element. Who do I look at? 7 or the thing under 5. So 7 is still in the race for sure. And then I have to look at the children of 5. Right now we're looking at 3.

Suppose this has some really large kids. As in numbers. And I find that this is the third element. Who do I look at for the fourth element?

AUDIENCE: [INAUDIBLE].

PROFESSOR: OK so this isn't in the race anymore, because it's the third. The fourth has to be either of these two guys. Or the kids here, right?

And it happens to be 7. So I take it out. If I want to look at the-- if I want to find the next element, who's in the race? This guy gets out of the race.

AUDIENCE: 7's kids. [INAUDIBLE].

PROFESSOR: OK so we have something. We're not really cutting up the heap, but we are sort of computing where the blade would go if we wanted to cut it up in K elements and N minus K elements. Does this make some sense? Nods, no nods.

AUDIENCE: I mean I guess you're never going to be going down farther than K.

PROFESSOR: So let's just understand the concept. And then we're going to do one more pass, write pseudocode, and understand the running time. Because this is still confusing, right? We'll need one more pass, otherwise we can't write the pseudocode into it. So does the concept make sense?

AUDIENCE: Is that K log K?

PROFESSOR: Yes. So the idea here is that I have a horizon that says, what are the next elements that I'm willing to consider? And first the horizon starts with just the root, because I know that's the minimum element. And when I take an element out of the horizon I put in its children. That's what I did all the time. So given a horizon how do I know what the next elements to extract out of the horizon?

AUDIENCE: [INAUDIBLE].

PROFESSOR: The mean, OK. So I want a data structure for the horizon that can extract means quickly. OK what am I going to use for the horizon? A min heap, excellent. So let's try to go for pseudocode.

Suppose we have H as our original heap. So H is a mean heap. We will make Z be our horizon. I can't use H again. It would be nice if I could, but I'll use Z because Z's also a letter in horizon. So Z's the mean heap.

And then first I will insert into Z. I'll insert the heap's root, right? So Z dot insert H of 1. Remember that heaps are actually arrays. I hinted to his earlier. So these nodes have are elements in an array.

So this is the first element, second, third, fourth, fifth, sixth. So we're using array backed heaps, and H of one is going to be the root. Then I'm going to compute the first K elements like this, for K in range-- sorry, for i in range K, so K is going to go from 1 to K.

What I want to do? Take, compute the ith element. How do I do that?

AUDIENCE: Extract min.

PROFESSOR: i equals Z dot extract min. And then I want to insert the children in the horizon. Right? How do I do that?

AUDIENCE: 2i and 2i plus 1.

PROFESSOR: OK so this is if I know the index, right? When I'm putting things in the heap the keys are going to be the values, so that I can take out the minimum.

AUDIENCE: [INAUDIBLE] heap first and then inserted H Y.

PROFESSOR: Yeah, OK. This is empty. And this is the input. OK so I need to use the numbers as the keys. So when I extract something out of the heap, so when I extract the first element it's going to say 2, it's not going to say 1. If I want to--

AUDIENCE: Why wouldn't Z dot extract [INAUDIBLE] because--

PROFESSOR: So this will give me the next key in the horizon.

AUDIENCE: But-- oh I see, you're starting out with just the first one.

PROFESSOR: Yeah.

AUDIENCE: Oh and then you want to add in the next.

PROFESSOR: So at the end of this whole thing, if I'm extracting them right, I can return this variable here. Because after K durations this is going to be the Kth element. So I return it and I'm done. The problem is I want this guy's index too, right?

So I can't just store the key in the heap. I have to augment the heap to let me store values. And I have to store the index.

So for this guy would have Z insert H of 1, and then it's index 1. Then when I get out the ith element I'll also get out it's index. A variable name for that?

AUDIENCE: j.

PROFESSOR: j. OK why would you name your variables like this? In the previous section I had a similar suggestion, i, i. So why would you name your variables like this?

AUDIENCE: [INAUDIBLE].

PROFESSOR: Job security. All right. So it's OK here. Try to not to do that when doing an exam or

an interview, because it reflects poorly on you. For an interview and for an exam you'll get us upset and we might be less lenient. Or at least explain what you're doing.

So extract min is going to give us the key. And it's going to give us index in the heap. What do we do afterwards?

AUDIENCE: We add to [INAUDIBLE] H of--

PROFESSOR: All right so when we take out 2-- so we start out with an horizon of 2. When we take it out 2's the only thing that's in the horizon first. Then we take it out and its two children get in the horizon.

Then we take out one of the children and put its children in the horizon. So when we take out a node we want to put its children in the horizon. So we're going to say Z dot--

AUDIENCE: Insert.

PROFESSOR: Insert. What do I insert?

AUDIENCE: H of I times 2. j times 2.

PROFESSOR: See? It's working already. The job security thing is working. And?

AUDIENCE: 2j plus 1. You have to do two lines.

PROFESSOR: OK, sweet. OK. Does this work? I mean does this do what we wanted it to do earlier?

AUDIENCE: Wait, we're extracting oh--

PROFESSOR: All right first nod.

AUDIENCE: I mean if K is small enough. Eventually you'll ask for something that is out of range.

PROFESSOR: Oh so you're thinking that eventually these will run out of range.

AUDIENCE: If you have your really lopsided array eventually you'll ask for something that's [INAUDIBLE].

PROFESSOR: OK what would we want to do in that case?

AUDIENCE: Just want to check to make sure that the [INAUDIBLE].

PROFESSOR: Yeah, but otherwise move on, right? If an element doesn't have kids, we don't add on to the horizon. So we need some bounce checks, exception checking, things like that in here.

And I won't add that because that will make it look long and ugly. So this is the idea. OK what's the running time for this?

AUDIENCE: [INAUDIBLE].

PROFESSOR: Cool. So creating heaps, initializing, all order 1, insertion, this heap is almost empty now. So this is order 1. Then these happen K times. And these are all operations on the heap Z.

And the heap for the heap Z, it has some number of elements. And it's not always going to have one element, because every time I'm extracting one element I'm adding two. So well how many elements is it going to have at most?

AUDIENCE: K.

PROFESSOR: OK why is that?

AUDIENCE: Because each time you add it it's one element.

PROFESSOR: So I extract one for sure. And then I add at most two elements. So the heap size grows by at most 1 in every iteration. So the heap size Z will have at most K elements.

So now I know the running for all these operations. What is it? Log K. Cool.

So it's K times log K. And the reason that it works, it's a bit harder to see. You have

to convince yourself maybe using this bigger tree, that whenever you're spending expanding the horizon you're expanding it the right way.

So the idea is again that whatever path you take down you're going to see ascending numbers. So when you're increasing the horizon you're always pushing it down in such a way that your invariant is that all the numbers in the horizon are smaller then their children. And so on and so forth.

So the horizon is always guaranteed to have the smallest number that you haven't extracted yet. And that's really the only thing you need. OK does this make some sense?

AUDIENCE: It never would have occurred to me on an exam.

PROFESSOR: Yeah exactly. This would not occur on an exam unless you think a lot, you're super inspired, all that. If it doesn't occur to you what do you do?

AUDIENCE: Go with the N log K solution.

PROFESSOR: OK, very good. Wait.

AUDIENCE: K log N.

PROFESSOR: OK, K log N or N log K, which one?

AUDIENCE: K log N.

PROFESSOR: Why? Two reasons. K log N is-- so two reasons, faster and simpler. So you write this down. And you get half score or 3/4 of the score and you're done. It's better than nothing, anything, and getting a 0, right?

I mean 3/4 of a score for two lines of pseudocode is reasonable, right? Two or three lines. This is three lines probably.

Also on most exams we're humans, right? We might mess them up, we might make them too long. If we make them too long, you want to get the most number of points. You'll have time to figure out one or two problems at that level. But if we give

you too many, for the rest of them you want to have something simple that gives you some of the points.

Same for an interview. For most interviews most people don't really have a clue how many problems you can solve, how many problems are reasonable. So you want, for every problem you want to show some solution reasonably fast.

And then see if they're happy. And if they're happy move on to the next problem. And if they're not happy only then spend more time.

So this is as important as that. If you look at the recitation notes we'll have some problems and we'll have some solutions. What are going to do, memorize the solutions? Yay, you know how to solve more problems. There are probably a million problems in total. That doesn't get you very far.

So what you want is to understand this process that we went through. So every time we tried something we got from some point to some point with a better running time. Well except for here. And where we had more insight on the problem. So this is the important part.

And I'm going to show you one more problem, really quickly. We're probably not going to be able to solve it, because it's hard. But we are going to talk about it and see if we can get some insight. Let's see, what do I want to erase? This. I like that.

All right, so we have an array random numbers, 7, 2, 5-- this time there's no order in it-- 8, 9, 4. And we tell you that the array has 2 to the N numbers, to make the problem easier. 1, 2, 3, 4, 5, 6, 7. 6.

So you have this array. And we want to answer queries of this shape. Say this array is E, and it has N elements, and you know that N is some 2 to the K. Minimum of all the elements from i to j. So you have two phases, just like we had on a problem on the exam.

You have a pre-processing stage where you get the array, you do some computation, you save some information. And then you have a querying phase,

where you have to answer these as fast as possible. I see most people have unhappy faces. Bad memories, huh?

OK let's not worry about that problem. Let's look at this one. So assuming you have as much time as you want to do the pre-processing, what's the fastest way you could answer these? Yes?

AUDIENCE: If you had as much time for pre-processing [INAUDIBLE] memorize it.

PROFESSOR: All right. So if we compute the answers to all possible solutions, right? How would I store that? So I want to do this in order 1. So how would I store these answers?

AUDIENCE: Just sort your array.

PROFESSOR: OK so I sort my array.

AUDIENCE: Then you want the minimum from i to j, so look at the ith element and that's your [INAUDIBLE].

PROFESSOR: OK so figure it out? Well I mean if I can sort it I can also say hey, why don't we use this array instead? And then I'll answer the queries.

You can go off a tangent trying to sort the elements and keep their keys. The important thing is if you think about it for awhile and you see that things stop making sense, back out. Look somewhere else.

We spent some time trying to find a solution based on sorting in my last section. It's not going to work. So--

AUDIENCE: Can't you just take the [INAUDIBLE] from i to j?

PROFESSOR: OK let's get to that in a bit. So let's keep that in mind. Because that's another point on the trade off curve. So if I want to serve my queries in order 1, then the way I do that is I will have a hash of all the arrays that look like i, j. So all the possible intervals.

And I'll store the answer here. The minimum the elements from i to j. And I can do a

hash lookup in order 1 and get the answer and return the answer. How many elements so I have here? So how much storage do I have to use for this?

AUDIENCE: O of N squared.

PROFESSOR: OK N values for this, N values for this, so roughly N squared. What's the time for computing this? Brute force, let's not think. What's the time for computing this?

AUDIENCE: N cubed.

PROFESSOR: N cubed. You're thinking. So I have unsquared elements here. For every element I have to compute the minimum of potentially order N elements, right?

So this is N cubed. I could reduce it to N squared by noticing that if I have the minimum of these elements, and I want to compute the minimum of these elements, really all I have to do is compute, compare this minimum with this element. So every time I start with an interval of size 1 and then I expand it by 1.

So I have my two for loops here. And I keep growing my minimum. So I could get down to order of N squared times.

So I have one solution that has order of N squared time and space, and then answers the queries in order 1. You had a solution you said where, what you do is, when you get a query you compute this, right? You were suggesting sorting the array.

That would be N log N. I would suggest not sorting it. Do the splicing, you look through all the elements, and you find the minimum.

AUDIENCE: I was saying that if the original E spans i to j and started at the--

PROFESSOR: So when you get a query the i's and j's change for every query. Otherwise we could compute the answer. So we have one answer where we take order N time to answer a query. And what do we do for pre-processing?

Nothing. Order 1. So these are two ends of a trade off, right? One possible extreme

is that you pre-compute all your answers. The other possible extreme is that you don't do anything and you brute force every answer. And now we want to find a point somewhere on this line between the extremes.

So the answer that we're going to show in the solutions uses order N log N space. And it answers the query by using order 1 elements in this order N log N data structure. So I have order N log N partial minima. And I will only use two of them.

So the total running time isn't actually order 1. But we only use order 1 elements. Let's start thinking very quickly. Let's think for about a minute, and then we'll go through the solution.

And there are multiple solutions. All of them are interesting in different ways. And there are other solutions that are equally fun and applicable with not the same running time.

Let me make some space here. So like I said, thinking is a useful process on its own. So you're getting better just by doing this.

AUDIENCE: [INAUDIBLE] using more than one space total?

PROFESSOR: We're using N log N space.

AUDIENCE: Oh and it takes constant time--

PROFESSOR: It will only look at two elements. It's actually not constant time. We're not going to worry too much about time. It turns out being log.

AUDIENCE: Ok, what was the order 1 then?

PROFESSOR: You only access order 1 elements. Order 1 partial minimum.

AUDIENCE: Oh, OK. Does it have to do with two different K?

PROFESSOR: Maybe.

AUDIENCE: I don't know what to do with that. There's probably some sort of tree involved.

PROFESSOR: So you're going to want to split things, right? Into halves. And you're going to want to be able to do this all the time. And we say 2 to the K so we don't have to worry about, oh my God what happens if the halves aren't equal?

You can usually solve this when you implement the problem. But it's useful to not worry about that when you come up with your first algorithm. If you're going to start dividing in halves.

AUDIENCE: [INAUDIBLE].

PROFESSOR: Um.

AUDIENCE: [INAUDIBLE].

PROFESSOR: So that leads to another useful solution. That leads to a solution that takes-- that has N log N storage and it will run in N log N time with N log N element axes.

So what you thinking of is you're going to have your array of elements, right? And say you want to find the minimum from here to here. You're going to have your array split in half. So you're going to find the minimum of this, and the minimum of this.

But to do that you'll have to recurse. So this is also say split in half. So you'll have to find-- so it turns out that if you do this, in the end you'll have log N minima that you have to look at.

But this is more, this is a cooler and more useful thing, so I'll try to put it on a PSet or something to make you think about it. So this is-- don't tell people yet. You might have a solution to a problem.

AUDIENCE: [INAUDIBLE].

PROFESSOR: OK. So what we thought of, or the way we thought of doing it, is 6, 7, 2, 5, 3, 8, 9, 4. So we compute these partial minima. We split the array into two. And these are the minima that we compute. Sorry, this is like this, this is like this.

So everything, so all the left half then these guys, then these guys, then this guy. Everything here, then these guys, then these guys, then this guy. So if your i and j are on different sides of the middle, then you do two lookups, you're done.

If they're in the same half, then you have a problem that's half the size. So you're going to have to take this array that's half the size, 2, 5. Split it into halves and do the same thing. And then we're going to have to do the same to this other one.

3, 8, 9, 4, split it into halves and do the same thing. So in the end you'll end up in someplace where your interval ages are on different sides of the middle. And you look at two elements and you're done.

Let's see how much space this takes. Can someone tell me a recursion for how much space, for how many minimums I would need to keep? So space for an elements is?

AUDIENCE: The first level you have 8. So go down by an order of 2.

PROFESSOR: So what's the first level?

AUDIENCE: Of 8 N.

PROFESSOR: So order N plus?

AUDIENCE: N over 2? t of N over 2?

PROFESSOR: OK S because it's space. N over 2. OK. And?

AUDIENCE: [INAUDIBLE].

PROFESSOR: You're missing something. Look at this picture. So this is the whole thing. Then I have a half. And then what else do I have?

AUDIENCE: 2.

PROFESSOR: The other half.

AUDIENCE: Oh, 2.

PROFESSOR: OK so the difference between these two is that one of them gives you order N, the other one gives you N log N. So I gave you the answer, so I can't ask you for the answer now. But where did we see this before? Pretend these are t's.

AUDIENCE: [INAUDIBLE].

PROFESSOR: Sorry? So these are t's, this is the recursion for [? more sort ?]. So once you put it up you don't draw the recursion tree and solve it. You say this is what we saw in [? more sort. ?] Therefore, the solution is N log N. So this is how you show you have N log N space, and it's pretty clear that you're only going to access two elements.

AUDIENCE: I don't understand how [INAUDIBLE].

PROFESSOR: How it works? So you have your i and you have your j. Let's make that one i here. If you want to find the minimum, if i and j are on different sides of the half, you have this and this.

And these two partial minima cover your entire interval. Now if they're on the same side of the half then you recurse to a smaller problem.

AUDIENCE: Well you don't have to there because you already have the minimum of that section.

PROFESSOR: Yeah.

AUDIENCE: It wouldn't work if you had 6 and 2, right? Or that.

PROFESSOR: Yeah.

AUDIENCE: Well why not just take 7 and 2 then? Why do you have to break up the entire panel?

PROFESSOR: Assume there's more things there.

AUDIENCE: Oh I see.

PROFESSOR: So if you have this, now it's no longer true, right? So wherever they are here, you do that. And remember your pseudocode has to be as simple as possible to reduce the probability of bugs. So you want to do the simplest possible thing, not have special

cases.

OK. By the way there's a study that shows that for good or bad programmers, if you have 1,000 lines of code, there's a constant probability of a bug. And the constants are different for good versus bad programmers, but it's still a constant.

So how many mistakes you make is directly proportional to how much you write. This is why we like simple solutions. OK, any questions on this?

So we have four problems. We didn't cover one. Look at the other one, look at the solution. Ideally look at the problem, think for at least half an hour, then look at the solution.

What I want you to take away is not just oh, here are three problems, let's memorize how we solve them. But the whole process thing, and how we played with data structures and how we used all the hints that we possibly could to build more insights into the problem. OK, cool.