PROFESSOR: All right. So I brought a few problems. They're obviously not the quiz problems, though some of them are supposed to be similar. What I have here might not be what you have on the quiz because we might drop quiz problems or because some of them are just meant to make you think and not to give away the solutions to the quiz.

Now, before we get started on this, do you guys have any burning questions or any concepts that you want covered? Based on that, I'll select which problems we do. Yes?

AUDIENCE: This actually relates not too much to the Pset. If you're looking at the time complexity to maybe transfer something from one table to another, it takes a lot more time, I would assume, to move the actual item to the new table than it does just to look at your point and be like, oh, there's nothing there. So if you were just going to look through an empty table of size m, the time to look through that empty table, I'm assuming, is much less than the time to actually move an item.

PROFESSOR: So you're saying we have m things here.

AUDIENCE: Yes.

PROFESSOR: Some might be nil, and some might have stuff in them, and you're going to resize that to presumably 2 times m, and the way you do that is you're going to move the elements, presumably by rehashing them, right?

AUDIENCE: Yes.

PROFESSOR: So these elements, at least when we use Python, we don't really store big elements anywhere. If you have a big object, we always work with references to that object.

So you remember the address where that object lies in memory, and since the memory is finite and small, addresses are all from 0 to a small number, so they're constant. So what you have here is not a big object. It's the address of the big object, so moving is always constant time.

AUDIENCE: What I'm saying is let's say that the table is completely full versus completely empty table. It would take more time to move everything out of the full table than it does just to look the empty table, right?

PROFESSOR: Let's see. So writing something here is order 1 time, right? So moving is order 1 time. Moving one element is order 1 time. What's accessing an element in a table in a list? You have a Python list. What's the cost of doing an index access?

AUDIENCE: It's also order 1, right?

PROFESSOR: OK. So order 1, index. Order 1, move. Suppose you have an empty table. How many indices do you do? How many times the index?

AUDIENCE: You look at each one, so it's order 1 times the length of the table.

PROFESSOR: m. So if the table is empty, you have order m indices and 0 moves. Total running time, order m. If you have a full table, how many times do you index in the table?

AUDIENCE: Still order m.

PROFESSOR: OK. How many times do you move stuff?

AUDIENCE: Order m.

PROFESSOR: Total running time?

AUDIENCE: It's order 2m, which is order m.

PROFESSOR: So it doesn't matter whether the table's full or empty.

AUDIENCE: OK. Just wanted to confirm that.

PROFESSOR: And this is how you do that. Cool. Thanks. Any other questions? Then we will go over problems in the order in which I like them, which is easiest to hardest so that I don't have to explain the hard ones.

Warm up problem one. So you have this recursion and you have to solve it, and you get a hint that n to the power of 1 over log n is 2, which is theta 1. So based on the hint, you can see that it's going to involve some math. It's going to get a bit ugly. So how do we solve recursions? Two methods. What are they?

AUDIENCE: Expand.

PROFESSOR: OK. Substitution formally, but basically, we expand this guy over and over again. And?

AUDIENCE: Trees.

PROFESSOR: Recursion trees. Which one do guys want to do first? If you only have one t here, anything works because you can keep expanding it and that works, so we can do either method. Which one do you guys want to go over? Trees. OK. So we start with the first node. The size of the problem is n. What's the cost inside here?

AUDIENCE: 1.

PROFESSOR: OK. So this creates one sub-problem. What's the size?

AUDIENCE: n to the 1/2.

PROFESSOR: OK. Square root of n equals n to the 1/2. You solved it already. What's the cost?

AUDIENCE: 1.

PROFESSOR: Do people remember this? Is anyone confused about what's going on here? OK. So two terms, something involving t and something not involving t. The thing involving t is what we want to get rid of. When we do our recursion tree, whatever is in here goes inside here, and this tells me how this number relates to this number. So when I go from one level to the next level, this is the transformation. n and becomes square root of n, so the transformation here is the same as the transformation here. What's next?

AUDIENCE: n to the 1/4.

PROFESSOR: OK. Cost?

AUDIENCE: 1.

PROFESSOR: OK. Do we need to do one more or do people see the pattern? Silence means one more. If you guys don't speak, we're going to go slow. What's here?

AUDIENCE: n to the 1/8.

PROFESSOR: What's here?

AUDIENCE: 1.

PROFESSOR: Let's hope everyone saw the pattern, and suppose we've done this for l levels, so we're at the bottom. What should the cost be at the bottom?

AUDIENCE: [INAUDIBLE].

PROFESSOR: Sorry. We don't start with the cost. What should the size of the problem be at the bottom?

AUDIENCE: n to the 1 over 2 to the i.

PROFESSOR: Let's say that this is level h, where h is the height of the tree.

AUDIENCE: Don't you want to do something like n to the 1 over n?

PROFESSOR: Yeah. OK, so we want something that looks like that?

AUDIENCE: If the recursion tree is height i, it is n to the 1 over 2 to the i, but 2 to the i should equal n, or approximately n.

PROFESSOR: Why? I like that, but why?

AUDIENCE: Because you need to go down until you're only looking at one element, and that would be one nth of the problem.

PROFESSOR: OK. So we want this guy to look like what? In fact, it doesn't exactly have to look like 1, but what's the advantage if we manage to get this guy to look like 1? We have a recursion. We don't have a base case here, right? A reasonable base case is T of 1 is theta 1. Whatever function that is, if you evaluate it at 1, you're going to get a constant, so you can say that.

Now, at the same time I can say that for any constant, c, T of c is theta 1. So if I take this constant here, which happens to be 2, but that's not to worry about that. If I take this guy here, I can put it in here. And I know that this guy here equals this guy here. So if I can make this guy here look like this guy here, then I'm done.

Make sense? If it makes sense, everyone should nod so that I know and I can go forward, or smile or something. So this should look like 1. This is order 1 if not 1. Let's make it order 1, because it's 2 in this case. What's the cost here?

AUDIENCE: 1.

PROFESSOR: 1. Everything inside the bubbles is order of already, so I don't need to write an order of. What do we do next?

AUDIENCE: Solve the [INAUDIBLE] equation. 2 to the [INAUDIBLE].

PROFESSOR: You're skipping one step. That's exactly what you do when you have the substitution method. You're going to get to something and you need to solve the equation. But for the tree, there are two steps. So we need to add up all the costs here and that's the total cost here.

In order to do that, first, we sum up over each level. And in this case, it's really simple because there's only one node per level, but if you have multiple nodes per level, you have to sum up for each level, and then you have to do a big sum. What's the sum for this level? 1. Come on, guys. You're scaring me.

AUDIENCE: 1.

AUDIENCE: 1.

AUDIENCE: It's all 1.

PROFESSOR: Excellent. So the only thing that I'm missing is to know how many levels I have, because the sum is going to be order h, whatever h is. How do we do that? n to the 1 over 2 to the power of h has to equal this guy, right?

AUDIENCE: Why would it equal that guy? We know it's less than that guy, but we don't know it's equal to that guy.

PROFESSOR: We have to make it equal because we can only stop when we get to the base case. So we have to expand the recursion tree until we get to a base case, and then we stop, and this is our base case because this is what the problem says should be our base case.

AUDIENCE: Right, but n to the 1 over 2 to the h is not equal to n to the 1 over log n.

PROFESSOR: Well, we can set h to be whatever we want. h is the height of the tree, so we don't know what it is. We have to find out what it is.

AUDIENCE: So let's say 2 to the h is equal to log n if you want to make it look like that.

PROFESSOR: Let me write down the equation to make sure you're right. You're probably right because you're thinking faster than me, but let me not embarrass myself and do this the right way. So you said 2 to the h is log n, right? Looks about right. So what's h?

AUDIENCE: Log base 2.

PROFESSOR: All right. Log log n. So T of n is order h. We got this from here. T of n is order h is order of log log n. Math people drowning, right? Any questions about this? Yes?

AUDIENCE: The first line on the right--

PROFESSOR: This?

AUDIENCE: Yeah. Is that your base case? What is that?

PROFESSOR: We got a hint with the problem that said, n to the power of 1 over log n is 2, which is order 1. So for the base case, we always want them to look like this. If we don't get a base case, we write our own base case, which is if you plug in a constant, you're going to get a constant. And since we're told that this guy is a constant, that's a pretty good hint that we want to get to it.

Let's see how we're doing on time. Good. Ready to move on to the next problem? Let's do a fun one. Some people might remember it from elementary school, but this time, we're going to look at it with our 6.006 eyes.

So suppose you have m coins, gold coins. One of them is face. The fake one is super light because it's not real gold. It's something that looks like gold. And we have a scale, and the scale is super accurate. It can weigh any coins on either side and tell us which side is heavier. Perfect accuracy, no need to worry about errors.

I want to find out which coin is the bad coin. What is the minimum number of experiments I have to do? So there is a strategy, and we can worry about that later, but using 6.006, what is the minimum number of experiments I have to do?

AUDIENCE: Log N times.

PROFESSOR: Not quite. So this is what you think is, and you can do log n with binary search, right? The problem with binary search is if I put half of my coins on the left, half of my coins on the right, one side is going to be heavier, right? So the answers are going to be this or this, but I never get this. I only get one bit of information instead of getting one trit. A trit is a base three digit. How many bits of information in a trit?

AUDIENCE: One and a half.

PROFESSOR: Roughly.

AUDIENCE: Log 3.

PROFESSOR: Log 3. And we know that it's base 2 because that's what we use in CS. So we're discarding a fractional bit of information if we're not allowing for this to happen. Anyone want to try something else? We have to prove this, by the way. We have to prove the minimum that we come up with.

AUDIENCE: You could just do it coin by coin, but that would take forever.

PROFESSOR: That's N. That's worse.

AUDIENCE: How about log base 4 N or something like that?

AUDIENCE: Can you explain to me why we can't just do binary search?

PROFESSOR: We can. It's definitely going to give us the correct answer, but it's not the minimum number of weighings because we're discarding a possible answer. So if you do binary search, you will never get that the two sides are equal.

AUDIENCE: Log base 3.

PROFESSOR: Log base 3 would be better because we have three choices all the time. Let's prove that. So the right answer happens to be log base 3 of N. Let's see how we would get it aside from guessing.

AUDIENCE: So you divide it into thirds and compare one third and one third, and if they're equal, then the light one is in the other third. And if they're not, [INAUDIBLE] light one. Then you just keep dividing by 3.

PROFESSOR: OK. So that's the strategy. What if I don't know the strategy? How do I do this without knowing the strategy?

AUDIENCE: What if the number of coins isn't divisible by 3?

PROFESSOR: Math people.

AUDIENCE: Yeah, but then how do you-- OK, never mind.

AUDIENCE: Just take the two extra coins and toss them out.

PROFESSOR: If it's not divisible by 3, you add fake coins that are good. I mean, you use good coins. But we're not worried about the strategy. I want us to think of a lower bound. This is a lower bound for an algorithm, right? You cannot do better than log 3 N experiments.

Does the word "lower bound" ring any bells? Is there any lecture where we talked about lower bounds? So if you sort and you're using a comparison model, what's the best you can do?

AUDIENCE: N log N.

PROFESSOR: N log N. Good. So sorting using CMP, the comparison model, is N log N. How did we prove that? One word. Well, two words. Decision trees. Does anyone remember what decision trees are? One person.

AUDIENCE: It's just a comparison thing, right? You're like, is it greater, is it less than, or is there some sort of question you're asking about each key.

PROFESSOR: Cool. Let's go over that a little bit. No matter what your algorithm is, it's going to weigh some coins and it's going to get an answer from the scale. And then based on that, it's going to weigh some other coins and get some answer from the scale. And it will do some experiments and then it will give you an answer. So if you draw a decision tree, it would look like this.

First, we start with 0 information. We weigh some coins. Based on that, we have three possible answers-- smaller, equal, greater. Now, if we're here, we're going to do another experiment. Three possible answers. If we're here, another experiment, three possible answers. If we're here, another experiment, three possible answers. Say we do a third experiment. One, two, three, one, two, three, one, two, three, one, two, three, one, two, three, one, two, three, one, two, three, one, two, three, one, two, three.

And then suppose we stop. If we stop, we have to give an answer. So this is an answer, this is an answer, this is an answer, answer, answer, answer, answer, answer. So how many answers do I have at the bottom if I have three levels? Here I have three experiments, so three levels in the decision tree. How many answers?

AUDIENCE: [INAUDIBLE].

PROFESSOR: 3 to the third because I start three at the first level, nine at the second, 27 at the third. Each time, I multiply by 3. So if I do three weighings, I can give at most 27 answers. If I have more than 27 coins, I can't possibly decide which one is bad because say if I have 30 coins, then I need to be able to give out 30 answers.

My algorithm has to have a place where it says the bad coin is coin one, coin two, coin three, all the way to coin 30. Here I only have 27 possible answers, so this isn't going to cut it for 30 coins. I need to do one more comparison so that I have a deeper tree. So suppose I have h comparisons instead. How many leaves? How many possible answers?

AUDIENCE: h to the third.

PROFESSOR: Almost.

AUDIENCE: 3 to the h.

PROFESSOR: 3 to the h. So 3 multiplied by 3 multiplied by 3 multiplied by 3 h times, so 3 to the h. It's no longer equal to 27. 3 to the h is the number of possible answers. This has to be bigger or equal to N. Otherwise, the algorithm is incorrect. So what can we say about h?

AUDIENCE: [INAUDIBLE].

PROFESSOR: We did all this without even thinking of what an algorithm would look like. This works for any algorithm. No matter how smart you are, no matter how much math you know, your algorithm is going to be bound by this. So the fact that the answer looks like this gives you some intuition for how to solve the problem.

If you want to solve the problem now and figure out the strategy, you know that you have a 3 here. So if you divide into 2 every time, you're not going to get to the right limit. So first you do this, you get a lower bound, and then you use your intuition to figure out what the lower bound means. In this case, it would mean the strategy that we heard earlier. You have to divide into 3 every time and then figure out what you do based on the comparison.

So your answer works perfectly once we have this. And also, once we have this, you know that your answer is correct because it's optimal. You can't do better than that. Any questions on decision trees?

So lower bounds are a boring topic in general. They tell you what you can't do. They don't tell you anything useful about what you can do. In some cases, being able to reason about a lower bound gives you a hint of the solution.

New problem. Suppose we have a 2D map. There's a hill, and you take a satellite picture of it at night, and you get a picture with bright pixels and not bright pixels. There are numbers showing how bright your pixels are. 1, 2, 1, 2, 3, 0, 0. I'm going to draw out an example so we can use our intuition. 0, 0, 1.

So suppose this is our map. It's W times H-- W of what these are, I think they're columns, and H of the other ones. And you want to find a certain picture inside it. You want to see how many times does a certain pattern show up. Say the pattern is small w times small h, and it looks like this. But this will be the input to your problem, so the pattern might be different. You can't hard code this in.

And this is useful. This problem is called a bunker hill problem. This is a hill, and this is a bunker. You take a picture of the hill. You want to know where the bunkers are so you can bomb them at night so then you can attack the place.

AUDIENCE: That's awful.

PROFESSOR: Thank you. I'll take that as a compliment So a nice way of solving this?

AUDIENCE: You could just go through each row, and then look for a match for the first row, and then--

PROFESSOR: Yep. Is this a match? That's what you're saying.

AUDIENCE: Yeah. We can see that's a match.

PROFESSOR: Is this a match? Is this a match? By the way, this is a match. This is not a match, this is not a match, this is not a match, this is not a match. Now we go down here. This is not a match, this is not a match, this is not a match, so on and so forth.

AUDIENCE: That wasn't what I was suggesting, but that's a good idea.

PROFESSOR: Maybe you're suggesting something smarter, and I don't want to let you do something smarter so that we look at the brute force approach first.

AUDIENCE: I mean, I was just saying take the first row of your bunker, and then compare it to other rows, and once you hit that, then check and see if the rest of--

PROFESSOR: Yeah, that's a bit smarter, so that's harder to reason about. Let's take this one and figure out the running time of it.

AUDIENCE: Does that mean even if you know it's not a match, you keep checking all nine of them?

PROFESSOR: Yeah. Say at the worst case, you only find out it's not a match all the way at the end.

AUDIENCE: Are you trying to look for all matches or just one?

PROFESSOR: All matches.

AUDIENCE: You're limited to n squared time almost no matter what, right? If you have a small bunker in a large field, you have to hit the small bunker every time.

PROFESSOR: Are you going to solve the problem for me?

AUDIENCE: Are we trying to find the [INAUDIBLE]?

PROFESSOR: No. We're trying to find out the running time for the dumb algorithm first. Humor me and let's solve this first, and then let's get to the efficient algorithm, OK?

AUDIENCE: Big W minus small w plus 1 times big H minus small h plus 1.

AUDIENCE: Where'd you get plus 1 from?

AUDIENCE: So it's WH?

AUDIENCE: Yeah.

PROFESSOR: Well, there's something missing here. This is how many positions I have that I have to look at. How much time does it take to compare the small images?

AUDIENCE: [INAUDIBLE].

PROFESSOR: This is smaller than wh, which is the input size, so it's scary if you have an algorithm that runs faster than the input size because it means you're not looking at all the input. So this is definitely bigger than the input size once we add the w times h here. Don't forget this guy. This is the naive algorithm, and if we discard the small order factors, we get that this is order of WHwh.

How can you do better? You have the answer, right? Let's let everyone else think for a minute, and then you can give me the answer if you want, or someone else can give me the answer. I guess you should because you thought of it first. Any ideas?

So you're thinking about the input size, right? Someone was thinking about the input size. So the input size is W times H, right? So if I have an algorithm that's W times H, that's optimal because it has to look at all the input. Well, we're going to have an algorithm that's W times H, so with that out of the way, does that inspire anyone as to what the solution is?

AUDIENCE: Do that thing that I was saying, just take the first row, but then you still have a W term.

PROFESSOR: Yeah. So let's make it better. It is the correct intuition. Now try to use a trick we learned in lecture to make that faster.

AUDIENCE: Just use the top left corner instead of the whole row.

PROFESSOR: OK. So we could use the top left corner, and if the top left corner doesn't match, then we don't have to check for matches. So this works for reasonably random data.

As long as we don't have a lot of false positives, we're going to run fast.

Now, the top corner of this one, if the map has a lot of 1's and then some 2's sprinkled all over it, most of the time, we'll have to go through the whole image so we're going to have a lot of false positives. How do we make our false positive rate go down?

AUDIENCE: Looks kind of like a rolling hash problem.

PROFESSOR: Looks like a rolling hash problem, exactly. Let's see if we can use rolling hashes.

AUDIENCE: But then you still have that lowercase w term, though.

PROFESSOR: How do we get rid of it?

AUDIENCE: [INAUDIBLE].

PROFESSOR: Sorry?

AUDIENCE: Wouldn't it be w? I mean the running time if we were just going through one row would be big W minus little w, times--

PROFESSOR: So where's your rolling hash?

AUDIENCE: I guess you can use the entire thing as a hash, too. That would kind of work.

PROFESSOR: So we want a hash for the whole thing. Instead of using this as the hash, we want a smarter hash.

AUDIENCE: It's the entire thing, and then as you move to the right, you can add those and subtract, and compare that with the hash.

PROFESSOR: OK. So we'd have a rolling hash that has everything in here, and then as I move to the right, I add these guys and I remove these guys. This is big W times big H, roughly, times small h because every time I move to the right, I have to do order h work. So I'm down from this thing to order of WHh, So it's better. It's one step forward. Now, let's make this even faster. What if I could do this in order 1 instead of order h? How would I do this in order 1?

AUDIENCE: You'd have to compress all the rows, and then take the hash of each column.

PROFESSOR: How would we compress them?

AUDIENCE: Take the hash of the column.

PROFESSOR: You want to compress the rows?

AUDIENCE: Yes. You divide it--

PROFESSOR: Let's not compress the rows.

AUDIENCE: You could take just your bunker, and then figure out the hashes of the three columns, and just run through like that. You'd still have to access each of those items. I don't really see how it's faster. I guess it's less, though. It's less. Maybe it's only 1.

AUDIENCE: So do you want to hash each little column [INAUDIBLE]?

PROFESSOR: So we're going to hash all these guys, and then we're going to have hashes for them, and we're going to do the regular Rabin-Karp for the hashes. Now, what happens when I go down?

PROFESSOR: You have to recompute everything.

PROFESSOR: Let's do better than recompute everything.

AUDIENCE: Do you want to [INAUDIBLE] downward on each column?

PROFESSOR: Yep. Rolling hash. I want to make this faster, so I have big W hashes. They're all little h inside. Here, I have to compute them brute force. I can't do anything better. But when I go from here to here, there's only one element going out and one element going in. Same for all these guys. Let's not make the picture uglier than it needs to be.

So I have big W rolling hashes. They're vertical rolling hashes. And then the rolling hashes hash columns, so my the sliding window that I have is little w rolling hashes. Each rolling hash is little h in size, so it's a hash of hashes. It's nested hashes.

And then when I go down, I only have to roll down each of the rolling hashes by 1, so that's constant time. So to go from here to here, to the slide the window one down, I have to roll this hash down, roll this one, this one, this one, this one, this one, and all of them roll down in constant time.

So when I'm adding a column to the hash, say I'm here and I want to go here, I roll down this hash and I have the answer. It's order 1. I'm adding it in order 1. Does this make sense?

AUDIENCE: It's tricky.

PROFESSOR: But it's not too bad, right?

AUDIENCE: You just need to do the vertical roll first, right?

PROFESSOR: Yep. To have the simplest possible code, you start with big W rolling hashes, do 1D Rabin-Karp, you roll everything down, 1D Rabin-Karp, and keep doing that. OK What's the running time for this?

AUDIENCE: WH.

PROFESSOR: WH.

AUDIENCE: Does this one have a space complexity about W, then, because [INAUDIBLE]?

PROFESSOR: Yeah. So my memory requirement went up to 4W. Is everyone happy with this? It's one of the few cases where an approach for solving a 1D problem generalizes to 2D. In most problems, you have to rethink the whole situation.

Let's do a hard problem. Enough with the easy ones. Two lists, roughly size N, and they're both sorted. Let me fill them out with random numbers. 5, 13, 22, 43, 56, 62, 81, 86, 87, 2, 3, 7, 9, 15, 19, 24, 28, 32.

So I have these lists. Let's be generous and say that all the numbers are different. They're sorted. I want to find the nth number, so the number with rank n, out of both lists as fast as possible.

AUDIENCE: Wouldn't it just be index?

PROFESSOR: So the thing is if, for example, n is 1, then it's this. This is the second number. This is the third. So if you take the lists and you combine them, then I want the result out of that.

AUDIENCE: Let's do merge sort and the combined [INAUDIBLE] index, right?

PROFESSOR: Full merge sort, N log N? No.

AUDIENCE: It's already sorted, though.

PROFESSOR: OK. So what do we do?

AUDIENCE: We just do merge. That's just order N.

PROFESSOR: So merge.

AUDIENCE: Merge [INAUDIBLE].

PROFESSOR: OK. So merge and then index is the first approach, which is order N. Then you said run the merge algorithm, but stop when you get to the little nth element, so that's a little bit better and we don't have to produce an array so the space complexity is down to order 1, right? Now let's do--

AUDIENCE: Logarighmic times n.

PROFESSOR: Yeah, exactly. This is linear. We have to get to logarithms. How do we get to logarithms?

AUDIENCE: Do a modified binary search.

AUDIENCE: What if you first looked at-- actually, I don't know what I'm going to say.

PROFESSOR: Anyone else? So modified binary search. Do you know the full answer, or do you want to start looking at the solution?

AUDIENCE: I have an idea [INAUDIBLE].

PROFESSOR: Let's see how it would work.

AUDIENCE: So if we take the n over second element on each row, the one that is lower, that's at least the n over second element, and the one that's higher--

PROFESSOR: So let's say this is our N1 and N2, and they're both order N initially. So this is N2 over 2 if this is smaller.

AUDIENCE: The lower one is at least the N over second element since everything before it is less than N. Does that make sense?

PROFESSOR: Yeah. So this is N over 2 are greater, right? This guy.

AUDIENCE: We also know that the element above it is at most the nth element because it's greater than--

PROFESSOR: So it's at most N1 plus N2 because that's how many you have in total, and the one on the top, you know that these ones are bigger than h, right? But you don't know anything about these ones, so it's minus N1 over 2. So it's at most N1 over 2 plus N2, the top element.

AUDIENCE: So then we can take the element three quarters of the way through N2 and one quarter of the way through N1 to do more [INAUDIBLE].

AUDIENCE: [INAUDIBLE]?

AUDIENCE: Yes.

PROFESSOR: Let's see what happens in each case. So if little n is here, then you divide. So if n is smaller than this, then you chop them up here and you've divided the problem into half. You're good. If it's bigger than this other number here, you've chopped the problem up and you're here. You're good. Now, the hard case seems to be when it's in between. So what do we do then?

AUDIENCE: Aren't those two numbers the same?

AUDIENCE: If it's between, take the upper half of the bottom one and the lower half of the upper one, right? If it's between 15 and 43, then you take everything in the upper half of N2 and you take the lower half of N1.

AUDIENCE: Yeah, you should always be taking the upper half of one and the lower half of the other in this case.

PROFESSOR: Really?

AUDIENCE: [INAUDIBLE] N2, 15 is at least the nth over 2 element. I think we're using three n's at the same time.

AUDIENCE: Are you using the little n or the big N?

AUDIENCE: Sorry. [INAUDIBLE] is element and the lists are called N and this is confusing. Can we rename the nth element to something like m or some other useful number? The kth element, OK. So the 15 at least the kth over 2 element, so it can't be anything on the left half of the--

PROFESSOR: k over 2? Why k over 2? This list is size N.

AUDIENCE: Sorry. I didn't pick the elements at N over 2. I picked the elements at k over 2.

PROFESSOR: Why would I do that? If N1 is greater than k, then I chop off the end of the list, right? If N2 is bigger than k, then I chop off the end of the list completely. If this list is sorted and I want the third element, I know that these are not the answer. No matter what's down here, these are not the answer.

So I know for sure that k is going to be bigger than N1, N2. So instead of going there, let's go at k over 2. And here, let's go for k over 2. Now this one looks a bit nastier. N1 plus N2 stays what it was before.

AUDIENCE: On the list where you got the element that was lower, you know that everything to the left of it is less than k over 2. The element number is lower than k over 2, so we're not using anything to the left of the 15. You can kill that section for us.

PROFESSOR: OK, so we can kill it, but then what's the rank? When I recurse, how am I going to know the rank that I'm looking for?

AUDIENCE: k minus what you killed.

AUDIENCE: You can save the branches.

PROFESSOR: So you want to kill this guy, right? So you want to kill these numbers. But here I have k over 2 numbers, and here I have k over 2 numbers. How do I know that it's not somewhere here?

AUDIENCE: How do you know that what's not somewhere there?

AUDIENCE: You compare 15 and 43, right? And then you see that 43 is bigger, and you see 15 is smaller, so then you would go to, I guess, k over 4 index in N1, and 3k over 4 in N2.

AUDIENCE: When you recurse, you said that you've killed k over 2 elements.

AUDIENCE: But you can also kill everything to the right of 43.

AUDIENCE: Yes. You can kill everything to the right of 43 since it can't be any of those elements, and you can kill everything to the left of 15. And then you repeat the algorithm again with the lists you didn't kill, except you also put in a term of we've already covered k over 2 elements.

PROFESSOR: So we want the element with the rank k over 4 over these lists. So I know for sure that what I have is either k over 2 or less than k over 2, right? So this is less than k over 2, and then I'm looking for a rank of k over 4. That seems to work. How does the running time look?

AUDIENCE: It should be O of log k.

AUDIENCE: I think it's log [INAUDIBLE].

PROFESSOR: So log k, log N1 plus N2. Are these different?

AUDIENCE: Yes.

AUDIENCE: If k is 1, the algorithm should only recurse once, even if N is 20 million.

PROFESSOR: OK.

AUDIENCE: But if k is 20 million and the list lengths are two million long, it'll take approximately those lengths to run.

PROFESSOR: OK. So what gets reduced, aside from the list size, k gets reduced. k seems to define the input size for the next iteration because I'll have at least k over 2 elements in one of these buckets. So it sounds like it should be log k. k is bigger than N1, N2, but it should hopefully be smaller than the sum because otherwise, why am I doing the problem? So this is definitely order of N1 plus N2. So this is a bound. This is a slightly tighter bound.

We have a different solution to the problem. All the possible solutions are hard to argue. They all come down to something like this. The one that we have requires you to use-- you have two indices, and you know that the sum of the indices is k, and you do binary search on the top and adjust the index on the bottom to keep the constraint that the sum of the two indices in N. And you can look at that in the notes that we're going to post. Are you guys tired? Do you want to look at one more thing, or are we done?

AUDIENCE: [INAUDIBLE].

PROFESSOR: Let's look at something reasonably easy. You guys can read this on your own. I'm not going to bore you with that. So suppose we have some functions, and we want to order them according to their asymptotic growth rate. Do people remember how to do this from Pset one? So the idea is that you take each function, you simplify it, and then you sort them. So let's have a couple of simple ones and then some hard ones, and we're going to stop in five minutes. What's this?

AUDIENCE: n to the fourth.

PROFESSOR: OK. Let's see. n choose 3. What is this?

AUDIENCE: [INAUDIBLE].

PROFESSOR: OK, very good. Why?

AUDIENCE: Something about choose is n times n minus 1 times n minus 2.

AUDIENCE: It's n factorial over n minus 2 factorial.

PROFESSOR: This comes out to be roughly n cubed. Cool. How about n plus log to the fourth of n?

AUDIENCE: N.

PROFESSOR: Yep. So even if I have a polynomial in a logarithm, it's still dominated by pure n. Now, suppose we want to order these guys together with-- which one doesn't look boring at all? n to the log n and 2 to the n. Let's sort them. Which one's the smallest? Which one's the biggest?

AUDIENCE: 2 to the n is bigger.

PROFESSOR: Let's start with the smallest ones, because I think that will be easy. So which one's the absolute smallest out of all these guys?

AUDIENCE: n.

PROFESSOR: OK. Then?

AUDIENCE: n to the third.

PROFESSOR: Cubed and fourth. So we have to compare these guys. How do we compare them? n to the power of log n and 2 to the n something.

AUDIENCE: [INAUDIBLE].

PROFESSOR: Take the logs. So when we have something confusing with exponentials, take the logs and see what we get. Logs are monotonic, so if you take the logs, you'll have the same relationship afterwards. So log of this is log of n to the power of log n, so it's log n times log n, so it's log 2 n. Log of 2 to the n is n. Which one's bigger?

AUDIENCE: n.

PROFESSOR: All right. And we're not going to solve this, but how would you go about solving this guy? What do you do to it?

AUDIENCE: Sterling.

PROFESSOR: Sterling, yep. You do Sterling, you go through the numbers, and you figure out the answer. And then if you have to do logarithms, you use logarithms to figure out where it belongs among these guys.

AUDIENCE: What's Sterling?

AUDIENCE: Sterling's formula. It's that gross thing that was on the board before we came in here.

PROFESSOR: So Sterling says that n factorial is ugly. 2 pi n here times n over e to the power of n. So what's this binomial? What's the formula for it? OK, formula for n choose k. Anyone?

AUDIENCE: It's n times 1 over 2 factorial times n minus k factorial.

PROFESSOR: In this case, it's n factorial over n over 2 factorial raised to the power of 2, right? And then we chug through the math and get to some answer. All right?