JASON KU: I'm Jason. Hopefully you guys saw me on Tuesday. This is our first ever 6.006 problem session that we'll be having on Fridays this term. It's really an experiment. We've never done this before. But one of the things that we were discussed while preparing for this class is that we have two different methods of instruction, formally, usually in this class-- a lecture, which is there to present you with the fundamental material, the data structures, and the algorithms that are the base, the foundation of what-- how you will be approaching problems in this class; and then the problem sets that you will work on, our applications of that material.

But there's usually a much different feel between those problems that we'll give you then the underlying foundational material. So the application of that material will feel very different. And a lot of times, there are tricks to approaching the problems or ways of approaching the problems that you kind of just have to figure out by working on the problem, sometimes going to office hours.

But what we want to do this term was to-- since we had the opportunity to be recorded by OCW, was to record us going through some problems that we've had on problem sets in the past so you could see how we would approach working on these problems that you'll be working on, at least in a similar vein. So that's the goal of these problem sessions.

In the past, for OCW, we have recorded a recitation, but we felt that that was a little less useful to you guys, because recitation is meant for interaction, questions-- one-on-one questions. We want to be a safe space for you guys to interact with the material with-- in a smaller environment that might not be recorded. So that's the goal of this-- these sessions that we will be doing on Fridays. Any questions about what we're going to be doing today?

OK. So we do have a handout up here by the door, which we may or may not have in the future. This is all an experiment, so you'll have to work with us as we figure this stuff out. They were posted on LMOD about an hour before this session. We'll try to keep that as a standard.

But it just shows you the questions we'll work on today in the session. It isn't just the problem set from last term. It's a selection of problems from previous terms, and some of them have been edited to be maybe a little shorter and things like that. So what we're going to do is just go through the problems one by one. I'll try to just show you how I'm approaching the problems, but at any point, if you want to, you can ask questions. That's fine. OK?

All right, so the first question we have is-- has a setup that's very similar to what you will have on your Pset1. It's essentially saying that it has usually many parts. This has two parts, an A part and a B part. I've omitted a B and a C that that was on last term's problem set.

And it has five functions each, and you're trying to order them increasing as-- based on their asymptotic behavior. So here are the functions that we have. Maybe I'll stick it up instead. All right, so we have a few sets of functions, and we just want to order them.

And some of the functions may be asymptotically equivalent-- in which case, when we are ordering these things, we're going to put those numbers in a set. So what we have as an example are three functions-- n, root n, and n plus root n. This is one, two, three. And what we're going to ask you to do is order those functions based on their asymptotic complexity.

So hopefully you guys can get this one. Which one's the slowest growth in terms-- root n, so number two. So if we say f of 2 will be our first one. And then, how about the other two?

STUDENT: [INAUDIBLE]

JASON KU: They're the same. They're both order n. So we would put in set brackets f1 and f3. And on your problem set, if you put just 2, and 1, and 3 here, that would probably be fine. But if you were to put 2 over here or not have these curly braces around here, those would not be correct, and you'd get points marked off. Does that make sense?

OK, so we're going to approach the first set of problems-- first set of functions, which is a little different than the second set of functions. Hopefully this one's a little easier. One of the common approaches that I have in going through these things-- some of these are in a form that is hard for me to tell how they would compare to other things.

Actually, most of these are fine, but in general-- can anyone, just by eyeballing, tell me an order that works for them? Yeah?

STUDENT: OK--

JASON KU: This is a little difficult to do on the spot with five functions.

STUDENT: Yeah. A little iffy about f1.

JASON KU: OK.

STUDENT: But f5 is definitely smaller than f2, which is-- smaller than f2 which is smaller than f4. And f1 in iffy.

JASON KU: OK, great. That's excellent. So what we've got here is, on f2, f3, and f5, you kind of have this n leading term. If we factor an n out of that, then we're comparing log with-- basically, we look over here at the polynomial function. This one's smallest out of them, and then the log factor is smaller than a polynomial factor. Log grows slower than linear.

And so this guy is smaller than f2 is smaller than f3. That's great-- that your colleague said. Hopefully proved in recitation today-- no, Wednesday-- this nice fact, I guess, that a is less than, asymptotically, this polynomial-- this log n to any power is asymptotically less than any polynomial for any positive a and b.

And in particular, there's actually a stronger thing you can say, which is little o. Did you guys talk about little o in recitation at all? Probably not. It's kind of the same as big O, except is big O minus theta. So things that are asymptotically equivalent are not going to be included in this set. So actually, these things are strictly asymptotic-- grow strictly asymptotically slower than any polynomial. Does that make sense? OK. So knowing this identity, or this relation, can we say anything about f1? Maybe someone else--

STUDENT: Any a and b, right?

JASON KU: Any a and b-- any positive a and B-- anyone else have a guess? Yeah?

STUDENT: That f1 is less than f5--

JASON KU: Yeah, f1 is less than f5, right? Because just using that identity, this-- sorry-- a here that erased, stupidly, is smaller than, say-- this is bigger than n, and n to the b, being 1, is bigger. And then, as your colleague pointed out before, this thing is exponential, so definitely higher than a polynomial.

OK, so that was very easy, right? So the answer here is, if I got-- if I remember correctly, f1, f5, f2, f3, and then f4-- great. So that one was pretty easy. How about b-- or d, I guess? Yeah?

STUDENT: Can I just another question--

JASON KU: Sure.

STUDENT: [INAUDIBLE] How would you go about proving that?

JASON KU: How would you go about proving that? So there is a proof in your recitation handout there. The method in which they proved that in the recitation handout was putting the two-- taking a ratio of the two functions and taking their limit as n goes to infinity. And if the top one-- it grows arbitrarily, then the top one would be-- grow asymptotically faster. And if the it went to 0, the bottom one would grow asymptotically faster. And if it went to some constant, then that would be asymptotically equivalent. Does that make sense?

In actuality, to make the limit easier to take, we take the limit of the logarithm of the ratio. It just made it easier. Does that makes sense? OK, so let's move on to b. b-- we have a polynomial, and an exponential, and then we have these things down here. What do these things in parentheses mean?

STUDENT: Choose.

JASON KU: Choose, right. It's a binomial coefficient. Does anyone know the binomial coefficient is? Yeah? Hopefully from 6.042 or something like that, or whatever your competition math background is-- but in general, we have this definition. This thing is what? Does anyone remember? What does n choose k mean? Yeah?

STUDENT: The number of ways to choose k objects from n [INAUDIBLE]

JASON KU: Yeah, the number of ways to choose k objects from n things-- I never remember this formula. And probably, a lot of you have memorized this formula. I'm not going to ask you to do it. I'm going to tell you how I think about this. What if I want to know the number of permutations of n choose k-- or I mean of n?

Sorry. Just how many permutations are there of n items? What is that? That's just n factorial, right? So what we do here is we'd want to choose some n number of things. I have n factorial different ways of choosing those, but then, essentially, in here and in here, k and n minus k-- I don't really care what their order is.

So I'm going to divide out the permutations of this stuff and this stuff. Does that makes sense? So the formula here as I remember it, that hopefully is correct, is n minus k factorial. So I'm getting all of the permutations of the whole thing divided by their constituents. Does that make sense? Did I do that right?

OK, cool-- so that's a nice transformation. So the first step is writing these in terms of factorials. That doesn't really help me any, because I don't know how big factorial is based-- with respect to these other things. Does anyone know how big factorial is? Yeah?

STUDENT: You can use Sterling's approximation.

JASON KU: You can use Sterling's approximation. So that's nice. Does anyone remember what Sterling's approximation is? No. I don't remember either. I always have to look it up. Sterling's approximation says n factorial is approximately-- and this approximately is much stronger than an asymptotic behavior. It's actually, as these things-- as n approaches infinity, these things are equal. The limit is the identity.

But the approximation is the square root of 2 pi n n/3 to the n. OK, that's fun. So what kind of growth is this? Super bad, right? It's definitely exponential or higher than exponential. It's n to the n-- something like that-- dividing out an e-- this is e, the base of the natural logarithm. And this is a constant, and so is pi. Pi is a constant-- kind of interesting that we have two transcendental numbers here. That's kind of fun mathematics.

I'm sure some-- one of your 6.042 instructors could tell you why. I can't right now off the top of my head. But this is an approximation that's very good. And actually, sometimes other people will think-- this is what people call Sterling's approximation. One weaker notion that sometimes is useful for you is, if you take the logarithm of both sides, this is asymptotically what? If I took the log of this thing--

STUDENT: Polynomial--

JASON KU: It is a polynomial thing.

STUDENT: [INAUDIBLE]

JASON KU: It's basically n log n. If we take a log of this thing, it would be various things. All right, let's do it out. 2 pi n n/e to the n-- so when we're inside a logarithm, multiplication-- we can split it out-- becomes addition. Division becomes subtraction. And this thing grows faster than all these other things, so we can ignore them when we add them out asymptotically.

And so what we end up getting is this n to the n. The n comes out on the logarithm, and you get something that's theta n log n. Oh, that's fun. This is something we might use later on in the class.

OK, but when we are comparing these functions, one of the nice things to do is convert them into something that's familiar to us so that we can compare them easily. So here, this thing is whatever that thing is, roughly square root n n/e to the n. This is, I'm going to say, theta. That's a little bit more precise. All right, then what about these two things? Let's start with the bottom one. Can someone tell me what this is, asymptotically? Yeah?

STUDENT: n cubed--

JASON KU: n cubed-- why is that? Well, if we plug this stuff into that definition here, we have n factorial over 3 factorial n minus 3 factorial. n factorial over n minus 3 factorial just leaves us with an n, an n minus 1, and an n minus 2 over 6. And if you multiply all that out, the leading term is an n cubed, so this thing is asymptotically n cubed. I skipped some steps, but hopefully you could follow that.

And then the last thing to remain is this one right there. That one's a little tricky. Anyone want to help me out here? What we can do is we can stick it into this formula, and then apply Sterling's approximation to replace the factorials. That makes sense?

OK, so what I'm going to do is-- let's do this in two steps. This is going to be n factorial over-- what is this? n/2 factorial-- and then what is n minus n over 2? That's also n/2. So this is going to be n/2 factorial squared. Is that OK? Yeah?

Now let's replace this stuff with Sterling's approximation and see if we can simplify. So on the top, we have 2 pi n n/e to the n over-- and then we've got a square here, pi n. I cancelled the 2-- n/2 over e to the n/2. Did I do that right? OK. I can't spell, and a lot of times, I make arithmetic errors, so catch me if I am doing one.

OK, so let's simplify this bottom here. I'm not going to rewrite the top. The bottom here-- we square this guy. It's the pi times n. And then this guy, n/2 squared-- that just stays is an n. Then we have n/2 to the n/e-- something like that. n/2 over e to the n-- that makes more-- me happier.

OK, so now we have this over this. How do we simplify? Well, we can cancel out one of the root n's. So we've got square root of pi n down here and square root of 2 up top. And then what have we got? We've got n to the n down here and n to the n down-- up there, so those cancel.

We got 1 over e to the n, 1 over e to the n up there. Those cancel. What's left in this term, when-- after we cancel? 1 over 2 to the n in the denominator, which is 2 to the n in the numerator-- so this thing is what this thing is right. Asymptote. We can get rid of these constants. And it's root n-- I mean, it's 2 to the n over root n, asymptotically. Does that makes sense, everybody?

OK. With that knowledge-- and I'm sorry for my messy board work-- what is the ordering of these functions then? Can someone help me out? Someone else-- Eric, I'm sorry. You can't answer. Come on, guys. You followed what I said. Start it out for me.

STUDENT: I think [INAUDIBLE] f2 and f5 [INAUDIBLE]

JASON KU: Right. So both of these things are asymptotically equivalent, so we should put those in brackets-- f2, f5.

STUDENT: And then it would go f-- [INAUDIBLE] 3.

JASON KU: f3. This one? Why this one?

STUDENT: Nevermind.

JASON KU: I'm just asking you to justify what you're saying.

STUDENT: Well, because I feel like we have the [INAUDIBLE]

JASON KU: Uh-huh.

STUDENT: [INAUDIBLE]

JASON KU: This one's the biggest?

STUDENT: [INAUDIBLE]

JASON KU: f4 is the biggest, right? So this one's definitely bigger than this, because it's n to the n, as opposed to 2 to the n. So for any n larger than 2 plus e is fairly obvious that that's bigger.

STUDENT: So wouldn't f3 be before f1?

JASON KU: Why would f3 be before f1?

STUDENT: [INAUDIBLE] 2 to the n divided by [INAUDIBLE]

JASON KU: But we're dividing by a polynomial factor, right? So it's going to be slower asymptotically than the first one right there. So you got it right. What is it? F3, f1, and f4-- OK? Cool-- so it's a little complicated, but just applying some logarithm an exponent rules, understanding that logarithmic factors grow slower than polynomial ones, and again, grow slower than exponential ones.

And being able to do some transformations of some of these mathematical quantities to get them in a polynomial-like looking form is how you're going to approach these problems. Does that make sense? All right, so we're going to move on to question 2 now. Yeah, you've got a question?

STUDENT: Yeah, I have a question. What did you say the theta bound was for 4?

JASON KU: Theta bound for 4? This guy?

STUDENT: [INAUDIBLE]

JASON KU: So it's just that. I don't know how to simplify that any further. You've got a polynomial factor here, and then this is an n to the n term divided by an exponential. Yeah?

STUDENT: For f3, which one did you-- how did you reduce-- which one is f3 [INAUDIBLE]

JASON KU: f3 took this little cycle here. What we did was we expanded out the definition of the binomial coefficient here. Then we applied Sterling, and then we simplified and got back. Does that make sense? Yeah?

STUDENT: [INAUDIBLE]

JASON KU: Sure.

STUDENT: Is there a reason that f3 is before f1?

JASON KU: Why is f3 before f1? If I erase the 2 and the pi, this thing is theta of that-- 2 to the n over a polynomial factor. It's over n to the 1/2. n to the 1/2 grows non-trivially. Right? And so this is going to decrease the running time of this thing by a polynomial factor.

You could think about-- we're multiplying this by n to the minus 1/2 as well. That's another way of thinking about it. Any other questions? OK, so we're going to move on to problem 2, I guess. I need a eraser.

So problem 2 is kind of a funny looking problem. The point of this problem is to get you to think about using some of the things we're going to be using in this class as a black box. What using something as a black box means is that has a kind of a public interface that you're allowed to work with, but I'm not allowed to see what's inside of it.

And a lot of times, what we'll do in this class is try to use a black box and just try to use the abstracted outer functions so that we can prove things about it. We can just accept those as true, and then use those to deal with our analysis. So what we're given in this problem is a data structure supporting a sequence interface that you heard about yesterday. What's a sequence interface again? How does it store items? Anyone remember? Yeah?

STUDENT: [INAUDIBLE]

JASON KU: Well, it-- OK.

STUDENT: [INAUDIBLE]

JASON KU: All right. So what your colleague is saying here is we list them in a contiguous array. Does anyone have a problem with that definition? Yeah, up there--

STUDENT: [INAUDIBLE]

JASON KU: OK. So one of the important things about this class is abstracting this idea of an interface versus an implementation. And so what this student down here was talking to me about an array as an underlying implementation, and what the student back there was talking about is a linked list. These are both things that can implement that interface.

But in reality, the interface is something abstracted outside of those ideas. We could implement with either of those data structures. So what makes the sequence interface a sequence interface? Yeah?

STUDENT: It's in order, or at least it's indexed in a specific way that allows for [INAUDIBLE]

JASON KU: So it's about the data that we're storing. We're storing some number of things, and the important thing is that the data structure is maintaining being able to find items in that set by maintaining an order on them. I usually like to call it an extrinsic order on these things. It has nothing to do with what the items are.

It has to do with how I put them in order. There's a first thing, there's a 10th thing, there's a last thing. Right? That's what a sequence of items is right. And so what this data structure is doing, that's the input that is what we have available to us in this problem, is some kind of data structure storing a sequence of things. And it can support these four operations-- an insert first, insert last, delete first, and delete last.

And it supports each of those things in constant time. You don't a data structure that does that yet. You will on your problem set 1, and we'll talk about another way to do that today. But we don't care how it's implemented. We just give you this black box that achieves these things. Yay-- awesome.

And so what we're trying to do is, we have this thing, and I want to be able to manipulate the sequence stored inside, but all I have access to are these external operations. So the idea is going to be let's implement algorithms for these-- some higher level operations in terms of these lower level things that are given to us. Does that makes sense? And this is actually a pretty easy question. Hopefully we'll have slightly more difficult ones for you on-- in a different context on problem set 1.

OK, so the first operation we're going to support-- or try to support-- is an operation called swap_ends. And what this is going to do is take the data structure that we gave-- another way you could do this is put this as a method on that data structure, but let's do this separately-- it's going to take that data structure that we gave-- that I gave you, that's storing the sequence, as the only argument. And what we're asking you to do is describe an algorithm to swap the first and the last items.

I it was an array, I could just look at index 0, look at that, look at the last one look at that, and swap them. But I don't have access to what that underlying representation is, so how would I do that using the things that we have available to us? This is a pretty easy question. What have we got?

STUDENT: Just a quick question.

JASON KU: Yeah.

STUDENT: Does the delete method also return whatever it deletes?

JASON KU: Yes. Yes, it does. So in general, if you actually take a look at the session notes, it's giving you a nice little reminder-- recall, the delete operations return the deleted item. OK? It says it right there on the thing. Yeah?

STUDENT: Oh, I also had a question.

JASON KU: Sure.

STUDENT: And this actually [INAUDIBLE]

JASON KU: Mm-hmm.

STUDENT: It's not related [INAUDIBLE] if they don't specify a space [INAUDIBLE] does that mean [INAUDIBLE]

JASON KU: Yeah, so one of the things that Eric talked about yesterday was generally, in this class, if you have-- usually what we'll give you is a running time bound on the things that you asked for. And because allocation of space by our model takes that amount of time, the amount of time-- the amount of space that we are using is going to be asymptotically upper bounded by the time that we're going to use for the algorithm.

And so generally, we'll ask you to stay within a time bound, and not ask you to do something separate with space but there are problems probably at the end of this unit where we might talk about space complexity. But usually, we will be very specific if we want you to think about space.

Any other questions? All right, so how do we implement this swap_ends thing? Yeah. This is a pretty easy one. Yeah?

STUDENT: [INAUDIBLE] say first equals [INAUDIBLE]

JASON KU: OK, so another thing about this class-- your colleague over here is trying to write code to me, which is great for a computer, and that's great if you're taking 6.009. It's not great if you're talking to your friends or if you're talking to me. I can't parse code in my head and compile it all the time.

Sometimes I can, but not all the time, especially when it gets to be a large program. So I want you to explain in words to me, and we want you to explain in words in your LaTeX submissions what it is the algorithm is doing. So can you start over with your description?

STUDENT: Words are hard.

JASON KU: Words are hard. I agree with you.

STUDENT: This is a computer science class.

STUDENT: I would delete the last [INAUDIBLE] and then take that value [INAUDIBLE]

JASON KU: OK. So proposal-- we have a sequence of things. Again, as Eric was doing in lecture yesterday, this isn't representing an array. It's representing a sequence. So this is the front first I guessed and last. And what your colleague was saying was to delete this guy and stick it on the front, maybe by using delete last and insert first. That sounds pretty good. Does that do what swap_ends is doing? Swap_ends-- swap the first and last items in the sequence.

STUDENT: It's probably better if we first store it in some other variable, so that way we can get the-- delete the n [INAUDIBLE]

JASON KU: So what your colleague is saying is, well, we've done kind of half of our work. The first one's still over here. That's no good. Does someone have a way to modify this? Yeah?

STUDENT: Before modifying, you can [INAUDIBLE] to modify. [INAUDIBLE] about the amount of things we're storing.

JASON KU: Yeah. So how much extra space can we use, for example?

STUDENT: The easiest way would be to delete the last [INAUDIBLE] the first, and then just, if we have them already kept [INAUDIBLE] but I don't know if I can keep two different variables at the same time.

JASON KU: Good question-- so the question-- can I use additional space? And in general, if we don't give you any restrictions on what you can store, then you can go wild. Do whatever you want, outside of this data structure. One of the things you could do is remove first all of these things, store it and some data structure you like, manipulate it as much as you want, and then insert first all the way back in and rewrite the thing.

But that's not going to give us constant time, which is what we're asking for. But if we don't tell you otherwise, feel free to-- probably, you're only allowed to store a constant number of things, since we have constant time, but generally, unless we say, no, you can't use additional space, you can use additional space. OK? So how would you do that?

STUDENT: I would probably erase the last one and the first one, keep them both, then answer the first one, answer everything else, and then answer the last one.

JASON KU: That's great. So what your colleague is saying-- we delete both of them, we store them in temporary variables, and then, one at a time, we insert each of them in their corresponding place using the functions that we have available. OK, so if I were to write little pseudocode for this, I might take the first one-- I'd delete first. I'm really abusing notation here. That's OK. You get what I'm saying.

Delete the last, and then store them in their respective places. Insert at the front-- which one am I going to insert? What's up? Speak up, guys.

STUDENT: x2--

JASON KU: x2-- yes. Thank you. And insert last, x1-- OK, that's pretty easy. Yeah?

STUDENT: In this case, what would [INAUDIBLE] I think this might be relevant. [INAUDIBLE] What would constitute a pseudocode versus [INAUDIBLE] accidentally writing Python syntax [INAUDIBLE]

JASON KU: All right, so for this problem, you'll see the solutions posted to this later on. In that one, I wrote up a description of what I was going to do, and then I actually-- because this was pretty easy, I actually wrote down some Python code to do whatever this thing was. But in general-- and it's actually OK to write Python or pseudocode of this form on your problem sets, or on an exam, or something like that.

But if we can't understand what your variables mean, if we can't understand what your pseudocode is doing, then that's not sufficient. So the reason why we ask for words is so that you can communicate those ideas well.

STUDENT: Just a follow-up on that-- so can you also have a combination of pseudocode and description?

JASON KU: Sure.

STUDENT: OK.

JASON KU: Yeah-- including both of them can be clarifying for you, potentially. Any other questions? This is not such an interesting question from an algorithm standpoint. This is a constant size problem kind of. I have this data structure. I do two operations. I need to do something.

And this is so easy that I'm really not even going to argue correctness-- I'm not even going to have to argue correctness to you, because we're essentially just doing exactly what we asked for. Most of the time in this class, when you're doing something non-trivial-- especially when you're doing something that has to recurse in some way-- we do want you to argue correctness.

But in this case, for example, the time analysis is very easy. We do for operations. They each take constant time, so this operation takes constant time-- done. Yeah? All right, so how about the second operation? Second operation at least allows us to use a little bit more. So

Shift_left D, K-- this is the operation we're supporting now is we are given this sequence, and what we want to do is take the first K, here and stick it over here at the back so that the-- these K go here. So the K-th item ends up being the last item and the K plus 1th item now becomes the first item. Does that makes sense?

OK. Again, this is actually not such an interesting algorithm from an algorithm standpoint, but it's hopefully helpful to talk about from an instructional point of view. OK? So how would I approach this problem? I need this operation to happen in order K time. Yeah?

STUDENT: OK. So first, go and delete-- set a variable x1 to be the d dot delete first element. Then do d dot insert at the last x1, write a for loop, and then do that K times. And that should take 2K steps, which is OK.

JASON KU: OK. So what your colleague was saying is that we're just going to delete this guy, stick it on there, do it K times. That sound good? Yeah. One of the things in this class that you have, in terms of implementation-- usually there are two ways-- at least two ways you could do something that takes longer than constant time.

You could write a four loop where you could use recursion. Sometimes approaching a problem would be good one way, rather than another. Why is it that a lot of computer scientists, as opposed to coding engineers, prefer to think about an algorithm recursively? Does anyone know why?

At least I do, when I'm explaining it from a theory standpoint. It actually might not be good from an implementation standpoint, because your computer can vectorized for loops and things like-- but that's not something we need to talk about. Why would we want to talk about a recursive algorithm maybe more?

STUDENT: It lets you break up the problem into much smaller, more manageable pieces.

JASON KU: OK, so recursion lets you break up the problem into small measurable pieces. That's actually true in some context. How I like to think about recursion a lot of times is, if I have a non-constant amount of work that I have to do, usually easy for me-- it's hard for me to hold a non-constant amount of information in my head.

What I want to do is think about a constant amount of information at any given point in time, because that's easier for me to argue on. It's easier for me to think about making arguments, case analysis on these small amount of things. And so one of the things you can do is, if you break it down so that I solve a slightly smaller problem recursively, and then do a constant amount of work and maintain some invariant, then it's very easy to argue things about it.

It's very easy for me to convince myself that this thing's correct. So I'm going to provide a recursive way of solving this problem. Can anyone set up maybe a recursive way of thinking about this problem, instead of putting this inside a for loop, like your colleague was saying? Yeah?

STUDENT: [INAUDIBLE] people do is just consider what will happen when k equals 0 you're not [INAUDIBLE] it at all.

JASON KU: OK.

STUDENT: [INAUDIBLE] what it originally was. And then-- but if that k is greater than 0, you just [INAUDIBLE] and then [INAUDIBLE] k is 1 less [INAUDIBLE]

JASON KU: So what your colleague is saying is setting up the-- a very nice thing. She's saying that, if we think about this recursively, we'll think about a base case-- which, your colleague was saying maybe K equals 0. And otherwise, if we're not at 0, what we'll do is we'll start it out, move one of these guys over, and then we have an instance where we want to shift K minus 1 things over. Want to do the same thing, but with K minus 1 things.

And so we can just call this thing for a smaller value of K. Does that makes sense? All right, so let's try to write that out. The first thing I'm going to write out is kind of a break. If I'm at a base case, let's not do anything to this thing. And maybe I also want some bounce checking to make sure that we're in range.

OK, so I'm going to say, if our K is less than 1, I don't think we should be doing anything to this array. So let's just not do anything. If K is less than 1 or K is bigger than the length of D minus 1-- so I don't know what to do if you're asking me to shift more than the things I have, so let's not do that.

Yeah-- because if it was length of D, we would just not move anything anyway, because we'd shift the whole thing. So we don't have to do anything. All right. If we're in either of these cases, we're just going to return, because I either shouldn't do anything to the array or I have no idea what you're talking about, if it's negative or something like that.

OK, so that's the first thing. Otherwise, what do we do? We shift one thing over and then we make a recursive call. Does that make sense? OK. So we'll delete the first thing as a temporary variable-- delete first. And then we'll insert last, x. And then we need to do the recursive call. So what's a recursive call look like? Yeah?

STUDENT: Shift_left D, K minus 1--

JASON KU: Yeah. So shift_left D, K minus 1-- OK? And then we can return. This thing doesn't need to return anything. It's just doing stuff to the thing. Right? And whenever we get this K, we make a call, that gets down to 0, we will terminate because we will return. We're in this range somewhere between we-- have an input after this line.

We know that K is somewhere between 1 and n minus 1. And what we'll do is, every time through this recursion, we will subtract 1 from K. So this is a nice, well-ordered sequence. We do the correct thing obviously in the base case, and as long as this thing was correct for a smaller value of K, this thing also does the correct thing, because we're shifting over one, as we are asked, and we're letting this do the work of the rest.

I don't have to think about that. I just have to think about this one loop, this one part of the thing that I'm doing. Constant amount of work is done in this section. And how many times do I call a function?

STUDENT: [INAUDIBLE]

JASON KU: Yeah. I think K minus 1 times, or-- I don't know. I forget. But it's ordered K for sure, right? And we do a constant amount of work per call, ignoring this extra call. Does that make sense? So this thing runs in order K, as desired. OK? Does that make sense? All right.

So now we will move on to question 3. Any questions about question 2? That one's really probably one of the easiest problems we've ever had on a problem set. Sorry to scare you. So problem 3-- OK, so this is a little block of text right here.

A dynamic array can support a sequence interface supporting worst case constant time indexing as well as insertion and removal of items at the back of the array in amortized constant time. So this is what we did yesterday in lecture, right? We showed how a dynamic array-- it's fast to do dynamic operations at the end.

OK. However, insertion and deletion at the front is not very efficient because, if you tried to do that, you'd have to shift everything over. That makes sense? All right, on the other hand, what we talked about yesterday was linked lists. They can be made to support insertion and deletion at both ends in constant time. OK, so that's a little foreshadowing of something you're going to do on Pset1.

But in lecture, we talked about that operation-- that data structure, a singly linked list, being good at dynamic operations at the front of the list, because essentially, we could just remember where the front of the list was and swap things in as needed. That makes sense? So on your problem set, what you're going to do is make end operations good on the linked list as well, as well as supporting another operation. But what's the problem with linked lists, as compared to dynamic arrays? Yeah?

STUDENT: Linked list lookups can take up to linear time.

JASON KU: Yeah, linked lists lookups can take linear time, because I have no-- I don't have the benefit of an array, where I can randomly access something in the middle by essentially just doing one arithmetic offset calculation from the front address and be able to find this thing further down in constant time using our model of computation of the random net access machine.

In a linked list, these things could be stored all over the place in memory, and I have to traverse those pointers until I get to the one that I'm looking for. That's a benefit of an array-based data structure versus a linked, pointer-based one. OK, so then we get to the meat of this question.

Show that we can have the best of both worlds-- we can have a data structure that supports worst case constant time lookup, just like an array, but amortized constant time dynamic operations from the back and the front of the sequence. Does that make sense? All right. Is this question or a-- OK.

STUDENT: Can you define amortize one more time?

JASON KU: Yes. Sorry about that. Can I define amortize one more time? OK, so this is a tough thing to define in general, but not that much. All right, so amortization usually you put in-- at least in this class, we're going to put in terms of a data structure.

So you have this thing. It supports some operations, and you're going to do a bunch of operations on that thing. There's not really a reason to have a data structure, unless you're going to do lots of things to it. Otherwise, you just write a single algorithm to do whatever it is that you want to do.

The value of the data structure is that you can do some work up front by making this thing make some of these operations faster. OK? So what amortization means is, OK, if I have, say, a dynamic array, where I'm going to be inserting things at the end, sometimes, when I add something, I'm going to spend a lot of time to add that thing. I'm going to spend linear time.

But what's the point of this data structure in the first place? The point is that I want to be able to potentially add a lot of things to this thing. Does that make sense? Amortization is saying that, even though sometimes this operation will be bad, averaged over many operations, this is going to have a better running time. That's the amortization.

So more formally, what that's going to say is, if I have an operation, the definition of it running in amortized some amount of time-- say K time or-- yeah, sure-- that means that, if I do n operations, generally for large N-- if I do that operation n times, the total time it takes me to do all of those operations is not going to be more than n times K. So on average, it's going to take me K time.

Now, in O-4-6 you'll get a more formal definition of that and you'll get a lot of ways of analyzing things, like a potential function and-- we're going to use in what we call charging arguments even today. So it's a much broader analysis paradigm than what we're going to talk about. We're only going to talk about it for this material with dynamic arrays, and we'll just kind of-- it's just kind of an introduction to that. But does that make sense?

STUDENT: Yes.

JASON KU: Amortized as a financial term, if you know from financial term, means over the long term, this is what it is on average. You can think about-- but that's different than running time. That's average running time of an algorithm. It's a much different concept.

What is an average running time? Well, that's hard to define, because it's talking about an average over all possible inputs, and then-- OK, so maybe some inputs are more likely than others, and so you've got a distribution on the inputs and you're trying to average the running time of-- this has nothing to do with that.

Amortization means that you have a-- usually a data structure that you're operating on, and you're doing an operation multiple times, and you're getting a benefit because you're doing that operation lots of times. And so when you are instantiating a Python list and you're doing push and pop operations on the back, that's-- or is it append--

STUDENT: [INAUDIBLE]

JASON KU: Append and pop? OK. I've been writing JavaScript a little bit recently. But so append and pop-- those operations, while not cheap all the time, are cheap well enough that, when we analyze an entire algorithm that might do a linear number of appends to this list, all of those appends added together will only take linear time, because I've done a linear number of them. Does that make sense?

STUDENT: Yeah. Thank you.

JASON KU: OK-- long-winded answer to your question. Sorry about that. Any other questions before we get going? All right. Anyone have any ideas of how we can use the ideas of a dynamic array and make it good for operations on both ends? I'll let someone else answer. I'll give a second, and then go to you in a sec. Yeah?

STUDENT: So [INAUDIBLE] dynamic [INAUDIBLE] left true on one end [INAUDIBLE]

JASON KU: Sure.

STUDENT: [INAUDIBLE] here we could leave some [INAUDIBLE]

JASON KU: That's an excellent idea. We're going to talk about two ways of doing this. Right. So what your colleague was saying was that, in lecture, when we're talking about dynamic ways and we want to make operations on the right side-- the end-- fast, what we did was we allocated some extra space at the end, and then, when we added things, we didn't have to reallocate.

We had space to put those things. So what your colleague was saying was, let's just do the same thing on both ends. Let's leave some extra space on the front and extra space on the back when we instantiate this thing, and then we can rebuild less frequently than if we didn't have that extra space. Does that makes sense?

OK, so what we had for, let's say, down here-- so this is question 3-- the idea of the dynamic array right was that we left some extra space here at the end so that, sure, we allocated more than we needed to, but when we insert things now, it's cheap. And we don't have to allocate more space for this thing until we've done a linear number of insertions.

This was n. This was n. Really any constant factor will do here. But if you had n things here, we'd be assured that I wouldn't need to rebuild this thing until I've done a linear number of operations. And so in a sense, I can charge the linear time operation of re-expanding this thing to each one of those operations. And so on average, it'll be constant. Does that makes sense?

Right. So instead, what your colleague was saying-- let's instantiate this thing with some extra space on both sides. OK? So now, as I insert thing here, insert thing here-- blah, blah, blah, blah, blah-- I'll definitely know that, after a linear number of insertions, when I rebuild this thing, I'll have done enough operations to pay for that expensive operation. Does that makes sense?

So that's the idea behind expanding this dynamic array to be kind of this dynamic deck. It's a doubly ended queue kind of system where I can do dynamic operations efficiently on both ends. So one of the things that we talked about yesterday was also removing right at the end.

Removing items from the back of this thing will decrease the number of items we're storing, right? That makes sense. And maybe we're just fine with that right. As a programmer, why might you not like just removing items until you got to nothing, and just leaving the space where it is? Yeah?

STUDENT: Might lock up a lot of memory [INAUDIBLE]

JASON KU: Yeah. So let's say, over the course of my program, I use this data structure. I'm just trying to fill it up with stuff, and then I remove all but like two things, and then I go about my business. I run through the program. But I'm never really using any but those two things for the rest of my program.

But now I've got-- I don't know-- maybe I did put 1,000, or a million, or a billion things in that thing, and then, when I-- as I decreased, as I removed things from that item, I still have all that space there being taken up by essentially nothing, because I've removed everything from it-- at least in my conception.

So what I would really like to maintain with this data structure is that at no point in time am I using more than a linear amount of space with respect to the number of things that are stored in it. Does that make sense? So in a dynamic array, what we do is, when we get small enough, let's resize this thing down so that we have-- we're using less space.

As I'm decreasing, as I'm popping things from the end of this thing, at what point do you think I should rebuild my array? When I'm no longer a linear amount? Well, that's a little hard to tell what that is in real life, because our ends aren't arbitrary. We need to actually have a time at which we need to transition over and copy things over. So when might we want to do that?

STUDENT: [INAUDIBLE]

JASON KU: Say again.

STUDENT: After n/2 [INAUDIBLE]

JASON KU: After n/2 removals-- OK. So I remove n/2 things. OK, so now we're kind of at a n/4 fill-- so we're using a fourth of the space. And now-- great. So you're saying rebuild. OK, so I'll stick everything in something that's now-- this is m. I'm going to call this m, and now we're sticking it into something that has size m/4. Sound good? Yeah? Yeah? Everyone OK with this?

STUDENT: [INAUDIBLE] m/4 [INAUDIBLE]

JASON KU: Oh, OK. So what you're saying is that we actually want to keep some extra space back here. And why is that? Because imagine if we just allocated this amount of space, and I removed the m/4 plus 1th item here, we resized down to this thing, and then I want to do an insertion again.

Well, then I have to re expand out to something like this, and that's maybe not going to be a good thing. We might have to bounce back and forth a lot. That's hard for me to think about what we're going to do. But if we always resize to a fill ratio that includes a linear amount of things on the end, then I know that, when I resize down, I'll be doing either a linear number of deletions or a linear number of insertions before I have to rebuild again.

So this charging argument again-- I have to do a linear number of cheap things before I have to do an expensive thing again. OK? So I resize down to be-- still keep a linear amount of extra space at the end. And with the double ended thing, you can write the same kind of policy. With the extra space, as your colleague was saying, we can just resize down always to shift these things to be placed in the middle with a linear amount of extra space on the ends. Does that make sense?

No questions? All right. That was a way in which we had to redefine an entirely new data structure. We took the ideas behind dynamic arrays and we extended those ideas to make this thing have extra space on both ends. But we kind of had to do that re-implementation all by ourselves. If we were doing code, that would be kind of gnarly.

But what if someone just gave us a dynamic array? What if someone gave you a Python list, and you wanted this functionality? I don't want to reimplement a dynamic array, but I want this behavior, so how-- any way that I could do that by reducing to using a dynamic array-- get this kind of running time? No?

No one thinks that we can do this. This is impossible. No? No ideas? No ideas-- let's say I had-- I have a dynamic array that's good on one side. Is there anything I can do to support dynamic operations on both sides of a sequence? Yeah?

STUDENT: Are we able to just use [INAUDIBLE]

JASON KU: Oh, that's supposed to be empty, right? Yeah. So what your colleague is saying-- yeah, let's do that. Let's have one pointing forwards, one pointing backwards. This is the first of a certain thing. When we were doing just a dynamic array here, where we had to rebuild everything, it was important that we kept track of where the front thing was so that we could do time indexing.

As this thing changed, we would now have to compute where our index was in this thing by adding it to where the front was. On this one, we've got some similar problems. So what I'm going to do is I'm going to divide the sequence I'm trying to store up into two sections, maybe about the same size. So each of these contains a linear number of items. That's how I'm going to instantiate my thing with a linear amount of extra space on both ends.

So now, as I insert on either side or delete from either side, it's going to work just like a dynamic array. I have to do some arithmetic here to figure out where-- if I was trying to access these items, I'd have to subtract from wherever this thing-- I have to do some index arithmetic, but that's tedious, but you could do it. OK. There's one caveat, one problem that you run into in using something like this. And what would that be? Yeah?

STUDENT: I'm not sure, but you store things in the second half of a dynamic array.

JASON KU: In here?

STUDENT: In the first one.

JASON KU: In here?

STUDENT: [INAUDIBLE]

JASON KU: Right. So what I'm doing here is actually, I'm thinking of this as two dynamic arrays, but I'm viewing this one in reverse. So this is actually the last of this dynamic array. Does that make sense? All right. So if that's the situation I'm in, is-- am I done? Do I have to care about anything else? You guys are all like, we're done, and I would not give you full points. Why aren't we done? Yeah?

STUDENT: [INAUDIBLE]

JASON KU: OK, so what your colleague is saying is we somehow got to merge these into one array. So we're getting around that by keeping indexes to here and being able to do index arithmetic to kind of simulate an array underneath. So we can compute where these indices should be. Anyone have another problem with an underspecified data structure here? Yeah?

STUDENT: [INAUDIBLE] it could be that [INAUDIBLE]

JASON KU: I see. So what your colleague is saying, which is exactly correct-- if I were removing things, removing things, removing things, I have nothing else in here. If I try to pop from this end again, I'm going to have to pop from the beginning of this thing, which I don't really-- that's going to break something of what I'm doing. It's not maintaining the invariants of what I want on my data structure. And so the only caveat here is that, when I reduce down to one of these is empty, what do I do?

STUDENT: You have to cut the other one in half [INAUDIBLE]

JASON KU: You cut this thing in half, move these elements over. But that's going to leave these things in the middle here, right? The nice thing that happens here is I've done a linear number of options-- operations. I now have an amortized cost build-up that I can spend to now rebuild the entire data structure.

Does that make sense? I can now, once I get down to this thing, take whatever the remaining things are, split it in half, put it into two entirely new arrays, copy them all over, and now I've restored my invariant, where I'm, again, a linear amount of operations away from having to do an expensive operation again. Does that make sense?

So while we were able to reduce to using these dynamic arrays for a lot of the cases, we actually had to do a little bit more work to make this work out. That make sense? OK, cool-- so that's two ways of approaching problem 3. In the last little bit, we're going to talk about the last problem. All right, that makes sense. I'm going to erase this picture, if that's all right with you guys.

STUDENT: [INAUDIBLE]

JASON KU: What's up? It's not all right? Well, too bad-- watch the video. OK, so problem 4-- also a fairly accessible, shall we say, coding question-- what we're doing on problem 4 is we've got this nice little story at the beginning, which is about this woman Jen and her friend Barry, who are trying to sell ice cream to elementary school kids.

They're basically lined up at Jen's truck, and she's like, oh, there's too many students here. So she calls up her friend Barry. He has another ice cream truck, parts at the end of the line, and the students-- what they want to do is, to make it more fair, is they're going to take the last half of the line, reverse it to make it more fair. I don't know.

It's a stupid situation, but the underlying thing is what we're doing is-- part A here is we have-- we're giving you a linked list-- a singly linked list, and what I want you to do-- the singly linked list-- all it has is a notion of size, how long it is. It has a size and it has a head this list-- it has a size and it has a head.

And this head is a pointer to a node, and the node has just one-- two things stored in it. It has who-- the name of the child that's there, and the next pointer to the next node. That's what a singly linked list is. So node has an item key and a next pointer. This next pointer points to the next node in the sequence. OK?

And the question is asking, if we give you a linked list that has 2n nodes, I want you to take the last n nodes, and reverse their order, and do this to the data structure. You're not going to return a new data structure. You're going to modify the existing nodes. And actually, here is-- goes back to your question. What are we limited to in how we approach this problem?

What this problem serves as your algorithm should not make any new linked list nodes or instantiate any new non constant sized data structures. So it's not like I can write through this whole thing, find out where the n plus 1th node is, read out all of those names, store them in an array somewhere, and then rewrite them back out. I'm not allowed to store more than a constant amount of stuff outside of this linked list, and I'm not able to make any new nodes. Essentially, I just have to probably keep these items where they are and move around the nodes. Yeah?

STUDENT: Can you use non-constant space without creating a data structure?

JASON KU: So if you're using non-constant space, you're instantiating some kind of data structure, whether it be in an array or--

STUDENT: [INAUDIBLE]

JASON KU: Sure. I'm wanting you not to do that. Yeah. Any other questions? So how are we going to do this problem? Anybody? Anyone have approach for how I might approach this problem? Yeah?

STUDENT: In order to get to the second half you don't have to do all of that [INAUDIBLE]

JASON KU: Sure.

STUDENT: Could you start-- probably start counting backwards so that you can get them in the back order, and then meet it in the middle so [INAUDIBLE]

JASON KU: Interesting. So there's a lot of things-- let's break this down. So a lot of times, when we're asking you to construct an algorithm-- a lot of times, it makes sense to develop an outline or a game plan of constituent parts that you might want to approach this problem with. So the first thing that your colleague over here was saying was, at some point, we need to find out where the middle of this thing is. Does that makes sense?

So maybe the first thing we want to do to approach this problem is, one, find n-th node. That's the end of the first set of children. OK? Then I have a second thing that I want to do. What's the next thing I have to do? I have to reverse the pointers of everything after the n-th node, right?

OK, so second thing-- reverse, I guess, next pointers of everything after the n-th node-- the nodes n plus 1 to 2n. Does that make sense? And after I reversed all of those things, what do I have? I have a first block. This points like that. And now we've got this thing, and we've reversed all the pointers like this. That's after step 2. Is that what we want? Yeah?

STUDENT: Step 3 would be [INAUDIBLE]

JASON KU: So this is my new end. I'm going to call this node a, and this node b, and this node C. So tell me, in terms of a, b, and c, what I'm supposed to do. Yeah?

STUDENT: Quick question-- how would we reverse the next pointer? I get what you're saying, but--

JASON KU: Right.

STUDENT: --to actually made that happen--

JASON KU: Yeah. So to actually make that happen, this thing has an next pointer. It's pointed to-- pointing to some node. I'm needing to relink it to the thing before me, so I better remember what was before me so I can set node b.next equals the thing before me, instead of the thing after me. Does that make sense? So that would be relinking the--

STUDENT: And then it disconnects the linked list.

JASON KU: It disconnects the linked list possibly temporarily.

STUDENT: Oh, OK. It's temporary, and therefore, it still works out [INAUDIBLE]

JASON KU: Well, we have to relink everything to make sure it's temporary. It's very possible, when you're dealing with linked data structures, to unlink something and not have a reference back to it, and now this thing is in memory that your garbage collector hopefully will pick up. But if you're writing in a language that's not garbage collected, then that's called a memory leak. That's no good.

OK, so how do I relink these things? This is the picture that I have right now. How do I make this into a linked list, where it's here and then reversed? Yeah?

STUDENT: You can link a to c [INAUDIBLE]

JASON KU: Yeah. So I replace this pointer from a to b to make it point to c instead, and then, whatever my pointer is to b-- from b-- b is reversed-- it's pointing to a-- let's set that equal to none. So basically, the last step here is clean up ends.

And in LaTeX write-up, you'd want to specify, what are the things that you're relinking? But this was a coding question, and so we actually gave you code to work with. So I'm going to see whether I can live code this for you in front of you.

OK. So here was our code submission site from last term. And what I have here is my template from last term, Pset1. It opens this folder. It's got a bunch of things in it, the LaTeX template that you have, and then a bunch of these Python files. So I'm going to-- where is it? Here.

OK, so these are the files that are in my directory. I've given you a version of this linked list sequence. And then we have two more code questions-- a tests file and a reorder_students file. So reorder_students look something like this. It has a template of the code that we're going to want you to write, with inputs and outputs. And you're putting your code here. And this function doesn't need to return anything.

All right, and then we also give you this linked list implementation, which is what's in your recitation handout. I'm actually going to ignore most of this stuff-- really just that this thing contains an item in next in your node-- I'm not actually going to look at the items at all-- and a head and size in my linked list at the top level. OK?

But this is just to tell you what's in there. So that's what's going to be input to my thing, and if I go here and I run the tests document that you gave me, it fails because I don't have anything. It didn't do anything to the list. OK? All right. And in fact, if I go into here to the tests and I-- what is it? It's reordering the students here. I print the linked list that you gave me.

I'm going to have a line break here. What we can see is, when I do this-- here are my test cases. Here's a linked list. And what's happening is it's just spitting out the same linked lists. I haven't done anything to it. All right, so we need to do something to it. How are we going to do that?

All right, so let's implement this function. And I'm going to get rid of this stuff, because-- get rid of that. All right, so we need to reorder the students. So I'm going to break this up into three parts that we have here. We're going to find the n-th node. So how do we find the n-th node? This thing has a size on it, so let's at least figure out what n is.

So let's set n equal to-- I think I can use length, because I've implemented that on my thing, and it's going to be whatever the length is over 2. And I'm defined by the problem statement that I'm only going to have even inputs. And I'm going to set, at first, my a to be the starting place. I'm going to just have a little temporary variable that's going to say this is going to be equal to the head of my list.

And what I'm going to do is what your colleague was saying-- is I'm just going to loop through n times until I reach the n-th thing. Actually, how many times do I have to travel through next pointers to get to node a? n minus 1, actually-- yep. So this is going to be for. I don't care about this loop variable, so I'm going to just use that n minus 1 times.

What am I going to do? I want to replace a with the thing it's pointed to, so I'm going to just walk down this thing. a equals a.next. And now, after the end of this loop, what is a? a is the n-th node. I've now made it the n-th node-- fantastic. And now I'm going to say that b is going to be the next one.

Just in terms of my write-up, I labeled these things as a, and b, and c, and so in my mind, I'm going to want to use the same kind of notation here so that I can understand my code. OK, so b is going to be the next thing. And now, in this process, as I'm going to flip things around, what I'm going to do is I'm going to keep track of three nodes. I'm going to keep track of x, which is the node that I'm going to be relinking. And what else do I need to keep track of? If I'm--

STUDENT: [INAUDIBLE] destination.

JASON KU: Yeah, where I came from and where I'm going to, because that's what I'm going to need to relink. In particular, I'm going to have someone pointing to me, which I'm going to call next previous-- or the x previous. When I'm going to label it, it's going to be the next thing. All right? Does that make sense?

So in my first situation, I'm-- the first thing I need to relink is b, so that's going to be my x. And x previous is going to be a. Does that makes sense? So I'm going to instantiate those two variables. x and x_p are going to be b and a. Sorry. That's right. Yeah. Maybe it makes more sense to have x previous and x equal ab. All right, that's in the right order. Either way is fine.

And then I want to go through a loop. I'm going to be doing a loop way. You can do it a recursive way, if you want. Here's a loop way, in which I'm just going to loop through how many times? How many pointers am I going to relink as I go down this thing? I need to relink the pointers of all of these guys. How many are there?

STUDENT: [INAUDIBLE]

JASON KU: How many?

STUDENT: [INAUDIBLE]

JASON KU: n-- there are n of them. So for-- I don't care about the loop variable here either. I'm going to do this n times. And what am I going to do? I'm going to first figure out who my next guy is. I'm going to set x_n equals what? x.next-- all right, so now I know who's next to me right, so I can go there later after I relink my pointer. I'm remembering that.

Now, I don't care about what's stored in x.next, because I've stored it locally. That makes sense. All right, so now I am free to relink that next pointer to my previous guy. And now I can essentially shift my perspective over, so the thing that I'm going to relink now is the next one.

So x previous and x now equals x, x_next. Does that make sense? Just relinked things over-- so that's the end of step 2. Now, as I got down this at the end of this for loop, where is x? What is x_p, x, and x_next-- or x_n? Really, I'm only keeping track of x and x_p here. So what are x_p and x at the end of this loop? I've done this n times. I started with b at x. So what is x? Yeah? So we have a vote that x is c.

STUDENT: [INAUDIBLE]

JASON KU: So this is a little interesting. All right. I will tell you that c is either x_p, x, or x_n. So we have one vote for x. Who says something else? Eric doesn't like x. There are only two other choices. Does someone say something? x_p-- I will argue that it is x_p.

Why? Because I'm at b. There are n things. I did n operations, and every operation, I move 1 over. So when I've done n minus 1 things, I'm at c, the n-th one. Now x is none, because there's null pointer at the end of the list. So x_p is c, so I'm going to set p equal to x_p, which is-- it's just for me to remember what these things are.

And I just relink these two pointers. a.next should be c and b.next should be none. Does that make sense, everybody? Let's see if we did it right. So we save that thing and we run Python on the test cases, and it did the right thing apparently-- maybe. Let's see-- ran five test cases-- OK.

All right, so let's take a look at this. We had this linked list-- Lilly, Sally, Cindy, Maisy, Sammy, Davey. And what it turns into is Lilly, Sally, Cindy, which is correct. And then it reverses this last part of the list-- Danny, Sammy, Maisy-- cool, awesome. But these are the test cases we gave you. So let's try this against our code checker.

So I select the file. Where do I go? I think I'm in my desktop here in session 1, and template, and reorder students. I submit it. Please work. Please work. Please work. And 100%-- and now we're happy, and we can go party. OK. All right, so that's the first problem. Hopefully this was helpful to you. We will release problem set 1 tomorrow, and good luck on it.