[SQUEAKING]

[RUSTLING]

[CLICKING]

ZACHARY ABEL: Good morning. Good afternoon. Welcome to lecture 6. Today we will be finishing up our discussion of summations, that Brynmor started for us last time, and then moving on to asymptotic notation, which is one of the bread-and-butter topics of algorithm analysis. It's a really useful language and terminology that we can use to talk about algorithms and compare algorithms without getting bogged down in all the details. But before we do that, let's talk about sums. And before we do that, let's play with blocks.

So here is a very cool question. I have a stack of blocks. You could imagine it's even taller than this. I just grabbed six for now. And what I want to do is lean them out over the table and-- whoop, don't want him to fall. That's criterion one.

What I'm hoping for is to get this top block all the way off of this table. So none of the block is over the table. So currently, I'm not quite there. We can see there's an inch, half an inch left. So it seems possible, maybe.

So. The rules are we can only have one block per layer. We can't double up. We can't do weird angles and whatnot. It's just slide these left and right until you're happy. And hopefully, it doesn't fall. Are we clear on the game? Nice. Who thinks it's possible? Raise your hand. Who thinks it's not possible? All right. Well, let's try. Hopefully, I don't hurt myself.

So I'm going to follow a nice, greedy strategy. So I'm going to start from the top. And then I'm just going to keep sliding left as much as I can until it's about to fall over. So if I go past the center of mass, it's going to fall over. So I'm going to go right up until that edge and stop. And don't worry about the physics. We're not here to talk about physics.

But I'm going to do the same. So let's freeze the top two layers and try again. Let's slide this over until it's just about to fall. It feels right around there. All right, let's keep going. All right, up, up. Yep, OK. It leaned a little bit. I'm getting worried, but we'll make it work. Let's see. Yeah, OK.

And I think I might already be off the table, which was my goal. But let's go a little further. Why not? Yes. OK. And finally, right around. OK, that's as far as I'm willing to go I think that is off the table.

AUDIENCE: Whoa.

ZACHARY ABEL: It's possible.

[CLAPPING]

Yes, thank you.

[APPLAUSE]

I cheated. I put little marks on the back, so I knew how far to go. But it is possible. And now we can raise the question. How far can we go? Can we get two lengths out? Here's what that would look like, two lengths out. Can we get five lengths out, 10 lengths out? Let's stop at 10.

Who thinks we can get 10 lengths out over the table? Who thinks 10 is possible? Who thinks 10 is not possible? All right, you're both right. I'll explain why in a second.

Let's not have an accident during lecture. There we go. So, I'm not going to prove this now. But turns out that greedy strategy I just used is the best you can do. It's not too hard to prove. It's in the textbook, if you're curious.

But let me tell you what that greedy strategy ends up looking like, if I have some chalk. Here we go. So, as you might imagine, this first block sticks out half a unit over the next block. So its center of mass is right here on the edge. By the way, I'm imagining these blocks are one unit wide.

The next one, right here, that's a quarter unit. Because that's what it takes to put the center of mass of these two blocks combined right over the edge of the third block. Turns out this right here is a sixth. This right here is an eighth, and all the way down until the very end, until it is the very end. The n-th block has a gap of 1 over 2n. So this is block 1, 2, 3, 4, 5, all the way to n.

Turns out this is the best you can do. Which means the distance you can get with n blocks is 1/2 plus a 1/4 plus 1/6 up to 1 over 2n, which-- let's factor out the 1/2, just because that's cleaner. That's the sum. K equals 1 to n of 1 over k. This is also known as 1/2 times the n-th harmonic number, harmonic number.

This is a standard name for this sum, 1 plus 1/2 plus 1/3 plus 1/4, up to 1 over n And 1/2 of that is exactly as far as we can get with n blocks, right? So now the question is, how big does this get? If I have 1,000 blocks, how far can I get? If I have a million blocks, if I have an unlimited number of blocks, how far can I get?

Which means we're asking about this sum, this harmonic number. I want to know about that sum. Do we have a closed form for it? No, not really. But as we saw last time, we don't especially need a closed form if all we care about is approximation. So let's approximate that sum.

So we have our sum Hn, which is defined as the sum of the first n reciprocals. And we want to approximate it. And can someone remind me what tool we have to approximate sums like this? Yes?

AUDIENCE: The integral.

ZACHARY ABEL: The integral method, absolutely. So this function 1 over x is a decreasing function. And so we have our integral method, our integral bound, that says that our sum Hn is between the integral plus the last term. In this case, the last term is-- sorry, plus the first term, 1, and the integral 1 to n of 1 over x dx plus the last term, which is 1 over n.

So this bound is integral plus f of n. Less equal to the sum is less equal to the integral plus f of 1. And I claim this bound is really easy to remember. I plus fn, I plus f1, are pretty simple things.

And also, if you're dealing with an increasing function instead of a decreasing function, it's the same two bounds, but they're on the other side. Just because if it's increasing, then fn is going to be bigger than f1. So certainly, fn should be on the bigger side.

But this is a really clean bound. And we can write out what this says. Integral from 1 over x dx from 1 to n, that's ln of n. Let's assume we can do that without writing it down. You don't want to see me integrate live. That will go poorly.

But what this says is that ln of n plus 1 over n is less equal to Hn, is less equal to ln of n plus 1. All right? Which is a pretty nice bound.

But we're really interested in-- well, first, let's answer a question. If we want to get 10 blocks out, then we're asking for Hn to be about 20. Which means we're asking for these bounds. These bounds are basically, ln of n. The 1 over n and the 1 don't affect things all that much. So really, Hn is basically, ln of n.

I'm being imprecise here. But we'll be more precise about it in a bit. But for now, it's about ln of n, which means to get 10 blocks out, we need Hn equals 20. Because the distance was 1/2 Hn, which means we need n is approximately e to the 20. We need that many blocks. We need about 400 billion blocks to get 10 blocks away from the table.

Each one is about an inch and a half thick. So that gets us to-- carry the 1-- 40 times the distance to the moon. Probably going to run into some other issues before you're able to stack something that tall. So mathematically, yes, we can get 10 blocks over the table; physically, not so much. That's why you're both right.

But all of this was really an excuse to get us here. Hn is approximately ln of n plus some smaller terms. And let's try to make that a little more precise.

What do we mean by that? What do we mean by Hn is approximately ln of n? Well, let me give us a definition.

Let's write that f of n-- that's a tilde-- f of n is tilde of g of n. So this is asymptotically equivalent. So f and g are asymptotically equivalent if and only if the limit of fn over g of n as n goes to infinity is 1. If you divide them and get a limit of 1, then these functions are asymptotically equivalent.

And in this case, well, we have these nice bounds on Hn. Well, let's take that inequality. Let's divide all three sides by ln of n. What are we going to get? We're going to get 1 plus 1 over n ln of n is less equal to Hn over ln of n, is less equal to 1 plus 1 over ln of n.

And as we take the limit, as n goes to infinity, the left side looks like 1. This thing in the denominator gets big, so 1 over that gets small. The left side goes to 1. Lim n goes to infinity of Hn over ln of n.

And on the right side, we have 1 plus something that goes to 0. Because that denominator gets arbitrarily big. So the limit equals 1.

I learned this in high school as the squeeze theorem. If you can bound your limit between two things that are the same, then the limit is that thing. But what we've concluded here is that Hn divided by ln of n limits to 1 as n goes to infinity. And therefore-- sorry, Hn over ln of n equals 1, which implies that Hn is asymptotically equivalent to ln of n.

So when I wrote approximately ln of n up there, probably what I meant was this much more precise thing, that these are approximately equal in the sense that their ratio limits to 1. Does that make sense? Nice. All right.

Now, before moving on to asymptotics fully, there's one other topic in and summations that I want to talk about, which is not summations at all, but products. And really, what I want to say about products is that often, you can analyze products as if they were sums, turn into sums. For example, if we care about n factorial, which is here, 1 times 2 times 3, all the way up to n-- who can tell me how to turn this into a sum? It's something written on this board. Yes?

AUDIENCE: [INAUDIBLE].

ZACHARY ABEL: Yeah, take logs. Perfect. What happens if we take the log of both sides? So log of n factorial-- here, let's be precise. We'll call it ln of n factorial, the natural log, so log base e. Well, remember, when you take logs products turn to sums. In this sense, this becomes ln of 1 plus ln of 2 plus ln of 3 plus ln of n.

So if we want to understand n factorial, we could just as well understand its log. And its log is the sum of a bunch of other logs. And now we have a sum. And we can use all of our summation tools on this sum to get information about n factorial.

For example, we could use the integral bound. ln of x is an increasing function this time, so we'll use the increasing version of the bound, which says that-- here we go. So ln of 1 plus the integral from 1 to n of ln of x dx is less equal to the sum of ln of k. k equals 1 to n, which is less equal to ln of n plus the same integral.

And if we drew out that integral-- like I said, you don't want to see me integrate live. I will get it wrong. But if we trust my notes, this integral comes out to n ln of n minus n plus 1. Ln of 1 is 0, thankfully, keeps things simple. And so what we end up finding is that n ln of n minus n plus 1 is less equal to this sum, which, as we know, is ln of n factorial, which is less equal to this same integral plus another ln of n, n plus 1 ln of n minus n plus 1.

So we were able to translate this product into a sum and then use our summation tool to get us a bound on ln of n And if we want a bound on n factor-- sorry, ln of n factorial. I misspoke. If we want a bound on ln-- I misspoke again, apologies. If we want a bound on n factorial itself, we can just undo the log by exponentiating all three of these.

Take e to this, e to this, e to this. That preserves our inequalities. And then what we'll end up getting, after we simplify it a bit, is that n to the n over e to the n minus 1 is less equal to n factorial, is less equal to n to the n plus 1 over e to the n minus 1. So same denominator on both sides, we have n to the n versus n to the n plus 1 on the two sides.

So the two bounds are off by a factor of n, not as good as we had last time. We were off by an additive 1, less than that even. But a factor of n isn't bad for a number as big as n factorial. This isn't a bad bound. Is everyone clear on how we got to this bound? Wonderful.

It's not the best bound we have. Ow. Let me tell you two better ones. Oh, sorry, before that, is this bound enough to say that n factorial is asymptotically equivalent to one side or the other? I see some head shakes. Can someone tell me why? That's all right. I'm tired, too.

These two bounds are too far apart from each other. They're a factor of n apart. And so we can't say that this divided by this limits to 1 or this divided by this limits to 1. Because we have an n range in there to account for. And n goes to infinity. So we don't know where in this range n factorial lies. And so we can't pin down that limit going to a specific value. So this isn't tight enough to say that n factorial is asymptotically equivalent to something.

But that's where Stirling's approximation comes in. Stirling's approximation says that n factorial is asymptotically equivalent to the square root of 2 pi n times n over e to the n. We're not going to prove that, but it's really useful. You should write it down. You might end up using it in the problem set, hint, hint, extremely useful.

But remember, this is just saying that n factorial divided by this limits to 1. So when n is really, really big, the quotient of these two things is close to 1. What it doesn't tell us is what happens for smaller values of n. What if n is just 10? Can we use this to approximate 10 factorial?

How about 100? How about 1,000. I don't know. This doesn't really tell us. It doesn't tell us how good an approximation this is. All it says is that it's good in the limit.

So let me tell you an even stronger bound. I don't know who to credit this bound to. But it's even more precise than this one. It says that n factorial divided by this thing, 2 pi n, n over e to the n-- so this is a thing that Stirling tells us limits to 1. And now we have some bounds on how quickly it does that, e to the 1 over 12n, e to the 1 over 12n plus 1.

So not only does this limit to 1, it does so very, very fast, exponentially fast. And this, we can plug in specific values of n and see, OK, how close is this approximation to n factorial? This is an effective bound, one we can get actual approximations out of. This is just a limiting bound.

But of course, this is a lot more complicated. Sometimes, we don't want to deal with all these details, and this is enough. Sometimes, even that's too much, and that's enough. So pick your level of granularity as you need it.

We're also not going to prove this bound, that's even harder than Stirling. But it's true and useful. Questions on that? All right. So that's everything I wanted to say about summations.

And now let's jump back to asymptotics. We already started talking about asymptotics with asymptotic equivalence. And let's do an experiment. Let's do an exercise. We've already seen a sorting algorithm. We saw swap sort in lecture 4, where you have a list. You want to put it in sorted order. And you do that by finding a pair that's out of order and next to each other and changing their order.

And you keep doing that, Keep. Finding pairs that are next to each other and out of order, until the whole thing is sorted. So I'm going to call that swap. We saw that the number of swaps that you'll need to make is exactly equal to-- the number of swaps is exactly equal to the number of inverted pairs; so the number of pairs of two numbers in our list, where the big one is somewhere to the left of the small one.

So they're in the wrong order, and we'll eventually need to change order. And, well, how many inverted pairs could there be in the worst case? Well, in the worst case, every pair is inverted. 1, the number 1, is inverted with 2 through n.

And 2 is inverted. Well, it's already inverted through 1. We already counted that. But 2 would be inverted with 3 through n. Those are new inversions that we haven't counted yet. 3 is inverted with everything after it plus 1. So this is the worst case number of swaps we'll have to make.

And as we've seen, we know how to compute this sum. This is n minus 1 times n divided by 2, also known as n squared over 2 minus n over 2. And this, I'm going to call S of n. We have our function S of n, S being reminiscent of swap sort. It's a function describing the worst case number of swaps we'll need to make in swap sort.

On Thursday, we're going to see another sorting algorithm called merge sort I'll describe it then. I'm not going to describe it now. We don't need to know what it does or how. We just need to know how fast it runs. How many comparisons does it have to make?

So let's define m of n. This is going to be n log 2 of n minus n plus 1. And on Thursday, we'll see that merge sort takes at most, this many steps to complete.

And now I want to know, between these two functions, between these two algorithms, which one is a better choice for sorting lots of numbers? If these are the two algorithms I have to pick from, and I know their run times, they're n squared over 2 minus n over 2 versus n log 2n minus n plus 1. This is S. This is m. Which one's the better choice? And sorry, that was rhetorical.

Let's talk through it. Let's do an eye tracker to see which parts of the various formulas we look at when making this comparison. And just to emphasize, I'm most worried about what happens with lots and lots and lots of numbers. I'm trying to sort a million, a billion, a trillion numbers. So I'm interested in what happens in the limit as n gets really, really big.

If we're trying to 10 numbers, well, I don't care. However you want to do it is fine. 10 is small. But with big data-- no one uses that term anymore. I don't know why. With lots and lots of data, these differences matter.

So let's see how we make this comparison. What are we going to look at? Well, the first thing my eye goes to is this right here, which is to say, that's the last place my eye goes. I don't care about that term. The n squared over 2 is much more important.

By the time we're up into the n squareds, a little, measly n over 2 doesn't really make much of a difference. Right? Intuitively, anyway, I haven't made this precise. But the first thing that I do, when looking at this, is I just completely ignore that n over 2. I don't care.

Likewise, I don't care about this n minus 1 that we're subtracting. The n log n grows faster than that. So with big N, this is going to dominate. And this is just going to be a rounding error. So that's the first thing I do, ignore the lower terms. Now I'm just comparing n squared over 2 versus n log n.

The next thing I ignore is this over 2. I don't really care. I don't care that we're dividing by 2. If this had been n squared divided by 1,000, if this had been a million times n log n, is that really going to change my determination? For small n values, yeah, those constants matter. Those lower terms matter, for that matter, when n is small.

But in the limit, as n gets bigger and bigger, that extra factor of n is going to overtake that extra factor of log n, no matter what constants we throw in there. Right? So I don't really care about those as well. Really, all I'm focusing on is the n squared versus the n log n. Does that make sense?

Everything I just did was completely informal. I haven't justified any of that. But we will. But first, we need to motivate it. And there's one other reason, one other motivation, for not really caring about this constant factor.

With swap sort, what we're we counting, the number of actual swaps we make. We didn't say how you're finding those swaps. But there are clean ways to do it, so that the number of steps you take in the whole algorithm is about n squared over 2.

But wait, we have to do more steps than just the swaps, right? We have to read numbers from memory. We have to compare them. We have to write numbers to memory. We have to increment our index. We have lots and lots of instructions that we have to do during this algorithm.

So maybe it's just those four instructions every time we make a swap. So it's really 4 times n squared over 2 instructions. Instead of counting swaps, we're now counting instructions. And I'm being very loose about that. But it doesn't matter. OK.

Well, not all instructions are created equal. Every instruction on your CPU happens on a fixed, regular clock. And maybe adds take-- when we have to increment our index, maybe that takes, I don't know, five cycles. And comparisons take 15 cycles. And read and writes to memory take, I don't know, 200 cycles. Because memory is a lot slower than arithmetic.

I made up these numbers, by the way. These aren't real. Don't trust me on that. But in this fiction, if we're counting cycles instead of instructions, we're doing one of each of these per round or something. So really, we're doing, if I did my math right, about 210 times n squared cycles. So we're changing our unit. We're measuring different things.

Let's measure something else. Say we're on a 2.4 gigahertz processor, 2.4 gigahertz processor. How long does this take in real life time? Well, it takes about 210 divided by 2.4 billion seconds-- sorry, times n squared seconds.

So if we change what we're measuring, if we change our units, instructions, cycles, seconds, the thing that stays is the n squared. The constant changes, but it's always proportional to n squared. Get a better CPU, higher throughput, your time decreases, but only by a constant factor. It's still proportional to n squared.

This is another reason that it's useful to be able to just ignore that constant. I don't care what it is, because I don't really care specifically what units we're measuring. I just care that it's proportional to the right answer. Does that make sense, why we might want to throw away information like this? Because it's easier to think about. N squared is easier to think about than all this other junk cluttering it up.

One more motivating example, what's the population of the US? Does anyone know, just off the top? Yes?

AUDIENCE: 330 million.

ZACHARY ABEL: About 330 million, absolutely, about 335-- what is it? I wrote it down-- 893,238 is the population of the US as of January 1, 2024, according to the US Census Bureau. Is that more useful than what your colleague said, 330 million? I don't get any more information out of that in my measly, little brain. 330, I already have trouble conceiving of a number that large. I don't need all the extra detail.

Same for algorithm analysis, I want to be able to just focus in on the parts that matter and have the option to strip off the parts that don't. So that's our motivation. We want to be able to communicate less information about the runtimes of our algorithms, or about other functions, while still being mathematically precise about it and making sure we know what was lost. What information are we retaining, and what information are we losing, never to get again?

Because sometimes, you need that extra information. Throwing away all this stuff is a great, broad-strokes way to compare certain algorithms. But if we have two algorithms that are both proportional to n log n, oh well, now that constant factor really matters. And we better go and figure out what it is.

Likewise, if we're comparing the US population between this year and last year, well, last year, it was also about 330 million and change. So if all we know is 330, 330, we can't compare them. We don't know if it went up or down or by how much. Turns out, last year, it was about 334.2 million. So now we had a reason that we wanted more information, wanted more precise measurements of the population than just 330 million.

But wait until you have that reason. Let's do the broad stroke stuff first. And often, that'll be enough. Does that make sense, as a motivation, for why we want to throw away information about our functions? Nice.

So let's talk about how to do that. And the main one we're going to talk about is called Big O notation. The idea-- this is not yet a definition, still an idea. We're going to say that f of n belongs to big O of g of n, weird notation. But you'll get used to it. When? Well, roughly f of n is less than or equal to g of n, roughly.

If we say f is big O of g, what we're saying is that an algorithm with runtime f of n is at least as fast, maybe much faster than, an algorithm with runtime g of n, with some caveats, with caveats. First of all, let's throw away constant factors. If we have to scale g up by 10, and now g is bigger, that's fine. That's still big O.

And likewise, we're going to ignore-- I don't care what happens for small numbers, ignoring small n. Don't need them. I care about what happens in the limit. What are the trends when n gets bigger and bigger? So these are my intuitions, also ignoring lower order terms. I still haven't defined that either, by the way.

But we talked about them over here. The n over 2 was a lower order term compared to n squared over 2. I want to be able to ignore things like that. So this is still not precise. Let's finally, finally, make it precise, after this question. Yes?

AUDIENCE: [INAUDIBLE].

ZACHARY ABEL: Oh, no, I want to give the definition. Thank you. So our definition, our actual, final definition of big O is the following. f of n is n big O of g of n means there exists a number greater than 0, and there exists a natural number n0, such that for all n greater or equal to n0, f of n is less equal to C times g of n.

And if we're not restricting to positive functions, sometimes we put it in absolute value in there. That's less important. The other parts of the formula are the more important parts.

There exists a constant factor C so that f is less equal to C times g. That's this, ignoring constant factors. There exists an n0 so that all n greater than n0, greater than or equal to n0-- so there's some starting point, so that after that starting point, this inequality is always true. And I don't care what happens before n0.

So that's this point here, ignoring small n. Small n are the ones smaller than n0, n sub 0 is sometimes pronounced n0, N-A-U-G-H-T . Question?

AUDIENCE: [INAUDIBLE] that f of n is [INAUDIBLE]?

ZACHARY ABEL: Oh, gosh, yes. Thank you, g of n-- not the line I wanted to mess up today. Thank you so much. Yes, f of n is in big O of g of n, means roughly that g is bigger if you're allowed to scale it by a constant and throw away small values. All right? Yes?

AUDIENCE: [INAUDIBLE].

ZACHARY ABEL: Yeah. So fn is in big O of g of n means that there's some positive value C and some starting point n0, so that for all input values that are bigger than n0 or equal, fn is less equal to C times g of n. So this is extending our intuition that f is smaller than g. But you're allowed to put in a constant factor, if you need to, before that's true. Yes?

AUDIENCE: Why do you need the constant factor? I thought we said beforehand that we don't need the constant factor.

ZACHARY ABEL: Yeah. So the question was, why are we putting in a constant factor, when the whole point was to ignore the constant factor? We ignore it by saying that there exists a constant factor, and I don't care what it is. As long as some constant exists that makes this formula true, then f is big O of g. So we put it in so that we can not care what its value is. Yes?

AUDIENCE: Can that constant be between 0 and 1?

ZACHARY ABEL: Can the constant be between 0 and 1? Good question. Sure, why not? If you want to make g smaller, and it's still bigger than f, go for it, no problems there.

All right, let's do some examples. Hopefully, some concrete examples will make this a bit clearer. Let's say f of n is n. g of n is n squared. Is f of n in big O of g of n? So is n in big O of n squared?

Well, if this is supposed to say that the thing on the right is bigger, then hopefully, we would want n squared to be bigger than n. And thankfully, it is. In this case, we don't need the C to help us. f is already smaller than g. n is already smaller than n squared. n is less equal to 1 times n squared for all n, for all natural numbers n.

So this would be our n0 value of 0. And this would be our C value of 1. All we have to do is show that some n0 exists and some C exists to make this statement true. And then we're able to conclude that n is big O of n squared. Question?

AUDIENCE: [INAUDIBLE] n0 be 1 [INAUDIBLE] squared in less than n [INAUDIBLE].

ZACHARY ABEL: Ah, good question. Don't we need n0 to be 1 because this inequality is false when n is between 0 and 1? Excellent point. I'm going to give two fixes for that. First of all, we could change this to 1, absolutely. That's fine.

The other change is that let's go ahead and assume that f and g are mapping the natural numbers to the real numbers. So the inputs are just 0, 1, 2, 3 and not all the real numbers in between. Often, that's what we use when we're analyzing algorithms because we don't have fractional inputs to algorithms. You could absolutely do all of this analysis and all these definitions if your domain was the reals instead. But let's deal with naturals. Great point.

Also, in fact, why don't we go ahead and assume that all our functions are positive valued? Then we won't need that weird absolute value. You can still do a lot of this if you allow some of your functions to be 0 or negative. But let's not clutter our minds with that today. Let's go ahead and assume that all of our functions map the natural numbers to positive real values. Yes?

AUDIENCE: Is your definition of the top the same thing as saying f of n is asymptotically equivalent some constant [INAUDIBLE].

ZACHARY ABEL: The question was, is this statement equivalent to saying that f is asymptotically equivalent to some constant multiple of g? That's a great question. The answer is no. And I'll challenge you to think about why.

Let's do a few more examples. Actually, I want to come back over here. f of n is n squared. g of n is n. Is f of n in big O of g of n? Is n squared smaller than n, if you're allowed to scale by a constant?

And this time, the answer is no. An algorithm that takes n squared time certainly wants to look slower, take longer, than an algorithm that takes n time. So if big O is going to be useful for algorithm comparison, we hope that this would be false.

And thankfully, it is. And let's talk about why. So if n squared were in big O of n, we would need there exists a C, and there exists an n0, so that for all n greater or equal to n0-- let's see, n squared is less equal to C times n. I just wrote out the definition.

So we would need to be able to pick some constant and some starting point, so that after that starting point, n squared is always less than or equal to that fixed constant times n. But no matter what constant we pick, when n gets big enough, this inequality is always false. We can't pick a constant big enough so that this remains true for all N, for all big N.

If we pick C is 1,000, then from 1,0001 on, this inequality is going to be false. If C is a million, then from a million and 1 on, this inequality is false. No matter what C we pick, the left side will overtake the right side.

And increasing n0 doesn't help with that. Even if we ignore more small values, all the big values fail. You can't ignore all the values, so can't choose C big enough. So thankfully, n squared is not big O of n.

Let's do another example. Is n squared plus 3n plus 7, Is. That big O of n squared? We said we wanted to be able to ignore lower order terms, ignore things that look smaller. Well, this one ends up being true. And let's see why.

Well, there are lots of ways to prove this. I'm going to be extremely wasteful and do the following. n squared plus 3n plus 7 is less equal to n squared plus 3n squared plus 7n squared. All I'm doing is saying this n is less equal to n squared. This 7 is less equal to 7n squared.

And this equals 11n squared. And so we were able to show that this function is less equal to 11 times this function for all n. So we can take n0 equals 0, and C is 11, and we're done.

So, lower order terms like this at most force us to pick a bigger C If we want. But again, we don't care about the specific value of C, as long as such a value exists. So we can always swallow up these smaller terms, just absorb them into that C constant. Yes?

AUDIENCE: What [INAUDIBLE] know by looking at it that [INAUDIBLE] but obviously, n is going to be much smaller than n squared [INAUDIBLE] know what the function-- is one function bigger than the other [INAUDIBLE]?

ZACHARY ABEL: Good question. What if we didn't know whether one function is bigger than another function just by looking at it? Well that is our task. We're trying to compare these functions. And if we don't know which one is bigger, then we better figure that out before we get our answer. So I don't have a general way to do that. I can just say that it's necessary.

Let's do another example. 3n is that big O of n minus 4. We were also saying, we'd like to be able to ignore lower-order terms on the right-hand side. And intuitively, that n minus 4, that minus 4 term doesn't really change anything. This should still look like big O of n.

And we can make that precise. n minus 4 is greater than or equal to n minus, well, n over 2 when n is at least 8. All we're saying is that the n function overtakes the 4 function. Excuse me. So this 4, we can pretend it's an n over 2. We've lost a lot, but it doesn't matter. Because this is still a constant times n.

So this function is bigger than a constant times n. And we can use that to do our proof Let's see, 3n is less equal to 6 times 1/2 n, which is less equal to 6 times-- that's that, so n minus 4 when n is at least 8. And so now we have our C. And we have our n0.

3n is less equal to a constant times n minus 4, which is the function we care about. So in the same way, we can ignore lower-order terms on the right, again, by just swallowing them into the constant C at the cost of a slightly higher constant C. Does that make sense? Nice.

Now, some of you might recognize the definition of big O as looking a lot like epsilon delta definitions of limits. And if you haven't seen those, that's fine. Don't worry. It doesn't matter.

But they are similar concepts. And we have a useful fact that compares the two, that I will write down now, theorem. So we're assuming that f and g are functions from N to R+ to the positive reals if the limit of f of n over g of n as n goes to infinity exists and equals infinity. So if f over g gets arbitrarily big, then there's no constant you can choose to bring g back up to f, because f gets bigger and bigger compared to g.

Then f is not in big O of g. Likewise, if the limit exists and is less than infinity, then f is in big O of g. And finally, if the limit does not exist, inconclusive, f might be big O of g. f might not be big O of g. If the limit doesn't exist, the limit doesn't help us get an answer.

But if the limit does exist, it tells us the answer. If the limit exists, it tells us yes in this case, no in that case. So this is a one-sided test. If the limit exists, you're golden. If the limit doesn't exist, you have to fall back to the more complicated definition.

I'm not going to prove that. It's not difficult to prove. It's in the notes, if you want to read about it. But let's use it. Let's see. Is n squared in big O of a million n?

Well, how do we test that? Let's use this limit test. We can look at the limit of f over g of n squared over a million n. As n goes to infinity, n squared over a constant times n, that's infinity. And so the limit exists. It equals infinity. And that tells us that f is not big O of g. So, very often, if you're working with actual functions, and you can actually take the limit, and the limit actually exists, you get your answer much faster than if you were trying to think through the C's and the n0's and whatnot.

Another example is 1,000 in big O of n squared. What the heck do I even mean by that? 1,000 isn't a function of n. I thought we were talking about functions of n. Right? Well, it is a function of n if you want to think of it that way.

Let me plot it for you. Let me plot the n squared. This is n squared. And let me plot the function that takes every n and sends it to the value 1,000. So this is f of n, which is defined as no matter what your input is, ignore it, and spit out 1,000. It's the constant function that spits out 1,000.

So when you see something like this, when you see a constant in one of these asymptotic notations, it's really talking about the constant function that sends every n to that value. And in that sense, now we can ask, is this constant function big O of n squared? And we can answer it the same way as before by looking at the limit of f over g.

The limit of f, which is 1,000, over g, which is n squared, as n goes to infinity, that limit is 0. So yes, yes, 1,000 is big O of n squared. When the limit exists, it's great. We're very happy.

More examples-- we're trying to be permissive with constant factors. So is it true that 3 to the n is big O of 2 to the n? If constant factors don't matter, then maybe I should be able to change this constant to a different constant, and it won't really change, according to big O. Is this true?

And this time it's not. Because we can try our limit test. The limit of f over g, that's 3 to the n over 2 to the n. That's the limit of 1.5 to the n, which is infinity. So the answer is no.

Changing this 2 to a 3 isn't changing by a constant factor. It's changing by a constant factor n times. And that n gets bigger and bigger. So the difference between the two sides gets bigger and bigger. So this is a no.

Some quick notation, I've been writing f of n is in big O of g of n. And this is my preferred notation. This is set notation. F of n is an element of this set. So big O of g of n is really a set of all the functions that have the relationship we described.

This, I think, is the clearest notation. It's what I encourage you to use. But you're going to see some other notations. For example, you're going to see f of n equals big O of g of n, which is just super confusing. I don't like it. I don't recommend using it. But it is pretty standard in the math community, so you are going to see it.

So I have to warn you. You're also going to see f of n is less equal to big O of g of n. This is, again, an abuse of notation. fn is a function. This thing on the right is a set. But it's communicating the same idea. In fact, it's emphasizing that same idea.

Big O means that g is bigger than f, roughly. And so this less equal sign is just a reminder of that. So this is also good notation I recommend it if you want to use it.

You're also going to see f of n is big O of g of n instead of is an element of big O of g of n. That's also fine, pretty clear, unambiguous. All of these mean exactly the same thing. Sound good? All right.

What you should not write ever-- so we were saying, for example, we know that merge sort m of n is less equal to big O of swap sort of n, those two functions we looked at earlier, the n squared one and the n log n one. So this is a true statement, right? The n1 was smaller.

If we want to say that the S is bigger, should we say that Sn is greater or equal to big O of m of n? And from the way I'm phrasing the question, the answer is probably no. No, we should not write this. That's like saying you must be at least 60 inches or less to ride this ride. How tall do I need to be? 60. What? It's not correct.

Big O is functions that are smaller than m of n And if we're supposed to be bigger than some functions that are smaller than m of n, it doesn't make sense. Don't write that.

However, this is still an idea that we often want to communicate. S is bigger than m. We want to emphasize that's is bigger than m instead of emphasizing that m is smaller than S. And we have different notation for that.

Instead of a big O, we use S is big omega of m of n. So big omega is just the reverse of big O. We have our definition. f is in big omega of g means that g is in big O of f. Let's take that as our definition.

And we're getting lots of symbols around here. So let's see if we can compile them into a table. So we started with f is asymptotically equivalent to g. The definition is that the limit as n goes to infinity of f of n over g of n equals 1. The idea, the intuition, is that f is equivalent to g up to lower order terms, lower order terms. That was our intuition.

Then we had big O. f is in big O of g. The definition this time was there exists C greater 0. There exists n0 greater or equal 0, so that for all n greater or equal to n0, f of n is less equal to C times g of n. So that's our definition. Our intuition is that f is less equal to g up to constant values, up to constant factors, and I'll say lower order terms and ignoring small numbers. But that one does that, too.

Please don't confuse the intuition with the definition. And please don't use the intuition language in your proofs. Use the definition language in your proofs. So think about it this way, but write it this way.

And we just defined f is in big omega of g means the definition is that g is in big O of f. And the idea is that g is greater-- sorry, is that f is greater or equal to g up to constants and lower order terms. So we have a more permissive version of less equal. That's big O. A more permissive version of greater or equal, that's big omega. Questions about that? All right.

What did I want to say about that? Let me define one more while I'm over here. f is in theta of g. That's a capital Greek theta. And this is going to mean, roughly, f equals g-- sorry, f is approximately equal to g up to constants and lower order terms. And the definition is just f is in big O of g, and g is in big O of f. So f and g are roughly within constant factors of each other in both directions.

And I want to call attention to this first one versus this last one. Asymptotic equivalence said that f and g were roughly equal up to lower order terms. Theta says that f and g are approximately equal up to constant factors and lower order terms. Theta is more permissive than tilde. Where did I leave off on the boards? Over here? Yeah, over here. Good.

So we can say that our S of n function, our simple sort function, n squared over 2 and junk, is in theta of n squared. It's both greater than n squared and less than n squared, if you're allowed to put constants in there.

What's not true is Sn is asymptotically equivalent to n squared. This is false. Because if you take this and divide by that and take the limit, you get 2, not 1. That n squared over 2 matters. So this cares what that constant factor is. This doesn't.

This would be a true statement, if you put the 1/2 in there. So the asymptotic equivalence does single out that constant factor. Whereas, theta ignores that constant factor. Was there a question?

AUDIENCE: So [INAUDIBLE] be the same thing as saying there exists a C such that tilde [INAUDIBLE]?

ZACHARY ABEL: So the question was similar to your previous question. Does f in theta of g mean that there exists a constant such that f is tilde of C times g? That's a really good question. And the answer, I will tell you, is actually the other way.

If there's a constant factor between them, and then the limit goes to 1, then absolutely, they are theta of each other. But there doesn't have to be that constant factor. And let's look at an example of that.

Let's see, 3 plus sine of n is that theta of 1. 3 plus sine of n, we want to know, is it theta of 1? So is it bounded below and bounded above by some constant times 1?

And in this case, yes, it is. Because 2 is less equal to 3 plus sine of n, which is less equal to 4. So we can scale this by a constant. And it's always smaller than our function. We can scale this by a different constant. And it's always bigger than our function. And that's the condition we needed for big O on both sides. So yes, this is true.

However, you're never going to find a constant where 3 plus sine of n is asymptotically equivalent to that constant times 1. Because that ratio isn't going to have a limit. This is an example where the limit of 3 plus sine of n over 1 does not exist. The function just keeps doing this forever and never settles down and picks a single value.

So this is an example where the limit between them does not exist. But still, they are big O of each other. Right? In fact, in both directions.

We mentioned earlier, that if the limit doesn't exist, sometimes they are not big O of each other. Can anyone think of an example of that, two functions whose ratio does not have a limit, where f is not big O of g? A little tricky-- let me give let me give an example.

Let's say that f of n is n, and g of n is 1 when n is odd and n squared when n is even. So if we plot f of n is just n, g of n is following 1 or n squared, depending where you are. And it's zigzagging between the two.

If you scale g by a constant, you'll never be able to make it bigger or smaller than n, because the ratio is too much, depending on whether you're looking at the odds or the evens. You can't scale it in either direction. So f in big O of g is false. And g in big O of f is also false.

And it was specially constructed so that limit of f over g as n goes to infinity does not exist. Because on the odds, f over g grows really big. And on the evens, f over g grows really small. Sound good? Nice.

Speaking of getting really big and really small, there are two more asymptotic symbols that we need to add to our table. And those are the last two, I promise. The first one I want to define is called little o.

So f is in little o of g. The definition is going to be that the limit as n goes to infinity of f of n over g of n equals 0. And the intuition is that f is much, much, much smaller than g. If we want to say that f is insignificant compared to g, well, that's just saying that the limit between them goes to 0. So that's going to be our definition of little o.

Notation-wise, little o and big O look similar. So when you're writing those by hand, please try to exaggerate. I like to write my little o's really, really small, even smaller than other lowercase letters. And I like to write my big O really, really big, not hilariously big, but noticeably big, so no one's going to confuse it for a little o. Annoying notation that those look similar-- so just be aware, and try to correct for it.

And there's the reverse. f is in little omega of g. That's the lowercase omega compared to the capital omega up there. It means the opposite, that the limit as n goes to infinity of f of n over g of n equals infinity, a.k.a. g is in little o of f. So if we want to stress that f is much, much bigger than g, we can use the little omega notation, that f is much, much bigger than g.

And these are useful, I mean, for the same reasons the other things are useful. They're giving us more vocabulary to compare functions in ways that let us think about only a small amount of information about each function. But little o is exactly what we meant by lower order term, claim that f is going to be asymptotically equivalent to f plus h if and only if h is in little o of f if and only if the limit of h over f is 0, h of n over f of n as n goes to infinity.

So lower order term just means the ratio goes to 0 in the limit. And that's consistent with what we've been saying with lower order terms all throughout today. Lower order term just means limit goes to 0. And we have a special notation for that little o.

f is asymptotically equivalent to f plus h means that according to asymptotic equivalence, the h didn't change anything. The h was insignificant. So that's another way to think about what we mean by lower order term. Make sense? Nice. Cool.

Last thing I want to do today is a big, old warning. Asymptotic notation and induction don't mix easily. If you're ever doing asymptotics and induction in the same breadth, step back. Take care. Let's see what can go wrong.

So, theorem, question mark, 2 to the n is big O of 1. Now, we really, really hope this isn't true. In algorithm terms, this is an algorithm that takes exponential time. This is an algorithm that takes constant time. Certainly, this one is supposed to be the smaller one. So 2 to the n should absolutely not be in big O of 1.

But let's prove it anyway. Proof by induction, let's do our base case. n equals 0. So 2 to the 0 is in big O of 1. That's saying that 1 is big O of 1, which it certainly is.

Inductive step, assume n is at least 0 and that 2 to the n is in big O of 1. We want to show 2 to the n plus 1 is in big O of 1. But, well, 2 to the n plus 1, that's 2 to the n plus 2 to the n, which is big O of 1 plus big O of 1 by our inductive assumption. And that is big O of 1. So 2 to the n plus 1 is big O of 1. And we're done. We proved our induction. What the heck went wrong? Yes?

AUDIENCE: [INAUDIBLE] 2 to the n is [INAUDIBLE] is not equal to [INAUDIBLE].

ZACHARY ABEL: 2 to the n plus 1 is not equal to 2 to the n plus 2 to the n? It is. 2 to the n plus 1 is just 2 times 2 to the n, which I can write as two copies of 2 to the n added together. I will tell you, the mistake isn't in this line at all.

I know I did something really sketchy right here. What do I mean by O of 1 plus O of 1? That's a set, and that's a set. Why am I able to add them? Also, I accidentally used that when I meant set membership instead of equality.

But actually, this is kind of fine. This is an abuse of notation. We didn't really define what this means. But all we mean by it, O of f plus O of g, this is just the set of functions a plus b, where a is in O of f, and b is in O of g. This just means the set of things you can form by taking one function from this side and one function from that side and adding them together.

And it is true that in this case of 1 plus O of 1 equals of O of 1 as sets. So that's fine. That's not the mistake. Yes?

AUDIENCE: You proved that [INAUDIBLE] every number [INAUDIBLE] equal to the [INAUDIBLE]. But you didn't prove the function [INAUDIBLE].

ZACHARY ABEL: So let me reiterate. I proved that every number 2 to the n is big O of 1 itself. But I didn't prove the function 2 to the n is big O of 1. That's exactly the idea.

So let's see what we actually proved here. This here is a correct proof by induction. But what is it proving? It's a proof of what? What we really proved is that 1 is in big O of 1. That was our base case.

We proved that 2 is in big O of 1. 4 is in big O of 1. 8 is in big O of 1. 16 is in big O of 1. We proved that each individual power of 2, when considered as a constant function, is big O of 1. So every constant function is big O of 1. Yeah, that's true. We proved it by induction.

That has absolutely nothing to do with the 2 to the n function. It has absolutely nothing to do with the definition of 2 to the n being in big O of 1. So we didn't prove the right theorem. We proved something completely irrelevant. Does that make sense?

So this is why induction and asymptotics don't mix. When you see a function of n, you can't break that function into individual constant functions, one for each n value. You can't induct on n. You have to treat that function holistically as a single function. Sound good? All right. Does anyone have any questions about anything we talked about? Yes?

AUDIENCE: Is [INAUDIBLE] the set of [INAUDIBLE] over g [INAUDIBLE] smaller [INAUDIBLE] that goes to 0 [INAUDIBLE]?

ZACHARY ABEL: Excellent question. Does little o imply big O? So if we know that f-- oops, sorry-- if we know that f is in little o of g, does that imply that f is in big O of g? Yeah. Yeah, absolutely. And let's think about why.

So, little o is defined by, let's see, f-- sorry, f over g, that limit equals 0. But remember, we already said early on that if the limit exists, then it tells you whether or not it's big O. And in this case, the limit is 0. So it tells us yes, we are big O.

So little o is absolutely stronger than big O. It implies big O. We certainly don't go the other way. Likewise, let's see, there are lots of other implications between this, between these.

So theta, if f is theta of g, well, that certainly implies big O and big omega, because that's how we defined theta. And in turn, these also imply little o and little omega. Because big O implies little o, as we just saw, which means big omega implies little omega, because we're just swapping the order. So yeah, we have lots of implications like this.

Let me write it this way. This implies little o, and this implies little omega. And as we already saw, asymptotic equivalence implies theta, which in turn, implies all the others. Yes?

AUDIENCE: Wait. [INAUDIBLE].

ZACHARY ABEL: I got these arrows wrong. Thank you. No. See, I shouldn't ad lib at the board. I'm always going to get it wrong. Yeah, I apologize. Theta implies the big ones, not the little ones. The little one implies the big one, not the other way around. Sorry for the confusion. One more question? All right, thanks, everyone. We'll see you tomorrow at recitation.