ProgrammingAbstractions-Lecture14

Instructor (Julie Zelenski):Good afternoon. Apparently, Winter Quarter is over and it's now Spring. I don't know what happened to the weather, but I'm pretty happy about it. I hope you are too. Let's see. Where are we at? We are picking back up with the text. Looking at Chapter 7. Chapter 7 is gonna be the theme for pretty much the whole week. Talking about algorithms and different ways of analyzing them, formalizing their performance, and then talking about sorting as an example problem to kind of discuss in terms of algorithms and options for solving that problem. The mid-term, there's something. I'll say mid-term and everybody will shut up. Mid-term next Tuesday, so in terms of your planning and whatnot. Our mid-term is pretty late in the quarter. We had till we got to a substantial body of stuff to test and so that will come up next Tuesday.

What I give out today was some sample problems that are taken from old exams to give you an idea of the kind of things we're gonna ask you to do. Also a handout that was built up over time by the section leaders, which just includes kind of strategies and ways to think about prepping and getting yourself ready for the exam coming up. So hopefully those will be useful to you. I will put out a solution to the problems that I gave you in next time's class, but I'm trying to make sure that you have the problems without the solution in hand as a way of encouraging you to think about the problems from a blank piece of paper answering standpoint, which I think is the best way to get yourself ready for thinking about the actual exam.

So it is Tuesday evening and it is at Terman Auditorium, which is kind of just over the way in the Terman Building. All right. Yeah. Anybody totally done with Boggle? Ooh, look, way in the back. Hey, you can raise your hand high. You say I rock. Good for you. Other people making good progress on Boggle? Anybody who's gotten somewhere along the way run into anything they think they can save heartache and sadness for their fellow comrades by telling us up in advance? Smooth sailing? No problems? If you've got your recursion mojo, you are set for Boggle actually. Okay.

Okay. So let's talk about algorithms. Algorithm is one of the most interesting things to think about from a CS perspective. Kind of one of the most intellectually interesting areas to think about things is that often, right? We're trying to solve a particular problem. We want to put this data into sorted order. We want to count these scores and place them into a histogram. We want to search a maze to find a path from the start to the goal. For any of those tasks, right? There may be multiple ways we could go about solving that problem that approach it from different angles, using different kinds of data structure, solving it in forward versus solving it in reverse, is it easier to get from the goal back to the start? In the end the path is invertible, so maybe it doesn't matter which end we start from.

Would it be better to use an array for this or a vector? Would a set help us out? Could we use a map? Could I use iteration? Could I use recursion? Lots of different ways we could do this. Often sometimes the decision-making will be about, well, do I need to get the perfect best answer? Am I willing to take some approximation? Some estimation of a good answer that gives me a rough count of something that might be faster to do than doing a real precise count? So what we're gonna be looking at, right? Is taking a particular problem, sorting is gonna be the one that we spend the most time on, and think about, well, some of the different ways we can go about putting data in sorted order and then talk about what the strengths and weaknesses of those algorithms are compared to one another.

Do they differ in how much time it will take to sort a particular data set? Does it matter if the data is almost sorted? Does it actually affect the performance of the algorithm? What happens if the data set is very large? How much memory is gonna be used by it? And we'll be thinking about as we do this, right? Is often that probably the most important thing is how does it run? How much time does it take? How much memory does it use? How accurate is its answer? But also given that brainpower is often in short supply, it's worth thinking about, well, is the code easy to develop, to write, to get correct, and kind of debug? It may be that a simpler algorithm that doesn't run as fast, but that's really easy to get working, might actually be the one you need to solve this particular problem and save the fancy thing for some day when really you needed that extra speed boost.

So I'm gonna do a little brainstorming exercise with you. Just to kind of get you thinking about there's lots of different ways to solve what seems like the same problem. All right. I'd like to know how many students are sitting in this room right now. All right. I look around. Michelle, what's the easiest way and most reliable way to just get a count, an accurate count, of everybody in this room?

Student: Start by identifying what kind of [inaudible].

Instructor (Julie Zelenski):Yeah. Just have a strategy. Some people will use the front row. And I see one, two, three, four, five and then I go back and I just work my way through the room all the way to the end, right? What I've done, if I actually remember my grade school counting, right? Then I should have come up with a number that matched the number of people in the room and tells me, okay, there's 100 people, let's say, that are here. So that will definitely work. That's the easiest most ways to, sort of, likely approach you'd think of. You need to count something, well, then count them, right?

Now, I'm gonna talk about ways that we could do it that maybe aren't so obvious, but that might have different properties in terms of trading it off. Let's say that the goal was to count all of the people in Stanford Stadium, right? And so I've got a whole bunch of people sitting there. The count in turn walking around the stadium doesn't scale that well. However long it took me to work my way through the rows in this room, right? When you multiple it by the size of the Stanford Stadium is gonna take a long time and as people get up and go the bathroom and move around and whatnot I might lose track of where I'm at and all sorts of complications that doesn't really work well in the large.

So here's another option. Anyone else want to give me just another way of thinking about this? Yeah?

Student: Multiply yourself into more Julies.

Instructor (Julie Zelenski):Because we can only – one Julie's not really enough, right? So recursion. Can recursion help us out here, right? Can I do some delegating, right? Some subcounting to other people? So in the case of this room, it might be that I pick somebody to count the left side, somebody to count the right side, and somebody to count the middle. And maybe even they themselves decide to use a little delegation in their work and say, well, how about I get some volunteers on each row? That's sort of idea would work extremely well in the Stanford Stadium as well because you just divide it up even further. You say, well, here's this row, this row, that row and kind of having all of these delegates working in parallel to count the stadium, right? Getting recursion to kind of solve that same problem.

What if I wanted – I was willing to tolerate a little bit of inaccuracy? What if I wanted an estimate of the people that were in this room and I was willing to accept a little bit of sampling or estimation error?

Student: Maybe you could take, like, a little area of seats and count how many people are in those seats and then multiply it by how many of those areas are in the room?

Instructor (Julie Zelenski):Yeah. So it's like this capacity. Somewhere there's a sign, right? On one of the doors here that will say, oh, this room seats 180 people. So I know that there are 180 seats without doing any counting. So if I took a section. So I take ten seats, let's say, or 18 seats for that matter. I take 18 seats and I count those 18 seats how many are occupied and let's say half of them of the 18 that I look at were occupied then I can say, well, the room's about half full, right? And then I say, well, okay, 180 seats, half of them, right? Means 90 people are here. So in that case it's not guaranteed to be accurate, right? If I happen to pick a particularly dense or sparsely populated section, right? I'd be getting some sampling error based on that, but that would also work in the Stanford Stadium, right?

We know how many seats are in the Stanford Stadium, right? You pick a section and you count it. You count 100 seats worth to get a percentage of how full it is and then just multiply it out to see what you get. There's a variation on that that's kind of a – maybe not the first thing that would occur to you, right? Is just that the idea of counting some subsections is kind of interesting as a way to kind of divide the problem and then say, well, in the small can I do some counting that will then scale up? So, for example, if I had everybody in this room and I said, okay, think of a number between one and ten. Everybody just think of a number independently on your own. Okay.

If you were thinking of the number four, raise your hand. I got one, two, three, four, five, six, seven. Seven people. And said, okay, well, if I figure that everybody was just as likely to pick a number between one and ten as four, then those seven people, right? Represent 10 percent of the population. So maybe there were 70 of you. Again, totally based on randomness and sampling error, right? If I happen to pick a number that's very unpopular with my crowd, right? Or very popular, I get kind of an artificial estimate. But it does tell you a little bit about the nature of just solving this problem. It’s like how much error am I willing to tolerate and how much time I’m willing to invest in it, how big is the problem I’m trying to solve? What kind of things can I do that might, in the end, give me some estimate or accurate count, right? Depending on what I’m willing to invest in the time spent. So random thought for you. Here’s another one. I can take the access class list, right? And I can take the first ten people and call out their names and find out if they’re here and on that, right? I would get this estimate of, well, what percentage of my students actually come?

So if my class list says that there’s 220 students, which it does because where are they? And I take the first of ten students, right? And I call them out and I discover three of them are here I can say, oh, about 30 percent of my 220 students come, 66 of you. And then I would also have the advantage of knowing who was a slacker and I could write it down in my book. No, no, no. I’m sure you guys are all at home watching in your bunny slippers. All right. So let’s talk about in terms of computer things. Like that the situation often is you have a problem to solve, right? You need to do this thing. You need to solve this maze or compute this histogram or find the mode score in your class or something like that and you have these different options about how you might proceed.

Which data structure you use and how you’re gonna approach it and stuff like that. Certainly one way to do it, and not a very practical way to do it, would be to say, well, I’ll just go implement the five different ways I could do this. For example, when I was running your maze assignment that’s what I did. I wrote, like, ten different maze solvers until I decided which one I was gonna give you. That’s not very practical, right? If your boss said to you, you know, you need to solve task A and you’re response was I’ll solve it ten different ways and then I’ll come back to you with the best one. That might take more time than you’ve got. So you would have to write it, you’d debug it, you’d test it, and then you could kind of time it using your stopwatch to find out how good did it do on these data sets.

Yeah. Well, there’s some issues with that. First of all, you have to go do it, right? Which is kind of a hassle. And then also it’s subject to these variations, like, well, what computer were you running on? What OS were you running? What other tasks were running? Like did it – it may not be completely reliable whatever results you saw. What we’re gonna be more interested in is doing it in a more formal abstract way, which is just analyzing the algorithm from a pseudocode standpoint. Knowing what the code does and the steps that it takes, right? What can we predict about how that algorithm will behave? What situations will it do well? What situations will it do poorly? When will it get the accurate answer or an estimate? How much time and space can we predict it will use based on the size of its input?

That would tell us based on just some descriptions of the algorithm which one might be the best one that’s gonna work out in practice. Then we can go off and really implement the one that we’ve chosen. So this is actually working more the mathematical sense, less on the stopwatch sense. But helps us to analyze things before we’ve gone through all the process of writing the code. So what I’m gonna do with you today is a little bit of just analysis of some code we didn’t write to talk about how it behaves and then we’re gonna go on and talk about the sorting problem and some of the options for that.

So this one is a function that converts the temperature. You give it a temperature in Celsius and it converts it to the Fahrenheit equivalent by doing the multiplication and addition. So the basic, sort of, underpitting of kind of looking at it formally is to basically realize that we’re gonna count the statements that are executed. Assuming, right? In this very gross overgeneralization that each action that you take costs the same amount. That doing the multiply and the divide and the addition and a return, that all those things are, like, one unit. In this case, I’m gonna call it a penny, right? It took a penny to do each of those operations. That’s not really accurate to be fair. There’s some operations that are more expensive at the low level than others, but we’re gonna be pretty rough about our estimate here in this first step here. So there’s a multiply, there’s a divide, there’s an add, there’s a return, right? And so I have, like, four statements worth of stuff that goes into asking a temperature to be converted. The other thing you’ll note is that, does it matter if the temperature is high or low? If we ask it to convert Celsius of zero, Celsius of 100, Celsius of 50 it does the same amount of work no matter what. So actually that the – for whatever inputs, right? You give it this function pretty much will take the same amount of time. That’s good to know, right? There’s certain functions that will be like that.

So we would call this a constant time function. It’s, like, no matter what you ask it to convert it takes about the same amount of time. That doesn’t mean that it takes no time, but it is a reliable performer of relative inputs. So let’s look at this guy. This is one that takes in a vector and then it sums all the values in the vector and then divides that sum by the total number of elements to compute its average. Okay. So, again, kind of using this idea that will everything you do cost you a penny? How much is it gonna cost you to make a call to the average function?

Well, this is a little tricky because, in fact, there’s a loop in here, right? That’s executed a variable number of times depending on the size of the input. So first let’s look at the things that are outside the loop, right? Outside the loop we do things like initialize the sum, we initialize I before we enter the loop, right? We do this division here at the bottom and this returns. There are about four things we do outside of getting into and then iterating in the loops. So just four things we pay no matter what. Then we get into this loop body and each iteration through the loop body does a test against the I of the size to make sure that we’re in range, right?

Does this addition into the sum and then increments I. So every iteration has kind of three statements that happen for every element in the vector. Zero, one, two, three, and whatnot. So what we get here is a little bit of constant work that’s done no matter what and then another term that varies with the size of the input. If the vector has ten elements, right? We’ll do 30 steps worth of stuff inside the loop. If it has 100 elements we do 300. In this case, right? For different inputs, right? We would expect the average to take a different amount of time. Yeah?

Student: If n, the vector size MP though we still initialize I?

Instructor (Julie Zelenski):We still initialize I. So I was actually counted in here, right? The init I, right? That was in there. We actually have one more test though. We do a test and then we don't enter the loop body. So there's actually one little piece maybe we could say that there's actually like three n plus five. There's one additional test that doesn't enter into the loop body. We're gonna see that actually that we're gonna be a little bit fast and loose about some of the things that are in the noise in the end and look more at the big leading term, but you are right. There is one more test than there are alliterations on the loop. That last test that has to fail.

The idea of being that, right? We have three statements per thing. A little bit of extra stuff tacked onto it if we double the size of the vectors. So if we compute the average of the vector that has 100 elements and it took us two seconds. If we put in a vector that's 200 elements long, we expect that it should about double, right? If it took two seconds to do this half-sized vector, then if we'd have to do a vector that's twice as long we expect it's gonna take about four seconds. And that prediction is actually at the heart of what we're looking at today. Is trying to get this idea of, well, if we know something about how it performs relative to its input can we describe if we were to change the size of that input to make the maze much bigger or to make the vector much smaller?

What can we predict or estimate about how much time and memory will be used to solve the problem using this algorithm? So here is one that given a vector computes the min and the max of the vector and it does it with two loops, right? One loop that goes through and checks each element to see if it's greater than the max it seems so far and if so it updates the max. And similarly almost identical loop here at the bottom that then checks to see if any successive element is smaller than the min it's seen so far and it updates the min to that one. Okay.

So a little bit of stuff that happens outside the loop, right? We init I two times. We init the min and the max, so we've got, like, four statements that happen outside. And then inside the loop, right? We've got a test and an assignment, a comparison, an increment, and we have two of those loops, right? And so we have a little bit of stuff outside the loops, about eight instructions worth that happens per every element. So we look at it once to see if it's greater than the max. We actually look at everything again, right? To see if it's the min. I'm gonna use this actually as written, right? You might think, well, I could make this better, right?

I could make this better by doing these two things together, right? Inside that loop, right? So that while I'm looking at each element I can say, well, if it's the max or if it's the min, right? If I look at it just once I can actually kind of do both those comparisons inside the loop and what we're gonna see is that in terms of kind of this analysis that's really not gonna make any difference. That whether we do a loop over the vector once and then we go back and loop over it again or whether we do twice as much stuff to each element inside one loop, it's for the purposes of the analysis we're looking at it ends up coming down to the same things. Which is, yeah, there is something that depends on the size of the input directly. Which is to say that if it were it will increase linearly with the change in size.

So I have these numbers, like, four statements for the Celsius to Fahrenheit. Three n plus four. Eight n plus four. These tell us a little bit about, well, will get extremes always take more time than average or C to F? That given the way those numbers look it looks like, well, if you plugged in the same value of n you'd have the same size vector. That it should take more time to compute get extremes versus compute the average. We're gonna discover that, actually, we're not gonna, actually, try to make that guarantee. That what we're – we're not really so much interested in comparing two algorithms and their constant factors to decide which of these two that kind of look about the same might be a little bit faster. What we're gonna try and look at it is giving you kind of a rough idea of the class an algorithm fits in. What its growth term, the kind of prediction of its growth, is.

In this case, both averaging get extremes in terms of growth, both have a leading term that depends on n with some other noise and a little bit of a constant thrown in the front. That means that both of them, right? We'd expect to grow linearly in a straight line based on the change in n. So if you doubled the size of the thing then average would take twice as long as it previously did. So should get extremes. But the data point for does average of a vector of 100 elements take the same amount of time as get extremes of 100 is not what we're looking at predicting, right? We're trying to just tell you things about average against itself over different ranges of inputs.

So if we know that average takes two milliseconds for a 1,000 elements. If we double the size of that we expect it to take twice as long, four milliseconds. If we increase it by a factor of ten we expect to increase the time by a factor of ten, right? So it should go up to 20 milliseconds. Get extremes might take more or less for a particular element. We expect it probably will take more because it does a little bit more work per element, but we really want to time it to be confident about that rather than making any estimation here.

So in terms of what's called big O notation we're gonna see that we're going to kind of take those statement counts and we're gonna summarize them. That we can do a little bit of niggling to figure out what those numbers are, but what we're gonna then do is take the leading term, the largest term with the largest coeff – value of the input number, in this case n. Ignore any smaller terms and then drop all the constants, all the coefficients, right? If you know it takes n over 2, it's just gonna be n. If you know that it takes ten statements, then it's gonna just be constant if there's no term of n in there. If it takes 10n then it's still n. 10n and 2n and 25n and 1/15n all end up just being n.

So we might have this idea that we've estimated the time using n and having those constants in. Well, when it comes time to describe it in terms of big O, we focus on what's the – oh, that, the subscript got lost on that. And that just makes no sense as it is right there. I will fix it in a second. So the 3n plus 5, right? The leading term is n, the coefficient 3 gets dropped. It's just linear. We expect that as we change the input the time will change linearly with that change. The 10n minus 2, same class. O of n. Even though it didn't have kind of the same constants coming into it, right? It has the same growth pattern that they both are lines.

The slope of them might be slightly different, but in terms of big O we're being pretty loose about what fits into this class. 1/2n squared minus n, the leading term here is the n squared. The n, subtract it off. When n is small n squared and n are actually kind of in range of each other, but then what we're looking at is in the limit. As n gets very, very large, n gets to be 1,000, n gets to be – n squared is a much bigger number, right? When n is a million then n itself was and so as we get to these larger points kind of out into the limit this term is the only one that matters and this is just a little bit of the noise attached to it. So that's why we're gonna summarize down to that.

What this one is supposed to say, and it actually is incorrect here, it just should be two to the n and n to the third power, which summarizes to two to the n. Two to the n grows much, much more rapidly than n cubed does and so as n gets to be a very large number. Even a fairly small number, you know, two to the tenth, right? Is all ready 1,000. Two to the 20th is a million, which is much bigger than these guys over here. So as we get to the long run it must be much bigger. I'm gonna put a note to myself though to fix that before I put that on the web.

Student: It should be O to the 2n?

Instructor (Julie Zelenski):Yeah. It should be O to the 2n. So that should be two to the n, n to the third, and my two to the n there. My subscripts got blasted out of that. And so we're trying to describe this growth curve, like, in the limit, right? We're avoiding the details when they don't matter and they don't matter when n gets big enough, right? That only the leading term, right? Is predicting without this other stuff kind of just ignoring it. So this is very sloppy math, right? So those of you who are more trained in the, sort of, mathematical sciences may find this a little bit alarming. Which is just how kind of fast and loose we're gonna be. The only way all these terms left and right just kind of summarizing down to, okay, here's this leading term and how it relates to n. Everything else, right? Totally uninteresting.

We could have a function, for example, that did 1,000 conversions of Celsius to Fahrenheit, but if every time you call it does 1,000 conversions that means no matter what input you give to it doesn't change. That's considered O of one or a constant time. Similarly, right? For doing average. If we did an average that somehow operated over the vector one time and another one that did it ten times or 20 times looked at each element 20 times, right? They would still both be O of n. Whatever time they took, right? On a vector of a certain length we double that length. We'd expect to double the time. More formally, right? In terms of what the math really means is that we say that O of F of n, so O, if it's big O of some function, it describes an upper bound on the time required.

Meaning that for sufficiently large values of n and some constant C that we get to pick that that would bound the curves. So if you imagine what the real curve looks like, it grows and maybe it flattens out or maybe it goes up very sharply. That what C of F of n gives you kind of some upper bound of that. A curve that stays above it at – for at some point of n and the limit, right? Will dominate it from there to affinity. So describing kind of where it's gonna hit at that upper bound gives us some measure of what's going on. Okay.

So we can use this to predict times. That's probably what's most valuable to us about it is knowing that I have an n – a linear algorithm. So an O of n algorithm we'll call linear. And n squared algorithm I'll call quadratic. N to the third I'll called cubic, right? That's gonna tell us that those curves, right? You know what a line looks like. Well, you know what a parabola looks like coming out of an n squared where it's growing much more sharply and early kind of making the decision. So it might be, right? That if I know that it takes three seconds to do something for 5,000 elements then I have a linear algorithm. That 10,000 elements, right? Should take twice as long. 20,000 should take another, a factor of two on that. So 12 seconds worth. That I may have an n squared algorithm.

So one that I expect to actually perform more poorly in the long run, right? This n squared versus n tells you that it's gonna more sharply grow. That for some values of n in simply 5,000 is the case where the n squared algorithm actually outperforms it in a small case. That's not uncommon actually. But, right? As it grows, right? As I get to larger and larger values of n the fact that it is going by factor of four, right? The square of the doubling as opposed to linear means it's gonna quickly take off. And so I take my 5,000 elements that took two and a half seconds. I put an input that's twice as large into it. It's gonna take a factor of four, right? From there. And if I go from 10,000 to 20,000 another factor of four is gonna bring that up to 40 seconds or so compared to my more modestly growing linear algorithm there.

So if I was facing some task, right? Where I had an option between a linear algorithm and a quadratic algorithm it's telling me that in the long run the quadratic algorithm for sufficiently large inputs, right? Is going to bog down in a way that the linear one will be our kind of strong performer in the larger cases. So some algorithms actually have a variable run time expectation that you cannot say with all assuredness how much time it will take because actually depending on the input and the characteristics of the input it may do more or less work. So average looks at every element in the vector and sums them all up and does the division. It doesn't matter what elements are in there. It's very reliable in that sense.

Something like search. So this is a linear search function that given a vector of names and a particular key just walks the vector looking for a match. If it finds it, it returns true. If it exhaustively searched the whole thing and didn't find it, it returns false. Okay. So we've got what looks like an O of n loop that we'll look at the things, but there are some cases, right? Where it just doesn't do very much work at all. For example, if it finds the key in the very first spot, right? If I'm looking for Bob and Bob is the first thing in there, then I immediately return true and I do no more work. And so it doesn't matter whether Bob was followed by a million more names or ten more names, right? That, in fact, it is a constant time operation to just access that first element and return it. Similarly for other things that are close to the front.

It looks at a few of them and it's done. And so we would call those the best case of the algorithm. So we can divide this into things. It's like, well, what of the best case input, right? What can we expect out of the performance? Well, the best-case input would be it's found in the first few members. In which case it's an O of one algorithm for those situations. That's not often very interesting to say, well, here's a particular input that causes it to immediately be able to calculate something in return. Yeah. Not so interesting. So it's worth knowing that such a thing exists, but it turns out it's not likely to tell us a lot about the general performance of this algorithm.

If it's in the middle or it's in the last slot, right? We're gonna be looking at a lot of the elements to decide that it's there or not. In the absolute worst case, right? So what's the input that causes it to do the most amount of work is the one where it doesn't find it at all. Where it looks at every single element, never finding the match, and then comes out and returns that false. And the worst case, in this case, is a fairly likely thing to happen, right? We're searching because we happen to believe it might not be there and that gives us this upper bound on how bad it gets. So in the worst case it looks at everything and it is definitely O of n. So we have this kind of constant best case, an O of n worst case, and then our average case is gonna be somewhere in the middle there. This actually is a little bit harder to predict or to compute precisely in most situations because you have to know things about, well, what are the likely range of inputs? So typically the definition is that it's averaged over all the possible inputs. Well, given the search it's pretty hard to say what are all the possible inputs for it? It's like you can make some assumptions, like, well, all in – all permutations of the list are equally likely and the name is in the list about half the time, let's say.

You can just – you have to make up some parameters that describe what you believe to be the likely inputs and then you can say, well, if it were in the list, if it's equally likely to be in the front as in the back, then on average it's gonna look at n over two. It's just as likely to be in all the front slots. So, let's say, if you were gonna call it n times and the name you were looking for was going to be in each individual slot exactly one time, then the total random perfect permutation case. So then it would have looked at n over two of them. Sometimes it looked at one, sometimes it looked at n minus one, sometimes as it looked at n over two, sometimes n over two plus one, and whatnot. That over all of them it looked at about half.

And then if there was another set of calls to search for it, right? Where it never found it, it would be looking at n, right? And so you can say, well, sometimes we look at n over two, sometimes we look at n. On average we're looking at about three-quarters of them, right? In big O, since we get to be a bit sloppy here, we can say, well, this all just boils down to being linear. That O of n still described to the growth in the average case, which is the same number we got for the worst case. So this is a little bit tricky, but this is actually the one that's probably the most interesting, right? Which is just in normal practice, how is this thing gonna behave?

So the last little thing I need to complete before we can go on and start applying this to do something interesting, is to talk a little bit about how we do recursive algorithms because they're a little bit trickier than the standard iteration and counting. So the counting statements are kind of saying, oh, I've got this loop followed by this loop and this thing happening, right? Will get you through the simple cases. Well, what do we do when we have a recursive algorithm, right? Well, we're trying to compute the time spent in solving something that, in affect, is making a call to the same algorithm and so we're gonna likely have some recursive definition we need to work through.

So this is the factorial function. If n equals zero, return one, otherwise return n times the factorial n minus one. We're interested in doing kind of the statement counts or kind of summarizing the time for an input whose value is n. And so what we write down is what's called a recurrence relation that largely matches the code in terms of saying how much work is done in the base case? In the different cases that are being handled? Typically there is a base case or two where you say, well, if it's in these cases, right? We do these easy things. So if n is exactly zero then we do O of one worth of work, right? We do a little bit of constant work here. We do a comparison and a return.

In the otherwise case when it is not zero, then we do a little bit of work. In this case, a return, a multiply, a little bit of constant work plus whatever time it takes to compute the factorial of n minus one. So we have T of n defined in terms of T of n of something else, right? Which is exactly what we'd expect in a recursive function. This is called a recurrence relation, so that solving for T of n means working with a side that refers back to that same T, but on a smaller version of the problem in n over two and n minus one. Some other variation of that recursive call. So let me show you how we make that go into closed form.

The idea – and actually I'm just gonna do this on the board because I think it's easier if I just write what's going on and you can watch me develop it. So I have this recurrence relation. Even equals one if n equals zero and then it's one plus T of n minus one otherwise. So I'm trying to solve for T of n into a closed form. So I'm gonna start with it's non-recur – the non-base case form. So the recursive step. And then what I'm gonna do is I'm actually just gonna reply repeated substitution to take this term and expand it out. So I know it's one plus whatever it takes to do T of n minus one. So if I go back over here and I say, well, T of n minus one must be one if n minus one equals zero or it's one plus T of n minus two. So kind of plugging it into the original formula and getting the expansion one layer in.

So I can say, well, this is one plus one plus T of n minus two. And if I apply that again, right? I should get another term of one. So after I have done this I times then I will have a bunch of ones added up together and I will have subtracted an I from T down to some term. So I'm just gonna – each of these represents kind of like this is a little bit of work from the n, which does the n minus one, which brings the quality of my two. So each of the cul's kind of in the stack frame has a little bit of a contribution to add to the whole thing and then what we're looking at is how many times do we have to do that expansion and substitution, right? Before we hit this base case, right? Where T of n exactly equals one.

So we're looking for the case where n – actually it's n equals zero. So where n – so I want to do this until n minus I equals equals zero. Okay. So I need to have done this I times where I is n. So if I say I set I equaled to n, then I'll have one plus one plus one n times, and then I have plus T of the n minus n over here, which is my T subzero. T subzero, right? Immediately plugs into that base case and says, well, there's just one more thing to do when you get to that and so what I basically have here is n plus one. So n multiplications plus one little tack on for the base case, which in terms of big O is just linear. A little bit of extra work in the noise there, but that means that kind of as it seems more predictable, right? That factorial over particular input, right? Is linear in the number you ask to compute the factorial.

The factorial of ten takes ten multiplications, right? The factorial of 20 takes 20 multiplications. So if you change the size of your input, right? You double it; it should take twice as long. However much time it cost you to compute the factorial of ten is gonna take twice as much time to compute the factorial of 20. Okay. That kind of makes sense, but it's good to know kind of how I can do this math, right? To work this out, right? So this idea of taking the term, repeatedly substituting and expanding, generalizing my pattern, and say, well, after I substitutions worth where am I at? And these correspond and kind of thinking about the recursion tree. What calls are made and then how deep does the recursion go before I hit the base case and that tells me how to stop that expanding and then substitute back in for the base case to compute my total result.

I'm gonna do another one, so if you want to just watch and we'll do the math for this together, too. This is the Towers of Hanoi example that moves the tower away of n minus one off, moves the single disk on the bottom and then moves that tower back on. And so the recurrence that we're working with is one when n equals zero. So when n equals zero, right? We have a zero height tower to move and we actually do no work in the function, right? We just do that test and return, so if n equals zero there's no work to be done. So that's the easy case for us, right? Is when it's zero do no work. Otherwise, right? We move the bottom most disk. So we do a little testing and moving of that disk. We'll call that the constant amount of work that's in the function itself.

And it makes two recursive calls each of a tower of height n minus one. So otherwise, right? Two calls, which gives a little clue to what the tree looks like. It'll branch twice at each stop and then it's one closer to that base case gives us a sense that the recursion depth is likely to be linear here. So let me go through the process of making this work. I've got T of n equals one plus two, T to the n minus one. So then I take T to the n minus one and I plug it in over here to get one plus two T to the n minus two. This whole thing is in a – multiplied by two though because I have two of those, right? From the original call which then itself made two. So, in fact, if I multiply through, right? I've got one plus two plus four T to the n over two.

If I apply my substitution again, one plus two plus four times T to the n minus two is one plus two, T to the n minus three, and then let the multiplication go through again. One plus two plus four plus eight T to the n minus three. And so each expansion of this, right? Is causing the number of towers that are being moved around to go up by a factor of two.

So each time I do this, right? I went from two towers to move to four towers to eight towers, but those towers each got one shorter in the process. So kind of telling us a little bit about what the recursion tree looks like, right? Is there is a branching factor of two all the way down and that the depth of this thing is gonna bottom out linearly because this was a tower by ten, nine, eight, seven, and so on down to the bottom.

So if I imagined this happened I times, so to generalize my pattern. I've got one plus two plus four plus two to the I. So after I've done this that number of times, right? Actually it's two I minus one plus two to the I, T n minus I. So I have subtracted I off the heights of the tower. I have gone up by a factor of two each time I did that and then I have these sums in the front, which represent kind of the single disk that got moved in there. One plus two plus four plus eight plus so on down to there. And so the place I want to get to, right? Is where n equals zero. So I actually want to set n, I equal to n here. I wrote that backwards, let's say I equals n. I plug that in I've got one plus two plus four plus all the powers of two to the n minus one plus two to the n, T to the n minus n, which is T to the zero, which is just one.

So what I've got here is following along. Is T to the n is one plus two plus four plus two to the n, right? So two to the n minus one plus two to the n times one. So I've got the geometric sum here. You may or may not all ready know how to solve this one, but I'm just gonna go ahead and solve it in front of you to remind you of how the process works. I've got the powers of two. One plus two plus up to the n. So that's the term I'm looking for. I want to call this A just to mark it. And then what I'm gonna compute for you is what two times A is and I'm gonna write it underneath. So if I multiply this by two I have one times two, which is two. Two times two, which is four, right? Four times four at two, which is eight. And so on.

And so I should get basically the same sequence of terms, but shifted over by one. I don't have the factor of one and I do have an additional factor of two to the n times another power of two at the end. So I've got the whole term kind of shifted over by one. That's my two A so that's the same thing I'm looking for, but doubled. And then I'm gonna basically take this thing and I'm gonna add negative A to two A. So subtracting A off of two A so that all these terms will go away. If I have this minus this, this minus this all the way across, right? The terms I'll be left with is two to the n plus one minus one is A subtracted from two A so the two to the T to the n that I'm looking for is two to the n plus one minus one. That's my term there.

And if I summarize in my big O way, ignoring the constants, throwing away these little parts of the terms, right? That are in the noise. That what we really have here is something that grows, right? Exponentially by about a factor of two for each additional disk added to the tower. So however much time it took to move a tower of height ten, right? A tower of height 11 we expect to take twice as long. So not growing linearly at all, right? Much, much, much sharply growing, right? What exponential growth looks like. Two to the n, right? Even for small values of n. Ten, 20, right? Is up into the millions all ready in terms of disks that need to be moved.

So this sort of strategy, right? Is good to know. When you're looking at this recursive things, right? Kind of having this analysis. It says, okay, well, where did I start from? What's the time? And then just being pretty mechanical. Sometimes you can get to something here that requires a little bit of algebra to work out, but the most common patterns, right? You'll start to recognize over and again, right? And this one, right? The fact that it's branching by a factor of two telling you that at the kind of base case, right? You're gonna expect to see a two to the n expansion.

So just to give you a little bit of some idea that you can use big O to predict things based on getting some run time performance for some small values of n and then what you'd expect to see for larger values. So I have three different algorithms here. One that was just logarithmic in terms of n, so it should divide the input in half and kind of work on it. Something like binary search actually fits that profile. Something that's linear. Something that's n log n and something that's n squared. Then for different values of n I've plugged in actually the first ones I ran and then actually I – some of the other ones I had to estimate because they were taking way too long to finish.

So that for an input of size ten, all of them are imperceptible. These are in terms of seconds here. Taking fractions of seconds. This is on my machine that is running at a couple gigahertz, so it's got about a million instructions per second. You up that by a factor of ten, right? And they're still kind of in the subsecond range, but you can see that from the n squared algorithm took a bigger jump, let's say, than the linear algorithm did, which went up by a factor of ten almost exactly. Going up by another factor of ten and, again, sort of climbing, right? 10,000, 100,000, a million, a trillion. You get to something that's like 100,000, right? And an algorithm that's linear, right? Is still taking a fraction of a second.

But something that was quadratic now has climbed up to take several hours. So even for inputs that don't seem particularly huge, right? The difference between having an algorithm that's gonna operate in linear time versus quadratic is gonna be felt very profoundly. And as you move to these larger numbers you have things that you just can't do, right? You cannot run an n squared algorithm on a trillion pieces of data in your lifetime. It's just too much work for what the input is. Clinton?

Student: Yeah. How did it go down for a billion from like a million? From –

Instructor (Julie Zelenski):Oh, you know what, that just must be wrong. You're totally right. That should definitely be over some. Yeah. Well log n should really be – I think that when I copied and pasted from these terminal window. So, of course, I must have just made a mistake when I was moving something across, but you're totally right. This should be going up, right? Ever so slightly, right? This algorithms going very, very slowly logarithmic function, right? Almost a flat line, but that definitely should be up a little bit, right? From where it is. When you get to a trillion, right? Even the linear algorithm is starting to actually take some noticeable time, right? But things that are logarithmic still running very, very slowly. So this is an example of binary search.

Binary search operating on a million or trillion items is still just in a flash, right? Telling you whether it found it or not. Doing a linear search on a trillion or billion is several minutes' worth of my old computers time. And so just another way to look at them, right? Is to think about what those things look like in terms of graphed on the Cartesian plane that a constant algorithm is just a flat line. It takes the same amount of time no matter how big the input is. It's pretty small values of n down here, so it just shows the early stages, right? But logarithmic, right? Almost a flat line itself. A little bit above constant, but very, very slowly growing. The linear scale, right? Showing the lines and then the n squared and to the n showing you that they're kind of heading to the hills very quickly here. So even for small values of n they are reaching into the high regions and will continue to do so as that will be more pronounced, right? As you move into the higher and higher values of n.

All right. I get to tell you a little bit about sorting before we go away. Because some of them I'm just setting the stage for, like, okay, how can we use this to do something interesting? Knowing about how to compare and then contrast. That it tells you something. Let's try to solve a problem, right? That lends itself to different approaches that have different algorithmic results to them that are worth thinking about. So sorting turns out to be one of the best problems to study for this because it turns out sorting is one of the things that computers do a lot of. That it's very, very common that you need to keep data in sorted order. It makes it easier to search. It makes it easier to find the minimum, the maximum, and find duplicates. All these sort of things, right? Like by having, keeping it inserted or it tends to be more convenient to access that way.

It also makes a bunch of other things, other properties about the data, easy to do. If you want to find the top ten percent, right? You can just pick them off, right? If they've all ready been sorted. You want to group them into buckets for histograming, right? All these things actually are enabled by having them all ready be in sorted order. So it turns out that a lot of times that data that you get from other sources tends to first need to be put in sorted order before you start doing stuff on it. Okay.

Well, there's lots of different ways you can sort it. There are simple ways. There are complicated ways. There are fancy ways. There are ways that are dumb, but easy to write. Ways that are smart, but hard to write. And everything in between. So we're gonna be looking at it in terms of sorting vectors because that's probably the most common data structure that needs the sorting. But you can also imagine taking the same kind of algorithms and applying them to different kinds of data structures you have. Like starting a linked list. Okay.

So I'm gonna show you a sorting algorithm. It's probably the simplest and the easiest to write. If somebody – if you were stuck on a desert island without a textbook, but you happen to have a compiler and you needed to write a sorting algorithm to get your way off the island. It comes up all the time. This is probably the algorithm you're gonna use. It's called selection sort. And the idea behind selection sort is that it's going to select the smallest element and put it in the front. So if I have a big stack of test papers and I want to sort them in order of score, then I'm gonna go through there and find somebody who got the absolute lowest score. It's said, but true. Somebody had to be there. So I kind of work my through it and maybe I hold my finger on the one that I've seen that looks smallest so far.

So this one's a 40, oh, well, this one's a 38. Okay. Oh, look there's a 35. Oh, look, there's a 22. Nobody gets these scores in my exams, I'm just kidding. And then finally get down to the bottom. You say, okay, that 25, that was the smallest. I have a hold of it, right? And I'm gonna basically take that and bring it to the front. And in terms of managing a vector that actually there's a slight efficiency to be had by instead of kind of pulling it out of the stack and sliding everything down I'm actually just gonna swap it with the one that's in the very front. So whoever was the current first one is gonna booted to pull in this one and I'm gonna put them back where this other one came from.

So I have moved the very smallest to the top of the vector. Then I just do the same thing again, but now excluding that smallest one. So I've all ready seen that one and kind of set it aside. I start looking at the remaining n minus one and I do the same thing again. Find the smallest of what remains, kind of just walking through, going to find it, and swap it down into the second slot, and so on. A little bit of code. Doing it kind of the way I described it, right? Of tracking the minimum index, right? Looking from here to the end, so this four loop in the middle here starts at the current position and only considers from here to the very end of the vector and then finds any element which is smaller than the one I'm kind of currently holding my finger on and then when I'm done after that inner loop has executed I will know what the min index is and then there's a spot function here that just exchanges those two things.

The I slot now gets replaced with the one at min index and, again, exchanged in the contents of the array. I just do that n minus one times and I will have done it all. I'd like to show you animation, but I guess I'll just wait for that until Wednesday. We will watch it doing it's kind of thing in progress and you can kind of be thinking about for Wednesday what other ways you might be able to sort data than just this strategy.

[End of Audio]

Duration: 50 minutes