MIT OpenCourseWare
http://ocw.mit.edu

6.046J Introduction to Algorithms, Fall 2005
Transcript – Lecture 2

My name is Erik Demaine. You should call me Erik. Welcome back to 6.046. This is Lecture 2. And today we are going to essentially fill in some of the more mathematical underpinnings of Lecture 1. So, Lecture 1, we just sort of barely got our feet wet with some analysis of algorithms, insertion sort and mergesort. And we needed a couple of tools. We had this big idea of asymptotics and forgetting about constants, just looking at the lead term. And so, today, we're going to develop asymptotic notation so that we know that mathematically. And we also ended up with a recurrence with mergesort, the running time of mergesort, so we need to see how to solve recurrences. And we will do those two things today. Question? Yes, I will speak louder. Thanks. Good.

Even though I have a microphone, I am not amplified. OK, so let's start with asymptotic notation. We have seen some basic asymptotic notation. I am sure you have seen it in other classes before, things like big O-notation. And today we are going to really define this rigorously so we know what is true and what is not, what is valid and what is not. We are going to define, and unfortunately today is going to be really mathematical and really no algorithms today, which is sort of an anticlimax. But next lecture we will talk about real algorithms and will apply all the things we learned today to real algorithms.

This is big O-notation, capital O-notation. We have f(n)=O[g(n)]. This means that there are some suitable constants, c and n_o, such that f is bounded by cg(n) for all sufficiently large n. So, this is pretty intuitive notion. We have seen it before. We are going to assume that f(n) is non-negative here. And I just want f(n) to be bounded above by g(n). We have seen a bunch of examples, but something like 2n^2=O(n^3) defined.

And roughly this means if you drop leading constants and low order terms then this is less than or equal to that. So, big O corresponds roughly to less than or equal to. But this is the formalization. Another way to think of it formally, a funny thing about this notation is it is asymmetric. Normally, you think of equality being symmetric. If A=B then B=A. But it's not true here. We do not have n^3 being big O of n^2.

We don't even have big O of n^3 equaling n^2. So, we will see exactly what that means in a second. But before we get there, this is a bit bizarre notation and you should always think about what it really means. Another way to think about what it really means is that f(n) is in some set of functions that are like g. You could define big O[g(n)] to be a set of functions, let's call it f(n), such that there exist constants. They are the same definition, I think, fancy here, c and n_o, such that we have the bound f(n) is between zero and cg(n).

It is a bit of a long definition, and that is why we use the notation, to avoid having to write this over and over. You can think of instead of n^2 being equal to big O of n^3, what we really mean is that 2n^2 is in the set big O(n^3). When we write equal sign, we in some sense mean this in the set, but we are going to use equal sign. You could write this. And occasionally you see papers that write this, but this is the notation that we are going to use. That has the consequence the equal sign is asymmetric, just like this operator. We have some nifty ways that we actually use big O-notation.

And it is using it as a macro. By the way, we have a lot to cover today, so I am going to go relatively fast. If anything is unclear, just stop, ask questions, then I will slow down. Otherwise, I will take this as all completely obvious and I can keep going at full speed. The convention, this is intuitive, I guess, if you do some macro programming or something, but it's a bit more mathematical.

We have defined big O-notation and it equals big O of something. And so we have only defined big O when on the equal sign we have big O of some function. But it is useful to have some general expression on the right-hand side that involves big O. For example, let's say we have f(n) = n^3 + O(n^2). This is attempting to get an error bound. This is saying f(n) is basically n^3 but there are these lower order terms that are O(n^2). And so this means that there is a function, shorthand for a function, h(n) which is in O(n^2) or equals O(n^2) such that f(n) = n^3 + h(n).

It is saying that there are some lower order terms that are bounded above by some constant times n^2 for sufficiently large n, and that is what is here. And then f(n) equals, now this is a true equality, n^3 plus that error term. This is very useful here. Essentially, I am expressing what the lead constant is and then saying well, there is other stuff and it's all at most n^2. Saying that f(n) therefore is also order n^3, but that is a bit weaker of a statement. This is a bit more refined. We won't need to use this too often, but it is useful. Sometimes we will see, like in last class we even had a big O inside a summation. So, you can use them all over the place. The point is they represent some function in that set.

A bit less intuitive, and this is more subtle, is what it means to have big O on the left-hand side. It means the same thing, but there is some convention what equality means. And this is why equal sign is asymmetric. You should read equals like "is". Is means that everything over here is something over here. So, there is an implicit for all on the left-hand side and there exists on the right-hand side. This is a true statement. Anything that is n^2 + O(n) is also O(n^2), but not the other way around. So, this is a bit asymmetric. If you think about it, this is pretty intuitive but it is subtle so you should be careful.

This says for any expansion of the macro on the left-hand side, which should be f(n), there is an expansion of the macro on the right-hand side such that we get equality. And what this allows you to do is if you have a chain of equal signs relations, a chain of "is"s, then the very first one is equal to or bounded by the very last one. So, you can chain equal signs the way you normally would. You just cannot flip them around. Good. So, that's big O-notation. Any questions about that?

So, big O is great for expressing upper bounds. But we also want to talk about lower bounds. For algorithms, we usually care about upper bounds on their running time. Running times at most n^2 is at most n log n up to big O, but sometimes we need to express functions that are at least some quantity. For example, we will show that sorting requires at least n log n time in some model. So, we need some other notation for that. And the notation is big Omega-notation.

And it is pretty symmetric. I will just write out the set definition here. And we are going to write f(n)= big Omega[g(n)] to mean f(n) is at least some constant times g(n) -- -- for sufficiently large n. So, I am basically just reversing the inequality relation between f and g, nothing surprising, just to have it there. A random example, and now we will get a little bit more sophisticated, root n= big Omega(lg n).

And you should read this that up to constant factors root n is at least log n for sufficiently large n. So, omega sort of corresponds to greater than or equal to. Let me give you some analogies. We have big O, we have big omega, this is less than or equal to, this is greater than or equal to. And I am going to fill in some more here in a moment. It's nice to have all the usual operators we have. Normally we have strict less than, strict greater than and equal sign. And we want those sort of analogs in the asymptotic world where we ignore constant factors and ignore lower order terms. We have, for example, big Theta[g(n)]. This is a capital theta which means you write the horizontal bar in the middle as opposed to all the way through.

I didn't invent Greek, so that is the way it is. Theta means that you are less than or equal to and you are greater than or equal to up to constant factors, so it is the inner section of these two sets, big O and big Omega. That is sort of like equal sign but, of course, this is very different. You have things like n^2 is big Theta of 2(n^2) because you ignore constant factors, but all of these other relations, OK, n^2 + O(n) = Theta(n^2), but this does not hold with theta because square root of n is really asymptotically bigger than log n. And some of the other examples we saw like n^2 versus n^3, those don't hold with T.

And we have some strict notation which are the little o-notation and little omega-notation. There is no little theta because there is not notion of strict equality versus unstrict equality. Little o is going to correspond roughly to less than and little omega is going to correspond to greater than. This is a notation you will just have to get used to. And I am not going to define it precisely here because it is almost exactly the same. The difference is that instead of saying there exists constant c and n_o, you have to say for every constant c there exists a constant n_o. The relationship between f and g, this inequality must hold for all c instead of just for 1.

And so n_o can now depend on c. You can assume that really n is sufficiently large, but this gives you a strict inequality. No matter what constant you put here, in front of g, let's say we are doing little o, f will be still less than c times g for sufficiently large n. We have some random examples. We are again ignoring constants. n^2 is always less than n^3 for sufficiently large n. And it is a bit subtle here. I mean in order to prove something like this, it will become intuitive after you manipulate it a little bit. You have to figure out what n_o is in terms of c. I think it something like 2/c.

If we have less than or equal to, that should be right. As long n is at least this big, no matter how small of a c, you should think of c here as being epsilon now, in the usual epsilon and deltas. No matter how small c gets, still I can bound n^2 in terms of n^3, upper bound, but whenever you have theta you do not have either of these relations. For example, ½n^2 = Theta(n^2) and it is not little o(n^2) and it not little omega(n^2) because it is exactly n^2. You will get some sense in order relation out of this, although there are some messy behaviors as you will see in your problem set. Any questions about asymptotic notation? That is the quick rundown. Now we are going to use it to solve some recurrences.

Although we won't use it that much today, we will use it a lot more on Wednesday. OK. We will move onto the second topic of today, which is solving recurrences. You have probably solved some recurrences before in 6.042 or whatever discrete math class you have taken. We are going to do more and have some techniques here that are particularly useful for analyzing recursive algorithms, and we will see that mostly on Wednesday. There are three main methods that we are going to use here for solving recurrences. The first one is the substitution method.

There is no general procedure for solving a recurrence. There is no good algorithm for solving recurrences, unfortunately. We just have a bunch of techniques. Some of them work some of the time, and if you are lucky yours will work for your recurrence, but it is sort of like solving an integral. You have to just know some of them, you have to know various methods for solving them. It is usually easy to check if you have the right answer. Just like with integrals, you just differentiate and say oh, I got the right answer. And that is essentially the idea of substitution method.

Substitution method will always work, but unfortunately Step 1 is guess the answer. And you have to guess it correctly. That makes it a big difficult. You don't have to guess it completely. You can usually get away with not knowing the constant factors, which is a good thing because we don't really care about the constant factors. You guess the form. You say oh, it is going to be roughly n^2, and so it's some constant times n^2 presumably.

So, you guess that. We are going to figure out the constants. You try to verify whether the recurrence satisfies this bound by induction, and that is the key. Substitution uses induction. And from that you usually get the constants for free. You figure out what the constants have to be in order to make this work. So, that is the general idea. You will see a few examples of this. Actually, the same example several times.

Unfortunately, this is what you might call, I don't know. This is an algorithm, but it uses an oracle which is knowing the right answer. But sometimes it is not too hard to guess the answer. It depends. If you look at this recurrence, T(n) = 4T(n/2) + n, we should implicitly always have some base case of T of some constant, usually 1 is a constant, so we don't really care about the base case.

For algorithms that is always the case. And we want to solve this thing. Does anyone have a guess to what the solution is? Ideally someone who doesn't already know how to solve this recurrence. OK. How many people know how to solve this recurrence? A few, OK. And, of the rest, any guesses? If you look at what is going on here, here you have T(n/2) and let's ignore this term more or less. We have n/2 here. If we double n and get T(n) then we multiply the value by 4. And then there is this additive end, but that doesn't matter so much. What function do you know that when you double the argument the output goes up by a factor of 4? Sorry? n^2,yeah. You should think n^2 and you would be right. But we won't prove n^2 yet. Let's prove something simpler, because it turns out proving that it is at most n^2 is a bit of a pain.

We will see that in just a few minutes. But let's guess that T(n) = O(n^3) first because that will be easier to prove by induction. You sort of see how it is done in the easy case, and then we will actually get the right answer, n^2, later. I need to prove. What I am going to do is guess that T(n) is some constant times n^3 at most, so I will be a little more precise. I cannot use the big O-notation in the substitution method so I have to expand it out to use constants.

I will show you why in a little bit, but let me just tell you at a high level what is important in not using big O-notation. Big O-notation is great if you have a finite chain of big O relations, you know, n^2 is big O(n^3) is big O(n^4) is big O(n^4) is big O(n^4). That is all true. And so you get that n^2 is big O(n^4). But if you have an infinite chain of those relations then the first thing is not big O of the last thing. You have to be very careful. For example, this is a total aside on the lecture notes. Suppose you want to prove that n = O(1). This is a great relation. If it were true, every algorithm would have constant running time. This is not true. Not in Wayne's World notation.

You could "prove this by induction" by saying well, base case is 1 = O(1). OK, that is true. And then the induction step as well, if I know that n-1, so let's suppose that n-1 = O(1), well, that implies that n, which is (n-1) +1, if this is O(1) and 1 = O(1), the whole thing is O(1). And that is true. If you knew that (n-1) = O(1) and 1 = O(1) then their sum is also O(1), but this is a false proof. You cannot induct over big Os. What is going on here is that the constants that are implicit in here are changing. Here you have some big O of 1, here you have some big O of 1. You are probably doubling the constant in there every time you do this relation. If you have a finite number of doubling of constants, no big deal, it is just a constant, two the power number of doublings. But here you are doing n doublings and that is no good.

The constant is now depending on n. So, we are avoiding this kind of problem by writing out the constant. We have to make sure that constant doesn't change. Good. Now I have written out the constant. I should be safe. I am assuming it for all k less than n, now I have to prove it for k equal to n. I am going to take T(n) and just expand it. I am going to do the obvious thing. I have this recurrence how to expand T(n).

Then it involves T(n/2). And I know some fact about T(n/2) because n/2 is less than n. So, let's expand. T(n) = 4T(n/2) + n. And now I have an upper bound on this thing from the induction hypothesis. This is at most 4 times c times the argument cubed plus n. Continuing on here. Let's expand this a little bit. We have n cubed over 2 cubed. Two cubed is 8, so 4 over 8 is a half. So, we have ½cn^3 + n.

And what I would like this to be is, so at the bottom where I would like to go is that this is at most cn3. That is what I would like to prove to reestablish the induction hypothesis for n. What I will do, in order to see when that is case, is just write this as what I want, so this is sort of the desired value, cn3, minus whatever I don't want. This is called the residual. Now I have to actually figure this out. Let's see. We have cn^3, but only ½cn^3 here, so I need to subtract off ½cn^3 to get that lead term correct. And then I have plus n and there is a minus here, so it is minus n. And that is the residual.

In order for this to be at most this, I need that the residual is non-negative. This is if the residual part is greater than or equal to zero, which is pretty easy to do because here I have control over c. I get to pick c to be whatever I want. And, as long as c is at least, oh, I don't know, 2, then this is a 1 at least. Then I have n^3 should be greater than or equal to n. And that is always the case. For example, this is true if c is at least 1. And I don't think it matters what n is, but let's say n is at least 1 just for kicks. So, what we have done is proved that T(n) is at most some constant times n^3. And the constant is like 1.

So, that is an upper bound. It is not a tight upper bound. We actually believed that it is n^2, and it is, but you have to be a little careful. This does not mean that the answer is n^3. It just means that at most n^3 is big O(n^3). And this is a proof by induction. Now, technically I should have put a base case in this induction, so there is a little bit missing. The base case is pretty easy because T(1) is some constant, but it will sort of influence things. If the base case T(1) is some constant.

And what we need is that it is at most c times one cubed, which is c. And that will be true as long as you choose c to be sufficiently large. So, this is true if c is chosen sufficiently large. Now, we don't care about constants, but the point is just to be a little bit careful. It is not true that T(n) is at most 1 times n^2, even though here all we need is that c is at least 1. For the base case to work, c actually might have to be a hundred or whatever T(1) is. So,

be a little bit careful there. It doesn't really affect the answer, usually it won't because we have very simple base cases here. OK, so let's try to prove the tight bound of O(n^2).

I am not going to prove an omega bound, but you can prove an omega n squared bound as well using substitution method. I will just be satisfied for now proving an upper bound of n squared. Let's try to prove that T(n), this is the same recurrence, I want to prove that it is O(n^2). I am going to do the same thing. And I will write a bit faster because this is basically copying. Except now, instead of three, I have two. Then I have T(n) = 4T(n/2) + n. I expand this T(n/2). This is at most 4c(n/2)^2 + n. And now, instead of have 2 cubed, I have 2 squared, which is only 4.

The fours cancel. I get cn^2 + n. And if you prefer to write it as desired minus residual, then I have cn^2 - (-n). And I want this to be non-negative. And it is damn hard for minus n to be non-negative. If n is zero we are happy, but unfortunately this is an induction on n. It's got to hold for all n greater than or equal to 1. This is not less than or equal to cn^2. Notice the temptation is to write that this equals O(n^2), which is true for this one step. cn^2 - (-n), well, these are both order n, or this is order n, this is order n squared. Certainly this thing is O(n^2), that is true, but it is not completing the induction. To complete the induction, you have to prove the induction hypothesis for n with this constant c.

Here you are getting a constant c of like c + 1, which is not good. This is true but useless. It does not finish the induction, so you can sort of ignore that. This proof doesn't work, which is kind of annoying because we feel, in our heart of hearts, that T(n) = n^2. It turns out to fix this you need to express T(n) in a slightly different form. This is, again, divine inspiration. And, if you have a good connection to some divinity, you are all set. [LAUGHTER] But it is a little bit harder for the rest of us mere mortals. It turns out, and maybe you could guess this, that the idea is we want to strengthen the induction hypothesis. We assumed this relatively weak thing, T(k) is less than or equal to some constant times k^2. We didn't know what the constant was, that is fine, but we assumed that there were no lower order terms. I want to look at lower order terms.

Maybe they play a role. And if you look at this progression you say, oh, well, I am getting something like n^2 and the constants are pretty damn tight. I mean the fours are canceling and the c just is preserved. How am I going to get rid of this lower order term plus n? Well, maybe I could subtract off a linear term in here and, if I am lucky, it will cancel with this one. That is all the intuition we have at this point. It turns out it works.

We look at T(n) and this is 4T(n/2) + n as usual. Now we expand a slightly messier form. We have 4[c_1*(n/2)^2 - c_2*(n/2)] + n. This part is the same because the fours cancel again. So, we get c_1*n^2, which is good. I mean that is sort of the form we want. Then we have something times n, so let's figure it out. We have a plus 1 times n, so let's write it 1 minus c_2 over 2 times n. Oops, got that wrong. There is four times a two so, in fact, the two is upstairs. Let me double check. Right. OK. Now we can write this as desired minus residual. And we have to be a little careful here because now we have a stronger induction hypothesis to prove.

We don't just need it is at most c_1*n^2, which would be fine here because we could choose c_2 to be large, but what we really need is c_1*n^2 - c_2*n, and then minus some other stuff. This is, again, desired minus residual. And minus residual, let's see, we have a minus 1 and we have a minus c_2. That doesn't look so happy. Plus c_2, thank you, because that again looked awfully negative. It is plus c_2. I am getting my signs, there is a

minus here and there is one minus here, so there we go. Again, I want my residual to be greater than or equal to zero.

And if I have that I will be all set in making this inductive argument. Office hours start this week, in case you are eager to go. They are all held in some room in Building 24, which is roughly the midpoint between here and Stata, I think, for no particular reason. And you can look at the Web page for details on the office hours. Continuing along, when is c_2 - 1 going to be greater than or equal to zero? Well, that is true if c_2 is at least 1, which is no big deal. Again, we get to choose the constants however we want. It only has to hold for some choice of constants. So, we can set c_2 greater than or equal to 1. And then we are happy.

That means this whole thing is less than or equal to c_1*n^2 - c_2*n if c_2 is greater than or equal to 1. It is kind of funny here. This finishes the induction, at least the induction step. We proved now that for any value of c_1, and provided c_2 is at least one. We have to be a little more careful that c_1 does actually have to be sufficiently large. Any particular reason why? c_1 better not be negative, indeed. c_1 has to be positive for this to work, but it even has to be larger than positive depending. Sorry. I have been going so fast, I haven't asked you questions. Now you are caught off guard. Yeah? Because of the base case, exactly. So, the base case will have T(1) is c_1 time 1 squared minus c_2, we want to prove that it is at most this, and T(1) is some constant we have assumed.

We need to choose c_1 to be sufficiently larger than c_2, in fact, so c_2 has to be at least 1. c_1 may have to be at least a hundred more than one if this is 100. This will be true if c_1 is sufficiently large. And sufficiently large now means with respect to c_2. You have to be a little bit careful, but in this case it doesn't matter. Any questions about the substitution method? That was the same example three times. In the end, it turned out we got the right answer. But we sort of had to know the answer in order to find it, which is a bit of a pain. It would certainly be nicer to just figure out the answer by some procedure, and that will be the next two techniques we talk about. Sorry?

How would you prove a lower bound? I haven't tried it for this recurrence, but you should be able to do exactly the same form. Argue that T(n) is greater than or equal to c_1*n^2 - c_2*n. I didn't check whether that particular form will work, but I think it does. Try it. These other methods will give you, in some sense, upper and lower bounds if you are a little bit careful. But, to really check things, you pretty much have to do the substitution method. And you will get some practice with that. Usually we only care about upper bounds. Proving upper bounds like this is what we will focus on, but occasionally we need lower bounds. It is always nice to know that you have the right answer by proving a matching lower bound.

The next method we will talk about is the recursion-tree method. And it is a particular way of adding up a recurrence, and it is my favorite way. It usually just works. That's the great thing about it. It provides you intuition for free. It tells you what the answer is pretty much. It is slightly nonrigorous, this is a bit of a pain, so you have to be really careful when you apply it. Otherwise, you might get the wrong answer. Because it involves dot, dot, dots, our favorite three characters, but dot, dot, dots are always a little bit nonrigorous so be careful.

Technically, what you should do is find out what the answer is with recursion-tree method. Then prove that it is actually right with the substitution method. Usually that is not necessary, but you should at least have in your mind that that is required rigorously. And probably the first few recurrences you solve, you should do it that way. When you really understand the recursion-tree method, you can be a little bit more sloppy if you are really sure you have the right answer. Let's do an example.

We saw recursion trees very briefly last time with mergesort as the intuition why it was n log n. And, if you took an example like the one we just did with the recursion-tree method, it is dead simple. Just to make our life harder, let's do a more complicated recursion. Here we imagine we have some algorithm. It starts with a problem size n, it recursively solves a problem of size n/4, it then recursively solves a problem of size n/2, and it does n^2 work on the side without nonrecursive work.

What is that? I mean that is a bit less obvious, I would say. What we are going to do is draw a picture, and we are just going to expand out that recursion in tree form -- -- and then just add everything up. We want the general picture, and the general principle in the recursion-tree method is we just draw this as a picture. We say well, T(n) equals the sum of n^2, T(n/4) and T(n/2). This is a weird way of writing a sum but why not write it that way. This is going to be a tree.

And it is going to be a tree by recursively expanding each of these two leaves. I start by expanding T(n) to this, then I keep expanding, expanding, expanding everything. Let's go one more step. We have this n^2, T(n/4), T(n/2). If we expand one more time, this is going to be n^2 plus two things. The first thing is going to be (n/4)^2, the second thing is going to be (n/2)^2. Plus their recursive branches. We have T(n/16) and T(n/8). Here my arithmetic shows thin. This better be the same, T(n/8), and this should be T(n/4), I believe. You just keep going forever, I mean, until you get down to the base case where T is a constant. So, I am now going to skip some steps and say dot, dot, dot. This is where you have to be careful.

We have n^2, (n/4)^2, (n/2)^2. Now this is easy because I have already done them all. (n/16)^2, (n/8)^2, (n/8)^2 again, (n/4)^2 and et cetera, dot, dot, dot, of various levels of recursion here. At the bottom, we are going to get a bunch of constants. These are the leaves. I would like to know how many leaves there are. One challenge is how many leaves in this tree could there be? This is a bit subtle, unlike mergesort or unlike the previous recurrence we solved, the number of leaves here is a bit funny because we are recursing at different speeds. This tree is going to be much smaller than this tree. It is going to have smaller depth because it has already done down to (n/16). Here it has only gone down to (n/4).

But how many leaves are there in this recursion tree? All I need is an upper bound, some reasonable upper bound. I can tell you it is at most T(n^10), but that is a bit unreasonable. It should be less than n, good. Why is it less than n? Exactly. I start with a problem of size n. And I recurse into a problem that n/4 and a problem that says n/2. When I get down to one I stop. So, n/4 + n/2 = ¾n, which is strictly less than n. So, definitely the total number of leaves has to be at most n. If I start out with n sort of stuff and get rid of a quarter of it and then recurse, it is definitely going to be less than n stuff at the bottom. So, strictly less than n leaves.

At this point, I have done nothing interesting. And then the second cool idea in recursion trees is you don't just expand this tree and see what it looks like and then say, well, God, how the hell am I going to sum that? You sum it level by level. That is the only other idea. It usually works really, really well. Here it is a bit complicated and I have to think a bit to figure out n^2 is n^2. That is the first level. Easy. The second level, I have to think a lot harder.

There are three kinds of mathematicians, those who can add and those who cannot, and I am the latter kind so I need your help. Can you add these things together? It's n^2 over

something. Please? (5/16)n^2. Now I really need your help. I think that one I could have done, but this one is a little bit harder. I will go look at my notes while you compute that. Any answers? 73/256. Anyone else confirm that? It seems a bit high to me. 73 does not sound right to me. 64? Closer. It is actually important that we get this right. The 256 is correct. I can tell.

Everyone should know that 16^2 = 256. We are computer scientists. 25, good. We have two people saying 25, therefore it is correct by democracy. [LAUGHTER] 25 is also what my notes say, and I computed it at home. (25/256)n^2 is the right answer. Now, did anyone notice something magical about this progression? It squares each time, good. And, if we were going to add these up, you might call it? A geometric series, very good. So, it turns out this is geometric. And we know how to sum geometric series, at least you should.

We started n^2. We know that at the bottom, well, this is not quite a level, we get something like n, but we are decreasing geometrically. So, the total, I mean the solution to the recurrence is the sum of all the numbers in this tree. If we added it up level by level and then add up all the levels that is going to give us the answer. This is the total computed level by level. It is just a cute way to compute it. It usually gives you nice answers like geometric answers.

We have n^2(1 + 5/16 + 25/256 + ...). And, if we believe in fate and we see this three number recurrence, we know that we have the right answer. In general, it is going to be (5/16)k, at least we hope, and so on. And it keeps going. It doesn't go on infinitely, but let's just assume it goes on infinitely. That will be an upper bound that goes on forever. This is all times n^2. Now, if you are going to know one thing about geometric series, you should know that 1 + ½ + ¼, if you sum all the powers of 2 you get 2. We are computer scientists. We have got to know at least the binary case. This is like writing 0.1111111 in binary, actually, 1.11111. And 11111 forever is the same as 1, so this is 2.

This is even smaller. We have 5/16, that is less than a half and then we are squaring each time, so this is even less than 2. If you want, there is a nifty formula for solving the general geometric series, but all we need is that it is a constant. This is O(n^2). It is also O(n^2). It is pretty obvious that it is O(n^2) because the top thing is n^2. So, there is our lower bound of n^2. And we have it within a factor of 2, which is pretty good. You actually get a better factor here. So, that is recursion-tree method. It is a little shaky here because we have these dot, dot, dots, and we just believe that it is geometric. It turns out most of the time it is geometric. No problem here. I would definitely check it with the substitution method because this is not obvious to me that it is going to be geometric. In the cases we will look at in a moment, it will be much clearer, so clear that we can state a theorem that everything is working fine.

And still time, good. So, that was recursion-trees. There is one more method we are going to talk about, and you could essentially think of it as an application of the recursion-tree method but it is made more precise. And it is an actual theorem, whereas recursion trees, if the dot, dot, dots aren't obvious, you better check them. The sad part about the master method is it is pretty restrictive. It only applies to a particular family of recurrences.

It should be T(n) = aT(n/b) + f(n). Am I going to call it f? Yes, I will call it f. In particular, it will not cover the recurrence I just solved because I was recursing on two different problems of different sizes. Here, every problem you recurse on should be of the same size. There are a subproblems. A way to think of this is a recursive algorithm. You have a

subproblems. Each of them is of size n/b, so the total costs will be this. Then you are doing f(n) nonrecursive work.

A few constraints. a should be at least 1, should have at least 1 recursion. b should be strictly greater than 1. You better make the problem smaller or else it is going to be infinity. And f should have some nice property. f(n) should be asymptotically positive. How many people know what asymptotically positive means? No one. OK, you haven't read the textbook. That's OK. I haven't read it either, although don't tell Charles. And he'd notice. And what might you think asymptotically positive means? That we can do a little bit better. Sorry? Yes, it means for large enough n, f(n) is positive.

This means f(n) is greater than zero for n, at least some n_o, so for some constant n_o. Eventually it should be positive. I mean, we don't care about whether it's negative 1 for n=1, not a big deal. It won't affect the answer because we only care about the asympotics within. The master method, you gave it a recurrence of this form, it tells you the answer. That is the great thing about the master method. The annoying thing about the master method is that it has three cases. It is a big long. It takes a little bit longer to memorize than all the others because the others are just ideas. Here we need to actually remember a few things. Let me state the theorem. Well, not quite yet. There is one very simple idea, which is we are going to compare this nonrecursive work f(n) with a very particular function n^(log_b(a)).

Why n^(log_b(a))? You will see later. It turns out it is the number of leaves in the recursion tree, but that is foreshadowing. So, it is either less, equal or bigger. And here we care about asymptotics. And we have to be a little bit more precious about less, equal or bigger. You might think well, it means little o, big Theta, or little omega. It would be nice if the theorem held for all of those cases, but it leaves some gaps. Let's start with Case 1. Case 1 is when f is smaller. And not just that it is little o, but it is actually quite a bit smaller. It has got to be polynomially smaller than n^(log_b(a)).

For some positive epsilon, the running time should be this n to this constant log base b of a minus that epsilon, so it is really polynomially smaller than n^(log_b(a)). We cannot handle the little o case, that's a little bit too strong. This is saying it is really quite a bit smaller. But the answer then is really simple, T(n) = Theta(n^(log_b(a))). Great. That is Case 1. Case 2 is when f(n) is pretty much equal to n^(log_b(a)). And by pretty much equal I mean up to poly log factors. This is log base 2 of n to the power k. You should know this notation. For example, k could be zero.

And then they are equal up to constant factors, for some k greater than or equal to zero. Less than will not work, so it is really important that k is non-negative. It should probably be an integer. It doesn't actually matter whether there is an integer, but there it is. It could n^(log_b(a)) times log n or just times nothing, whatever. Again, the solution is easy here, T(n) = Theta(n^(log_b(a))* lg^(k+1)(n)).

Presumably it has to be at least times log k. It turns out it is log to the k plus 1 of n. That is Case 2. We have one more case which is slightly more complicated. We need to assume slightly more for Case 3. But Case 3 is roughly when f(n) grows bigger than n^(log_b(a)). So, it should be capital Omega, here is one place where we get to use omega, (n^(log_b(a)) + epsilon) for some positive epsilon. It should grow not just bigger but polynomially bigger. Here it was growing just a log factor bigger, poly log, and here it is a polynomial factor.

In this case, we need another assumption about f because we worry a little bit about how quickly f grows. We want to make sure that as you go down the recursion f gets smaller. It would be kind of nice if f gets smaller as you go down, otherwise you are, again, trying to sum to infinity or whatever. I see why this is for some epsilon prime greater than zero. What I would like is that if I just sort of take the recurrence, this T(n) and just throw in fs instead, f(n) should be somehow related to af(n/b). What I would like is that f(n), which is at the top of the recursion tree, should be bigger than the thing at the next level down. The sum of all the values at the next level down should be bigger by some constant factor.

Here I have the next level down is at most some 1 - e, something strictly less than 1, some constant strictly less than 1 times the thing at the top level. I need that to make sure things are getting smaller as I go down. Then T(n) = Theta[f(n)]. And that is the theorem. This is the master theorem or whatever you want to call it. It is not named after some guy name Master. It is just the master of all methods because it is very easy to apply. Let's apply it a few times. It is a bit much to take in all at once. And then I will give you a sketch of the proof to see that it is really not that surprising this is true if you look at the recursion-tree. But first let's just try using it.

For example, we could take T(n) = 4T(n/2) + n. This is a, this is b, this is f(n). The first thing we should compute is n^(log_b(a)). This I think even I can do. Log base 2 of 4. Yeah, log base 2 I can do. This is n^2. OK, so is f(n) smaller or bigger than n^2? Well, f(n) = n. n^2 is clearly bigger by a polynomial factor. So, we are in Case 1. What is the answer? n^2, yeah. It is T(n^(log_b(a))), which here it is just n^2. Let's do some slight variation. I am going to keep a and b the same and just change f. Let's say T(n) = 4T(n/2) + n^2. This is like drill spelling. n^2 is asymptotically the same as n^2 even up to constants.

What is the answer? This is Case 2. It is slightly harder. What is k in this example? Zero. The answer is? Survey says? n^2 log n. Good. And a couple more. T(n) = 4T(n/2) + n^3. What is the answer? n^3. This is Case 3. I know this is pretty boring. At this point we are just applying this stupid theorem. How about n^2/lg n? What is the answer? Good. In this case no one should answer. It is a big tricky. I forget exactly the answer.

I think it is like n^2 log log n over log n, no? Oh, no. n^2 log log n, that's right. Yeah. But you shouldn't know that, and this doesn't follow from the master method. This is something you would have to solve, probably with the recursion-tree would be a good way to do this one, and you need to know some properties of logs to know how that goes. But here the master method does not apply.

And so you have to use a different method. OK. The last thing I want to do is tell you why the master method is true, and that makes it much more intuitive, especially using recursion-trees, why everything works. This is a sketch of a proof, not the full thing. You should read the proof in the textbook. It is not that much harder than what I will show, but it is good for you to know the formal details. I don't have time here to do all of the details. I will just tell you the salient parts. This is the proof sketch or the intuition behind the master method.

What we are going to do is just take the recursion-tree for this recurrence and add up each level and then add up all the levels and see what we get. We start with f(n) at the top after we have expanded one level. Then we get a different problems, each of n/b. And after we expand them it will f(n/b) for each one. They are all the same size. Then we expand all of those and so on, and we get another a subproblems from there. We are going to get like

f((n/b)^2). That is sort of decreasing geometrically the size, and so on and so on and so on, until at the bottom we get constant size problems.

This is a bit special because this is the base case, but we have some other constant at the bottom. We would like to know how many leaves there are, but that is a little bit tricky at the moment. Let's first compute the height of this tree. Let me draw it over here. What is the height of this tree? I start with a problem of size n. I want to get down to a problem of size 1. How long does that take? How many levels?

This is probably too easy for some and not at your fingertips for others. Log base b of n, good. The height of this tree is n^(log_b(a)), because it is just how many times I divide by b until I get down to 1. That is great. Now I should be able to compute the number of leaves because I have branching factor a, I have height h. The number of leaves is a^h, a^log_b(n). Let me expand that a little bit. a^log_b(n), properties of logs, we can take the n downstairs and put the a upstairs, and we get n^(log_b(a)). Our good friend n^(log_b(a)).

So, that is why Our good friend n^(log_b(a)) is so important in the master method. What we are doing is comparing f, which is the top level, to n^(log_b(a)), which up to theta is the bottom level. Now the leaves are all at the same level because we are decreasing at the same rate in every branch. If I add up the cost at the bottom level, it is Theta(n^(log_b(a))). I add up the things at the top level it is f(n), not terribly exciting.

But the next level, this is a little bit more interesting, is af(n/b), which should look familiar if you had the master method already memorized, it is that. So, we know that af(n/b) has decreased by some constant factor, 1-epsilon prime. We have gone down. This is a constant factor smaller than this. And then you sum up the next level. It is going to be like a^2f(n/b^2). I see that I actually wrote this wrong, the parentheses. Sorry about that. It is not (n/b)^2. It is (n/b^2). So, this sequence, in Case 3 at least, is decreasing geometrically. If it is decreasing geometrically up to constant factors, it is dominated by the biggest term, which is f(n). Therefore, in Case 3, we get Theta[f(n)].

Let's look at the other cases, and let me adapt those cases to how much time we have left. Wow, lot's of time. Five minutes. Tons of time. What to do? Let me write that down. Case 3, the costs decrease. Now, this is a place I would argue where the dot, dot, dot is pretty obvious. Here, this is damn simple, it is a^kf(n/b^k). And, in Case 3, we assume that the costs decrease geometrically as we go down the tree. That was sort of backwards to start with Case 3. Let's do Case 1, which is sort of the other intuitively easy case.

In Case 1, we know that f(n) is polynomially smaller than this thing. And we are sort of changing by this very simple procedure in the middle. I am going to wave my hands if this is where you need a more formal argument. I claim that this will increase geometrically. It has to increase geometrically because this f(n) is polynomially smaller than this one, you are going to get various polynomials in the middle which interpret geometrically from the small one to the big one.

Therefore, the big one dominates because it is, again, geometric series. As I said, this is intuition, not a formal argument. This one was pretty formal because we assumed it, but here you need a bit more argument. They may not increase geometrically but they could increase faster, and that is also fine. So, in Case 3, you are dominated, I mean you are always dominated by the biggest term in a geometric series.

Here it happens to be f(n) and here you are dominated by n^(log_b(a)) with a bottom term, oh, Theta. Case 2, here it is pretty easy but you need to know some properties of logs. In Case 2, we assume that all of these are basically the same. I mean, we assume that the top is equal to the bottom. And this is changing in this very procedural way. Therefore, all of the ones in the middle have to be pretty much the same. Not quite because here we don't have the log factor. Here we have a log to the k. We have n^(log_b(a)) times log to the kn. Here we don't have the log to the k. So, the logs do disappear here. It turns out the way they disappear is pretty slowly.

If you look at the top half of these terms, they will all have log to the k. The bottom half they will start to disappear. I am giving you some oracle information. If you take logs and you don't change the argument by too much, the logs remain. Maybe halfway is too far. The claim is that each level is roughly the same, especially the upper most levels are all asymptotically equal. Roughly the same. And, therefore, the cost is one level, here like f(n) times the number of levels, h. And h is log base b of n. B is a constant so we don't care. This is Theta(lg n). And, therefore, we get T(n) = (n^(log_b(a)) lg^(k+1)(n)) times another log n.

So, we get [f(n)lg n]. That is the very quick sketch. Sorry, I am being pretty fuzzy on Cases 1 and 2. Read the proof because you will have to, at some point, manipulate logs in that way. And that is all. Any questions? Or, you are all eager to go. OK. Thanks. See you Wednesday.