MITOCW | ocw-6.046-lec3

Good. Log base 2 of 7 is going be a Good morning everyone. Today we are going to do some algorithms, back to algorithms, and we are going to use a lot of the, well, some of the simpler mathematics that we developed last class like the master theorem for solving recurrences. We are going to use this a lot. Because we are going to talk about recursive algorithms today. And so we will find their running time using the master theorem. This is just the same as it was last time, I hope, unless I made a mistake. A couple of reminders. You should all go to recitation on Friday. That is required. If you want to, you can go to homework lab on Sunday. That may be a good excuse for you to actually work on your problem set a few hours early. Well, actually, it's due on Wednesday so you have lots of time. And there is no class on Monday. It is the holiday known as Student Holiday, so don't come. Today we are going to cover something called Divide and Conquer. Also known as divide and rule or divide et impera for those of you who know Latin, which is a tried and tested way of conquering a land by dividing it into sections of some kind. It could be different political factions, different whatever. And then somehow making them no longer like each other. Like starting a family feud is always a good method. You should remember this on your quiz. I'm kidding. And if you can separate this big power structure into little power structures such that you dominate each little power structure then you can conquer all of them individually, as long as you make sure they don't get back together. That is divide and conquer as practiced, say, by the British. But today we are going to do divide and conquer as practiced in Cormen, Leiserson, Rivest and Stein or every other algorithm textbook. This is a very basic and very powerful algorithm design technique. So, this is our first real algorithm design experience. We are still sort of mostly in the analysis mode, but we are going to do some actual design. We're going to cover maybe only three or four major design techniques. This is one of them, so it is really important. And it will lead to all sorts of recurrences, so we will get to use everything from last class and see why it is useful. As you might expect, the first step in divide-and-conquer is divide and the second step is conquer. But you may not have guessed that there are three steps. And I am leaving some blank space here, so you should, too. Divide-and-conquer is an algorithmic technique. You are given some big problem you want to solve, you don't really know how to solve it in an efficient way, so you are going to split it up into subproblems. That is the divide. You are going to divide that problem, or more precisely the instance of that problem, the particular instance of that problem you have into subproblems. And those subproblems should be smaller in some sense. And smaller means normally that the value of N is smaller than it was in the original problem. So, you sort of made some progress. Now you have one, or more likely you have several subproblems you need to solve. Each of them is smaller. So, you recursively solve each subproblem. That is the conquer step. You conquer each subproblem recursively. And then somehow you combine those solutions into a solution for the whole problem. So, this is the general divide-and-conquer paradigm. And lots of algorithms fit it. You have already seen one algorithm that fits this paradigm, if you can remember. Merge sort, good. Wow, you are all awake. I'm impressed. So, we saw merge sort. And, if I am clever, I could fit it in this space. Sure. Let's be clever. A quick review on merge sort. Phrased in this 1, 2, 3 kind of method. The first step was to divide your array into two halves. This really doesn't mean anything because you just sort of think, oh, I will pretend my array is divided into two halves. There is no work here. This is zero time. You just look at your array. Here is your array. I guess maybe you compute n/2 and take the floor. That takes constant time. And you say OK, I am pretending my array is now divided into the left part and the right part. And then the interesting part is that you recursively solve each one. That's the conquer. We recursively sort each subarray. And then the third step is to combine those solutions. And so here we really see what this means. We now have a sorted version of this array by induction. We have a sorted version of this array by induction. We now want the sorted version of the whole array. And we saw that was the merge problem, merging two sorted arrays. And that we saw how to do in linear time, order n time. I am not going to repeat that, but the point is it falls into that framework. I want to write the running time and merge sort as a recurrence. You have already seen the recurrence, you have already been told the solution, but now we actually know how to solve it. And, furthermore, every algorithm that follows the divide-and-conquer paradigm will have a recurrence of pretty much the same form, very much like our good friend the master method. Let's do it for merge sort where we sort of already know the answer and get a bit of practice. This is the merge sort recurrence. You should know and love this recurrence because it comes up all over the place. It comes from this general approach by just seeing what are the sizes of the subproblems you are solving and how many there are and how much extra work you are doing. You have here the size of the subproblems. It happens here that both subproblems have the same size roughly. There is this sloppiness that we have, which really should be T of floor of n over 2 plus T of ceiling of n over 2. And when you go to recitation on Friday you will see why that is OK, the floors and ceilings don't matter. There is a theorem you can prove that that's happy. You can assume that N is a power of 2, but we are just going to assume that for now. We just have two problems with size n over 2. This 2 is the number of subproblems. And this order n is all the extra work we are doing. Now, what is the extra work potentially? Well, the conquering is always just recursion. There is sort of no work there except this lead part. The dividing in this case is trivial, but in general it might involve some work. And the combining here involves linear work. So, this is the divide-and-conquer running times. So, this is the nonrecursive work. And that is generally how you convert a divide-and-conquer algorithm into a recurrence. It's really easy and you usually get to apply the master method. Here we are in Case 2. Very good. This is Case 2. And k is zero here. And so in the recursion tree, all of the costs are roughly the same. They are all n to the log base b of a. Here n to the log base 2 of 2 is just n. So these are equal. We get an extra log factor because of the number of levels in the recursion tree. Remember the intuition behind the master method. This is n log n, and that is good. Merge sort is a fast sorting algorithm n log n. Insertion sort was n squared. In some sense, n log n is the best you can do. We will cover that in two lectures from now, but just foreshadowing. Today we are going to do different divide-and-conquer algorithms. Sorting is one problem. There are all sorts of problems we might want to solve. It turns out a lot of them you can apply divide-and-conquer to. Not every problem. Like how to wake up in

solve. It turns out a lot of them you can apply divide and conquer to. Not every problem. Like how to wake up in the morning, it's not so easy to solve a divide-and-conquer, although maybe that's a good problem set problem. The next divide-and-conquer algorithm we are going to look at is even simpler than sorting, even simpler than merge sort, but it drives home the point of when you have only one subproblem. How many people have seen binary search before? Anyone hasn't? One, OK. I will go very quickly then. You have some element X. You want to find X in a sorted array. How many people had not seen it before they saw it in recitation? No one, OK. Good. You have seen it in another class, probably 6.001 or something. Very good. You took the prerequisites. OK. I just want to phrase it as a divide-and-conquer because you don't normally see it that way. The divide step is you compare X with the middle element in your array. Then the conquer step. Here is your array. Here is the middle element. You compare X with this thing if let's say X is smaller than the middle element in your array. You know that X is in the left half because it is sorted, a nice loop invariant there, whatever, but we are just going to think of that as recursively I am going to solve the problem of finding X in this subarray. We recurse in one subarray, unlike merge sort where we had two recursions. And then the combined step we don't do anything. I mean if you find X in here then you've found X in the whole array. There is nothing to bring it back up really. So, this is just phrasing binary search in the divide-and-conquer paradigm. It is kind of a trivial example, but there are lots of circumstances where you only need to recurse in one side. And it is important to see how much of a difference making one recursion versus making two recursions can be. This is the recurrence for binary search. We start with a problem size n. We reduce it to 1. There is an implicit 1 factor here. One subproblem of size n over 2 roughly. Again, floors and ceilings don't matter. Plus a constant which is to compare X with the middle element, so it is actually like one comparison. This has a solution, log n. And you all know the running time of binary search, but here it is at solving the recurrence. I mean, there are a couple of differences here. We don't have the additive order n term. If we did, it would be linear, the running the time. Still better than n log n. So, we are getting rid of the 2, bringing it down to a 1, taking the n and bringing it down to a 1. That is making the running time a lot faster, the whole factor of n faster. No big surprise there. Let's do some more interesting algorithms. The powering a number problem is I give you a number X. I give that as like a real number or floating point number or whatever. And I give you an integer n, at least zero. I want to compute X to the power n. So, it's a very simple problem. It is, in some sense, even easier than all of these. But here it is. And divide-and-conquer is sort of the right thing to do. So, the naÃ”ve algorithm is very simple. How do you compute X to the power n? Well, the definition of X to the power n is I take X and I multiply by X n times. So, I take X times X times X times X where there are n copies of X totally. And that's X to the n. No big surprise. That is n multiplications, or n minus 1 multiplications, theta n time. But that's not the best you can do for this problem. Any suggestions on what we might do using divide-and-conquer? Has anyone seen this algorithm before? A few, OK. For the rest? Testing on the spot creativity, which is very difficult, but I always like a challenge. I mean random ideas. What could we possibly do to solve this problem in less than linear time? How is this sort of a divide-and-conquer problem? We have to inputs, X and n. Yeah? We could try to divide on X. It seems a bit hard. It is just some number. Or, we could try to divide on n. Any guesses? Look at X to the n over 2, very good. That is exactly the idea of the divide-and-conquer algorithm. We would like to look at X to the n over 2. This is going to be a little bit tricky. Now we are going to have to pay attention to floors and ceilings. What I would like to say is while X to the n is X to the n over 2 times X to the n over 2. And this is true if n is even. If it is odd then I need to be a little bit more careful. But let's just think about the intuition why this is a good divide-and-conquer algorithm. We have a problem of size n, let's say. We convert it into, it looks like two subproblems of size n over 2, but in fact they are the same subproblems. So, I only have to solve one of them. If I compute X to the n over 2. Yeah, I know X to the n over 2. So, there is one conversive call, problem of size n over 2, then I square that number. And that is one computation. So, exactly the same recurrence as binary search, log n time much better than n. Cool. I also have to solve the odd case. So, n is odd. I will look at n minus 1 over N minus 1 better be even. And then I am missing another factor of X. If n is odd, I am going to have to do one recursive call and two multiplications. The same recurrence. One recursive problem of size n over 2, plus constant time. The dividing work here is dividing by 2 and the combination work is doing one or possibly two multiplications. And this is lg n. And if all you are allowed to do is multiply numbers, lg n is the best you can do. Good. Simple but powerful. Whenever you want to compute a power of a number, now you know what to do. Does anyone not know the definition of Fibonacci numbers and is willing to admit it? OK, so this is a good old friend. I will write down the definition as just a reminder. And, in particular, the base cases. Fibonacci numbers, I will claim, are very important because it appears throughout nature. You look at certain fruits, you see the Fibonacci sequence. If you count the number of little bumps around each ring. If you look at the sand on the beach and how the waves hit it, it's the Fibonacci sequence I am told. If you look all over the place, Fibonacci sequence is there. How does nature compute the Fibonacci sequence? Well, that is a different class. But how are we going to compute the Fibonacci sequence as fast as possible? You have probably seen two algorithms. The most naÃ”ve algorithm is the recursive algorithm. Where you say OK, f of n. I say well, if n is zero, return zero, if n is 1, return one. Otherwise, recursively compute f of n minus 1 and f of n minus 2, add them together. How much time does this algorithm take, for those who have seen it before? This is not obvious to guess. It doesn't have to be exact. OK. And how many people have seen this algorithm before and analyzed it? Half, OK. So what is the running time? Really, really long, very good. Anymore precise answers? What's that? Exponential, yes. That is also correct and more precise. I will be even more precise. Maybe you haven't seen this analysis before. It's phi to the n where phi is the Golden Ratio. Again, Golden Ratio appears throughout the world in mathematics. This is probably the only time in this class, I'm afraid, but there we go. It made its cameo so we are happy. This is called exponential time. This is bigger than one, that's all you need to know. This is exponential time. Exponential time means basically some constant to the power n. Exponential time is a very long time. It's bad. Polynomial time is good. [LAUGHTER] This is what we want, polynomial time algorithms. This class is basically entirely about polynomial time algorithms. Question? Oh, say what the algorithm does again. Define function Fibonacci of n? I check for the base cases. Otherwise, I recursively call Fibonacci of n minus 1, I recursively call Fibonacci of n minus 2 and add those two

Otherwise, recursively call Fibonacci of n minus 1, recursively call Fibonacci of n minus 2 and add those two numbers together. So, you get this branching tree. You are solving two subproblems of almost the same size, just additively smaller by one or two. I mean you are almost not reducing the problem size at all, so that's intuitively why it is exponential. You can draw a recursion tree and you will see how big it gets and how quickly. I mean by n over two levels, you've only reduced on one branch the problem from n to n over 2. The other one, maybe you've gotten from n down to one, but none of the branches have stopped after n of two levels. You have at least 2 to the power n over 2 which is like square root of 2 to the power n, which is getting close to phi to the n. So, this is definitely exponential. And exponential is bad. We want polynomial. N squared, n cubed, log n would be nice. Anything that is bounded above by a polynomial is good. This is an old idea. It goes back to one of the main people who said polynomial is good, Jack Edmonds who is famous in the combinatorial optimization world. He is my academic grandfather on one side. He is a very interesting guy. OK, so that's a really bad algorithm. You have probably seen a somewhat better algorithm, which you might think of as the bottom-up implantation of that recursive algorithm. Or, another way to think of it is if you build out the recursion tree for Fibonacci of n, you will see that there are lots of common subtrees that you are just wasting time on. When you solve Fibonacci of n minus 1, it again solves Fibonacci of n minus 2. Why solve it twice? You only need to solve it once. So, it is really easy to do that if you do it bottom-up. But you could also do it recursively with some cache of things you have already computed. So, no big surprise. You compute the Fibonacci numbers in order. And each time, when you compute Fibonacci of n, let's say, you have already computed the previous two, you add them together, it takes constant time. So, the running time here is linear, linear in n, and as our input. Great. Is that the best we can do? No. Any ideas on how we could compute Fibonacci of n faster than linear time? Now we should diverge from what you have already seen, most of you. Any ideas using techniques you have already seen? Yes? Yes. We can use the mathematical trick of phi and psi to the nth powers. In fact, you can just use phi, phi, pi, pho, phum, whatever you want to call this Greek letter. Good. Here is the mathematical trick. And, indeed, this is cheating, as you have said. This is no good, but so it is. We will call it naÃ”ve recursive squaring and say well, we know recursive squaring. Recursive squaring takes log n time. Let's use recursive squaring. And if you happen to know lots of properties of the Fibonacci numbers, you don't have to, but here is one of them. If you take phi to the n divided by root 5 and you round it to the nearest integer that is the nth Fibonacci number. This is pretty cool. Fibonacci of n is basically phi to the n. We could apply recursive squaring to compute phi to the n in log n time, divide by root 5, assume that our computer has an operation that rounds a number to its nearest integer and poof, we are done. That doesn't work for many different reasons. On a real machine, probably you would represent phi and root 5 as floating point numbers which have some fixed amount of precise bits. And if you do this computation, you will lose some of the important bits. And when you round to the nearest integer you won't get the right answer. So, floating point round off will kill you on a floating point machine. On a theoretical machine where we magically have numbers that can do crazy things like this, I mean it really takes more than constant time per multiplication. So, we are sort of in a different model. You cannot multiply phi times phi in constant time. I mean that's sort of outside the boundaries of this course, but that's the way it is. In fact, in a normal machine, some problems you can only solve in exponential time. In a machine where you can multiply real numbers and round them to the nearest integers, you can solve them in polynomial time. So, it really breaks the model. You can do crazy things if you were allowed to do this. This is not allowed. And I am foreshadowing like three classes ahead, or three courses ahead, so I shouldn't talk more about it. But it turns out we can use recursive squaring in a different way if we use a different property of Fibonacci numbers. And then we will just stick with integers and everything will be happy. Don't forget to go to recitation and if you want to homework lab. Don't come here on Monday. That is required. This is sort of the right recursive squaring algorithm. And this is a bit hard to guess if you haven't already seen it, so I will just give it to you. I will call this a theorem. It's the first time I get to use the word theorem in this class. It turns out the nth Fibonacci number is the nth power of this matrix. Cool. If you look at it a little bit you say oh, yeah, of course. And we will prove this theorem in a second. But once we have this theorem, we can compute f of n by computing the nth power of this matrix. It's a two-by-two matrix. You multiply two two-by-two matrixes together, you get a two-by-two matrix. So that is constant size, four numbers. I can handle four numbers. We don't have crazy precision problems on the floating point side. There are only four numbers to deal with. Matrixes aren't getting bigger. So, the running time of this divide-and-conquer algorithm will still be log n because it takes a constant time per two-by-two matrix multiplication. Yes? Oh, yes. Thank you. I have a type error. Sorry about that. F of n is indeed the upper left corner, I hope. I better check I don't have it off by one. I do. It's F_n upper right corner, indeed. That's what you said. F of n. I need more space. Sorry. I really ought to have a two-by-two matrix on the left-hand side there. Thank you. So, I compute this nth power of a matrix in log n time, I take the upper right corner or the lower left corner, your choice, that is the nth Fibonacci number. This implies an order log n time algorithm with the same recurrence as the last two, binary search and really the recursive squaring algorithm. It is log n plus a constant, so log n. Let's prove that theorem. Any suggestions on what techniques we might use for proving this theorem, or what technique, singular? Induction, very good. I think any time I ask that question the answer is induction. Hint for the future in this class. A friend of mine, when he took an analysis class, whenever the professor asked, and what is the answer to this question, the answer was always zero. If you have taken analysis class that is funny. [LAUGHTER] Maybe I will try to ask some questions whose answers are zero just for our own amusement. We are going to induct on n. It's pretty much the obvious thing to do. But we have to check some cases. So, the base case is we have this to the first power. And that is itself [(1, 1), (1, 0)]. And I should have said n is at least 1. And you can check. This is supposed to be F_2, F_1, F_1, F_0. And you can check it is, F_0 is 0, F_1 is 1 and F_2 is Base case is correct, step case is about as exciting, but you've got to prove that your algorithm works. Suppose this is what we want to compute. I am just going to sort of, well, there are many ways I can do this. I will just do it the fast way because it's really not that exciting. Which direction? Let's do this direction. I want to use induction on n. If I want to use induction on n, presumably I should use what I already know is true. If I decrease n by 1, I have this property that this thing is going to be [(1, 1), (1, 0)] to the power n minus 1. This I already know, by the induction

property that this thing is going to be [(1, 1), (1, 0)] to the power n minus 1. This is already known, by the induction hypothesis, [(1, 1), (1, 0)] to the n minus 1. So, presumably I should use it in some way. This equality is not yet true, you may have noticed. So, I need to add something on. What could I possibly add on to be correct? Well, another factor of [(1, 1), (1, 0)]. The way I am developing this proof is the only way it could possibly be, in some sense. If you know its induction, this is all that you could do. And then you check. Indeed, this equality holds conveniently. For example, F_(n plus 1) is the product of these two things. It is this row times this column. So, it is F_n times 1 plus F_(n minus 1) times 1, which is indeed the definition of F_(n plus 1). And you could check four of the entries. This is true. Great. If that is true then I would just put these together. That is [(1, 1), (1, 0)] to the n minus 1 times [(1, 1), (1, 0)], which is [(1, 1), (1, 0)] to the n, end of proof. A very simple proof, but you have to do that in order to know if this algorithm really works. Good. Question? Oh, yes. Thank you. This, in the lower right, we should have F_(n minus 1). This is why you should really check your proofs. We would have discovered that when I checked that this was that row times that column, but that is why you are here, to fix my bugs. That's the great thing about being up here instead of in a quiz. But that is a minor mistake. You wouldn't lose much for that. All right. More divide-and-conquer algorithms. Still, we have done relatively simple ones so far. In fact, the fanciest has been merge sort, which we already saw. So, that is not too exciting. The rest have all be log n time. Let's break out of the log n world. Well, you all have the master method memorized, right, so I can erase that. Good. This will be a good test. Next problem is matrix multiplication, following right up on this two-by-two matrix multiplication. Let's see how we can compute n-by-n matrix multiplications. Just for a recap, you should know how to multiply matrixes, but here is the definition so we can turn it into an algorithm. You have two matrixes, A and B, which are capital levels. The ijth entry. Ith row, jth column is called little a_ij or little b_ij. And your goal is to compute the products of those matrixes. I should probably say that i and j range from 1 to n. So, they are square matrixes. The output is to compute C which has entry c_ij which is the product of A and B. And, for a recap, the ijth entry of the product is the inner product of the ith row of A with the jth column of B. But you can write that out as a sum like so. We want to compute this thing for every i and j. What is the obvious algorithm for doing this? Well, for every i and j you compute the sum. You compute all the products. You compute the sum. So, it's like n operations here roughly. I mean like 2n minus 1, whatever. It is order n operations. There are n^2 entries of C that I need to compute, so that's n^3 time. I will write this out just for the programmers at heart. Here is the pseudocode. It's rare that I will write pseudocode. And this is a simple enough algorithm that I can write it in gory detail. But it gives you some basis for this analysis if you like to program. It is a triply nested for loop. And I made a coding error. Hopefully you haven't written that far yet. I need c_ij to be initialized to zero. And then I add to c_ij the appropriate product, a_ik, b_kj. That is the algorithm. And the point is you have a nesting of n for loops from 1 to n. That takes n^3 time because this is constant and that is constant. So, very simple, n^3. Let's do better. And, of course, we are going to use divide-and-conquer. Now, how are we going to divide a matrix? There are a lot of numbers in a matrix, n^2 of them in each one. There are all sorts of ways you could divide. So far all of the divide-and-conquers we have done have been problems of size n into some number of problems of size n over 2. I am going to say I start with some matrixes of size n-by-n. I want to convert it down to something like n/2-by-n/2. Any suggestions how I might do that? Yeah? Block form the matrix, indeed. That is the right answer. So, this is the first divide-and-conquer algorithm. This will not work, but it has the first idea that we need. We have a n-by-n matrix. We can view it, this equality is more, you can think of it as, it's really the thing, a two-by-two block matrix where each entry in this two-by-two block matrix is a block of n/2-by-n/2 submatrixes. I will think of C as being divided into three parts, r, s, t and u. Even though I write them as lower case letters they are really matrixes. Each is n/2-by-n/2. And A, I will split into a, b, c, d, times B, I will split into e, f, g, h. Why not? This is certainly true. And if you've seen some linear algebra, this is basically what you can do with matrixes. Now I can pretend these are two-by-two and sort of forget the fact that these little letters are matrixes and say well, r is the inner product of this row with this column. It is ae times bg. Let me not cheat or else it will be too easy. r=ae+bg, s=af+bh, t=ce+dh and u=cf+dg. It's nothing like making it too hard on yourself. OK, got them right. Good. I mean this is just a fact about how you would expand out this product. And so now we have a recursive algorithm. In fact, we have a divide-and-conquer algorithm. We start with our n-by-n matrix. Well, we have two of them actually. We divide it into eight little pieces, a, b, c, d, e, f, g, h. Then we compute these things and that gives us C, just by sticking them together. Now, how do we compute these things? Well, these innocent-looking little products between these two little numbers are actually recursive matrix multiplications. Because each of these little letters is an n/2-by-n/2 matrix so I have to recursively compute the product. There are like eight recursive multiplications of n/2-by-n/2 matrixes. That is what bites us. And then there are like four additions, plus minor work of gluing things together. How long does it take to add two matrixes together? n^2. This is cheap. It just takes n^2. Remember, we are trying to beat n^3 for our matrix multiplication. Addition is a really easy problem. You just have to add every number. There is no way you can do better than n^2. So, that is not recursive. That is the nice thing. But the bad thing is we have eight of these recursions. We have T(n)=8T(n/2)+Theta(n^2). And I have erased the master method, but you should all have it memorized. What is the solution to this recurrence? Theta(n^3). That is annoying. All right. A is 8, b is 2, log base 2 of 8 is 3. Every computer scientist should know that. n^log_b(a)=n^3. That is polynomially larger than n^2, so we are in Case 1. Thank you. Let's get them upside down. This is n^3, no better than our previous algorithm. That kind of sucks. And now comes the divine inspiration. Let's go over here. There are some algorithms like this Fibonacci algorithm where if you sat down for a little while, it's no big deal, you would figure it out. I mean it is kind of clever to look at that matrix and then everything works happily. It is not obvious but it is not that amazingly clever. This is an algorithm that is amazingly clever. You may have seen it before which steals the thunder a little bit, but it is still really, really cool so you should be happy to see it again. And how Strassen came up with this algorithm, he must have been very clever. The idea is we've got to get rid of these multiplications. I could do a hundred additions. That only costs Theta(n^2). I have to reduce this 8 to something smaller. It turns out, if you try to split the matrices into three-by-three or something, that doesn't help you. You get the same problem because we're using fundamentally the same algorithm, just in a different order. We have got to somehow reduce the

We're doing fundamentally the same algorithm, just in a different order. We have got to somehow reduce the number of multiplications. We are going to reduce it to 7. The claim is that if we have two two-by-two matrices we can take their product using seven multiplications. If that were true, we would reduce the 8 to a 7 and presumably make things run faster. We will see how fast in a moment. You can compute it in your head. If you are bored and like computing logs that are non-integral logs then go ahead. All right. Here we are. This algorithm is unfortunately rather long, but it is only seven multiplications. Each of these P's is a product of two things which only involves addition or subtraction, the same thing. Those are seven multiplications. And I can compute those in 7T(n/2). Oh, indeed it is. Six was wrong. Six and seven are the same, very good. You know, you think that copying something would not be such a challenging task. But when you become an absent-minded professor like me then you will know how easy it is. OK. We have them all correct, hopefully. We continue. That wasn't enough. Of course we had seven things. Clearly we have to reduce this down to four things, the elements of C. Here they are, the elements of C, r, s, t, u. It turns out r=P_5+P_4-P_2+P_6. Of course. Didn't you all see that? [LAUGHTER] I mean this one is really easy, s=P_1+P2. t=P_3+P_4. I mean that is clearly how they were chosen. And then u is another tricky one, u=P_5+P_1-P_3-P_7. OK. Now, which one of these would you like me to check? Don't be so nice. How about s? I can show you s is right. Any preferences? u. Oh, no, sign errors. OK. Here we go. The claim that this really works is you have to check all four of them. And I did in my notes. u=P_5. P_5=(ae + ah + de + dh). That is P_5. Check me. If I screw up, I am really hosed. (af - ah) = P_1. P_3 has a minus sign in front, so that is (ce + de). And then we have minus P_7, which is a big one, (ae + af - ce - cf). OK. Now I need like the assistant that crosses off things in parallel like the movie, right? ah, de, af, ce, ae, thank you, and hopefully these survive, dh minus minus cf. And, if we are lucky, that is exactly what is written here, except in the opposite order. Magic, right? Where the hell did Strassen get this? You have to be careful. It is OK that the plus is in the wrong order because plus is commutative, but the multiplications better not be in the wrong order because multiplication over matrixes is not commutative. I check cf, OK, dh, they are in the right order. I won't check the other three. That is matrix multiplication in hopefully subcubic time. Let's write down the recurrence. T(n) is now 7. Maybe I should write down the algorithm for kicks. Why not? Assuming I have time. Lots of time. Last lecture I was ten minutes early. I ended ten minutes early. I apologize for that. I know it really upsets you. And I didn't realize exactly when the class was supposed to end. So, today, I get to go ten minutes late. OK. Good. I'm glad you all agree. [LAUGHTER] I am kidding. Don't worry. OK. Algorithm. This is Strassen. First we divide, then we conquer and then we combine. As usual, I don't have it written anywhere here. Fine. Divide A and B. This is sort of trivial. Then we compute the terms -- -- for the products. This means we get ready to compute all the P's. We compute a+b, c+d, g-e, a+d, e+h and so on. All of the terms that appear in here, we compute those. That takes n2 time because it is just a bunch of additions and subtractions. No big deal. A constant number of them. Then we conquer by recursively computing all the P_i's. That is each our product of seven of them. We have P_1, P_2 up to P_7. And, finally, we combine, which is to compute r, s, t and u. And those are just additions and subtractions again, so they take n^2 times. So, here we finally get an algorithm that is nontrivial both in dividing and in combining. Recursion is always recursion, but now we have interesting steps one and three. The recurrence is T(n) is seven recursive subproblems, each are size n/2 plus order n^2, to do all this addition work. Now we need to solve this recurrence. We compute n^log_b(a), which here is nlog_2(7). And we know log base 2 of 8 is little bit less than 3 but still bigger than 2 because log base 2 of 4 is 2. So, it is going to be polynomially larger than n^2 but polynomially smaller than n^3. We are again in Case 1. And this is the cheating way to write n log base 2 of 7, nlg7. lg means log base 2. You should know that. It is all over the textbook and in our problem sets and what not, nlg7. And, in particular, if I have my calculator here. This is a good old-fashion calculator. No, that is wrong. Sorry. It is strictly less than 2.81. It is strictly less than 2.81. That is cool. I mean it is polynomially better than n^3. Still not as good as addition, which is n^2. It is generally believed, although we don't know whether you can multiply as fast as you can divide for matrices. We think you cannot get n^2, but who knows? It could still happen. There are no lower bounds. This is not the best algorithm for matrix multiplication. It is sort of the simplest that beats n^3. The best so far is like n^2.376. Getting closer to 2. You might think these numbers are a bit weird. Maybe the constants out here dominate the improvement you are getting in the exponent. It turns out improving the exponent is a big deal. I mean, as n gets larger exponents really come out to bite you. So, n^3 is pretty impractical for any very large values of n. And we known that Strassen will beat normal matrix multiplication if n is sufficiently large. The claim is that roughly at about 32 or so already you get an improvement, for other reasons, not just because the exponent gets better, but there you go. So, this is pretty good. This is completely impractical, so don't use whatever this algorithm is. I don't have the reference handy, but it is just trying to get a theoretical improvement. There may be others that are in between and more reasonable but that is not it. Wow, lots of time. Any questions? We're not done yet, but any questions before we move on for matrix multiplication? OK. I have one more problem. Divide-and-conquer is a pretty general idea. I mean, you can use it to dominate countries. You can use it to multiply matrices. I mean, who would have thought? Here is a very different kind of problem you can solve with divide-and-conquer. It is not exactly an algorithmic problem, although it is computer science. That is clear. This is very large-scale integration. The chips, they are very large scale integrated. Probably even more these days, but that is the catch phrase. Here is a problem, and it arises in VLSI layout. We won't get into too many details why, but you have some circuit. And here I am going to assume that the circuit is a binary tree. This is just part of a circuit. Assume for now here that it is just a complete binary tree. A complete binary tree looks like this. In all of my teachings, I have drawn this figure for sure the most. It is my favorite figure, the height four complete binary tree. OK, there it is. I have some tree like that as some height. I want to imbed it into some chip layout on a grid. Let's say it has n leaves. I want to imbed it into a grid with minimum area. This is a very cute problem and it really shows you another way in which divide-and-conquer is a useful and powerful tool. So, I have this tree. I like to draw it in this way. I want to somehow draw it on the grid. What that means is the vertices have to be imbedded onto dots on the grid, and I am talking about the square grid. It has to go to vertices of the grid. And these edges have to be routed as sort of orthogonal paths between one dot and another, so that should be an edge and they shouldn't cross and all these good things because wires

one dot and another, so that should be an edge and they shouldn't cross and all these good things because wires do not like to cross. There is the obvious way to solve this problem and there is the right way. And let's talk about both of them. Neither of them is particularly obvious, but divide-and-conquer sort of gives you a hint in the right direction. So, the naÃ”ve imbedding. I seem to like the word naÃ”ve here. I am going to draw this bottom up because it is easier, so leave three grid lines and then start drawing. I don't know how big that is going to be. Here is the bottom of our tree. This is like the little three nodes there. And then I leave a blank column and then a blank column. I don't actually need to leave those blank columns, but it makes a prettier drawing. And then we work our way up. There is the tree, which should be aligned, on a grid. No crossings. Everything is happy. How much area does it take? By area, I mean sort of the area of the bounding box. So, I count this blank space even though I am not using it and I count all this blank space even though I am not using it. I want to look at the height. Let's call this H(n). And to look at the width, which I will call W(n). Now, it is probably pretty obvious that H(n) is like log n, W(n) is like n or whatever. But I want to write it as a recurrence because that will inspire us to do the right thing. H(n). Well, if you think of this as a recursion-tree, in some sense. We start with the big tree. We split it into two halves, two subtrees of size n/2 indeed because we are counting leaves. It is exactly n/2 on each side. Then for height they are in parallel so it is no big deal. The height is just the height of this thing, one of these subproblems plus one. The width, you have to add together the two widths and also add on 1. You don't have to add on 1 here, but it doesn't matter. It is certainly at most 1. H(n) = H(n/2) + Theta(1), there you do have to add 1, and W(n) = 2W(n/2) + O(1). The usual base cases. I mean, these are recurrences we should know and love. This is log n, I sort of have already given away the answers, and this better be linear. This is again Case 1. And to the log base 2 of 2 is n, which is the answer, much bigger than 1. And here n to the log base 2 of 1 is n to the zero, which is 1, which is the same so we get log n. The area is n log n, but if you are making chips you want the area as small as possible so you can fit more good stuff in there. So, we would like to aim for, well, we certainly cannot do a better area than n. You've got to put the leaves down somewhere, but this is already pretty good. It is only a log factor off, but we want to aim for n. How could we get n? Any guesses on what needs to change in this layout? Not how to do it because that is not obvious, but in terms of height and width what should we do? It is pretty hard to get the height smaller than log n, I will tell you, because this is a tree. It cannot really get its width down to less than log n. What could we do to make the product linear? Just random ideas. What are two functions whose product is n? Square root of n and square root of n. That is a good choice. Were there other suggestions? n times constant. Yeah, n times constant would be nice. But I claim you cannot get either of these down to less than a constant. You could aim for n over log n by log n, that is more likely, but I think that is almost impossible. Root n by root n is the right answer, so let's go with that. So, root n by root n. We haven't seen any recurrences whose solution is root n, but surely they are out there. Let's say the goal is to get W(n) = Theta(root n) and to get H(n) = Theta(root n). If we did that we would be happy, because then the area is the product is linear. How? What is a recurrence that is in the usual master method form whose solution is root n? I mean, you could think of it that way. Recurrence is a bit tricky, but let's just think of n^log_b(a). When is log base b of a Î©? Because then n^log_b(a) is root n. And there is some hope that I could get a root n solution to recurrence. This is designed by knowing that it is divide-and-conquer, and therefore it must be something like this. It is easy once you know the approach you are supposed to take and you can try this approach. When is log base b of a Î©? Lots of solutions, shout them out. 4 and 2, that is a good one. I better get this right. Log base 4 of 2 is Î© because the square root of 4 is 2. So, let's aim for this. Why not? When would we get log base 4 of 2? This is b, this is a, so it should be 2T(n/4) plus something. And if I want the n^log_b(a) to dominate, it has got to be polynomially smaller than root n. So, this should be n^1/2-epsilon. But it could be smaller. It could be 1. Zero would be nice, but that is probably too much to hope for. So, something smaller, strictly polynomially smaller than root n. That is our goal. And now comes the magic. If you played with this for a while you would find it, I think, at this point. When you know that you are somehow solve this problem of size n with two subproblems of size n/4 what could you do? Well, if you start thinking of things as squares, this is the natural thing that happens. This is called the H layout. You can imagine why. It would be much easier to draw if I had a grid board, a graph board, whatever, if that exists. This is a recursive layout. I am only going to draw a couple iterations, but hopefully you can imagine the generalization. I take four Hs, a good plan because I want problems of size n/4. This has n/4 leaves. This has n/4 leaves. This is the root, by the way, in the middle. This has n/4 leaves. This has n/4 leaves. So, I have four problems of size n/4. Somehow I have got to get that down to two. Thankfully, if I look at width or if I look at height, there are only two that matter. And these two matter and these two get along for free. They are going in parallel, just like we had with height over here. But I get that both in height and in width. If I measure, well, now they are equal, so I will just call them the length. We have L(n)-by-L(n). And if I compute well, what is L(n)? I have here L(n/4) because there are only a quarter of the leaves in this one or in that one, then I have a constant, Theta(1), no big deal, and then I have L(n/4) again. So, I get the recurrence that I wanted. L(n) = 2L(n/4) + Theta(1). And that has solution square root of n, as we claimed before. Again, we are in Case 1 of the master method. Cool, ha? This is a much more compact layout. Charles, did you invent this layout? No. But I know it appears on your PhD thesis and you extended it in various directions. So, this is sort of a classic cool layout of trees into grids and another application of divide-and-conquer. I mean this is not particularly useful for algorithms directly. It is useful for VLSI layout directly. But it gives you more flavor of how you should think. If you know what running time you are aiming for, like in problem sets in quizzes often we say here is the running time you have got to get, think about the recurrence that will get you there. And that could inspire you. And that is it. Recitation Friday. Homework lab Sunday. No class Monday. See you Wednesday.