MITOCW | 3. Insertion Sort, Merge Sort

PROFESSOR: So today's lecture is on sorting. We'll be talking about specific sorting algorithms today. I want to start by motivating why we're interested in sorting, which should be fairly easy. Then I want to discuss a particular sorting algorithm that's called insertion sort. That's probably the simplest sorting algorithm you can write, it's five lines of code. It's not the best sorting algorithm that's out there and so we'll try and improve it. We'll also talk about merge sort, which is a divide and conquer algorithm and that's going to motivate the last thing that I want to spend time on, which is recurrences and how you solve recurrences. Typically the recurrences that we'll be looking at in double o six are going to come from divide and conquer problems like merge sort but you'll see this over and over.

So let's talk about why we're interested in sorting. There's some fairly obvious applications like if you want to maintain a phone book, you've got a bunch of names and numbers corresponding to a telephone directory and you want to keep them in sorted order so it's easy to search, mp3 organizers, spreadsheets, et cetera. So there's lots of obvious applications. There's also some interesting problems that become easy once items are sorted. One example of that is finding a median.

So let's say that you have a bunch of items in an array a zero through n and a zero through n contains n numbers and they're not sorted. When you sort, you turn this into b 0 through n, where if it's just numbers, then you may sort them in increasing order or decreasing order. Let's just call it increasing order for now. Or if they're records, and they're not numbers, then you have to provide a comparison function to determine which record is smaller than another record. And that's another input that you have to have in order to do the sorting.

So it doesn't really matter what the items are as long as you have the comparison function. Think of it as less than or equal to. And if you have that and it's straightforward, obviously, to check that 3 is less than 4, et cetera. But it may be a little more complicated for more sophisticated sorting applications.

But the bottom line is that if you have your algorithm that takes a comparison function as an input, you're going to be able to, after a certain amount of time, get B 0 n. Now if you wanted to find the median of the set of numbers that were originally in the array A, what would you do once you have the sorted array B?

AUDIENCE: Isn't there a more efficient algorithm for median?

PROFESSOR: Absolutely. But this is sort of a side effect of having a sorted list. If you happen to have a sorted list, there's many ways that you could imagine building up a sorted list. One way is you have something that's completely unsorted and you run insertion sort or merge sort. Another way would be to maintain a sorted list as you're getting items put into the list.

So if you happened to have a sorted list and you need to have this sorted list for some reason, the point I'm making here is that finding the median is easy. And it's easy because all you have to do is look at-- depending on whether n is odd or even-- look at B of n over 2. That would give you the median because you'd have a bunch of numbers that are less than that and the equal set of numbers that are greater than that, which is the definition of median.

So this is not necessarily the best way, as you pointed out, of finding the median. But it's constant time if you have a sorted list. That's the point I wanted to make.

There are other things that you could do. And this came up in Erik's lecture, which is the notion of binary search-- finding an element in an array-- a specific element. You have a list of items-- again a 0 through n. And you're looking for a specific number or item.

You could, obviously, scan the array, and that would take you linear time to find this item. If the array happened to be sorted, then you can find this in logarithmic time using what's called binary search. Let's say you're looking for a specific item. Let's call it k. Binary search, roughly speaking, would work like-- you go compare k to, again, B of n over 2, and decide, given that B is sorted, you get to look at 1/2 of the array.

If B of n over 2 is not exactly k, then-- well, if it's exactly k you're done. Otherwise, you look at the left half. You do your divide and conquer paradigm. And you can do this in logarithmic time. So keep this in mind, because binary search is going to come up in today's lecture and again in other lectures.

It's really a great paradigm of divide and conquer-- probably the simplest. And it, essentially, takes something that's linear-- a linear search-- and turns it into logarithmic search. So those are a couple of problems that become easy if you have a sorted list. And there's some not so obvious applications of sorting-- for example, data compression.

If you wanted to compress a file, one of the things that you could do is to-- and it's a set of items-- you could sort the items. And that automatically finds duplicates. And you could say, if I have 100 items that are all identical, I'm going to compress the file by representing the item once and, then, having a number associated with the frequency of that item-- similar to what document distance does.

Document distance can be viewed as a way of compressing your initial input. Obviously, you lose the works of Shakespeare or whatever it was. And it becomes a bunch of words and frequencies. But it is something that compresses the input and gives you a different representation. And so people use sorting as a subroutine in data compression.

Computer graphics uses sorting. Most of the time, when you render scenes in computer graphics, you have many layers corresponding to the scenes. It turns out that, in computer graphics, most of the time you're actually rendering front to back because, when you have a big opaque object in front, you want to render that first, so you don't have to worry about everything that's occluded by this big opaque object. And that makes things more efficient. And so you keep things sorted front to back, most of the time, in computer graphics rendering.

But some of the time, if you're worried about transparency, you have to render things back to front. So typically, you have sorted lists corresponding to the different objects in both orders-- both increasing order and decreasing order. And you're maintaining that. So sorting is a real important subroutine in pretty much any sophisticated application you look at.

So it's worthwhile to look at the variety of sorting algorithms that are out there. And we're going to do some simple ones, today. But if you go and look at Wikipedia and do a Google search, there's all sorts of sorts like cocktail sort, and bitonic sort, and what have you. And there's reasons why each of these sorting algorithms exist. Because in specific cases, they end up winning on types of inputs or types of problems.

So let's take a look at our first sorting algorithm. I'm not going to write code but it will be in the notes. And it is in your document distance Python files. But I'll just give you pseudocode here and walk through what insertion sort looks like because the purpose of describing this algorithm to you is to analyze its complexity. We need to do some counting here, with respect to this algorithm, to figure out how fast it's going to run in and what the worst case complexity is.

So what is insertion sort? For i equals 1, 2, through n, given an input to be sorted, what we're going to do is we're going to insert A of i in the right position. And we're going to assume that we are sort of midway through the sorting process, where we have sorted A 0 through i minus 1. And we're going to expand this to this array to have i plus 1 elements. And A of i is going to get inserted into the correct position.

And we're going to do this by pairwise swaps down to the correct position for the number that is initially in A of i. So let's go through an example of this. We're going to sort in increasing order. Just have six numbers. And initially, we have 5, 2, 4, 6, 1, 3. And we're going to take a look at this.

And you start with the index 1, or the second element, because the very first element-- it's a single element and it's already sorted by definition. But you start from here. And this is what we call our key. And that's essentially a pointer to where we're at, right now. And the key keeps moving to the right as we go through the different steps of the algorithm.

And so what you do is you look at this and you have-- this is A of i. That's your key. And you have A of 0 to 0, which is 5. And since we want to sort in increasing order, this is not sorted. And so we do a swap.

So what this would do in this step is to do a swap. And we would go obtain 2, 5, 4, 6, 1, 3. So all that's happened here, in this step-- in the very first step where the key is in the second position-- is one swap happened.

Now, your key is here, at item 4. Again, you need to put 4 into the right spot. And so you do pairwise swaps. And in this case, you have to do one swap. And you get 2, 4, 5. And you're done with this iteration.

So what happens here is you have 2, 4, 5, 6, 1, 3. And now, the key is over here, at 6. Now, at this point, things are kind of easy, in the sense that you look at it and you say, well, I know this part is already started. 6 is greater than 5. So you have to do nothing.

So there's no swaps that happen in this step. So all that happens here is you're going to move the key to one step to the right. So you have 2, 4, 5, 6, 1, 3. And your key is now at 1. Here, you have to do more work. Now, you see one aspect of the complexity of this algorithm-- given that you're doing pairwise swaps-- the way this algorithm was defined, in pseudocode, out there, was I'm going to use pairwise swaps to find the correct position.

So what you're going to do is you're going to have to swap first 1 and 6. And then you'll swap-- 1 is over here. So you'll swap this position and that position. And then you'll swap-- essentially, do 4 swaps to get to the point where you have 1, 2, 4, 5, 6, 3. So this is the result. 1, 2, 4, 5, 6, 3.

And the important thing to understand, here, is that you've done four swaps to get 1 to the correct position. Now, you could imagine a different data structure where you move this over there and you shift them all to the right. But in fact, that shifting of these four elements is going to be computed in our model as four operations, or four steps, anyway. So there's no getting away from the fact that you have to do four things here. And the way the code that we have for insertion sort does this is by using pairwise swaps.

So we're almost done. Now, we have the key at 3. And now, 3 needs to get put into the correct position. And so you've got to do a few swaps. This is the last step. And what happens here is 3 is going to get swapped with 6. And then 3 needs to get swapped with 5. And then 3 needs to get swapped with 4. And then, since 3 is greater than 2, you're done. So you have 1, 2, 3, 4, 5, 6.

And that's it. So, analysis. How many steps do I have?

AUDIENCE: n squared?

PROFESSOR: No, how many steps do I have? I guess that wasn't a good question. If I think of a step as being a movement of the key, how many steps do I have? I have theta n steps. And in this case, you can think of it as n minus 1 steps, since you started with 2. But let's just call it theta n steps, in terms of key positions.

And you're right. It is n square because, at any given step, it's quite possible that I have to do theta n work. And one example is this one, right here, where I had to do four swaps. And in general, you can construct a scenario where, towards the end of the algorithm, you'd have to do theta n work. But if you had a list that was reverse sorted. You would, essentially, have to do, on an average n by two swaps as you go through each of the steps. And that's theta n.

So each step is theta n swaps. And when I say swaps, I could also say each step is theta n compares and swaps. And this is going to be important because I'm going to ask you an interesting question in a minute. But let me summarize. What I have here is a theta n squared algorithm. The reason this is a theta n squared algorithm is because I have theta n steps and each step is theta n.

When I'm counting, what am I counting it terms of operations? The assumption here-- unspoken assumption-- has been that an operation is a compare and a swap and they're, essentially, equal in cost.

And in most computers, that's true. You have a single instruction and, say, the x86 or the MIPS architecture that can do a compare, and the same thing for swapping registers. So perfectly reasonably assumption that compares and swaps for numbers have exactly the same cost. But if you had a record and you were comparing records, and the comparison function that you used for the records was in itself a method call or a subroutine, it's quite possible that all you're doing is swapping pointers or references to do the swap, but the comparison could be substantially more expensive.

Most of the time-- and we'll differentiate if it becomes necessary-- we're going to be counting comparisons in the sorting algorithms that we'll be putting out. And we'll be assuming that either comparison swaps are roughly the same or that compares are-- and we'll say which one, of course-- that compares are substantially more expensive than swaps. So if you had either of those cases for insertion sort, you have a theta n squared algorithm. You have theta n squared compares and theta n squared swaps.

Now, here's a question. Let's say that compares are more expensive than swaps. And so, I'm concerned about the theta n squared comparison cost. I'm not as concerned, because of the constant factors involved, with the theta n squared swap cost.

This is a question question. What's a simple fix-- change to this algorithm that would give me a better complexity in the case where compares are more expensive, or I'm only looking at the complexity of compares. So the theta whatever of compares. Anyone? Yeah, back there.

AUDIENCE: [INAUDIBLE]

PROFESSOR: You could compare with the middle. What did I call it? I called it something. What you just said, I called it something.

AUDIENCE: Binary search.

PROFESSOR: Binary search. That's right. Two cushions for this one. So you pick them up after lecture. So you're exactly right. You got it right. I called it binary search, up here.

And so you can take insertion sort and you can sort of trivially turn it into a theta n log n algorithm if we are talking about n being the number of compares. And all you have to do to do that is to say, you know what, I'm going to replace this with binary search. And you can do that-- and that was the key observation-- because A of 0 through i minus 1 is already sorted. And so you can do binary search on that part of the array.

So let me just write that down. Do a binary search on A of 0 through i minus 1, which is already sorted. And essentially, you can think of it as theta log i time, and for each of those steps. And so then you get your theta n log n theta n log n in terms of compares.

Does this help the swaps for an array data structure? No, because binary search will require insertion into A of 0 though i minus 1. So here's the problem. Why don't we have a full-fledged theta n log n algorithm, regardless of the cost of compares or swaps? We don't quite have that. We don't quite have that because we need to insert our A of i into the right position into A of 0 through i minus 1.

You do that if you have an array structure, it might get into the middle. And you have to shift things over to the right. And when you shift things over to the right, in the worst case, you may be shifting a lot of things over to the right. And that gets back to worst case complexity of theta n.

So a binary search in insertion sort gives you theta n log n for compares. But it's still theta n squared for swaps. So as you can see, there's many varieties of sorting algorithms. We just looked at a couple of them. And they were both insertion sort.

The second one that I just put up is, I guess, technically called binary insertion sort because it does binary search. And the vanilla insertion sort is the one that you have the code for in the doc dis program, or at least one of the doc dis files.

So let's move on and talk about a different algorithm. So what we'd like to do, now-- this class is about constant improvement. We're never happy. We always want to do a little bit better. And eventually, once we run out of room from an asymptotic standpoint, you take these other classes where you try and improve constant factors and get 10%, and 5%, and 1%, and so on, and so forth.

But we'll stick to improving asymptotic complexity. And we're not quite happy with binary insertion sort because, in the case of numbers, our binary insertion sort has theta n squared complexity, if you look at swaps. So we'd like to go find an algorithm that is theta n log n.

And I guess, eventually, we'll have to stop. But Erik will take care of that. There's a reason to stop. It's when you can prove that you can't do any better. And so we'll get to that, eventually.

So merge sort is also something that you've probably seen. But there probably will be a couple of subtleties that come out as I describe this algorithm that, hopefully, will be interesting to those of you who already know merge sort. And for those of you who don't, it's a very pretty algorithm.

It's a standard recursion algorithm-- recursive algorithm-- similar to a binary search. What we do, here, is we have an array, A. We split it into two parts, L and R. And essentially, we kind of do no work, really.

In terms of the L and R in the sense that we just call, we keep splitting, splitting, splitting. And all the work is done down at the bottom in this routine called merge, where we are merging a pair of elements at the leaves. And then, we merge two pairs and get four elements. And then we merge four tuples of elements, et cetera, and go all the way up.

So while I'm just saying L terms into L prime, out here, there's no real explicit code that you can see that turns L into L prime. It happens really later. There's no real sorting code, here. It happens in the merge routine. And you'll see that quite clearly when we run through an example.

So you have L and R turn into L prime and R prime. And what we end up getting is a sorted array, A. And we have what's called a merge routine that takes L prime and R prime and merges them into the sorted array.

So at the top level, what you see is split into two, and do a merge, and get to the sorted array. The input is of size n. You have two arrays of size n over 2. These are two sorted arrays of size n over 2. And then, finally, you have a sorted array of size n.

So if you want to follow the recursive of execution of this in a small example, then you'll be able to see how this works. And we'll do a fairly straightforward example with 8 elements. So at the top level-- before we get there, merge is going to assume that you have two sorted arrays, and merge them together. That's the invariant in merge sort, or for the merge routine. It assumes the inputs are sorted-- L and R. Actually I should say, L prime and R prime.

So let's say you have 20, 13, 7, and 2. You have 12, 11, 9, and 1. And this could be L prime. And this could be R prime. What you have is what we call a two finger algorithm.

And so you've got two fingers and each of them point to something. And in this case, one of them is pointing to L. My left finger is pointing to L prime, or some element L prime. My right finger is pointing to some element in R prime.

And I'm going to compare the two elements that my fingers are pointing to. And I'm going to choose, in this case, the smaller of those elements. And I'm going to put them into the sorted array.

So start out here. Look at that and that. And I compared 2 and 1. And which is smaller? 1 is smaller. So I'm going to write 1 down. This is a two finger algo for merge. And I put 1 down.

When I put 1 down, I had to cross out 1. So effectively, what happens is-- let me just circle that instead of crossing it out. And my finger moves up to 9. So now I'm pointing at 2 and 9. And I repeat this step.

So now, in this case, 2 is smaller. So I'm going to go ahead and write 2 down. And I can cross out 2 and move my finger up to 7. And so that's it. I won't bore you with the rest of the steps.

It's essentially walking up. You have a couple of pointers and you're walking up these two arrays. And you're writing down 1, 2, 7, 9, 11, 12, 13, 20. And that's your merge routine.

And all of the work, really, is done in the merge routine because, other than that, the body is simply a recursive call. You have to, obviously, split the array. But that's fairly straightforward.

If you have an array, A 0 through n-- and depending on whether n is odd or even-- you could imagine that you set L to be A 0 n by 2 minus 1, and R similarly. And so you just split it halfway in the middle. I'll talk about that a little bit more. There's a subtlety associated with that that we'll get to in a few minutes.

But to finish up in terms of the computation of merge sort. This is it. The merge routine is doing most, if not all, of the work. And this two finger algorithm is going to be able to take two sorted arrays and put them into a single sorted array by interspersing, or interleaving, these elements. And what's the complexity of merge if I have two arrays of size n over 2, here? What do I have?

AUDIENCE: n.

PROFESSOR: n. We'll give you a cushion, too. theta n complexity. So far so good.

I know you know the answer as to what the complexity of merge sort is. But I'm guessing that most of you won't be able to prove it to me because I'm kind of a hard guy to prove something to. And I could always say, no, I don't believe you or I don't understand.

The complexity-- and you've said this before, in class, and I think Erik's mentioned it-- the overall complexity of this algorithm is theta n log n And where does that come from? How do you prove that?

And so what we'll do, now, is take a look at merge sort. And we'll look at the recursion tree. And we'll try and-- there are many ways of proving that merge sort is theta n log n.

The way we're going to do this is what's called proof by picture. And it's not an established proof technique, but it's something that is very helpful to get an intuition behind the proof and why the result is true. And you can always take that and you can formalize it and make this something that everyone believes. And we'll also look at substitution, possibly in section tomorrow, for recurrence solving.

So where we're right now is that we have a divide and conquer algorithm that has a merge step that is theta n. And so, if I just look at this structure that I have here, I can write a recurrence for merge sort that looks like this. So when I say complexity, I can say T of n, which is the work done for n items, is going to be some constant time in order to divide the array.

So this could be the part corresponding to dividing the array. And there's going to be two problems of size n over 2. And so I have 2 T of n over 2. And this is the recursive part. And I'm going to have c times n, which is the merge part. And that's some constant times n, which is what we have, here, with respect to the theta n complexity.

So you have a recurrence like this and I know some of you have seen recurrences in 6.042. And you know how to solve this. What I'd like to do is show you this recursion tree expansion that, not only tells you how to solve this occurrence, but also gives you a means of solving recurrences where, instead of having c of n, you have something else out here. You have f of n, which is a different function from the linear function. And this recursion tree is, in my mind, the simplest way of arguing the theta n log n complexity of merge sort.

So what I want to do is expand this recurrence out. And let's do that over here. So I have c of n on top. I'm going to ignore this constant factor because c of n dominates. So I'll just start with c of n. I want to break things up, as I do the recursion.

So when I go c of n, at the top level-- that's the work I have to do at the merge, at the top level. And then when I go down to two smaller problems, each of them is size n over 2. So I do c times n divided by 2 [INAUDIBLE]. So this is just a constant c. I didn't want to write thetas up here. You could. And I'll say a little bit more about that later. But think of this cn as representing the theta n complexity. And c is this constant.

So c times n, here. c times n over 2, here. And then when I keep going, I have c times n over 4, c times n over 4, et cetera, and so on, and so forth. And when I come down all the way here, n is eventually going to become 1-- or essentially a constant-- and I'm going to have a bunch of c's here.

So here's another question, that I'd like you to answer. Someone tell me what the number of levels in this tree are, precisely, and the number of leaves in this tree are, precisely.

AUDIENCE: The number of levels is log n plus 1.

PROFESSOR: Log n plus 1. Log to the base 2 plus 1. And the number of leaves? You raised your hand back there, first. Number of leaves.

AUDIENCE: I think n.

PROFESSOR: Yeah, you're right. You think right. So 1 plus log n and n leaves. When n becomes 1, how many of them do you have? You're down to a single element, which is, by definition, sorted. And you have n leaves.

So now let's add up the work. I really like this picture because it's just so intuitive in terms of getting us the result that we're looking for. So you add up the work in each of the levels of this tree. So the top level is cn. The second level is cn because I added 1/2 and 1/2, cn, cn. Wow. What symmetry.

So you're doing the same amount of work modulo the constant factors, here, with what's going on with the c1, which we've ignored, but roughly the same amount of work in each of the levels. And now, you know how many levels there are. It's 1 plus log n.

So if you want to write an equation for T of n, it's 1 plus log n times c of n, which is theta of n log n. So I've mixed in constants c and thetas. For the purposes of this description, they're interchangeable. You will see recurrences that look like this, in class. And things like that.

Don't get confused. It's just a constant multiplicative factor in front of the function that you have. And it's just a little easier, I think, to write down these constant factors and realize that the amount of work done is the same in each of the leaves. And once you know the dimensions of this tree, in terms of levels and in terms of the number of leaves, you get your result.

So we've looked at two algorithm, so far. And insertion sort, if you talk about numbers, is theta n squared for swaps. Merge sort is theta n log n. Here's another interesting question. What is one advantage of insertion sort over merge sort?

AUDIENCE: [INAUDIBLE]

PROFESSOR: What does that mean?

AUDIENCE: You don't have to move elements outside of [INAUDIBLE].

PROFESSOR: That's exactly right. That's exactly right. So the two guys who answered the questions before with the levels, and you. Come to me after class.

So that's a great answer. It's in-place sorting is something that has to do with auxiliary space. And so what you see, here-- and it was a bit hidden, here. But the fact of the matter is that you had L prime and R prime. And L prime and R prime are different from L and R, which were the initial halves of the inputs to the sorting algorithm.

And what I said here is, we're going to dump this into A. That's what this picture shows. This says sorted array, A. And so you had to make a copy of the array-- the two halves L and R-- in order to do the recursion, and then to take the results and put them into the sorted array, A.

So you needed-- in merge sort-- you needed theta n auxiliary space. So merge sort, you need theta n extra space. And the definition of in-place sorting implies that you have theta 1-- constant-- auxiliary space.

The auxiliary space for insertion sort is simply that temporary variable that you need when you swap two elements. So when you want to swap a couple of registers, you gotta store one of the values in a temporary location, override the other, et cetera. And that's the theta 1 auxiliary space for insertion sort.

So there is an advantage of the version of insertion sort we've talked about, today, over merge sort. And if you have a billion elements, that's potentially something you don't want to store in memory. If you want to do something really fast and do everything in cache or main memory, and you want to sort billions are maybe even trillions of items, this becomes an important consideration.

I will say that you can reduce the constant factor of the theta n. So in the vanilla scheme, you could imagine that you have to have a copy of the array. So if you had n elements, you essentially have n extra items of storage. You can make that n over 2 with a simple coding trick by keeping 1/2 of A.

You can throw away one of the L's or one of the R's. And you can get it down to n over 2. And that turns out-- it's a reasonable thing to do if you have a billion elements and you want to reduce your storage by a constant factor. So that's one coding trick.

Now it turns out that you can actually go further. And there's a fairly sophisticated algorithm that's sort of beyond the scope of 6.006 that's an in-place merge sort. And this in-place merge sort is kind of impractical in the sense that it doesn't do very well in terms of the constant factors.

So while it's in-place and it's still theta n log n. The problem is that the running time of an in-place merge sort is much worse than the regular merge sort that uses theta n auxiliary space.

So people don't really use in-place merge sort. It's a great paper. It's a great thing to read. Its analysis is a bit sophisticated for double 0 6. So we wont go there. But it does exist. So you can take merge sort, and I just want to let you know that you can do things in-place.

In terms of numbers, some experiments we ran a few years ago-- so these may not be completely valid because I'm going to actually give you numbers-- but merge sort in Python, if you write a little curve fit program to do this, is 2.2n log n microseconds for a given n.

So this is the merge sort routine. And if you look at insertion sort, in Python, that's something like 0.2 n square microseconds. So you see the constant factors here.

If you do insertion sort in C, which is a compiled language, then, it's much faster. It's about 20 times faster. It's 0.01 n squared microseconds. So a little bit of practice on the side. We do ask you to write code. And this is important. The reason we're interested in algorithms is because people want to run them.

And what you can see is that you can actually find an n-- so regardless of whether you're Python or C, this tells you that asymptotic complexity is pretty important because, once n gets beyond about 4,000, you're going to see that merge sort in Python beats insertion sort in C.

So the constant factors get subsumed beyond certain values of n. So that's why asymptotic complexity is important. You do have a factor of 20, here, but that doesn't really help you in terms of keeping an n square algorithm competitive. It stays competitive for a little bit longer, but then falls behind.

That's what I wanted to cover for sorting. So hopefully, you have a sense of what happens with these two sorting algorithms. We'll look at a very different sorting algorithm next time, using heaps, which is a different data structure.

The last thing I want to do in the couple minutes I have left is give you a little more intuition as to recurrence solving based on this diagram that I wrote up there. And so we're going to use exactly this structure. And we're going to look at a couple of different recurrences that I won't really motivate in terms of having a specific algorithm, but I'll just write out the recurrence. And we'll look at the recursion tree for that. And I'll try and tease out of you the complexity associated with these recurrences of the overall complexity.

So let's take a look at T of n equals 2 T of n over 2 plus c n squared. Let me just call that c-- no need for the brackets. So constant c times n squared.

So if you had a crummy merge routine, and it was taking n square, and you coded it up wrong. It's not a great motivation for this recurrence, but it's a way this recurrence could have come up.

So what does this recursive tree look like? Well it looks kind of the same, obviously. You have c n square; you have c n square divided by 4; c n square divided by 4; c n square divided by 16, four times. Looking a little bit different from the other one.

The levels and the leaves are exactly the same. Eventually n is going to go down to 1. So you will see c all the way here. And you're going to have n leaves. And you will have, as before, 1 plus log n levels. Everything is the same.

And this is why I like this recursive tree formulation so much because, now, all I have to do is add up the work associated with each of the levels to get the solution to the recurrence. Now, take a look at what happens, here. c n square; c n square divided by 2; c n square divided by 4. And this is n times c. So what does that add up to?

AUDIENCE: [INAUDIBLE]

PROFESSOR: Yeah, exactly. Exactly right. So if you look at what happens, here, this dominates. All of the other things are actually less than that. And you said bounded by two c n square because this part is bounded by c n square and I already have c n square up at the top.

So this particular algorithm that corresponds to this crummy merge sort, or wherever this recurrence came from, is a theta n squared algorithm. And in this case, all of the work done is at the root-- at the top level of the recursion. Here, there was a roughly equal amount of work done in each of the different levels. Here, all of the work was done at the root.

And so to close up shop, here, let me just give you real quick a recurrence where all of the work is done at the leaves, just for closure. So if I had, magically, a merge routine that actually happened in constant time, either through buggy analysis, or because of it was buggy, then what does the tree look like for that?

And I can think of this as being theta 1. Or I can think of this as being just a constant c. I'll stick with that. So I have c, c, c. Woah, I tried to move that up. That doesn't work.

So I have n leaves, as before. And so if I look at what I have, here, I have c at the top level. I have 2c, and so on and so forth. 4c. And then I go all the way down to nc.

And so what happens here is this dominates. And so, in this recurrence, the whole thing runs in theta n. So the solution to that is theta n. And what you have here is all of the work being done at the leaves.

We're not going to really cover this theorem that gives you a mechanical way of figuring this out because we think the recursive tree is a better way of looking at. But you can see that, depending on what that function is, in terms of the work being done in the merge routine, you'd have different versions of recurrences. I'll stick around, and people who answered questions, please pick up you cushions. See you next time.