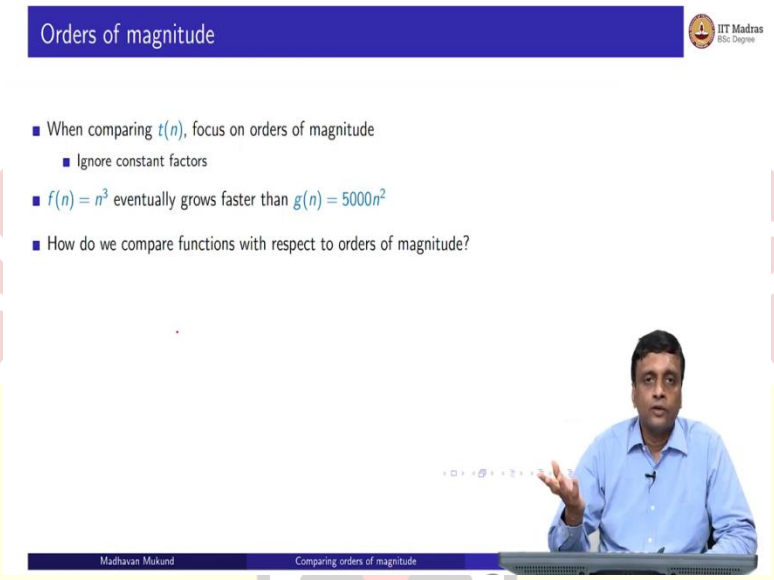
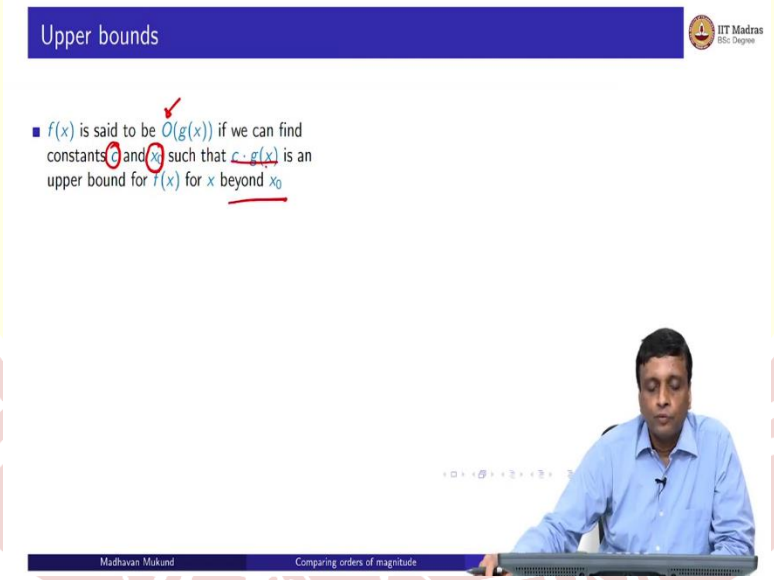
IIT Madras

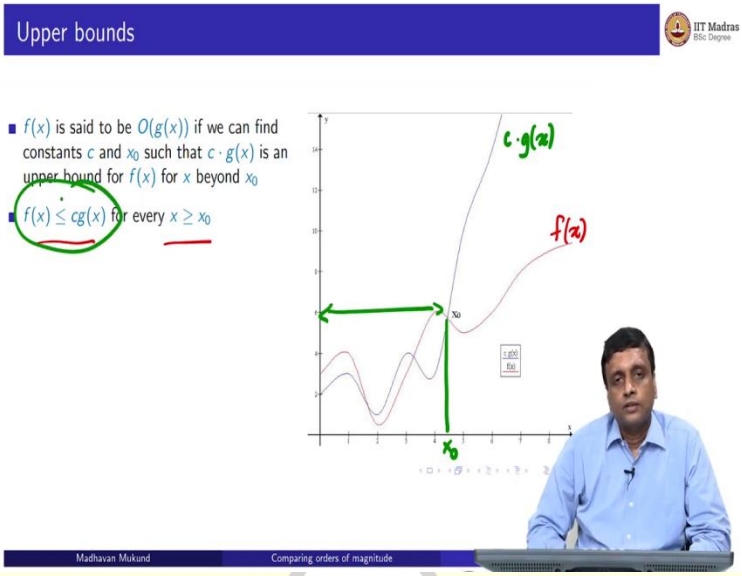
ONLINE DEGREE

Programming, Data Structures and Algorithms using Python
Professor Madhavan Mukund
Comparing Orders of Magnitude

(Refer Slide Time: 00:13)







So, we are talking about the analysis of algorithms. So, we said that we are interested in measuring up to orders of magnitude. And we are looking at this asymptotic complexity. So, we want to know what happens is n becomes large. So, we would like to say that n cube eventually grows faster than 5000n squared. So, we now want some way of writing this down precisely. How do we argue? Or how do we describe the fact that f of n is a bigger function of n than g of n. So, how do we compare functions so we can say this algorithm is better than that algorithm asymptotically?

So, in order to compare one function against another function, we have this notion of an upper bounds. So, this big O notation as it is called so this big O. So, this O here is a capital O, and it is called Big O. So, we say that f of x is big O of g of x. If in some sense g of x sits above f of x. Now, because we are ignoring constants, and orders of magnitude is what matters, we are allowed to make g of x sit above f of x by multiplying g of x by some suitable constant.

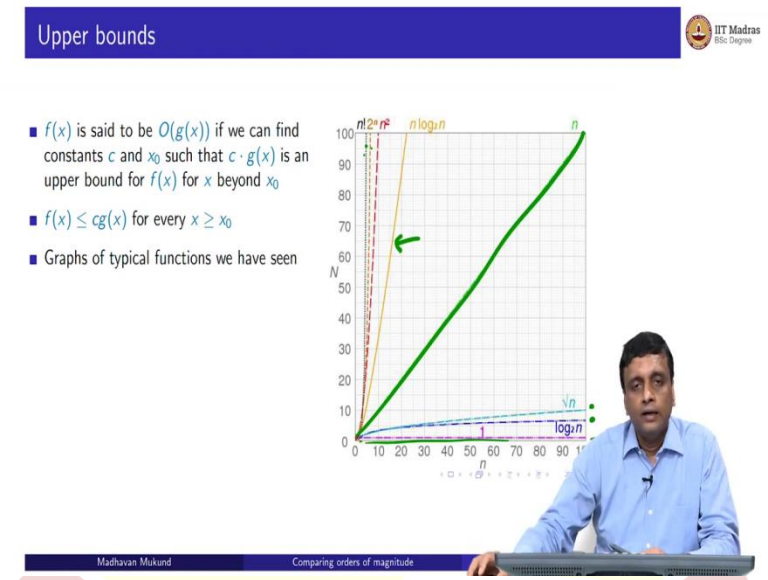
And like we saw with that n cubed and n squared example, this may not happen initially. So, they you might need to go to large enough n for this to happen. So, you need to find two values, you need to find an x not. And you need to find a c, such that if you go beyond x not, then f of x is below c of g of x. So, if you want to write it more mathematically, you say that for every x bigger than equal to x not, f of x is smaller than equal to c times g of x.

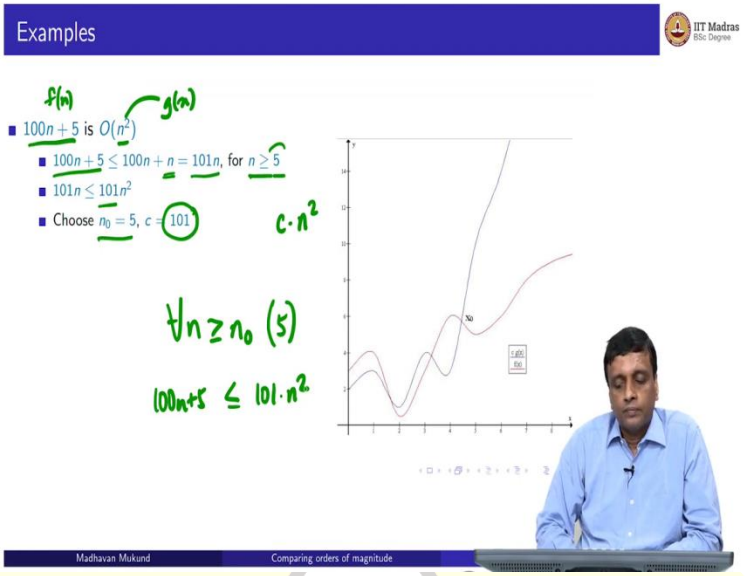
So, you have to find I mean claim that f of x is over g of x, you have to find so g and f are given to you. You are told what is f of x, you are told, what is g of x? The question is, is g of x an

upper bound for f of x. In other words, f of x, in general, smaller than g of x as a function of x. So, for this, you need to find these two constants, c the multiplier for g and x not, the threshold beyond which this holds.

So, here is a picture on the right, so this is our f of x and for some suitably chosen, chosen g, this is our c times g of x. And this, at this point is our x not. So, if you believe that this function continues to behave similarly to the right beyond what we cannot see, you can see that in this interval, the relationship between f and c times g is a little bit ambiguous. Sometimes f is below g, sometimes f is above g. But once it crosses x not, you have this property that f of x is always below c times g of x. So, this means that in such a situation, I can declare that f of x is big O of g of x.

(Refer Slide Time: 02:52)





So, just to give an idea about how the, so basically it is saying that one graph is above the other graph that is what we are saying, well, if you draw it. So, here are some of the graphs that we talked about when we saw this table of functions. So, here is a particularly simple graph, which is just a constant function.

So, this is a so not many algorithms have this kind of constant behavior. But you can imagine an algorithm which takes a list and returns the first element of the list. So, it does not really matter how large the list is, because it is always going to just pick up the first element which is available and give it to you.

So, this is a typical example of an algorithm which takes constant time its input does not matter. So, then you obviously have a straight line at the bottom. Now, log n is kind of hugging the bottom, it does not grow very fast, square root of n also does not grow very fast, but it grows faster than log of n. So, square root of n is an upper bound for log of n.

So, you can say log of n will be big O of square root of n, but not vice versa. This is our y equal to n, where I have a straight line, which is kind of at 45 degrees. And now n log n is it looks almost like a straight line. But it is, basically at this scale, it looks like it is closer to n squared. But as this line grows up, you can find out that n log n is much closer to n than it is to n squared.

So, n squared is this parabola n cube, we have not drawn two to the n is even steeper than n squared, and n factorial is even steeper than two to the n. So, this we saw in the table last time

and now you can see it in the graph. So, in Big O sense, each of the lower curves is going to be in big O of the upper curve.

So, what I said before is that to show something formerly is big O, you have to demonstrate these two constants, c and x not. So, let us do a couple of examples. And then we will see a simpler way of doing it informally. So, let us look at two functions 100n plus 5, and n squared. So, I want to argue that 100n plus 5, so this is my f of n. And my g of n is n squared.

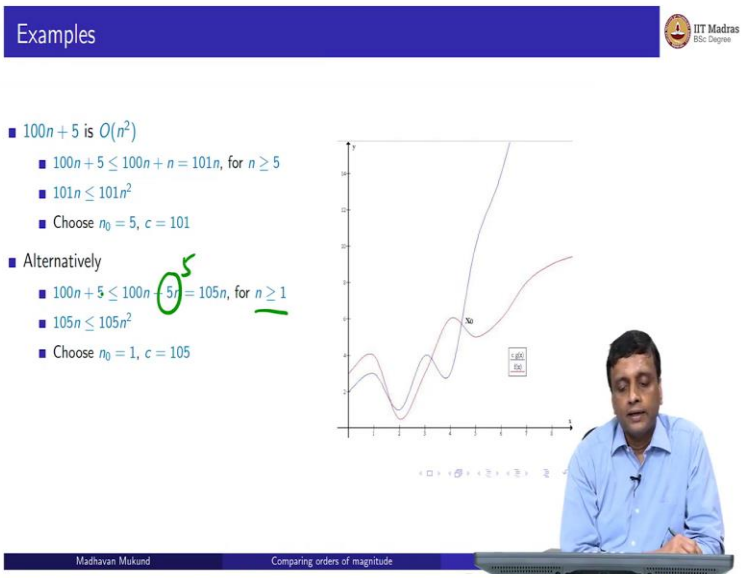
So, I want to argue that f of n is big O of g of n where g of n is n squared. So, I need to find this constant. So, I do a little bit of manipulation, I look at 100n plus 5. And I say that, 5 will be less than n, if n is bigger than 5. So, if I take 100n plus 5, and instead I write 100n plus n, then the right hand side is going to be bigger than the left hand side, provided n is bigger than 5.

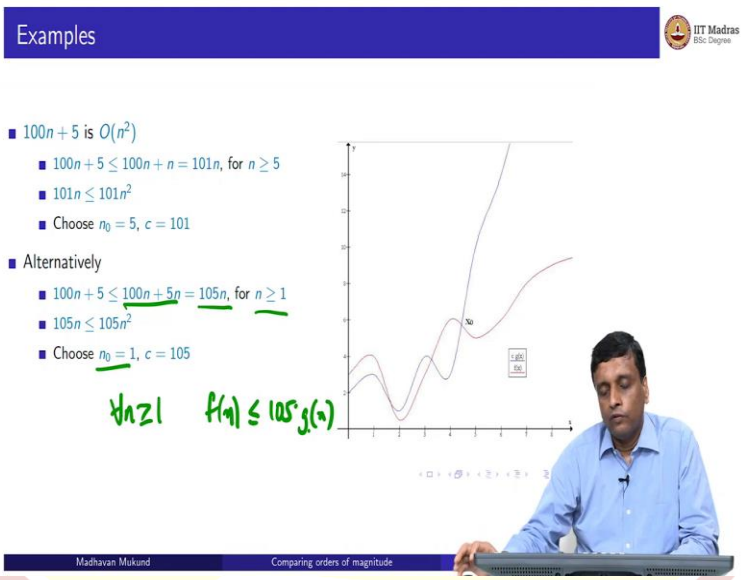
So, I choose n bigger than 5, then I have 100n plus n, but 100n plus n is nothing but 101. So, 100n plus 5 is less than equal to 101n. Since n is always positive, remember, we are looking at functions where n describes an input size, and obviously, the input size is not going to be negative or something.

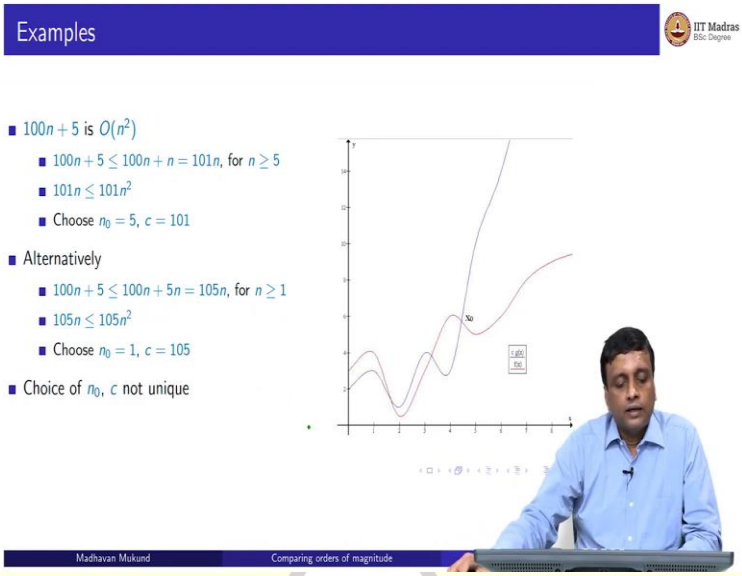
So, since n is only going to be positive 101n for n bigger than 5 is going to be smaller than 101n squared. So, therefore, if I take now 101n squared, it is of the form c times n squared and n square to the function I was looking at. So, my constant that I have discovered is 101. And the lower bound beyond which this happens is 5. So, n not is 5.

So, for all what we are saying is for all n greater than n not, which is 5, in this case, 100n plus 5 is less than or equal to 101 times n squared. So, this is the kind of analysis that we have done to show that 100n plus 5 is big O of n squared. Now, it turns out this is not unique, I mean, this choice of c and n not really depends on your cleverness at finding out how to establish this and there may be more than one way to do it.

(Refer Slide Time: 06:35)



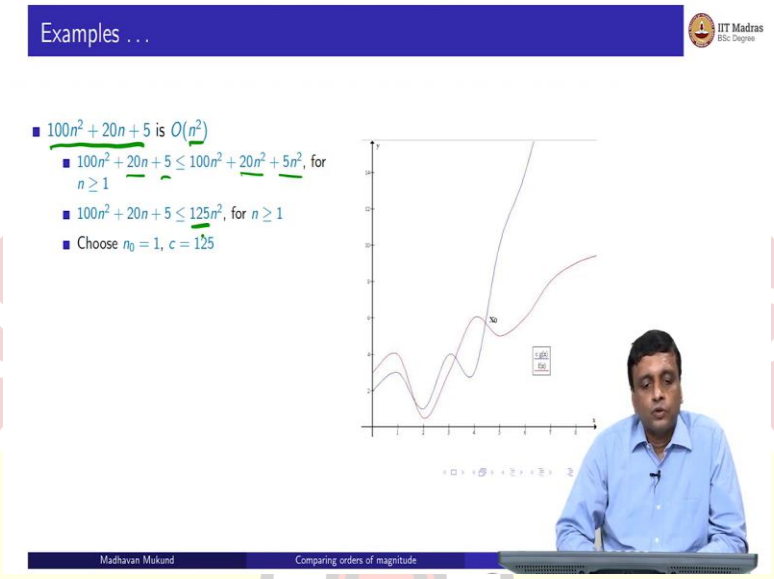


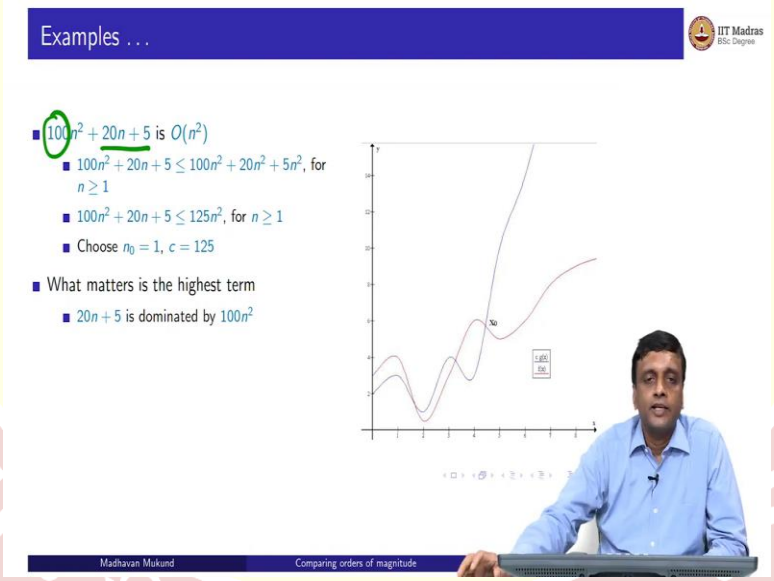


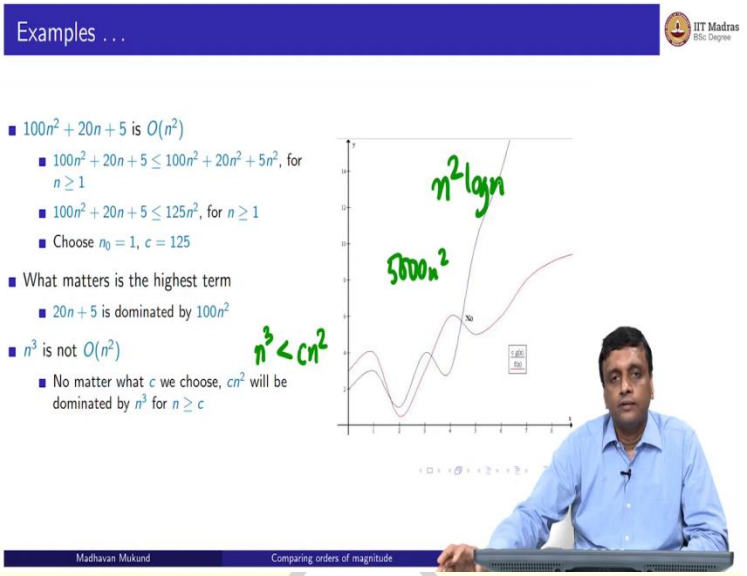
So, for the same example, instead of writing 100n plus n. I could write 100n plus 5 times n, that is also true, except now it becomes true even for n bigger than equal to 1, because once I have n equal to 1, 5n becomes 5, so this is 5 and this is 5, and this is 5, so they are the same. And if I go to something like two, then 5 n will become 10.

And so, this right-hand side will become larger than 5. So, therefore, for n greater than 1, we have 100n plus 5 is smaller than 100n plus 5n. And now, if I combine these two, I get 105. So, now I have a different choice, I say now, if I take any for all n bigger than 1. I have that f of n is smaller than equal to 105 times g of n. So, now I have a different n not in a different c. So, this is one way of doing this.

(Refer Slide Time: 07:28)







And it shows that the choice of n not and c is not unique. Now, let us look at something where both sides have the same kind of highest degree. So, we have here a quadratic 100n squared plus 20n plus 5, this is our f of n. And we have another quadratic which is does not have any coefficients does not have any linear terms and so on just n squared.

But I still claim that 100n squared plus 20n plus 5 is big O of n squared. So, again, you can do a similar trick, you can kind of convert all of these into n squared form. So, you can say that for n bigger than equal to 1 this 100n 20n multiply by n to get 20n squared, 5n multiply by n squared to get 5 n squared.

So, I get that this whole thing is smaller than 100n squared plus 20n squared plus 5n squared. So, if I combine that I get 125n squared. So, now, we can say that the left hand side though it has a kind of a bigger coefficient for the n squared term, and it has these linear terms, it is still below 125 times just n squared. So this left hand side is big O of n squared.

So, what really matters, in some sense is the highest term. So, what we are saying is that any polynomial in which the highest term is 2, is big O of any other polynomial is basically big O of n squared. So n squared is as big as any polynomial of degree two in this order of magnitude notation.

So, in particular, we have this constant 100, which does not matter, we also have this extra linear term, which is going to be dominated by the quadratic term. So, all these things go away. So,

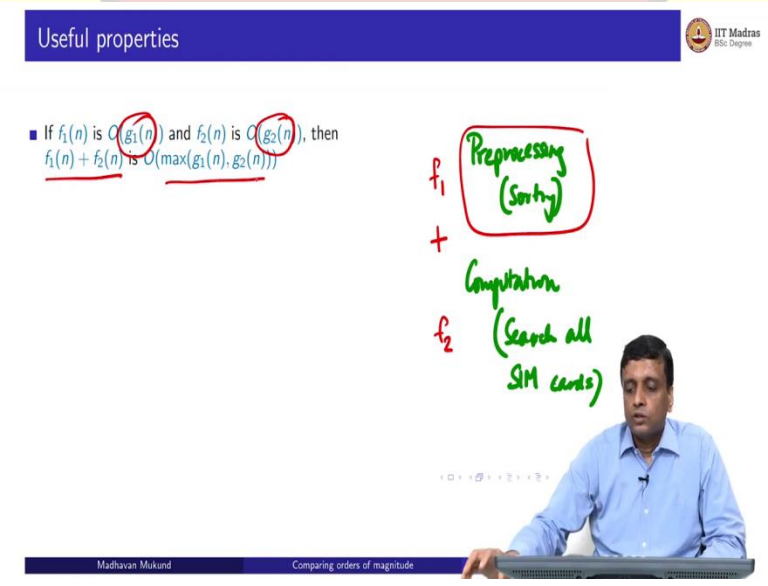
really, we are looking at the highest power when we are comparing, say polynomial, then the highest power is what matters.

So, by the same token, if I look at n cube, then it cannot be n squared, O of n squared, because if it is O of n squared, then I will get something like c times n squared. And I will (())(09:23) n cube to be less than c times n squared for all n above a certain thing, but we know that once we cross c, remember, we saw the particular example 5000n squared right at the beginning.

So, once we cross c n cubed is going to get higher than c times n squared no matter what c we choose, so we can never find such as c. And so something with a higher power n cubed can never be big O of something with a smaller power. So, the simple rule of thumb is you just look at your function, especially when it has polynomial terms.

And you can look at the highest power and decide based on that and of course, sometimes it is more than just a polynomial. It could be something like n squared log n or something. So, now n squared log n would be smaller than n cubed, because log n is smaller than n, but n squared log n will be bigger than n squared, and so on. So, you can do a rough and ready. We go calculation. But to formally verified, you have to go through the mechanics of using the constant and finding the minimum level beyond which it holds.

(Refer Slide Time: 10:24)

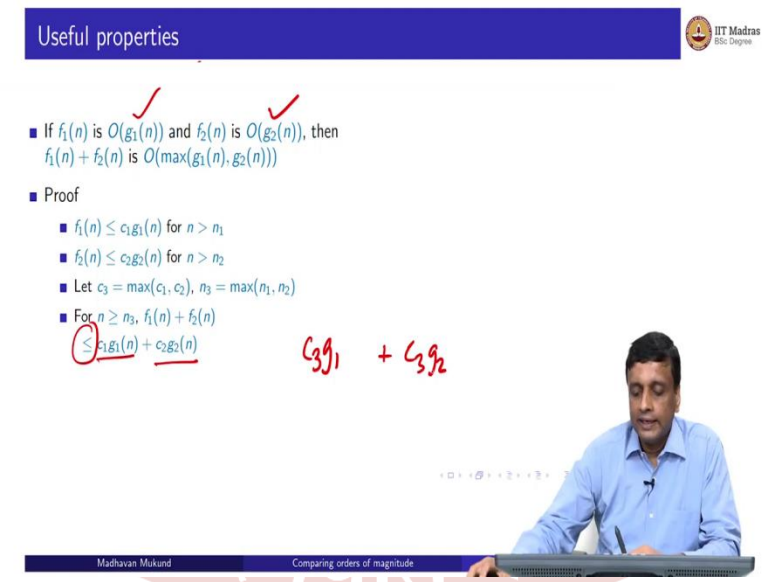


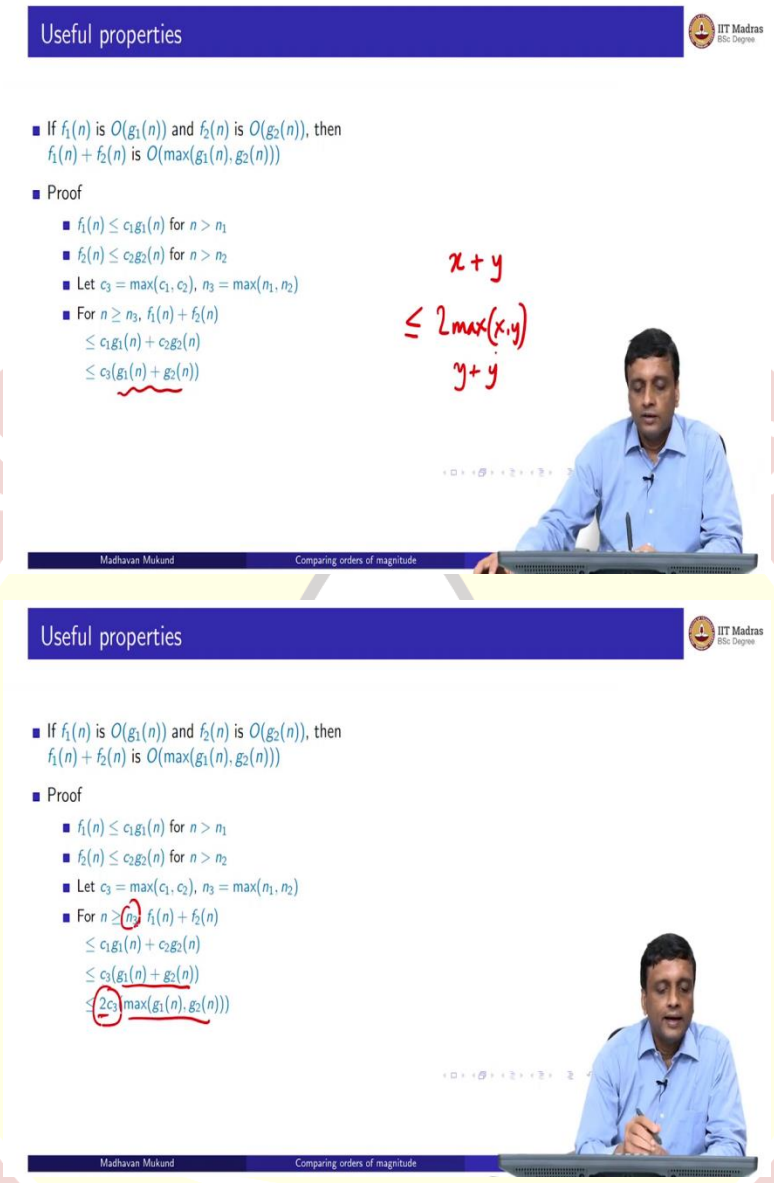
So, one of the useful properties of this big O notation is that if I add two functions, then they lie below the maximum of the two in terms of big O. So, we will see why this is important. So, supposing I have a pre processing step, like for instance, in the Aadhaar card example, this is my sorting step. And then I have an actual computation step.

This is a search, search all SIM cards. So, now, I might do some kind of analysis and get some f1 as the complexity of that, and I might do some analysis f2 as a complexity of this. So, I have two different algorithms, one which sorts the cards, and one assumes the cards are sorted and then compares. So, the total time that I want to reason about for this algorithm is f1 plus f2.

Because the first step to do f1 work, then have to do f2 work. So, that is what this is saying? So, f1 of n plus f2 of n taken input, I first do f1 work, and then I do f2 work. What can I say about this? If I know something about each individual step. So, in each individual step, I know that f1 is within g1 of n and O is f2 this is g2 as an. And the claim is that the whole thing f1 plus f2 is within the maximum of g1 and g2. So, it will not exceed the maximum of g1 and g2.

(Refer Slide Time: 12:00)





So, why is this the case? Well, just from the definition, we know that f 1 of n is below c 1 times g 1 for some n above n 1, and it is below f 2 is below c 2 times g 2 for n above n 2. This is just the big O definition being stated concretely for f 1 and g 1. So, I have these four constants c 1, c 2 and n 1, n 2. So what I will do? I will take the maximum in each case, I will let c 3 be the maximum of the 2 c is c 1 and c 2. And I will let n 3 be the maximum of n 1 and n 2.

So, now I want to look at what happens for f 1 n f 1 plus f 2. So, I will look above the maximum. So, I will look above n 3. So, since I am above maximum of n 1 and n 2, I know that f 1 is below its upper bound. And I know that f 2 is also below its upper bound, because both of them have already crossed their threshold, because n 3 is bigger than n 1 and n 3 is bigger than n 2, because

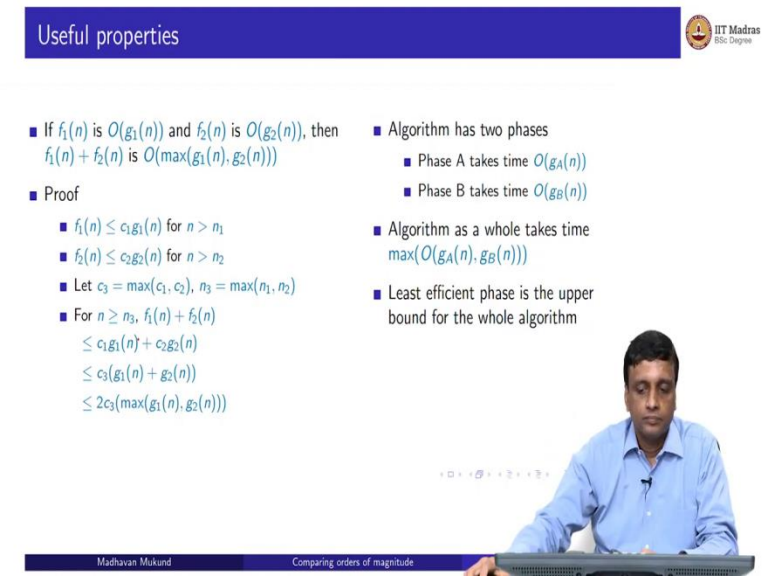
it is the maximum of the 2. So, for both f 1 and f 2, I can use this fact that they are big O and get this inequality saying that they are below c 1.

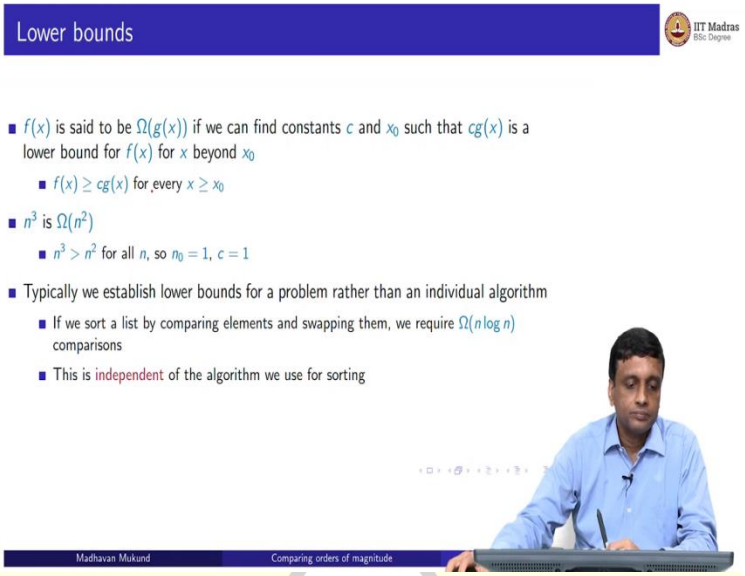
Now, because c 3 is bigger than both c 1 and c 2, I can write this as c 3 g 1, plus c 3 g 2. This is also valid, because if it is smaller than c 1 times g 1 it is smaller than c 3 times c 3 is even bigger than c 1, same to c 2. So, if I take that c 3, and then I collapse it, I get c 3 times g 1 plus g 2. So, I am just doing some simple algebraic manipulation here. So, I have taken c 3 g 1 plus c 3 g 2 and collapsed it into c 3 times g 1 plus g 2.

Now, I have g 1 plus g 2. So, in general, if I have some 2 values, x and y, if I take the bigger of the 2, let us assume that y is the bigger of the 2, then I will have y plus y. So, I claim that x plus y is always going to be smaller than 2 times the maximum of extent x and y. Because instead of the smaller value, I put the maximum instead of the bigger value it is itself the maximum, so that is 2 times the maximum. And what I started with is smaller than that.

So, therefore this is going to be less than 2 times. So, c 3 times this is there and I am going to replace this by 2 times the maximum. So, now with the new constants 2 times c 3 and n 3, I have shown that f 1 plus f 2 is big O of max of g 1 g 2.

(Refer Slide Time: 14:21)





So, going back to our earlier discussion about that sim card, Aadhaar card case, the importance of this is now when I have these algorithms which take multiple phases, and I analyze each phase at a time. So, I tell you that the first phase takes gA of a phase A takes order of gA time and phase B takes order of gB time. From this I can conclude that the algorithm as a whole takes a maximum of order of gA and gB. So, in practice what this means is that the each individual block contributes some time and the largest of these blocks dominates the overall running time of the algorithm.

I do not have to add them up and get some new number. It is the maximum of the running time of each block or each phase. So, the least efficient phase is the bottle neck. So, this is what we are saying if I have to do a sequence of steps and for each step, I have calculated this asymptotic complexity individually, then to get an estimate for the whole algorithm, I can take these steps and find the worst step worst section and say that dominates.

So, that makes it much easier to analyze algorithms, because I can now break up the algorithm into blocks. And I can say, this block takes so much time. And after this block ends, it goes to this block. So, this block takes so much time, so it is f 1 plus f 2, but then the maximum of these 2 will dominate the running time.

So, this is for upper bounds. Sometimes we might want the other way, we want to say that a function is above another function. So, this is called a lower bound. So, earlier, we said fx is below gx. Now, we want to say fx is above gx. So, this is written using this Greek letter omega

capital omega in particular. And so similar to the other one you want to find these constants, but here, instead of saying that f of x is smaller than or equal to g of x, you are saying f of x is bigger than or equal to g of x.

So, this says that now, g of x is a lower bound. It saying that f of x will never go after a certain point, if I know something of g of x, I know f of x is always above. So, it is going to take at least so much time, if I can say that is what you say, for a lower bound. It cannot take less than so much time. So, g of x is some known quantity, you will say f of x, I do not know what it is, but it is O big omega of g of x, it cannot take less time than g of x.

So, here is an example you can say that, remember, we said that n cubed cannot be big O of n squared, but n cubed is omega of n squared, because simply put n cubed is bigger than n squared for all n. So, if I just take trivially my horizontal limit to be n equal to 1. And my constant factor to be 1, I get that for every n bigger than 1, n cubed is bigger than n squared and so n cubed is n squared is a lower bound for n cubed.

Now, this is not something that we typically establish for an algorithm individually. We do not say that this algorithm is a lower bound for that algorithm or something like that. Rather, what we say is that, for a problem as a whole, I must take at least too many steps. So, this is a lower bound for how many steps of work this problem requires to be solved.

For example, If I take an unsorted sequence, and I asked you to search for a value, which is not there. I claim that no matter how you do it, whether you scan it from left to right, right to left. Whatever way you do it, if you do not have any particular structure to the order of the values, there is no way to determine that the value is not there without examining every value.

So, I can then argue that searching for an element and an unsorted list mean this is what I gave you is an informal argument. But you can formalize this day, that searching for a value an informal in value which does not exist in an unsorted list is going to have a lower bound of the size of the list. Unless I look at every element, I cannot conclusively declare that this value is not there.

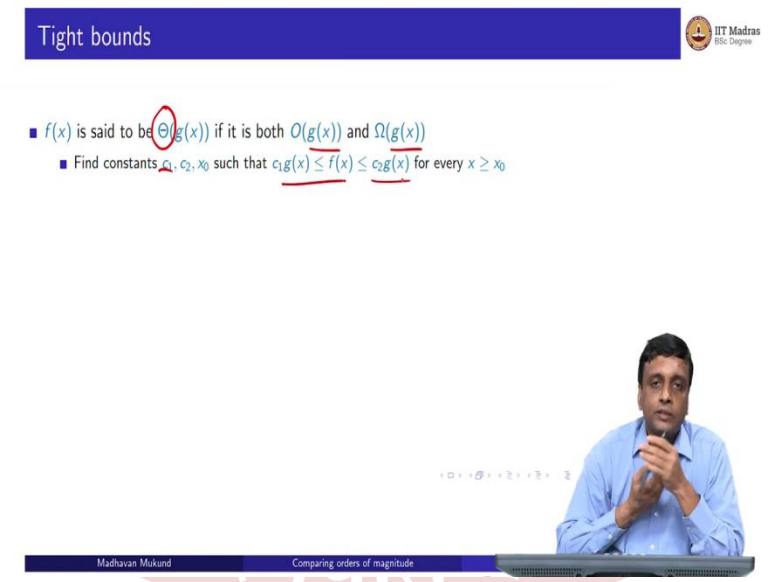
Similar example happens in sorting. Now, in most sorting algorithms that we will see, the way that you saw it is you compare 2 elements in an array and you want to put it in a particular

ascending order, you want the first element to be smaller than the second element. And if it is not, you exchange them. So, this compare and swap, so you compare 2 elements of what you are allowed to do to sort an array is usually to compare elements, and to rearrange them.

So, in this model, where you are comparing and rearranging, it turns out that you need to compare at least n log n times. So, this is a problem, this is a lower bound for sorting as a problem. So, it says that no matter how clever your sorting algorithm is, you are not going to be able to use this compare and swap type of algorithm and get away with less than n log n comparisons, because that is a lower bound.

So, independent of whatever algorithm we choose. So, the same thing with the earlier thing about searching for a value, which is not present in a list, independent of how you scan an unsorted list, whatever strategy, you look at all the odd positions and all the even positions, you look at left to right, right to left, you start in the middle and go forwards and backwards. All of these are going to have to force you to see the entire list. So, you cannot do better than the order n time.

(Refer Slide Time: 19:27)

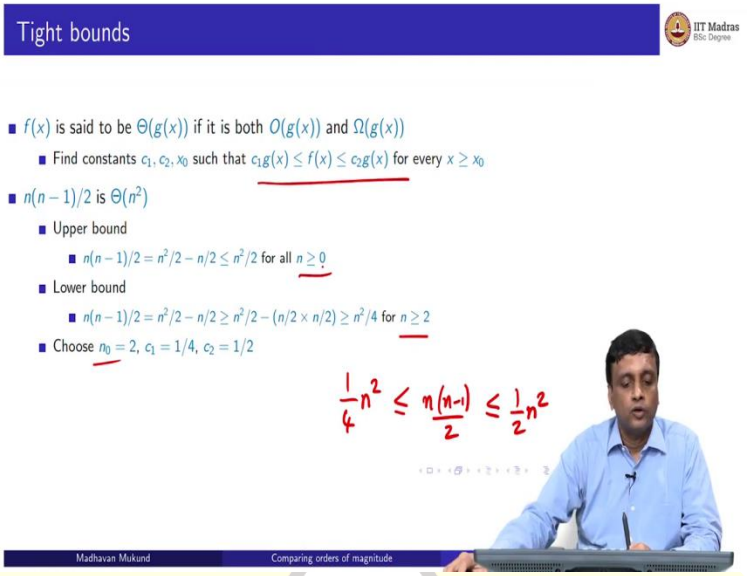


Tight bounds

- $f(x)$ is said to be $\Theta(g(x))$ if it is both $O(g(x))$ and $\Omega(g(x))$
  - Find constants c_1, c_2, x_0 such that $c_1 g(x) \leq f(x) \leq c_2 g(x)$ for every $x \geq x_0$
- $n(n-1)/2$ is $\Theta(n^2)$
  - Upper bound
    - $n(n-1)/2 = n^2/2 - n/2 \leq n^2/2$ for all $n \geq 0$

Tight bounds

- $f(x)$ is said to be $\Theta(g(x))$ if it is both $O(g(x))$ and $\Omega(g(x))$
  - Find constants c_1, c_2, x_0 such that $c_1 g(x) \leq f(x) \leq c_2 g(x)$ for every $x \geq x_0$
- $n(n-1)/2$ is $\Theta(n^2)$
  - Upper bound
    - $n(n-1)/2 = n^2/2 - n/2 \leq n^2/2$ for all $n \geq 0$
  - Lower bound
    - $n(n-1)/2 = n^2/2 - n/2 \geq n^2/2 - (n/2 \times n/2) \geq n^2/4$ for $n \geq 2$



Now, in some rare situations, we are able to show that there is a lower bound for the algorithm that I am trying to solve into for the problem I am trying to solve. And I have an algorithm whose upper bound is the same as that lower bound. So, this is what is called a tight bound, and it is written theta. So, as you would expect, what you want is the same function like you want to say that I have the best possible sorting algorithm.

So, if I argued that n log n is a lower bound for the number of comparisons, and I do have an actual algorithm which performs in order n log n comparisons, then I have something which is effectively the best I can do. Because it does as many comparisons as is required and no more. So, you want the same function in this case n log n, to be both an upper bound and the lower bound.

So, you want to say that there is a constant such that f is bigger than c 1 times g. And there is another constant such as f is smaller than c 2 times g. In which case now, f is sandwiched, it is essentially the same as g. So, here is a simple example. So, we want to claim that n into n minus 1 by 2 is theta of n squared. So, we have to first show that there is an upper bound and then we have to show this lower bound. So, here is the upper bound.

So, I just take n into n minus 1 by 2 and I expand it, so I get n squared by 2 minus n by 2. Now, because n is positive, I can this am subtracting some positive quantity rates in the worst case and subtracting 0, but when becomes 1, I am subtracting half and becomes 2 and subtracting 1. So, I am subtracting something positive from n squared by 2.

So, if I do not subtract, I get something bigger. n squared by 2 minus n by 2 must be smaller than equal to n squared by 2. So, I am now claiming I am looking for something with respect to n squared. So, I claim that this factor half is my multiplier that I want. So, for all n, n into n minus 2 is going to be upper bounded by half times n squared.

Conversely, if I do the lower bound, I do the same expansion. And now I say, instead of subtracting this I subtract more so, I subtract n by 2 times n by 2. n by 2 times n by 2, if n by 2 is a fraction, when you square it becomes smaller, half squared is 1 by 4. 1 by 4 squared is 1 by 16. So, a number smaller than 1, when you square them make them even smaller. But numbers bigger than 1 become bigger.

So, long as this quantity is bigger than 1, n squared, n by 2 whole squared is going to be bigger than n squared. So, this is a same thing as saying that n is bigger than or equal to 2. So, for n bigger than equal to 2, if I subtract n by 2 times n by 2, what I get to something. Where I am subtracting more than I am subtracting here. So, this quantity is bigger than this quantity.

So, this is what I need for an upper bound. But what is this quantity, this is n squared by 2 minus n squared by 4, which is equal to n squared by 4. So, we are saying that n squared by 2 minus n by 2 is bigger than 1 by 4 n squared. So, I have another constant now 1 by 4. So, now with these 2 constants, I have that 1 by 4 of n times n squared is smaller than n into n minus 1 by 2 is smaller than half n squared. So, this is the form in which I want this thing to show that this is theta. And what is the value it because in 1 case, I need 2 in 1 case, I need zero. If I choose n to be bigger than 2, both of these inequalities will hold.

(Refer Slide Time: 22:55)

So, to summarize, we have saw these 3 notations to compare function, so big O is an upper bound notation. So, f is big O of g, if g is an upper bound for f, so beyond a certain point, some constant times g is always sitting above f. And this is useful to describe upper bounds for worst case running time.

For problems as a whole and rather than individual algorithm, sometimes we want to, we will try to establish lower bounds, you must do at least so much work, no matter how you try to solve it. So, it is not very useful to establish a lower bound for an algorithm individually, but for a problem as a whole. And this is given using omega, we just invert the condition you say beyond a certain point, f is always bigger than c times g.

And finally, you might have a situation where you have a matching upper and lower bound, and this is a tight bound. And if it so happens that you have an algorithm whose upper bound matches the known lower bound for the problem you are trying to solve, then you are in good shape because you are found an optimal algorithm for that problem.