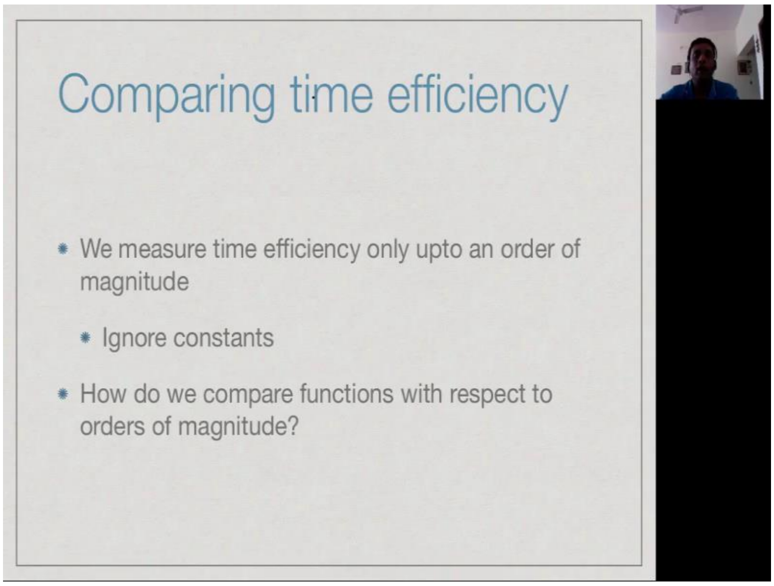
Design and Analysis of Algorithms
Prof. Madhavan Mukund
Chennai Mathematical Institute

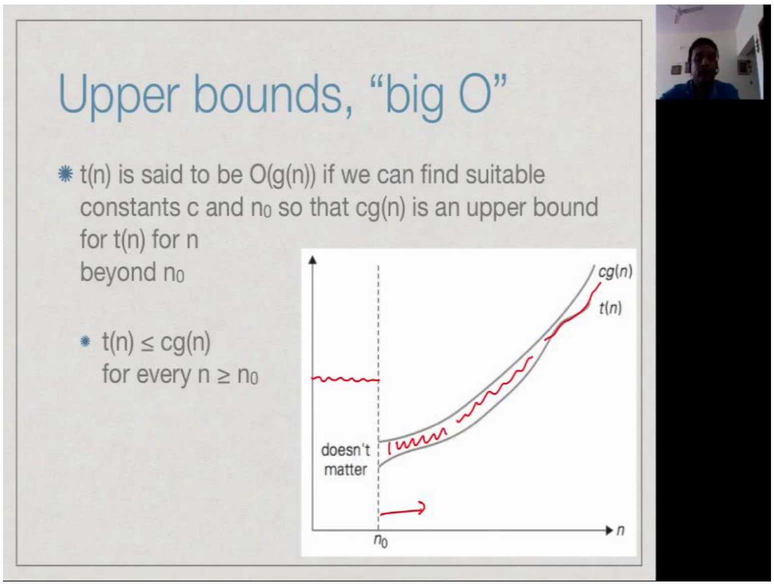
Week - 01
Module – 07
Lecture - 07

(Refer Slide Time: 00:02)



So, we have said that we will measure the time efficiency of algorithms only upto an order of magnitude. So, we will express the running time as a function t of n of the input size n, but we will ignore constant. So, we where only said that t of n is propositional to n square or n log n or 2 to the n . So, now, the next step is to have an effective way of comparing is, running times across algorithms. If i know the order of magnitude of 1 algorithm, and the order of magnitude of another algorithm how do I compare?

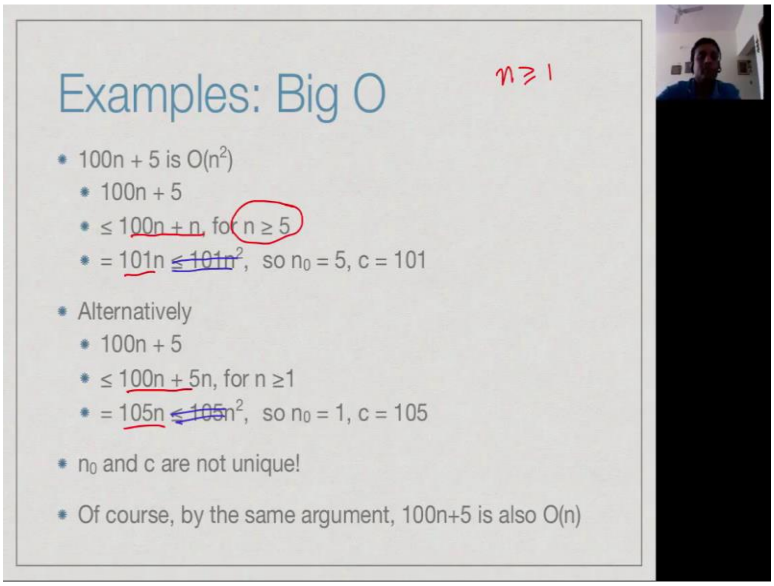
(Refer Slide Time: 00:34)



So, the notation we need or the concept we need is that of an upper bound which is given by the notation big O. So, we say that a function g of n is an upper bound for another function t of n if beyond some point g of n dominates t of n. Now, remember that g of n is going to be now a function which is an order a magnitude. So, we are thrown away all the constant factors which we play a role in g of n. So, we allow ourselves this constant. So, we say that it is not g of n al1 which dominates t of n, but g of n times some constants.

So, there is a fixed constants c and beyond some limits. So, there is a initial portion where we do not care, but beyond this limit we have that t of n always lies below c times g of n. In this case c times g of n and is upper bound for t of n and we say that t of n is big O of g of n.

(Refer Slide Time: 01:34)

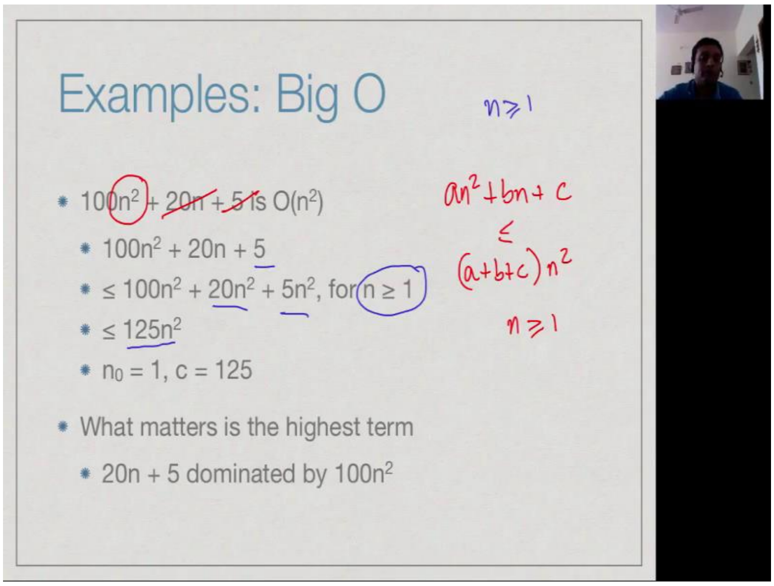


So, let us look at an example. So, supposing we have this function t of n is 100 and plus5 then, we claim that it is big O of n square now remember that n is suppose to be the input size. So, the input size to a problem is always going to be at least 1, there is no problem that needs to be solve if you are input to zero and certainly we cannot have negative. So, we are always having in mind the situation that, n is bigger than or equal to 1. So, if we now start with our function 100 n plus 5 then, if you choose n to be bigger than 5 then n will be bigger than this value. So, we can say 100 n plus 5 is smaller than equal to 100 n plus n.

And now we can collapse this is 101 right. So, 100 n plus 5 is smaller than 101 provided n is bigger than to 5, now, since n is at least 1 n square is bigger than n. So, 101 times n is going to be smaller than 100 and 1 n square. So, by choosing n 0 to be 5 and c to be 101 we have established that n square it is an upper bound to 100 n plus 5. So, 100 n plus 5 is big o the n square. Now, we can do this using a slightly different calculations, we can say that 100 n plus 5 is smaller than 100 n plus 5 n for n bigger than 1 because n is at least 1. So, 5 times n is going to be at least 5. So, now, if you collapse is we get 105 n now, but the same logic 105 n the smaller than 105 n square whenever n is bigger than 1.

So, new way of establishing the same fact, where we have chosen n 0 equal to 1 and c equal to 105 right. So, n 0 and c or not unique right depending on how we do the calculation in we might find different n 0 and different c. But it does not matter how we choose them so, long as we can establish the fact that beyond a certain n 0 there is a uniform constant c such that c times g of n dominates t of n. Notice that the same calculation can give us a tighter upper bound, this is kind of a loose upper bound you would expect the 100 n is smaller than n square. But we can also say that this is big O of n, why is that? Because if you just top the calculation at this point we do not come to this stage at all you have establish that 100 n plus 5 is less equal to 101. But, the same values n 0 equal to 5 and c equal to 101 this also tells us that 100 n plus 5 big O. Likewise at this point if you just ignore this step then we say that 100 n plus 5 is smaller than 105 n. So, for n 0 equal to 1 and c equal to 105 you have established this 1.

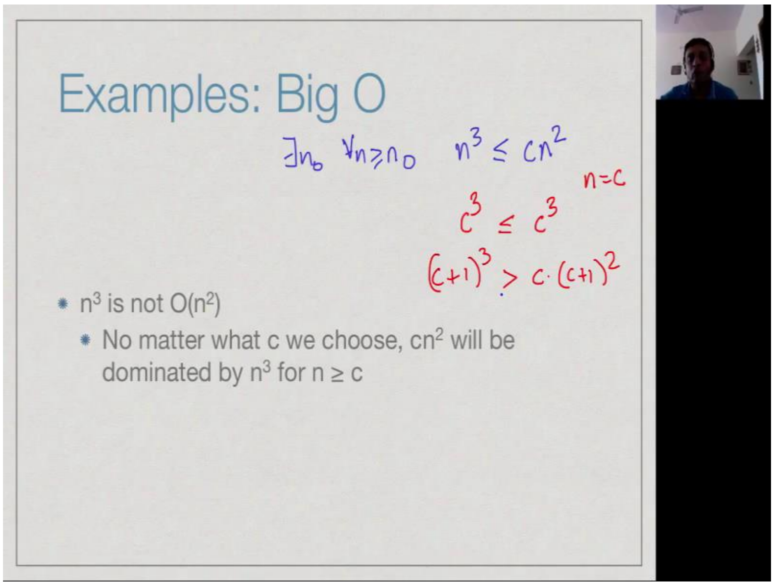
(Refer Slide Time: 04:15)



Let us look at us another example supposing we look at 100 n square plus 20 n plus 5. Now, again assuming that n is bigger than 1 we know that we can multiply by n and do not get n is smaller. So, 20 n will be dominated it 20 n square right and 5 will be dominate the 5 times n times n 5 n square. So, I now have 100 n square plus 20 n square plus 5 n square, is bigger than my original function 100 n square plus 20 n plus 5. So, I combine these, I get 125 n square and now all I have assumed is that, n is bigger than equal to 1. So, for n 0 equal to 1 and c equal to 125 we have that n square dominates 100 n square plus 20 n plus 1. So, you can easily see that, in general if I have a n square plus b n plus c right this is going to be dominated by a plus, b plus, c times n square right. So, this is going to be less than this for all n greater than equal to 1. So, we can generally speaking, take a function like this and ignore the lower terms because they are dominated by the higher term in just focus on the value with the highest exp1nt.

So, in this case in this whole thing n square is the biggest term therefore, this whole thing this going to be big over the n square. So, this is a very typical shortcut that we cans take, you can just take an expression ignore the coefficients pick the largest exp1nt and choose that to be the big O right

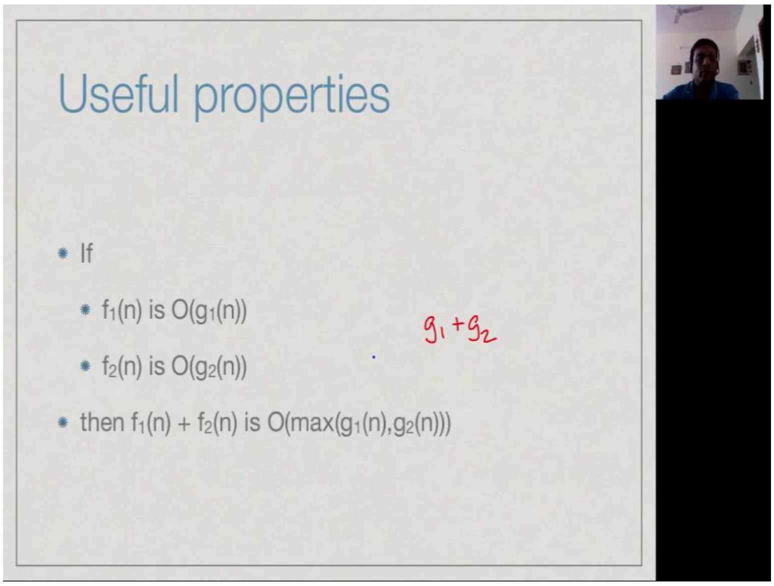
(Refer Slide Time: 05:37)



Now, we can also show that things are not to be good. So, for instances its intuitively clear that, n cube is bigger than n square now, how do we formally show that n cube is not big O of n square. Well, supposing it was, then there is exists some n 0, such that for all n bigger than equal to n 0, n cube must be smaller than are equal to c times n square right. If this works we go up n square this is what we must have. Now supposing, we choose n is equal to c then we have on the left hand side c cube, on the right hand side we have c cube and certainly we have that c cube less than equal to c cube. If i go to c plus 1 I will have c plus 1 whole cube and this side I will have c times c plus 1 whole square and now the problem is, this is bigger because c plus 1 whole cube is bigger than c times c plus 1 whole square.

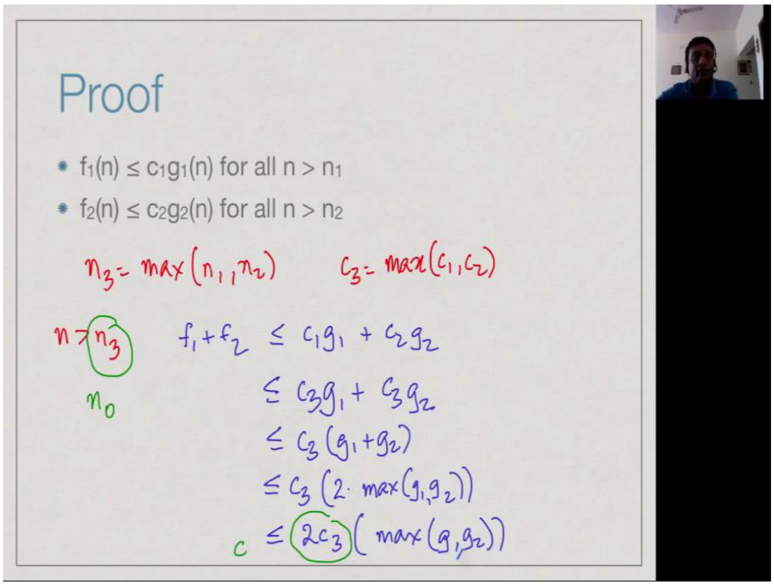
Therefore, no matter what c we choose, if we go to n equal to c we will find that inequality that we want gets flipped around. Therefore, there is no c that we can choose to make n cubes smaller than c n square beyond a certain point and therefore, this is not bigger. So, our intuitive idea that n cube grows faster than n square can be formally proved using this step function.

(Refer Slide Time: 00:07)



Now, here is the useful fact about big O, if I have a function f 1 which is big O of g 1 and another function f 2 which is big O of g 2 then, f 1 plus f 2 is actually dominated by the max of g 1 and g 2. You might think it is g 1 plus g 2 this is the obvious thing that comes to mind looking at this, that f 1 plus f 2 is smaller than g 1 plus g 2, but actually it is max.

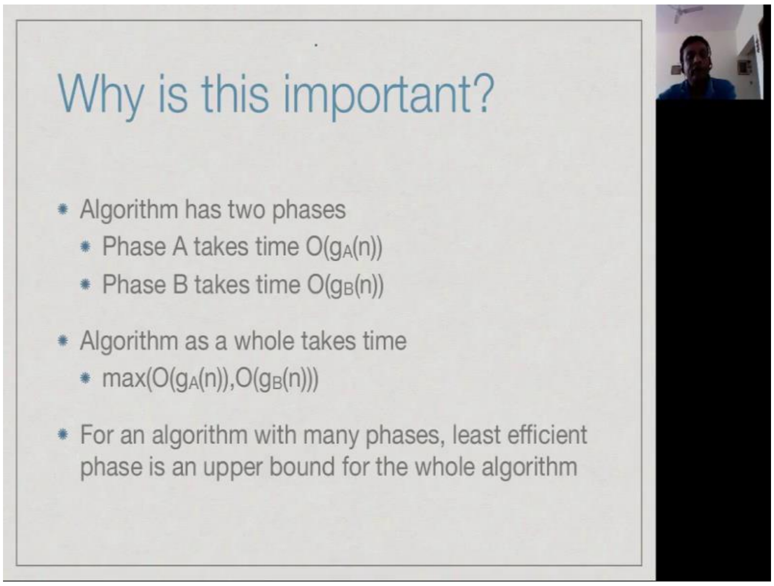
(Refer Slide Time: 07:31)



How do we prove this? Well, is not very difficult. By definition if f 1 is big o of g 1 there exists some n 1 such that beyond n 1 f 1 is dominated by c 1 of g 1 c 1 times g 1. Similarly, if f 2 is big O of g 2 there is an n 2 such that beyond n 2 f 2 is dominated by c 2 times g 2 right. So, now, what we can do is we can choose n 3 to be the maximum of n 1 and n 2, and we can choose c 3 to be the maximum of c 1 and c 2. So, now, let us see what happens beyond n 3, beyond n 3 both these inequalities are effective. So, we have f 1 plus f 2 will be less than c 1 and g 1 plus c 2 times g 2 right. Because, this is beyond both n 1 and n 2 so, both f 1 is less than c 1 and g 1 whole and f 2 less than c 2 g 2 whole. So, I can add the 2 and, this is the first obvious thing that we said is it g 1 plus g 2, but now we can be a little clever we can say there we have c 3. So, c 1 is smaller than c 3 because this is an maximum c 2 a smaller than c 3. So, I can combine these and say that this is less than c 3 g 1 plus c 3 g 2.

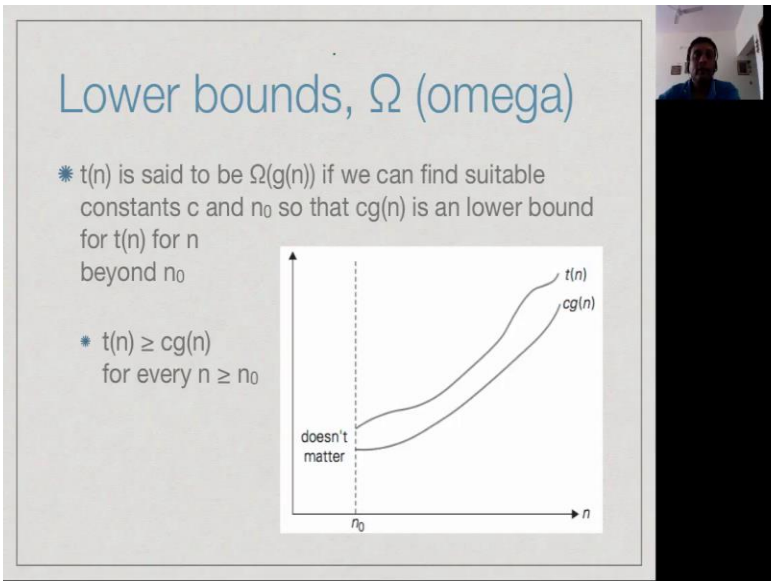
Now, having combined these I can of course, push them together and say this is less then c 3 times g 1 plus g 2. But g 1 plus g 2 if I take the maximum of those then 2 times the maximum will be bigger than that. So, I will get this is less than c 3 times 2 times the maximum of g 1 and g 2 right. I can take this 2 out and say that therefore, this is less then equal to 2 c 3 and max of g 1 and g 2 right. So, now, if I take this as my n 0 and this as my c then I have established that for every n bigger than n 0 and maximum n 1 and n 2 there is a constant which is 2 times the max of c 1 c 2 such that f 1 plus f 2 is dominated by c times max of g 1 g 2. Why this mathematical fact useful to us?

(Refer Slide Time: 09:51)



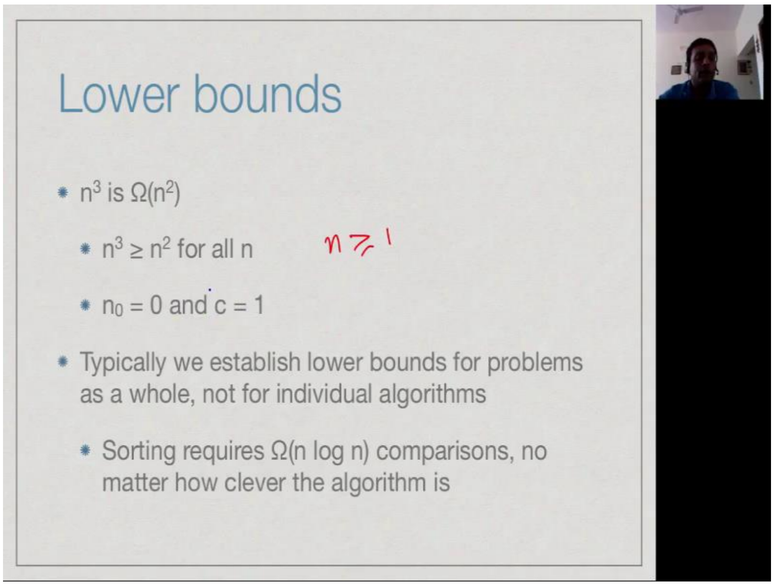
So, very often when we are analyzing an algorithm, it will have different phases. It will do something in one part then it will continue to some other thing and so, on. So, we could have 2 phases, phase a which takes time big O of g a and phase b which takes time big O of g. So, now, what is the good upper bound for the overall running time of the algorithm. So, the instinctive thing would be to say g a plus g b. But what this result tells us is that it is not g a plus g b that is useful for the upper bound, but the maximum of g a and g b right. In other words, when we are analyzing an algorithm it is enough to look at the bottle necks. You go to many steps look at the steps which take the maximum amount of time, focus on those and that will determine the overall running time of the other. So, when we look at a function on algorithm which has a loop, we typically look at the loop how long does the loop go. We ignore, may be the initialization that takes place before the loop or some prints statement that takes place after the loop because that does not contribute as much as the complexity as the loop itself ok. So, when we have multiple phases, it is the most inefficient phase which dominates the overall behavior and this is formalized by the result we just saw.

(Refer Slide Time: 11:01)



Now, there is a symmetric notion to an upper bound namely a lower bound. So, just like we said that t of n is always lying below c f c times g of n. We might say that t of n always lies above c times g of n and this is described using this notation omega. So, this is just a symmetric definition which just says that t of n is omega of g of n, if for every n beyond n 0 t of n lies above c times g of n for some fixed constituency. So, here we have the same thing we have an initial thing that we are not interested in, because at this point nothing can be said. But beyond this n 0 we have that t of n lies above so, t of n is always above c times g of n

(Refer Slide Time: 11:50)

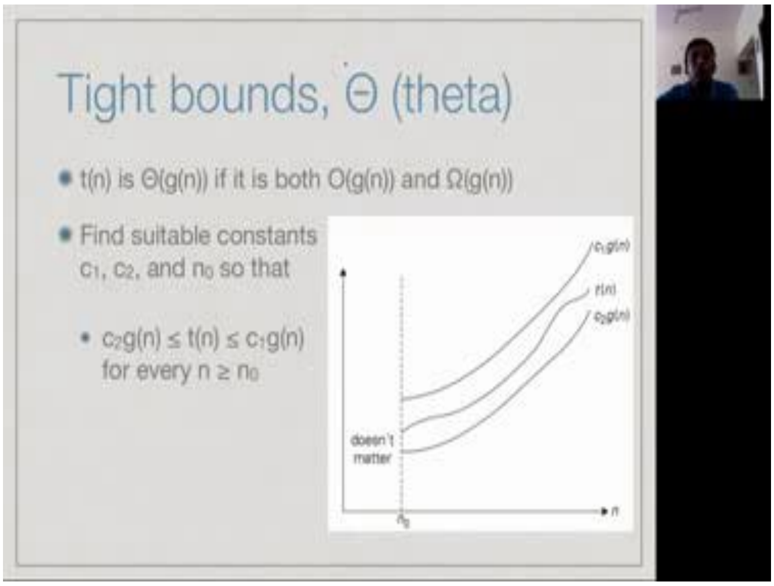


So, we earlier saw that n cubed is not big O of n square, but intuitively n cube should be lying above n square and this is certainly the case because, n cubed is greater than equal to n square for every n bigger than equal to 1 right. So, at n equal to 1 both are 1, but n equal to 2 this will be 8 this will be 4 and so, on. So, if given n 0 equal to 0 or n 0 equal to 1 and c equal to 1 we can establish this. Now of course, when we are establishing an upper bound we are using talking of about the algorithm we have. You are saying this algorithm has an upper bound of so, much and therefore, I can definitely solve the problem with in this much time. Now, when we are talking about lower bounds it is not that useful to talk about a specific algorithm. It is not so, useful to say that this algorithm takes at least so, much time.

What we would like to say something like this problem takes at least so, much time, no matter how you write the algorithm it is going to take at least so, much time. So, typically what we would like to do to make a useful lower bound statement is to say that a problem takes the certain amount of time no matter how you try to solve it. So, the problem has a lower bound rather than the algorithm has a lower bound.

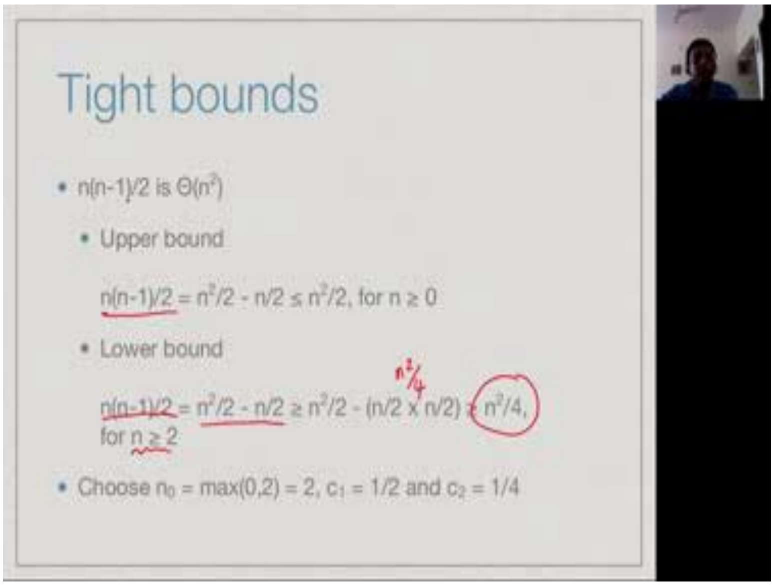
Now, as you might imagine this is the fairly complex into say because what you have to able to show is that no matter how clever you are, no matter how you design an algorithm you cannot do better than a (Refer time 13:13). This is much harder than saying I have a specific way of doing it and I am analyzing how to do that. So, establishing lower bounds is often very tricky. One of the areas where lower bound is have been established is sort. So, it can be shown that, if you are relying on comparing values to sort them then, you must at least do n log n comparisons, no matter how you actually do the sorting. No matter how clever you are sorting algorithm, it cannot be faster than n login in terms of comparing elements but, this hard to do remember, because you really show this independent of the algorithm.

(Refer Slide Time: 13:51)



Now, we could have a nice situation where we have matching upper and lower bounds. So, we say that t is theta of g of n if it is both, we go g of n and omega of g of n. In other words, in suitable constants t of n can be dominated by g of n, and it also lies above g of n for two different constants of course. So, what this really means is that, t of n and g of n are basically are same order of the magnitude, they are essentially the same function therefore, you have reached a kind of optimum value.

(Refer Slide Time: 14:26)

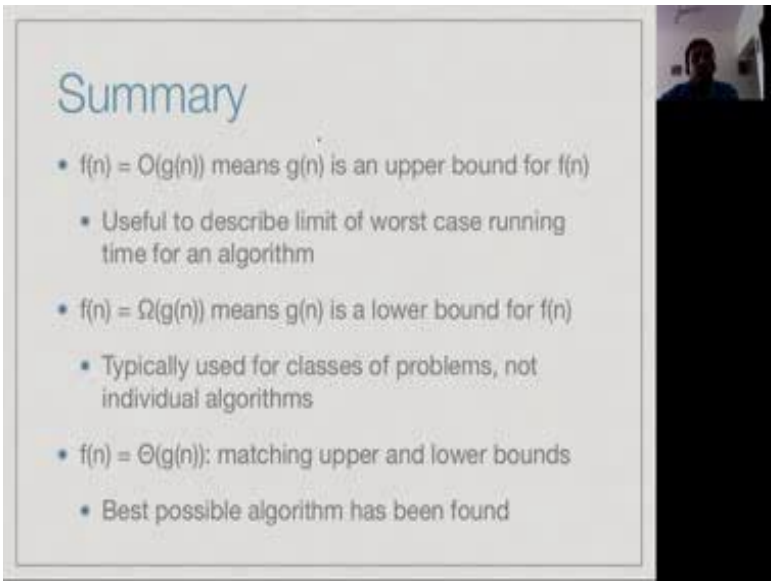


So, as an example we can say since that n into n minus 1 by 2 is theta of s square. In order to prove something like this, we have to show that there is an upper bound, that is we can find a constants (Refer time: 14:37) dominates this and the lower bound. There is another constant (Refer time: 14:41) is below this. So, for the upper bound we just expand our n into n minus 1 by 2. So, we get n squared by 2 in the first term and n minus n by 2. Now, since it is upper bound n squared by 2 minus n by 2, if I ignore n by 2, this is going to be less than n squared by 2.

Therefore, now I have an upper bound saying that, the constant half this is dominated by n square for n bigger than 0. On the other hand, if I want to do a lower bound then, I will say same thing I will expand out to n into n minus 2, I will get same expansion. And now I will want to lower bound, so, now what I will do is I will make this even smaller. I will say that I subtract not n by 2 but n by 2 times n by 2. So, this will be bigger than this, because I am subtracting more. N squared by 2 minus n by 2 will be bigger than n squared by 2 minus n by 2, but this is again n squared by 4. So, I have n squared by 2 minus n squared by 4, so this simplifies to n squared by 4. In other words, I have shown that n into n minus 1 by 2 is bigger than equal to n squared 4. But now, in order to justify this, to justify that n by 2 is increasing, n must be at least 2. Because if n smaller than 2 this is the fraction so I am actually reducing.

Here, I have different n greater than equal to 2. I have established a lower bound result that for n bigger than equal to 2 n into n minus 1 by 2 is above one fourth of n square. So, therefore now if we chose our constant to be 2 for all values bigger than 2, I have that n into n minus 1 by 2 is less than half of n square and n into n minus 2 is bigger than one fourth of n square. So, I have found this matching upper and lower bound which shows that n into n minus 1 theta of n square.

(Refer Slide Time: 16:44)



So, to summarize when we use big O, we have discovered an upper bound. If we say f of n is big O of g of n, it means that g of n dominates f of n so f of n is no bigger than g of n. And this is useful to describe the limit of the worst case running time. So, we can say that worst running time is upper bounded by g of n. On the other hand, if we use omega we are saying that f of n is at least g of n, g of n is lower bound (Refer time: 17:14).

As we described this is more useful for problems as a whole, sorting as a general problem rather than for individual algorithm. Because it tells us no matter how you do something, you will have to spend at least that much time, but this hard to establish. And if you have a situation where a lower bound has been established for a problem and you find an algorithm which achieves the same bound as an upper bound then, you have found in some sense the best possible algorithm. Because, you cannot do any better than g of n because we have a lower bound of g of n and you have achieved g of n because you have shown your algorithm as big O of g of n. So, theta is a way of demonstrating that you have found an algorithm which is asymptotically as efficient as possible.