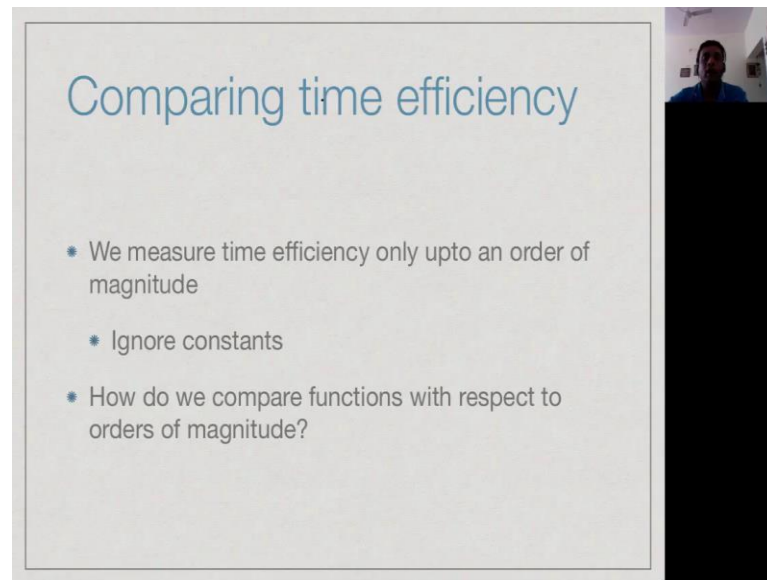


Design and Analysis of Algorithms
Prof. Madhavan Mukund
Chennai Mathematical Institute

Week - 01
Module – 07
Lecture - 07

(Refer Slide Time: 00:02)

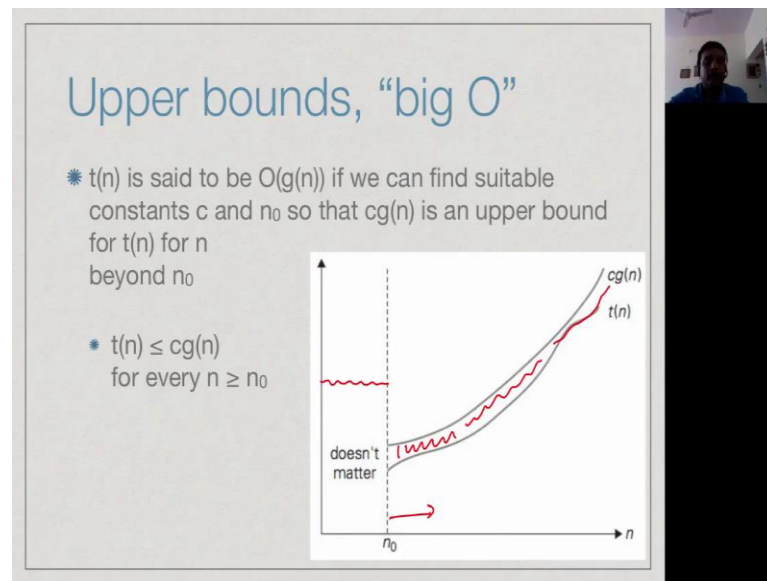


Comparing time efficiency

- We measure time efficiency only upto an order of magnitude
- Ignore constants
- How do we compare functions with respect to orders of magnitude?

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 all 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)

Examples: Big O

$n \geq 1$

- $100n + 5$ is $O(n^2)$
 - $100n + 5$
 - $\leq 100n + n$ for $n \geq 5$
 - $= 101n \leq 101n^2$, so $n_0 = 5$, $c = 101$
- Alternatively
 - $100n + 5$
 - $\leq 100n + 5n$, for $n \geq 1$
 - $= 105n \leq 105n^2$, so $n_0 = 1$, $c = 105$
- n_0 and c are not unique!
- Of course, by the same argument, $100n+5$ is also $O(n)$

So, let us look at an example. So, supposing we have this function t of n is 100 and plus 5 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 $100n$ plus 5 then, if you choose n to be bigger than 5 then n will be bigger than this value. So, we can say $100n$ plus 5 is smaller than equal to $100n$ plus n .

And now we can collapse this is 101 right. So, $100n$ 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 $100n$ plus 5. So, $100n$ plus 5 is big o the n square. Now, we can do this using a slightly different calculations, we can say that $100n$ plus 5 is smaller than $100n$ plus $5n$ 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 $100n$ 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 $100n$ 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 $100n$ plus 5 big O. Likewise at this point if you just ignore this step then we say that $100n$ plus 5 is smaller than $105n$. So, for n_0 equal to 1 and c equal to 105 you have established this 1.

(Refer Slide Time: 04:15)

Examples: Big O

$n \geq 1$

- $100n^2 + 20n + 5$ is $O(n^2)$
- $100n^2 + 20n + 5$
- $\leq 100n^2 + 20n^2 + 5n^2$, for $n \geq 1$
- $\leq 125n^2$
- $n_0 = 1, c = 125$
- What matters is the highest term
- $20n + 5$ dominated by $100n^2$

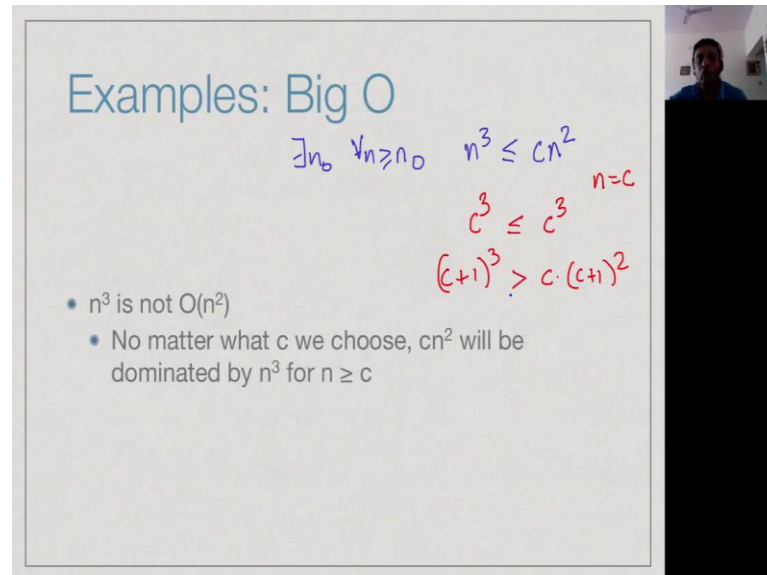
Handwritten in red: $an^2 + bn + c$, \leq , $(a+b+c)n^2$, $n \geq 1$

Let us look at us another example supposing we look at $100n$ square plus $20n$ 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, $20n$ will be dominated it $20n$ square right and 5 will be dominate the 5 times n times n $5n$ square. So, I now have $100n$ square plus $20n$ square plus $5n$ square, is bigger than my original function $100n$ square plus $20n$ plus 5 . So, I combine these, I get $125n$ 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 $100n$ square plus $20n$ plus 1 . So, you can easily see that, in general if I have a n square plus b 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 exponent.

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 exponent and choose that to be the big O right

(Refer Slide Time: 05:37)



The slide is titled "Examples: Big O" in blue text. It contains handwritten notes in blue and red ink. The blue notes include the definition $\exists n_0 \forall n \geq n_0, n^3 \leq cn^2$. The red notes include $n=c$, $c^3 \leq c^3$, and $(c+1)^3 > c \cdot (c+1)^2$. Below these, there is a bulleted list in blue ink.

Examples: Big O

$\exists n_0 \forall n \geq n_0, n^3 \leq cn^2$

$n=c$

$c^3 \leq c^3$

$(c+1)^3 > c \cdot (c+1)^2$

- n^3 is not $O(n^2)$
- No matter what c we choose, cn^2 will be dominated by n^3 for $n \geq c$

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 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)

Useful properties

- If
 - $f_1(n)$ is $O(g_1(n))$
 - $f_2(n)$ is $O(g_2(n))$
 - then $f_1(n) + f_2(n)$ is $O(\max(g_1(n), g_2(n)))$

Handwritten red notes: $g_1 + g_2$ and a dot.

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)

Proof

- $f_1(n) \leq c_1 g_1(n)$ for all $n > n_1$
- $f_2(n) \leq c_2 g_2(n)$ for all $n > n_2$

Handwritten red notes: $n_3 = \max(n_1, n_2)$ and $c_3 = \max(c_1, c_2)$

Handwritten green notes: $n > n_3$ (circled), n_0 , and c

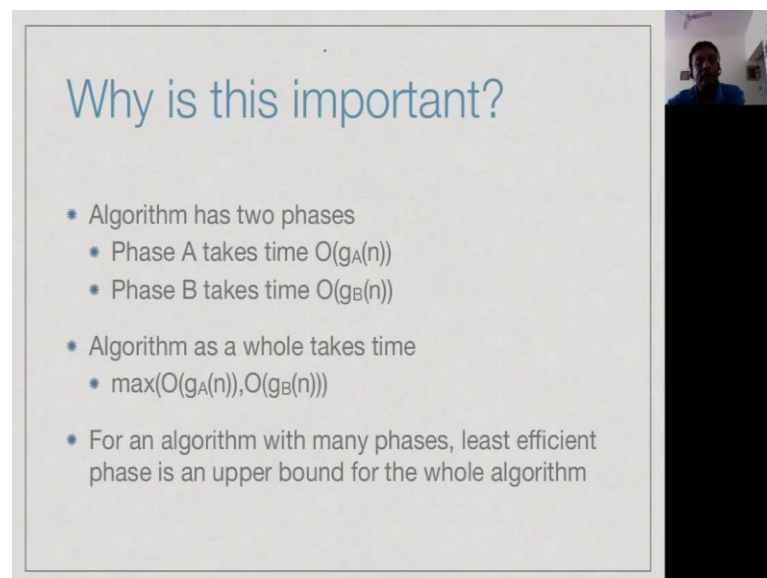
$$\begin{aligned} f_1 + f_2 &\leq c_1 g_1 + c_2 g_2 \\ &\leq c_3 g_1 + c_3 g_2 \\ &\leq c_3 (g_1 + g_2) \\ &\leq c_3 (2 \cdot \max(g_1, g_2)) \\ &\leq 2c_3 (\max(g_1, g_2)) \end{aligned}$$

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 than 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 than 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)



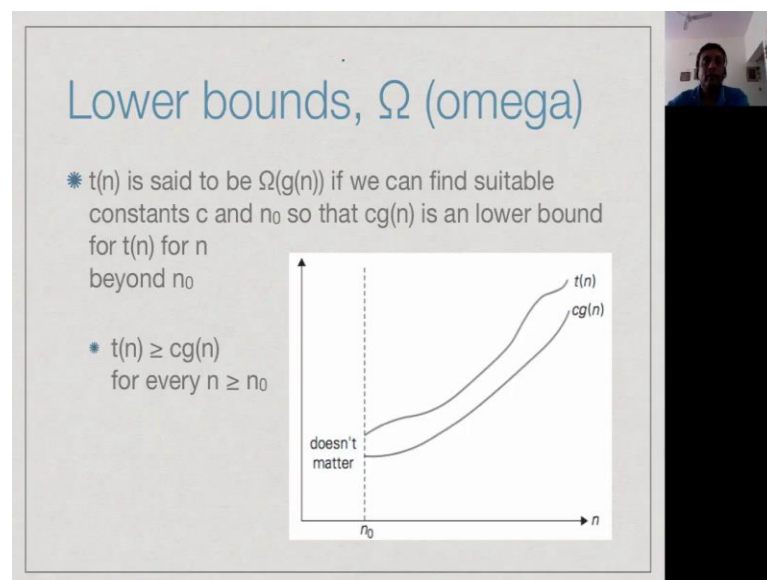
Why is this important?

- Algorithm has two phases
 - Phase A takes time $O(g_A(n))$
 - Phase B takes time $O(g_B(n))$
- Algorithm as a whole takes time
 - $\max(O(g_A(n)), O(g_B(n)))$
- For an algorithm with many phases, least efficient phase is an upper bound for the whole algorithm

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_b . 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

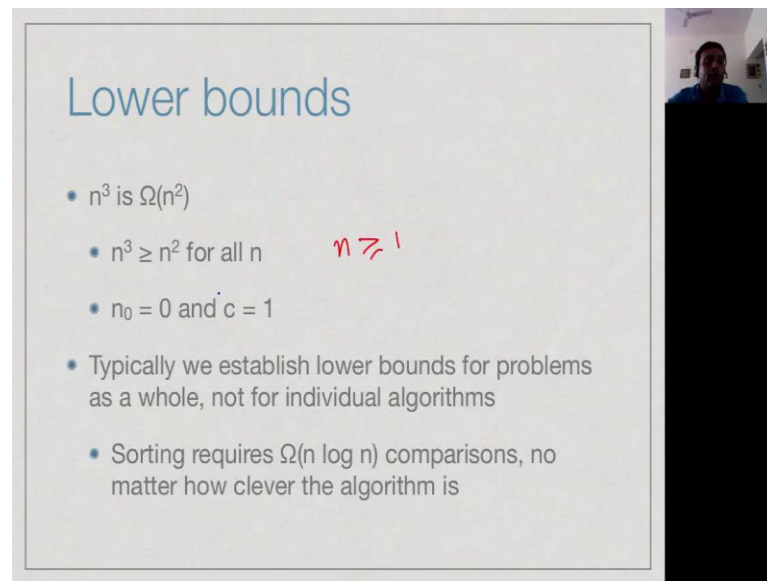
and g b right. In other words, when we are analyzing an algorithm it is enough to look at the bottle necks. You go to many steps look at the steps which take the maximum amount of time, focus on those and that will determine the overall running time of the other. So, when we look at a function on algorithm 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 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)



Lower bounds

- n^3 is $\Omega(n^2)$
- $n^3 \geq n^2$ for all n $n \geq 1$
- $n_0 = 0$ and $c = 1$
- Typically we establish lower bounds for problems as a whole, not for individual algorithms
- Sorting requires $\Omega(n \log n)$ comparisons, no matter how clever the algorithm is

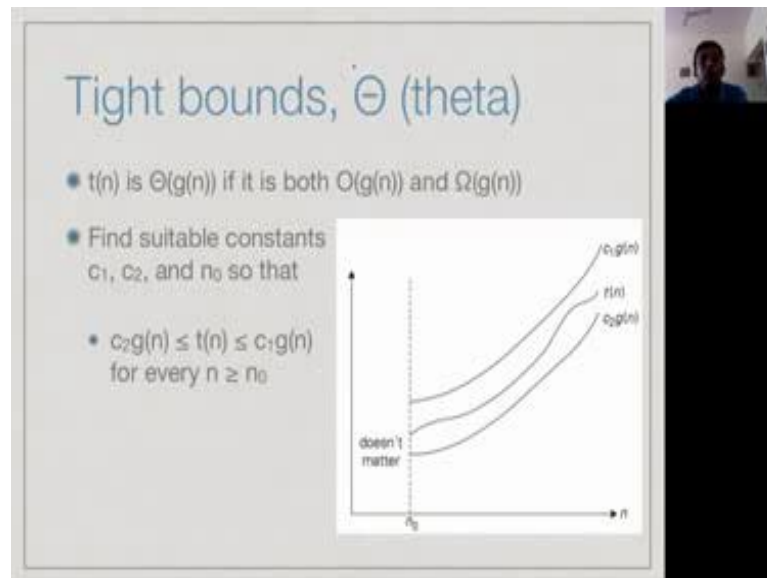
So, we earlier saw that n cubed is not big O of n square, but intuitively n cube should be lying above n square and this is certainly the case because, n cubed is greater than equal to n square for every n bigger than equal to 1 right. So, at n equal to 1 both are 1, but n equal to 2 this will be 8 this 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 your sorting algorithm, it cannot be faster than $n \log n$ in terms of comparing elements but, this is hard to 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 Θ of g of n if it is both, we go O of g of n and Ω 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 the 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)

Tight bounds

- $n(n-1)/2$ is $\Theta(n^2)$
- Upper bound
$$\underline{n(n-1)/2} = n^2/2 - n/2 \leq n^2/2, \text{ for } n \geq 0$$
- Lower bound
$$\underline{n(n-1)/2} = n^2/2 - n/2 \geq n^2/2 - (n/2 \times n/2) \geq \underline{n^2/4}, \text{ for } n \geq 2$$
- Choose $n_0 = \max(0, 2) = 2$, $c_1 = 1/2$ and $c_2 = 1/4$

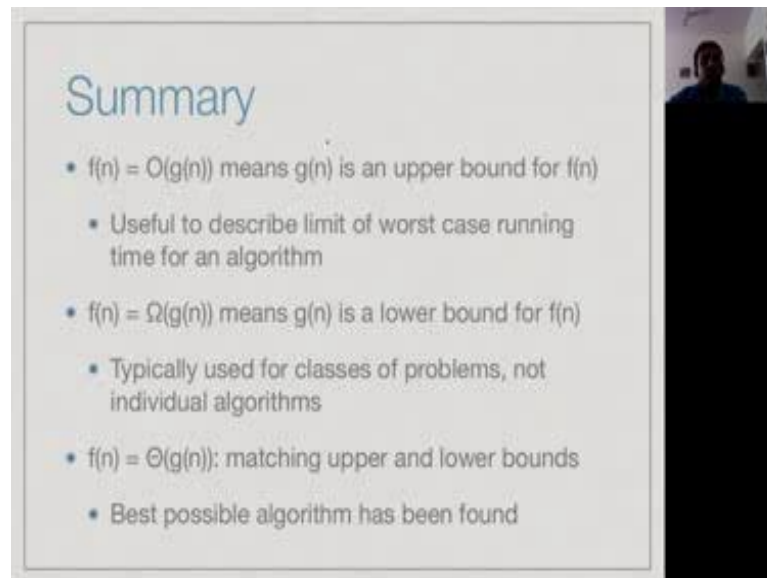
So, as an example we can say since that n into n minus 1 by 2 is theta of n squared. In order to prove something like this, we have to show that there is an upper bound, that is we can find a constant (Refer time: 14:37) dominates this and the lower bound. There is another constant (Refer time: 14:41) 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 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

$n^2 - n$ is less than half of n^2 and $n^2 - 2n$ is bigger than one fourth of n^2 . So, I have found this matching upper and lower bound which shows that $n^2 - n$ is $\Theta(n^2)$.

(Refer Slide Time: 16:44)



Summary

- $f(n) = O(g(n))$ means $g(n)$ is an upper bound for $f(n)$
 - Useful to describe limit of worst case running time for an algorithm
- $f(n) = \Omega(g(n))$ means $g(n)$ is a lower bound for $f(n)$
 - Typically used for classes of problems, not individual algorithms
- $f(n) = \Theta(g(n))$: matching upper and lower bounds
 - Best possible algorithm has been found

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.