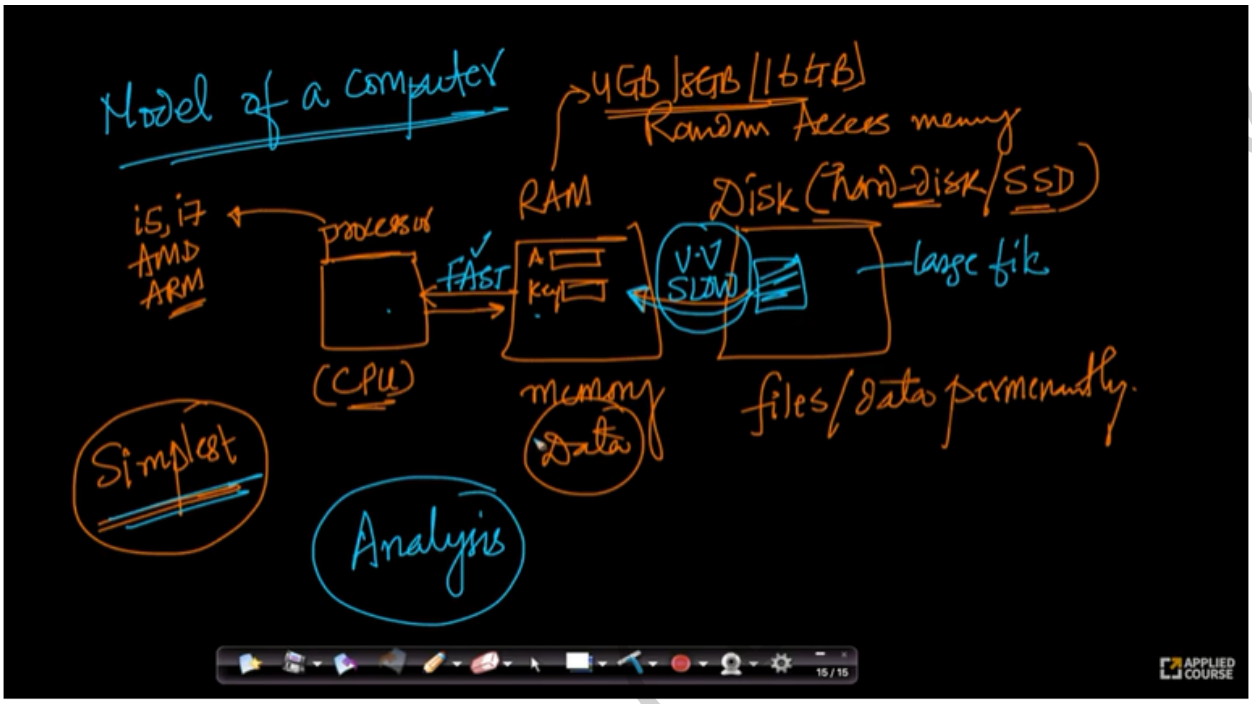
9.1 Model of Computation



Timestamp: 7:38

For analysis of the standard algorithms and datastructures we will use the above simple model of the computer which contains the processor(also called cpu), the RAM and the Disk.

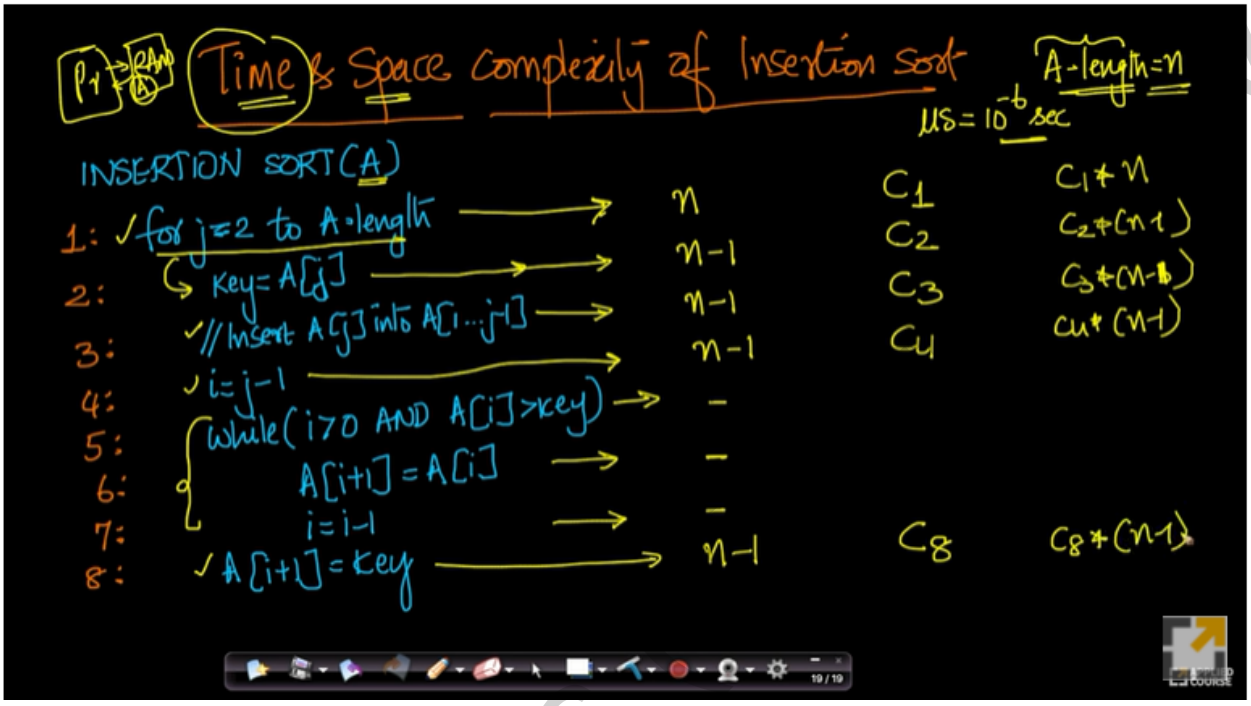
Processor: This is the electrical component responsible for all computations that take place in the computer. It gets the data required for the computation from the RAM and writes the output of the computation to the RAM.

RAM: RAM is a temporary storage device also called memory that stores the data to feed into the processor and collects the output data from the processor. It gets the data from the permanent storage i.e disk and if required can write data to the disk. The data inside the RAM is lost when the computer is shut down.

Disk: Disk is a permanent storage that stores all the data. Whenever this data is required by the processor it supplies that data to the RAM which in turn supplies to the processor.

There are other models of computer involving cache(an on-chip memory for storing values that are mostly used), multi-core cpus (capable of parallel computing),etc. But as for the discussion of standard algorithms and datastructures we only use the above simple model of the computer.

9.2 Space and Time Analysis of Insertion Sort-1



Timestamp: 6:39

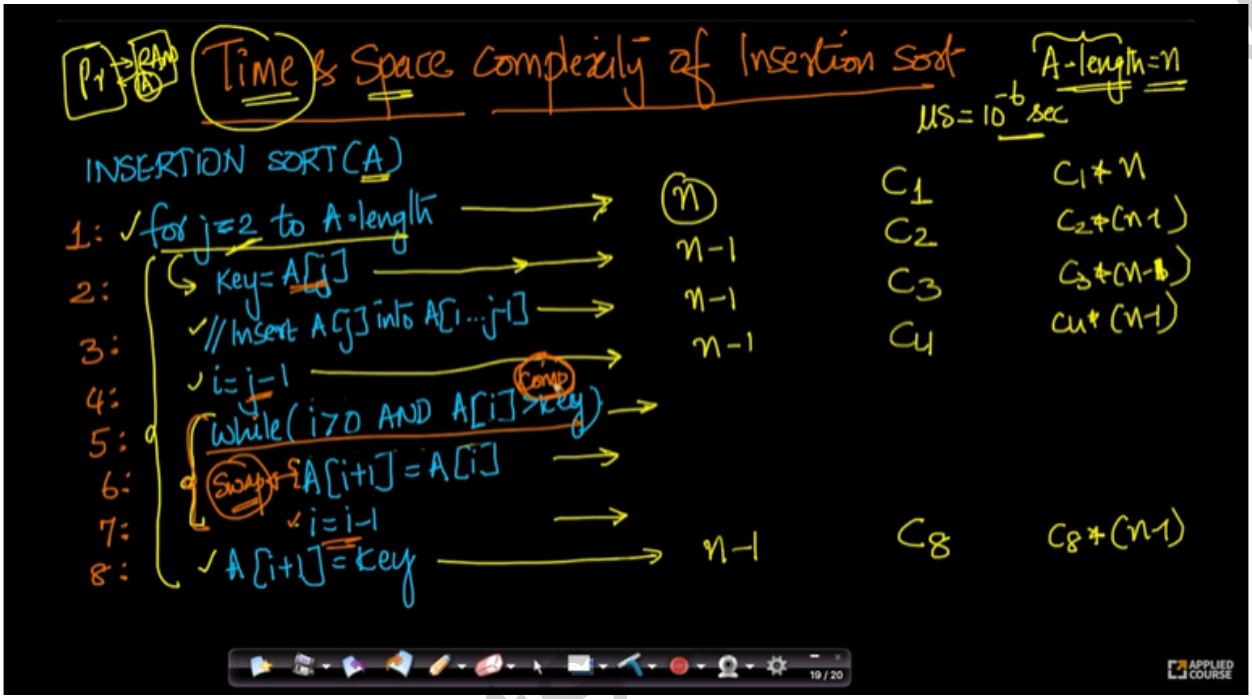
In this video we start analysis the space and time complexity of the insertion sort algorithm based on its pseudo code.

Notice that the first line of the code, the ‘for’ statement runs for n times while j takes values from 2 to n+1 (although the loop executes from 2 to n but at n+1 it has to check for condition and terminate). Let us say it takes C1 amount of time to execute it once. Then the amount of time it takes to run n times is C1*n.

Since the loop runs for n-1 times (from j=2 to n) each statement in the loop 2,3,4,8 runs for n-1 times(We will discuss the run time of remaining statements in the next video), assuming that the time taken to execute code in lines 2,3,4,8 for one time is C2,C3,C4, C8 the total time taken for executing code in lines 2,3,4,8 are C2*(n-1), C3*(n-1), C4*(n-1), C8*(n-1) respectively.

We will discuss the time taken for executing remaining statements and the total time for executing the complete algorithm in the next video.

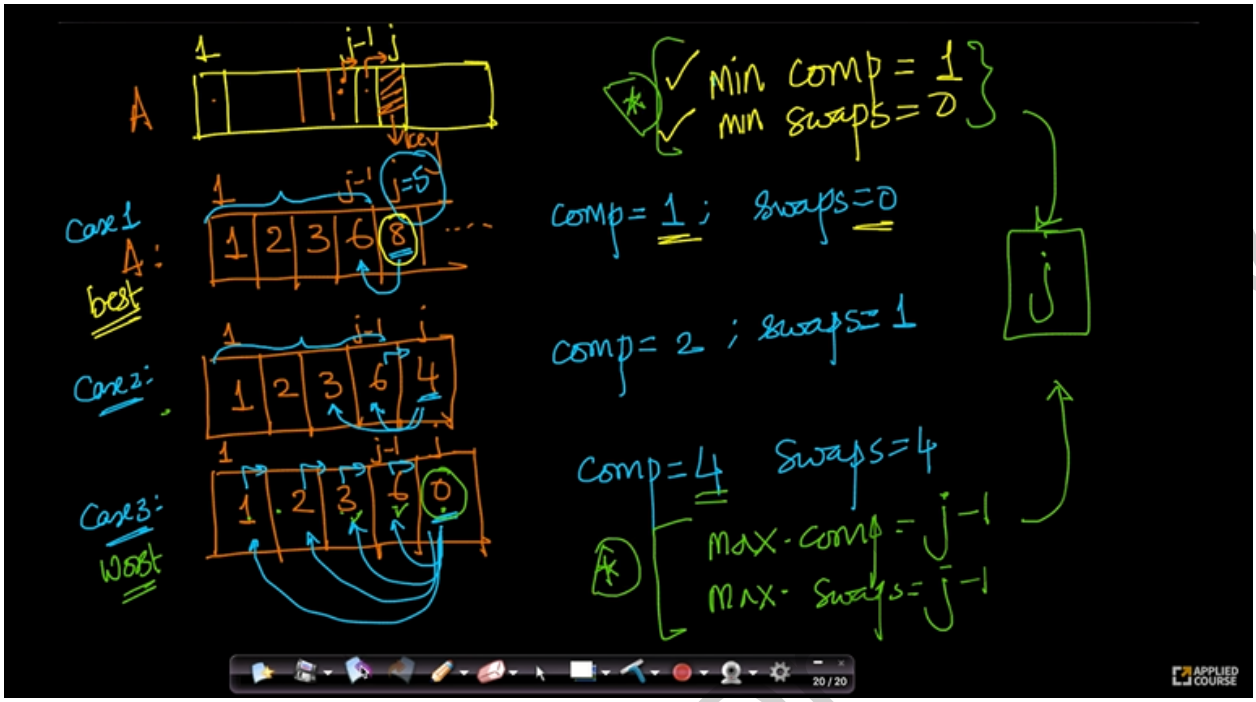
9.3 Space and Time Analysis of Insertion Sort-2



Timestamp: 2:42

Notice that the operation that is taking place at line 5 is comparison and at line 6 is swap(not two elements swap, but like moving element from one position to other) and at line 7 simple decrement operation.

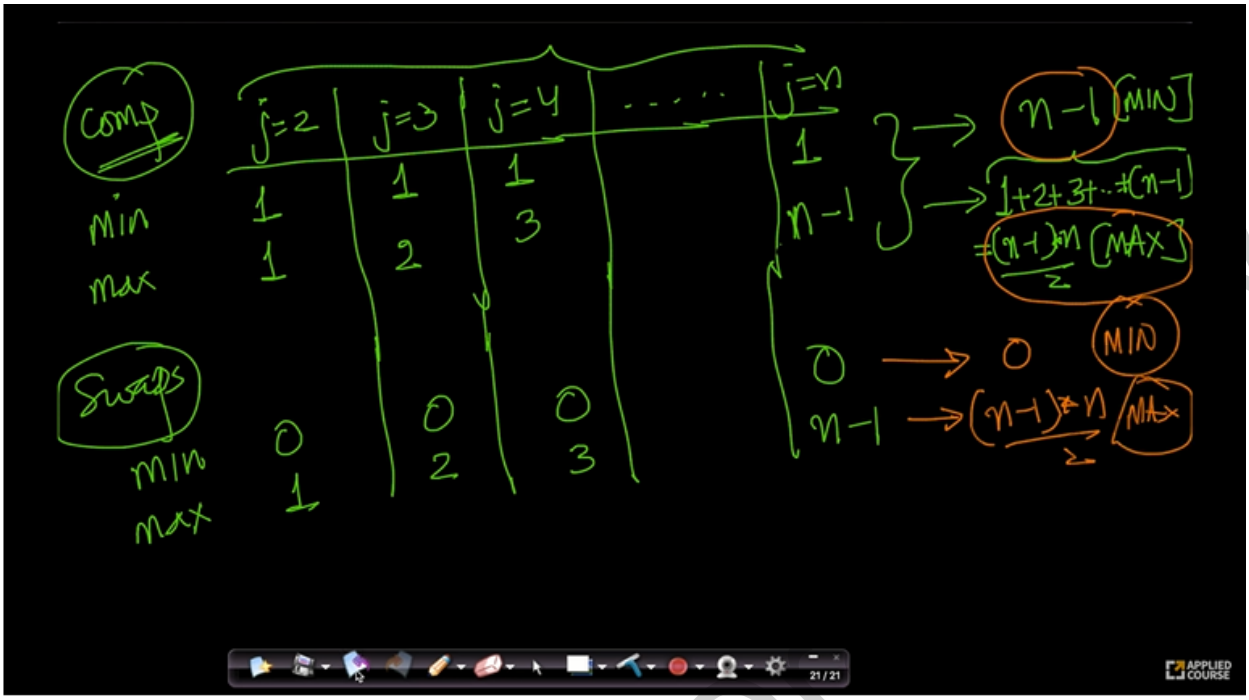
To understand, how many times each operation is executed, please look at the below figure.



Timestamp 7:04

Notice in the above figure, in the best case for any j we need to do 1 comparison and 0 swaps (since in this case A[j] 8 is greater than 6) while in worst case for any j we need to do j-1 comparisons and j-1 swaps (like in above figure we need to move 0 to the left after comparing with each element)

Notice that j takes values from j=2 to j=n, hence we have to calculate the total time for both the swap and the comparison for all values of j. Please take a look at the below figure.



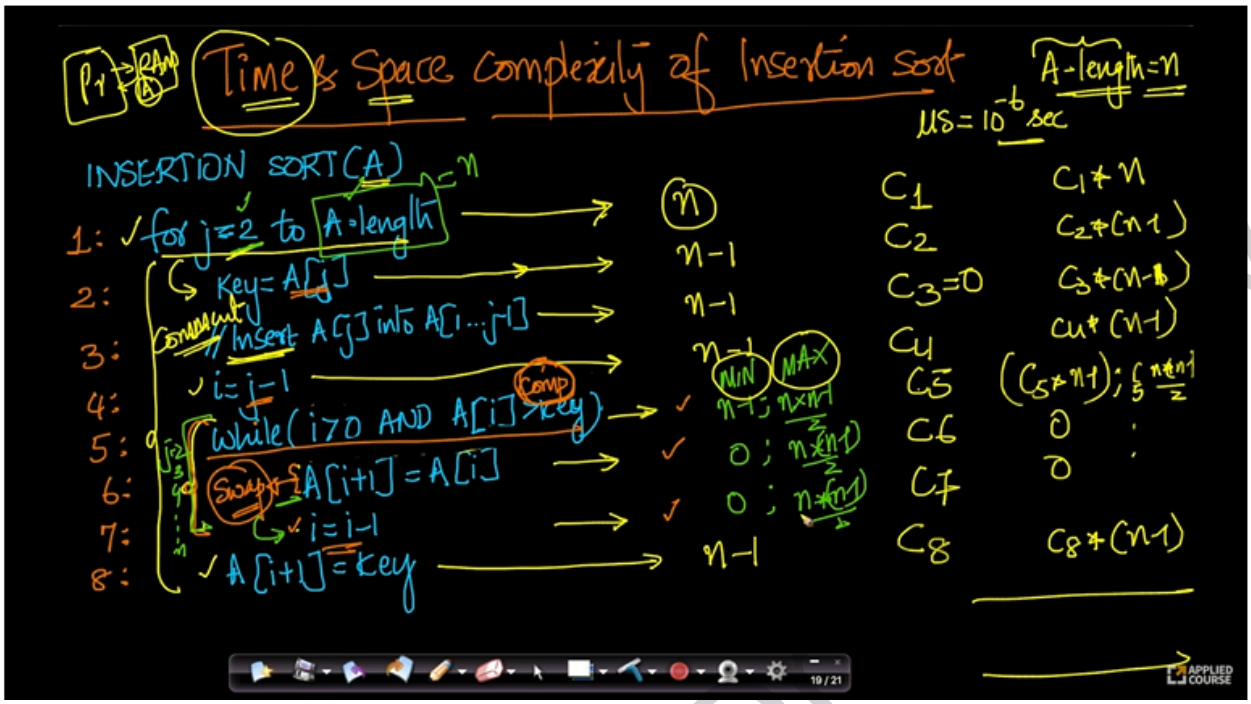
As in the above figure,

In best case the number of comparisons is 1 for a single value of j, so for j=2 to n. Number of comparisons is 1*(n-1) = n-1.

In worst case the number of comparisons is j-1 for a single value of j, so for j=2 to n. Number of comparisons is (2-1) +(3-1) +....+(n-1) = n*(n-1)/2

In best case the number of swaps is 0 for a single value of j, so for j=2 to n. Number of comparisons is 0*(n-1) = 0

In worst case the number of swaps is j-1 for a single value of j, so for j=2 to n. Number of comparisons is (2-1) +(3-1) +....+(n-1) = n*(n-1)/2



Timestamp: 12:57

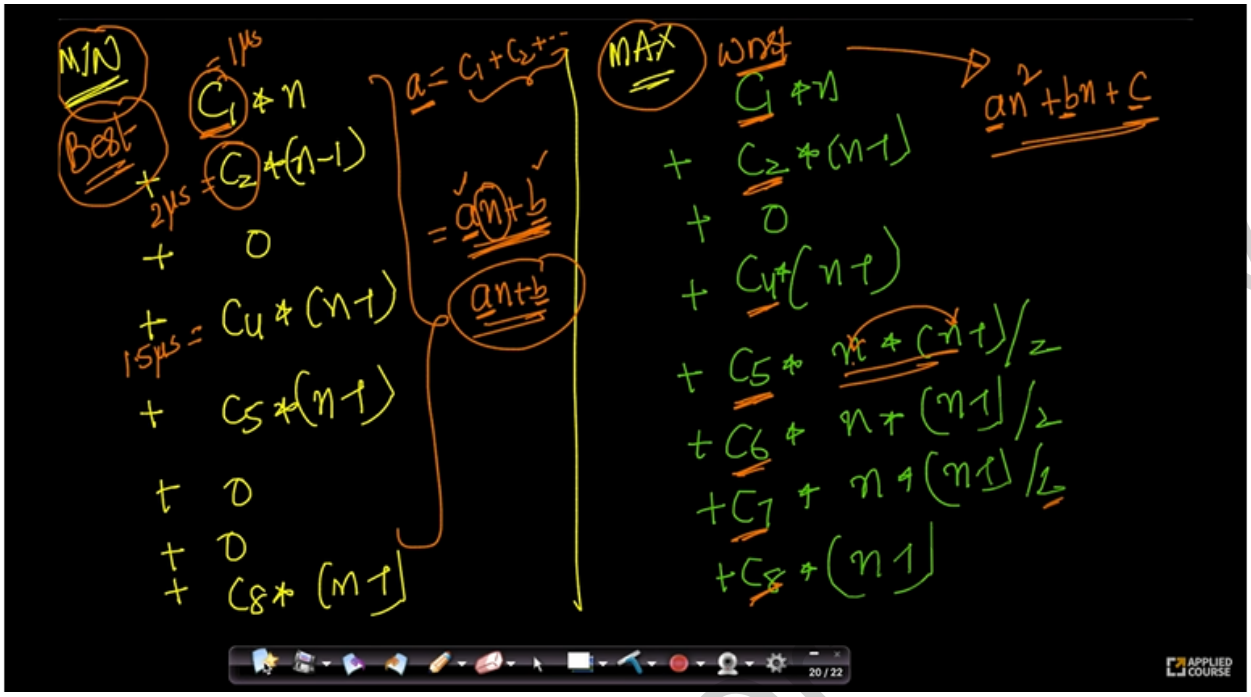
Similar to the previous video let us say the time taken for executing the code in line 5,6,7 once be C5,C6,C7.

In best case, code in line 5 runs n-1 times, line 6 0 times and line 7 0 times.

In worst case, code in line 5 runs n*(n-1)/2 times, line 6 n*(n-1)/2 times and line 7 n*(n-1)/2 times.

Note that since 3rd line is a comment, C3=0

So the total time taken for executing all the lines in best case and worst case is as below.

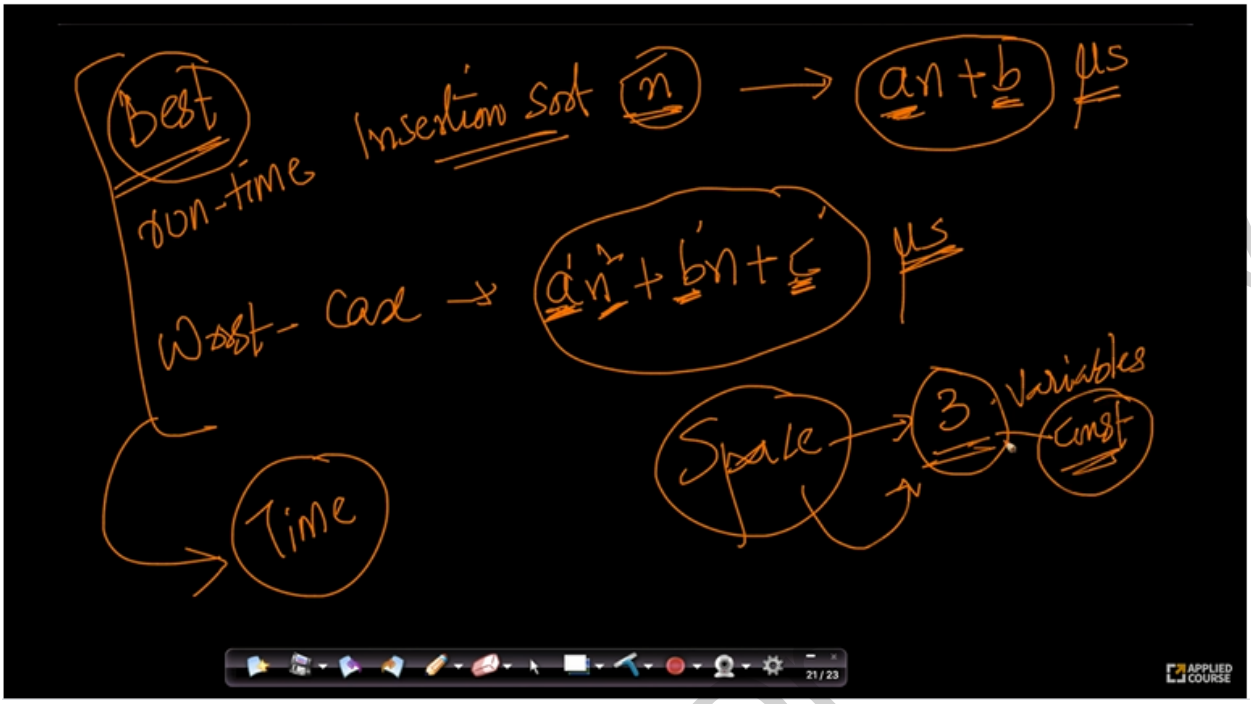


Timestamp 18:14

So as shown in the above figure

Best case time taken for executing the insertion sort in best case is **an+b** for a and b be as some combination of C1,C2...C8

Worst case time taken for executing the insertion sort in worst case is $a'n^2 + b'n + c'$ for a',b',c' as some combination of C1,C2...C8 and /2's.



Timestamp: 21:41

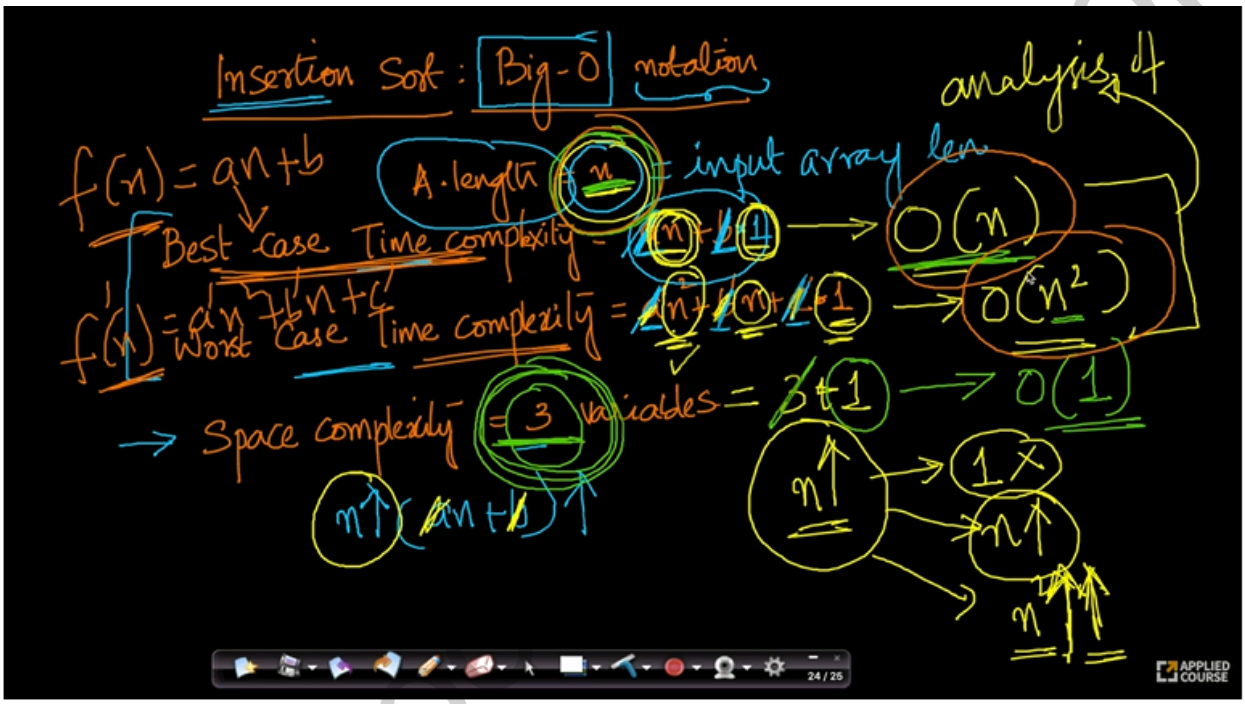
Notice that space and time complexity are expressed in terms of n, the size of the input. As for space complexity, whatever the size of the input, we are only using 3 **additional** variables apart from the actual input array. Since this 3 is independent of the size of the array n we treat it as constant.

Hence concluding, insertion sort has best case time complexity of an+ b, worst time complexity of $a'n^2 + b'n + c'$ and constant (3 variables) space complexity for input size n.

9.4 Insertion Sort: Big O-notation

Remember that in the previous video,we discussed that

“Hence concluding, insertion sort has best case time complexity of an+ b, worst time complexity of $a'n^2 + b'n + c'$ and constant (3 variables) space complexity for input size n.”



TImestamp: 14:21

The best case time complexity of insertion sort an+ b for the simple case of a=1 and b=1 is n+1. When n=1, its value is 1+1=2, when n=10 its value becomes 10+1=11, when n=100 its value becomes 100+1=101.

Notice that as n increases the best case time complexity of n increases proportional to n. Hence its time complexity is said to be O(n).

The worst case time complexity of insertion sort $a'n^2 + b'n + c'$ for the simple case of a’=1, b’=1 and c’=1 is n^2+n+1.
When n=1, its value is 1+1+1=3, when n=10 its value becomes 100+10+1=111, when n=100 its value becomes 10000+100+1=10101.

Notice that as n increases the worst case time complexity of n increases proportional to n^2. Hence its time complexity is said to be $O(n^2)$. Although it also increase with n, the increase with n^2 is much more larger when compared to the increase with n.

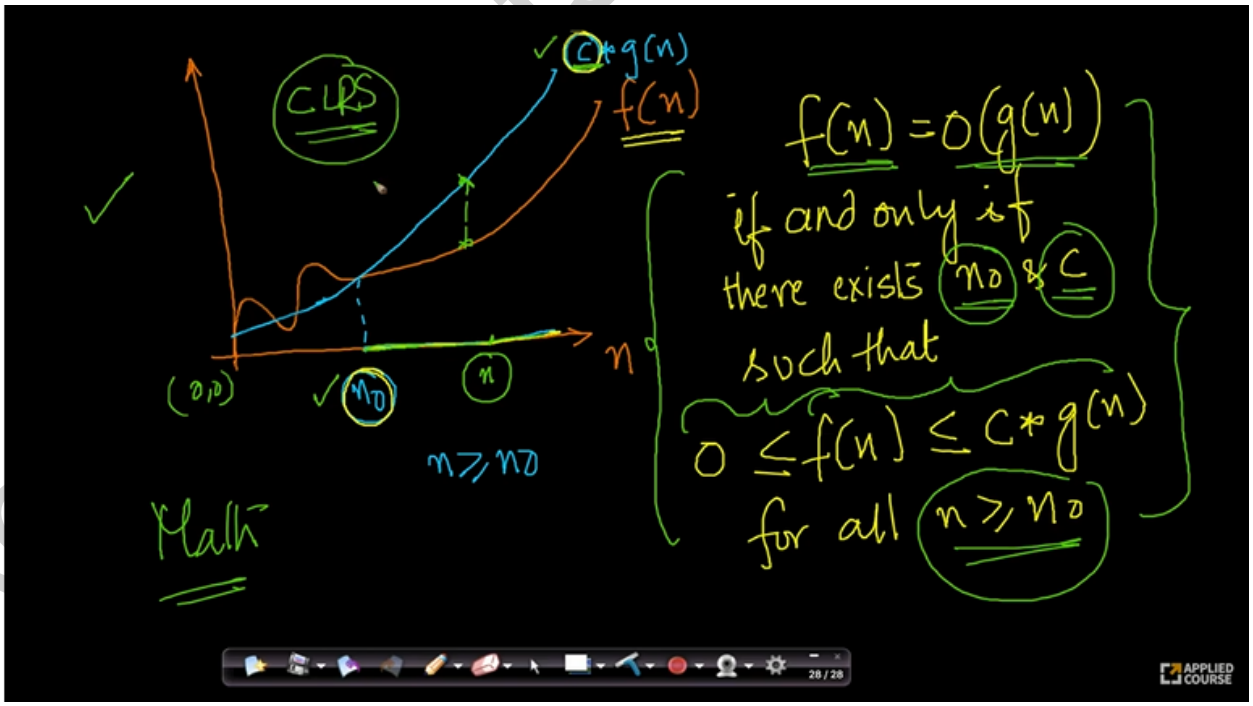
Please have a look at the graphs in the below link to understand how the values of n and n^2 increases as n increases.

https://stackoverflow.com/questions/23329234/which-is-better-on-log-n-or-on2

Coming to the space complexity, as n increases the space complexity does not increase and remains same(3 variables) hence insertion sort is said to have constant space complexity and is denoted by O(1).

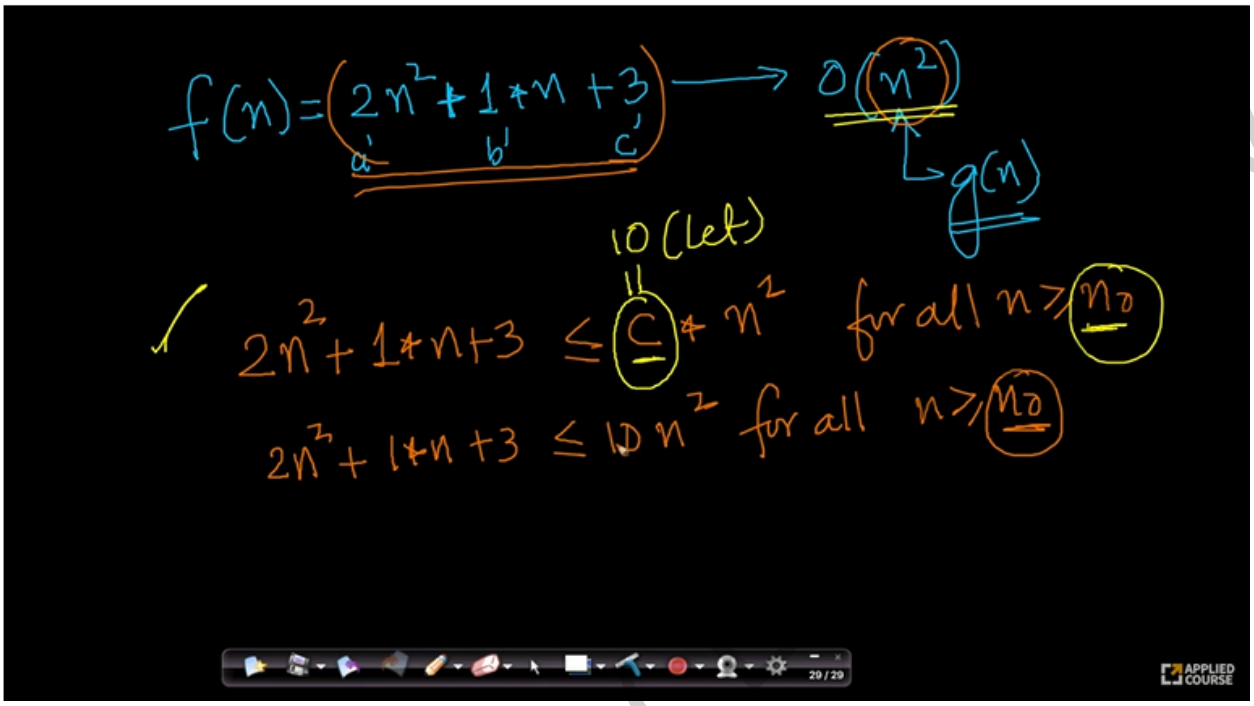
To conclude, the best case time complexity of insertion sort is O(n), the worst case time complexity of insertion sort is $O(n^2)$ and the space complexity of insertion sort is O(1).

9.5 Notations : Big O



Timestamp: 6:10

As shown in the above figure, for any functions f(n) and g(n), function f(n) is said to be O(g(n)) if and only if for some constants n_0 and c , 0<= f(n) <=c* g(n) for all n>=n_0.



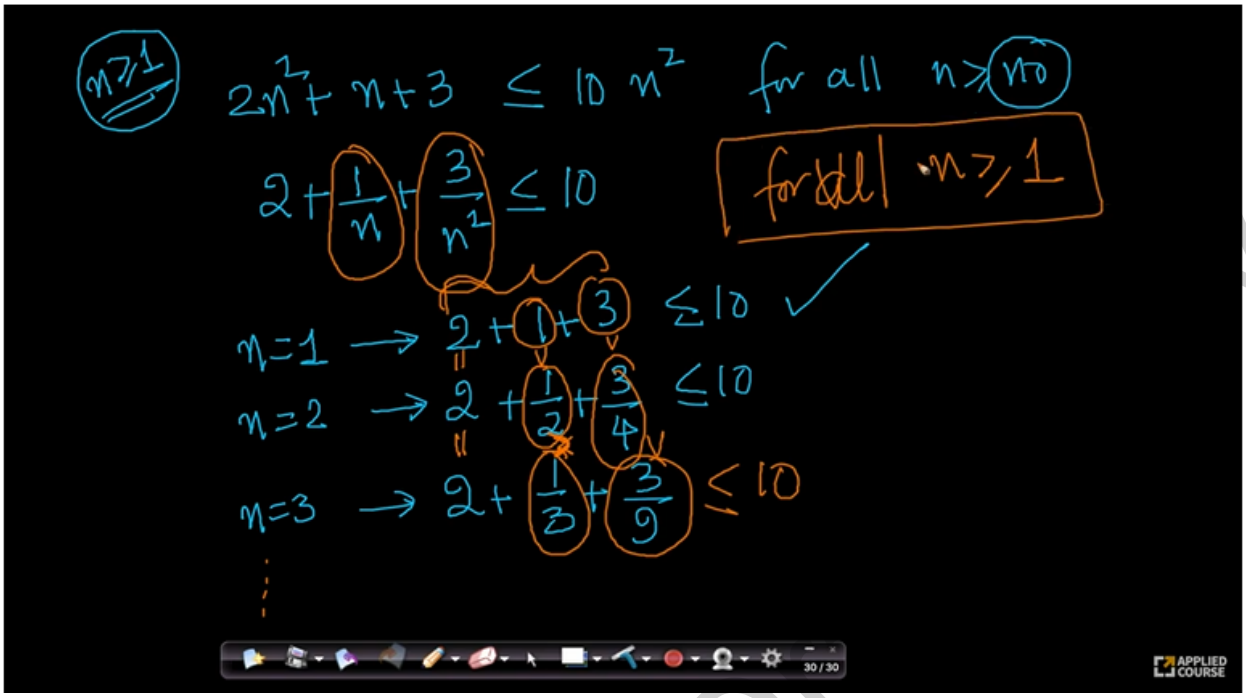
Timestamp: 10:15

Let us say we have some function f(n) = $2n^2$ + n +3 to prove that it is O(n^2) (Note: g(n)= n^2) , we need to find some c and some n_0 that satisfies the condition:

$2n^2 + n + 3 < = c*n2$ for all $n >= n_0$

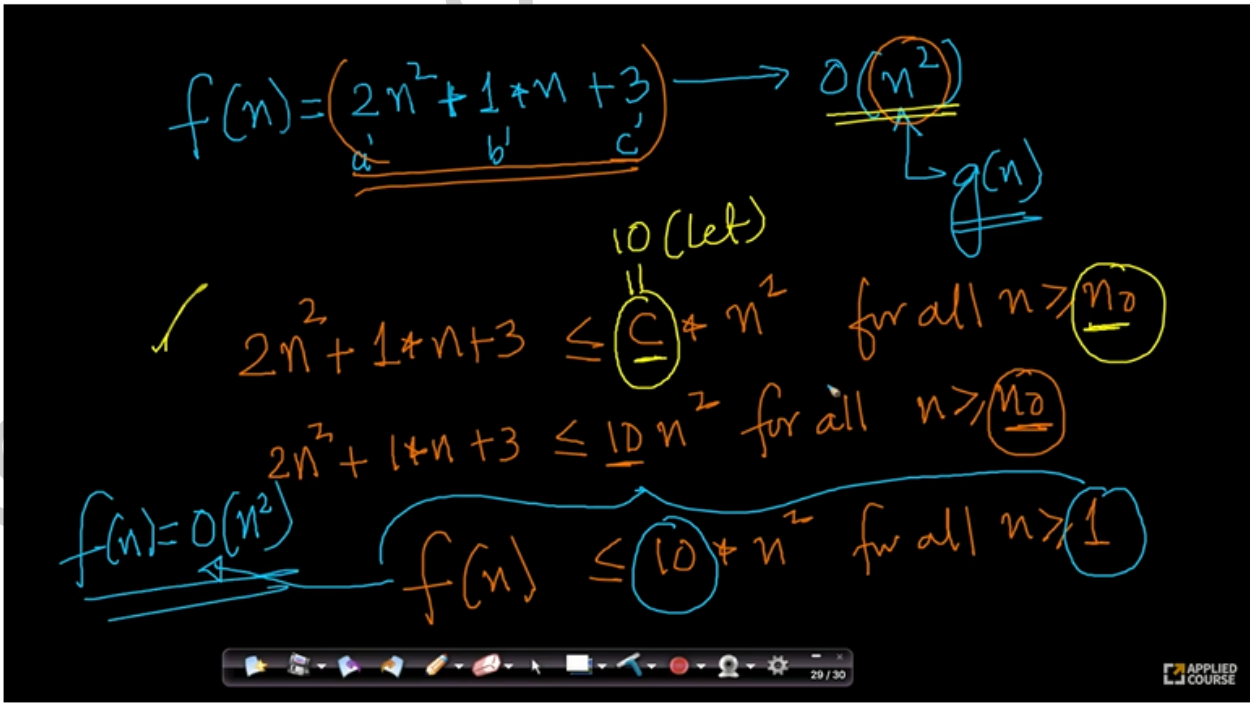
Let c=10 then $2n^2 + n + 3 <= 10n^2$ for all $n >= n_o$

We need to find n_o that satisfies this condition.



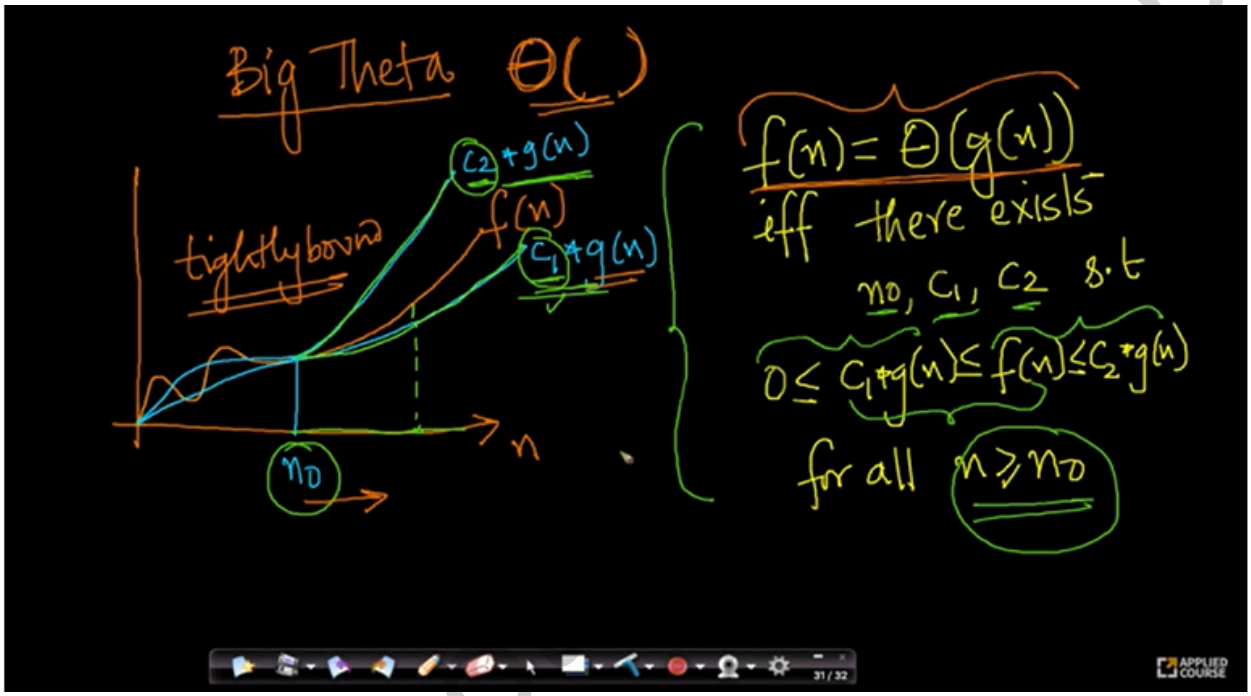
We can observe that for n=1,2,3 this inequation holds and as n increases the value of $2n^2+n+3$ decreases further hence it will always be less than 10. Hence for all n>=1, this inequation holds.

Therefore we are able to find such n_o =1 and c=10 for which the inequation $2n^2 + n + 3 <= c*n2$ for all $n >= n_0$ holds. Thereby proving that f(n) is $O(n^2)$.



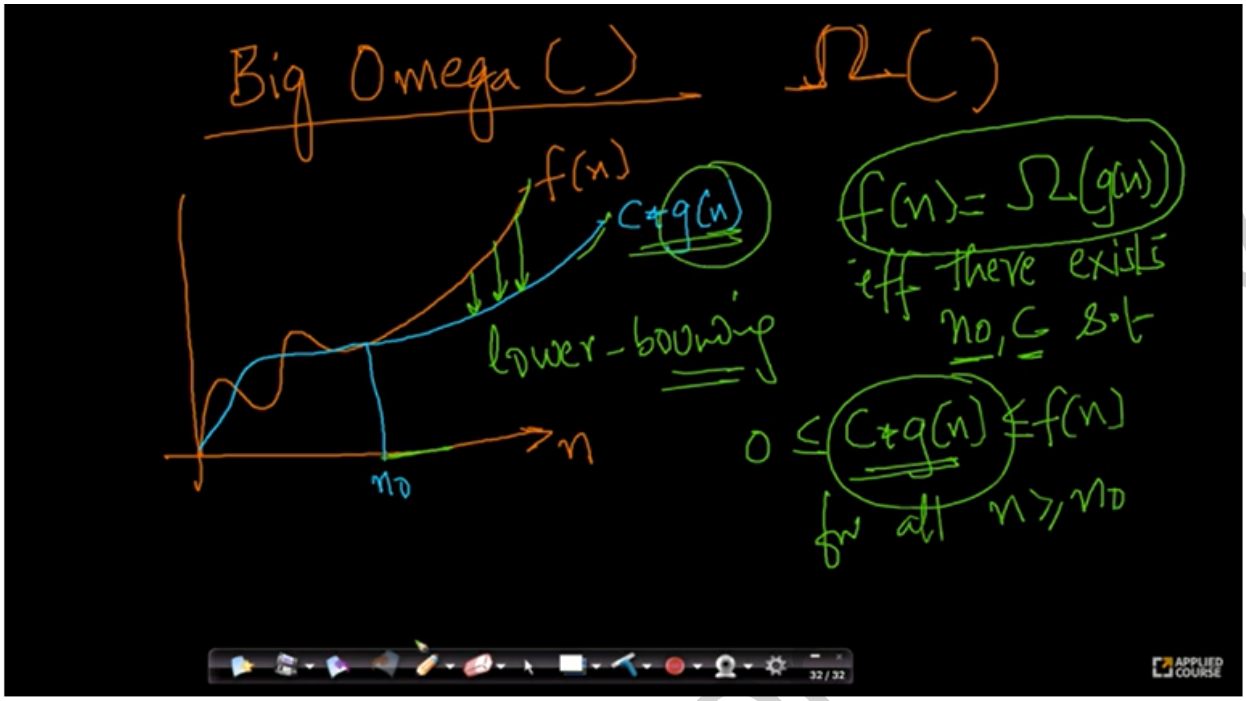
Similarly for any values of a', b' and c' we can find n_o and c for which the inequation satisfies. Therefore $f(n) = a' n^2 + b'n + c$ is always $O(n^2)$.

9.6 Notations : Big Omega, Theta



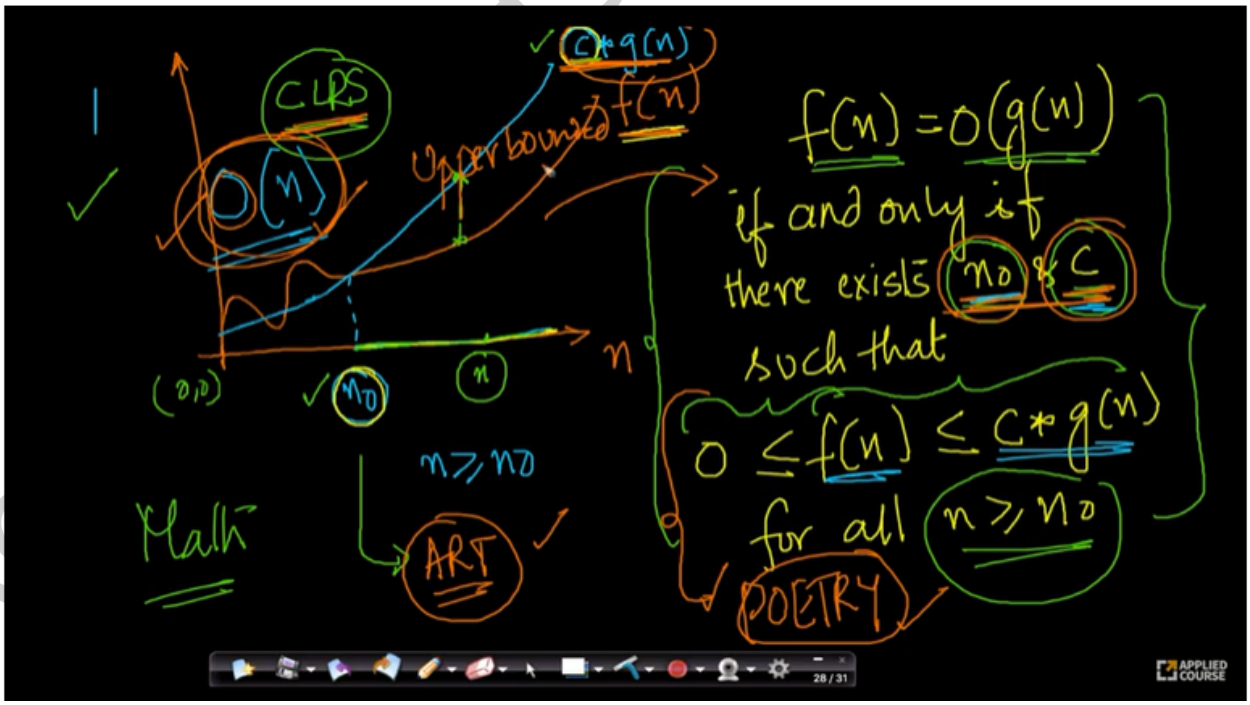
Timestamp: 4:39

If there exists two functions f(n) and g(n), then f(n) is said to be Θ (g(n)) if and only if there exists constants n_o, c_1,c_2 such that 0 <= c_1 * g(n) <= f(n) <= c_2 * g(n) for all n>= n_o . We can think of f(n) being tightly bounded (meaning both upper bounded and lower bounded) between c_1*g(n) and c_2*g(n) incase of f(n) = Θ (g(n)).



Timestamp: 8:03

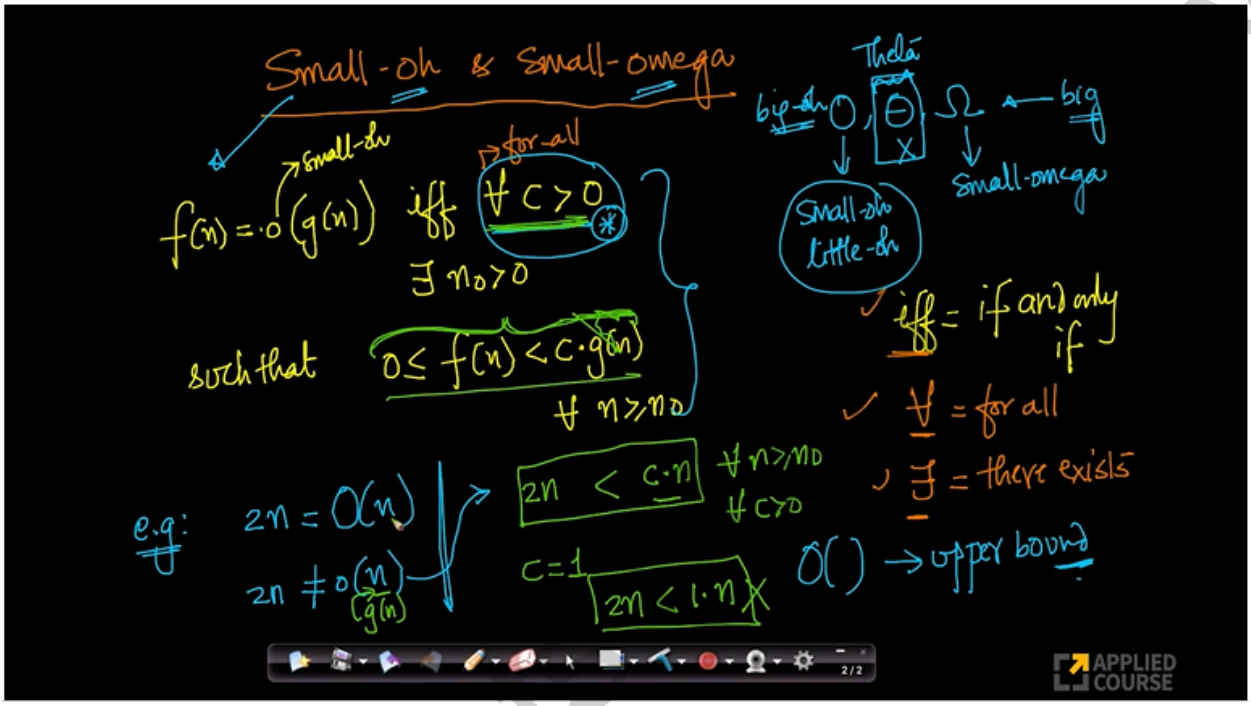
If there exists two functions f(n) and g(n), then f(n) is said to be Ω (g(n)) if and only if there exists constants n_o, c such that 0 <= c * g(n) <= f(n) for all n>= n_o . We can think of f(n) being lower bounded by c*g(n) incase of f(n) = Ω (g(n)).



Timestamp: 5:39

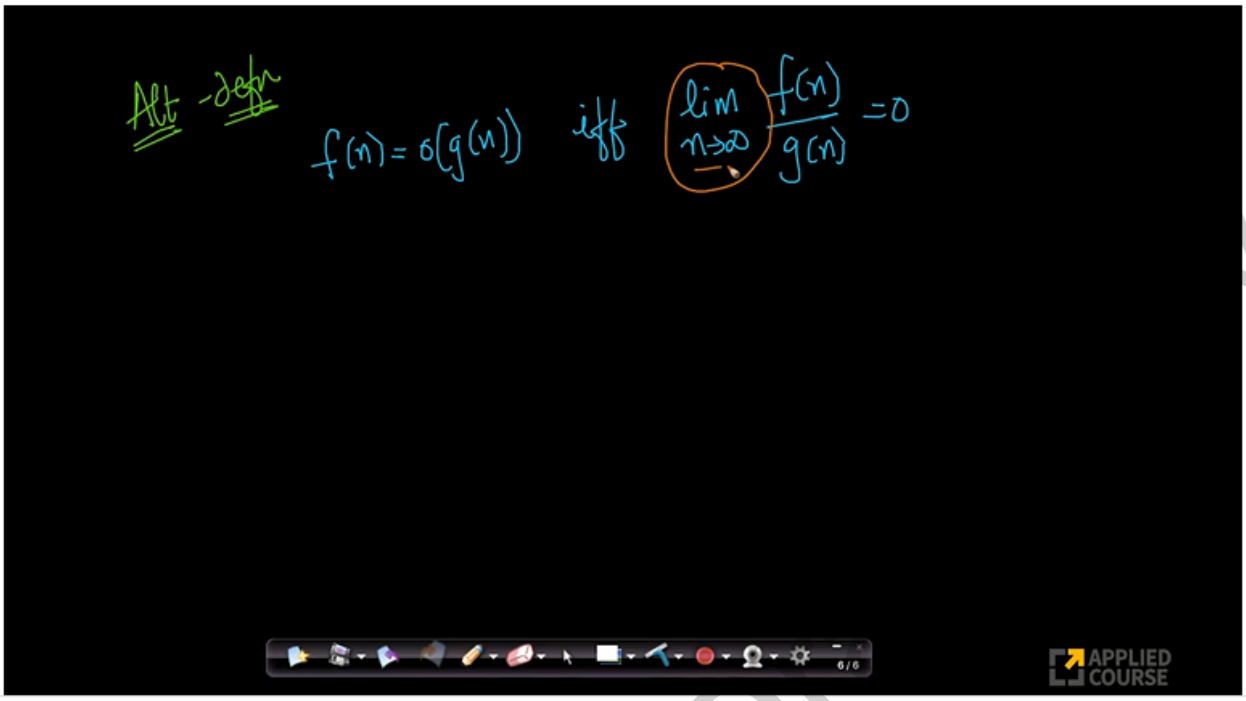
Incase of f(n)=O(g(n)), f(n) is upper bounded by some constant c*g(n).

9.7 Notations : Small O,Omega, Theta



Timestamp: 23:59

Definition1: If there are two functions f(n) and g(n) then f(n) = o(g(n)) (Note o!=O) if and only if for all c>0 there exists a constant $n_o > 0$ such that 0<= f(n) < c*g(n) for all n>= n_0



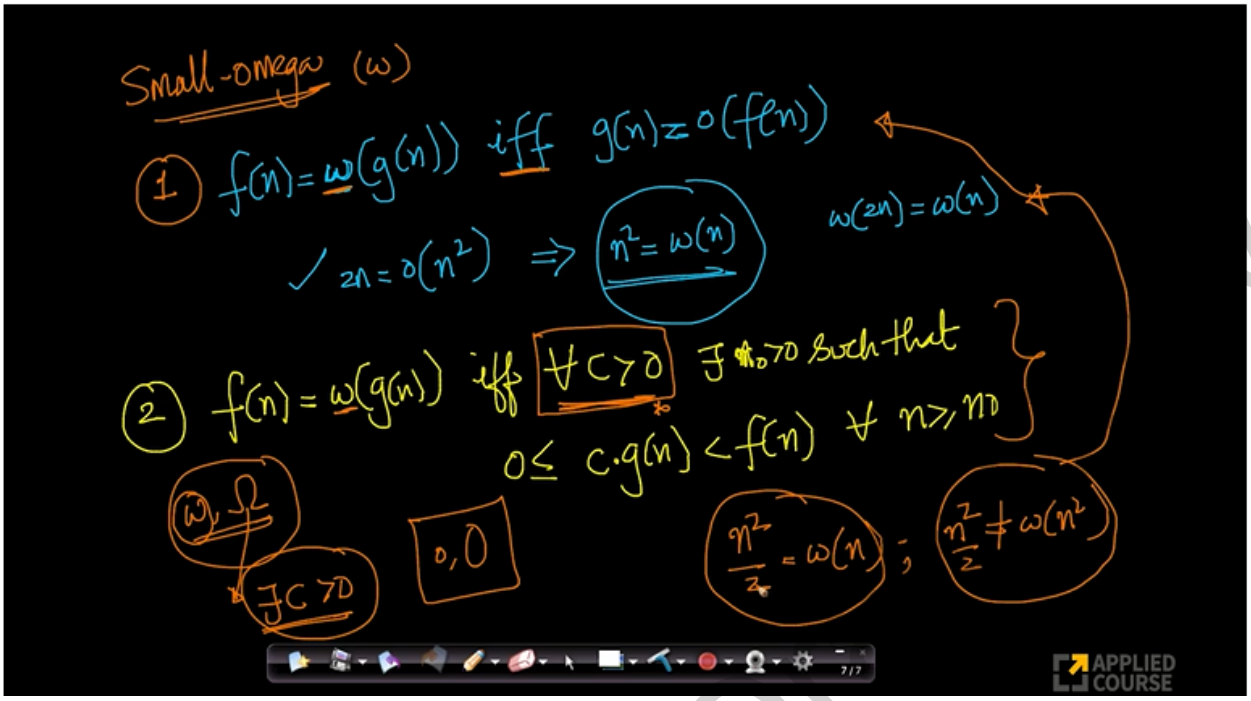
Timestamp: 13:54

Alternate definition: If there are two functions f(n) and g(n) then f(n) = o(g(n)) if and only if

Lim f(n) / (g(n) =0.
n-> ∞

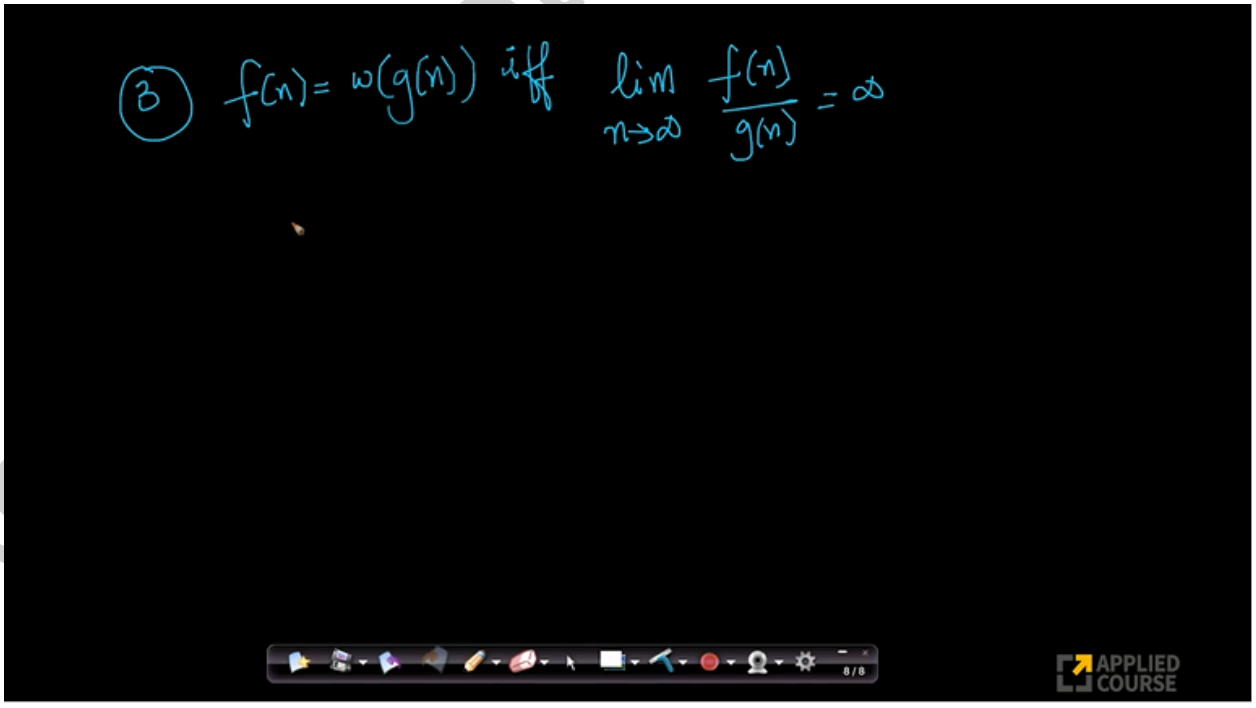
For example:

Function $f(n) = 2n^2$ is $O(n^2)$ but not $o(n^2)$.



Definition1: If there exists two functions f(n) and g(n), then f(n) = ω (g(n)) if and only if g(n)=o(f(n)).

Definition 2: If there exists two functions f(n) and g(n), then f(n) = ω (g(n)) if and only if for all c>0 there exists a constant n_o>0 such that 0<= c*g(n) < f(n) for all n>=n_o

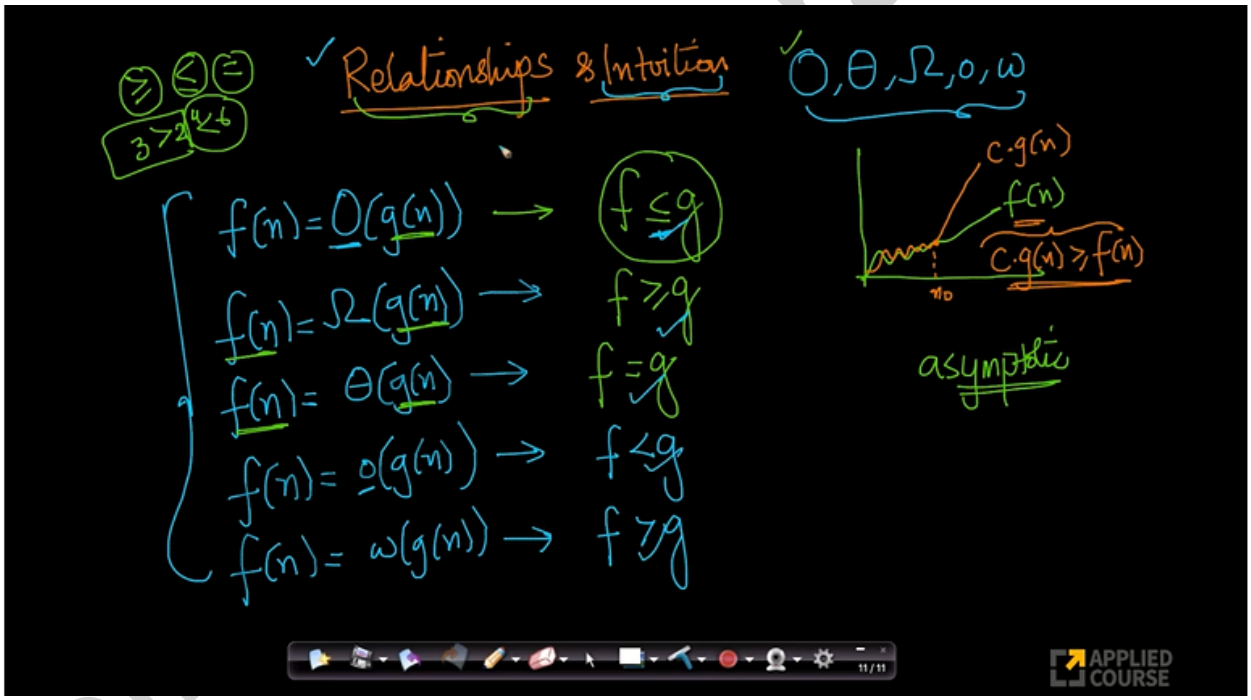


Timestamp: 20:36

Definition 3: If there exists two functions f(n) and g(n), then f(n) = ω (g(n)) if and only if
Lim f(n) / (g(n) =0.
n->∞

The key difference between O and o is that the condition should be satisfied **for any** constant c in O but should satisfy **for all** c>0 in case of o. The same difference applies in case of Ω and ω as well.

9.8 Relationship between various notations

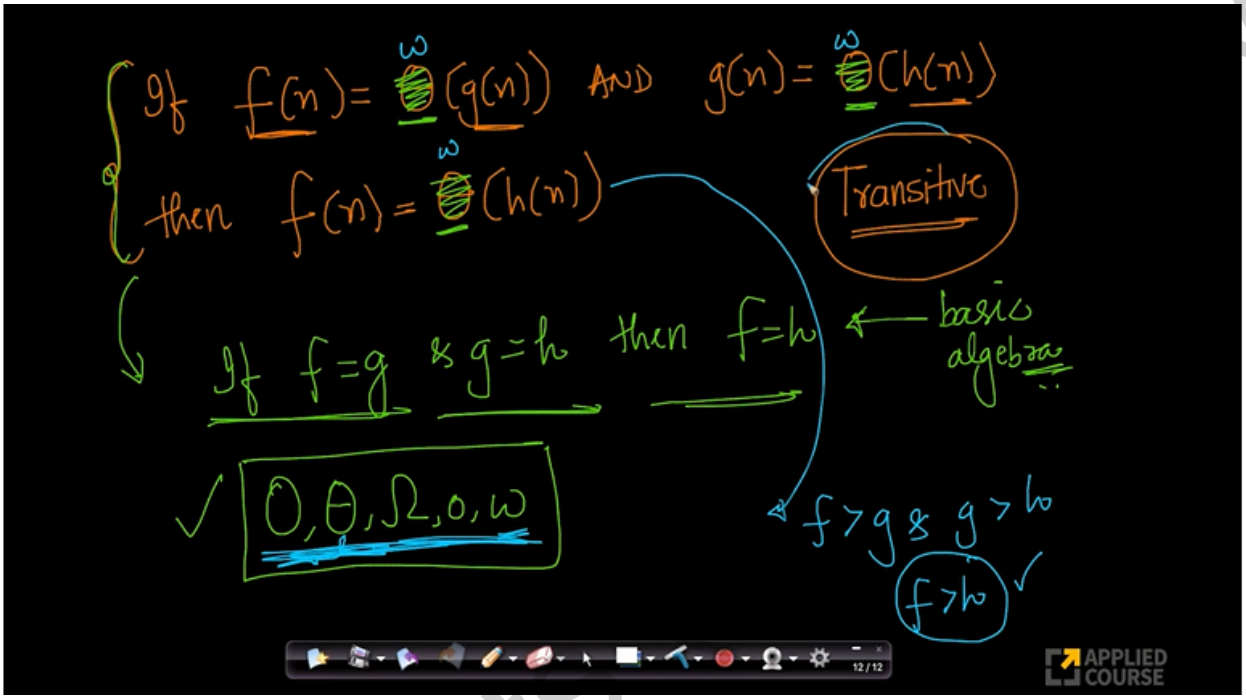


Timestamp: 4:53

Our various notations can be thought of as f<=g, f>=g,etc as shown in the above figure.

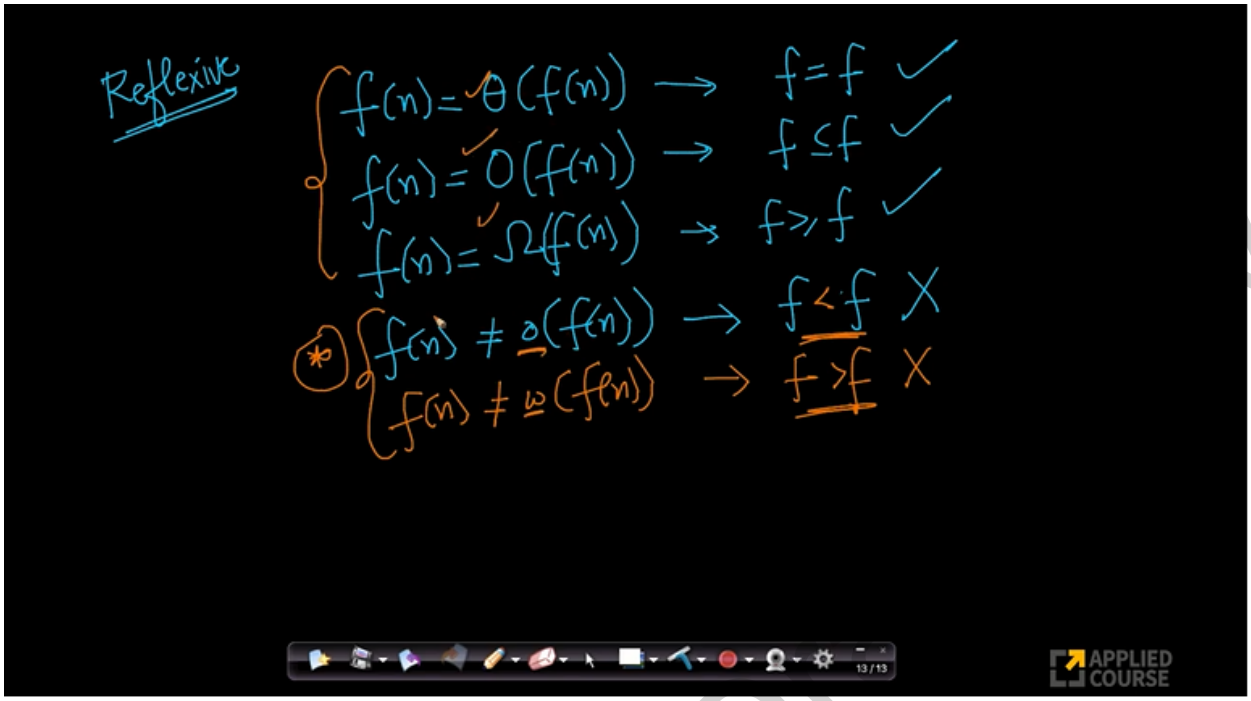
f(n)= O(g(n)) can be denoted as f<=g since f is upper bounded by g.
f(n)= Ω(g(n)) can be denoted as f>=g since f is lower bounded by g.
f(n)= Θ(g(n)) can be denoted as f=g since f is tightly bounded by g.

f(n)= o(g(n)) can be denoted as f< g since f is strictly upper bounded by g (referring to 0<= f(n) < c*g(n) in the definition of o)
f(n)= ω(g(n)) can be denoted as f>g since f is strictly lower bounded by g (referring to 0<= c*g(n) < f(n) in the definition of ω)



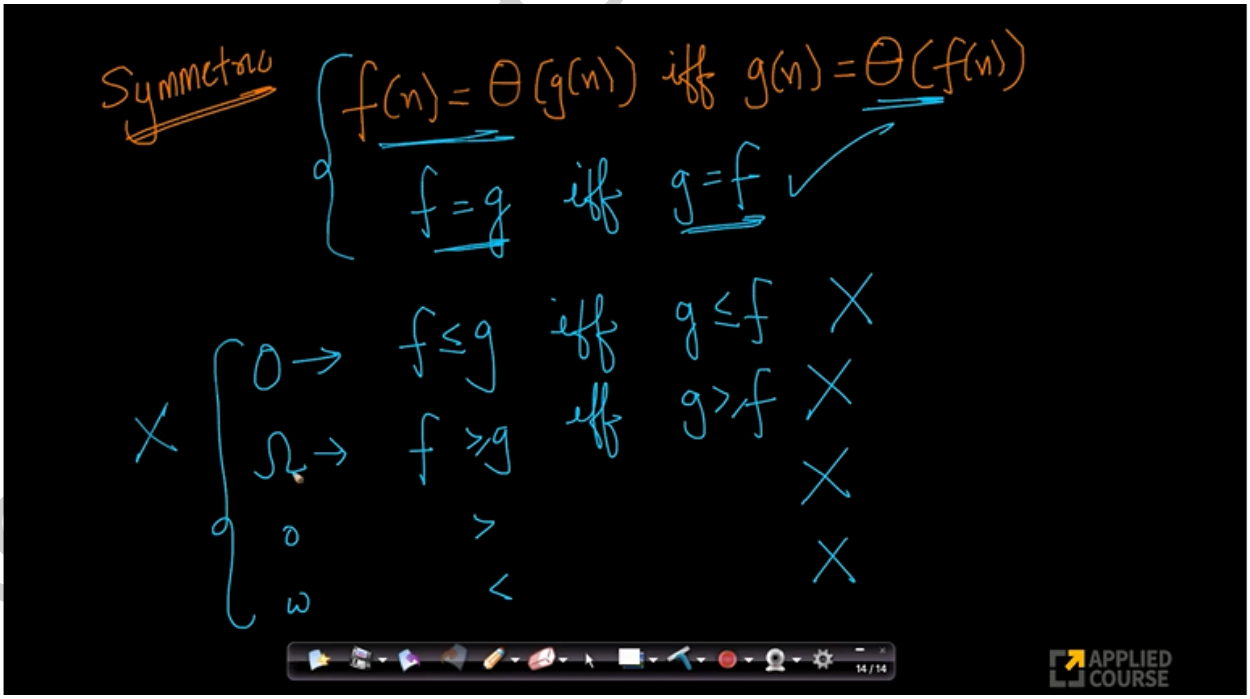
Timestamp: 8:35

All the notations O,Θ,Ω, o, ω hold transitive property as shown in the figure.



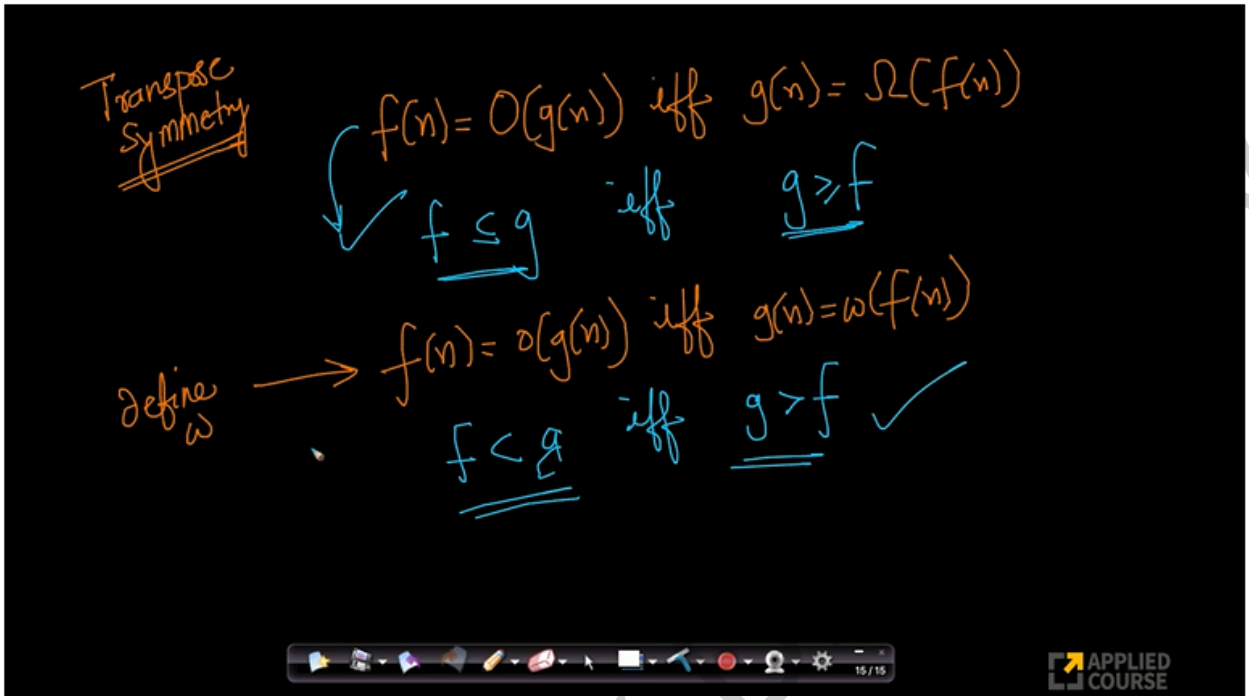
Timestamp: 10:55

Reflexive property only holds for O,Θ,Ω but doesn't hold for o, ω as shown in the figure.



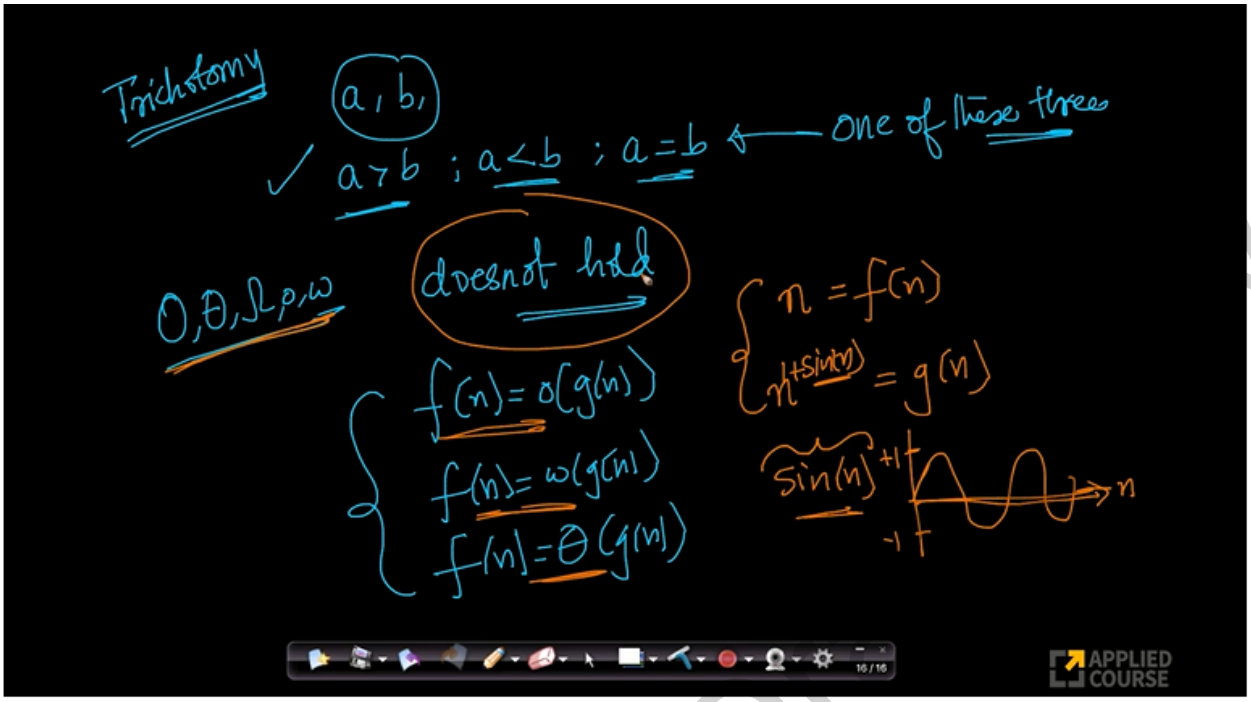
Timestamp: 12:57

Symmetric property only holds for Θ as shown in the figure.



Timestamp: 15:42

Transpose symmetry property holds for the pairs of (O,Ω) and (o,ω) as shown in the figure.

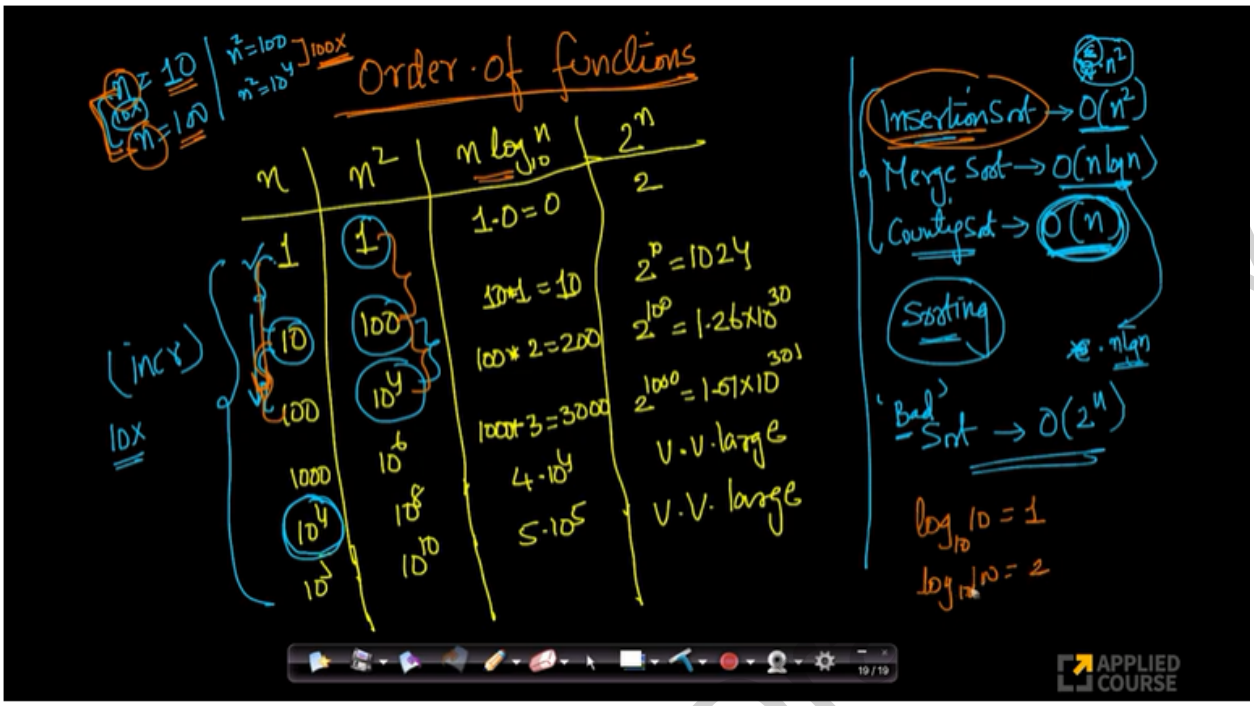


Timestamp: 18:22

Trichotomy is the property by which we can say for any two numbers a,b either of a>b, a<b and a=b must be true.

This property does not hold for our notations O,Θ,Ω, o, ω.
An example to prove this is f(n)= n and g(n) = $n^{(1+\sin(n))}$, notice that g(n) takes values n^0, n^1, n^2 hence cannot be expressed as g<f or g>f or g>f.

9.9 Order of common functions & real world applications



Timestamp: 5:45

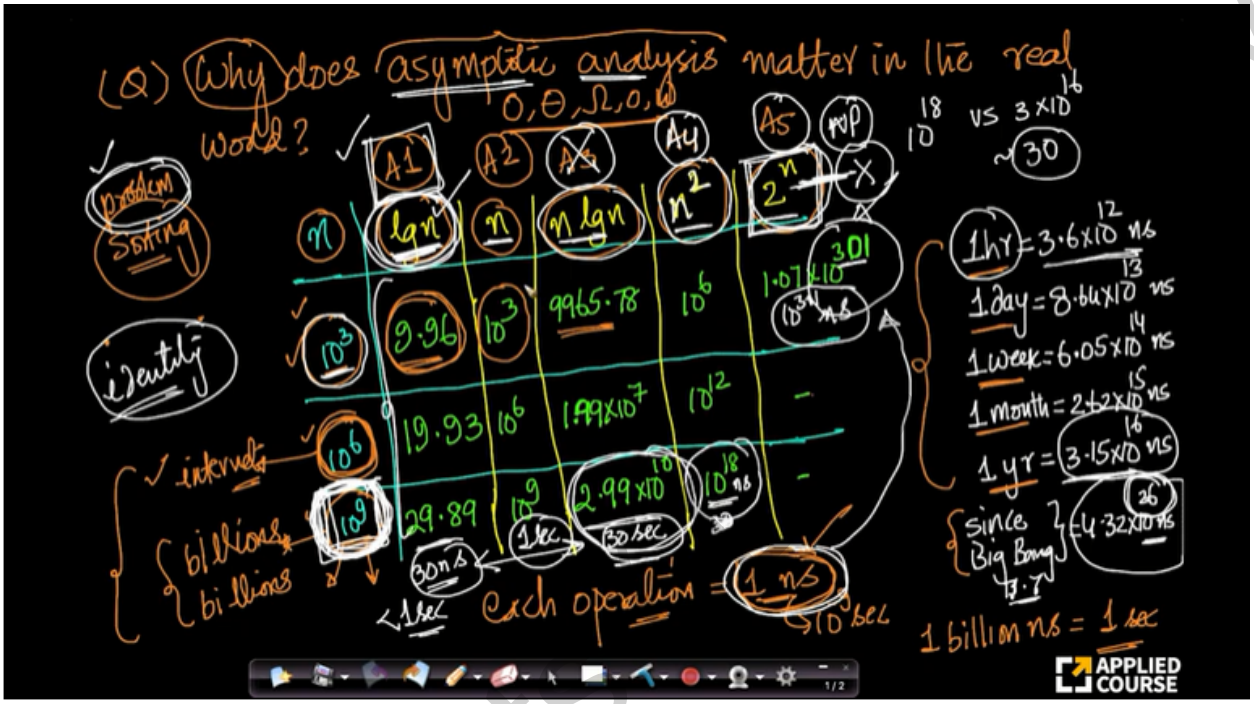
Notice that as in the above figure, as n increases, 2^n increases much more than n^2 which increases more than nlogn which increases more than n.

Please refer the order_of_common_functions section in the below link that shows different time complexities in increasing order.

Comparing these computation complexities of the algorithms helps us choose the right algorithm that can save both run-time (time complexity) and memory requirement (space-complexity) for solving a problem.

https://en.wikipedia.org/wiki/Big_O_notation#Orders_of_common_functions

9.10 Why does asymptotic analysis matter in real world?



Timestamp: 15:24

Often in read world where some countries like India and China have populations of billions, the value of n might be around 10^9, in such cases while algorithms like logn and n can take from a few seconds to days some algorithms like 2^n can take time very much longer than the time since universe existed.

Hence making an extra effort to come up with algorithms of lesser time complexity can vastly save our time. Hence they are very important in real-world scenarios.