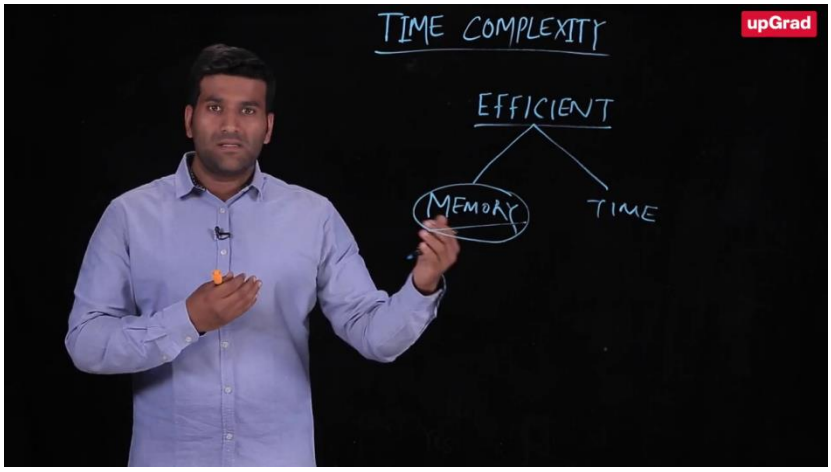
Transcription

Time Complexity

So, the next module which we will be covering is time complexity. So, this is one of the most important module, because here we will try to understand how do you measure whether between two algorithm, which one is more faster or efficient.

Now, when I talk efficiency, how do you measure it? So, there are two major component to measure efficiency of an algorithm.



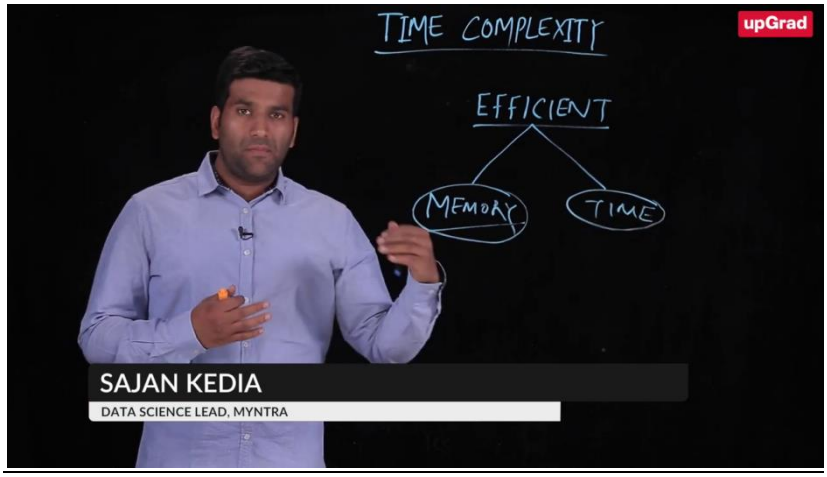
So, let's just talk about this. So, there are two main components. So, first is the memory and second is the time.

So, let's talk about the memory part first. So, as you know that we have a very limited memory. If you are any algorithm in your local machine or computing power, so you have very limited memory space.

Now, how do we use memory space in any algorithm? So, when you are writing your program, you must be using different variables or list or dictionary. So, whenever you create a new list or dictionary, at that time you are using

the memory space. So, that's why whenever you are writing efficient code, you should always think about not initialising too many variables or not using too many unnecessarily copy of the same list or dictionary. So, that will save you memory space and make the program efficient.

Now, let's talk about the time part.



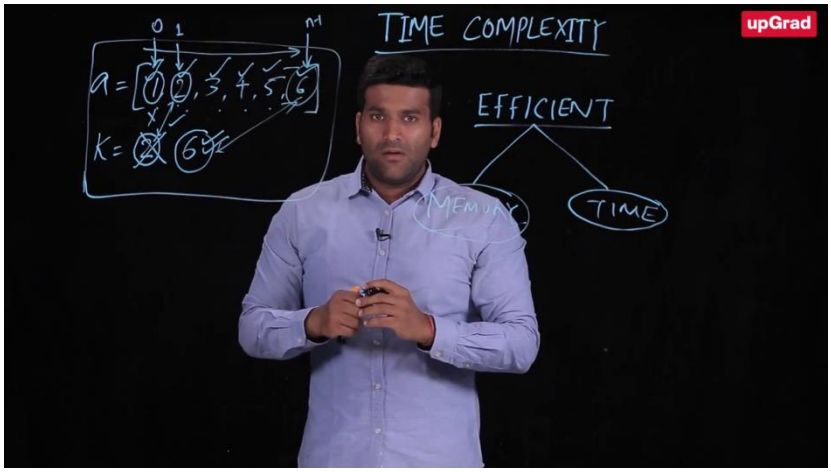
So next, we will talk about the time part. So, how do you measure which algorithm is better in terms of how much time it is taking? So, first let's talk about what is program or what is algorithm.

So, algorithm is nothing but just a series of steps or series of operations, which we operate one after other. Now, for some operation, it can take more number of iteration. So, it will take more time. For some operation, it can take less time.

So, let’s understand the time part with a very simple example. So, I will just take example of one simple list.

So, assume you have a list A = 1, 2, 3, 4, 5, 6. Now, assume you have a list and you have to search a particular item here.

So, let's say you have to search K = 2 here.



So, as we have seen in the previous module, so for this, what you can do is you can just write a for loop where you will just iterate from I = 0 to I = last element, which is n – 1 and you will just check if this element is equal to this. So, you will return it.

So now, if we just try to run this approach or if we try to run set of operation, we can see how time factor is very important here. So, let's run here. So first, I will check whether 2 = 1. So, this is not equal.

Now next, I will check for this index 1. So, here 2 and this K, both are equal. So, I will just stop here and I will just return that this is the index of the element.

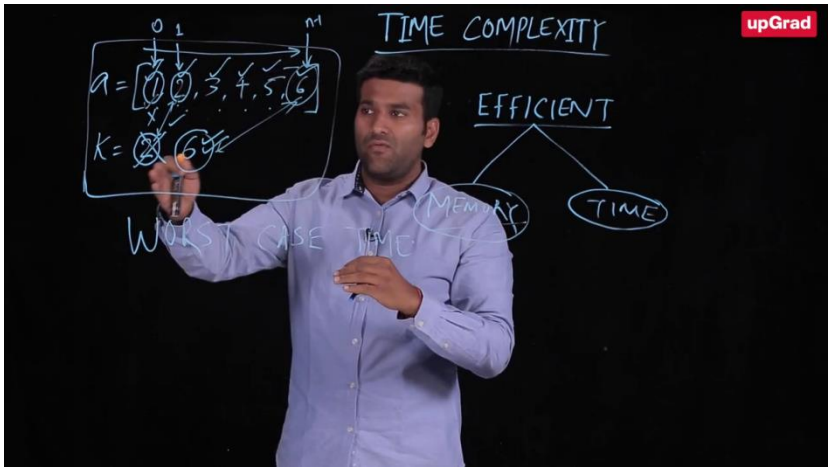
Now, what if you have to search K = instead of 2, you have to search 6. Now, in that case, you need to iterate from here, you will first check with 1, 2, 3, 4, 5, and then at last you will check with 6, which is the last element.

So, now here, if you see just for two different input, for the first one, it will take much lesser time because immediately you are able to find the element here, but when you are trying to search 6 here, you are first comparing here 1, 2, 3, 4, 5, 6. So, at the sixth iteration, you are able to find this element.

So, you can get some basic intuition that when you are searching these two element, the time what it will take for the first element will be much lesser as compared to the 6.

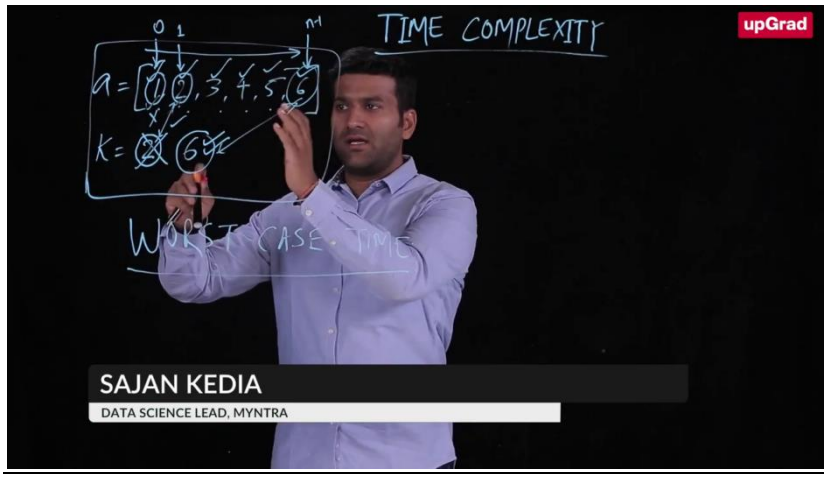
So, that is the basic intuition of the time complexity or the time efficiency part. Now, one thing which we can notice from this example is, if anyone is asking you that what is the worst time or how much time this algorithm or this piece of code will take.

So, ideally, you should tell the maximum time it is taking. So, maximum time it will take will be example 6, not for example 2, right. Because in the worst case, you may need to iterate from here to here, which is the last element and then you give the output.



So, that is the reason for any piece of algorithm or any code, whenever somebody asks us how much time it will take, we will always tell about the worst case time.

So here, there is no point of telling how much time it takes for the first element or second element search. We will always tell what is the time it takes for the last element to search. So, this is the idea of worst case time.

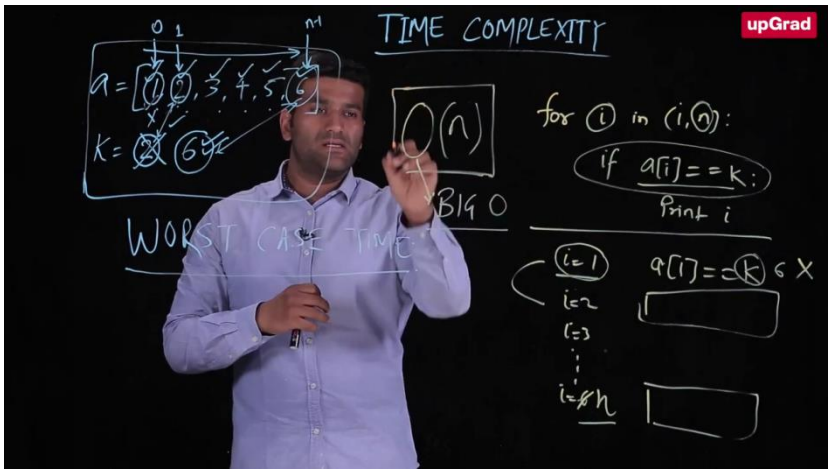


So, till now we got the basic intuition or idea behind how do you see that for these two example, this one is taking more time as compared to the previous example.

Now, let's just try to go and write the pseudocode or algorithm for this and then see how do we compare the same scenario there.

So, for this example in list finding the element, the pseudo code will be. Let's say there is a for I in from 1 to n.

Let's say, we are just iterating from 1 to n and we will just search if A[i==K], then print i. So, assume this is a very simple pseudocode representation of this example. And now, let's try to find out how much time this algorithm will take.



So, now here, let's us run the first in the for loop, let's go for the first iteration, which is I = 1. So now, for I = 1, I will be checking this condition for the first time, which is A[1==K] and K is 6.

So, as you can see here that A[1] is 1 and this is not equal to 6. So now, we will go to the next iteration, which is I = 2 and we will do this check operation again and similarly, we will go for I = 3 and we will go till I = 6, which is the last element or in the case of algorithm, this is the n'th element.

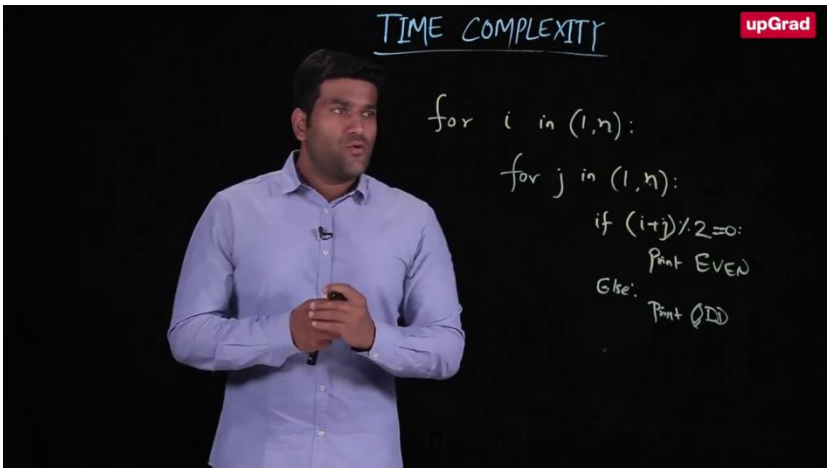
So, we are going till I = 6 or I = n. So, ultimately, what we are doing is we are doing this check operation, this particular check operation we are doing n number of times.

So, that's how we can say that this algorithm is taking order of n time, because we are doing this operation n number of time. And this same philosophy of running order of n time, we denote by this symbol order of n time and we call this O notation.

So, right now, do not be scared by what is this O. I will explain you in the further example, but for now just understand that for any time complexity of any algorithm, we represent in this kind of notation, where we have O of n.

So, this is the actual time complexity how much time it is taking. So, for this operation it is taking order of n time.

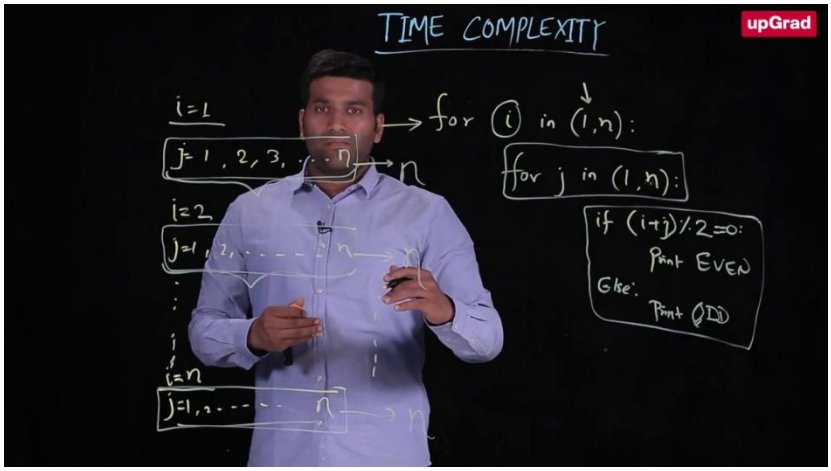
So, we have seen a very simple example where the algorithm was taking order of n time. Now, let's take slightly more complex example, where let's see how do we calculate the time complexity.



So, basically, I am just writing two for loop, for I = 1 to n; for J = 1 to n and what I am doing is I am just checking if I + J is even or odd. So, let's say if I + J is %2 = 0, then we can print even or else, we can print odd.

So now, if you see this algorithm, what do you think will be the time complexity of this particular approach?

So, so let's try to run the algorithm one by one.



So first, what we will do is first we will run for I = 1. So, for I = 1, we will run the next loop of for J in 1 to n. So, where we will run J = 1. Then we will do J = 2, 3, up to n we will run, right.

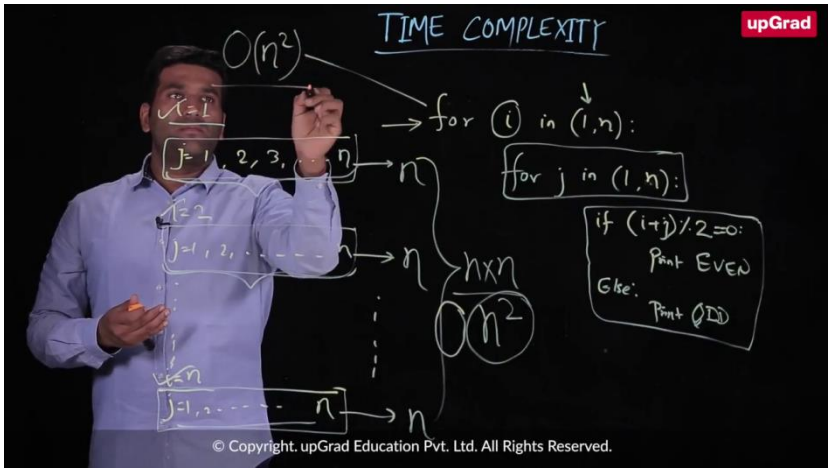
And we will do this operation of checking whether this number is the summation of I and J is even or odd that we will do n number of times, right, which is coming from this for the internal for loop.

Now again, the same operation we will do for when we increase the I value from 1 to 2. So, let's say when I = 2, again we will run the for loop from J = 1 to n, so which will be equal to J = 1 to n. And again, we will check this operation n number of time.

Now, similarly, when we increase I = 3, 4, and we can go up to I = n, which is the last number iteration for I and even for the last iteration we will run this for internal for loop n number of time, which is J = 1, 2, up to north.

Now, if you have to find out how much time this whole algorithm will take. So, what you can do is you can just sum how much time it is taking for all the check operation or all the finding this even or odd operation.

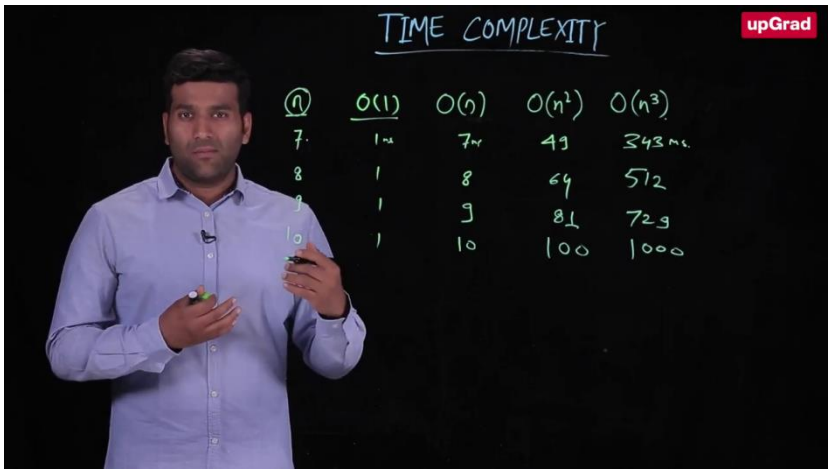
So now, you can see for this operation, it is taking n time. For this operation, it is taking n time and similarly, again for this, it is taking n.



And how many times this n internal J loop will come? It will come 1, 2, up to n time.

So, what we can do is we can do n x n, which is n^2. So, we can simply say that the time complexity or time algorithm will be taking will be order of n^2 and how do we denote is by just doing O of n^2. So, the time complexity of this approach will be order of n^2.

Now, I have already showed you how do you compute the time complexity of any algorithm. Now, to just give you understanding of why it is very important, we can take a small example for different time Complexities.



So, let's say we have time complexity like order of 1; order of n; order of n^2; and order of n^3. Let's assume we have n = 7, 8, 9, and 10. So, assume that you are comparing between different algorithm and you have to choose like which one you have to choose and which one is faster and these are the time complexity of the different algorithm.

So, let's see how it varies for different time complexity. So, we have taken an example number of input as 7, 8, 9, and 10. So for O of 1, it will take the same time, which is constant time. Let's say 1 millisecond.

Now, for order of n, this will take exact same time as the number of operations. So here, n is 7, so this will take 7 time; this will take 8; this will take 9; and this will take 10.

Now, for order of n^2, this will take 7^2, 8^2, And 10^2. So, this is the time complexity if we have these operations for n^2.

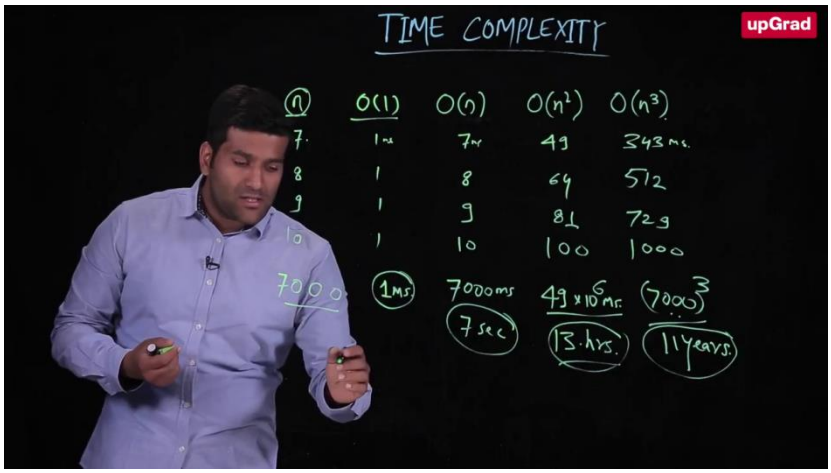
Now, for n^3, for 7, it will be 343; for 8, it will be 512; and for 9; it will be 729; and for 10, it will be 1000.

Now, let's just try to see if we are increasing the n, because right now you cannot, I mean you can see that the time complexity or time is not very huge, because this is taking 7 milliseconds versus this is taking 343 milliseconds.

So now, just for the explanation purpose, I have taken n sample size as very small 7, 8, 9, 10, but if you see the practical implementation industry, you will be having millions of records.

Now, if you take let's say example of any Flipkart or Myntra, we can have millions of product. So, if you have to search a particular product, you need to iterate for millions of rows or records. So, it can be 1 lakh or 2 lakh.

So, just to see how the time complexity will change for that, we can take example very huge.

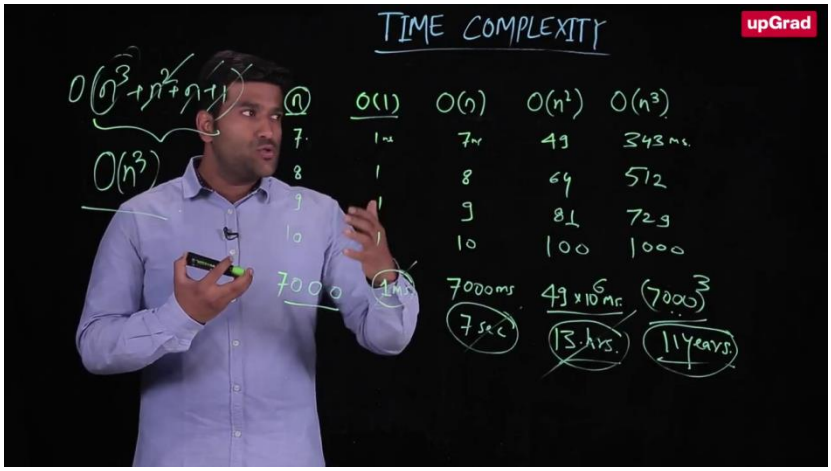


So, let's say n we can take as 7000 and we can compute the same number for n = 7000 and see how it varies.

So, for order of 1, it will take same time, which is 1 millisecond; for order of n, this will take 7000 millisecond, which is 7 seconds. This is 1 millisecond. Now, for n^2, this will take 7000 x 7000, which is 49 x 10^6 millisecond.

Now, if you convert this into hours, this will be roughly around 13 hours; whereas if you compute 7000 n^3, so that will be 7000 x 7000 x 7000, so that will be, if you compute this number, this will be approximately 11 years.

So now, you can just see where I am increasing the number, my sample size if it is increasing, which is the practical case in the industry. So, you can see for different time complexity how the time is changing. For O of 1, it will still take 1 millisecond; for order of n, it will take 7 seconds; for O of n^2 it can take 13 hours, whereas for n^3 algorithm, it can take up to 11 years.

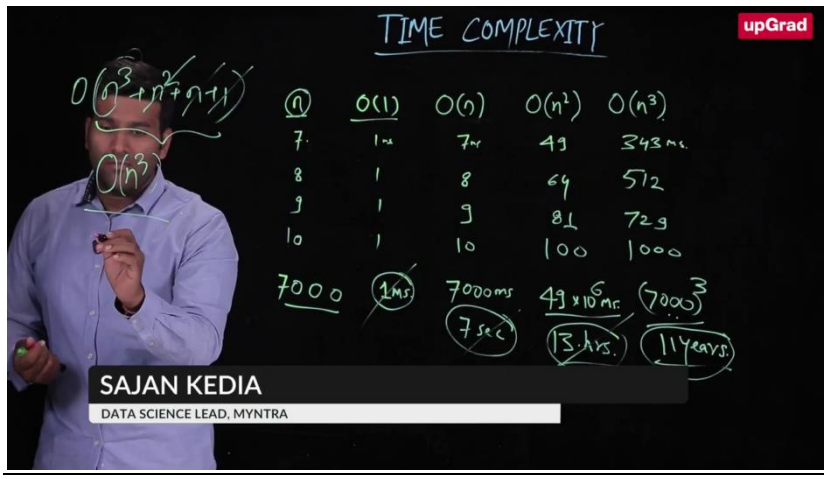


So now, assume if you have some function or for some algorithm, the time complexity is something like order of $n^3 + n^2 + n + 1$. So, in this case, as you can see here, even if we neglect this 13 hour, 7 seconds, 1 millisecond. This 11 year is a very huge number.

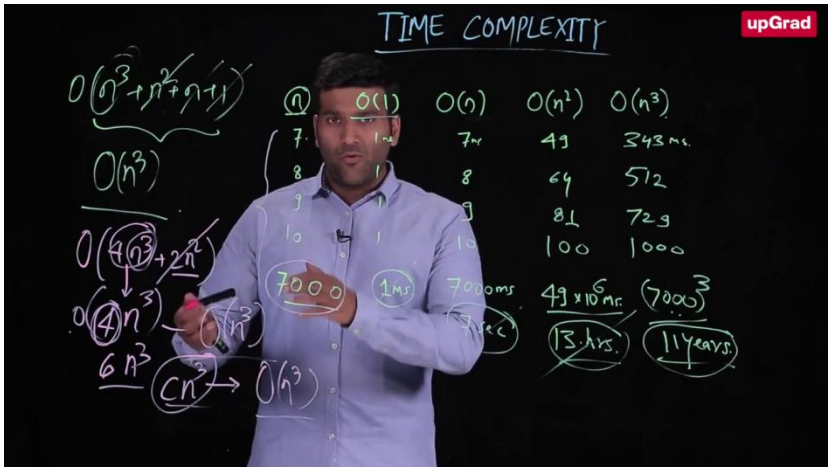
So that means, if you just remove these or if you just do take the smaller n^3, n, and 1 into consideration, this n^3 will be a very huge number. So ultimately, while finding the time complexity of algorithm, if you have this kind of functions, so directly you can say this is order of n^3, which is O of n^3.

So here, in this whole example, I have assumed that for one operation it is taking 1 millisecond of time, but in practical scenario, in industry you can have different computer or different computing machine where the processor is very fast. But still you can see whatever faster computing machines you get, but if your time complexity increase, then it can take really, really huge time.

So that's why, when you are writing your code or you are deciding the algorithm, you should choose algorithm for which the time complexity is very less. Because that can save you really huge amount of time.



Now, similar to this.



Let's assume you have another example where you have, let's say, 4 n^3 + 2 n^2. So, in this case, what will be the final time complexity?

So, as we have already seen here, that in front of n^3, n^2 will be a very small number. So, we can directly ignore this part. As we have seen here for n^3 and n^2 or for n. So, 4 n^3 will be very huge as compared to the 2 n^2 part. So, we can clearly ignore this part.

Now, for 4 n^3, here you can see 4 is a constant number. So, as you increase the n, so as we are increasing n from 7 to 7000. So here, just the n^3 part is increasing the time complexity, not the constant part.

So, that is why, even for O of $4n^3$, we can directly write as order of n^3, because here n^3 is majorly increasing the time complexity. So now, here instead of constant as 4, we can have any other constant. Like, we can have 4 or $6n^3$ or it can be any constant C.

So now, if you see here, if you just have 4 x n^3, which is 11 year, so you can see that this constant is a very small number in front of the n^3. So, that is why this order of Cn^3 will ultimately, we can call it as order of n^3.

So, for this part, we are not providing any proof here. We can share additional resources in the text, which is there in the module.