Programming & Data Structures
Dr. P.P. Chakraborty
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur
Lecture 19
Asymptotic Growth Functions

We are studying the complexity of algorithms and in order to analyze an algorithm and we need to find out exactly how to compare two different algorithms because our interest is to find out the best possible program which will work in the most efficient way. So in order to do that we need some formal mechanism to analyze and compare between two programs or two algorithms and we discussed in the previous class that the complexity measures are usually the time and the space required by a program. And the first and most important of both of them, amongst both of them is that of time unless space is exponential, time is absolutely the most important criteria.

Therefore we will study formal mechanisms of how to measure the time required by an algorithm or by a program. The first thing we encounter was that having pure cpu time is not a good idea because cpu time will change depending on the type of machine, the instruction set it has and it becomes impossible to compare between different programs. Secondly, the same program on the same computer for different inputs will take different amounts of time. So we must have a mechanism to really compare between two algorithms or two programs which is not only reasonably machine independent but also has got which we can measure the same program using the same structure of our analysis for various kinds of inputs.

Now once we go for machine independence and for input size independent or input independent analysis we have to make certain approximations. The first is we cannot go in for a CPU time and second is we have to, we cannot say exactly for what input what the amount of time that will be taken. Now when we are measuring the complexity of algorithms, we have to remember that whenever we try to analyze an algorithm we have to be very careful about what input is coming to the problem. For example if you have to find this, if you have to write a sorting program of n numbers and you want to analyze it, you have to know very carefully that if you are going to have only 10 numbers to be sorted then obviously you will have to, if you are going to have a million numbers to be sorted the strategy may be totally different. And if you do not know the size of the numbers, it may range from 100 to 20 million then again the strategy will be different. If it is known then it will always range from 200 to 500 then the strategy may be different.

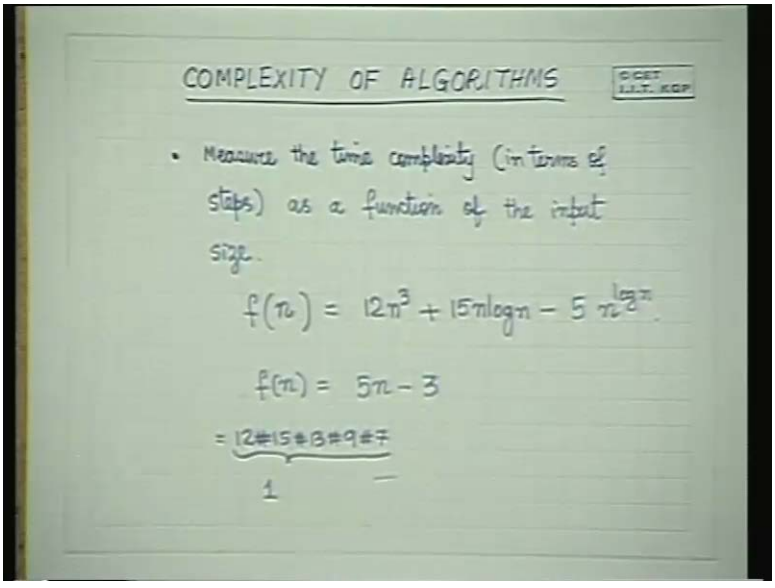
Now when we know that programs are to be written for small size of input, for example we have to multiply matrices which are only 10 by 10 and 2 matrices are to be multiplied. In present technology of computers, the time required is not very large unless this matrix multiplication is required to be done million numbers of times. So we need to compare and analyze these when actually the time is very large, the real time of using this operation is very large.

In such situations and to make it machine independent and independent of the size of the number of inputs or the variation in input, we make certain approximations. The first approximation that we make is that we assume a model of computation and in that model of computation, we give for every operation which can be done on fixed size data, we give it some unit time. And we say addition is done in one unit time, addition of two fixed numbers is done in unit time, fixed size numbers and fixed size means size for which the size is a constant, we know there is the limit on the constant say some maximum will be say 10 bit numbers or maximum will be 15 bit numbers. Then there is a limit to it, you cannot say any arbitrary sized number and all additions of arbitrary size numbers is constant. We will come to this issue because this is very important.

So we assume, we make the assumption that fixed size operations take unit amounts of time and we decide which takes what amount of time. Based on this unit amount of time we try to find out, what is the time required, how many units of time does this particular program or function take. And in terms of units of time also, we cannot say 20 units or 30 units because we want to make it independent of the size of the input or the variation in the input.

Now we cannot say for this input it will take this, that input so we cannot enumerate out how many steps it is going to take for what input because the inputs may actually the number of inputs itself may be infinite. So we have to make it a function of something of the inputs and one important criteria which is being found to be most useful in analyzing the complexities of algorithms is to make it a function of the input size. So when we measure the complexity of algorithms, we do two things we measure the time complexity in terms of steps as a function of the input size.

(Refer Slide Time: 10:08)



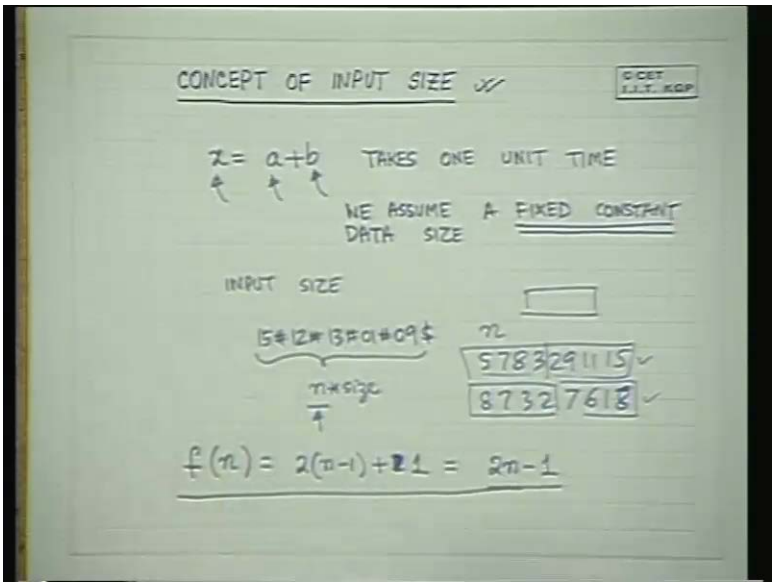
Very informally we say that for some problem if the input size, see the input size may be parameterized by 1 parameter or 2 or 3 or whatever because if there are 5 inputs may be

we can give 5 different parameters for 5 of them but then you can combine all of these 5 and make it into one size also. So under normal circumstances you will come to other, we say f (n) is equal to 12 n cubed plus 15. We will find out the number of steps, n log n minus 5 n to the power log n, some function we have got. And this we call the time complexity function of this particular problem where n is a measure of the input size. Now see in our sorting example or in finding a maxima of n numbers, we found that f (n) is equal to some 5 n minus 3 or something like that where n is the number of numbers which are, of which maximum is to be found. Now there is a very important thing which has to be understood about this n. You can say that n is the numbers of number which are input.

Now since this is a mathematical equation and people say this is the function of n, so people say how did you say that there are n inputs. I would say that there is only one input, you just combine all the numbers and make it into one number and say for example I will say, I will give you a character string which is one input which is like this 12 hash 15 hash 13 hash 9 hash 7 and you read this is as a character string. And I will say you have got one input only and then you break it up, find out and sort this and return one character string. So does that mean that the input size is one? Then for the same problem if you write, if you just take the input in a different way, you will get for one case you will say my input size is n because I am taking n numbers. In another case I will say my input size is one because I am taking one number and here it will be 5 n minus 3 and the other case I will say what is there, it is just a constant because it is just dependent on the length of the input. The complexity of sorting if I take it as a character string will be proportional to the length of the input, alright. So the concept of input size has to be understood very very carefully, so that we are not misled when we are doing the analysis.

Let us remember that each of our operations, when we have said that an operation takes one step that meaning is defined this way that for fixed sized numbers, that means fixed size means there is a maximum limit on the size of inputs to an operation. For example if I say addition x is equal to a plus b takes constant time, takes one step, one unit time then there is an underlying assumption that the size of a, size of b and size of x all three of them are bound by a fixed constant alright, all three of them are bound by a fixed constant.

(Refer Slide Time: 16:47)



Therefore for all your operations there is a fixed constant. Now if you understand the computer architecture carefully, this fixed constant is usually the size of your data. You know when you say a 16 bit computer, with the data size as 16 bits, data unit is 8 bits so that is the limit. It cannot be variable that way though if you say addition of 2 numbers takes constant time then addition of 2 numbers of each of one million size will also be assumed to take constant time that will give you a wrong analysis. So when we say this takes constant time, we assume a fixed constant data size. This will be represented in log n bits, suppose a number 25 say this 25 will be represented, 16 numbers can be represented in 4 bits, 25 numbers can be represented in 5 bits. Similarly there will be a maximum limit; in any computer you will see there is a maximum limit on the size of a number.

Therefore now when I talk about input size, I know that there is fixed maximum constant on the size of one input, alright. So the size of that input, input size that function that n is a function of that fixed constant. So, now if I say my n numbers will be read as 15 12 as a character string and this is how it will be read as a character string and I will separate it out. I know there is a fixed size on my numbers representation, this will be some n into that size so that will be the input size n. So this way we will get even whatever way you do it, the size of your input will be a function otherwise it will be a constant for all algorithms or it will be of the function of the same n for all algorithms, it cannot be a constant in one case and a variable in another case. So we have to be very clear about this input size. There are other issues about input size which we shall come to a bit later on.

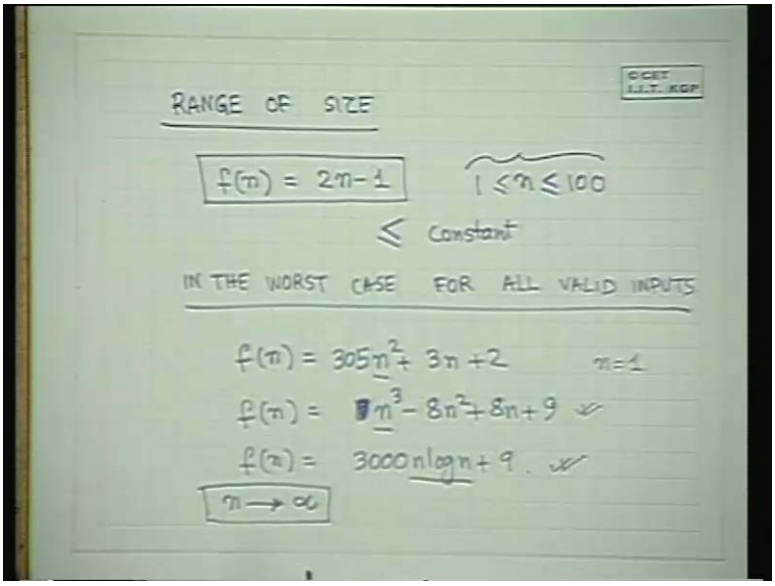
So that way the problem of adding up two arbitrary size number is not constant time. When you say I will give you two arbitrary size numbers and you give me the result of that then you have to tell me what is that n. What is, so I will now say the complexity of adding two arbitrary size numbers is f (n) where this n is a function of the maximum constant size on which you can really add two numbers and then you can write a program

in which you will say for example if I give you 5 7 8 3 2 9 11 15 one number is like this. So its 5 8 7 3 9 2 1 1 1 5 and I ask you to add it with 8 7 3 2 7 6 1 8. Then what will we do? We will have a fixed size, we will break up this number into so many numbers of that size, we will break up this number into so many numbers of that size and then we will do our addition. We will add this part with this part, we know this part, this is our fixed size so this takes constant time and then we will take a carry and we will store it in another such fixed size variable.

Then we will add this and add the carry and we will do normal addition like we do bitwise. So now if there are n such, so if now this is n my addition operation will be proportional to n such additions. It will be 1 plus 2 plus 2 plus 2 because one addition at the base there is no carry; otherwise I have to do one addition with itself and one with the carry. So it will turn out to be 2 n minus 1 plus 2 sorry 2 n minus 1 plus 1, this is 1 and therefore each of them 2, so this algorithm for multiplying 2 arbitrary size the numbers to add two arbitrary sized numbers my complexity will be 2 n minus 1.

So this is how we make a realistic analysis, we have to be very pragmatic and you have to know clearly what assumptions you have made when you make, when you analyze this otherwise you would have simply said multiplying any size number is a constant because my addition of two arbitrary size numbers is a constant because my addition operation is a constant. This is not a correct analysis. So you have to be very pragmatic about your analysis.

(Refer Slide Time: 21:45)



So the concept of input size is important and secondly you have to know the range of your operation. That is I know that my input size will vary from 10 to 20, if I know then I know that is actually a constant. For example I will have to sort 100 numbers only effectively you know you have to sort a maximum of 100 numbers, so your complexity is a constant time. Even if you say that my addition of 100 numbers suppose I give you 100

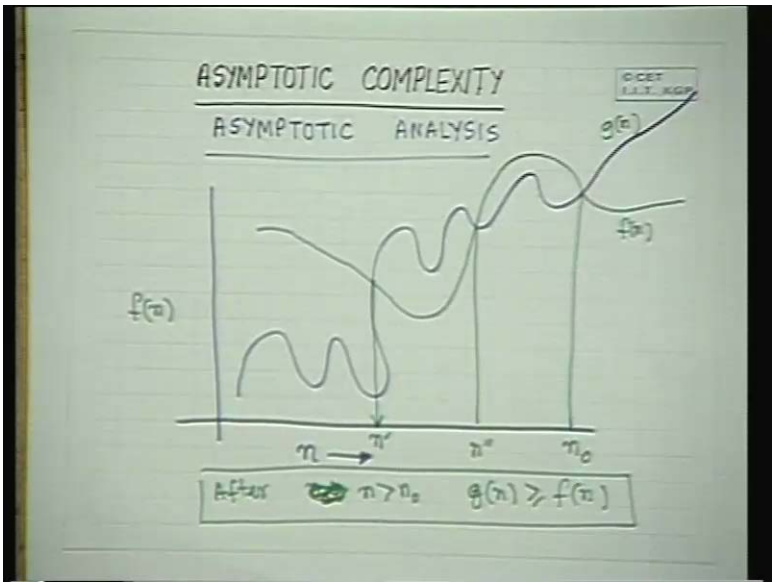
number and ask you to add it, you can say that obviously my for addition of n numbers I have to make n minus 1 additions add this, add this, add this, add this and store it in a temporary variable some n minus 1, 2 n minus 1 additions. But if I know that n is less than equal to 100 then obviously this is bound by a constant. So my total complexity is no more a function of n. It is a function of n in the range from 1 to 100 but it is also upper bounded by a constant, I know that I will not take more than so much time.

So I have to analyze it within this range, it is not difficult to analyze in small ranges because usually the time required in such small ranges is low. The difficulty comes when you have to analyze it in the worst case for all inputs, for all valid inputs. That is how will you add n numbers where n can be anything then this function becomes very important. So if I have one which is going to do sorting in an another which is going to do sorting in and the third which is going to do sorting in… Sorry this is 5 n square minus 3 n plus 2, this is n cube minus 8 n square plus 8 n minus 9, this is 3000 n log n minus 9. Now when n is 1 this will turn out to be very good. You can make it, let's make it positive otherwise this will look very odd. So when n is 2 then this will become possibly the smallest. Let us make this also bigger 305 n square.

Now when n becomes slightly more, this will become better but we are analyzing for all valid inputs. When n tends to become very very large then this will be best. So depending on range of the inputs, the complexity becomes important which works better. Secondly these constants are important because these constants have come out of a lot of approximations. It is the highest order term which really dictates matters when n tends to large. And why are we interested when n tends to large? Because these are usually positive growth functions, when n becomes large this suddenly cannot become zero that is as when the input becomes larger and larger, the time take by the algorithm become less and less, you will really find such problems which are provided to you.

So these are usually positively growing functions and the time really becomes a crunch when n tends to infinity. And that is why we try and see how to compare these, specially in worst case for all valid inputs when n becomes large do they have a proper behavior. That sort of analysis is called asymptotic complexity analysis or it is also called asymptotic analysis. That is suppose we have a function, after having defined the input size n.

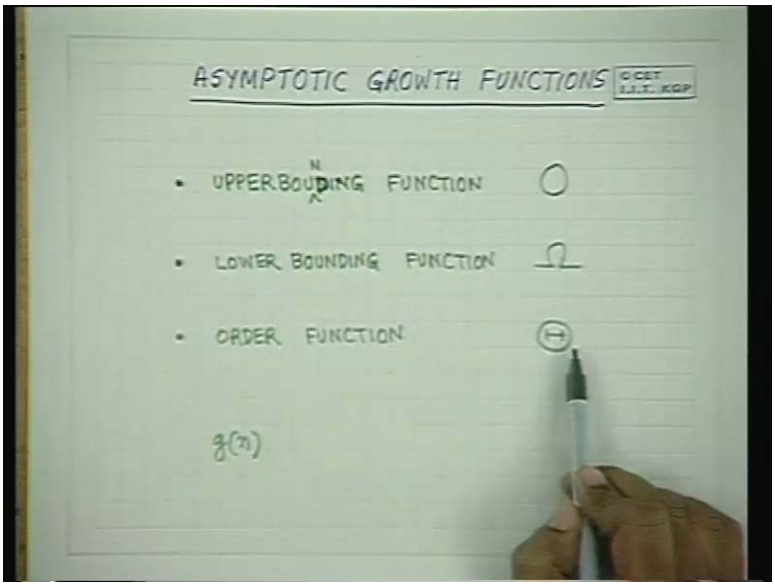
(Refer Slide Time: 24:50)



One of them is like this. The other one, now based on these two we cannot differentiate between them. In this range, in the range up to this say n dashed, this one is worst than this, this is better, this is f (n). Up to this range this is better, again in this range this is better and if through out the ranges they start flipping this is better and that is better, this is better and that is better then you cannot say anything about surely about somebody being better than somebody else. But if after a certain point, one of them is definitely better than the other beyond this which is called the n_0 point then we say that this let us call this f(n) and this let us call this g(n) we say that f(n) is less than g(n) asymptotically. That is after n is not greater n greater than n_0 and usually these are positively growing functions you start do not, they are not like sine waves and all like that continuously going up and down and up and down because we are measuring time and time is usually expected to increase with more increase in input size. So after n this is said to be greater than this. This is how you compare them asymptotically.

Now to compare these asymptotically we have certain formal notations and definitions to use. And we have to be careful about when we say one is better than another and less than another. In such a situation when this is always better than this, we say this is something like this but even this may not happen. This function we will come to this example little later on. So we have what is called the concept of asymptotic growth functions. And here we define three kinds of asymptotic functions one is called the upper bounding function which is used, called the big O notation.

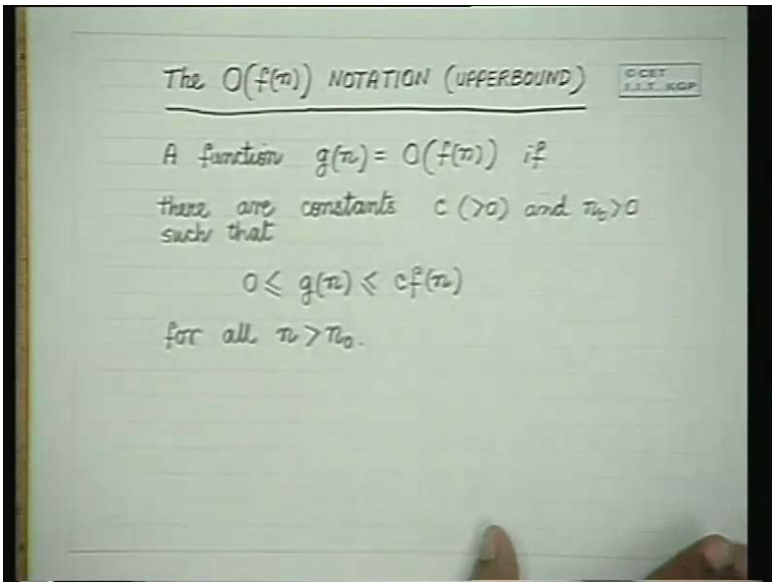
(Refer Slide Time: 26:38)



The second is sorry the lower bounding function, the third is the order function. These are the three symbols or notations used for. This is given g(n), we try to find out another function which upper bounds this that is beyond a certain range that is when n becomes very large this function will be always greater than this function. Lower bounding function is then this function will be less than this function beyond a certain limits and order function is that both are equivalent in some sense.

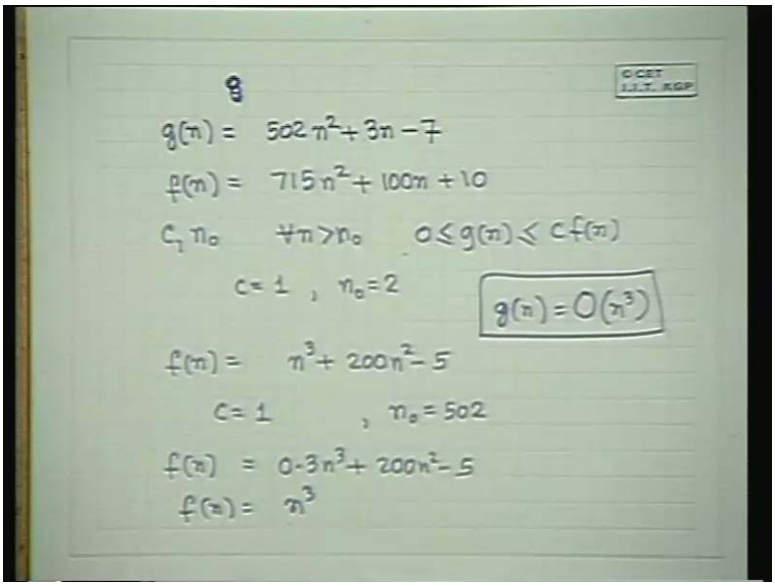
Now we will see their definitions and understand their meanings. The o n notation, the o f(n) notation which is called the upper bounding, asymptotic upper bounding function.

(Refer Slide Time: 28:32)



A function g(n) is equal to o of f(n), so given two functions f(n) and g(n) both asymptotically positive that is beyond a certain n both of them are always positive, they never become negative. So we always work on asymptotic positive function, we don't work on anything else. When g(n) is equal to o of f(n) that is g(n) is upper bounded by f(n), if there are two constants c and n_0 both of them positive such that 0 is less than equal to g(n) is less than equal to c into f(n) for all n greater than n_0. A function g(n) is upper bounded by a function f(n) if there are constants c and n_0 both positive such that for all n greater than n_0, g(n) is greater than equal to 0 obviously positive function and it is also less than equal c into f(n).

(Refer Slide Time: 35:06)



Let us consider one or two examples. We are finding out g, let g (n) be… Any questions? This is an equation, so there is no problem. This is not an assignment; this is an equation so you write this side or that side it does not matter. Upper bounding, o g(n) is equal to f(n) no no no this is a notation, it's some notation. Now you can use it this way or that way, the notation is this, g (n) is upper bounded by f (n) this is the notation. So this is this, this is g (n), this is f (n). So what we have to find? Two constants c and n_0 such that for all n greater than n_0, 0 is less than equal to g(n) and less than equal to c into f(n). Here the choices are very simple and obvious; you choose c equal to 1 and n_0 equal to anything, 2. You will satisfy this condition, so you will say that g (n) is upper bounded by f(n).

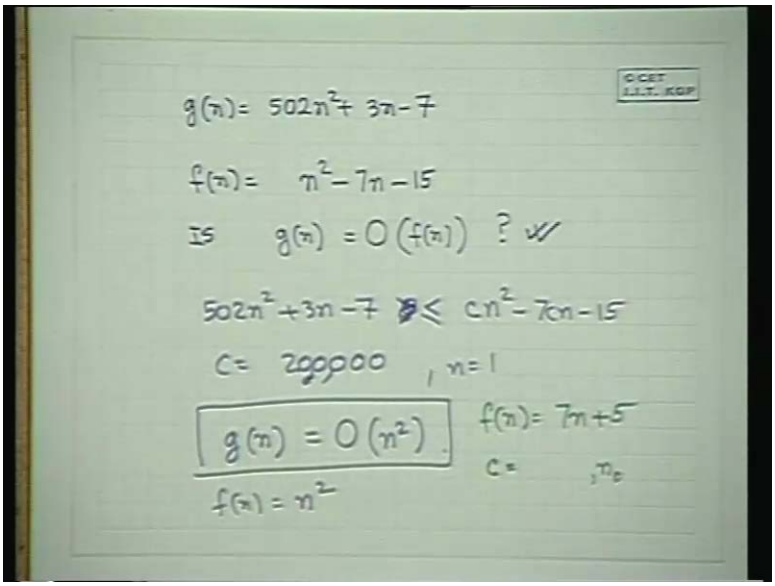
Let us take another f (n). Here is f (n) an upper bounding function of g (n). When n tends to large because when n tends to large, n cube is going to become much more than 500 n square but to formally prove this, this you have to find a value of c and n_0, any constant c and any n_0 and choose c is equal to… What c will you choose? You have to choose the c and you have to choose the n_0. Choose c equal to 1, n_0 equal to 502. Now beyond that obviously this one is going to be larger than this one. So you will satisfy you have, once you find these two you may be sure that beyond n this function is going to beat this

function. Now even if there was a constant here, now let us have a look. If f(n) was equal to 0.3 n cube, even then this would be an upper bounding function of this because you would have found some other c and n_0 that way this asymptotic analysis takes care of the speed of the machine saying that okay these constants could have been done faster in some other machine but asymptotically this if it is asymptotically an upper bound in one system, in one machine it will remain an asymptotic upper bound in another machine also because two machines will differ amongst each other by a proportion only because each operation if that is done in one step this may be done in 0.0001 steps.

So the original function, if this is an n cube plus b n square plus c n will only differ in a, b and c the other things will remain constant because we have taken it in terms of input size which is independent in both of them. Now if for one particular a, b and c you can find constants to make it an upper bound, for another value of a, b and c you will find some other constants, so which effectively means that when n tends to large, a function will be… So you can say that this is if you choose f (n) is equal to n cube also, it will still remain an upper bounding function. So that is why you can say g (n) is equal to upper bounded by n cube. You will see in textbooks, a notation like this comes. Why does a notation like this come? Because these constants become unimportant. If something is upper bounded by 0.3 n cube plus 200 n square minus 5, it will be also upper bounded by this.

So we leave only the highest term to compare between them. Finally suppose g(n) is equal to 502 n square plus 3 n minus 7 and f(n) is equal to n square minus 7 n minus 15. Is g(n) upper bounded by f(n)?

(Refer Slide Time: 38:24)

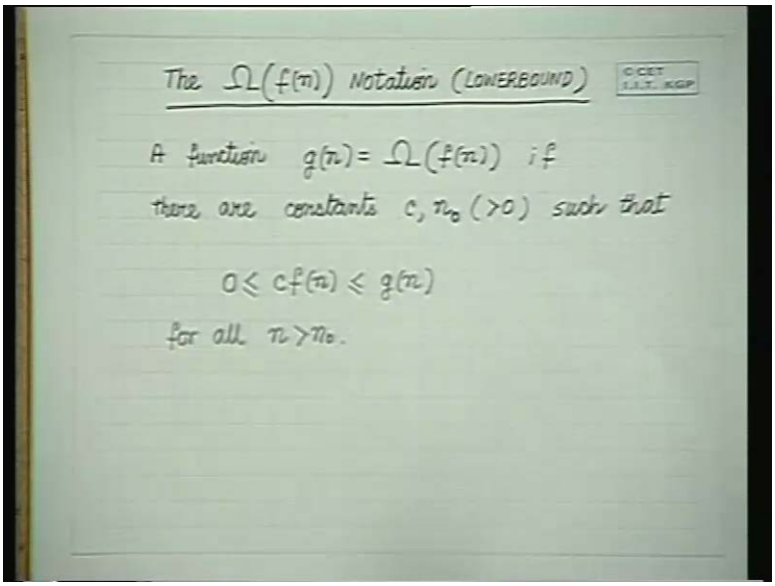


What is the answer is g(n) upper bounded by f(n)? Yes or no that means you have to prove that is 502 n square plus 3 n minus 7 greater than sorry less than or equal c n square minus 7 c n minus 15 for some beyond some n_0. You will get it. Why don't you, why

don't you choose c equal to 20? Then from n equal to 1, it will start becoming greater. Just have a look, choose c equal to 20,000 sorry, we have chosen 200,000 and it is 2 lakhs. Then for n equal to 1 this will be greater, for n equal to 2 it will continue to be greater. So g(n) is upper bounded, you will be able to find 2 constants and therefore you can say g(n) is equal to upper bounded by, it is upper bounded by f(n) and if f(n) was equal to n square also it would have still be upper bounded by f(n). But if I choose f (n), suppose I choose f (n) is equal to 7 n plus 5 then I will be able to get a constant c, I will choose c to be 200 lakhs may be but I will not be able to get an n_0 beyond which this function will be less than equal to this function always, I will not be able to get.

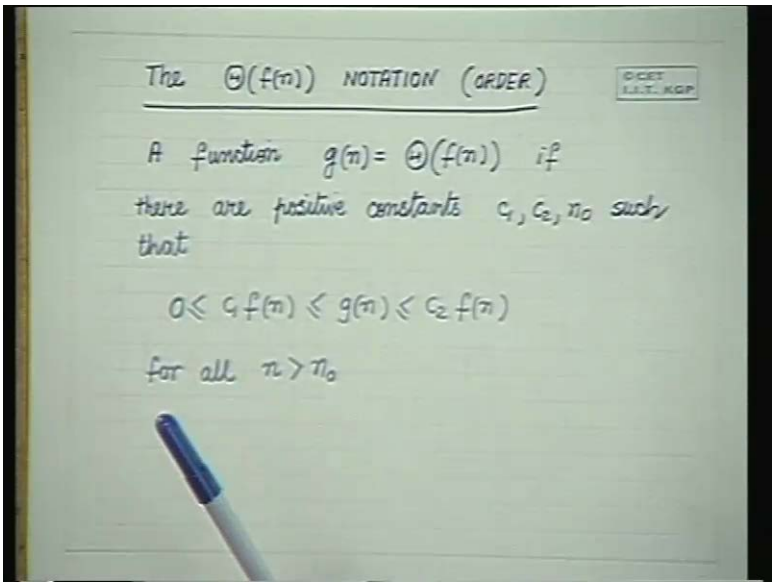
So this f (n) is not an upper bounding function of g (n). So upper bounding functions give you a measure of whether this algorithm can be, if there is a function whose growth is greater than or equal to the growth of this function when n tends to large. So, if you have understood these upper bounding functions clearly, the equivalent notation is a lower bounding function which is just the reverse notation.

(Refer Slide Time: 39:20)



A function g(n) is lower bounded by another function f(n), if there are constants, positive constants c and n_0 greater than 0 such that 0 less than equal to c(n) less than equal to g(n) for all n greater than n_0. So this is the other one and you can say that if f(n) is upper bounded by g(n) then g(n) is lower bounded by f(n) because if you can find the constant from that side, you take one by that and you will get the constants on this side also. You will find constants on this side. The next is the order notation which identifies equivalent growth functions. Suppose I have got f(n) and g(n) such that f(n) is upper bounded by g(n) and f(n) is also lower bounded by g(n).

(Refer Slide Time: 42:02)



I will give you an example. Then you will get f(n), you will also get both you will get. So you can see that these asymptotically the growth of these functions are equivalent they only differ by a constant in the sense that if you could put these algorithm on to a faster machine, this may run faster than this. It is just the speed of the machine will make the difference, even for all n beyond a certain range this will run faster. On a slower machine this will run faster, if this works on a slower machine this will run, if this runs on a faster machine, this on equal machine may be this will run faster. So when you have two functions which are equivalent in growth then you characterize them by the order notation which just combines the g (n) and f (n) which says a function g(n) is of order f(n) if there are positive constants c_1 c_2 and a n_0 such that g(n) is lower bounded by c_1 f(n) and upper bounded by c_2 f(n) for all n greater than that n_0 beyond that n_0.

Once you can get an order notation, you have characterized the growth of that algorithm. You say algorithm is order n square then you know that this is exactly what it is going to take. When you say it is upper bounded by n square, it may take log n time also. When you say it is lower bounded by log n, it can take exponential time also but when you say that it is of order n square then you have accurately characterized it both on the basis of its lower and upper bounds. That is why you will notice when you analyze an algorithm asymptotically you try to find out what is the order of the algorithm. And when you are designing algorithms or data structures, we would like to design those whose orders of complexity are minimized that is what we mean by designing time efficient algorithms, algorithms whose orders of complexity are minimized.

So next once we have understood what are these asymptotic growth functions and got used to this notation, we will see how to find out the orders of complexity of some of the algorithm that we have designed and what they look like, how they compare with each other. So we will see that in the next class.