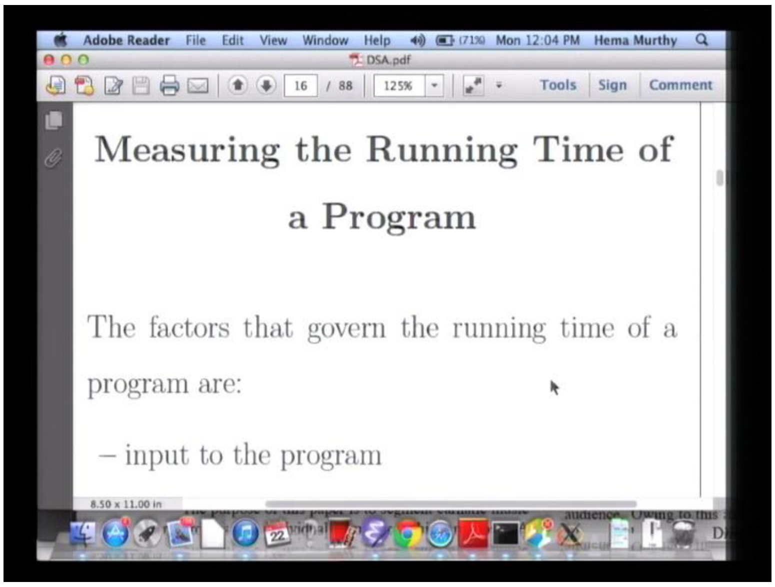
Programming, Data Structures and Algorithms
Prof. Shankar Balachandran
Department of Computer Science and Engineering
Indian Institute Technology, Madras

Module - 2
Lecture - 25
Measuring running time of a program
Size of input Big-O and other notation
Sum rule and product rule

In the previous lecture, **w**hat did we talk about? We talked about we did a recap on the basic building blocks of programs. And then we said how we can use. We kind of, informally looked at the arrays and recursive data structure. And we looked at some programs which can operate on these 2 data structures. And already we also looked at another example of prime classification. And we said how do we know, how efficient the programs that we write are. And we just did; we informally said why checking up to the square root of n for prime classification is more efficient; primarily because it runs fewer times through the loop. That is what we saw. Now, what we will do is in this class we will look at measuring the running time of a program.

(Refer Slide Time: 00:54)



Measuring the Running Time of a Program

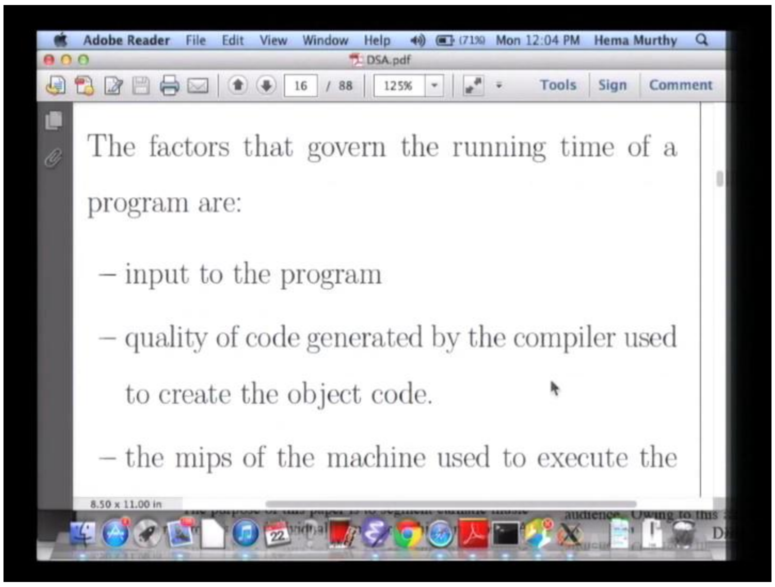
The factors that govern the running time of a program are:

– input to the program

And when we measure the running time of a program, we will see how we can formally referred, define the running time of a program in a sort of a machine independently.

What do you think that governs the running time of the program? Surely, the input to the program matters.

(Refer Slide Time: 01:10)

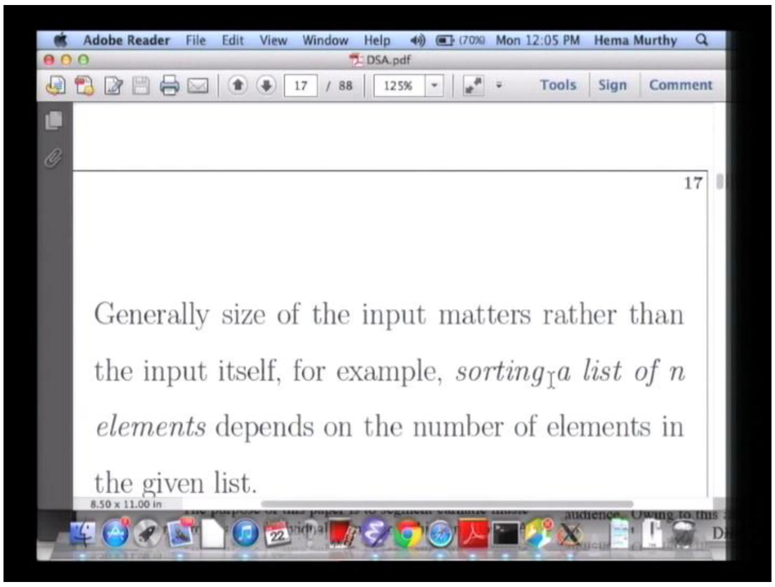


The factors that govern the running time of a program are:

- input to the program
- quality of code generated by the compiler used to create the object code.
- the mips of the machine used to execute the

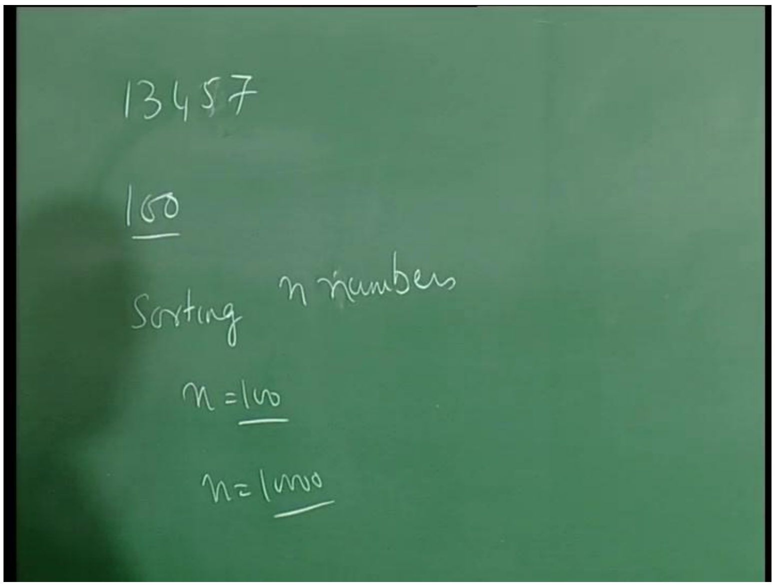
For example, if I am doing prime classification again, if I am giving a very very large number, then it is going to take a longer time. Whereas, if I am testing for the number 2 or 3, it is going to be very fast. An other issue could be that it could also be the compiler. If I have very bad compiler, the compiler does not generate code which is very efficient. Then, it is possible that there can be difference in the running time of the program. And of course, finally how good am I using a i7 or am I using an ordinary Pentium machine and so on and so forth. The MIPS of the machine can also decide how efficient your program is going to be...

(Refer Slide Time: 01:52)


Generally size of the input matters rather than the input itself, for example, *sorting a list of n elements* depends on the number of elements in the given list.

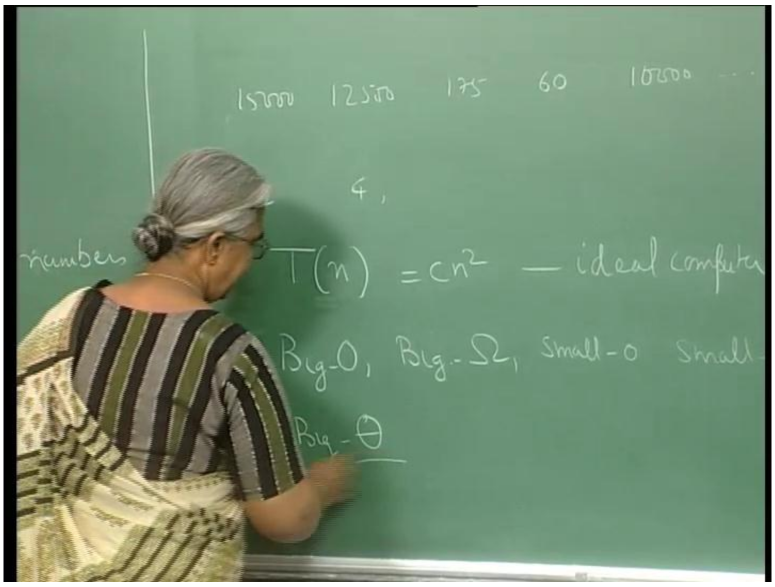
Now, what we would like to do is clearly when I write an efficient algorithm, we should, you know today you have i5, you have a i6, i7 and so on and so forth. So, can we do in a kind of a machine independent way is the question. How can you perform complexity analysis or analyze the running time of a program in a machine independent way. And sometimes, for example, depends upon the; the another input is we talked about the input matters. Let me give you an example.

(Refer Slide Time: 02:29)



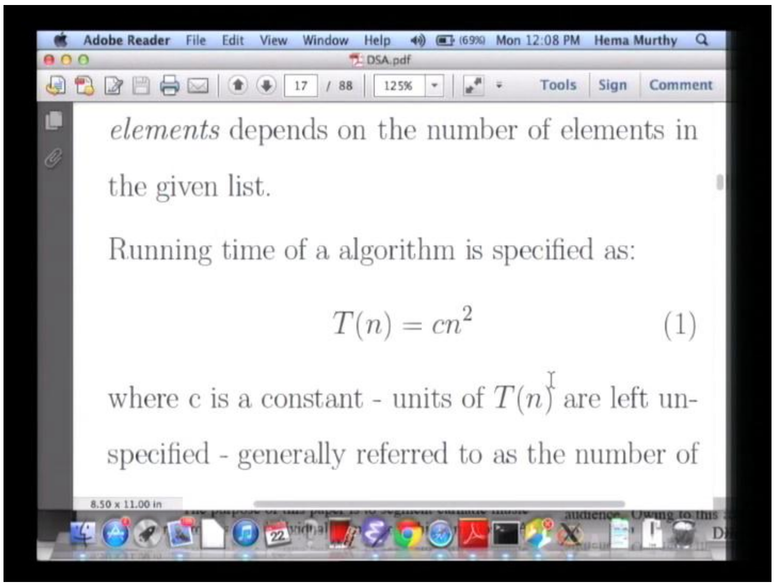
You are looking at the prime classification problem clearly to determine whether number like 13457 is a prime or not. It is going to take much longer than determining whether the number like this is prime or not. So, the size matters; the size of the input. An other situation could be; if for example, if I am looking at sorting n numbers. You have already done some basic sorting algorithm in the programming course. So, you know that if n is equal to hundred, then the number of computations required is going to be smaller than n equal to, let us say, ten thousand for that matter. So, what is interesting is if I take these sorting of n numbers, let us say I am looking at sorting of ten numbers only.

(Refer Slide Time: 03:22)



And let us say these ten numbers are, you know, 15000, 12500, 175, 60, 10000 and let us say something like this, even if the size of the number is large. On either hand if I have, you know 10000 numbers to be sorted. And numbers are 2, 4 or they are all single digit numbers are; you know double digit at most and no more than that; repeated number so on and so forth. Clearly even if the contents, the numbers are large and in the other case the numbers are small, clearly we look at sorting program. It simply does not matter what the value of the number is. But, what simply matters is the number of elements that you have to solve. Number of elements; if I am sorting, for example, I am going to use an array for sorting. And it is going to depend upon whether n is hundred or n is equal to ten thousand. So, clearly what is the fundamental difference here? The size of the input, rather than the input itself, matters in the sorting program.

(Refer Slide Time: 04:39)



elements depends on the number of elements in the given list.

Running time of a algorithm is specified as:

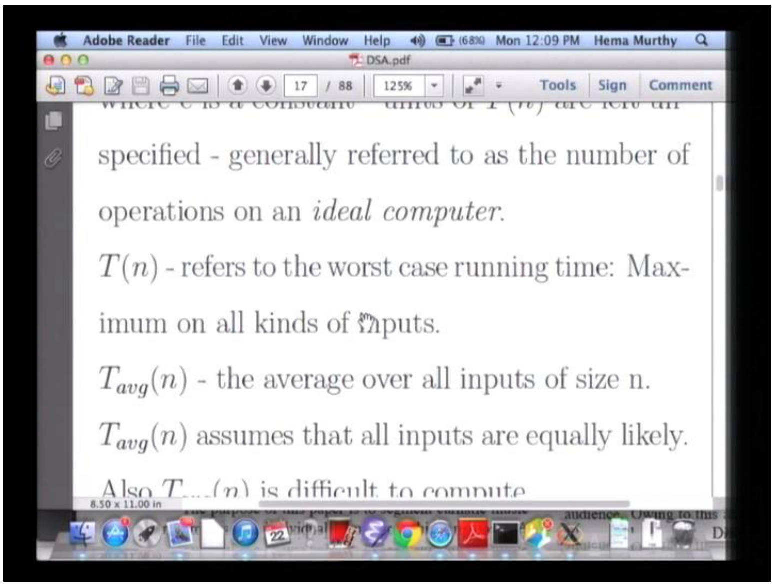
$$T(n) = cn^2 \qquad (1)$$

where c is a constant - units of $T(n)$ are left unspecified - generally referred to as the number of

So, this is essentially what we are going to look at and how do we define running time. So when we define running time, running time of an algorithm is defined in terms of n; where n is either the input of the number or the number of elements in your input like the sorting program. And this is written as C into n squared. What is C now? For example, what is it telling me? C is a constant over here.

And generally what we do is we leave the units of T of n unspecified. And why do we do this? The primary reason we do this is the following. So, what we do is the units of T of n are kind of kept unspecified. And why do we keep the units of T of n unspecified because we do not want to say, we do not want it to be defined in terms of the MIPS of the number. That is, how many million instructions can it perform. But, what we do is we generally refer to these units as operations that can be performed on an ideal computer. We will talk about what we mean by this in a little while.

(Refer Slide Time: 05:59)


specified - generally referred to as the number of operations on an *ideal computer*.

$T(n)$ - refers to the worst case running time: Maximum on all kinds of inputs.

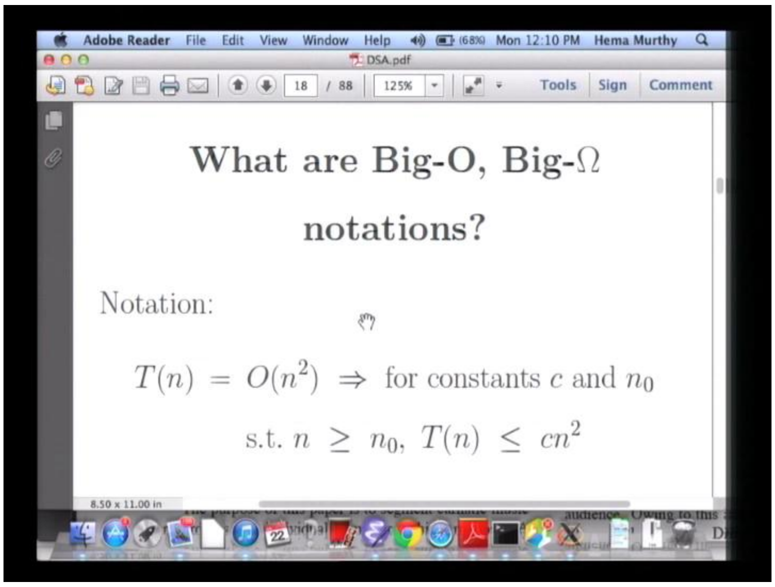
$T_{avg}(n)$ - the average over all inputs of size n.

$T_{avg}(n)$ assumes that all inputs are equally likely.

Also $T_{avg}(n)$ is difficult to compute

T of n generally refers to what is called the worst case time complexity. That means; what do we mean by worst case running time? We are saying maximum over all possible inputs. That is generally the meaning of T of n. And T average of n is you are looking at the average over all inputs of size n. And what it means is that generally T of n is little bit difficult to compute. This will be done in your algorithms course.

(Refer Slide Time: 06:29)

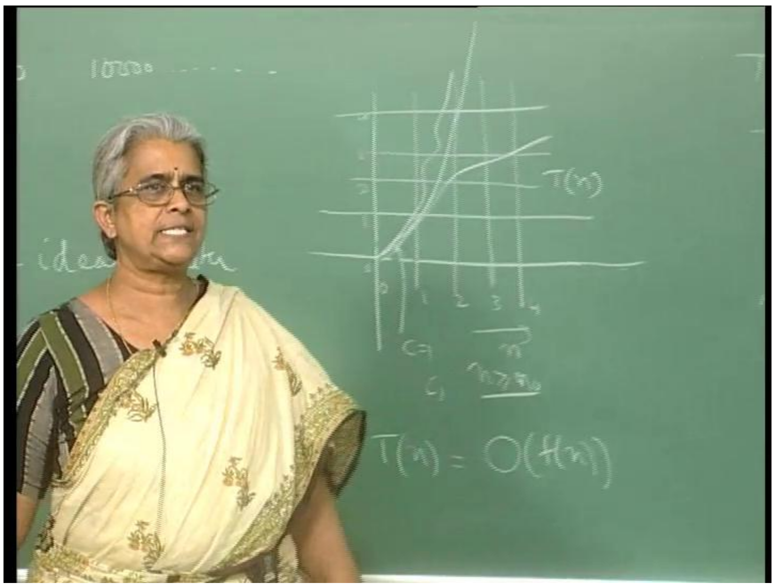

What are Big-O, Big-Ω notations?

Notation:

$$T(n) = O(n^2) \Rightarrow \text{for constants } c \text{ and } n_0$$
$$\text{s.t. } n \geq n_0,\ T(n) \leq cn^2$$

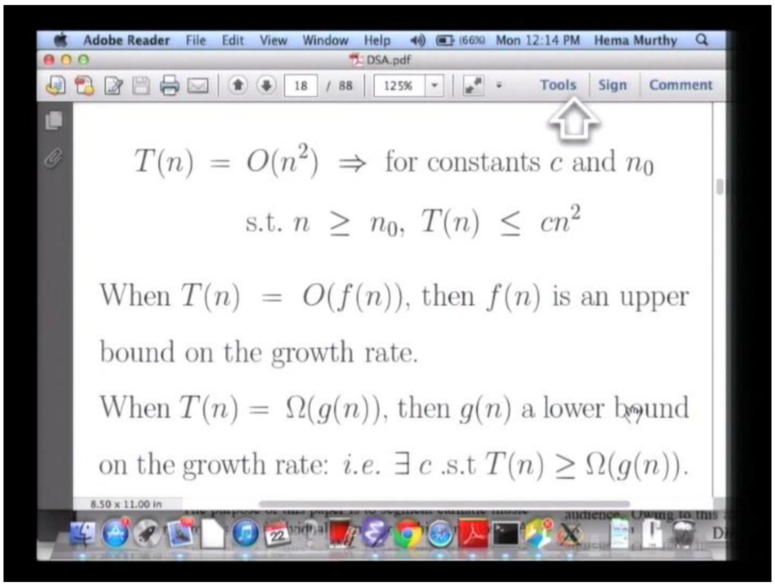
Next, what we will do is I will just give you a very brief overview of the kind of notation that I am going to use as part of the data structures course. You will use more formal notation. They are what are called, what we need as part of this course. They are something called a Big_ o notation, a Big_omega notation. And there is a small_ o notation and then there is a small_ omega notation and there is also a Big_ theta notation. All these details you will learn in the next course on algorithms. So, what I am going to do is I am just going to talk about the Big_o and Big_ omega notations, which are useful for the analysis in this particular course. So, what is it mean? T of n is equal to big O of n square means what? There are constant C and n naught, such that for n greater than or equal to n naught, T of n is less than or equal to C n squared. What is the meaning of this now?

(Refer Slide Time: 07:36)



Let us take n square and let us take c is equal to one for that matter. Then, what do we have? Zero, then we have one and let us say this is 0, 1, 2, 3, 4 and so on. Then if this is n over here, for 0 it is 0; for one it is one; and for 2 it becomes 4. So, this is let us say 0, 1, 2, 3 and 4, for example. It becomes like this. So, basically the curve is not linear, but it is kind of quadratic; the way this function grows. What is the meaning of it? If I have T of n what it tells me is that this is like a bound. So this, here I have chosen C equal to one. What it tells me is when T of n is less than or equal to C n squared, if t is something like this, then it tells me that the running time of the given program is going to be bounded by this curve. I have taken a special case where C is equal to one. But in general, it can be any constant C. And it tells me that for a particular n naught, what do we mean? n one greater than or equal to n naught and a particular C. If this is T of n, then T of n is guaranteed to run in less than or equal to that C of n squared time.

(Refer Slide Time: 09:16)



$$T(n) = O(n^2) \Rightarrow \text{for constants } c \text{ and } n_0$$

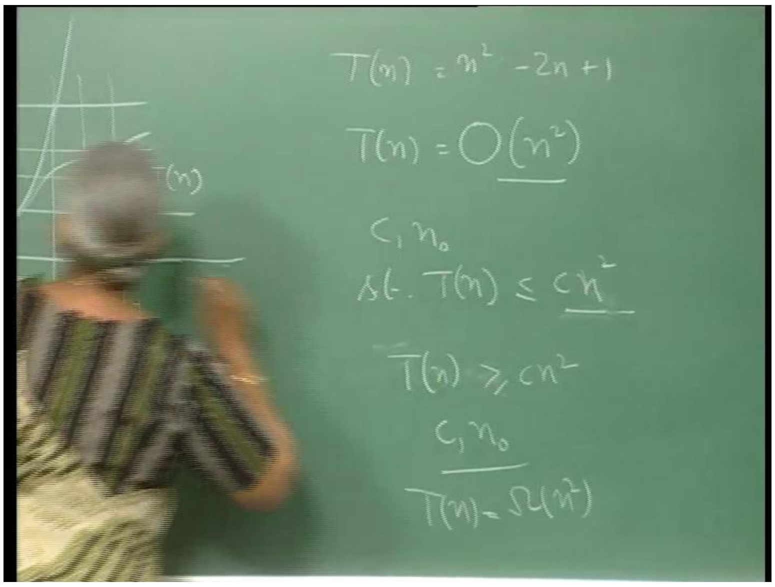
$$\text{s.t. } n \geq n_0,\ T(n) \leq cn^2$$

When $T(n) = O(f(n))$, then $f(n)$ is an upper bound on the growth rate.

When $T(n) = \Omega(g(n))$, then $g(n)$ a lower bound on the growth rate: $i.e.\ \exists\, c$ .s.t $T(n) \geq \Omega(g(n))$.

So, it is a very very important point over here. So T, when the T of n is written as basically O of f of n, let me give you a few examples.

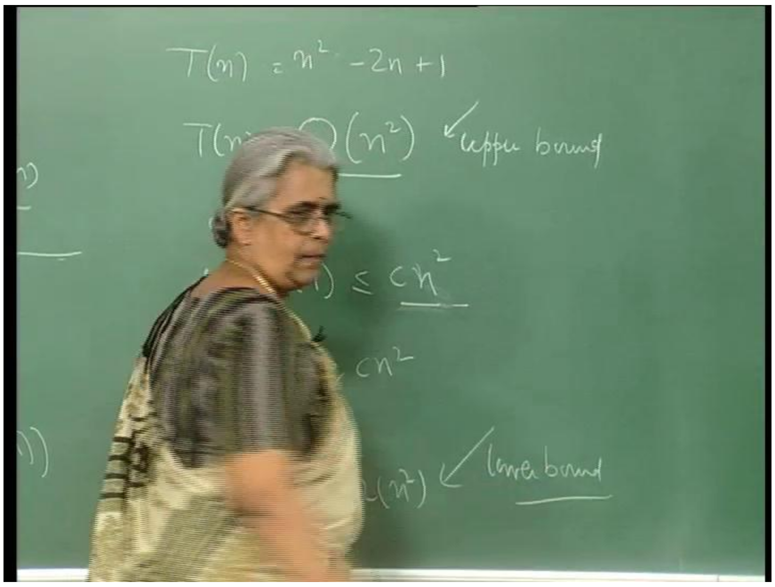
(Refer Slide Time: 09:33)



If I have T of n is equal to n square minus 2 n plus one, then we would write T of n equal to big O of n squared; because it is possible to find a C and a n naught, such that T of n is always less than or equal to C and n naught. So, this is the fundamental point of view. In this particular case, for example, if I take C equal to 2 itself, it gets satisfied. So, this is the important point about time complexity analysis. There is an other notation called the omega notation. And it is just the other way round.

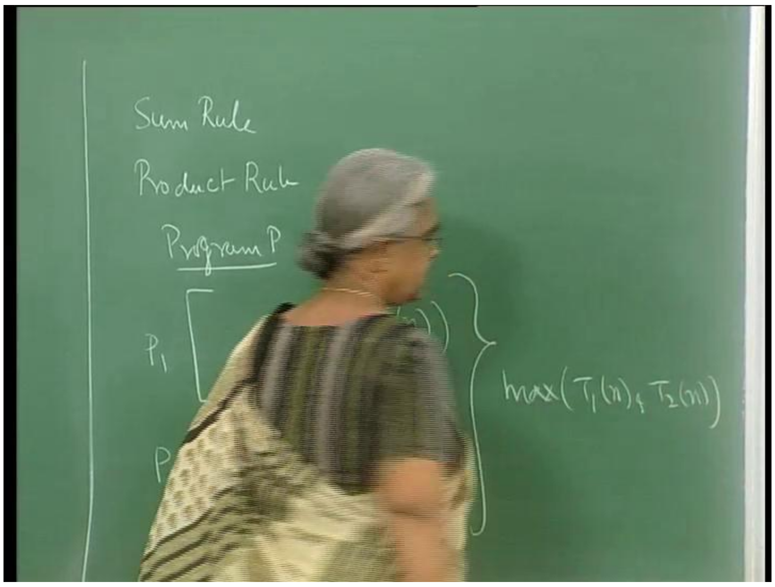
So, what it tells me is the T of n here is less than or equal to C n naught over, C n squared over here. And we have another one. It says T of n is greater than or equal to C n squared over here. Again, you can find a C one and n naught. So, what we say is that when I am looking at T of n is equal to some omega of n squared, then what we are saying is it will always be above this curve and not below that curve. So, that is the meaning of complexity analysis over here.

(Refer Slide Time: 11:03)



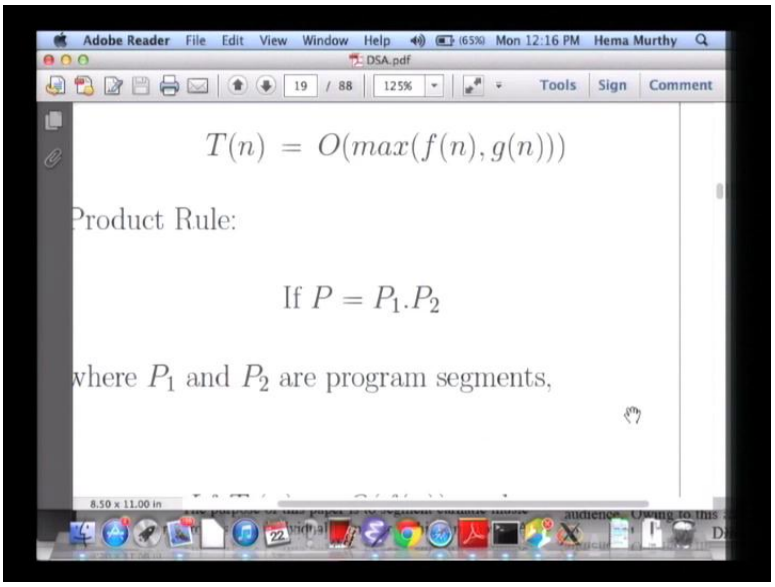
There are 2 bounds. This is like the upper bound on the growth rate of the algorithm; as on increases. And this is the lower bound. There are stricter notations like the theta n small. And looser notations like the, looser bounds likes the small_ o and small_ omega. All this you will relearn in your algorithms course. Now, the question is how do we compute the running time of a program.

(Refer Slide Time: 11:39)



I am going to take very very simple ways of doing. There are 2 primary rules called the sum rule and the product rule. Let us see what these 2 rules are and how do you compute them in a given particular program. What the sum rule states is if I have a program like this, let me call this the program P. And this program P is made up of 2 segments P one and P 2. Then, the sum rule states that the computation of the; what it tells me is the compute time for program P one is T one of f of n and the compute time for P 2 is T 2 of f of n. Then, it says the running time for the entire program P is the max of this; max of T one of n and T 2 of n. This is what the sum rule tells us.

(Refer Slide Time: 12:56)



$$T(n) = O(max(f(n), g(n)))$$

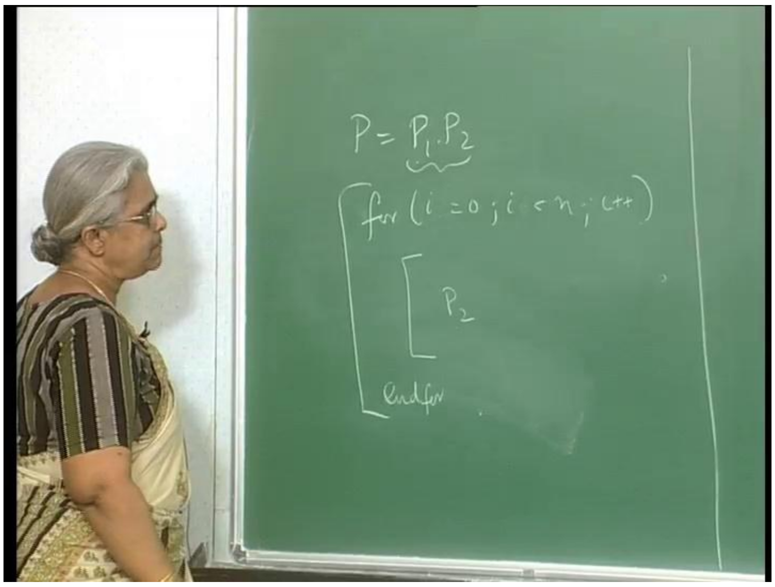
Product Rule:

If $P = P_1.P_2$

where P_1 and P_2 are program segments,

Now there is an other rule, which is called the product rule. And what this product rule tells us is that if I have a program, what do we mean by?

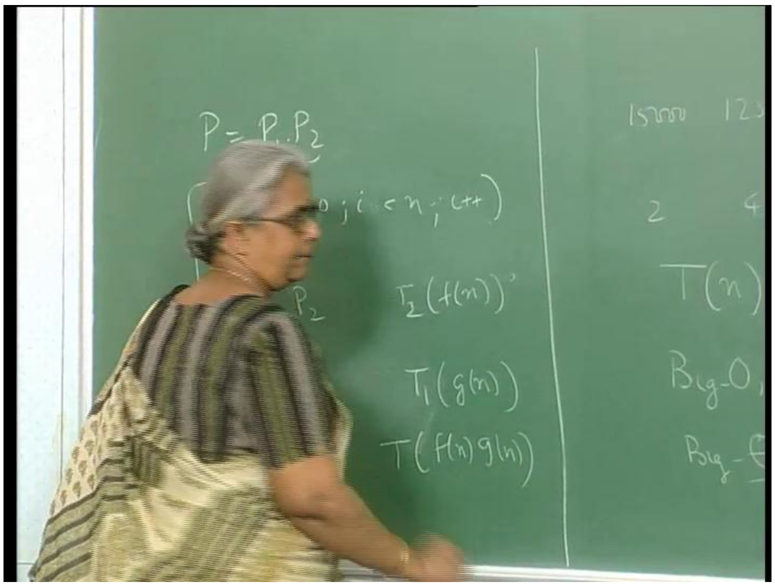
(Refer Slide Time: 13:14)



It says that in the product rule, if P is made up of P one into P 2, what do we mean by this? That is, a program; for example, if I have for i equal to 0; i less than n; i plus plus.

And I have a program segment here. This is P 2, end for. And this is P one. It tells me that P 2 is executed P one times.

(Refer Slide Time: 13:49)



Then the time complexity for this is given by if T one is equal to order of f one of n, T one of f of n let us say. And P 2 is, sorry, P 2 of f of n and T one is g of n. Then, the overall time complexity for the program is T of f of n into g of n. That is what it tells us. Let us look at it with an example in the next lecture.