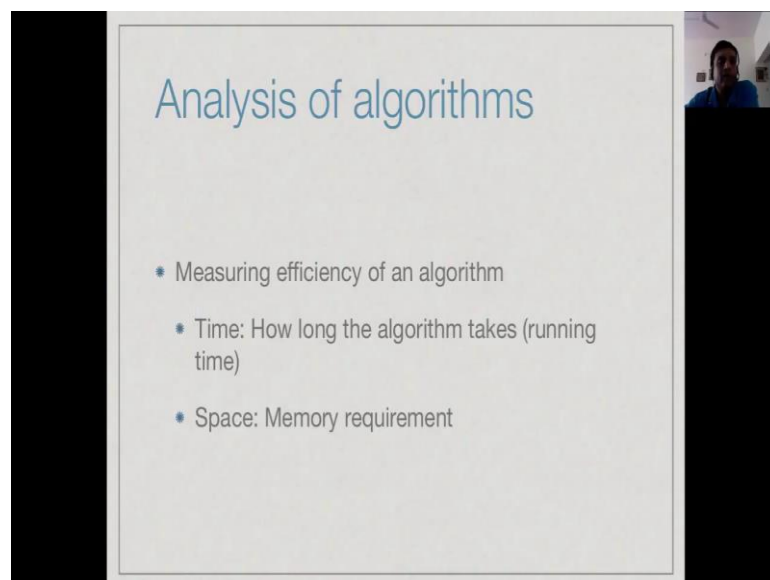


Design and Analysis of Algorithms
Prof. Madhavan Mukund
Chennai Mathematical Institute

Week - 01
Module - 05
Lecture – 05

So, this course is called **design and analysis of algorithms**. So, design is something that is easier to understand. We have a problem, we want to find an algorithm, so we are trying to design an algorithm. But what exactly does **analysis mean?**

(Refer Slide Time: 00:14)

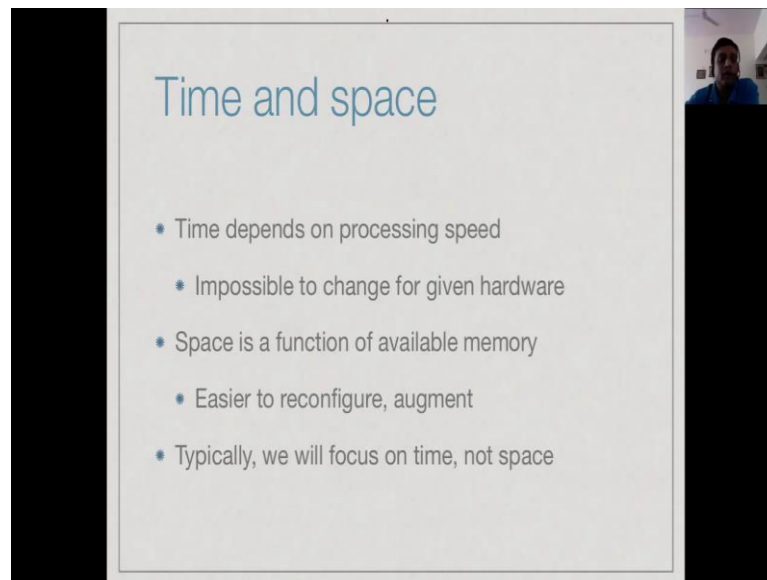


Analysis of algorithms

- Measuring efficiency of an algorithm
- Time: How long the algorithm takes (running time)
- Space: Memory requirement

So, analysis means trying to estimate how efficient an algorithm is. Now, there are two fundamental parameters that are associated with this. **One is time, how long does the algorithm take to execute on a given piece of hardware.** Remember an algorithm will be executed as a program, so we will have to write it in some programming language and then run it on some particular machine. And when it runs, we have all sorts of intermediate variables that we need to keep track of in order to compute the answer. So, **how much space does it take? How much memory does it require?**

(Refer Slide Time: 00:47)



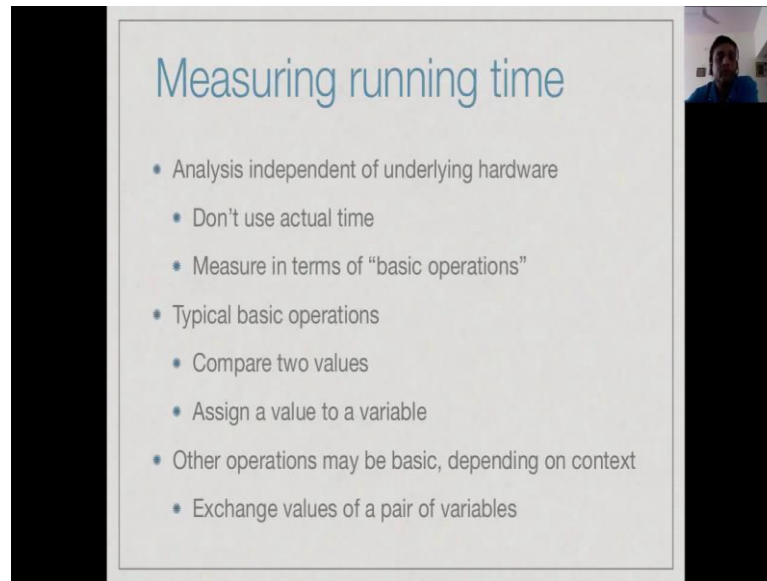
Time and space

- Time depends on processing speed
 - Impossible to change for given hardware
- Space is a function of available memory
 - Easier to reconfigure, augment
- Typically, we will focus on time, not space

Now, we will argue that, in this course at least, we **will focus on time rather than space**. One reason for this is, time is a rather more limiting parameter in terms of the hardware. It is not easy to take a computer and change its speed. So, if we are running algorithm on a particular platform, then **we are more or less stuck with the performance that the platform can give us in term of speed**.

Memory, on the other hand, is something, **which is relatively more flexible**. We can add memory card and increase the memory. And so in a sense, space is a more flexible requirement and so we can think about it that way. But essentially, for this course we will be focusing on time more than space.

(Refer Slide Time: 01:32)



Measuring running time

- Analysis independent of underlying hardware
 - Don't use actual time
 - Measure in terms of "basic operations"
- Typical basic operations
 - Compare two values
 - Assign a value to a variable
- Other operations may be basic, depending on context
 - Exchange values of a pair of variables

So, if you are looking at time, we have to ask how we measure the running time. So, of course, we could run a program on a given computer and report the answer in seconds or in milliseconds, but the problem with this is, that the running time of an algorithm measured in terms of particular piece of hardware will, of course, not be a robust measure.

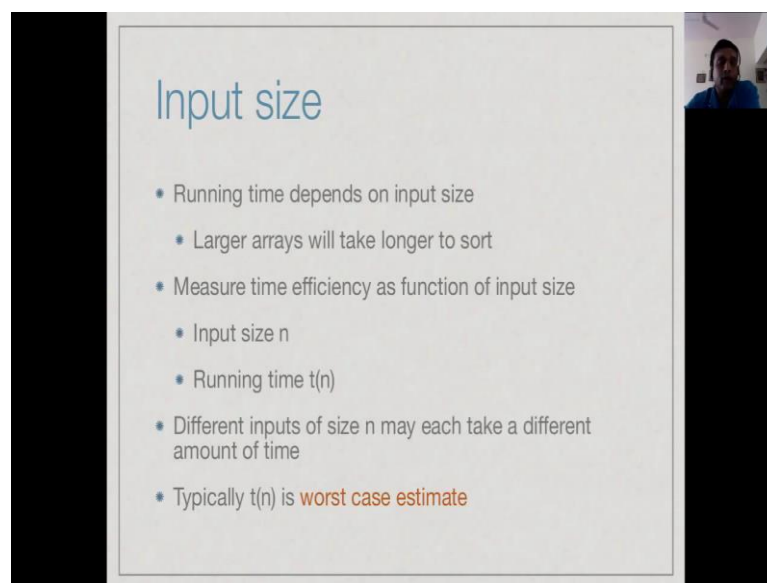
We run it on different computer or we use a different programming language, we might find, that the same algorithm takes different amount of time. More importantly, if we are trying to compare two different algorithms, then if we run one algorithm on one computer, run the other on other computer, we might get a misleading comparison. So, instead of looking at the concrete running time in terms of some units of time, it is better to do it in terms of the some abstracts units of how many steps the algorithm takes. So, this means, that we have to decide what a notion of a step is. A step is some kind of a basic operation, a simple one step operation, that an algorithm performs and the notion of the step, of course, depends on what kind of language we are using.

If we look at a very low-level, at an assembly language kind of thing, then the steps involves moving data from the main memory to register, moving it back, doing some arithmetic operation within the CPU and so on. But typically, we look at algorithms and we design them and we implement them at a higher level. We use programming languages such as C, C plus plus or Java where we have variables, we assign values to

variables, we compute expression and so on. So, for the most part we will look at basic operations as what we would consider single statement or steps in a high-level language. So, this could be an operation such as assigning a value of X equal to Y plus 1 or doing a comparison, if A less than B, then do. So, checking whether A is less than B is one step for us.

Now, we could look at slightly more elaborate notions of steps. For example, we might assume, that we have a primitive operation to actually exchange the values in two variables though we know, that to implement is we actually have to go by a temporary variable. What we will see is, that we will come up with a notion, which is robust so that the actual notion of the basic operation is not so relevant because we will be able to scale it up and down according to what notion we choose to focus.

(Refer Slide Time: 03:55)



Input size

- Running time depends on input size
 - Larger arrays will take longer to sort
- Measure time efficiency as function of input size
 - Input size n
 - Running time $t(n)$
- Different inputs of size n may each take a different amount of time
- Typically $t(n)$ is worst case estimate

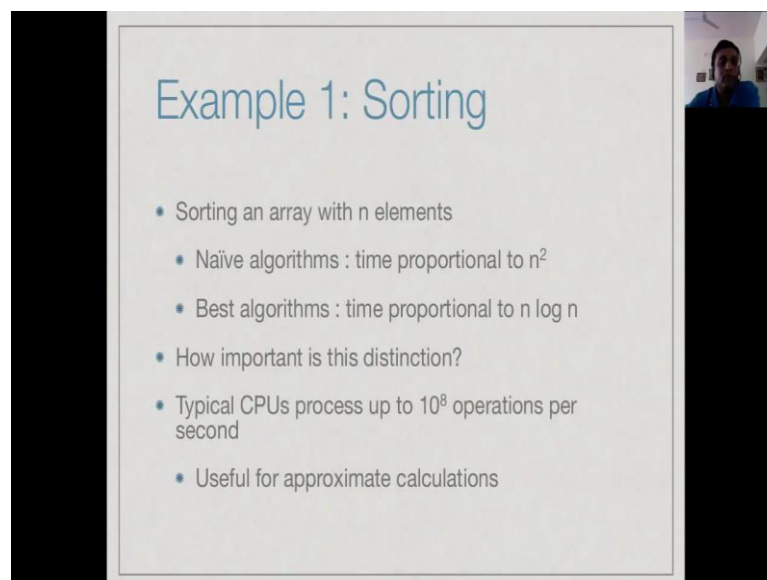
The other important to think to realize, of course is, that the algorithm will take a different amount of time depending on the size of the problem that it is presented with. It is quite natural and obvious, that if we are trying to sort an array, it will take longer to sort a large array than it will take to sort a short array. So, we would like to represent the efficiency of an algorithm as a function of its input size.

So, if the input is of some size n , then it will take time t of n where t will be function depending on the input where even this is not immediately an obvious definition because not all inputs of size n are going to take the same amount of time. Some input will take

less time, some inputs will take more time. So, what do we take? So, it will turn out, that the notion, that we will typically look at is to look at all the inputs of size n and look at the worst possible one that takes a longest time, right. So, this is called a worst case estimate, it is a pessimistic estimate, but we will justify as we go along. But this is what we mean.

Now, when we are looking at the time efficiency of an algorithm what we mean is, what is the worst possible time it will take on inputs of size n and express this as a function of n ? So, before we formalize this, let us just look at a couple of examples and get a feel for what efficiency means in terms of practical running time on the kinds of computers that we have available to us.

(Refer Slide Time: 05:13)



Example 1: Sorting

- Sorting an array with n elements
 - Naïve algorithms : time proportional to n^2
 - Best algorithms : time proportional to $n \log n$
- How important is this distinction?
- Typical CPUs process up to 10^8 operations per second
- Useful for approximate calculations

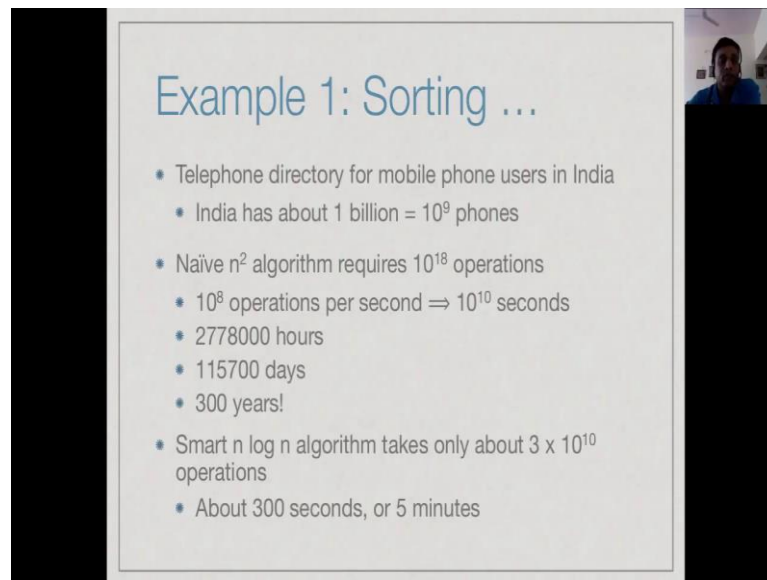
So, let us start with sorting, which is a very basic step in many algorithms. So, we have an array with n elements and we would like to arrange these elements, say in ascending or descending order, for further processing. Now, a naïve sorting algorithm would compare every pair of elements more or less and try to determine how to reorder that. This would take time proportional to n square because we are comparing all pairs of elements. On the other hand, we will see soon in this course, that there are much cleverer sorting algorithms, which work at time proportional to $n \log n$. So, we can go from a naive algorithm which takes time n square to a better algorithm, which takes time $n \log n$.

So, what we have really done is, we have taken a factor of n and replaced it by factor of $\log n$. So, this seems a relatively minor change. So, can we see what the effect of this change is? So, one way to look at this is to actually look at concrete running times. We said, that we will not look at running time as measuring the efficiency of the algorithm, but of course, the efficiency of the algorithm does have an impact on how long the program will take to execute, and therefore on how usable the program is from our practical perspective.

So, if you take a typical CPU that we find on a desktop or a laptop, which we have these days, it can process up to about 10^8 operations. These are the basic operations in the high-level languages like an assignment or checking whether one value is less than another value, right. We can say, that it takes about, it can do about 10^8 operations. Now, this is an approximate number, but it is a very, we need to get some handle on numbers so that we can do some quantitative comparisons of these algorithms. So, it is a useful number to have for approximate calculations.

Now, one thing to remember is that this number is not changing. Obviously, we use to have a situation where CPUs were speeding up every one and a half year, but now we have kind of reached the physical limits of current technology. So, CPUs are not speeding up. We are using different types of technologies to get around this, parallelizing, multicore and so on. But essentially, the sequential speed of a CPU is stuck at about 10^8 operations per second.

(Refer Slide Time: 07:46)



The slide is titled "Example 1: Sorting ...". It contains a bulleted list comparing two sorting algorithms for a dataset of 1 billion mobile phone numbers in India. The naive n^2 algorithm is shown to be impractical, requiring 10^{18} operations, which would take approximately 2.8 million hours (115,700 days or 300 years) if performed at 10^8 operations per second. In contrast, a smart $n \log n$ algorithm is shown to be highly efficient, requiring only about 3×10^{10} operations, which would take about 300 seconds, or 5 minutes.

- Telephone directory for mobile phone users in India
 - India has about 1 billion = 10^9 phones
- Naive n^2 algorithm requires 10^{18} operations
 - 10^8 operations per second $\Rightarrow 10^{10}$ seconds
 - 2778000 hours
 - 115700 days
 - 300 years!
- Smart $n \log n$ algorithm takes only about 3×10^{10} operations
 - About 300 seconds, or 5 minutes

So, now when we take a small input, suppose we are just trying to rearrange our contact list on our mobile phone. We might have a few hundred contacts, maybe a thousand contacts, maybe a few thousand contacts. If you try to sort a contact list, say by name, it really would not make much difference to us whether we use n square or $n \log n$ algorithm. Both of them will work in a fraction of second and before we know it, we will have the answer. But if we go to more non-trivial sizes, then the difference becomes rather stark.

So, consider the problem of compiling a sorted list of all the mobile subscribers across the country. So, it turns out that India has about one billion, that is, 10 to the 9 mobile subscribers. This includes data cards, phones, various things, but these are all people who are all registered with some telecom operator and own a sim card. So, this is a number of sim cards, which are in active use in India today. So, suppose we want to compile a list of all owners of sim cards in India in some sorted fashion.

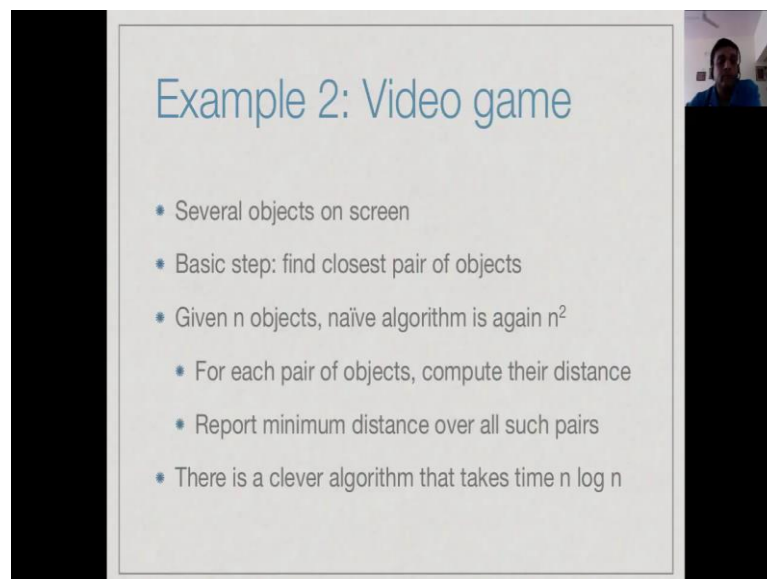
Since we have 10 to the 9 subscribers, if we run an n square algorithm, this will take 10 to the 18 operations because 10 to the 9 square is 10 to the 18 . Now, 10 to the 18 operations per seconds, since we can do only 10 to the 8 operations in 1 second, will take 10 to the 10 seconds. So, how much is 10 to the 10 seconds? Well, it is about 2.8 million hours; it is about 115 thousand days, that is, about 300 years. So, you can imagine, that if we really want to do this using an n squared algorithm, it would really not be practical

because it would take more than our lifetime, more than several generations in fact, to compute this on the current hardware.

On the other hand, if we were to move to the smarter $n \log n$ algorithm, which we claim we will find, then it turns out, that sorting this large number of own users takes only 3 times 10 to the 10 because the log to the base 2 of 10 to the 9 is 30. It is useful to remember, that 2 to the 10 is 1000. So, log to the base 2 of 1000 is 10. Now, since logs add 1000 times, 1000 is 10 to the 6, right, so the log of 10 to the 6 is 20, log of 10 to the 9 is 30. So, 30 into 10 to the 9 $n \log n$ is 3 times 10 to the 10. So, this means, that it will take about 300 seconds. So, it will take about 5 minutes.

Now, 5 minutes is not a short time, you can go and have a tea and come back, but still it will get done in a reasonably short amount of time, so that we can then work with this useful information and go on as opposed to 300 years, which is totally impractical.

(Refer Slide Time: 10:35)



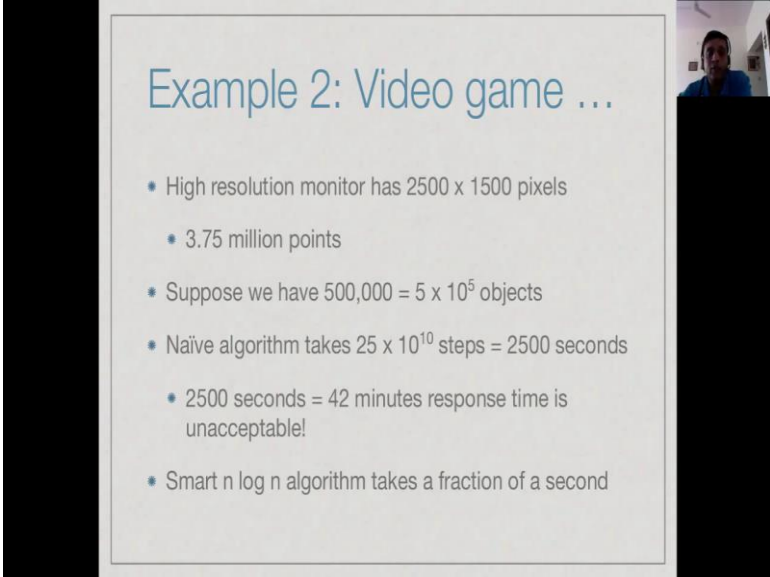
Example 2: Video game

- Several objects on screen
- Basic step: find closest pair of objects
- Given n objects, naïve algorithm is again n^2
 - For each pair of objects, compute their distance
 - Report minimum distance over all such pairs
- There is a clever algorithm that takes time $n \log n$

So, let us look at another example. So, supposing we are playing a video game, right. So, this might be one of the action type games where there are objects moving around the screen and we have to know, identify certain object, shoot them down, capture them and whatever. So, let us assume that as part of the game in order to compute the score, it has to periodically find out the closest pair of objects on the screen.

So, now, how do you find the closest pair of objects among group of objects? Well, of course, you can take every pair of them, find the distance between each pair and then take the smallest one, right. So, this will be an n^2 algorithm, right. So, you compute the distance between any two objects and then after doing this for every pair you take the smallest value. Now, it turns out, that there is a clever algorithm, again, which takes time $n \log n$. So, what is this distinction between n^2 and $n \log n$ in this context?

(Refer Slide Time: 11:35)



Example 2: Video game ...

- High resolution monitor has 2500×1500 pixels
 - 3.75 million points
- Suppose we have $500,000 = 5 \times 10^5$ objects
- Naïve algorithm takes 25×10^{10} steps = 2500 seconds
 - 2500 seconds = 42 minutes response time is unacceptable!
- Smart $n \log n$ algorithm takes a fraction of a second

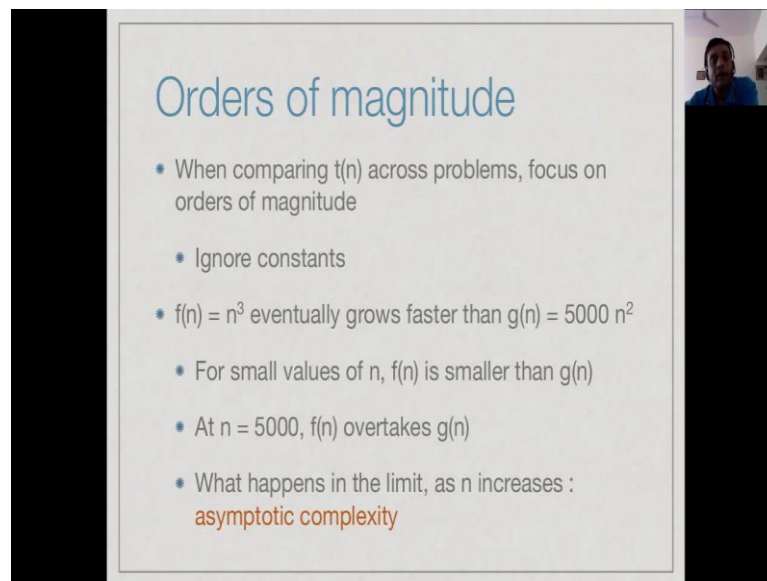
Now, on a modern kind of gaming computer with a large display it is not unreasonable to assume, that we could have a resolution of say, 2500 by 1500 pixel. If we have one of these reasonable size 20 inch monitors, we could easily get this kind of resolutions. So, we will have about 3.75 millions points on the screen. Now, we have some objects placed at some of these points.

So, let us assume that we have five lakh objects, five hundred thousand objects placed on the screen. So, if you were to now compute the pair of objects among these five hundred thousand, which are closest to each other and we use the naïve n^2 algorithm, then you would expect to take 25×10^{10} steps because that is 5 into 10 to the 5 whole square. 25 into 10 to the 10 is 2500 second, which is around 40 minutes.

Now, if you are playing a game, an action game in which reflexes determine your score, obviously, each update cannot take 40 minutes. That would not be an effective game. On

the other hand, you can easily check, that if you do $n \log n$ calculation for 5 into 10 to the 5, you take something like 10 to the 6 or 10 to the 7 seconds. So, this will be a fraction of second, 1-10th or 1-100th of the second, which is well below your human response time. So, it will be, essentially, instantaneously. So, we move from a game, which is hopelessly slow to one in which it can really test your reflexes as a human.

(Refer Slide Time: 13:00)



The slide is titled "Orders of magnitude" in blue text. It contains a list of bullet points:

- When comparing $t(n)$ across problems, focus on orders of magnitude
- Ignore constants
- $f(n) = n^3$ eventually grows faster than $g(n) = 5000 n^2$
 - For small values of n , $f(n)$ is smaller than $g(n)$
 - At $n = 5000$, $f(n)$ overtakes $g(n)$
- What happens in the limit, as n increases :
asymptotic complexity

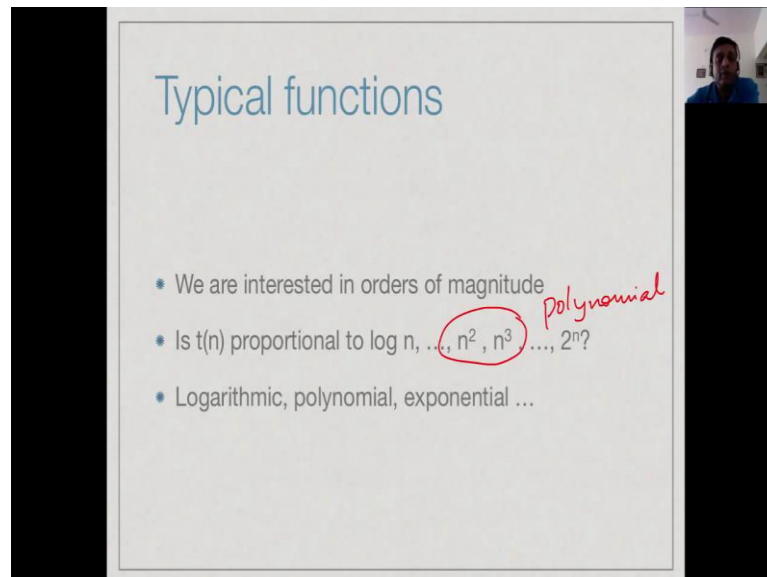
A small video inset in the top right corner shows a man with a beard and glasses speaking.

So, we have seen in these two examples, that there is a real huge practical gulf between even something as close to each other as $n \log n$ and n squared. The size of the problems, that we can tackle for n squared are much smaller than the size probably we can tackle for $n \log n$. So, when we look at these functions of n , typically we will ignore constants. Now, this will partly be justified later on by the fact, that we are not fixing the basic operation, but essentially by ignoring constant we are looking at the overall function of the efficiency as the function of n , in the, as it increases, right. So, it is what we call asymptotic complexity, as n gets large how does the function behave.

So, for instance, supposing we have some function, which grows like n squared with large coefficient and something, which goes like n cube with a coefficient of 1, initially it will look like n squared, say $5000 n$ squared is much more than n cube, but very rapidly, say at 5000, right. At 5000 we will have, both will be 5000 cube and beyond 5000 the function n cube will grow faster than function $5000 n$ squared, right.

So, there will be a point beyond which n cube will overtake n square and after that it will rapidly pull out. So, this is what we would typically like to measure. We would like to look at the functions as functions of n without looking at the individual constants and we will look at this little more in detail.

(Refer Slide Time: 14:30)



Typical functions

- We are interested in orders of magnitude
- Is $t(n)$ proportional to $\log n$, ..., n^2 , n^3 , ..., 2^n ? *polynomial*
- Logarithmic, polynomial, exponential ...

So, since we are only interested in orders of magnitude, we can broadly classify functions in terms of what kind of functions they look like, right. So, we could have function, which are proportional to $\log n$; functions, which are proportional to n , n squared, n cube, n to the k ; which is, so any n to the **fixed k is a polynomial**. So, we can call these, call these functions, you will look at, as polynomial, right. So, these are polynomial. And then we could have something, which is **2 to the n if 2 to the n essentially, come, comprises of looking at all possible subset**. So, these are typically the brute force algorithms where we look at every possibility and then determine the answer, right. So, we are **logarithmic polynomial and exponential**. So, what do these look like in terms of the numbers that you ((Refer Time: 15:23)).

(Refer Slide Time: 15:25)

Typical functions $t(n)$...

Input	$\log n$	n	$n \log n$	n^2	n^3	2^n	$n!$
10	3.3	10	33	100	1000	1000	10^6
100	6.6	100	66	10^4	10^6	10^{30}	10^{157}
1000	10	1000	10^4	10^6	10^9		
10^4	13	10^4	10^5	10^8	10^{12}		
10^5	17	10^5	10^6	10^{10}			
10^6	20	10^6	10^7				
10^7	23	10^7	10^8				
10^8	27	10^8	10^9				
10^9	30	10^9	10^{10}				
10^{10}	33	10^{10}					

Feasibility

So, if you look at this chart, the left column varies the input size from 10 to the 1 to the 10 and then each column after that is the different type of complexity behaviour as function, which ignores constants and only looks at the magnitude. So, we have $\log n$ and then we have something, which is polynomial n , n square, n cube. In between we have $n \log n$ we saw before and then we have these two exponential function, which are 2 to the n and n factorial, right. So, now, in this we are trying to determine how efficiency determines practical useable.

So, now, if you look at this, we said, that we can do 10 to the 8 operation in 1 second. So, maybe we are willing to work at 10 seconds. So, 10 to the 9 operation is about the limit of what we would consider. The efficient 10 to the 10 operation would mean 100 second, which would mean 2 minutes and may be 2 minutes is too long.

So, now clearly, if we are looking in logarithmic scale, there is no problem. Up to 10 to the 10, right, we can do everything in about 33 step, which will take very, very ((Refer Time: 16:37)). Now, in a linear scale given, that we are expecting 10 to the 9 as our limit, this becomes our limit, right. So, we can draw a line here and say, that below this red line things are impossibly slow. $n \log n$ is only slightly worse than n . So, we can do inputs of size 10 to the 8 because $n \log n$ to 10 to the 8 is something proportional to 10 to the 9.

Now, there is this huge gulf that we saw. If we move from $n \log n$ to n^2 , then the feasibility limit is somewhere between 10^4 and 10^5 and together, 10^5 , we already reach time and running time of 10^{10} , which is beyond what we want and now if we move to n^3 then the feasibility limit drops further. So, we cannot go beyond an input size of 1000 right. So, we can see that there is a drastic drop and when we come to exponentials really beyond some trivial things of the size 10 or 20, we cannot really do anything, anything, which is even a 100 will be impossibly slow, right.

So, we have this kind of, this sharp dividing line of feasibility right. And so we can see that it is important to get efficient algorithm because if we have inefficient algorithm, even if we think that they work correctly, we will not be able to solve problem of any reasonable size right. When we are looking at computational algorithms running on a PC, you would really expect our input sizes to be 100s, 1000s and much more right. So, we are shorting things. We really do expect large volume to data if you want to you look at voters list, population data or data coming out of say biology experiment about genetic material, now these things are typically very large amount of data, so it is really important to get the less possible algorithm to do these things otherwise the problem at hand will not be effectively solved.