



IIT Madras

ONLINE DEGREE

Programming, Data Structures and Algorithms using Python
Professor Madhvan Mukund
Analysis of Algorithms

So, in the first week we looked at some motivation for studying algorithms. And then towards the end, we looked at one specific problem regarding SIM cards and Aadhaar cards, where we showed that doing the clever thing can save you an enormous amount of time. So, this is the design of an efficient algorithm; but we also need to analyze these algorithms. So, let us spend a little bit of time trying to understand what analysis of an algorithm means.

(Refer Slide Time: 00:36)

Measuring performance

- Example of validating SIM cards against Aadhaar data
 - Naive approach takes thousands of years
 - Smarter solution takes a few minutes
- Two main resources of interest
 - Running time — how long the algorithm takes
 - Space — memory requirement
- Time depends on processing power
 - Impossible to change for given hardware
 - Enhancing hardware has only a limited impact at a practical level
- Storage is limited by available memory
 - Easier to configure, augment
- Typically, we focus on time rather than space

Madhvan Mukund Analysis of algorithms

So, essentially, we are trying to compute the performance or the expected performance of an algorithm. So, remember, this problem where we were trying to validate whether the other entries in SIM cards were actually valid with respect to the other database. So, the Naive approach was to compare each SIM card along with every other Aadhaar card; so, have a nested loop for every SIM card, check whether the Aadhaar card appears in the other database. And this would be something which we roughly said will take n squared time, if n is the number of Aadhaar cards.

So, we said that typically we assumed a realistic case where we had 1 billion; we actually have more than that more than 100 crore, Aadhaar cards in India. But, 100 crore Aadhaar cards and a similar number of SIM cards, we said if we actually tried this Naive approach to take us several 1000 years. On the other hand, if we were able to arrange the Aadhaar cards so that they were in,

say, ascending order; then we saw this clever strategy, where we keep checking the midpoint and reducing the search space that we needed to look for the SIM card value by half each time. And then we cut it down to a significantly smaller amount of a few minutes.

So, this was a kind of informal analysis that we did. So how do we formalize this? So first of all, what are we interested in when we are computing the performance of an algorithm? The obvious thing is time? This is what we were talking about when we were looking at the Aadhaar card and SIM card example. How long does the algorithm actually take to run? A less obvious thing is space. How much memory do we need to store all the values that the algorithm computes on its way to computing the final answer? So, time of course, depends to some extent, on what we are running the process on.

So, I had claimed that Python takes can compute 10^7 operations per second; and I did not qualify on what machine I was talking about. Well, it turns out that hardware is such that over the last 10 or 15 years, there has not been really much dramatic speed up in terms of the processing power of the CPUs that we have in our. You know, in the laptop or the desktop that we are using at home, they are all pretty much the same. So, this 10^7 applies in all of these. And if you remember, we ran the experiments to show that it takes 10^7 on Google's co lab, which is on the cloud.

So, if you are running on a normal machine, you can assume that. So basically, you are running your stuff on some computer, and it is very hard to change the configuration of that computer. Of course, you could buy a faster machine. But we will see that changing to a faster machine that is even 10 times as fast will only have a limited impact. So, if you have something which is inherently long to compute, it is not helped much to change the hardware. So, you really have to look at something which is efficient, independent of the hardware; and that is how we will compute our time.

Now, storage in a sense is a little easier to modify, because we can more easily typically add additional memory to our laptops, then we can add power to our CPUs. So, in essence, storage is easier to augment, and so partly for this reason and partly; because storage comes with a lot of other issues, which are not easy to assess. We will focus when we talk about performance more on the time rather than the space aspect of the algorithm.

(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 take different amounts of time
 - We will return to this point later

Example 1 SIM cards vs Aadhaar cards

- $n \approx 10^9$ — number of cards
- Naive algorithm: $t(n) \approx n^2$
- Clever algorithm: $t(n) \approx n \log_2 n$
 - $\log_2 n$ — number of times you need to divide n by 2 to reach 1
 - $\log_2(n) = k \Rightarrow n = 2^k$

Madhavan Mukund Analysis of algorithms

So, clearly the time that an algorithm takes is not invariant; it is not that every time I run the algorithm, it will take the same amount of time. And the most obvious dependence that this running time has is on the input size. So, say for example, I am trying to rearrange a list or an array in sorting order in ascending order. Obviously, the larger the list is, the more elements that I have, the longer it is going to take me to to arrange them.

Similarly, if I am searching for a value, the larger the list, the longer it will take me find the value. So, input size clearly determines the performance of the algorithm. So, what we have to do is talk about how this algorithm actually behaves as a function of the input size.

If I, if I double the input size, what will happen to the time will it double? Will it go up by a factor of 4? Will it go up by a factor of 400? So, we talked about the time for an input of size n of n ; so we think of it as a function. A function which takes the input size as an input, and produces an estimate of the running time as the output.

Now, even when we fix n input size, it is not clear that all inputs of the same size are equally hard to process. For instance, if I am rearranging a table in ascending order or a list in ascending order, and it is already in ascending order; then clearly, I do not have to do much work.

So, maybe I can just do one scan to verify that it is in ascending order and say nothing needs to be done. Or maybe I searching for a value sometimes you might find the value very fast; it might

be the first position that will look in the list that has that value. So, clearly, there is going to be a lot of variation even for the same input size. So, we will have to come back to this a little later and argue about what input we are looking at. You know which input of size n are we looking at, or which inputs of size n are we looking at.

So, with this minimal background, we will come back and address these points more formally; let us go back to our SIM card-Aadhaar card example. So, there we said that n the input size was roughly 10^9 , which is 1 billion or 100 crores; and this is related to the population of India. So, roughly everybody has an Aadhaar card, and the number of SIM cards is equal to the population; not because everyone has a SIM card, but because as we said, there are multiple SIM cards with some people. So, we said that the Naive algorithm, if we look at it as a function of n takes time proportional to n^2 ; because we had these two nested loops, where you had an outer loop, which goes cycles through n SIM cards.

And for each of the n SIM cards, you will cycle through the n Aadhaar numbers to look for the correct match; so, n times n , n^2 . So, that is how we got that n^2 . And when we gave this clever algorithm, which would have the interval to search, and if you think about that having process; basically, takes an interval of linked then, and then it keeps dividing it until it comes down to an interval of size 1. So, we start with n , we divide by 2, we divide by 2, we divide by 2, and we come down to 1. So, if you reverse the process, we start with one we multiply by 2, we multiply by 2, and we reach end.

So, basically 2^k to that many multiplications is n , so 2^k is n . So, the number of times we have to divide n by 2 to reach 1, is the same as the number of times you have to multiply 1 by 2 to reach n . So, it is that power, 2^k is n , so this is the log. So, the log to the base 2 of n is how many times I have to probe or look into that Aadhaar card list, for each SIM card. There are n SIM cards, so n times log of n . So, we will typically write just log, we will not use this 2; but for the moment, we will use to emphasize that we are dividing by 2. But usually, if we do not write the base of the log, you can assume it is 2.

(Refer Slide Time: 07:32)

Input size ...

Example 2 Video game

- Several objects on screen
- Basic step: find closest pair of objects
- n objects — naive algorithm is n^2
 - For each pair of objects, compute their distance
 - Report minimum distance across all pairs
- There is a clever algorithm that takes time $n \log_2 n$

- High resolution gaming console may have 4000×2000 pixels
 - 8×10^6 points — 8 million
- Suppose we have $100,000 = 1 \times 10^5$ objects
- Naive algorithm takes 10^{10} steps
 - 1000 seconds, or 16.7 minutes in Python
 - Unacceptable response time!

Madhavan Mukund Analysis of algorithms

Input size ...

Example 2 Video game

- Several objects on screen
- Basic step: find closest pair of objects
- n objects — naive algorithm is n^2
 - For each pair of objects, compute their distance
 - Report minimum distance across all pairs
- There is a clever algorithm that takes time $n \log_2 n$

- High resolution gaming console may have 4000×2000 pixels
 - 8×10^6 points — 8 million
- Suppose we have $100,000 = 1 \times 10^5$ objects
- Naive algorithm takes 10^{10} steps
 - 1000 seconds, or 16.7 minutes in Python
 - Unacceptable response time!
- $\log_2 100,000$ is under 20, so $n \log_2 n$ takes a fraction of a second

Madhavan Mukund Analysis of algorithms

So, now this Aadhaar card example is not the only case where one can get a dramatic improvement; which makes a difference between something being practical and something being wildly unusable. So, let us imagine a hypothetical video game; so, in a video game, there are usually several items on the screen.

So, let us assume this is one of those games where there are spaceships and aliens and, and whatever obstacles flying around. And these are being tracked by you the game player and of course by the game designer, because the game designer also has to keep updating the screen every time something happens.

So, let us assume that one of the basic things that the game designer needs to do is to calculate among all the objects which are currently on the screen, which two are closest to each other. So, it could be that for instance, it is trying to calculate something about some hostile piece, which is going to attack some other piece, and so on.

So, this is a basic step, say we want to calculate the closest pair of objects. So, we have a large number of objects on the screen. And at every instant that the game progresses, we need to keep track of each pair of objects is the closest to each other. So, let us assume that now the number of objects is our input; so, we have n objects, and we want to find out which pair among these n are closest to each other.

So, how would you do this naively? You will take each object and measure the distance from that object to every one of the $n - 1$ objects on the screen. So, will have to compare n objects to $n - 1$ objects. So, of course, you can divide by 2 because you do not have to compare it in both directions; but still, it is roughly n into $n - 1$ by 2, which is n^2 .

So, if we want to compute all pairwise distances, and then determine the minimum one; then we have to actually do n^2 work. Now, just like we saw in the earlier example, you can actually replace this brute force or Naive n^2 by an $n \log n$; there is actually an $n \log n$ algorithm for this problem also.

So, what would the impact of this improvement be in this case? So, let us assume that we are working on some standard hardware. So, if you look at a typical laptop, the type of laptop that you have at home; it will have maybe 1500 to 2000 pixels horizontally and about 1000 pixels vertically. But, if you are really playing on some fancy gaming console, you might have a large number of pixels like 4000 this way and 2000 this way; so that is about 8 million pixels. So, at the moment, we are not particularly concerned about the pixels except that the more pixels you have, the more objects you can have on the screen.

So, let us assume that this game designer has actually programmed this game; so that there can be at any given time up to one lakh, one hundred thousand, 10^5 objects on the screen. Now, if we do our naive algorithm for computing which pairs are closest to each other, then we are going to take roughly 10^5 times 10^5 , which is 10^{10} steps in order to make this update, to compute which pair is closest to each other. And if you do a simple

calculation based on our earlier estimate of how fast Python is, this will take us 1000 seconds; so, 1000 seconds is 16 minutes.

So, this means that if this is part of the game, that is every time you make a you move your joystick or whatever, you move your cursor or you shoot something; this has to take 16 minutes to recalculate the state of the screen. Obviously, this game is not going to be very exciting to play. Now, you might think okay, I would not do this in Python; I would do it in c plus plus use a faster language. So, it is C plus plus is about 10 times as fast as Python; so what will happen in c plus plus is this will become 1.67 minutes, so a minute is a long time. So, after you make a move, if it is going to take hundred seconds for it to update, it is not going to be an effective game.

So, this is why an n squared algorithm will be totally useless in this scenario. Because the response time as a user of the system, you expect a response time will be almost instantaneous; so, that you can play it in real time. You cannot make a move and then think about go back and sit for half a minute or one minute, and then come back and make another move. And what happens if we use this $n \log n$ thing? What is log of one hundred thousand? Well, 2 to the power of 10 is roughly 1000, because it is 1024. And 2 to the power 20 is 10 to the power 6, because it is 1000 times 1000; this is 10 to the power 5.

So, you can make a kind of if you take this, then 2 to the power 19 will be roughly five lakhs; it is 512,000 and this is hundred thousand. So, it is going to be something similar to 1718, certainly less than 20; so, log of one lakh is less than 20. So, we have now 20 times n is 10 to the 5, which is something like 2 into 10 to the power 6.

So, 10 to the power 6 is one tenth of 10 to the power 7. So even in Python, this will take only point two seconds; and if you are using a faster language like c plus plus will take point 02 seconds. So, it will really be almost within your is certainly faster than our human reaction time, which is typically closer to point one second.

So therefore, this will now be almost like any move you make, the world changes instantly. So, this is just another illustration as to why designing an algorithm which goes from n squared to $n \log n$, can make a huge impact on how useful that algorithm is in practice.

(Refer Slide Time: 12:59)

Orders of magnitude



- When comparing $t(n)$, focus on orders of magnitude
 - Ignore constant factors
- $f(n) = n^3$ eventually grows faster than $g(n) = 5000n^2$
 - For small values of n , $f(n) < g(n)$
 - After $n = 5000$, $f(n)$ overtakes $g(n)$

$$n = 8$$
$$8^3$$
$$\underline{\underline{5000 \cdot 8^2}}$$



Madhavan Mukund

Analysis of algorithms

Orders of magnitude



- When comparing $t(n)$, focus on orders of magnitude
 - Ignore constant factors
- $f(n) = n^3$ eventually grows faster than $g(n) = 5000n^2$
 - For small values of n , $f(n) < g(n)$
 - After $n = 5000$, $f(n)$ overtakes $g(n)$
- Asymptotic complexity
 - What happens in the limit, as n becomes large

$$5000^3$$
$$6000^3$$
$$5000 \cdot 5000^2$$
$$5000 (6000^2)$$



Madhavan Mukund

Analysis of algorithms

सिद्धिर्भवति कर्मजा

- When comparing $t(n)$, focus on orders of magnitude
 - Ignore constant factors
- $f(n) = n^3$ eventually grows faster than $g(n) = 5000n^2$
 - For small values of n , $f(n) < g(n)$
 - After $n = 5000$, $f(n)$ overtakes $g(n)$
- Asymptotic complexity
 - What happens in the limit, as n becomes large
- Typical growth functions
 - Is $t(n)$ proportional to $\log n, \dots, n^2, n^3, \dots, 2^n$?
 - Note: $\log n$ means $\log_2 n$ by default
 - Logarithmic, polynomial, exponential, ...

2^n
number of subsets
of n elements



Madhavan Mukund

Analysis of algorithms

So, now let us get back to the more technical aspect, which is how do we actually evaluate and compare these different algorithms formally. So, we cannot just wave our hands and say, this looks like n squared that looks like $n \log n$ and so on; we need a way to actually compute this and compare this.

So, the first thing is that we will always be interested in comparing these functions. Remember, t of n is the time taken for an input of size n ; but we will be interested on t of n up to some order of magnitude. In other words, we will ignore what are called constant factors. So, for instance, if you look at something like n cubed, and look at n squared, it seems obvious that n cubed is bigger than n squared.

But what if I stick a 5000 in front of n squared? Then is n cubed bigger than n squared? In some sense, it is still bigger; because even though for small n , 5000 n squared will be more than n cubed. So, supposing I take n is equal to 8, for example; on one side, I will write 8 cube and here I will get 5000 times 8 squared. So, it is clear that in this situation for a small value of n , and 5000 n squared is not as good as n cube. But we are not interested really in these inputs of size 8 and 10, and so on. We really want to know what happens because our algorithm is going to run on arbitrarily large inputs.

So, we are really looking at what is called the asymptotic complexity; as n becomes large, which of these two algorithms is going to be better, which of these two functions is going to go faster. So, here you can check that once I reach 5000, on the right side, I have 5000 times 5000 squared

and on the squared; and on the left-hand side, I have 5000 cube. And now if I go beyond that like 6000, I will have 6000 cube; and here I'll have 5000, which is a fixed constant times 6000 squared. So, once I cross 5000, the function on the right-hand side 5000 n squared is going to go below n cubed and stay below n cubed; so, this is the kind of comparison that we want to make.

So, we are interested in this t of n in different forms or shapes; so, we have seen this logarithmic form. So, logarithmic complexity comes for instance, when we are doing this half interval searching what is called binary search. When we keep dividing by 2 the size that we are searching in; then in $\log n$ steps, we reduce our search interval to one point, and then we have found it or not found it. So, that is something which is logarithmic. Now, we already saw examples of polynomials; we have seen n squared algorithms, these two nested loops. You can have three nested loops; you will get n cubed and so on. So, these are all called polynomials, you could have exponential something which takes 2 to the power n .

So, 2 to the power n is the number of subsets of n elements. So, imagine that you have a situation, where you have a large number of objects of different shapes and sizes; and you want to fit them into a loading load them into a delivery truck. Now, you want to find out which subset of objects will fit this truck most effectively, so that you minimize the number of trips that you make on the truck.

So, the Naive solution would be to try every subset; and once you pick a subset, you will try to fit it in as well as possible. And finally, you among all the subsets that you examine, you find the one which leaves the least empty space in the truck and you choose that one. So, this would be something to take 2 to the power n .

(Refer Slide Time: 16:31)

Orders of magnitude



Input size	$\log n$	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^{12}			
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}	10^{11}				



Madhavan Mukund

Analysis of algorithms

Orders of magnitude



Input size	$\log n$	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^{30}	10^{157}
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^{12}			
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}	10^{11}				



Madhavan Mukund

Analysis of algorithms

सिद्धिर्भवति कर्मजा

Input size	Values of $t(n)$					
	$\log n$	n	$n \log n$	n^2	n^3	2^n
10	3.3	10	33	100	1000	1000
100	6.6	100	66	10^4	10^6	10^{30}
1000	10	1000	10^4	10^6	10^9	10^{157}
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^{12}		
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}	10^{11}			



Madhavan Mukund

Analysis of algorithms

So, now we can tabulate these different functions, and try to see how much time a typical input will take in terms of number of steps as a function of n . So here, on the left-hand side, we have a column, which is the input size which is growing in multiples of 10. So, we start with input of size 10, which is a very small input; and we go all the way up to 10 to the power 10, which is 10 billion, or one hundred crores. It is in a powers of 10, we increase the input and then we see what happens; so the columns correspond to different functions, so, this is our logarithmic function. This is a polynomial with degree one is a linear function, this is n squared and cubed and so on.

In between n and n squared, we have $n \log n$. This is what we saw when we talked about the SIM card-Aadhaar card algorithm; or that video game algorithm, which we claimed had an efficient $n \log n$ algorithm. And then I described to you a situation, where 2 to the n might make sense, if you are trying out all possible subsets of some set in order to determine which is the best one. And if you are trying out all possible subsets, but up to rearrangement; supposing you are looking for all possible permutations, which do something and which one is best. Then you might end up with something which is proportional to N factorial, because there are n factorial ways of arranging n objects in different sequences.

So, we have these different things; and now the question is what happens when say n is 10. So, when n is 10, if you look at the first row here, then already n factorial has become quite large; it is already within one tenth of python's one second limit, all the other values are reasonable. But now, if I go to 100 at this point, this number has already gone well beyond anything reasonable

that we can do; 10 to the power 157 operations is going to take forever. Even 2 to the power n has now exploded, because remember that 2 to the power 100, is going to be 2 times 2 times 200 times; if I just multiply 10 times I get 1000, and I am going to keep multiplying it.

So, I am going to get something like 10 to the power 30. So, 10 to the power 30 again, if you think about even with a very fast language 10 to the power 8, or 9 or maximum 10 to the power 10 operations per second, you are going to do. You are going to take 10 to the power 20 seconds in order to get things done; so this again has become infeasible.

So, what you can see is that if your complexity of an algorithm is a large function of n ; then very small inputs will drive it to an infeasible range, where you cannot execute. You cannot take a 2 to the power n algorithm and hope to even execute it for 100 objects. So, if you are shifting house, and you have 100 objects in your house, then you cannot apply this algorithm.

So, what happens now as we go up the line, so we go to 1000 at 1000 n cubed is already approaching an infeasible limit. So, in Python, this will be talking about 100 seconds; 100 seconds is like 2 minutes. And that is acceptable in some situations, but not acceptable and others like in that video game situation.

So, if we go to 10 to the 4, then already n cubed is off our list; because 10 to the 12 seconds or 10 to the 12 operations is going to take something like 10,000 seconds, that is not good. So, now as we see, at 10 to the power 6 for instance, n squared also goes up above any reasonable limit; and at 10 to the power 7, 8, 9 we still can manage with $n \log n$.

So, what we see in this thing if you want plot it, is you can kind of draw a feasibility line. You can say that up to here, maybe it's feasible, up to here is feasible; anything below this red line is not feasible. Then when I come here, above this is feasible, and I come here above this is feasible and so on. And then maybe here, you can say that at this point, this is say let us say 10 to the 10 is feasible; and then you can say that up to here. So, this red step staircase tells us that, you can go up to very large inputs if your function is up to $n \log n$. But if it goes to n squared, you are already down to inputs of the size, one lakh or less.

Typically, 10,000 is more like it, because with 10,000 you can at least hope to do it in a few seconds. With one lakh already, you are pushing it too; so maybe we should actually draw this

line above 10 to the power 10, just to be more. So, so what we are saying is that there are two sides to this story. So, the one side of the story is estimating how fast your algorithm will work; and from that, you can determine what your algorithm can be used for. What type of problems, what sizes of problems, it makes sense to use it. Conversely, if you know what size a problem you are going to use, then you can determine whether you can get away with an inefficient algorithm or search for a better one.

(Refer Slide Time: 21:19)

Measuring running time

- Analysis should be independent of the 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
- Exchange a pair of values?
 $(x, y) = (y, x)$
 $t = x$
 $x = y$
 $y = t$
- If we ignore constants, focus on orders of magnitude, both are within a factor of 3
- Need not be very precise about defining basic operations

Madhavan Mukund Analysis of algorithms

So, we still have this problem of measuring running time. So, in this Python example, I had defined for you a timer class, which actually measured the running time of the code as it was running. But then that does not really help us much, because it is should be independent of the. I mean, we should be able to make an independent assessment of the time, like we have done with our other things. Without really having to physically run it and see how long it takes on a particular input of that size. So, we will typically measure this running time in terms of what we will arbitrarily call basic operations. So, a basic operation you can imagine is one line of Python code, which does not involve calling a function.

So, what is the basic operation if I write and if for a while; it says if x less than y, or while n less than 100. Then you have to check that the current value of x is smaller than the current value of y, or check that the current value of n is less than 100. So, this comparison of two values can be a basic step. Similarly, if I say assign y equal to x plus three, then that is also a basic step; so,

assignment or a comparison is a basic step. So, this notion of a basic step is somewhat flexible, because it also depends on what language you are looking at. So, in Python, you would have seen this kind of clever way to use pairs or tuples to exchange; in particular to exchange two values.

I mean, this is used as it is essentially a multiple assignment, you can say `x comma y equal to zero comma zero`. But in particular, you can assign each of them to the other one; but in practice, this does not work. So, you can imagine that these are values, supposing these values are actually physical objects. So, supposing you have two bottles, one bottle has got some water and the other bottle has got some juice. And you now want to transfer the water to the juice bottle and the juice to the water bottle; you have enough space to store them in the end. But in between, you cannot do it without using a third bottle; because, if you try to start pouring the water into the juice, it will get contaminated and vice versa.

So, you will need a third bottle into which you will pour one of the two and then you will exchange. So, the same ways, how you would normally do this in a standard programming language? You will create a new temporary name called `t` and you will park the value of `x` that you have now there. So, that when you replace `x` by `y`, because that is what you want to do, you want to swap `x` and `y`; so, `x` has to take the value of `y`. When you take the value of `y`, the value of `x` that you had before is not lost, it is stored in `t`. So, then you can go to `t` and copy that value back into `y`. So, in essence this, what we look think of as a Python with basic operation on the left; takes three steps on the right.

So, we do not want to really bother about this. So, if we ignore these constants as we said, we would do and look at orders of magnitude. If something takes three times, another thing it just could be a way of implementing these basic operations. And we do not want to worry too much about having to do this kind of low-level accounting. So, in order to simplify our accounting, in some sense, we will focus on just the statement level execution of programs.

And we will be able to collapse multiple statements into one block saying, this block takes seven, seven steps. But it will always take only seven steps; it will not take steps proportional to the input. So, we will just count it as one block. So, we do not have to be very precise about defining the basic operations. And this is one motivation for looking at this running time in terms of orders of magnitude.

(Refer Slide Time: 24:50)

What is the input size



- Typically a natural parameter
 - Size of a list/array that we want to search or sort
 - Number of objects we want to rearrange
 - Number of vertices and number edges in a graph
 - We shall see why these are separate parameters



× 10 × 10 × 10 × 10 × 10 × 10 × 10

Madhavan Mukund

Analysis of algorithms

What is the input size



- Typically a natural parameter
 - Size of a list/array that we want to search or sort
 - Number of objects we want to rearrange
 - Number of vertices and number edges in a graph
 - We shall see why these are separate parameters
- What about numeric problems? Is n a prime?
 - Magnitude of n is not the correct measure

387
4367 $\rightarrow 10^x$

× 10 × 10 × 10 × 10 × 10 × 10 × 10

Madhavan Mukund

Analysis of algorithms

सिद्धिर्भवति कर्मजा

What is the input size



- Typically a natural parameter
 - Size of a list/array that we want to search or sort
 - Number of objects we want to rearrange
 - Number of vertices and number edges in a graph
 - We shall see why these are separate parameters
- What about numeric problems? Is n a prime?
 - Magnitude of n is not the correct measure
 - Arithmetic operations are performed digit by digit
 - Addition with carry, subtraction with borrow, multiplication, long division ...

237
895
112
1132



Madhavan Mukund

Analysis of algorithms

What is the input size



- Typically a natural parameter
 - Size of a list/array that we want to search or sort
 - Number of objects we want to rearrange
 - Number of vertices and number edges in a graph
 - We shall see why these are separate parameters
- What about numeric problems? Is n a prime?
 - Magnitude of n is not the correct measure
 - Arithmetic operations are performed digit by digit
 - Addition with carry, subtraction with borrow, multiplication, long division ...
 - Number of digits is a natural measure of input size
 - Same as $\log_b n$, when we write n in base b



Madhavan Mukund

Analysis of algorithms

So, the other thing that we have to be clear about is what constitutes the input size. So, we said of course that if you are manipulating lists and arrays; the number of elements in the list the size of the list is a natural parameter. The larger the list, the more time you are likely to take to search or sort or whatever. If we are rearranging objects, like we were looking at, for example, trying to find an optimum collection of objects to fit in our delivery truck; then the number of objects would be the natural thing. We have seen before and we will see in this course also graphs as a very powerful way of representing relationships.

And then algorithms on graphs have to manipulate these graphs. So, graph as you remember consists of these vertices and some edges between them. So, there are pictures like this, where

you have the nodes or the vertices connected by edges. Now, there are two natural parameters, there is the number of vertices, in this case 5; and then you have the number of edges, which in this case is also 5, not 6. So, there are six edges and five vertices. So, you might think that if I fixed the number of vertices, the number of edges anyway cannot grow beyond a certain level, because every edge has to connect two points.

And so, I only have a fixed number of points, the number of edges is bounded by the number of points squared roughly. But you could have graphs in which you have very few edges, and that actually makes a difference. So typically, it will turn out that when we are looking at graphs, the input size consists of the number of vertices, and separately the number of edges. Both of them are important for us, they are separate parameters. Another very typical class of problems that we are interested in when computing is problems about numbers. The most famous such question would be to identify whether a number is prime or not? So given n is n a prime.

So, how should we measure the complexity of an algorithm which claims to determine whether n is a prime? Should it be something proportional to the value of n ? So, if we say something is proportional to the value of n ; that means that if I give you some number like 387, and ask you whether it's a prime. And then I give you a number like 4364, 5; that is not going to be prime, okay, 4367 and onwards prime. So, this is right going from the top to the next number; so, this is more than 10 times. But, do you expect that it will take 10 times the effort to check whether a four-digit number is a prime compared to a three-digit number of prime.

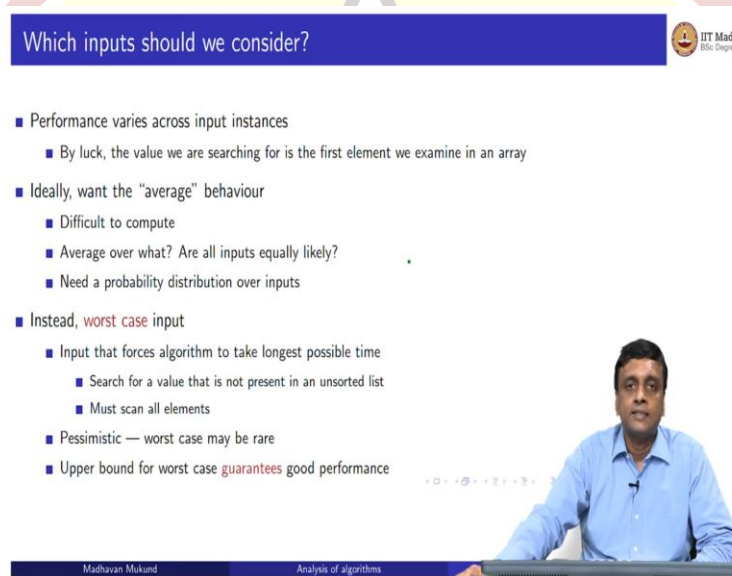
So, this is not typically a case and we know this; because when we do arithmetic, we now work on numbers, we know that we do not do this according to the magnitude, but according to the number of digits. So, if I add two numbers like this, then I will work from right to left; and I will do work proportional to the number of digits. So, I will say 9 plus 3 is 12, carry the 1 then I will say this is 11, carry the one and I will get 112. Now, if I had another digit on the right, then I would say 7 5 is 12, carry the one; then 9 plus 3 is 12 plus one carry the one; and I will get 1132, so this will be my new number.

So, it did not take me again 10 times the work, to go from adding two-digit numbers to adding three-digit numbers. It only took me one more column of work in terms of moving carries. So, addition, subtraction, multiplication, division, square roots, anything that you do, will typically

work based on the length of the number as represented in that format, say in decimal, or in binary, or whatever.

So, it turns out that we are interested really in that representation. So, we are really interested in number of digits. So, we want to say that as the number of digits grows, how does the thing grow? Not the value. So, the number of digits close by one, the value actually in decimal will multiply by 10. So, we really want it to be proportional to the logarithmic logarithm of the number that is the input size.

(Refer Slide Time: 28:52)



Which inputs should we consider?

- Performance varies across input instances
 - By luck, the value we are searching for is the first element we examine in an array
- Ideally, want the “average” behaviour
 - Difficult to compute
 - Average over what? Are all inputs equally likely?
 - Need a probability distribution over inputs
- Instead, **worst case** input
 - Input that forces algorithm to take longest possible time
 - Search for a value that is not present in an unsorted list
 - Must scan all elements
 - Pessimistic — worst case may be rare
 - Upper bound for worst case **guarantees** good performance

Madhavan Mukund Analysis of algorithms

So, now let us get back to the question that we had at the beginning, which is that I fixed the input size. But now if I look at an input of size n , there are very many different inputs of size n , and they all behave very differently. So, for instance, as I said before, if you are searching for an element in a list; it could be a very large list.

But you could just be lucky the value that you are searching for is at the beginning of the list. Or if you are using that binary search, it could be at the midpoint. So, with by just sheer luck on a particularly large input, your algorithm might actually execute very fast. So, ideally of course, we do not want to look at these extreme cases; these extreme good cases in particular, and be very optimistic about our algorithm.

What we would like to know is, if I try out different inputs, what is going to be the typical behavior across the different inputs. So, we would like some notion of average behavior, on an average how does it do? Now, unfortunately, this is a tricky problem. So, what is average mean? It means that I have to take the time. It takes for many different things and then you know take the mean; but first, I have to enumerate these many things. How does it how do I generate all the possible graphs that I will look for a graph algorithm for instance? How will I possibly generate all the possible different objects of size n with different values for their sizes?

So, even just enumerating all these inputs before checking how they do on these inputs is hard. Secondly, there is a question of whether all these inputs are actually equally likely? So, are you actually going to look for every possible combination of objects of different sizes? Or are some more typical than others? So, then we need a probability distribution also. Not only do we need to list out all the inputs, we also need to say what is the likelihood of each particular thing; and then we have to take technically what is called an expectation. We have to multiply the probability of each input multiplied by the time it takes for that input and add it up.

So, this is very hard to do, it is almost impossible; so that is why we cannot, in most situations, talk about average case. So, instead we go for an easier thing, which is the worst case? If we look at the algorithm, we can say this algorithm is going to get really stumped by this particular input, it is going to have to spend a lot of time to answer it.

So, a case in point is when I am searching for a value in a list when searching for a value in a list, then when do I get stuck? I get stuck, when it is not there in the list. I have to go through the entire process, to find out whether the value is there or not. So, if I am doing a linear scan, you know the one where it is not sorted, and I go from beginning to end.

Without looking at every value in the list, I cannot be sure that it is not there. I cannot just look at so supposing some your mother asked you to search for something in the kitchen; and you come back and say I looked on three shelves, and it is not there, they are not likely to get very far. So, you have to convince her that you looked in every possible shelf, and it was not there.

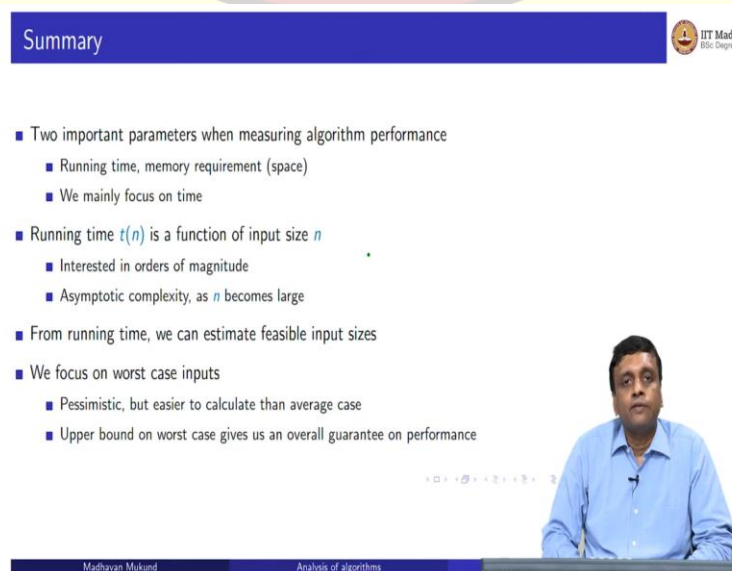
So, if you do not look at exhaustively at every elements, and something is not present; you will not be able to tell, so this is now a worst case. You know that this algorithm takes a maximum

amount of time; it must scan every element in my list, if the element I'm looking for is actually not in my list.

So, this is something which we can typically extract by looking at how the algorithm performs. We can look at the algorithm and say, this is a worst case. So, now if we can do this worst-case analysis, this gives us some idea about how our algorithm does. It says that there are situations where it does this badly. The problem is this is pessimistic, because this worst case may not actually happen many times; so, it might be a rare case. But, the good thing about doing a worst-case analysis is you can actually get a reasonable upper bound, like those $n \log n$ cases. Even if all the SIM cards had invalid Aadhaar cards; the algorithm that we looked at there, we claim will take $n \log n$ time.

Because it will take $\log n$ time per SIM card to determine that the Aadhaar card Aadhaar number is not there in the other database, and we do this for n . So, if you can get a good upper bound for the worst case that means that overall, you have a good upper bound. So, that is one reason why we are quite interested in worst case analysis. When we are able to prove good upper bounds, worst case tells us a lot. When we are not able to prove good upper bounds, worst case may or may not be a realistic estimate of how good or bad an algorithm is. There are many algorithms which have not so good worst cases, but which work well in practice; because the kind of worst cases where they do badly, do not occur very often.

(Refer Slide Time: 33:17)



Summary

- Two important parameters when measuring algorithm performance
 - Running time, memory requirement (space)
 - We mainly focus on time
- Running time $t(n)$ is a function of input size n
 - Interested in orders of magnitude
 - Asymptotic complexity, as n becomes large
- From running time, we can estimate feasible input sizes
- We focus on worst case inputs
 - Pessimistic, but easier to calculate than average case
 - Upper bound on worst case gives us an overall guarantee on performance

Madhavan Mukund Analysis of algorithms

So, to summarize, there are two important measures that parameters that we look at, when we are thinking about the performance of our algorithm, running time and the space required memory required. But we will mainly focus on the time for because that is really more fundamental for us. So, we said that the running time is not just an absolute value; it is a function of the input size. So, we are interested in how long the algorithm takes on an input of size n , and we are interested in this up to some order of magnitude. So, we will ignore constants, we will kind of collapse things into basic operations; and not worry too much about what constitutes a basic operation.

And finally, we are looking at this thing asymptotically. So, we are not interested in the performance for a fixed n , but rather what happens as n becomes large. So, as this function, as this algorithm works on larger and larger inputs, how does the behavior change? So, one of the things we saw in that table was that if we know something about the upper bound on the running time of the algorithm. It also tells us for what classes of inputs, it will be feasible. So, it is a very important part of using an algorithm to be able to tell what its running time is; because only when you know what is running time is.

Will you know whether the problem we are trying to solve has a feasible solution using this algorithm or not? And finally, we said that there are many inputs of a given size, and we can hope in some situations to come up with some kind of average performance, but this is very hard. So, instead we will typically look at what is called the worst case; so we will reverse engineer from the algorithm. The input that takes the longest time, and based on this we will give an upper bound on the running time of the algorithm.