Design and Analysis of Algorithms, Chennai Mathematical Institute

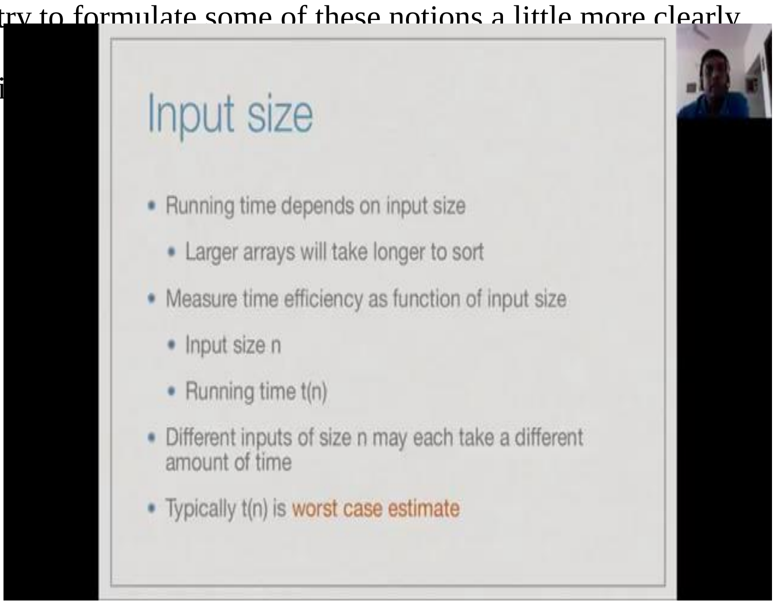
Prof. Madhavan Mukund

Department of Computer Science and Engineering,

Week - 01
Module - 06
Lecture - 06

So, we argued that we would like to measure efficiency of an algorithm in terms of basic operations, and we would like to compute the running time of an algorithm in terms of a function of its input size n, and we also saw that if you go from say n square to n log n then the size of inputs you can effectively handle becomes dramatically larger. Now, today we will try to formulate some of these notions a little more clearly.

(Refer Slide Ti

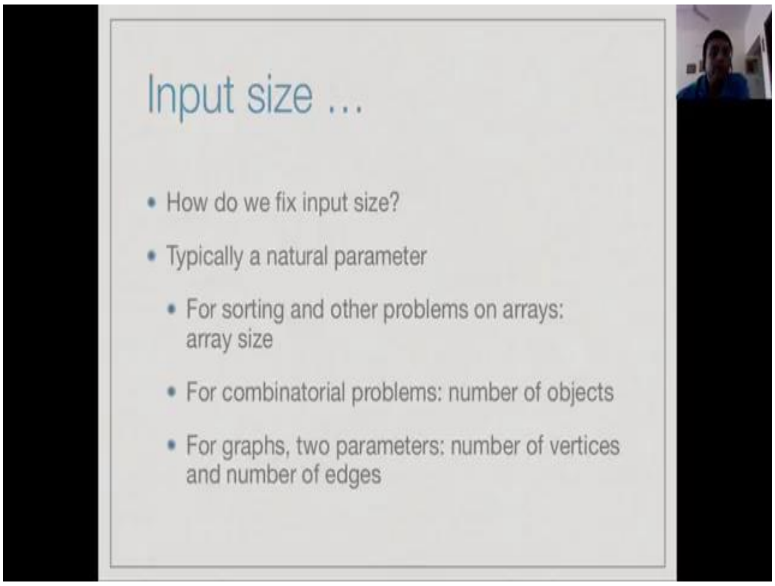


Input size

- 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

So, the first thing is the input size. So, remember that the running time of an algorithm necessarily depend on the size of the input. So, we want to write the running time as some function t of n. And the main thing to remember is that not all inputs of size n will give the same running time. So, there is going to be a notion of worst case estimate which you will need to explain and justify.

(Refer Slide Time: 00:54)



Input size ...

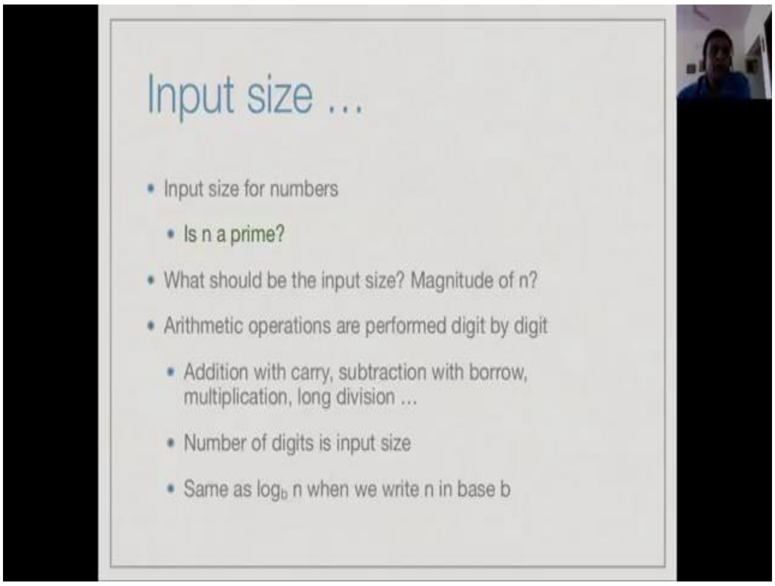
- How do we fix input size?
- Typically a natural parameter
  - For sorting and other problems on arrays: array size
  - For combinatorial problems: number of objects
  - For graphs, two parameters: number of vertices and number of edges

Before we do this let us look at the notion of input size itself - how do we determine the input size for a given problem? So, the input size more or less represents the amount of space it takes to write down the distribution of the problem or becomes a natural parameter of the problem. So, for instance, when we are sorting arrays what really matters is how many objects there are to solve, so we have to move them around and rearrange them. So, the size of an array is quite a natural notion of input size for a sorting problem.

On the other hand, if you had trying to do something like rearranging elements or take say we have some items which we need to load into a container and we are looking for a optimum subset load in terms of weight or volume then the number of objects would be a natural input parameter. We saw in one of the early lectures an example of air travel where we constructed a graph of an airline route map where the nodes where the cities and the edges where the flights.

And we argued that both the number of cities and the number of flights will have an impact on any analysis we need to do. So, this is a general property of all graphs; if we have a graph then both the number of nodes or vertices and the number of edges will determine the input size.

(Refer Slide Time: 02:14)



Input size ...

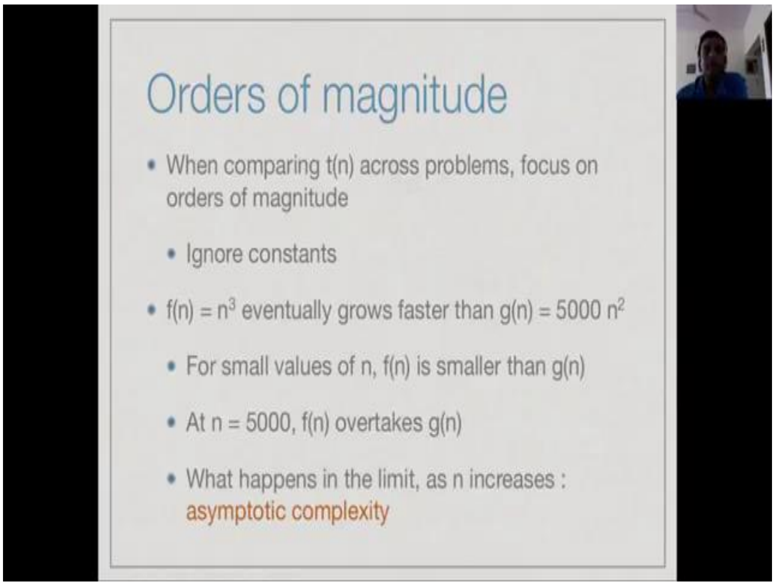
- Input size for numbers
  - Is n a prime?
- What should be the input size? Magnitude of n?
- Arithmetic operations are performed digit by digit
  - Addition with carry, subtraction with borrow, multiplication, long division ...
  - Number of digits is input size
  - Same as $\log_b n$ when we write n in base b

Now, there is an important class of problems where we have to be a little bit careful about how we talk about input size, and these are problems involving numbers. Suppose we were to write an algorithm for primality checking whether the given number is prime. Now, how should we think of the size of the input? For instance, suppose we ask it to solve the question for say 5003 and then for 50003 then 50003 is roughly 10 times 5003. So, would we expect the time to group proportional to 10. So, should the magnitude of n actually p ignores the input size.

Now, it is quite obvious when we do arithmetic by hand, the kind of arithmetic we do in school, that we do not go by magnitude, we go by the number of digits. When we do arithmetic such as addition with carry we right down the numbers in columns then we atom them column by column. So, the number of digits determines how many columns we have to add. The same is true with subtraction or addition or multiplication or long division and so on.

So, clearly, it is a number of digits which matters. And the number of digits is actually the same as the log. If you have numbers written in base 10 then if we have a 6 digit number its log is going to be 5.something. And we go to log 6, we will have a 7 digit number and so on. So, the number of digits is directly proportional to the log, so we can think of the log of the number as the input size. So, this is a special case. So, for arithmetic function with log in numbers, it is not the number itself which is the input size, but the size of the number as expressed in how many digits with excess to write down the number.

(Refer Slide Time: 04:00)

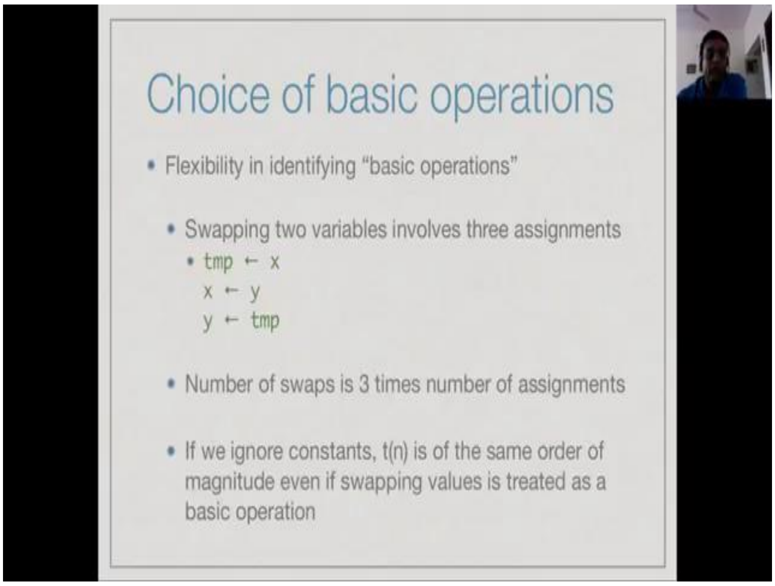


Orders of magnitude

- 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

Now, the other thing, we mentioned this that we are going to ignore constants. We are going to look at these functions in terms of orders or magnitude, thus the function grow as n, n square, n cube, and so on. So, one justification for this is because we are ignoring the notion of a basic operation or rather we are being a bit vague about it.

(Refer Slide Time: 04:20)



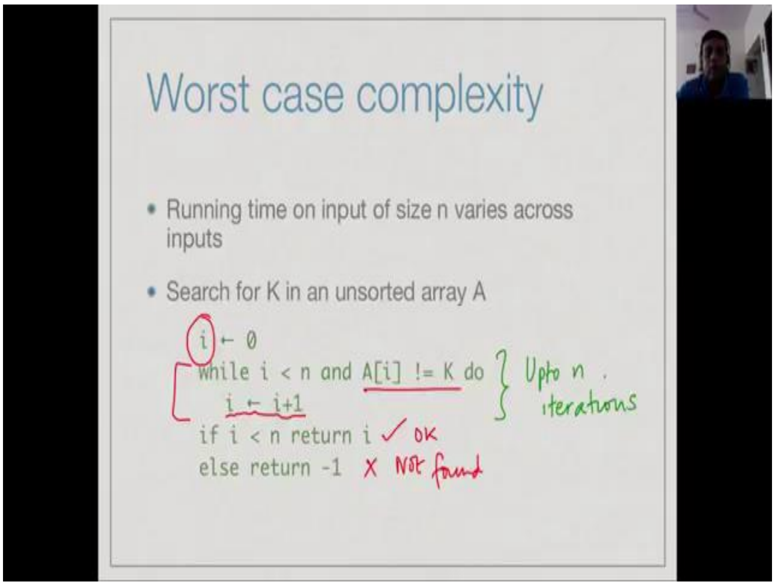
Choice of basic operations

- Flexibility in identifying "basic operations"
  - Swapping two variables involves three assignments
    - tmp ← x
x ← y
y ← tmp
  - Number of swaps is 3 times number of assignments
  - If we ignore constants, t(n) is of the same order of magnitude even if swapping values is treated as a basic operation

So, let us look at an example. So, supposing, we would originally consider our basic operations to be assignment to variables or comparisons between variables. Now, we decide that we will include swapping 2 values exchanging the contents of x and y as a basic operation. Now, one of the first things we learn in programming is that in order to do this we need to go via temporary variables. So, in order to exchange x and y in most programming languages you have to first say x in temporary variable then copy y to x and then this store y from the temporary variable; this takes 3 assignments.

So, if we take swap as a basic operation in our language as compared to a calculation where we only look at assignments we are going to collapse 3 assignments into 1 basic operation. So, there is the factor of 3 differences between how we would account for the operation if we account for swap as a single operation. So, in order to get away from worrying about these factors of 3 and 2 and so on, one important way to do this is to just ignore these constants when we are doing this calculation. So, that is in other motivation for only looking at orders of magnitude.

(Refer Slide Time: 05:30)



Worst case complexity

- Running time on input of size n varies across inputs
- Search for K in an unsorted array A

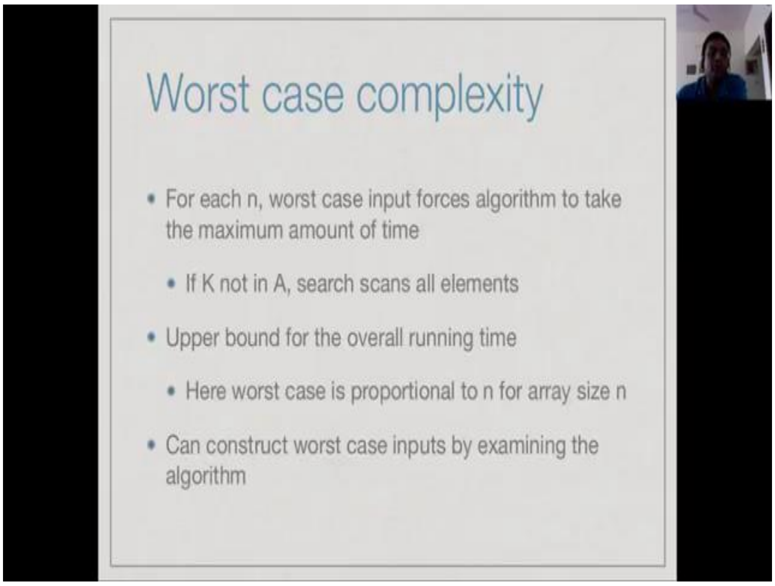
```
i ← 0
while i < n and A[i] != K do
    i ← i+1
if i < n return i
else return -1
```

So, let us come back to this notion of worst case. So, as we said we are really looking at all inputs of size n; and, among these inputs which inputs drive the algorithm to take the maximum amount of time. So, let us look at a simple algorithm here which is looking for a value k in an unsorted array A. So, in an unsorted array we have no idea where the value k can be. So, the only thing we can do is swap the beginning to i.

So, we start by initializing this index i which is in position to 0, and then we have a loop which says, so along as we have not found the array element. So, along as a i is not k we increment, right, we move to the next element. So, this is the loop, and when we exit this loop there are 1 or 2 possibilities, either we have found the element in which case i is less than 1, or we have not found the element in this case i is become n.

So, we check whether i is less than n; if i is less than n then we have found it, and if i is bigger than n which means it is not found. So, this is the simple algorithm. So, now, in this algorithm the bottle neck is this loop, right. So, this can take upto n iterations. Now, when will take n iterations? That is the worst case, right.

(Refer Slide Time: 06:47)



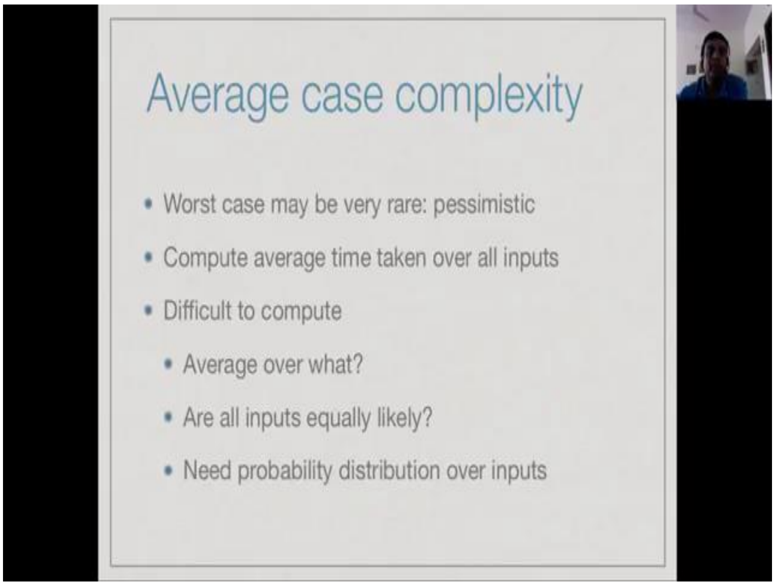
Worst case complexity

- For each n, worst case input forces algorithm to take the maximum amount of time
  - If K not in A, search scans all elements
- Upper bound for the overall running time
  - Here worst case is proportional to n for array size n
- Can construct worst case inputs by examining the algorithm

So, the worst case in this particular algorithm is it must go the end either if the last element is k, or more generally if there is no copy of k in the array. If there is no k in the array we have to scan all the elements to determine that k does not exist. So, this becomes our worst case input. So, it is important to be able to look at an algorithm and try to reconstruct what input to drive it to take the maximum amount of time. So, in this simple case, this simple example we can see that the case which forces us to execute the entire loop can be generated by choosing a value of k which is not in the array A.

And in this case, therefore the worst case is proportional to the size of the array n. The crucial thing to remember is that in order to determine which is the worst case input, we have to understand the algorithm and look at it. We cannot just blindly determine what is the worst case without knowing the problematic act. To different algorithms we have to come up with different worst cases depending on what the algorithm is supposed to do, and how the algorithm is constructing. So, the worst case input is the function of the algorithm itself.

(Refer Slide Time: 07:55)



Average case complexity

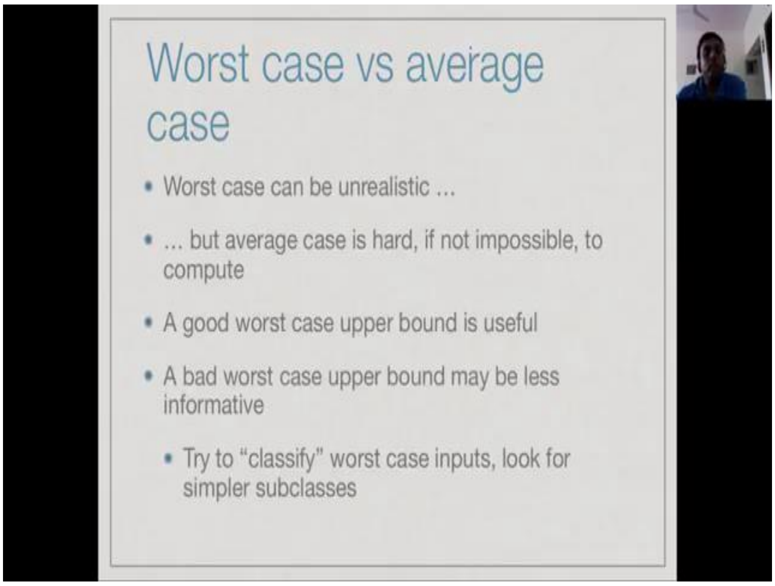
- Worst case may be very rare: pessimistic
- Compute average time taken over all inputs
- Difficult to compute
  - Average over what?
  - Are all inputs equally likely?
  - Need probability distribution over inputs

Now, we could look at a different measure, right. So, supposing we do not look at the worst case, we say, look at the average case, right. So, look at the all possible inputs and then try to see how much time it takes when each of the inputs are somehow average in time. Now, mathematically in order to compute this kind of an average we need to have a good way of estimating what are all the possible inputs to a problem. So, although this sounds like a very attractive notion, in many problems is very difficult.

So, supposing we are doing the airline route problem, how do we consider this space of all possible route maps, and what is a typical route map, and so on. So, what are we going to average over are all this inputs equally likely, so we need look at probabilities. And it is very often very difficult to estimate the probabilities of different types of inputs.

So, though it would make more sense from the point of view of the behavior of the algorithm in practical situations look at the average case that is how does it behave over a space of inputs. In practice, it is very hard to do this because we cannot really quantify this space of possible inputs and assign them with meaningful probabilities.

(Refer Slide Time: 09:09)



Worst case vs average case

- Worst case can be unrealistic ...
- ... but average case is hard, if not impossible, to compute
- A good worst case upper bound is useful
- A bad worst case upper bound may be less informative
    - Try to "classify" worst case inputs, look for simpler subclasses

To summarize, we look at worst case even though it could be unrealistic because the average case is hard if not impossible to compute. There are very limited situations where it is possible to do an average case analysis, but these are very rare. So, the good thing about a worst case analysis is if we can do a good upper bound, so in that, even in the worst case if algorithm performs efficiently then we have got a useful piece of information about the algorithm; that is this always works well.

On the other hand, if we find out that this algorithm has a bad worst case upper bound we may have to look a little further, how rare is this worst case, does this often arise in practice, what type of inputs are worst case, are they inputs that we would typically see, are there any simplifying assumptions that we can make which will rule out these worst cases and so on.

So, though worst case analysis is not a perfect way of doing it, it is something which is mathematically practical, it is something that we can kind of hope to compute. So, that is one good reason for doing it so that we actually come up with a quantitative estimate. And secondly, it does give us some useful information; even though in some situations it not be, may not be the most realistic situations that we have likely to come across in practice.