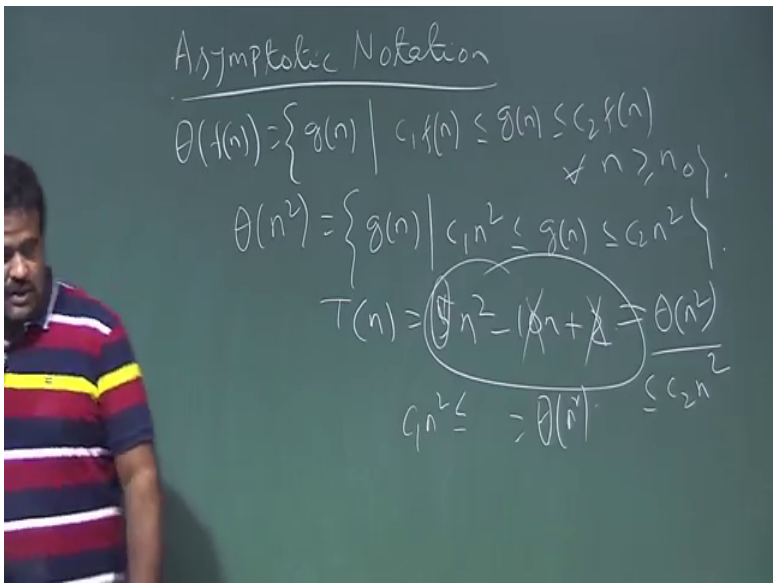
An Introduction to Algorithms
Prof. Sourav Mukhopadhyay
Department of Mathematics
Indian Institute of Technology, Kharagpur

Lecture – 03
Asymptotic Notation

(Refer Slide Time: 00:26)

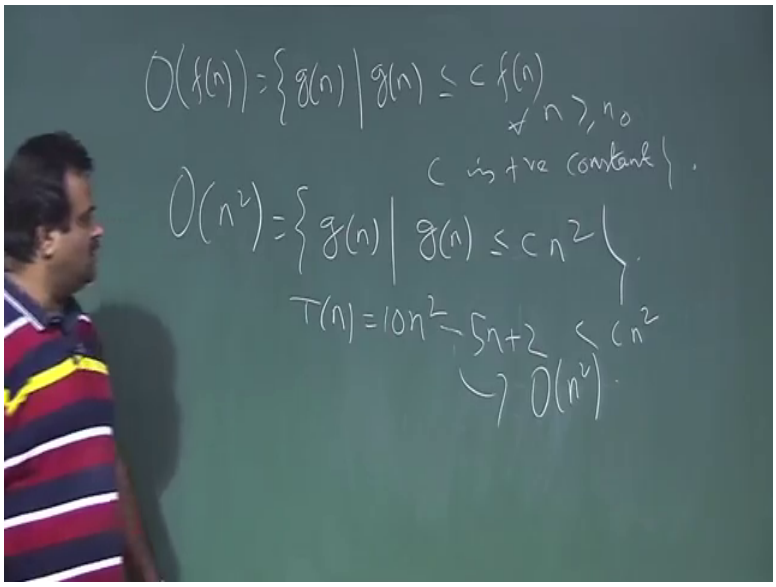


So, we are talking about asymptotic notation. So, we had big theta notation. So, we define the big theta of f of n as this is the mathematical definition of big theta. So, it is basically set of all function g n such that. So, g n is bounded both side by f n for all n greater than some n 0. So, this is the mathematical definition of it. So, for example, if we take the big theta of n square, this is basically set of all function which should be bounded by n square by both side. So, this is c of n square for all n greater than where c 1 c 2 are 2 constant.

So, for example, we have taking a. So, if our time complexity is T n say 5 n square minus 10 n plus 2. So, for this we can see that we can choose some constant suitable is c 1 c 2 for some after sum n, such that we can always. So, this is this is bounded by both c 1 n square less than equal to this less than equal to c 2 n square we can always choose c 1 c 2 some suitably some large constant or, so then this is basically belongs to big theta of n square, but we will just write this is big theta of n square and engineering sales what we have mention. So, we just ignore the all lower order term, because higher order term is n

square we all ignore lower order term there by ignore the leading coefficient. So, this is basically n square. So, that is big theta of n square, but we do have a mathematical definition.

(Refer Slide Time: 02:39)

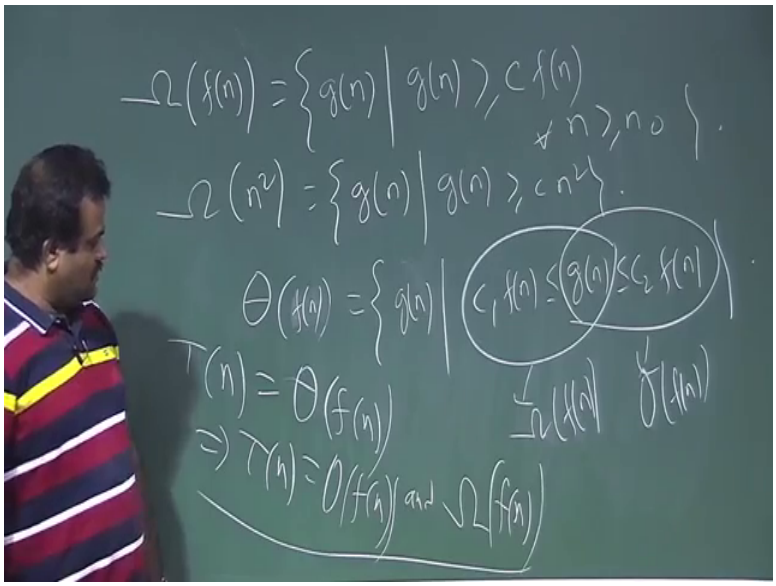


Now, we talk about big o. So, what do means by big O of n square or big O of n cube big O of any functions. So, big O is basically bounded above. So, sorry big O of f n is basically set of all function g n such that g n will be less than some c into f n, and this is true for all n greater than equal to c 0 and my c is a positive constant positive constant ok.

So, basically it is a upper bound. So, if we have a such a upper bound then say for example, big O of n square basically set of all function such that it is bounded above by n square. So, if we can choose a c such that positive c such that it is less than this any function like this say 10 n square plus 5 n plus 2 like this. So, our may be say minus 5 n like this.

Now, if we choose c to be for this example if we choose c to be greater than 10 then this can be shown as less than c n square so that means, this is basically big O of n square. So, this is the big O notation this is the upper bound. So, if our time complexity is this then this is basically big O n square now we do have another notation this is called big omega.

(Refer Slide Time: 04:26)



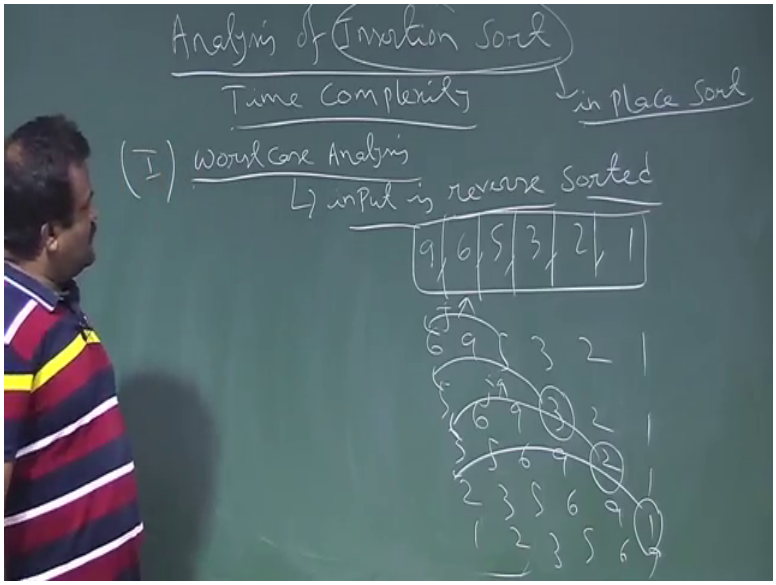
So, big omega of f n which is basically the lower bound, which is basically set of all function g n such that g n is greater than from c of f n, and this is true for all n greater than equal to n 0 this is the mathematical definition where c is a constant this c is a constant c is positive constant as we know. So, big omega of n square for example, is set of all function which is bounded below the n square, some c n square ok.

So, basically big theta is basically what big theta is basically big theta of n square or any other function is basically set of all function such that it is bounded both side by this function say any function f n c of f n less than equal to g n, less than equal to c 2 of f n now this is big theta now if we have. So, this if we have this this is big O big O of f n and if we have this only this is big omega of f n. So, if we have big theta means both big O and big omega. So, if T n is a big theta of say some f n this implies T n is both big O of f n, and big omega of f n.

So, big theta is basically both side bounded and this big O is basically upper bound and this is the lower bound. So, we usually go for upper bound because if we say that our time complexity is greater than 5 second for example, than that does not make any sense greater than 5 second does not mean that it will be less than something, but if we say our time complexity is less than 10 second something like that or less than some n square. So, that has some sense. So, usually we go for the upper bound I mean the big O of big theta of sorry big O of n square ok.

So, these are the asymptotic notation we will use for our time complexity, because this is to help us to get rid of from the machine dependencies.

(Refer Slide Time: 07:34)



So, now, with the help of this asymptotic notation we will analyze the insertion sort worst case we will analyse the insertion sort all the cases using this asymptotic notation. So, let us analyse the insertion sort, time complexity basically analysis of insertion sort.

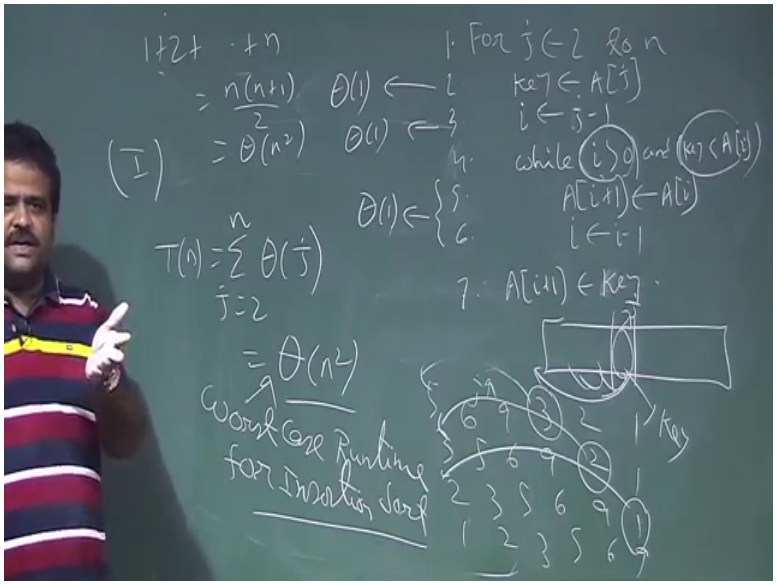
So, basically the time complexity. So, runtime time complexity means runtime and there is another one the space complexity. So, insertion sort space complexity means we are we in insertion sort we are using, we have given array, and we are only we are taking any extra array for that. So, it is a in place sort. So, insertion sort is a in place sort. So, in place sorting algorithm; that means, it is sorting in the array itself we are not taking help of any extra array to sort it to run the code, we just taking on key so that is not compare to a n n.

So, just one key we are taking we are taking some variable like i j. So, that is one or 2 variable. So, that is that order of theta of one size. So, that is why it is called in place sort; that means we are not taking any extra memory for running this code. So, we are sorting everything inside the array whatever array we have we are managing their. So, this is not taking any extra memory to run. So, now, that is the space complexity now we talk about time complexity. So, first, we know that we have to we do 3 types of analysis. So, first one is worst case analysis for insertion sort ok.

So, what is the worst case analysis for insertion sort? For insertion sort worst case is if the input is reverse sorted. So, for insertion sort worst case input is reverse sorted; for example, if we have say input like 9 reverse sorted 6, 5, 3, 2, 1 if this is our input now if you run this insertion sort on this input. So, you start with J is equal to this. So, we compare this. So, J as to 6 as to come here 9 5 3 2 1 now J is pointing here. So, then 5 as to come here, so5 6 9 3 2 1. So, J is this now 3 as to come here, so 3 5 6 9 2 1. So, J is this, now 2 as to come here. So, 2 3 5 6 9 1 now one as to come here, finally, we got 1 2 3 5 6 9.

So, for insertion this is the execution of the insertion sort for a reverse sorted input. So, for every word he as to come for what to the beginning so that is the worst case, so that is why it is taking maximum time. So, now we want to analysis this. So, if we have this for loop we know this is we have a for loop. So, what is the time complexity? So, time complexity basically summation of the.

(Refer Slide Time: 11:01)



Let us just. So, if we just write the insertion sort code again this is basically for J is equal to 2 to n key is equal to a J i is equal to J minus 1 and then we have a while loop while i is greater than 0 and key is less than A i then only we run this then A i plus 1 is A i and then i is equal to i minus 1. So, this is 1 2 3 4 5 6 and 7 is after this while loop A i plus 1 equal to key.

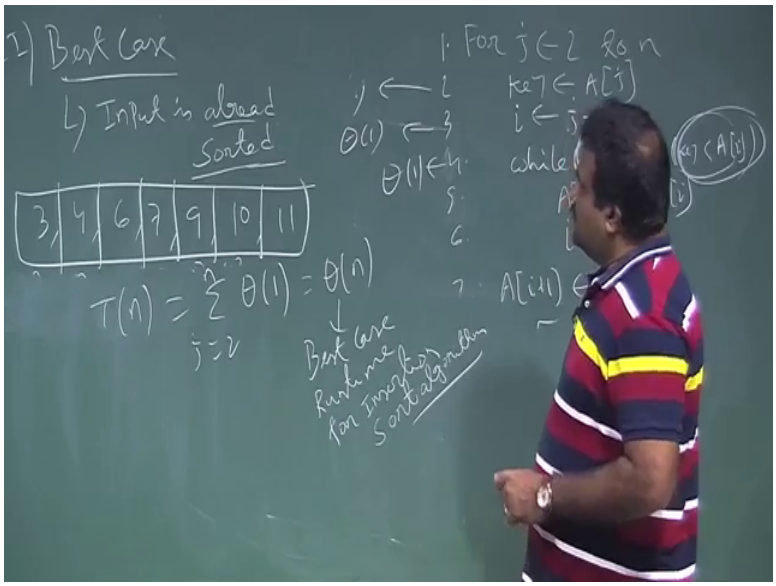
So, this is the code for the insertion sort, now to find the time complexity of it. So, we have a for loop. So, this is starting from J is equal to 2 to n. Now here we are doing something. So, this this is the assignment we are doing this is the worst case. So, this is the assignment we are doing. So, this is taking theta one time again we do not clear about how much time our how much time our hardware is taking. So, this is also theta one and here these 2 is also taking theta one time because here we are just assigning something again this is theta one this is theta one, but size of this loop is order of J for a given j. So, if you fix the J suppose this is our j, this is our J and this is our key.

Now, this key is comparing with this with this this this this the this key is coming long way to the beginning, and every time we are spending constant effort because that is come under theta one, but if we are machine dependent then it will take some different different machine we will take different different time, but we do not bother about that. So, we will just thing that this is taking some constant time some c this is not related to n. So, that is why it is basically theta of because J times theta of 1. So, for each if we fix the J since J has to come long way to the beginning.

So, every time it is comparing and doing something. So, those whole thing it as the theta of one and then, this is basically theta of J, now this is basically a arithmetic series like 1 plus 2 plus 3 plus up to n. So, this is basically n into n plus 1 by 2. So, this is theta of n square. So, this will give us theta of n square. So, this is the worst case runtime for insertion sort; worst case runtime for insertion sort this is in asymptotic sense asymptotic notation because we do not want the exact one, because exact one is a headache because it will be machine dependent things.

So, we do not care how much time exactly we are spending here, how much time we are spending this i greater than 0 checking, how much time are machine (Refer Time: 14:55) taking. So, we just take this as a constant time theta of 1. So, that way it is the asymptotic analysis. So, this is the worst case runtime for insertion sort. So, now, you want to do best case I yeah yeah best case for analysis for insertion sort. So, what is the best case?

(Refer Slide Time: 15:30)



So, this is the second is a best case best case analysis for insertion sort. So, when is the best case if the input is already sorted this is the best case for insertion sort if the input is input is already sorted. So, if the input is already sorted means what? If it is say for example, say if your input is 4 3 4 6 7 9 10 11 suppose this is our input suppose this is our input. Now if we want to run our insertion sort on this input so, on this this piece of code. So, we start with J is equal to 2 then for that i is pointing here. So, we compare this this is greater than. So, we are entering into this. So, this will stop here so, but we are doing some comparison over here. So, it is not entering here, but this will taking some time this check and this check at least e are doing this check. So, this is, but again this is constant time we are spending. So, that is also theta of 1.

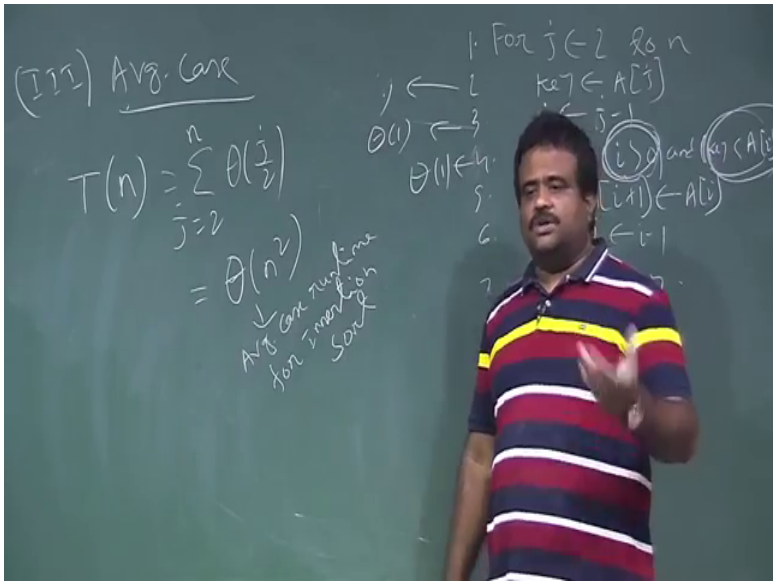
So, this is the i and this is J this pointing. So, we are not doing anything now J is pointing here. So, just a one comparison we are doing. So, theta one time we are spending for this. So, again nothing J is pointing here J is pointing here nothing J is pointing here i is pointing here nothing then J is constant. So, basically time is basically T n is basically summation of J is equal to 2 to n and every time we are just doing the one comparison, I mean just to check whether this key is and that will theta of runtime machine dependent independent I mean asymptotic sense ok.

So, this is basically theta of n. So, this is the best case runtime for insertion sort best case runtime this is linear, but this is only occur if the input is sorted; best case runtime for

insertion sort algorithm. So, this is occurring only if the input is sorted not for any other input. So, this is the only linear time linear time, but this is a very special case when the input is already sorted so, but the worst case is n square ok.

So, now we talk about another analysis which is called average case analysis. So, this is a best case which is big theta basically this is big O and big omega also.

(Refer Slide Time: 18:52)



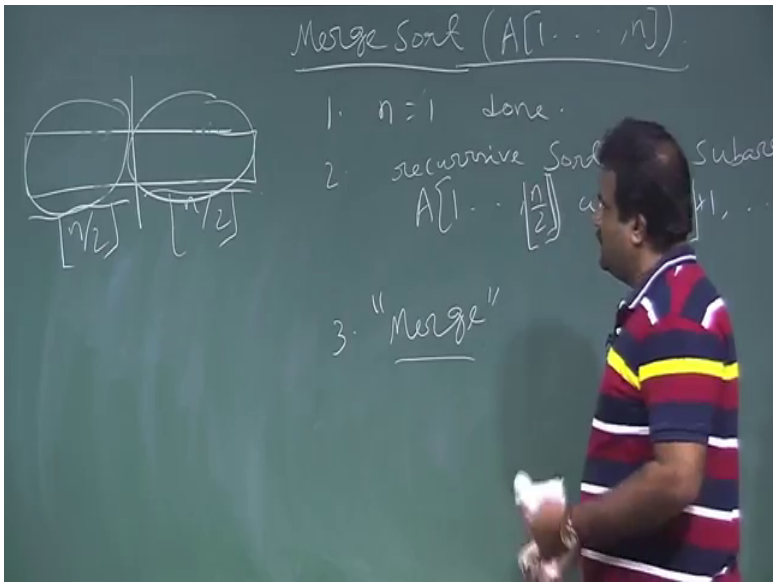
So, now, we talk about the third one which is the average case analysis for the insertion sort. So, average case when you do the average case analysis for any algorithm then we must have a something called some randomize algorithm; that means, we must choose the input some random randomization will be there, will talk about this in more detail. So, we will talk about randomize quick sort. So, any way, this is, but here intuitively we can just think that. So, worst case this is a J loop. So, this is basically summation of we have a outer loop J is starting from 2 to n. So, in a worst case J is coming all the way to the beginning and we are inserting that key, and in the best case J is just comparing 1.

So, in the average case we can say J is coming half way. So, this will be theta J by 2, but this is the intuitively analysis it is not exact average case analysis one should do because for that we need to take the distribution of the input pattern then we must talk about expected value of T n, but this is roughly intuitively we can say and like that the J is coming half way to the beginning. So, this is again will be because this is arithmetic

series just half will take common this is again theta of n square big theta of n square so; that means this is big O and big omega of n square.

So, this is basically average case runtime for average case for inserting sort. So, now, we have see inserting sort which is having the best case is linear which is theta one, but that is for worst case that is for particular input, it is not guaranteed for any other input this is the minimum time average case is also n square and the worst case is n square . So, now, we talk about a algorithm sorting algorithm where worst case is better than n square. So, that is called merge sort. So, how much time here? So, in merge sort this is also another sorting algorithm.

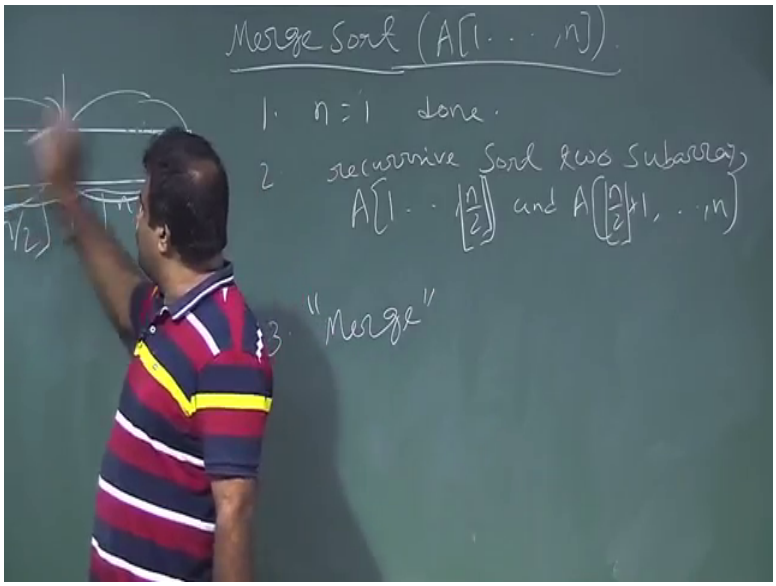
(Refer Slide Time: 21:29)



So, we have given a array of size n. So, the idea is this is a type of technique which is called divide and conquer technique. So, we have a problem of size n we divide the problem into sub problems of lesser size, and then we solve the sub problem recursively that is the conquer step and then we combine the solution of the sub problems to get a solution of the whole problems that is combine step, we will talk about in more details divide and conquer technique, but this is the one example of such a technique. So, basically we have a array of size n what we are doing? We are just dividing the array into 2 parts n by 2 n by 2 provided n is even or not, if n is not even we must put a lower sealing or upper sealing and you have you can (Refer Time: 22:30) talk a little, but here.

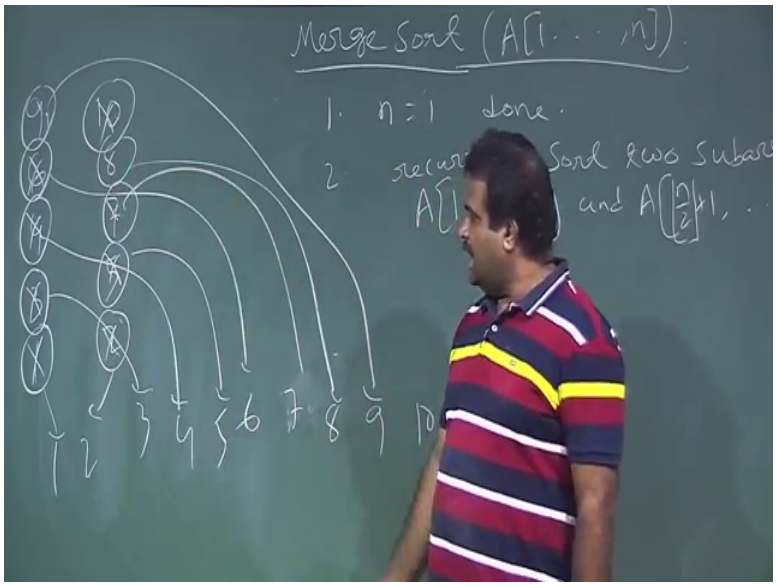
Now,. So, we have this sub array this sub array now we sort this sub array we sort this sub array we get simplified calling the same function merge merge sort, and then once we got the solution of this sub array once we see this is sorted once this is sorted then we call a combine step that is called merge sub routine we call a function merge we still take this to sorted array and give us the full sorted array. So, that is called merge let us write the code so; obviously, if n is equal to one you are done you written else what we are doing we just recursively sort 2 sub arrays recursively by calling the same function one sub array is 1 to n by 2 and A n by 2 plus 1 to n, but to be very precise we must put a ceiling over here because T his may not be always integer, but we were doing asymptotic analysis. So, this is. So, this is we just sort this sub array we are dividing the array into 2 part we sort this sub array we sort this sub array, then once we have the solution of this 2 sub array we call the what is called merge sub routine ok.

(Refer Slide Time: 24:01)



So, this is code for merge sort. So, this is the recursive calling. So, we call again to sort this we call again the merge. So, again it will divide into 2 sub array sub array. So, again it will divide sub array. So, this while will go down until we reach to the n is equal to one, once we reach to the n equal to one we merge this. So, the merge call will be log n the height of the tree like this anyway. So, now, we talk about this merge what is the merge sub routine. So, merge what it is taking what it is doing it is taking the 2 sub array 2 sorted sub array and it is giving on the whole array.

(Refer Slide Time: 24:51)



So, for example, suppose we have 2 sorted array say 9 6 4 3 1 and another thing say 10 8 7 5 and say 2 2 sorted array and they may not be equal and so this is the sorted array this is the sorted sub array.

Now, we call merge. So, for merge we what we take we point the low least element of each of this array. So, least element will be this element, now we compare this to least now whichever is the least among this will return it. So, will return one and we point the next least element of the array now we compare this two. So, 2 is the least we output 2. So, 3 is the least we output 3 4 5 6 5 is the least, we point this say 6 is the least 9. So, 7 is the least 8. So, 8 is the least. So, this is 7, 8 and then 10. So, 9 10 9 is the least and then 10.

So, this is the merge sub routine. So, this is the merge sub routine. So, basically we have 2 sorted array, and this merge is taking the merge is taking the input of this 2 sorted array and giving us the whole array. So, we will talk about the time complexity and the space complexity of this algorithm in the next class.

Thank you.