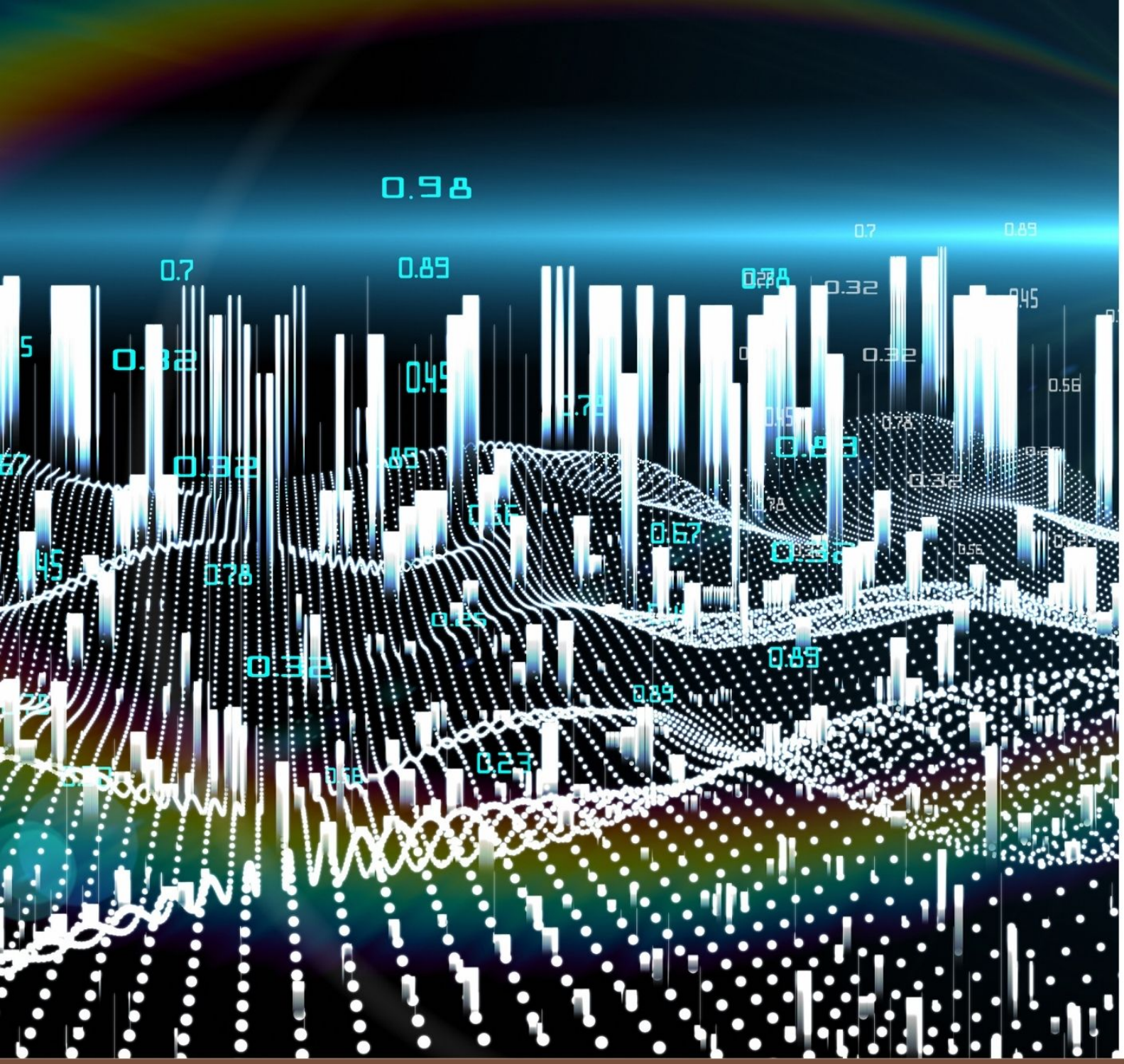


INTRODUCTION TO ALGORITHMS AND ANALYSIS

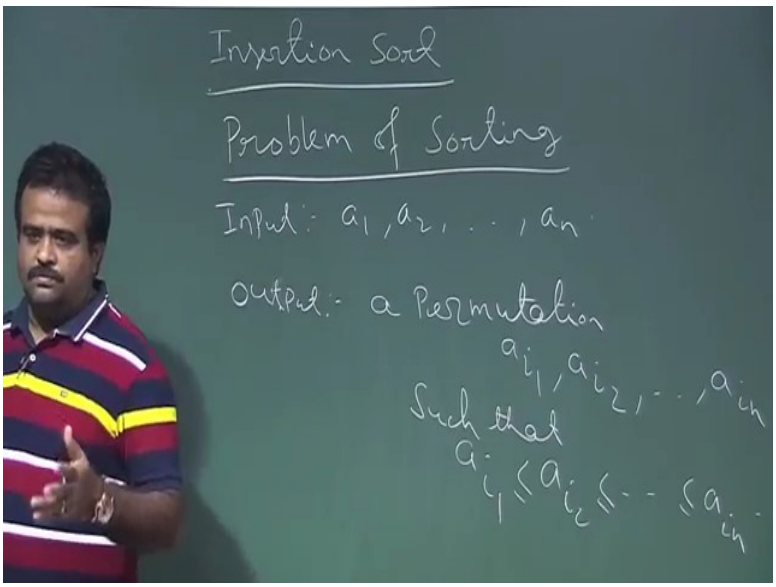
Prof. Sourav Mukhopadhyay
Computer Science andEngineering
IIT Kharagpur

INDEX

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

Lecture – 01
Insertion Sort

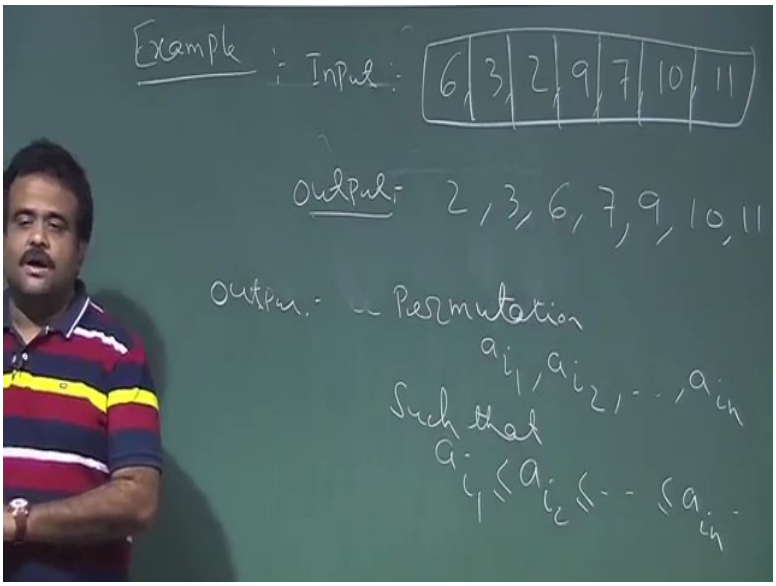
(Refer Slide Time: 00:25)



So, we start with the sorting algorithms or sorting problems. So, what is the problem of sorting? So, we have given n numbers as input say a[1], a[2], … , a[N]. Typically they are integers. We can deal with real numbers also (wherein we have to deal with real number operations) but these are typically integers. We have given an array of n numbers and the output will be the permutation of numbers a[1], a[2], … , a[N] such that it is sorted.

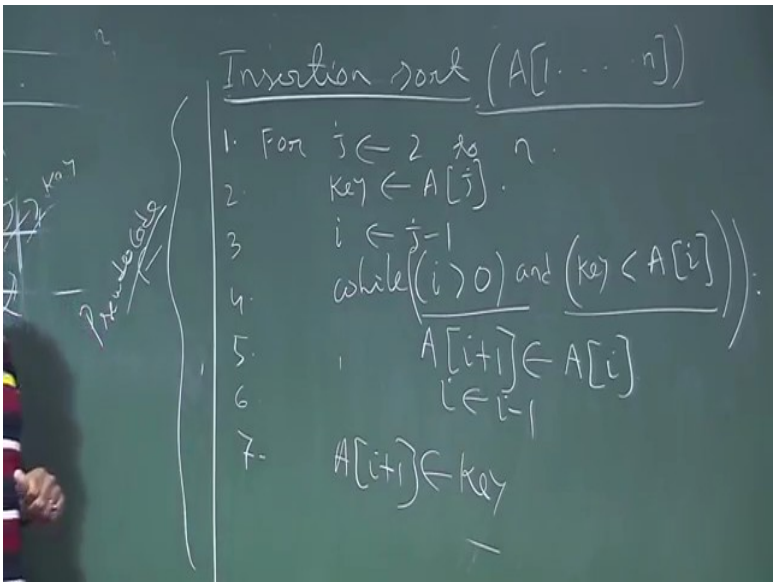
So, there are N! permutations. Among these we have to choose one permutation, that which is giving the sorted array. So, this will be the output of any sorting algorithm. So, this is in ascending order.

(Refer Slide Time: 02:00)



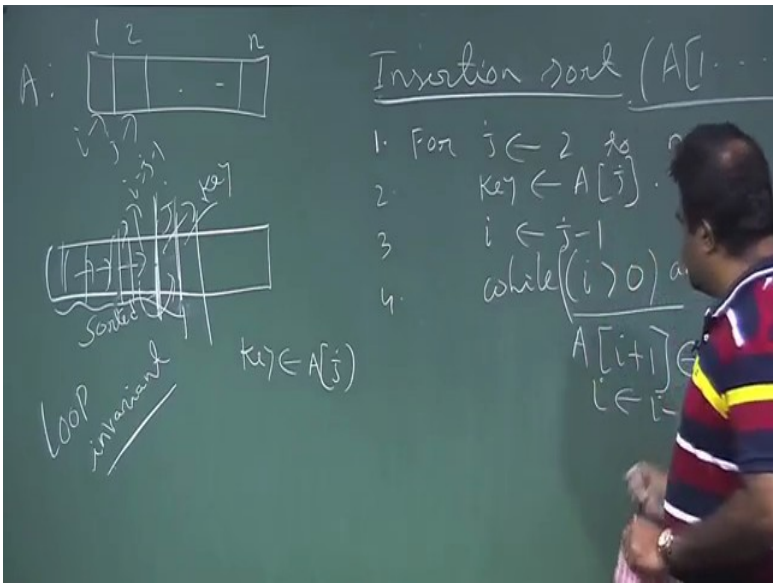
So, If we have input, say 6 3 2 9 7 10 11. we have 7 elements. So, you have a array of 7 element. Then, after sorting output will be the sorted array. So, it is basically a permutation on the array. We should get a permutation of elements which is sorted. We will discuss some sorting algorithms by which we should be able to get this output. So, we start with insertion sort.

(Refer Slide Time: 03:22)



Now, what is the input? Input is an array of N numbers.

(Refer Slide Time: 03:46)



So, we have an array of n numbers. Our array is starting from 1 to N(not zero to n minus 1 like in C language as we are not dealing with any specific language). We have N numbers and we need to sort these numbers. We need to write description of steps to get the required output. We can either write it in normal English form or we can write something in a compact way(this form is called pseudo code).

So, we will write pseudo code for this. First start with a j-loop, j is starting from 2 to N and then we take the key a[j]. Now this loop is starting from 2 to N. Now in the second step, we choose i = j-1.

Now, we want that at some point of time j will be pointing to the key a[j], and the part before j (all a[i] such that i<j) will be sorted. We want this property to be loop invariant i.e. wherever j will be pointing, the sub-array up to that point is sorted.

At every iteration of j, i is equal to j-1. Now if key(a[j]) is greater than a[i] then we can just increase j by 1 and there is no problem. The problem will occur only if this key is less than a[i], then we need to find the appropriate position of this key by shifting this one by one and each time we must compare key with the new a[i]. And this will continue until i is equal to 0.

If i is equal to 0 you must stop because there is nothing to compare. So, while i is greater than 0 (this is 1 condition) and if the key is greater than a[i], then we can just increase j (that will be taken care by the loop).

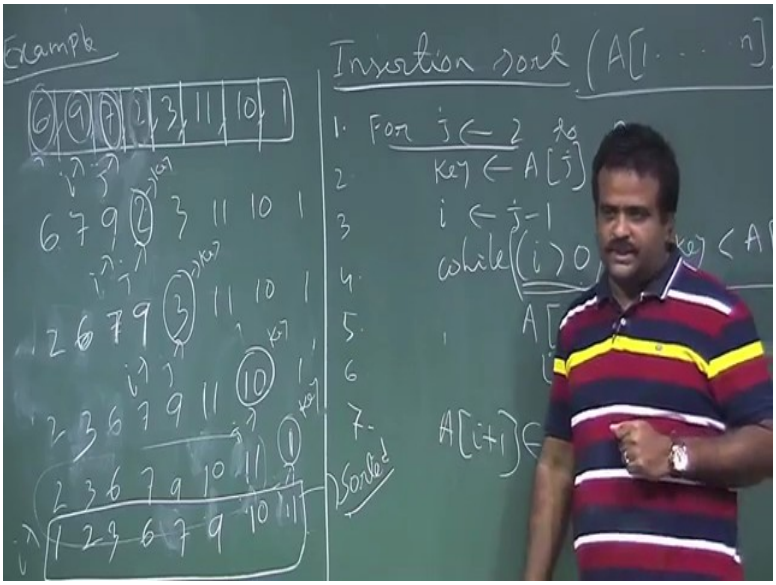
But if key is less than a[i] then we shift the value at a[i+1] (recall that j=i+1) one position to the left. So, a[i+1] will be copied to a[i] and i is decreased by 1 (i=i-1).

So, this is the step by step procedure. This is what is called pseudo code for this. It is basically a description of what we are doing.

So, this is not the plain English description, it is a very precise way of writing what we want to do in this algorithm. If you give this code to any programmer, the programmer can easily runs this code in their own computer language like C, C++, Java anything.

So, let us execute this by taking an example. Suppose we have a given input (array of numbers).

(Refer Slide Time: 11:10)



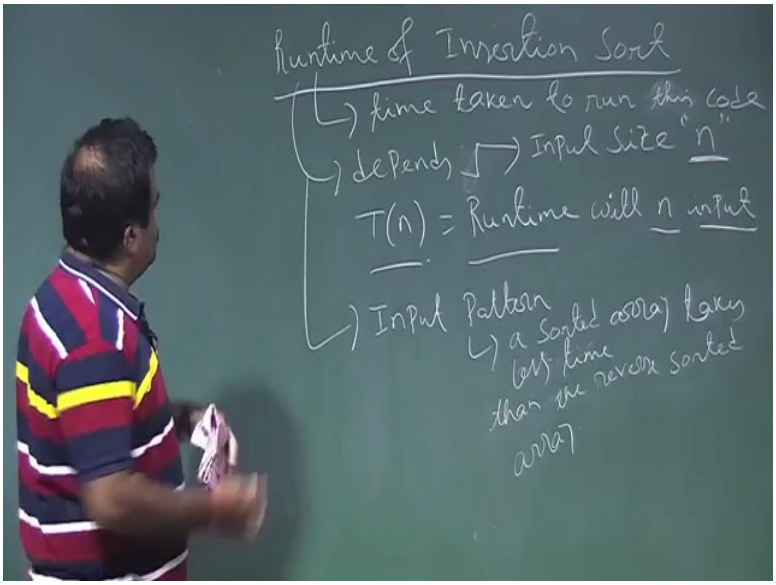
So, a = {6, 9, 7, 2, 3, 11, 10, 1} suppose this is our input (the given array) and we need to sort this array, There are 8 elements a[1], a[2], .. , a[8] and we need to sort it. Now, we are going to execute insertion sort. (For full execution example of insertion sort, please refer to the video at Time: 15:56)

We will now talk about runtime of this algorithm (how much time is it is taking by this algorithm to give the output) also called running time or time complexity.

Consider the previous array again a = {6, 9, 7, 2, 3, 11, 10, 1}. If we talk about runtime you can see that “10” has to shift just one position to left since it is smaller than “11”. Hence, we

are doing just one comparison, but for "1" we have to come a long way to the beginning. So, "1" has to be compared with everybody.

(Refer Slide Time: 20:37)



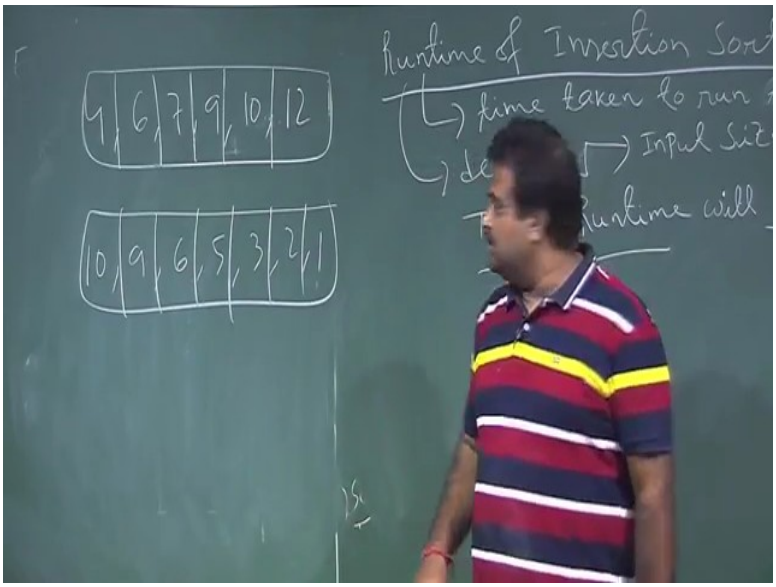
So, this means that runtime also depends on the input.

Now, Runtime means time taken to run this code. It depends on few things: first one is size of the input. See if you have to sort 10 numbers and if you have to sort 10,000 numbers, obviously 10,000 numbers will take more time than 10 numbers. Hence, it will depend on the input size that N. We want to parameterize the run time by N. So, we want to fix this N then talk about the run time.

So, we will parameterize this by N. T[N] is the run time for our code when the input size is N. Now once you fix the size of the input then let us see, what does runtime depend on. If our input is a = {6, 9, 7, 2, 3, 11, 10, 1}, Now recall that for "11" we had to do only one comparison but for "1", we had to go all the way to the beginning. So, runtime will depend on the input pattern as well.

So, if the input is already sorted then it will take less time, and if the input is reverse sorted then it will take more time (reverse sorted means everybody has to come to the beginning in each step of the for-loop). So, it will depend on the input.

(Refer Slide Time: 23:50)

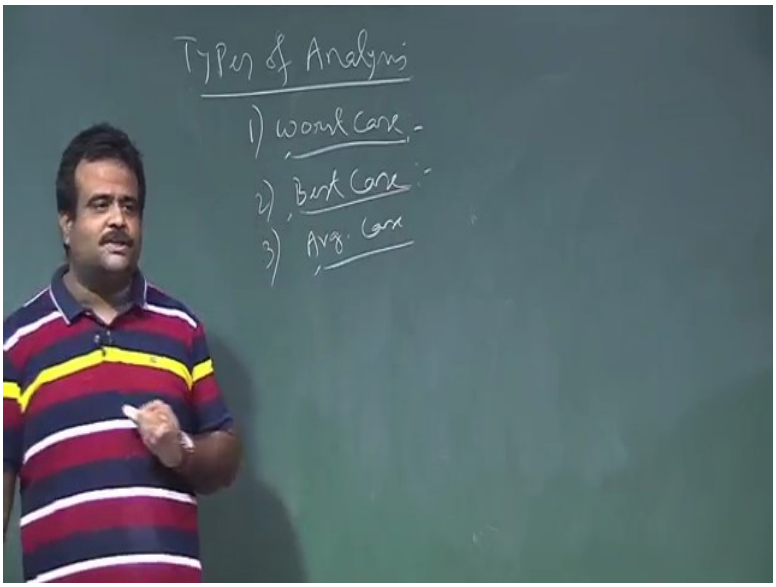


So, if our input is a = {4, 6, 7, 9, 10, 12} which is already sorted array, then if you run the insertion sort we just compare each element only once and stop.

This will take less time than if the input was reverse sorted. Say if we have input a = {9, 6, 5, 3, 2, 1} then everybody has to come to the beginning and it will take more. So, the runtime will depend on the input pattern, in the sense that an already sorted array is easier to sort and will take less time than if the array is reverse sorted it will take more time.

Based on this, we will talk about 3 types of analysis, worst case, best case and average case analysis. For insertion sort it will depend on the pattern of the input.

(Refer Slide Time: 26:05)



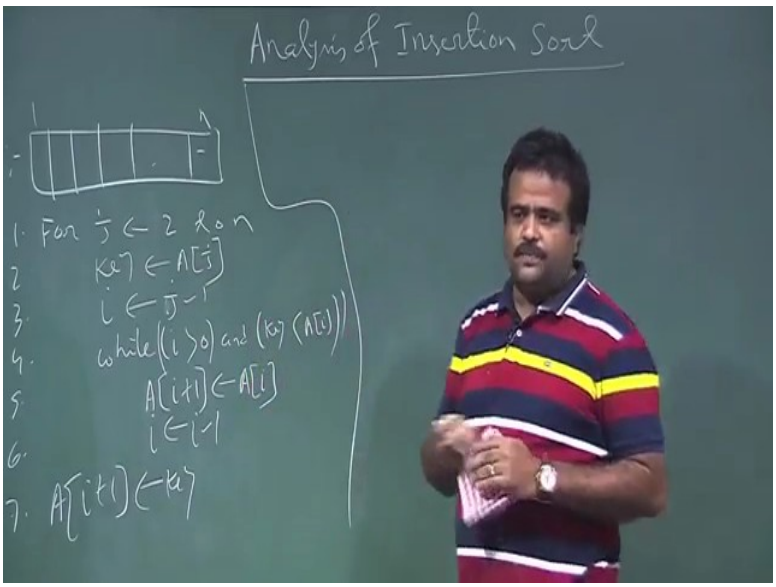
So, first one is worst case. We will talk in more details in the next class. Worst case for insertion sort is if the input is reverse sorted (then it takes maximum time) and the best cases is if the input for insertion sort is already sorted, (then we have to spend less time to sort it) and there is another case which is called average case. Average case means how much time does it take on an average (it should be some expected value). So, for expectation may be we need to take some distribution of the input pattern and then we talk about the expectation. So, this we will discuss in the next class.

Thank you.

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

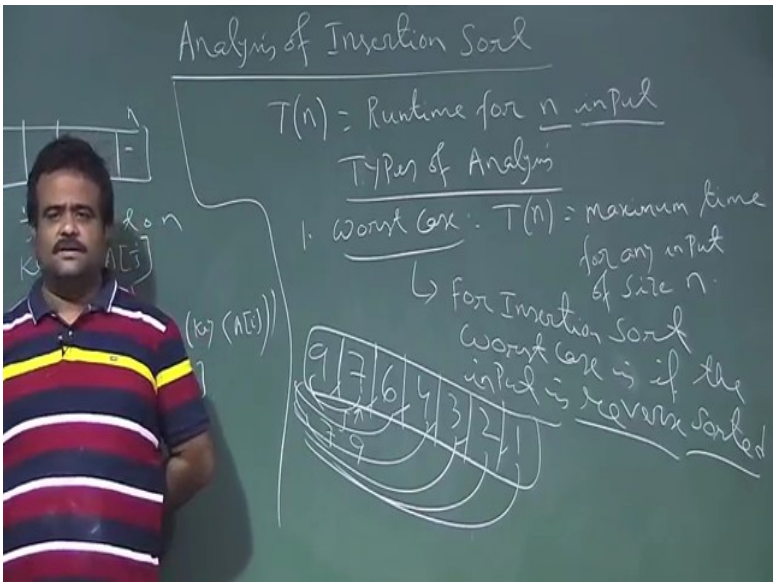
Lecture – 02
Analysis Of Insertion Sort

(Refer Slide Time: 00:30)



So, now we want to analyze the insertion sort. So, basically we want to analyze the time complexity of insertion sort. In the last class we have seen that the run time will depend on the size of the input (i.e. it will depend on N).

(Refer Slide Time: 01:46)



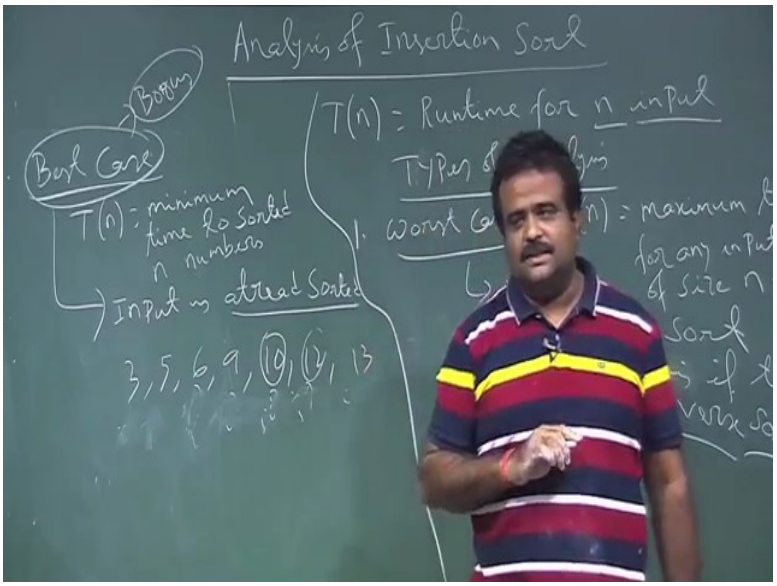
So, that is why we parameterize the time complexity by N. We have also seen that the runtime will depend on the pattern of the inputs. So, if we have an already sorted array then it will take less time, than if we have a reverse sorted array it will take more time. We will be doing three different types of analysis.

First one is worst case; that means, T[N] is the maximum time that our code will take to run for any input of size N. So, what is the maximum time our code is taking? Recall that for insertion sort worst case is if the input is reverse sorted.

So, that means, if we have numbers say a={ 9, 7, 6, 4, 3, 2, 1}, suppose this is the input and we want to run insertion sort on this input. Now since it is reverse sorted, everybody has to come forward. There is no other input which can give the more runtime than this. So, this is sort of guarantee that reverse sorted input will take maximum time.

So, worst case means that this is the maximum time my algo is taking which means, nobody can come with some input where my code is performing worse than what I am claiming. That is the reason, worst case is the most usual analysis one must do for any algorithm.

(Refer Slide Time: 05:28)



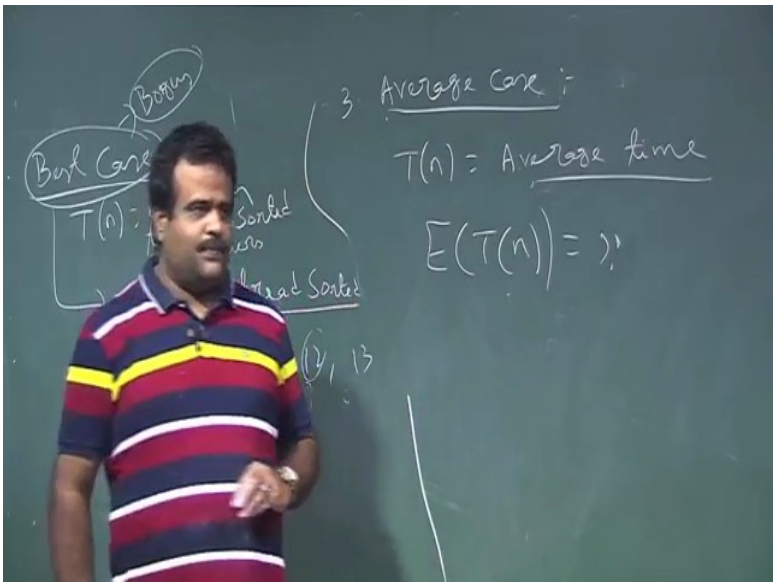
Best case means, the T[N] is minimum time taken to sort (we are talking about, minimum time our algo is taking for input of size N).

So, this is the minimum time. So, for insertion sort the best case occurs if the input is already sorted. So, if the input is already sorted, that means if we have a input like this say a={3, 5, 6, 9, 10, 12, 13}. For this input if we run the insertion sort, we are doing only one comparison for each element of the array. So, this is the best case scenario.

Now, whether you want to do the best case or worst case, that is very crucial point. So, best case means there is no upper_bound on the run-time. I mean, suppose Microsoft opens competition and ask this group to build sorting algorithm ask this group to build a sorting algorithm. Now you build a sorting algorithm and you submit it to the Microsoft and Microsoft gives your code to other group, and ask to find out an input for which the algorithm would perform badly. So the worst case time complexity will give you a sort of guarantee or upper bound on the running-time of the algorithm.

So, we will not prefer to do this analysis, because this is valid only for some certain type of input whereas the worst case analysis is the usual analysis we will do because it gives you a sort of guarantee.

(Refer Slide Time: 09:51)

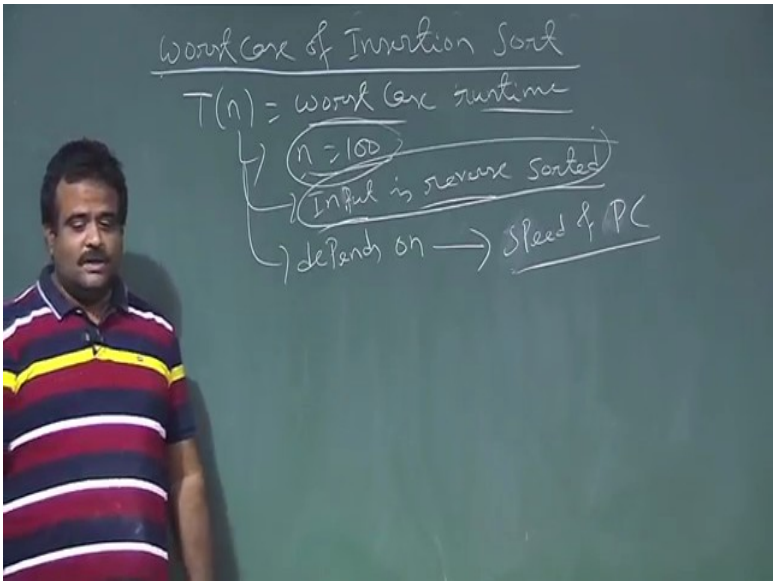


Now there is a third kind of analysis which is average case analysis.

Average case means on an average how much time it is taking to sort. T[N] is the average runtime that means for average to be good we need to take the expectation. So, maybe we need to deal with the expected value of T[N]. So, we need to assume some distribution on the input. Maybe we will do some type of analysis in our course, but this is basically average case expected runtime.

So, we know that worst case is the usual analysis one must go for.

(Refer Slide Time: 11:36)



Now suppose we want to do the worst case analysis of insertion sort. T[N] is the worst case runtime.

Suppose we are talking about we have say 100-element input (we are sorting 100-element input) and since it is worst case for insertion sort the input is reverse sorted.

Now, tell me if we fix these two situations will the runtime depend on any other thing or any other factor that might be interfering in the runtime. So, that is a crucial question. Suppose you are running this on a your band new laptop core 2 processor and you are running this same code (with a reverse sorted input), on an old computer your grandfather is having. So your laptop will take faster time.

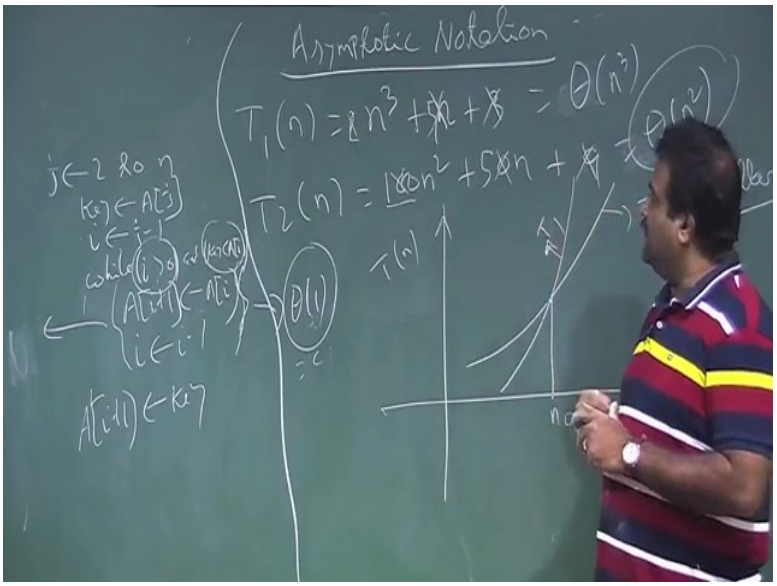
So, your laptop can give the output in a second, but our grandfather computer can take an hour to give the output. So, it will depend on the speed of the computer where we are running the code. Now it is depending on the computer where you are running the code then how will we judge which one is good?

Now, we have to run this code in the same computer not only that, we have to run this in a same computer in a same situation at the same time which is not possible.

So, it is very difficult to run to piece of code in a same environment in a same situation. So, that is a headache I mean it is not possible every time if our computer is multi user since every time some processor is running something. So, that is a botheration. So, then how can we judge the two pieces of code (which is faster than which) that is very difficult to judge in this way. So, to avoid this botheration what we need to bring in asymptotic notation.

This is a very big idea I means it is a great idea where we can get rid off from the computer or machine dependency .

(Refer Slide Time: 17:11).



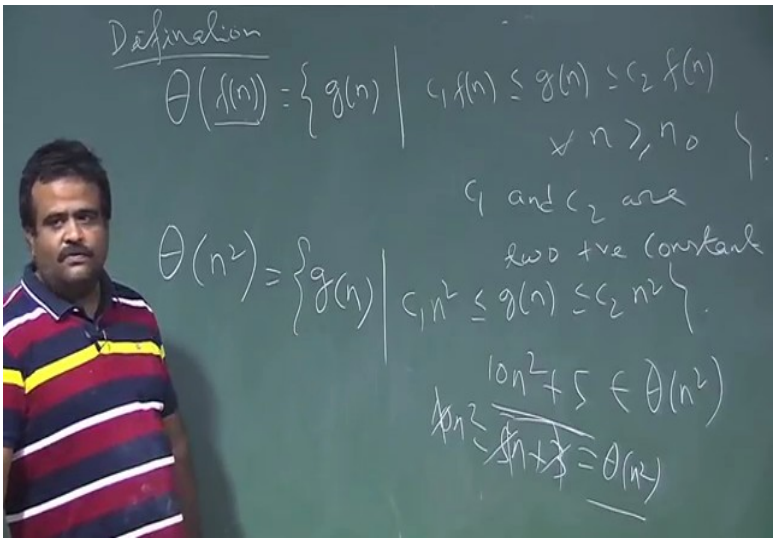
For insertion sort we have a loop nested inside another loop. So, suppose we have a sorting algorithm with T1[N]=2n^3+5n+3 as the runtime for our machine and T2[N]=100n^2+50n as the runtime for another machine. Now can you tell me which code is better?

If you draw the curves corresponding to these equations in terms of n, you will see that as we tend to larger N then T2[N] looks better, but if you take smaller N then T1[N] looks better, but if we make n large then it is it is clear that T2 is better . So, we want to see the growth. So, that is why we need to bring this asymptotic notation.

So, in asymptotic notation we ignore all the lower order termand then we ignore the leading coefficient and this will give us what is called big theta. As N tends to infinity we will get rid from this machine dependent term.

So big-theta is this asymptotic notation. So, now we have a method to compare two pieces of code without depending on the machine. So, that is why this is the best way, this is the very useful way to compare two code in an asymptotic sense. So, now, we formally define the asymptotic notation we have 3 asymptotic notations, one is “big-θ” another one is “big-O” another one is “big-Ω”.

(Refer Slide Time: 23:58)



So, this is the definition. So, first we define the big-theta. So, big-theta of f(N), it is basically set of all function g(N) such that it is bounded by c(f(N))less than equal to g(N) less than equal to c2(f(N)) and for all N after subequal we do not care about the low N because we want to see the asymptotic growth of this runtime.

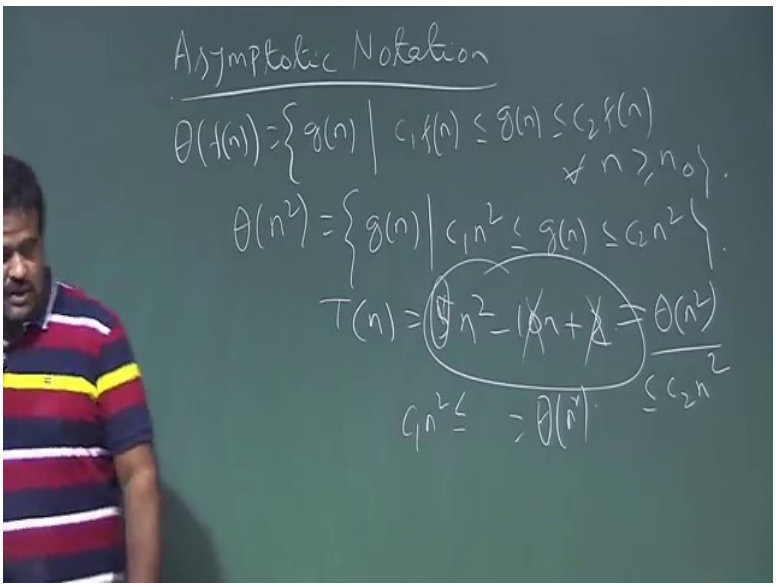
Similarly consider f(N)=10^2-5n+3, this also belongs to θ(N^2). So, as you said in engineering sense what we do we just ignore all the lower order terms, we ignore the leading coefficients. So, this is the asymptotic notation of big theta of n square. So, in the next class we will talk about another asymptotic notation what is Big O and Big Omega.

Thank you.

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

Lecture – 03
Asymptotic Notation

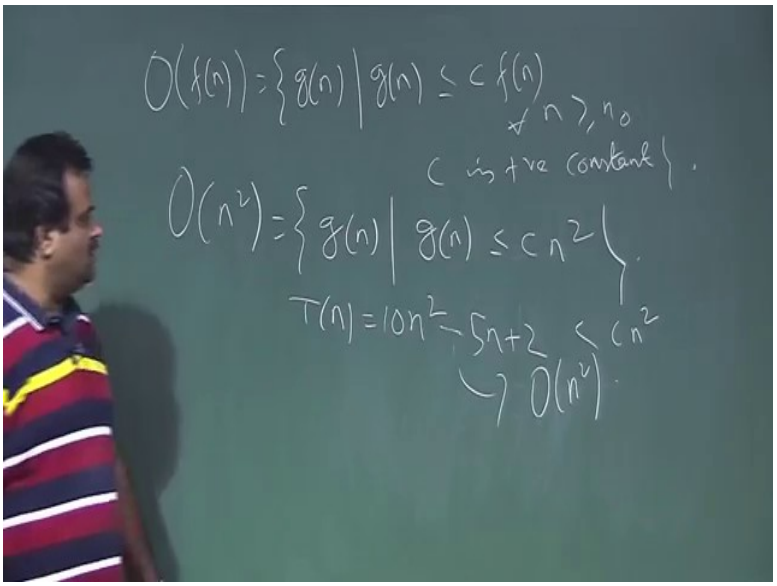
(Refer Slide Time: 00:26)



So, we are talking about asymptotic notation. So, we have Θ notation. We define the Θ(f(n)) as basically a set of all function g(n) such that it is bounded on both sides by f(n) for all n greater than some n0. So, this is the mathematical definition of it. So, for example, if we take the Θ(n^2), this is basically set of all function which should be bounded by n^2 by both side. So, this is c of n square for all n greater than n0 where c1 c2 are 2 constants.

So, for example if our time complexity is T[N] = 5n^2-10n+2. So, for this we can see that we can choose some suitable constant c1 and c2 such that this is bounded by both c1*n^2 and c2*n^2; so basically it belongs to big-theta of n^2 (but we will just write this as Θ(n^2). So, we just ignore the lower order term, because higher order term is n^2 and also ignore the leading coefficients.

(Refer Slide Time: 02:39)

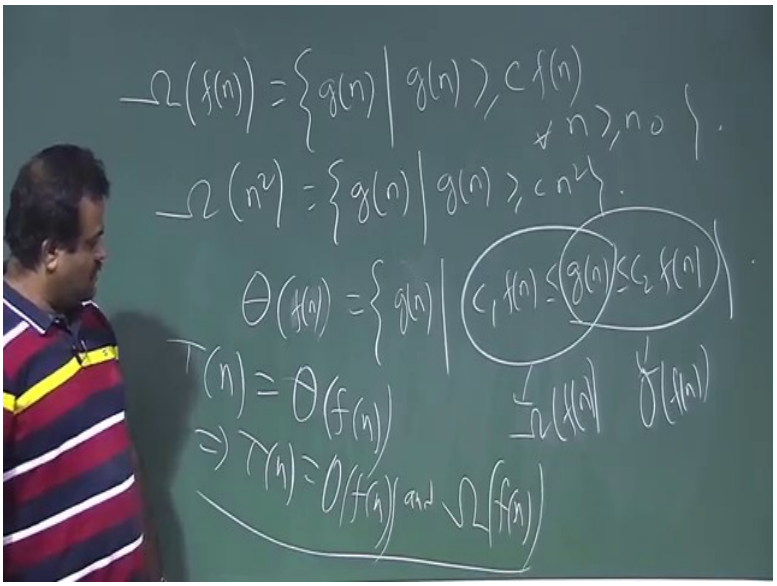


Now, we talk about big O. So, big O is basically bounded above. Big O(n) is basically set of all function g(n) such that g(n) will be less than some c*f(n), and this is true for all n greater than equal to n0 and my c is a positive constant.

So, basically it is a upper bound. So, if we have a such a upper bound then say for example, big O(n^2) is basically set of all functions such that it is bounded above by n square. So, if we can choose a constant c such that g(n) <= c*n^2 then g(n) belongs to big-O(n^2).

Now consider 10*n^2+5n+2. If we choose c to be greater than 10 then g(n) can be shown to be always less than c*n^2 so that means, this is basically big O(n^2). So, this is the big O notation this is the upper bound.

(Refer Slide Time: 04:26)



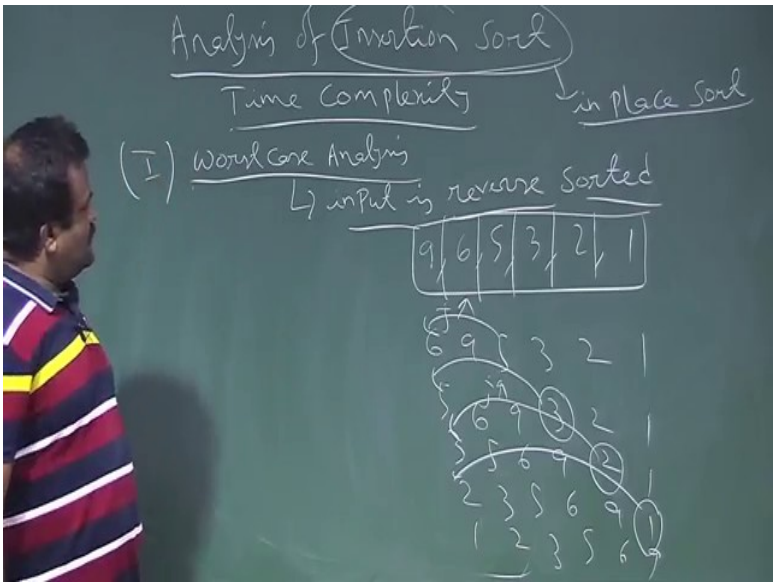
So, big-Omega of f(n) which is basically the lower bound, which is basically set of all function g(n) such that g(n) is greater than c*f(n), and this is true for all n greater than equal to n0 (this is the mathematical definition where c is a constant). So, big-Omega of n^2 for example, is set of all function which is bounded below by n^2.

So, big theta of n^2 or any other function is basically set of all function such that it is bounded both side. So, if T[N] is a big-theta of say some f (n) then it implies that T[N] is both big O of f(n), and big-Omega of f(n).

So, big theta is basically both side bounded and big O is basically upper bound and big-Omega 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 then that has some sense. So, usually we go for the upper bound I mean the big O.

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

(Refer Slide Time: 07:34)



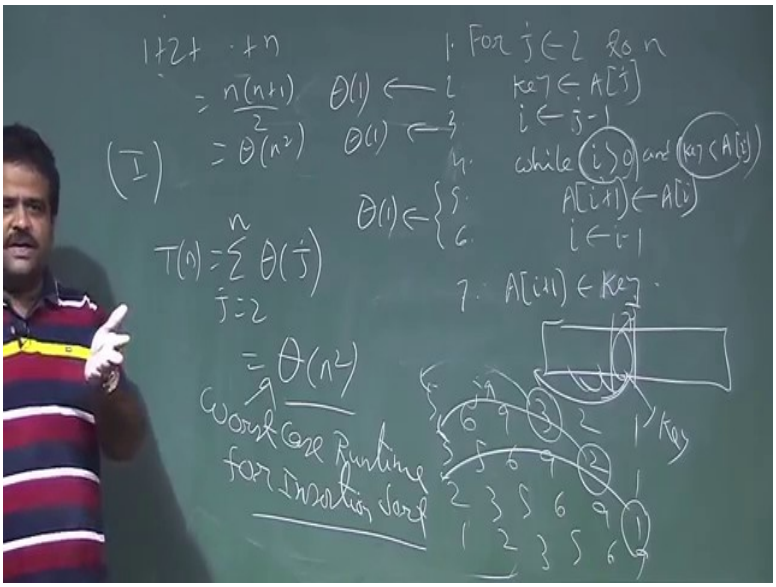
So, now, with the help of this asymptotic notation we will analyze the insertion sort with all the cases using this asymptotic notation.

So, there's also the space complexity analysis (which refers to how much we are using).Since in insertion sort we are using the given array only so it is a in-place sort. So, insertion sort is an in-place sort. We are using a small constant space for swapping too but that is just one or two variables. So, that is of the order of $\Theta(1)$ size.

Now we talk about time complexity. We know that we have to do 3 types of analysis. So, first one is worst case analysis for insertion sort.

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, 8, 7, 6, 5, 3, 2, 1 and you run insertion sort on this input. Then for every word we have to come to the beginning so that is the worst case. So, now we want to analysis this.

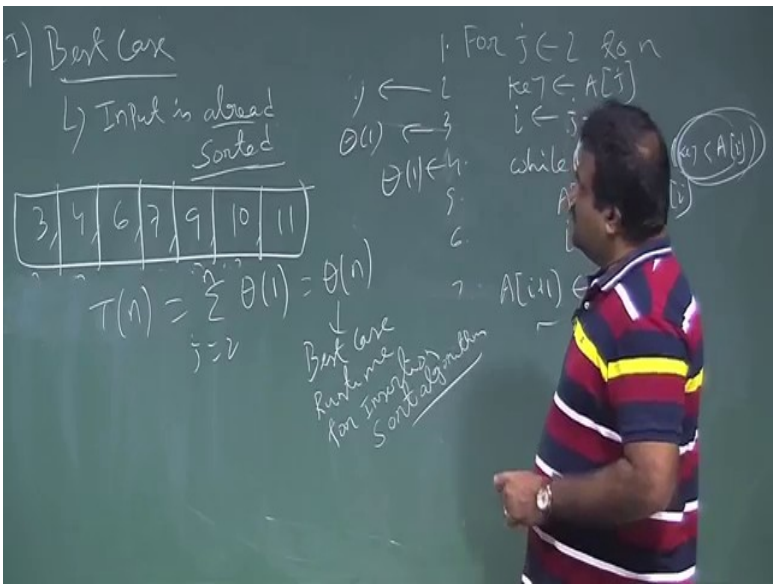
(Refer Slide Time: 11:01)



If we recall the code for the insertion sort, we can find the time complexity of it. So, we have a for loop and another for loop nested inside this and all operation in each iteration is taking Θ(1) time.

So in each iteration, it is comparing and doing something. So, if you analyse the series formed here, so for j=2 we do 1 comparison, for j=3 we do 2 comparisons and so on. Thus total no. of operations becomes n*(n+1)/2. So, this is theta of n square. This is the worst case runtime for insertion sort; in asymptotic sense because we do not want the exact one, because exact one is a headache because it will be machine dependent.

(Refer Slide Time: 15:30)

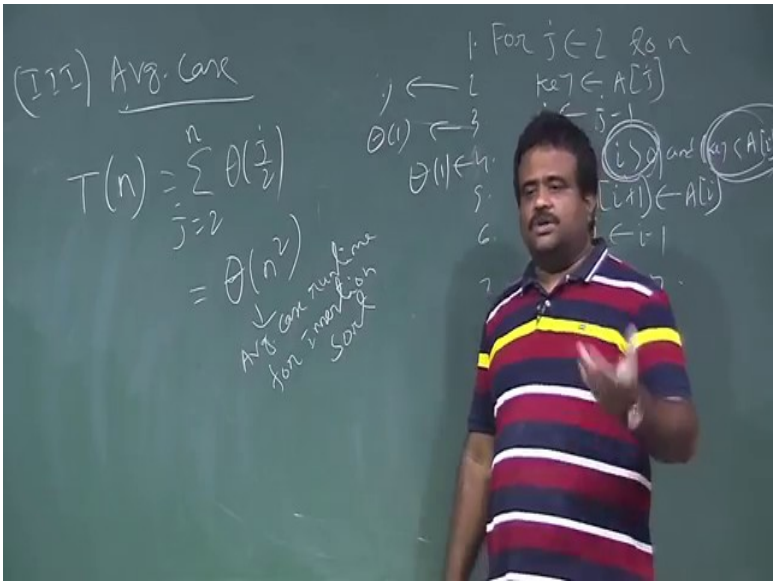


Now coming to the best-case analysis.

So, best case analysis for insertion sort. Say if your input is 3 4 6 7 9 10 11 suppose this is our input. Now if we want to run our insertion sort on this input then we can see that for j=2 we do only one comparison, for j=3 we again do only 1 comparison. Thus by analysis, a series follows and total no. of operations is of the order of O(n) and each operation is of the order of Θ(1).

So, the best case runtime for insertion sort is linear, but it only occurs if the input is sorted. So, now we talk about another analysis which is called average case analysis.

(Refer Slide Time: 18:52)

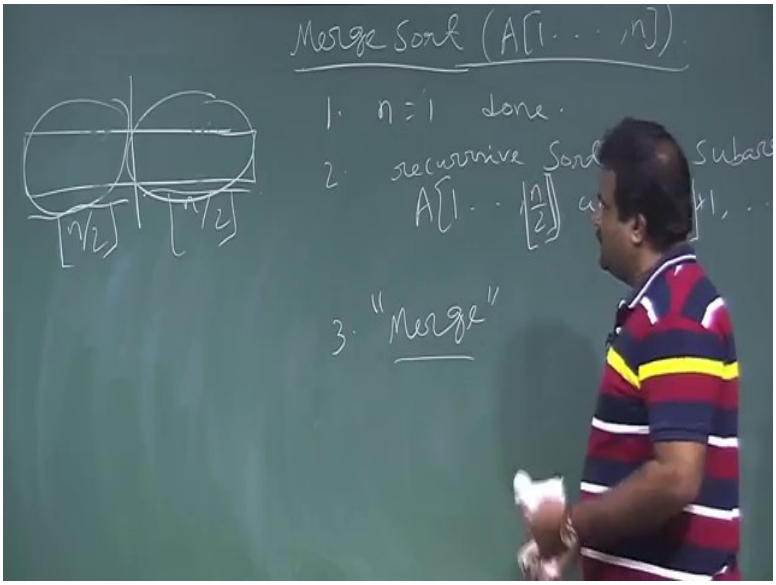


So, now, we talk about the third one which is the average case analysis for the insertion sort. So, average case when you do the average case analysis for any algorithm then we must have a something called some randomized algorithm; that means, some randomization will be there, (will talk about this in more detail).

So, in the average case we can say for each value of j (or for each iteration of the outer loop) we only need to come half way. But this is the intuitive analysis only. It is not exact average case analysis 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 if we analyse the series, the sum becomes n*(n/2) and this is again Θ(n^2).

So, this is basically average case runtime for insertion sort. So, now, we talk about a algorithm sorting algorithm where worst case is better than n square. So, that is called merge sort.

(Refer Slide Time: 21:29)



So, we have given an array of size N. So, the idea is that it 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 subproblems of lesser size, and then we solve the sub problem recursively (this is the conquer step) and then we combine the solution of the sub problems to get a solution of the whole problems (this is the combine step), we will talk about in more details for divide and conquer technique, but this is the one example of such a technique. So, basically we have an array of size n. We are just dividing the array into 2 parts of size n/2 and n/2 provided n is even or if n is not even we must put a lower ceiling or upper ceiling anyway you can be sloppy a little bit here.

So, we have an array and now we divide it into two parts/subarrays and sort these subarrays and then we call a combine step that is called **merge subroutine** i.e. we call a function **merge** which takes these two sorted subarrays and gives us the full sorted array.

So, we will talk about the time complexity and the space complexity of this algorithm in the next class.

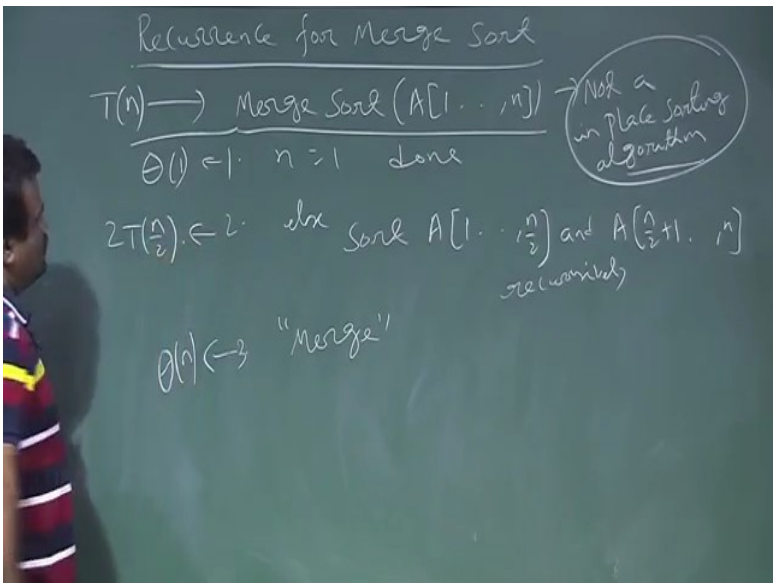
(For the exact code and a working example, please refer to the videos)

Thank you.

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

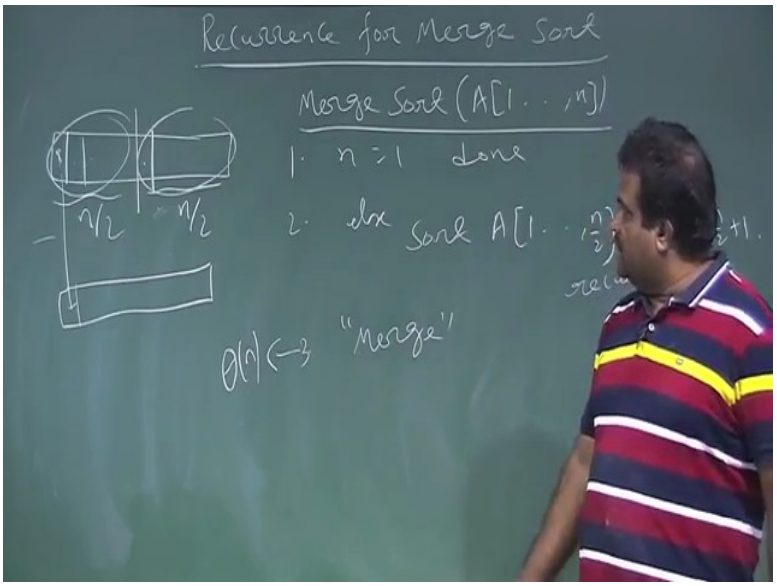
Lecture – 04
Recurrence For Merge Sort

(Refer Slide Time: 00:25)



So we talk about the time complexity of the merge sort or the analysis of the merge sort. So, just to recap, in a merge sort we have a array of size n and then if n is 1 we return the array itself otherwise else we recursively sort this subarrays 1 to n/2 and (n/2 + 1) to n recursively by calling the same function merge sort and then we call the merge.

(Refer Slide Time: 01:10)



So, basically we have this array now we partition into 2 parts this is n/2. I mean if n is even otherwise, it will be lower subarray and upper subarray, we sort this subarray recursively and once we got the two sorted subarrays then we call merge subroutine to get a sorted array. So, for merge what we are doing, we are taking the minimum. So, minimum will be here in this sub array minimum will be here. So, for this merge we need to take help of extra array. So, this is not in-place sorting algorithm. For the merge subroutine we need to take a help of a extra array.

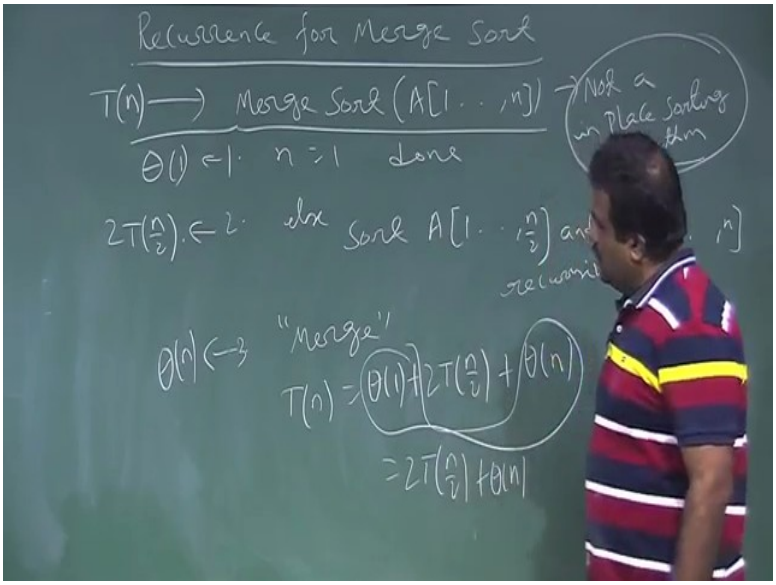
If we want this merge to be in linear time. So, why linear we will come to that later. So, we compare these two and whichever is the minimum we will put it here and then we point to the next one. So time complexity for this is order of n, this is linear time because we are just reading this every time we are returning 1 element. So, there are n element total n/2. So, basically we are spending linear time for that.

So, and, but this for this merge we need to take help of a extra array to store this element otherwise we cannot store here in a linear time not possible to store this in a linear time the merge can be done in the same array that is not possible. So, this is not in-place sorting algorithm.

Now, now we want to know the time complexity for this. Suppose T[n] is the time to sort n elements now how much time we are spending here? Here we are spending theta(1) time. Now this is theta(n). So, now how much time we are spending here? So, we have 2 call of

same size n/2 of the same merge. So, it will be basically 2 *T(n/2). So, the T[n] will be sum of this.

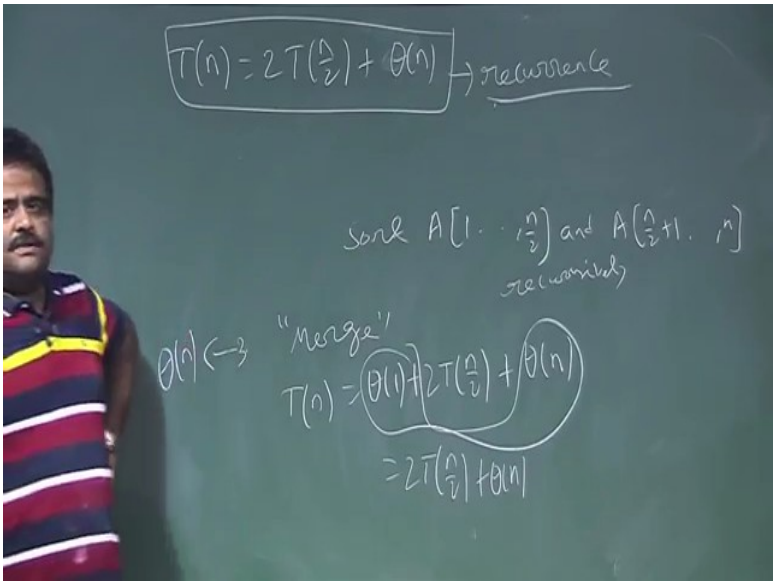
(Refer Slide Time: 03:52)



So, the T[n] will be basically “theta(1) + 2*T(n/2) + theta(n)”.

So, this theta of n and all will combine. So, this is basically “2*T(n/2) + theta(n)”. So, this is basically the time complexity. This function is called recurrence function.

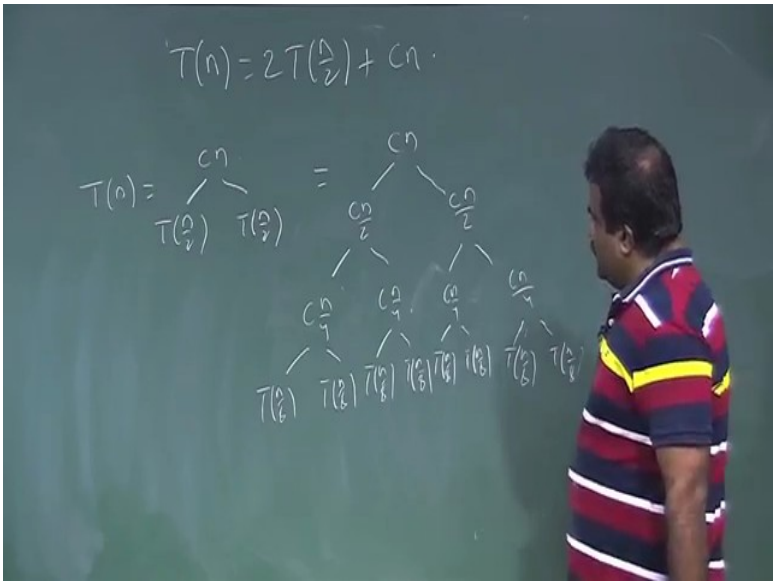
(Refer Slide Time: 04:34)



So, this is called recurrence of this algorithm. So, we got this recurrence. So, basically we got T[n] is equal to "2*T(n/2) + theta(n)" or some theta(n).

So, this is what is called recurrence or recurrence relation. Now the question is how to solve this recurrence. We need to get a solution. We are not happy with this type of expression. We want to know whether merge sort is of the order of n*n algorithm or n*log(n) algorithm n^3 or linear time. So, to solve this recurrence we can use what is called recursive tree method. So, basically you are going to solve this like theta n we can just put c of n some constant n.

(Refer Slide Time: 05:49)



So, basically, we have a array of size n we divide the array into 2 subarrays and then we call the merge and then we recursively sort these 2 subarrays then we call merge to get the solution of the whole array.

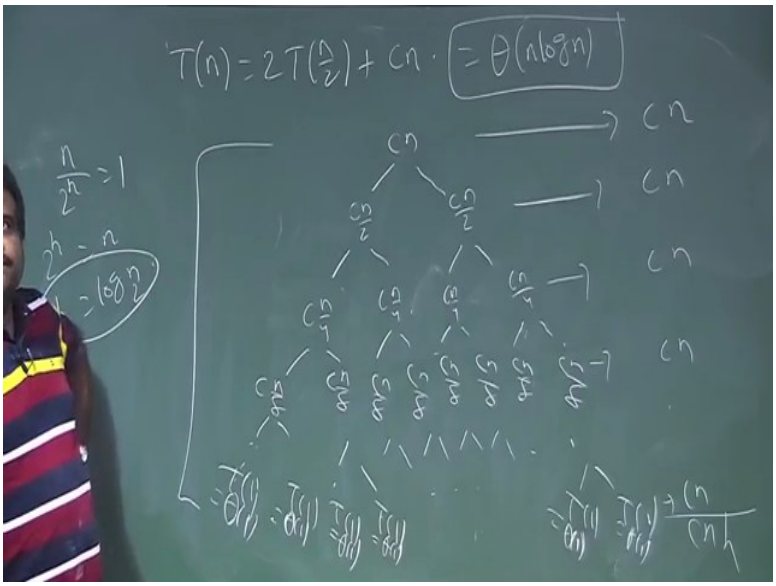
So, this is our T[n]. So, T[n] is basically can be written as c*n, T (n/2), T (n/2). So, basically we have a problem of size n this is what is called divide and conquer technique. So, this is a problem of size n we divide the problem into 2 sub problems and then we recursively solve this sub problems this sub problems and this will again take T(n/2), T(n/2) and then once we have the solution of these 2 sub problems we merge this with a cost of c*n that is called merge in the merge sort.

Now this is basically a sub problem, but this is again a problem of size n/2. Again we can further divide it into the sub problems, sub sub problems. So, this will be again can be c*n/2,

T (n/4), T (n/4). Since this is a subproblems of size n/2 so, again we further divide this and then we call the merge sort on these subproblems. So, it will divide into sub sub array. So, like this we continue until we reach to the size of one array, then we merge and come back like this.

So, again we further divide this into c*n/4, T(n/8), T(n/8) this is the merge cost c*n/4, T (n/8), T(n/8) this is c*n/4, T(n/8),T(n/8) this is basically c*n/4, T (n/8), T(n/8) . So, this way we will continue and then our time complexity is sum of all this. So, this way we will continue and we stop when it will reach to the point when n will be 1 because if it is 1 then we cannot further divide into sub problems.

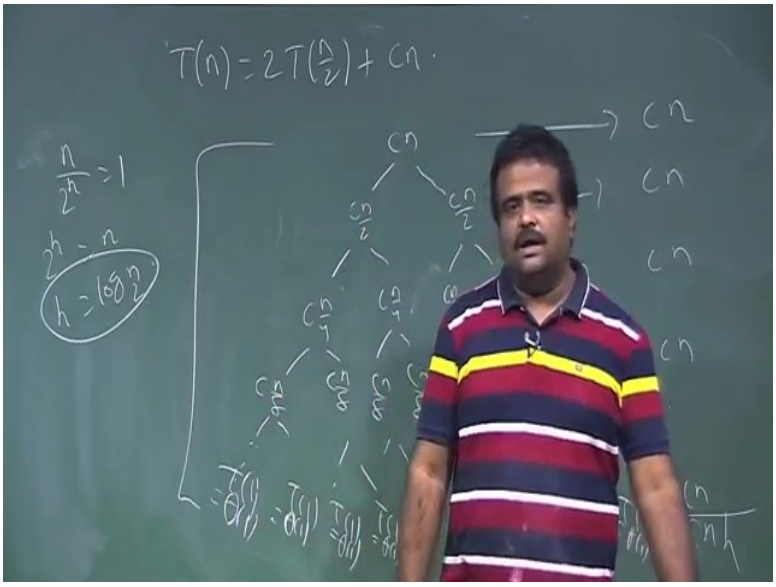
(Refer Slide Time: 08:40)



So, now our time complexity is the sum of these nodes basically. So, now, the question is how to get the sum of these nodes. So, T[n] is basically sum of all this. So, how to get the sum? We can get the sum level wise and then sum each of them. So, if you do that you will see that sum at all levels is c*n.

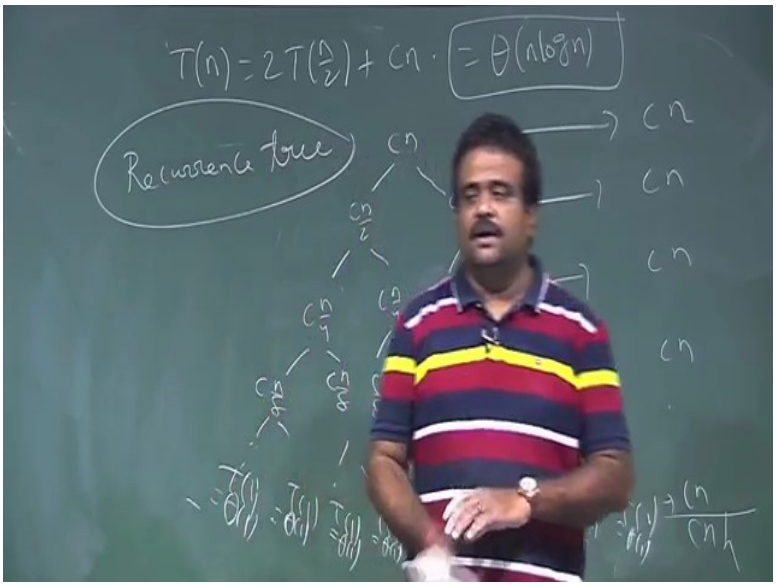
Now, what is the height of the tree? H is the height of the tree. now what is this height of the tree. Now how to get height of the tree? So if you observe properly, you will see that this is a binary tree. So, from here we can say 2 to the power h is n. So, h is log n base 2. So, height is height is log n.

(Refer Slide Time: 11:01)



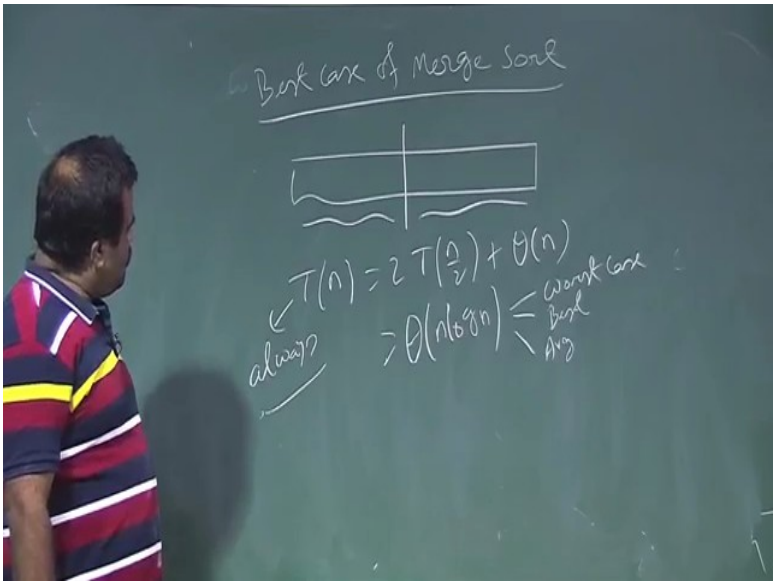
So, this is basically c*n*log n. So, the solution of this recursion is n*log(n). So, this is the time complexity for merge sort. So, this is what is called recursive tree. So, this method is called recursive tree method.

(Refer Slide Time: 11:47)



So, recurrence tree or recursive tree; so, this is one way we can try to get the solution. There are some other methods to solve the recurrence. Substitution method is one such method and follows an inductive approach. So, we have a full proof method which is called substitution method to solve the recurrence where we will see by the method of induction we can prove that our solution is correct. We are not doing the proof of the induction method in this course. So, now we will see, what is the worst case for merge sort what is the best case and what is the average case.

(Refer Slide Time: 13:16)



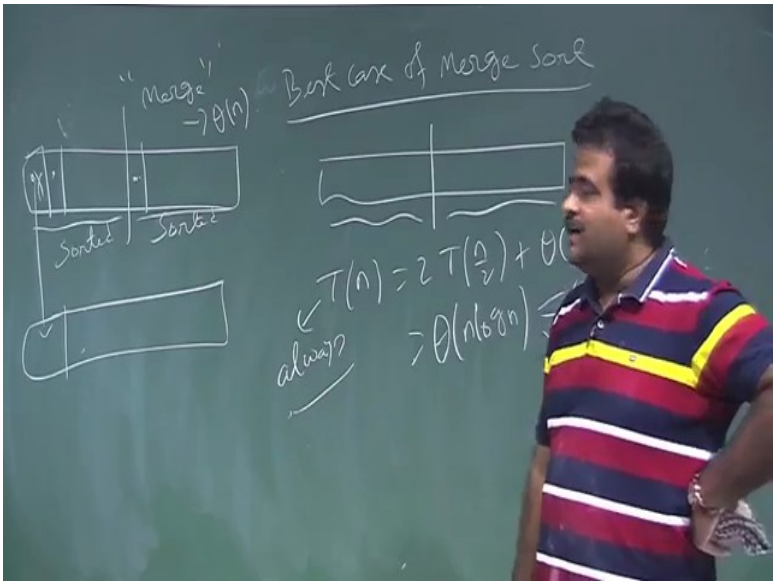
So, you want to do the type of analysis of merge sort. So, like in insertion sort we will see that if the input is sorted it is best case, but is it same here? So, if the input is sorted in the merge sort, will it make any difference? If the input are equal elements, will it make any difference? In our last approach, whatever input may be we are not caring we are just going to the middle we are dividing the array into 2 subarrays we sort them, then we merge them.

So, we really do not bother about the input we are just going to the middle of the array. We are dividing it, we are sorting this part, we are sorting that part, then we are merging and that is it. So, it is always recurrence will be like this for any input 2*T(n/2) + theta(1) always for all type of analysis. So, the solution is always this; this is same for all the cases because all the cases are same because we really does not bother about the input pattern we just divided the array into 2 sub arrays then we sort this sub array recursively we sort this sub array

recursively then we call the merge. So, thus this will be the recurrence in any input for any input.

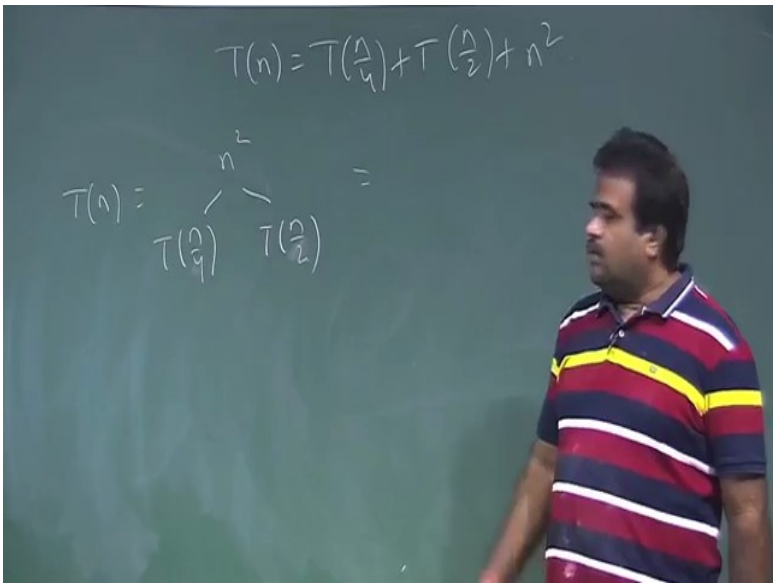
So, that is the case now we talk about so, but this is a n*log(n) time algorithm whereas, worst case where as the insertion sort is order of n*n algorithm so, but whichever is the better insertion sort or the merge sort? Insertion that has an extra advantage in the sense that it is an in place sort. We do not need the extra array to sort it, but merge sort is not an in place sorting algorithm.

(Refer Slide Time: 15:30)



Now, we will need to call merge. So, merge is merge subroutine. So, then we need to take help of extra array. So we are taking the list elements and compare the least 2 element whichever is the least 2 among this we are outputting there. Then we are comparing the next least and so on. So, this is the merge subroutine. So, now we want to do merge in linear time. So, for that we take help of a extra auxiliary array of size n otherwise we cannot make it in linear time. So, now we will take another example of recurrence where we will use the recursive tree method to get the solution.

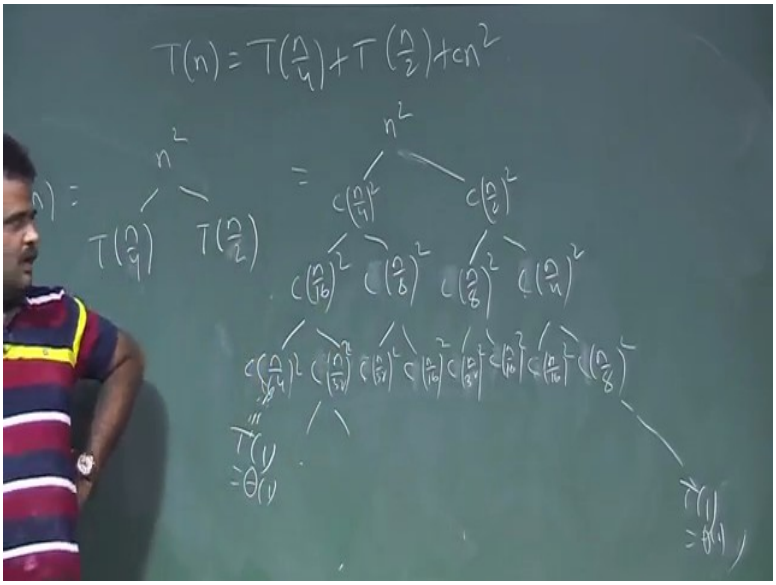
(Refer Slide Time: 16:59)



So, let us have another recurrence like suppose we have a recurrence like this T n is equal to T(n/4) + T(n/2) + n*n. So, we have a problem of size n we divide the problems into 2 sub problems and here they are not equal size one sub problems size is n by 4 another sun problems size is n by 2 then once you have the solution of this 2 sub problems we combine this 2 sub problems with a cost of order of n square and this is basically combine step or merge step.

Now the question is how to solve this recurrence. So, we will again use the recursive tree method to solve this. So, we can write this as this T (n by 2), T (n by 4) like this then again this is a problems of size n by 2. So, we can again divide into sub problems.

(Refer Slide Time: 18:38)



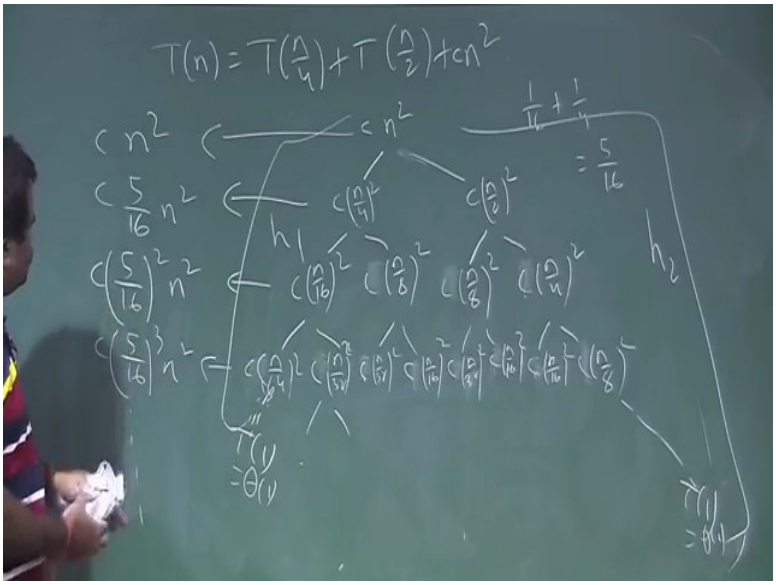
So, then this is basically n*n and this will be basically c. So, now, our size is n/4.

So, that is basically (n/4)*(n/4). You can just write in this way then it will divide into this T(n/16) and then this is T(n/8). So, basically here further we are dividing into subproblems; sub problems with the same recurrence. So, then again we divide into sub sub problems. So, this is of size this is of size n by 16. So, if we divide it into sub sub problems it will be T of n by 64 and this is T of n by 32 and this will be again T of n by 32 and this will be again T of n by 16 and this will be again T of n by 16 sorry, T of n by 32 and this is T of n by 16 and this is T of n by 16 and T of n by 8 and this will be c of that square c of this square c of this square c of this square and we will continue. So, which branch will end first. So, this is going rapidly. So, this will end. So, each of this branch will end with T 1.

So, each of this branch will end T 1. So, this will end first than this one. So, this will be going little longer. So, everybody will end at the at the point T 1 everybody will end at the point T 1 so, but this branch will end first because its going rapidly to this and this is will be c of that square c of that square c of that square c of that square c of that square c of that square c of that square like this.

So, every branch will be ending with T 1, T 1 means theta of 1 like this theta T(1) = theta(1). So, every branch is will be ending at theta(1). So, our time complexity is basically sum of the all nodes, so, now, how to get the sum. So, we will get the sum level wise.

(Refer Slide Time: 22:01)

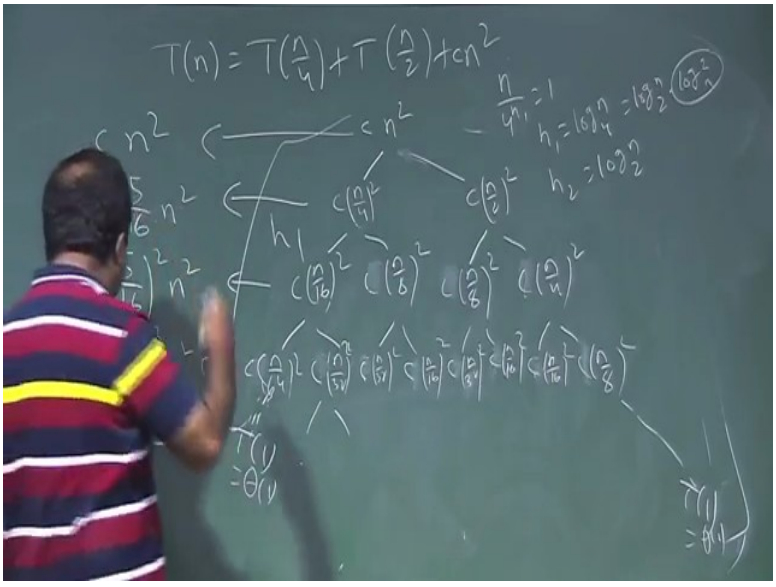


This is n square. So, what is this level sum. So, this is basically what 1 by 16 plus 1 by 4. So, this is 16. So, 5 by 16. So, this is basically c of that. So, this is c c of 5 by 16 n square ok

Now, if you just calculate this will be getting c of 5 by 16 square n square I mean if we just pick, we can verify this then this will be c of 5 by 16 n cube like this. So, this way it will be coming. So, this way it will be coming now. So, again here we are using our intuition what intuition that in case it like this. So, what is the at the i th level we are thinking that it will be 5 by 16 to the power i, now what is that prove that proof is not we are doing here. So, so either we have to proof that by method of induction or something else then we can say this is a probable method so, but we are not doing we are just using the intuition we are no presence to go up to this level after this, but we are using our intuition it is going like this. So, it will go like this.

So, this is the way it is going. So, now, if the height of this is the h1 and the height of the bigger tree is h2 then we can say this sum then we can say. So, h 1 is the earlier ending the tree. So, up to h1 it is bounding like this. So, up to h1 it is a complete binary tree and up to h 2 it is the complete. So, basically it is bounded by sum of this up to h1 left side and bounded by sum of that up to h2 now we want to take the sum of this term. So, what is h 1. So, we want to get h1. So, this is 4 this is 4 square this is like this, so, 4 16, so, 16 into 4, so, 4 4 plus 4 cube.

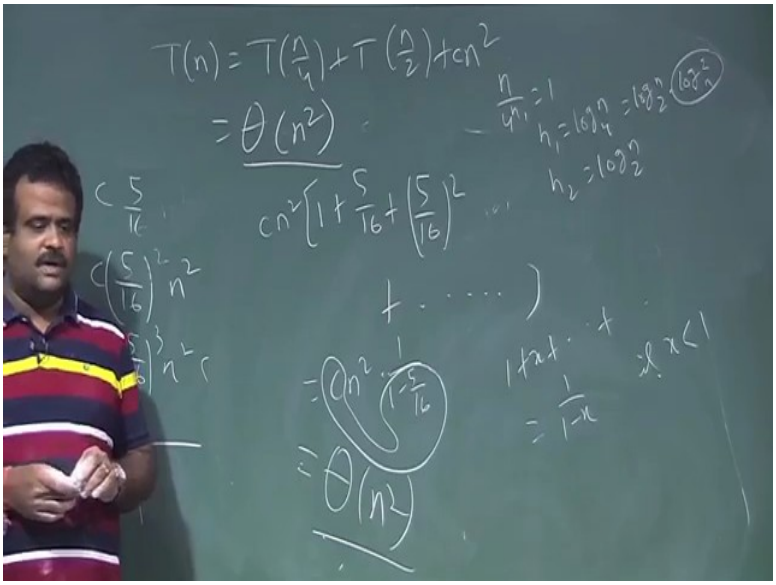
(Refer Slide Time: 24:57)



So, basically (n/)4 to the power h1 is equal to 1. So, h 1 is equal to basically log(n) to base 4. So, h2 is basically log (n) to base 2.

So, height is of the order of log(n). So, now, we need to take the sum of this and that will be the solution, so, how to get the sum of this? So, it is basically c c n square into 1 plus 5 by 16 plus 5 by 16 square plus so on.

(Refer Slide Time: 25:43)



Now, this is a finite times, but if we want to make it infinite this would be less than and now this is basically c of n square this is basically this series power series 1 plus x plus x square like this.

Now, this is basically 1 by 1 minus x if x is less than 1. So, this is basically 1 by 1 minus 5 by 16. So, this is again a constant merge with this. So, this is basically theta of n square this solution of this is theta of n square. So, this is basically theta of n square, but again this is not a probable method because this recursive method we are not we are assuming all level it is going like this. So, it will be 5 by 16 to the power i at the i level, but that proof we are not doing.

So, but this is giving us a idea what could be the solution for the recurrence In the next class we will talk about probable method which is the substitution method where we will take help of the induction method proof.

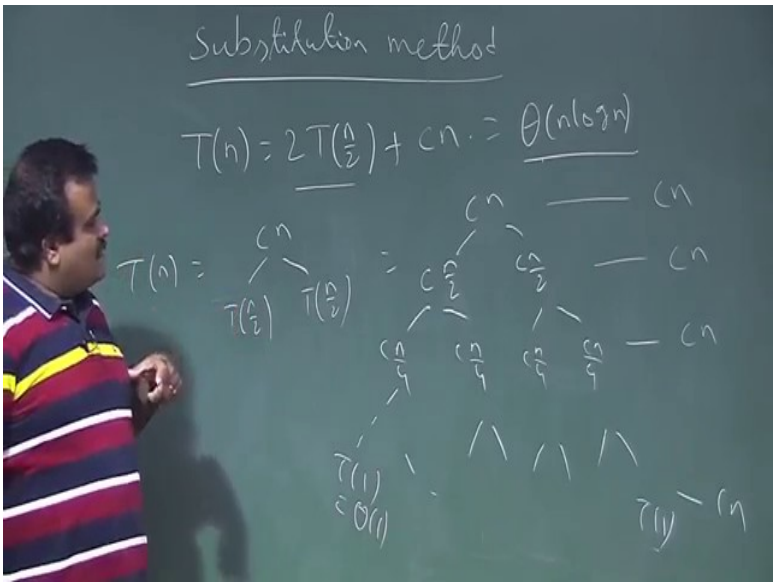
Thank you.

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

Lecture – 05
Substitution Method For Solving Recurrence

We talk about Substitution Method. This is the method to solve the recurrence. So, we have seen a recurrence which you are obtaining from the say merge sort.

(Refer Slide Time: 00:32)



We have seen the time complexity of merge sort. If we want to sort an array of size n then the recurrence we got T (n/2) + theta(n). So, this is merge cost and this is the cost for recursively solving the two sub arrays.

Now the question is how we can solve this type of recurrence. So, we have seen one method which is recursive tree method. So, in that recursive tree method we have seen that if we take this as c*n then we have seen T(n) is basically c*n + T(n/2) + T(n/2), then this is again a problem of size n/2. We further reduce this problem into sub problem. So, this is basically. So, c n by 2 c n by 2 and it will be T n by 4 T n by 4 T n by 4 T n by 4 like this. So, this way it will continue up to n is to the constant T 1.

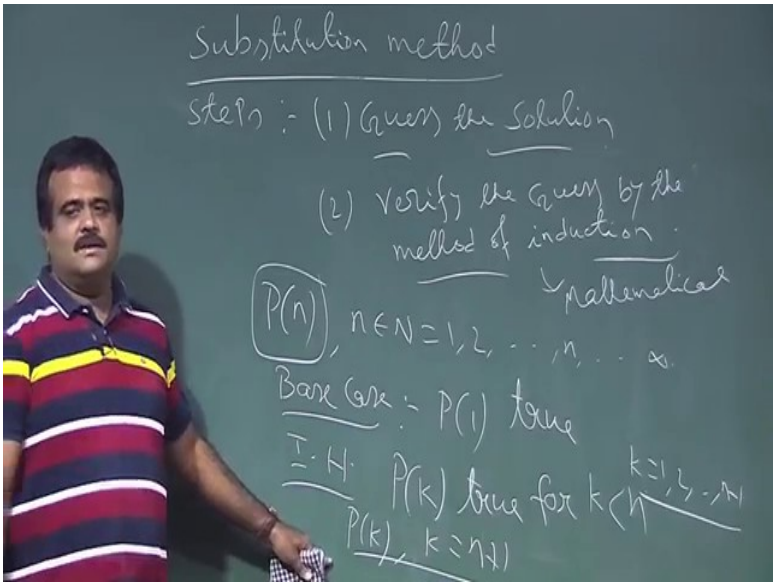
So, c n by 4 all are c n by 4 in this level. So, it is any at T 1 all the branches of this tree are ending at T 1 and T 1 is basically the problem of size one which is theta(1); cost to solve that

problem is theta(1). Now, the time complexity or runtime is basically sum of all these nodes, so to make the sum we have seen we can take c*n sum of the levels and we know the height of this tree is log(n). So, basically it is giving us the n*log(n) time complexity.

So, this is the recursive tree method which we have seen in the last lecture. This method has some drawbacks too. So, this is not a probable method in the sense that we are seeing this is c n c n c n, so is the proof that in the i-th level (also in the general in the i-th level) it will be c n. Now that proof we are not doing here. Unless we have to proof that by some way like method of induction or something or we can just say this is giving a rough idea of the solution.

So, this recursive tree method. Recursive tree method is not a full proof method in that sense. So, today now in this lecture we will talk a substitution method which is basically a method by induction; mathematical induction.

(Refer Slide Time: 03:50)



So, we will prove our solution by mathematical induction. So, basically it has following steps. So, we first guess the solution. Basically we have given a recurrence then we first guess the solution. That means, we assume this is the solution. So, then by the method of induction we verify our solution, verify the guess by the method of induction; so mathematical induction.

So, induction means anything we want to proof in terms of n; n is a natural number. Actually there are two version a mathematical induction: first version is suppose we want to proof some property that property is P(n) in terms of n where n is a natural number. So, we have to proof some property in terms of n for that mathematical induction is telling there is a base case, in the base case we prove that it is true for some lower value of n that is we can proof that the P(1) is true.

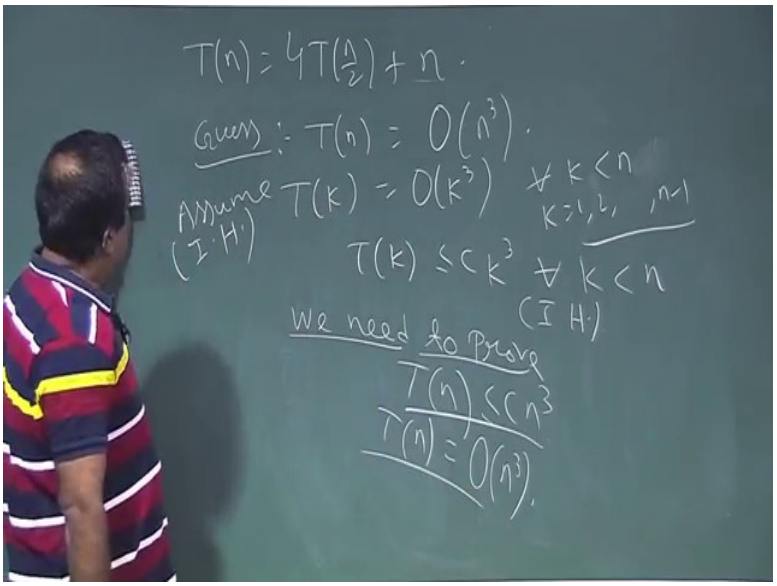
Then we have induction hypothesis step. So, we assume that P k is true there are two version for all k up to n. That means, P(k) is true for k = 1 … n-1. Then if we can prove that P n is also true; that means if we can proof that P(k) is true for k= n+1 if we can show that then we are done. Then we have a base case it is true for 1. Another version is; if we assume that P(k) is true for k = n and then if we can proof it is true for k = n+1 then it is true for all n.

So, we will take this version of the induction method so we assume that our result is true for all k up to n-1; that means, before n. And we need to prove that it is true for n. That means, we need to show P k is true. If we can prove that then we are done, then by the method of induction we can say this is true for all n. So, this is what we know the method of induction.

So, we will use these to verify our proof by the help of induction. So, given a recurrence; we first guess the solution and then we will try to justify our guess whether this is this guess is correct or not and that will do by the method of induction. And also we have to to solve the constant based on the base case.

Now let us take an example then it will be more clear.

(Refer Slide Time: 08:07)

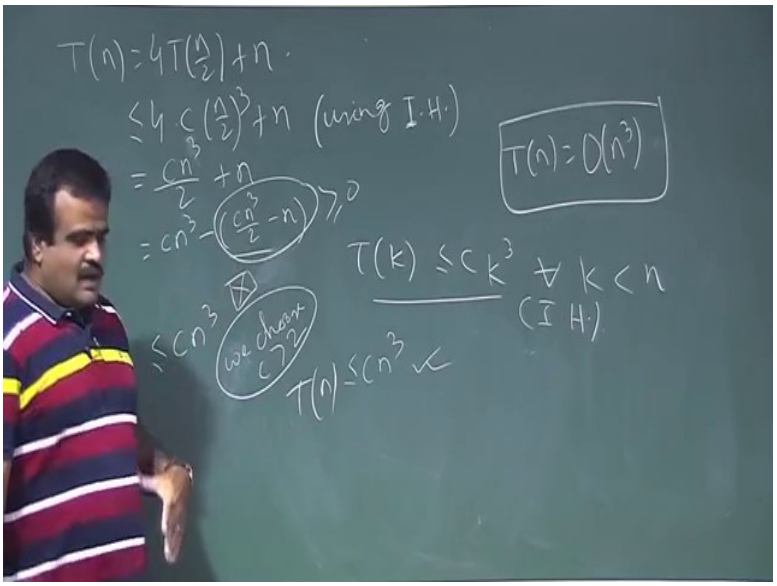


Suppose we have a recurrence like this T(n) is equal to 4*T(n/2) + n. Suppose this is our given recurrence and we need to solve this recurrence by substitution method. So, step one: guessing step. Suppose we are guessing the solution is O(n^3) suppose this is our guess. Now we need to verify this guess whether this is correct or not by the method of induction; so for that what we do is we assume this is the assumption. We assume this is the assumption or this is also called induction hypothesis. So, we assume that T k is basically order of k cube for all k up to n-1.

So, this is basically k is equal to 1 - to - n-1. This is our assumption we assume that our result is true for all k up to n-1. That means, what this means T(k) is less than c*(k^3) this is for all k up to n-1. So, this is our induction assumption or induction hypothesis.

Now we need to show, we need to prove; need to prove or need to show that T n is also less than c*n^3. If we can show this then T(n) is basically O(n^3). By assuming this if we can proof this then by the method of induction we can say this is true for all n; all natural number n.

(Refer Slide Time: 10:45)



Now, we write this recurrence, our recurrence is basically 4*T(n/2) + n.

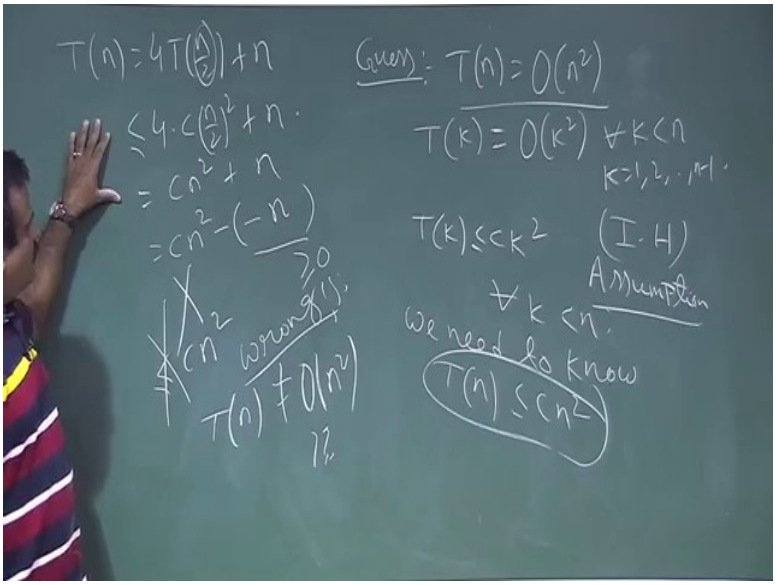
Now, we use this induction hypothesis. Now this is n by 2; n by 2 is less than n so n by 2 is k which is less than n. So, we can use this hypothesis. So, this is basically equal to; so c n cube by c n cube by 2 plus n. Now this we want to be less than or equal to c* n^3, if we can show this then we are done. For that we must able to write c* n^3 minus something and this quantity should be greater than 0. If it is greater than 0 then we have proved, then we can say this is less than c n cube.

So, let us check what is this quantity? This is basically c* n^3 this is (c* n^3)/2 - n. Now c is in our hand, c is a constant which we can play with (positive constant). So, if we just choose c to be greater than 2 then this guy is greater than 0 and then this is less than so it is done. That means, by taking this assumption we have shown this is less than c* n^3. By taking this assumption we have shown this result is true for k is equal to n also. And we can easily check the base case. Then, so the result is true for all n. So that means, T(n)=O(n^3).

So, the solution of this recurrence is O(n^3). This is the method, but basically we are using the mathematical induction. Now we want to guess whether it is n*n or not because this is our choice; we can have any guess. So, then we have to justify our guess by the method of

induction. So, this is our recurrence. So, O(n^2) is also basically O(n^3). So, we want to see whether we are getting this solution as big O of n cube or not.

(Refer Slide Time: 14:05)



So, this is our new guess; now we want to guess. Again we have the assumption step or the induction hypothesis step. So, we assume this is true for all k up to n; that means, k is equal to 1 to up to n-1. We assume that our result is true for up to n. This is our induction hypothesis, this is our assumption or induction hypothesis or this is our assumption.

That means T(k) is basically less than equal to c(k^2) for all k less than n. So, this is our assumption, now what we need to show? We need to show that T(n) is order of n^3; that means, this result is true for k is equal to n. That means, we need to show that T(n) is less than or equal to c(n^2), if we can show this then we can say T(n) is of the order of n^2.

So, how to show this? This is our recurrence, we will use this recurrence, so this is our k n by 2 is less than so this is our k, so we can use this induction hypothesis. So, this will be basically; this k is n by 2 which is less than n. So, we can use this induction hypothesis this is 4*c(n/2)^2 + n.

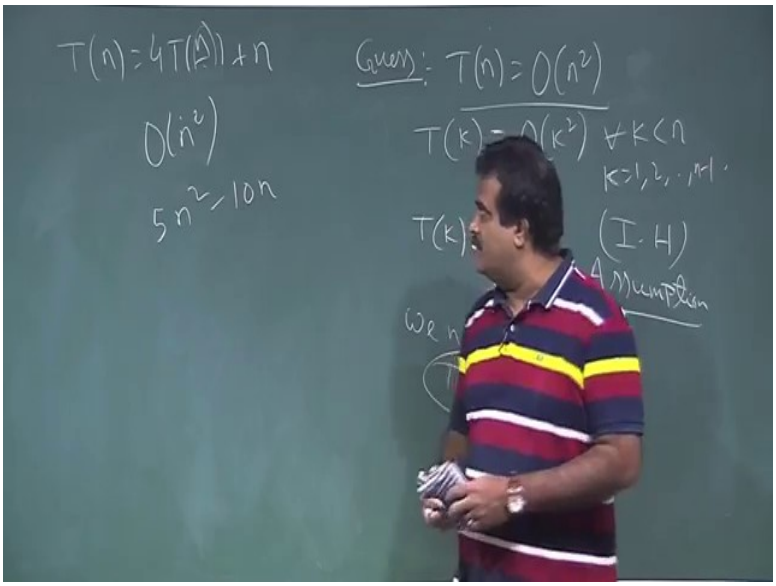
So, this is basically 4 - 4 cancels c n square plus n. Now, we want this to be less than equal to c(n^2) in order to get this result. In order to get this result we want this to be less than equal to c(n^2). So, for that what we need? We need this should be written as c(n^2) minus of something, and that something must be greater than 0. But what is that? That is basically

minus n which cannot be greater than 0 because n is natural number n start from 1, 2, 3 like this. So, the minus is cannot be natural number.

So, this is wrong, so this cannot be less than this. So, this is not true. So, then what will be our conclusion? So, if this is wrong can we just say that T n is not equal to O(n^2) or what, because we are not achieving this we took the assumption this is for up to n; now we try to prove it this will be true for n also, but we are not achieving this. So, can you conclude that this will be not solution, this is not O(n^2).

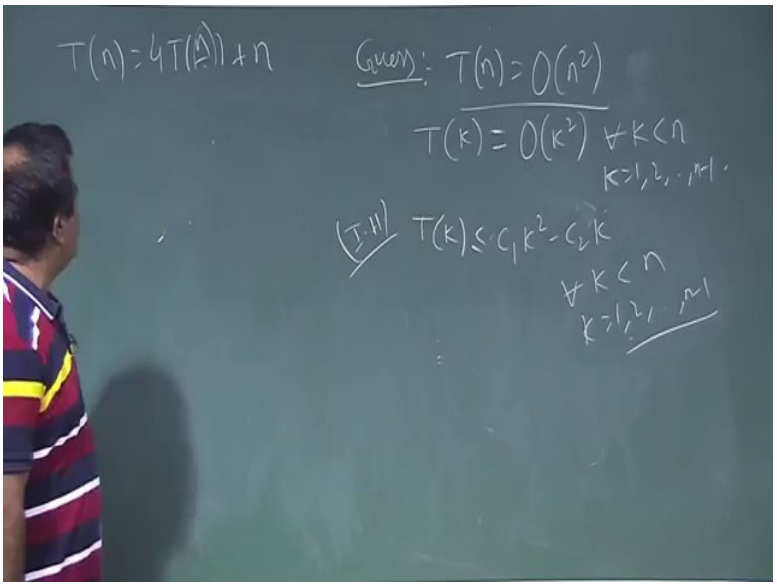
So, this way we cannot achieve this. Now the question is can we tighter this bound, can we have a tighter bound than this?

(Refer Slide Time: 18:43)



So, O(n^2); so this is also O(n^2) say 5*n^2 - 10*n this is also O(n^2), because we ignore the lower order term then we ignore the leading coefficient. So that means, this bound we want to make it tight. So, we assume this is of the order of O(n^2). Now here we just check instead of this bound we take little tighter bound.

(Refer Slide Time: 19:17)

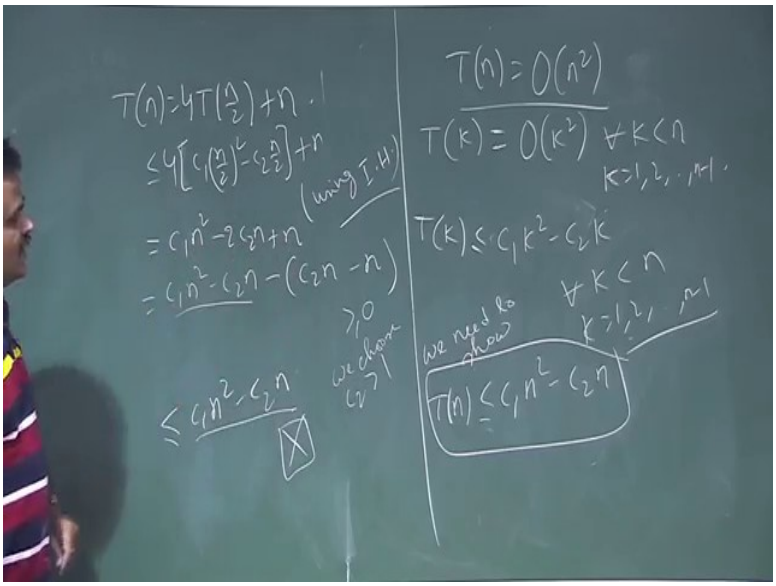


So, we take this to be c1*k^2 - c2*k^2 because that is also O(k^2), because this is the lower order term anyway you are going to ignore the lower order term. So, this is also O(k^2). So, we will take this tighter bound and then we will see whether this will help us to achieve that.

So, we just assume this is O(k^2) and O(n^2) means this also earlier bound was this is the more tighter than the earlier bound. And with the help of this tighter bound we want to see whether we can justify if the solution is O(n^2). So, this is basically our induction hypothesis step.

So, this is basically true for all k less than n. So, this is basically k is equal to 1 to n-1. Now we use this bound here to achieve that, so let us try that.

(Refer Slide Time: 20:35)



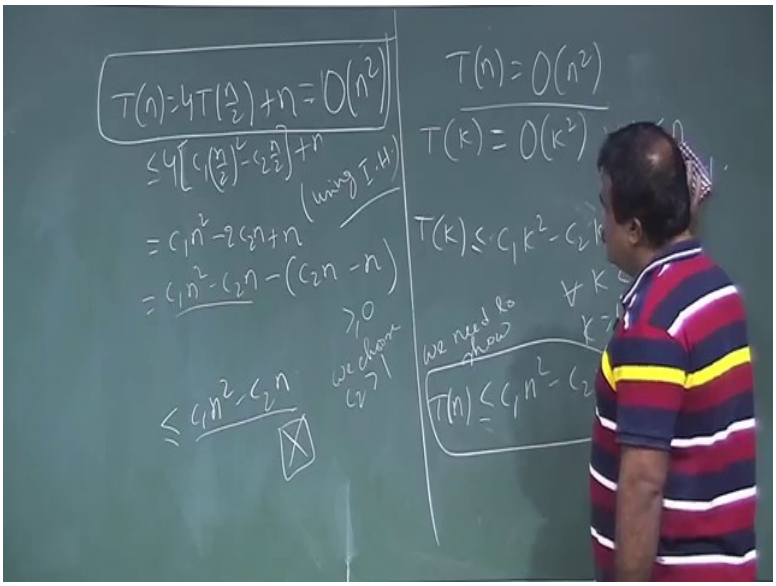
So, let us write our recurrence. So, this is our induction hypothesis, so let us write our recurrence T(n) is equal to 4*T(n/2) + n. Now we assume the induction hypothesis so this is basically less than equal to 4*c1*k^2. So, (n/2)^2. so this we can take another bracket minus c 2 n by 2 plus n this is we are. So, this is basically we are using this, this is we are using the induction hypothesis the assumption; this assumption we are using.

So, this is basically what? This is basically is cancelling out, so c 1 n square minus 2 c 2 n plus n; just we are simplifying this. Now this is the assumption we made. So, what we need to show? We need to show that this is true for k is equal to n also. That means, we need to show; so we need to show that T(n) is less than equal to c1*n^2 - c2*n, if we can show this then we are done. So, that is what we want to achieve. So, we want to show T n is less than this. So, in order to do that; so this is basically we want to show this is less than equal to c1*n^2 - c2*n.

So, to show this we can write this as c1*n^2 - c2*n -(something). So, this is the quantity we are looking for "(-something)" and if this term is greater than 0 then we are true. Then we can say this is less than this and we are done. So, what is this term? This term is c2*n - n . Now c 1, c 2 is the positive constant we can play with we can choose c 1 c 2.

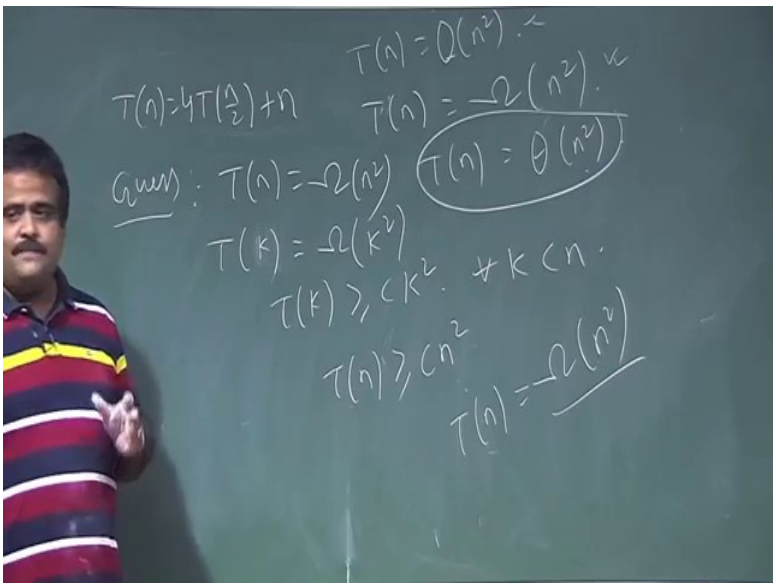
So, if we choose just c2 is greater than 1 then this is positive; thus we choose c 2 to be greater than 1. Then this is positive and this is done and it is true for k is equal to n also.

(Refer Slide Time: 23:58)



That means, the solution for this is basically big O of n square. So, this we prove by the help of mathematical induction. So, this is the big O of n square, now we want to see whether this is big theta. So, for big theta what we need to show? We need to show this is big omega. So, basically we want to achieve this solution, we have seen this is big O of n square.

(Refer Slide Time: 24:40)



Now, next we want to see whether this is big omega of n square and if both this satisfy if this is big omega and big O then it is basically big theta of n square.

So, now the question is how we can show this is big omega of n square. To show this we will follow the similar path, similar way. So, we guess this is our guess; we guess that T n is equal to big omega of n square so that means, we assume that T k is basically big omega of k square. And so we will do the same thing. This is the mathematical induction, so we will have this is greater than equal to some c*k^2 so this is true for all k less than n, then we will try to show that T n is greater than equal to c*n^2. Or else we have to tighter this bound. So, this way we will continue and we will get we can easily prove that T n is equal to big omega. This I am leaving you for exercise, just complete it. And we can show that this is a big omega of n square. Once we achieve this big omega of n square we have seen this is big O of n square, so both will give the solution as big theta of n square.

So, this is the substitution method. So, this is a full prove method in the sense here we are taking help of mathematical induction and we are justifying, we are proving our recurrence is true for all n. So, basically it has two main steps: one is assumption- we have to guess the solution. So, where from we can guess the solution? This recursive tree can help us to guess the solution. Suppose this is our say recurrence, now initially we guess whether it is n cube one can guess whether n square n to the power 4. So, how to get a guess? So, for that guess one can take help of the recursive tree and the recursive once we get some solution of the recursion from the recursive tree then we can justify the solution by using the mathematical induction. So, that is the substitution method. So, for the guessing purpose we can take help of the recursive tree.

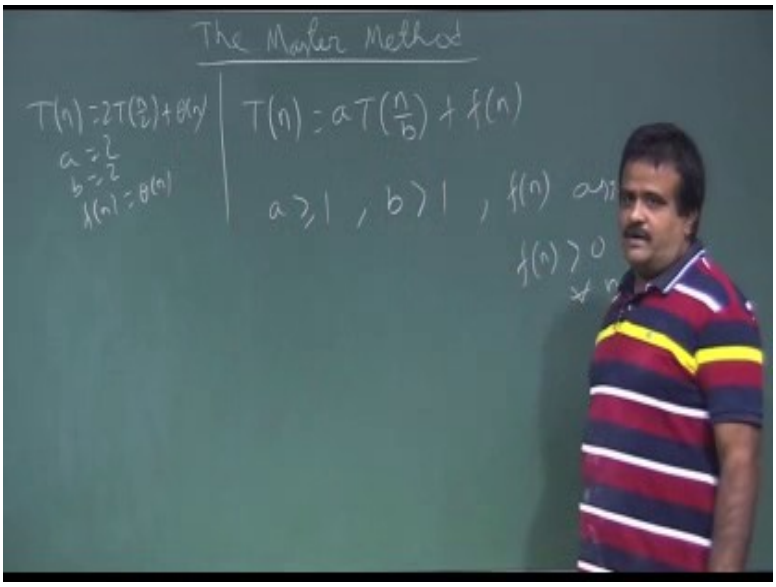
Thank you.

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

Lecture – 06
The Master Method

So we talk about another powerful method to solve the recurrence, which is called master method.

(Refer Slide Time: 00:36)



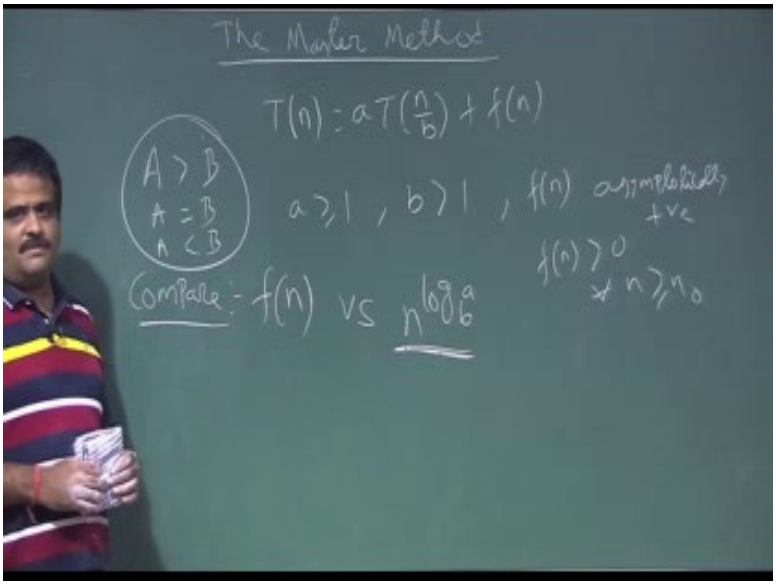
In this method the recurrence is of a particular form like T(n)= a*T(n/b)+f(n) where a is greater than equal to 1 and b is greater than 1 and f n is f n is positive function f n is basically called asymptotically positive so; that means, f n is greater than 0 for all n after starting value of n 0.

So, if it is negative then for the initial few values it is negative otherwise it is positive. In the most of the example we will see f n is throughout positive. So, if our recurrence is of this form then only we can solve it, we can we could it could be solved by master method for what it is there are three case of the master method we'll come to that what are the cases. So, now, the question is where from we can get this type of recurrence which algorithm can give us this type of recurrence. So, again we can think for a divide and conquer technique.

So, we have a problem of size n, we divide the problem into subproblems; a means sub problems. So, at least we should have one sub problem and the size of the sub problems is n/b. So, that is it should be able to reduce the size. So, that is why b is greater than one and this is the cost for combining the solution of the sub problems. So, this type of recurrence we can easily see like in merge sort. Merge sort is of this this type of recurrence. So, merge sort is T(n)=2*T(n/b) + theta(n).

So, merge sort a is equal to 2, b is equal to 2, and f n is equal to theta(n) or c (n). So, this is merge sort. This comes under this category.

(Refer Slide Time: 02:55)



Now in order to get the solution of this master method we need to compare. We compare basically 2 terms one is f(n) which is the merge cost. So, basically we compare f(n) verses n^(log(a) to base b). So, f(n) is the merge cost and as if this is the cost for recursively how much time you have spending here. So, we are solving this problem recursively we have a means sub problems.

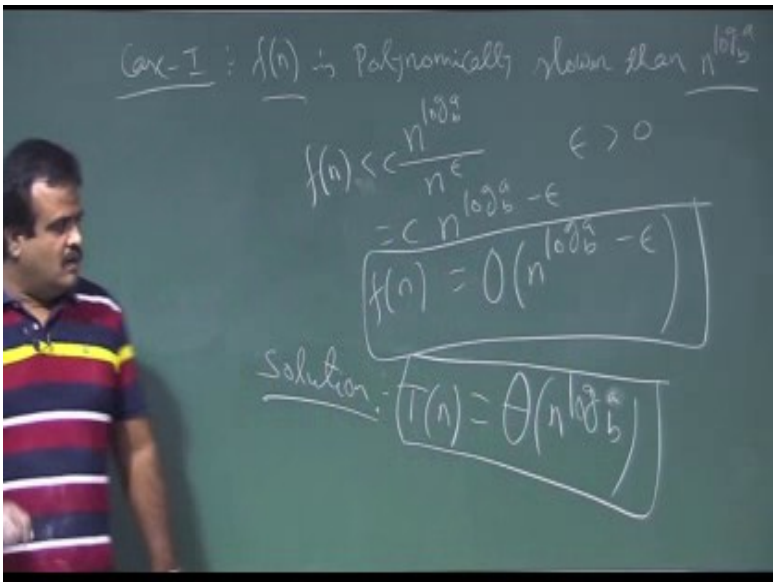
So, this is the time how much time we are spending here to solve this problem recursively, and this is the time for merging the solution. Now the runtime will be the time which is dominating because run time is the how much time we have recursively spending here and how much is the merge time the total time, now if this time if as if for the time being let us take this is the time for how much time we have spending here solving this problem

recursively, now if the that time is more if say that is a say for a example 1000 seconds say for a example and say this is 5 seconds.

So, what is the solution? Solution is thousand seconds. So, that means, run time will be the time which is; if we compare these two now which is more, which is dominating in the asymptotic sense; because we are talking about runtime in the asymptotic notation. So, the runtime will be the time which is the more among these two, because this is the time that we are spending here and this is the time for combining or merging. Now the run time will be the time which is dominating. So, if we compare 2 term there are three possibilities.

If we compare A B say 2 two terms A B. So that gives us, A is greater than B; A is equal to B; A is less than B. So, there are three possibilities. So, that is why we have three cases of the master method based on these three possibilities. So, first case is this is bigger, second case is they are similar and third case is other one is bigger. So, now, in asymptotic sense how we justify, how we mention that, or how we represent that one is bigger than another? So, that is also an important issue. So, there are three cases based on this comparison.

(Refer Slide Time: 05:37)



So, first is say case one; case one means we want this to be bigger and this to be slower. So, how we can write this. Let us write this f n is polynomially slower, f n is polynomially slower than this term, n^$\log_b a$. So, f n is polynomially lower than this term. So, that means, this is bigger so; that means, n^$\log_b a$ if we divide by a polynomial term, then also a f n lower or

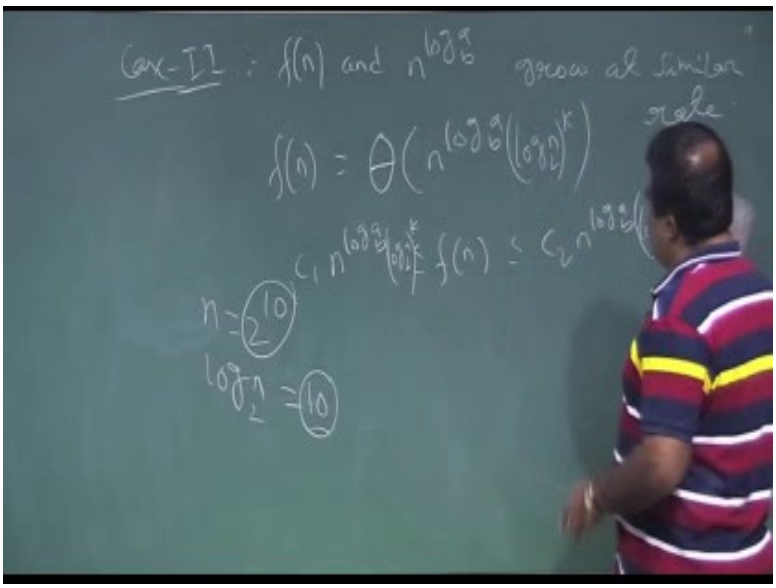
epsilon is greater than 0 if epsilon is one, then this is n, if epsilon is 2, then n square. So, this is a polynomial time.

So, if we divide this with the polynomial term then also f n is less than that. So, that is the says that f n is polynomially slower than this term. So, this is basically what. So, this is basically in the big O notation. So, this is basically c*n*log(-epsilon). So, that means basically f n is big O of n^($\log_b a$ - epsilon). So, this is the way we say one term is faster than the other term. So, this is the meaning of f n is polynomially slower than n^$\log_b a$.

So that means, even if we divide a polynomial factor with this, this is the n to the power epsilon, epsilon could be greater than zero; that means, it could be 1, 2 like this if it is 1, then this is n; n is a polynomial if it is 2; then n is square. So, these are polynomial terms so that means, if you divide a polynomial term with this then also f n is slower so; that means, this term is bigger this is the dominating term then what is the solution? This is called case one of the master method. Then the solution is the term which is dominating. Big theta of n^$\log_b a$. So, this is called case one of the master method.

So, this we have to remember. So, the n^$\log_b a$ is the dominating term. So, so this is case one; now case 2; case 2 is they are similar their growth is similar.

(Refer Slide Time: 08:54)

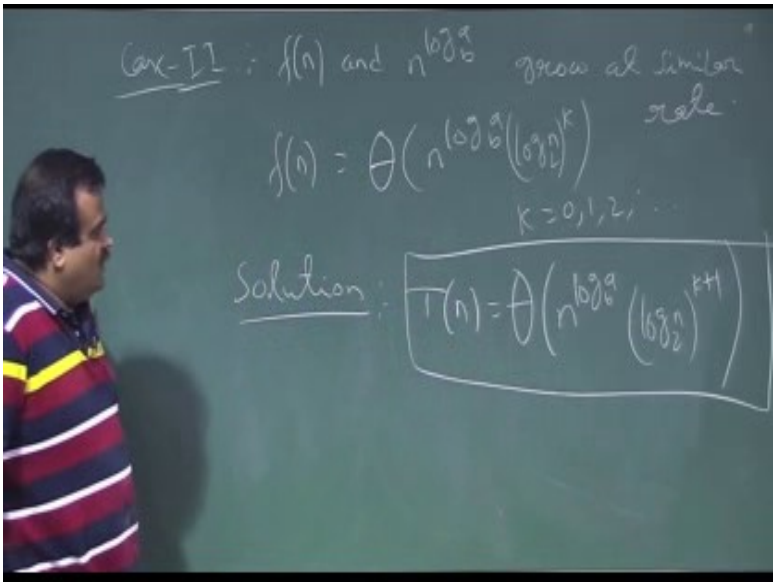


So, case 2 is basically they are similar. So, f n and n^$\log_b a$ the growth of f n and n^$\log_b a$ are similar. So, this and this grow at similar rate. So, say how we can represent this similar in a asymptotic sense.

So that means, if we can write this f n is equal to big theta of n^$\log_b a$ then they are similar. Because this means what this means there is 2 positive constant. So, f n is bounded both side by this so that means, they are exactly similar. Now here we are allowing that they could be different in the log factor because log n is small quantity like in terms of n because if say n is 2 to the power 10 then what is log n? Base 2 is basically 10. So, 10 is very less than compare to 2 to the power 10. So, log n is quite less number compare to n.

So, in that sense we are allowing them, they may differ in a log factor way or power of log factor. So, if this is a if they differ in the log factor way we can multiply this with log n log n base 2 or power of log factor. So this is the general term so; that means, f n will be $(\log_2 n)^k$.

(Refer Slide Time: 11:17)



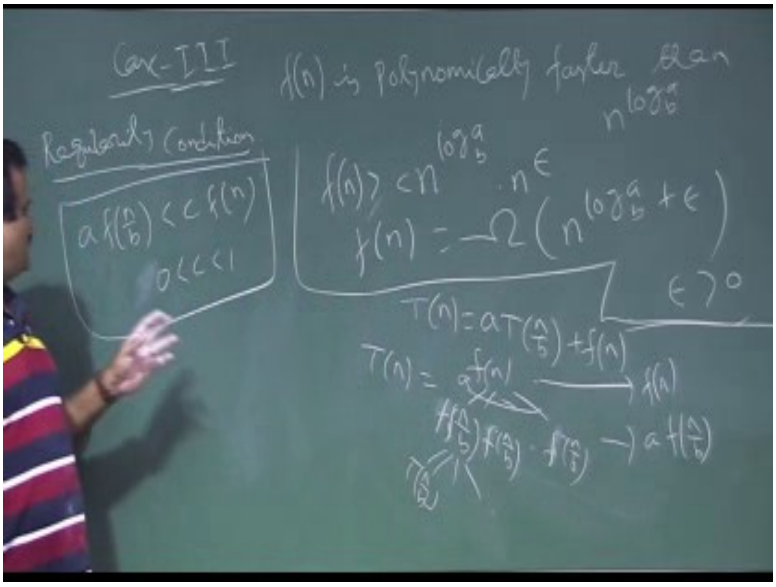
So, here this k could be if k is 0 then their growth is exactly similar there is n log factor, but if k is 1 then there is log factor difference that is allowed because log n is less than very much less than n.

So, then this is the case 2 then what is the solution? Then the solution will be. So, this T n is equal to big theta of n^$\log_b a$ and this log factor term will become to the power k plus 1. So, this is the solution for case 2. So, if k is equal to 0 then we have only log n because k 0 means

we have only log n here. So, we will take some example there. So, this we have to remember this is case 2 of master method. So, in this case their growth are similar now case 3; there is only one case left.

So, we have seen one is more we have seen this f n is less we have seen this f n and this n^$\log_b a$ are same or similar growth. Now case is when f n is bigger, f n is faster. So, again we have to use that polynomially faster way. So, let us just write that third case. So, this is the third case.

(Refer Slide Time: 13:08)



So, third case is f n is polynomially faster than n^$\log_b a$. Now how we can denote the polynomial faster in the asymptotic notation.

So, we can write so; that means, if we multiply a polynomial term with this n^epsilon then also f n is faster then also f n is greater. So, f n is greater than c of this so; that means, f n is basically this is big omega of n^($\log_b a$+epsilon) where epsilon is greater than 0. So, this is the way we represent in an asymptotic notation this is the meaning of f n is polynomially faster than n^$\log_b a$.

So, here in this case we have another double check we have another condition to check that is called regularity condition. So, this is a extra check point regularity condition. So, how we can get this regularity condition? So, let us just draw. So, what is our recurrence? Our recurrence is T n is equal to a*T(n/b) + f(n) so that means, we have a problem of size n we

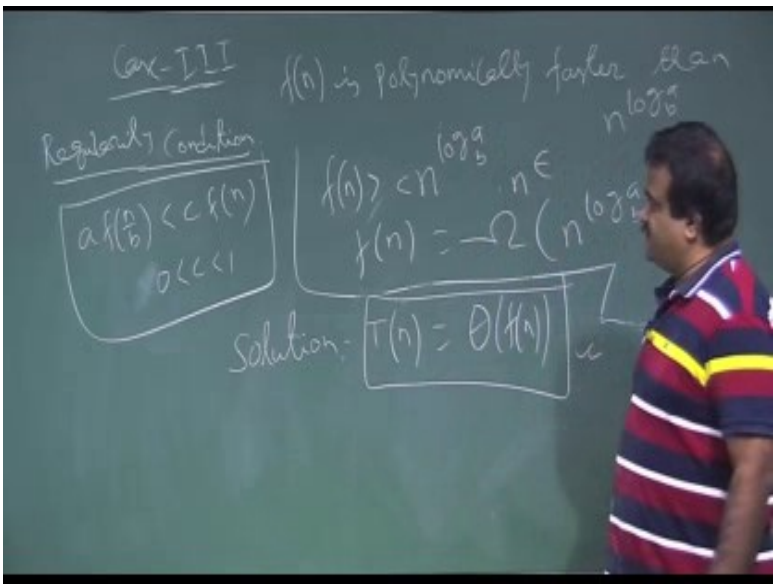
divide the problem into a means sub problems each of size n/b and this is the; f n is the cost for combining.

So, now if we draw the recursive tree. So, f n is the merge cost we have T(n/b), T(n/b), T(n/b), and this is a means. Now again this is the problem of size n/b we can further divide it into sub problem of size n/b square by this and this will be the f(n/b). So, all of this will be f(n/b). So, our total cost is basically sum of the. So, this level sum is f n and this level sum is a f(n/b). So, first level sum is f n. So, we want f n should be more or bigger so; that means, it should be bigger than at least the next level.

So, it should be much more bigger than this so; that means, if you take a fraction of it f n then also this must be lesser so that means, this is the regularity condition so; that means, a of f(n/b) must be less than equal to c*f(n) and this c must lie in between; so if you take a fraction of this then also f(n/b) must be less than that. So, that is the extra check point. So, that is the regularity condition. If this is more; this is more means if you take a fraction of this then also the next level cost will be lesser than that. So, that means, f n is really dominating.

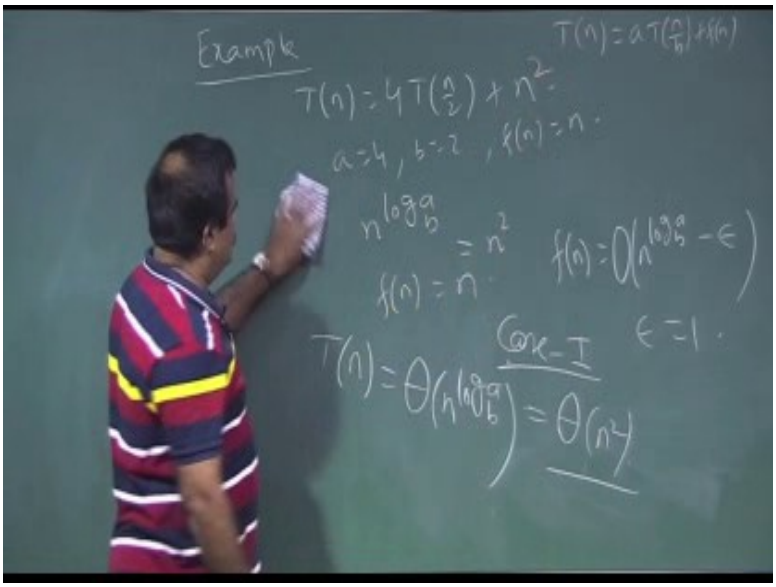
So, this is sort of extra checking this is called regularity condition. So, in that case this is the solution is. So, this is the case 3.

(Refer Slide Time: 17:05)



So, if we have case three then the solution will be this. So, solution is the dominating one that is f n. So, this is the solution for case 3. But for case 3, we have to remember we have an extra regularity condition.

(Refer Slide Time: 17:42)

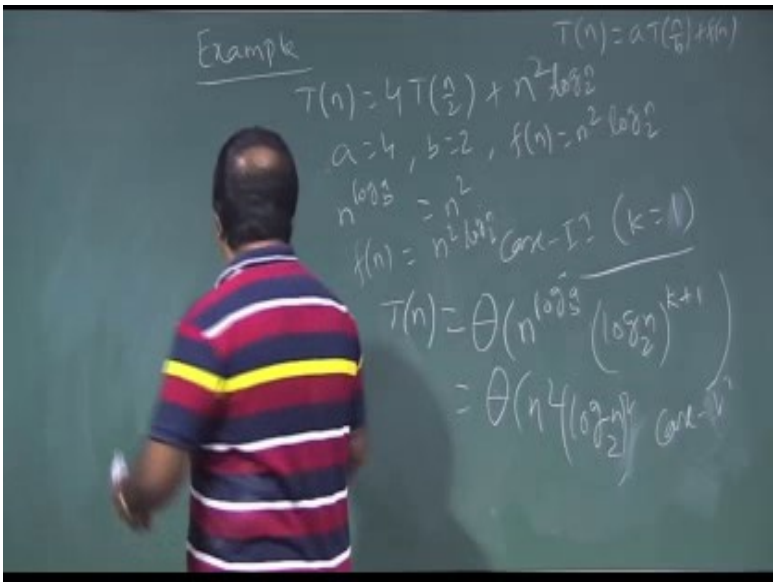


Now, we will take some example of these cases. So, we will take some example. So, let us take. So, if we take this T n to be 4*T(n/2) + n, and now we want to use master method on this recurrence.

So, what is the general form of master method? T n is equal to a*T(n/2), n/b + f(n). So, what is a over here? For this example a is 4, b is 2 and f n is n. So, now, we need to compute that n to the power $\log_b a$. So, what is n^$\log_b a$? So, b is 2 and a is 4. So this is n^2. So $\log_b a$ is 2 basically. So, this is n square and what is f(n); f(n) is a*n. So, which case is dominating? n^2 is more; n^2 is polynomially bigger than n.

So, that means f(n) is polynomially slower than n^$\log_b a$. So that means, we can write f n is basically big O of n^($\log_b a$ - epsilon). Case one for master method. So, case one of master method. So, T(n) is basically big theta of n^$\log_b a$. So, what is this? Big theta of n^2. So, this is the solution for this recurrence with the help of master method now we will check the case 2. So, a is 4, b is 2 thus, here f(n) is n^2.

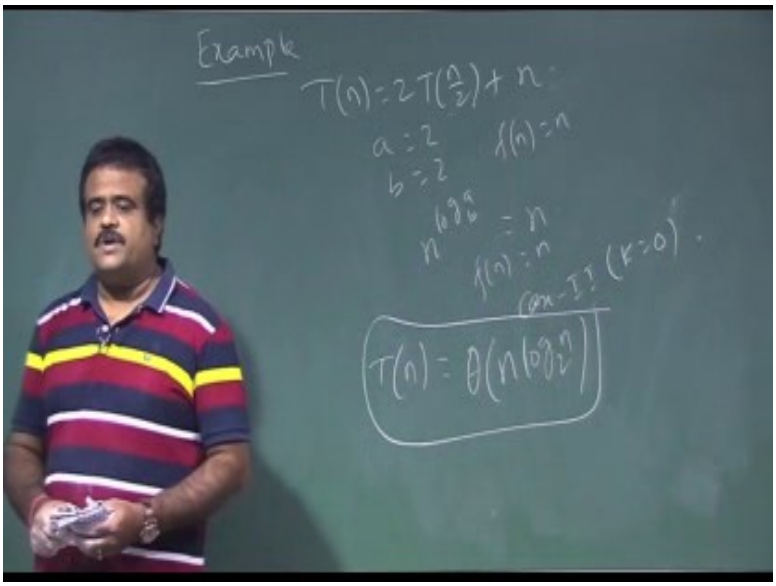
(Refer Slide Time: 20:08)



Now, what is $n^{\log_b a}$, $n^{\log_b a}$ is basically n^2. So, they are exactly similar. So, this is case 2 with k is equal to 0 they are exactly similar, i.e. there is no log factor. No logarithm term is involved there. So, case 2 with k is equal to 0 then what is the solution? Solution is basically T(n) is equal to big theta of $(n^{(\log_b a)} \times (\log_2 n)^{(k+1)}$. So, here k is 0.

So, it will be just log n. So, this is basically n^2. $\log_2 n$. So, this is the solution of this recurrence. Now if we have a $\log_2 n$ over here, then this is also case 2 this is if we have a log n over here this is also case 2 with k is equal to 1 and in that case solution would be n^2. So, this is basically case 2, yeah case 2 with particular value of k now we check the merge sort recurrence.

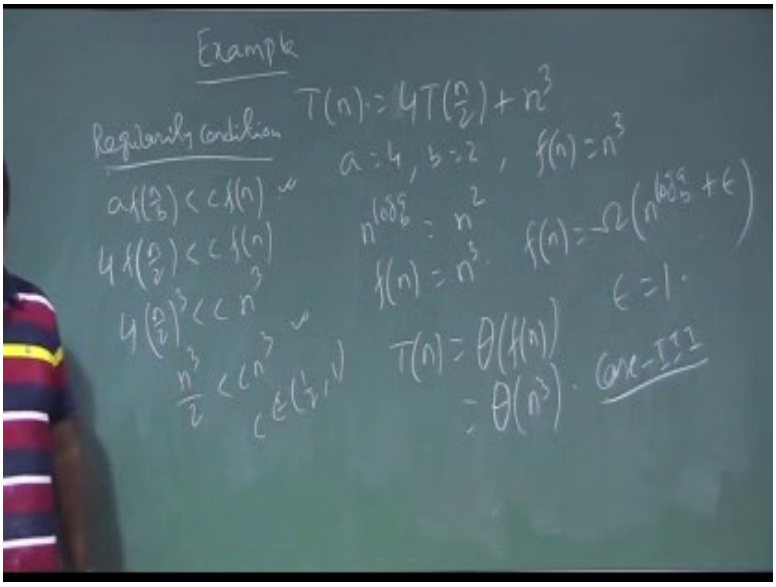
(Refer Slide Time: 21:55)



So, merge sort recurrence is T(n) = 2*T(n/2) + n. So, here a is equal to 2, b is equal to 2, f n is equal to n.

So, what is $n^{\log_b a}$? It is basically n and f n is also n. So, this is case 2 with k is equal to 0. So, case 2 of master method with k is equal to 0 then what is the solution? Solution is basically big theta of $n^{\log_b a}$. So, this is the solution we know for merge sort. Now we will talk about case 3 we take an example which will feed on the case 3.

(Refer Slide Time: 22:53)



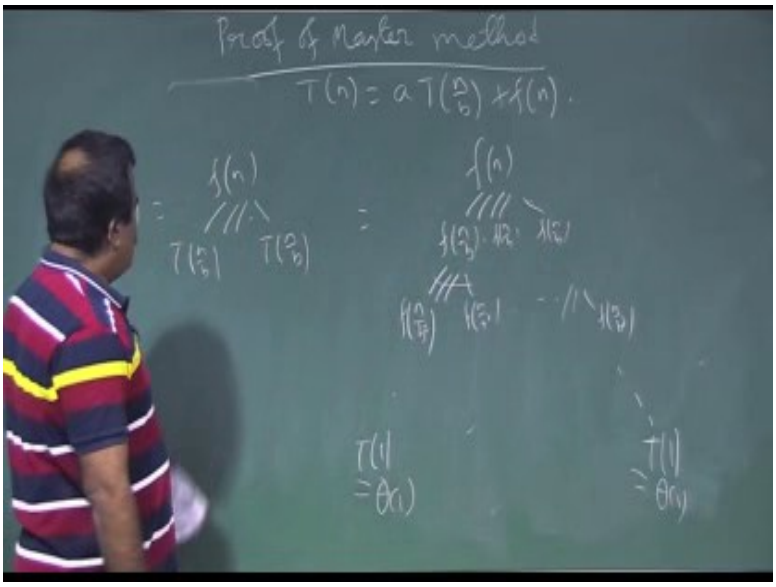
So, if we take this recurrence T(n) is equal to 4*T(n/2) + n^3. So, here a is 4, b is 2, f(n) is n^3.

So, f(n) is polynomially faster. So, this could be a candidate of case 3, but we need to check the regularity condition. So, f(n) is basically big omega of $n^{(\log_b a + \text{epsilon})}$. So, here epsilon is 1. So, this could be a candidate of case 3 provided we verify with the regularity condition. So, just let us check the regularity condition. So, this is the extra check point for the regularity condition. So, this condition is telling we should have a of f(n/b) is less than sum c of f(n).

So, we should get a c which is lies between 0 to 1 such that this will satisfy. So, a is 4, b is 2; so it becomes n/2. So, this is basically f n now f n is n^3. So, this is basically n cube by 2 less than c of n cube. So, for which one is of c this will occur. So, if we choose a any value c lies between half and one then this will satisfied. So, the regularity condition is also satisfying. So, then what is the solution? Solution is basically big theta of f n. So, order of n^3.

So, this is the solution for this, this is by case this is the case three of the master method, but we have to be careful regarding this extra check which is the regularity condition ok.

(Refer Slide Time: 25:38)

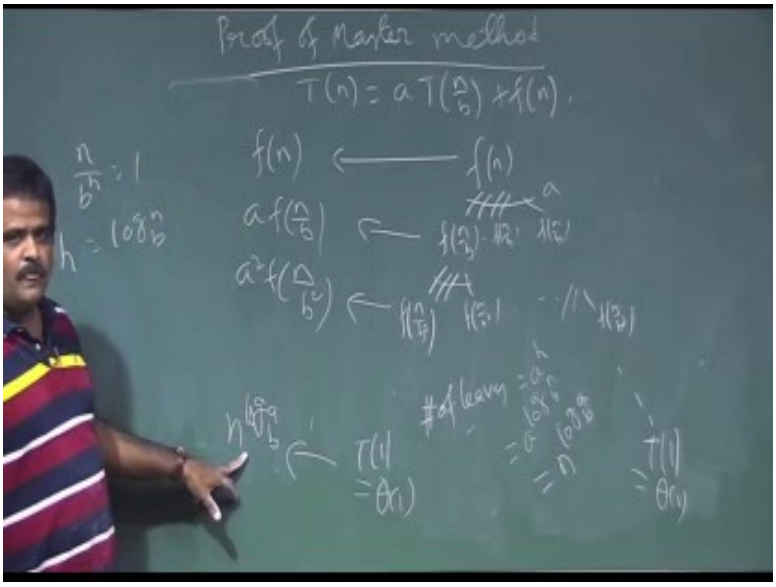


So, now we will see rough proof of this master method how this case is coming; proof of master method. So, this proof id is coming from; if you draw the recursive proof. So, this is

basically the recurrence T(n/b) + f(n). So, we tried to draw the recursive tree. So, T(n) is basically f(n) and then we have T(n/b), T(n/b). So, this we continue.

So, f n. So, each of this is f(n/b) and each of this again is basically f(n/b)^2 like this. So, this way you will continue until we reach to T-1. So, this is also f(n/b) dot dot dot. So, f(n/b). So, all are f(n/b)^2 like this, f(n/b)^2 like this. So, all the branch will stop at T-1 when the size is one. So, T-1 is basically theta of one. So, now, the cost is basically total cost, now what is the height of this tree? If the height is h then the height is coming like this (n/b)^h is 1.

(Refer Slide Time: 27:12)



So, height is basically $\log_b n$, now then what is the number of leafs? Number of leafs is basically.

So, at this level how many nodes are there a-many at this next level a^2. So, that means, at the level h, the number of nodes will be equal to the leaf. So, a to the power h. So, this is nothing, but a to the power $\log_b n$. So, this is our n to the power log a base b which we have comparing with f n. So, if you take the sum, this is f(n) this is basically a*f(n/b), this is a^2*f(n/b)^2 dot dot dot then this will be $n^{\log_b a}$.

So, this is the sum; now the case one is basically this is more. So, geometrically this is more; so that means, the solution will be this and case 2 with k is equal to 0 means they are similar. So, all the level they have similarity similar value of this.

So, the total cost will be this into height. So, that will be constant factor again. So, that is why for case 2, k is equal to 0 is coming by there. So, this into log of the height. That is why the case 2 and case 3 is dominating. So, this is the rough proof of the master method, but in the book in our text book Cormen, we have details proof by the induction method, if you have interest, you can have a look.

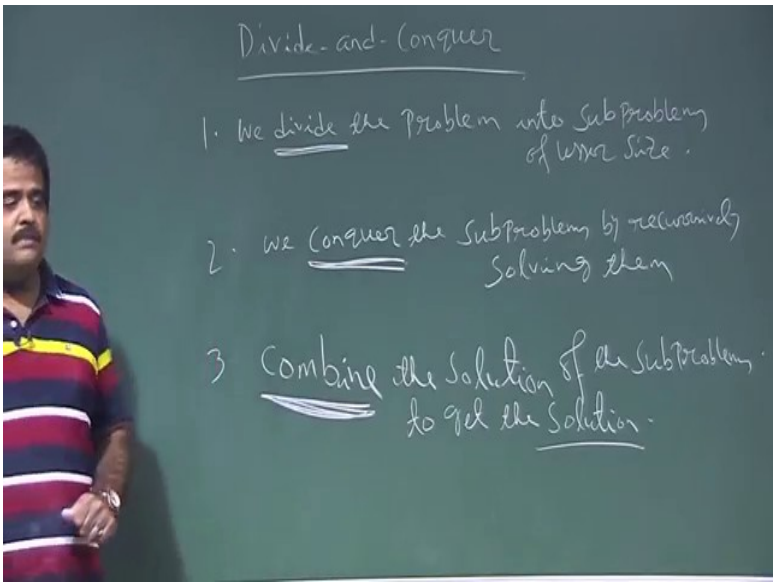
Thank you.

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

Lecture – 07
Divide And Conquer

So we talk about divide and conquer technique for solving certain type of problems. So, so basically it has 3 steps.

(Refer Slide Time: 00:35)



So, it is a design paradigm it has 3 steps. So, first step we divide the problem into subproblems so, that is the divide step. So, basically we have a problem of size n and we divide this problem into subproblems of lesser size, and then we solve these subproblems recursively that is the conquer step and then once we have a solution of these subproblems then we combine these solutions to get a solution of the whole problem. Let for example, merge sort. So, we have a sorting that is a sorting problem we need to sort an array of size n. So, we divide that into two sub arrays of equal size and we sort these sub arrays recursively, that is the conquer step and then we call the merge subroutine to combine to get a solution of the whole array. So, basically we divide the problem into subproblems of lesser size that is important.

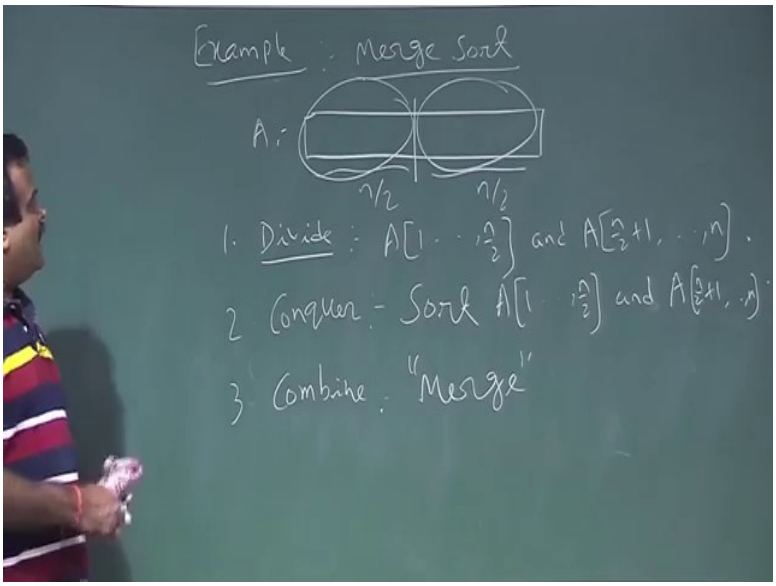
So, this is basically divide step then we conquer this subproblems by recursively solving. So, divide conquer step and then we have a combine step; that means, basically once we have the

solution of the sub problems then we combine by applying a combine step to get the solution of the whole problem. So, that is called combined step. We combine the solution of the sub problems to get the solution of the whole problem solution to get the solution of the whole problem.

So, this is the combine step. So in any divide and conquer technique they have 3 steps basically, divide step; we divide the problem into sub problem suppose we have problem of size n, we divide into the sub problems which is of lesser size then we recursively solve this sub problems by calling the same function recursively that is the conquer step, then once we have the solution of this sub problems we combine to get the solution of the whole problem. So, this is basically 3 fundamental steps for any different conquer approach.

So, the example is merge sort that is the divide and conquer technique.

(Refer Slide Time: 04:05)



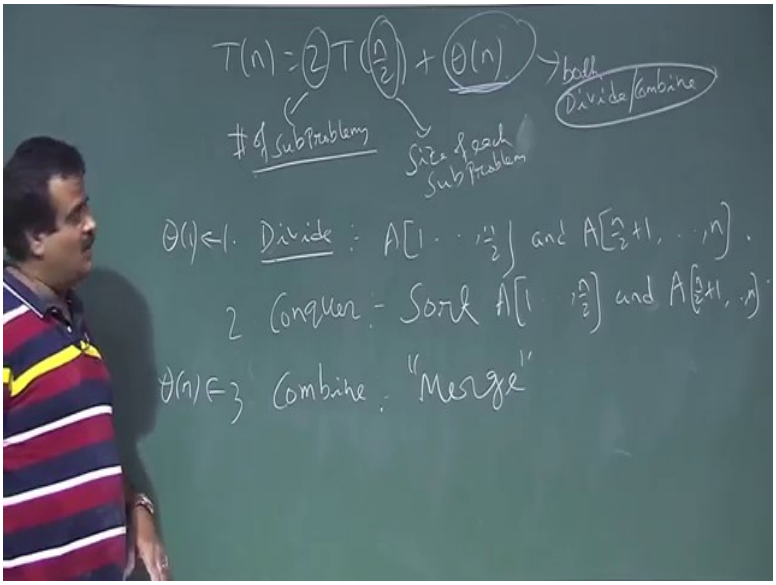
So, merge sort is an example of divide and conquer technique. So, basically it is a sorting problem. So, we have a. So, in merge sort, we have an array of size n this is the array of size n we are dividing the array into two subarrays n/2, n/2 and lower ceiling and upper ceiling, but if n is even though n/2 n/2. So, we that is the divide step we divide. So, this is the divide step we divide the array into two sub arrays.

So, this is the divide step we just divide into two sub arrays 1 to n/2 and n/2 + 1 to n and the conquer step is we sort this sub array and this sub array recursively by calling the same merge

sort. So, that is the conquer step. So, we are sorting this sub array 1 to this and we are sorting this right sub array. Now once we have two sorted arrays then we call the combine step; that is we call the merge sub routine to get a sorted array.

So, up to sorted array after this conquer step then we call the merge sort to get a sorted array. So, this is the merge sort and in a the recurrence is basically T(n) = 2*T(n/2) + theta(n).

(Refer Slide Time: 06:13)



So, any divide and conquer technique will have this type of recurrence here two is number of sub problems. So, here in merge sort we have two sub problems, and this n/2 meaning is size of each sub problems.

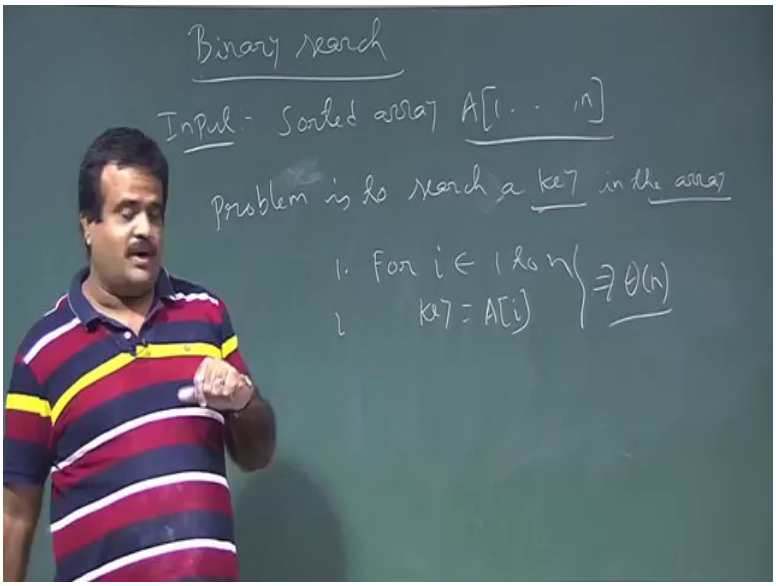
And this is basically merge sort; this is the cost for merge because merge sort this divide step is very trivial because we have just going into the middle just 10 by 2 we have just identifying n by 2 that is it is very trivial step for merge sort, but this is crucial this merge, merge will take linear time order of n, but this is cost for both divide and combine because when we talk about quick sort that is the another example of dividing conquer technique, there this division this divide step is more expensive than the combine.

Because divide step we need to call what is called a partition subroutine that will partition this array. So, this will be discussed at the time of quick sort. So, this is the time for both the step both divide and combine this is important. So, this is the any sort any general conquer approach having this type of recurrence where that cost will be denoting by the cost for both

the merging both the sorry both the division divide and combine, here division is trivial just say theta 1 here this for merge of this is theta 1 and this is theta n.

So, when we talk about quick sort this will be theta n and this will be nothing theta 1 may be. So, this is one of the example of divide and conquer techniques. So, we will discuss some more problem which can be solved using this technique. So, let us start with very simple one the binary search problem ok.

(Refer Slide Time: 08:59)



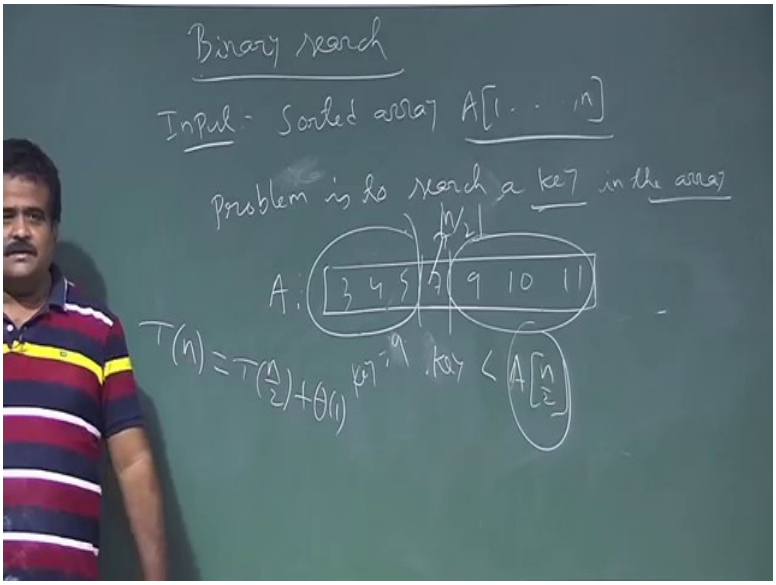
Binary search problem. So, what is the binary search problem? Binary search problem is basically we have a sorted array that is the input and we have another input as a key and we need to find a key whether this is in the array or not.

So, the input is we have a sorted array we have n numbers which is sorted and then another we need to search a and the problem is to is to search a value search a key in another input, this is also another input this is another integer we need to search this whether this is present in the array or not search key in the array. So, this is the problem. So, how we can do this search?

So, what is the naive approach if we have an array? So, we can just do the linear search just for i = 1 to n, we just check whether key is equal to A[i] .This is the search this will take theta(n) time because we are not bothering whether it is sorted or not, but we have an

information that they have sorted. So, that is why we need to make use of the fact that they are sorted. So, for that we will do the divide and conquer techniques.

(Refer Slide Time: 10:59)



So, this is an array which is sorted.

So, this is basically middle one n/2 for lower ceiling or upper ceiling anyway. So, now we compare our key with this value; if this value a*n/2 if key is equal n/2 then we are lucky we just return this. We got it sorted which is best case, so; there are 3 possibilities. Suppose this array is 3 4 3 4 5 7 9 10 11 and suppose we are looking for say 4 or we are looking for say 5. Suppose our key is 5 we are looking for 5. So, we check with the middle one middle one is 7 this is 7. So, 7 is not 5. So, 5 cannot be in the right part of the array. So, 5 has to be in the left part of the array if at all it is there. So, then we must look at the left subarray with the same key. So, that is the conquer step. So, in divide step we just compare with the middle element, if the middle element is equal then you return it if the middle element is less than then you look at the right part of left part of the array, and if the middle element is greater than suppose we are looking for say 9 then we must look at the this part of the array because we know 9 cannot be in the left part of the array.

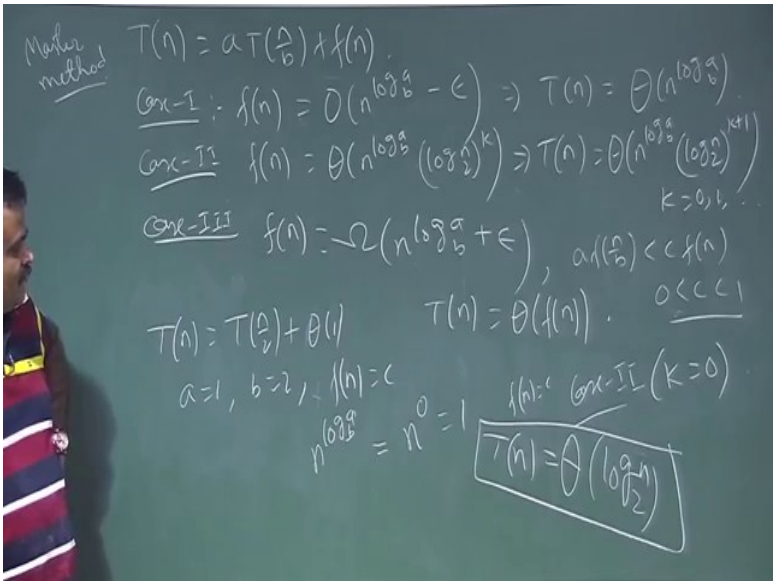
So, we will look at the right part of the array. So, this is what is called binary search algorithm. So, this is basically a divide and conquer technique. So, in the divide it is very clear; in the divide step we just go to the middle of the array element A[n/2] we which compare with the key, if the key is matching we are happy. So, we return the key this is the

best case otherwise if the key is less than that value middle one. So, you must have to look at the left part of the array that is the conquer step, then again we will search it in the technical left subarray. So, we reduce the problem into subproblems and then if it is greater then we look at this.

So, this is the divide and conquer technique. So, then what is the recurrence then. So, recurrence will be T(n) either we are going for left or we are going for right. So, we are again calling this search either on the left side whether it is worst case or best case we are getting immediately or in the right side. So, this is basically T(n/2) + theta(1), because just one comparison we are doing with the middle one. So, that comparison anyway you do not bother about which computer we are checking this comparison whether Pentium one processor or core two processor.

Because that comparison will take constant time so since we are using asymptotic notation we are going to use our good friend. So, this is our basically recurrence of this binary search. Now how to get the solution of this recurrence? Again we will take help of the what is called a master method.

(Refer Slide Time: 14:51)



So, how to get the solution for this? Let us recall the master method.

So, master method recurrence is a T(n/b) + f(n). So, it has 3 cases; case 1 is f(n), which is basically big O of ($n^{(\log_b a - \text{epsilon})}$).

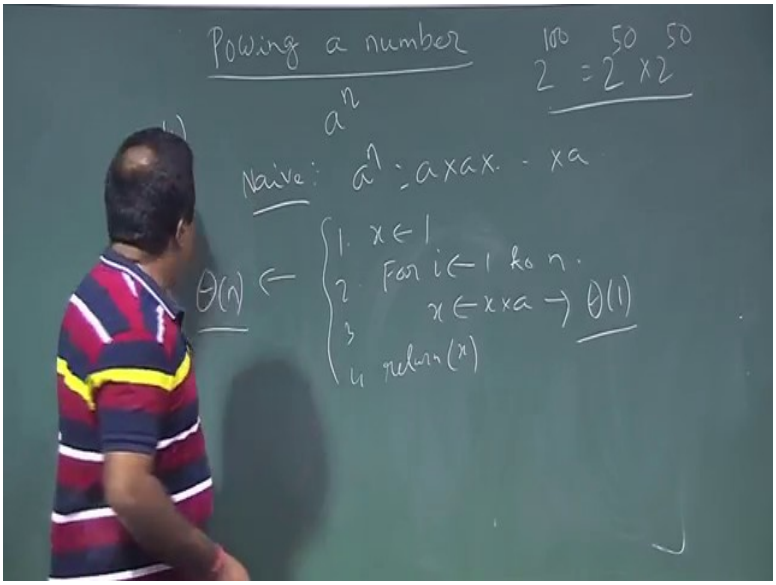
So that means, this is more, the solution is T(n) = big theta of ($n^{\log_b a}$) and case two is basically what case two is their growth is similar or they may vary by a log factor. So, this is basically f(n) is big theta of (($n^{\log_b a}$) x ($\log_2 n)^k$); then we vary in a log factor or power of log factor. So, in this case solution is basically big theta of (($n^{\log_b a}$) x ($\log_2 n)^{k+1}$) and this k is an integer and then we have a third case.

So, we will come back to this case 3 that is f(n) is more. So, f(n) is big omega of ($n^{(\log_b a}$ + epsilon)). So, here epsilon is positive, and we have the regularity condition like a*f(n/b) must be less than some c*f(n) and c lies between 0 and 1 this is the regularity condition we have. So, in that case solution is basically big theta(f(n)). So, this is master method now. So, now we have a recurrence T(n) = 2*T(n/2) + theta(1) that means, c is just a constant or we can write theta 1.

Now what is a? a is 1, what is b? b is 2 and what is f(n)? F(n) is basically constant theta(1) now we need to calculate $n^{\log_b a}$, $n^{\log_b a}$ this is basically n^0. So, this is basically 1 and f(n) is also constant. So, case 2 is with k = 0 and then what is the solution? Solution will be T(n) big theta of $n^{\log_b a}$ that is basically 1 and $\log_2 n$. So, this is the solution for the binary search and this we are getting by the help of master method.

Now let us talk about another problem which can be solved using the divide and conquer technique, which is called powering a number. So, this is how to find a^n; say 2^{1000} or 5^{100}.

(Refer Slide Time: 18:56)



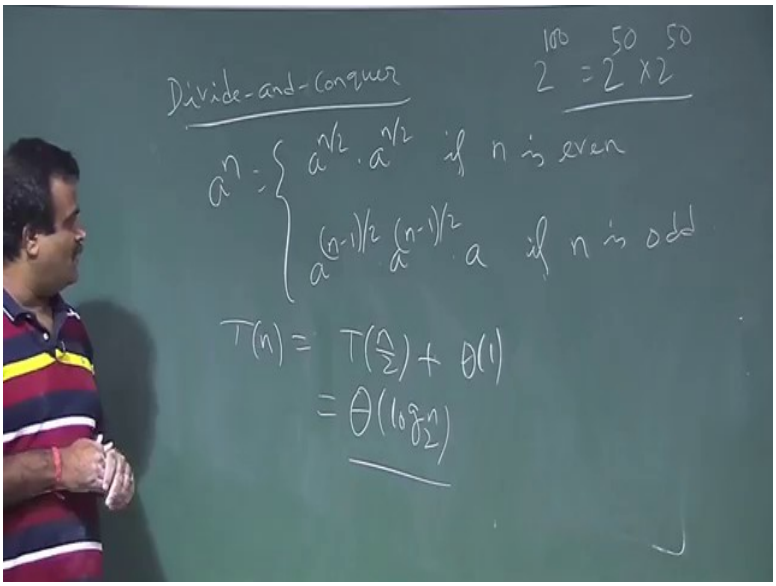
So, this is the second problem “powering a number”. So, basically we want to find a^n. So, for example, 2^100 we want to find or say 5^50 something like that. So, a to the power n where a or n are both integers. So, then now what is the a^n? This is the problem. Given a, given n, we need to find a^n. So, this is the problem. So, what is the naive approach?

So, a^n basically a x a x a x a…. n times. So, this is basically a for loop just simple code. So, before that you need to take a = x = 1 and then for i is equal to 1 to n: “x = x into a”. So, then this is the code then return x. So, x is basically a^n. So, then what is the time complexity of this code how many multiplications are happening? So, this is the loop of size n, so we have n-1 multiplications because this is a^n.

So, we have order of n multiplication and again each multiplication is taking theta 1 time again, we are in asymptotic notation we do not care about how much time our hardware is taking to multiply two numbers two integer. So, that time we are not really bothering. So, that time we are treating as theta(1). So, this time we are treating as theta(1). So, that is the asymptotic sense. So, this algorithm is basically theta(n) algorithm because we have a group of order n. Now the question is whether you can do something better than this.

So, now particularly the question is whether we can have a divide and conquer approach to solve this problem of powering a number. So, that is basically the idea is if we have to find 2 to the power 100. So, this we can find 2 to the power 50 and then you multiply it with it 2 to the power 50 again. So, that way, we have a problem of size high rate. So, you reduce this problem into subproblems say n by 2 like this.

(Refer Slide Time: 21:56)



So, that is the idea. So, let me write the divide and conquer step. So, this is the divide and conquer step for this problem powering a number.

So, what we are doing a to the power n can be written as a to the power n by 2 into a to the power n by 2, if n is even otherwise it is a to the power n minus 1 by 2 into a to the power n minus 1 by 2 into a, if n is odd that is it. So, this is the divide and conquer step. So, basically this formula give us the algorithm, this is the divide conquer step. So, we have a problem of size n we need to find out a to the power n. So, instead of find a to the power n we divide this problem into subproblems, will find a to the power n by 2 then once we get the solution of a to the power n by 2, we multiply with itself then that will give us the.

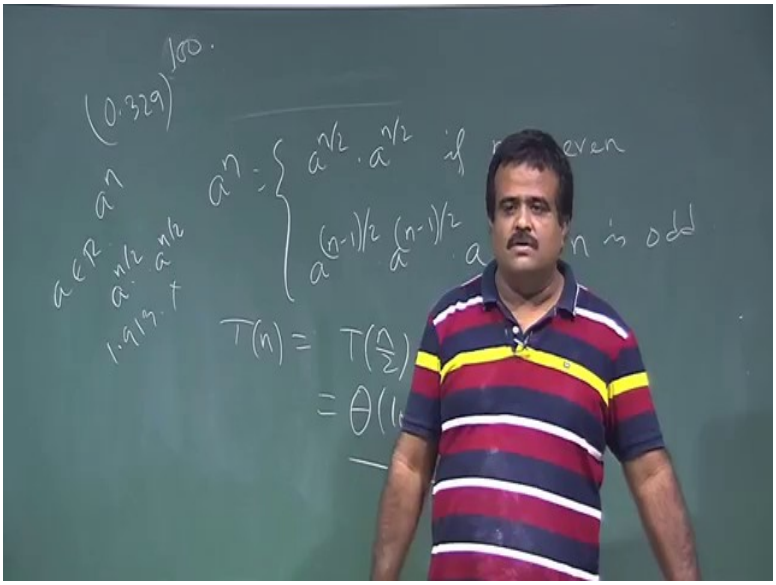
So, what is the divide step? Divide step is basically we just get a to the power we just get n/2 and the conquer step means we just try to get a^(n/2) that is the recursive call and then once we have the solution then we use combine step; combine step is this multiplication if it is even. If it is odd then, we have a^((n-1)/2) this is the divide step and then we recursively call this a to the power n minus that is also reduce the problem into subproblems of size ((n-1)/2) and then combine step.

So, then what is the recurrence? So, how many subproblems? Two subproblems. So, T(n) is basically 2*T(n/2) and then the divide step and combine step. So, even if it is even we have all multiplication even if it is odd we have 3 multiplications when the divide step is nothing. So, that is basically theta 1. Is this recurrence correct? no this is not correct because how

many sub problems do we need to really calculate twice of this because once we calculate this value a to the power n by 2, then we can store it we do not need to calculate it twice. So, this factor of two will not be there.

So, we have basically one subproblem. So, if we calculate a^(n/2) is stored somewhere and we multiply it with itself to get the combine step. So, this is the recurrence. Now what is the solution of this recurrence? Recall that this recurrence we have also seen this thing for the binary search method. So, this is basically log(n). So, this is a powering a number now here we are assuming these numbers are integers, but what if numbers are not integers then the problem is suppose we want to find out 0.329^500 or say 100 if we are dealing with the real numbers then it is an issue.

(Refer Slide Time: 25:33)



So, then it will be divide. So, this is a^n where a is a real number if it is a real number, a positive real number otherwise we will have the imaginary part of it. So, anyway, then the problem is a to the power n/2 will be a real number say suppose this is a 1.9123. Now if you multiply two real numbers then we have to think of their round off or precision. So, these are all the issues that will be involved when we deal with the real numbers. So, that will take more time than these steps, so then only the multiplication.

So, that is why in this powering a number we restrict our self on a^n where a is an integer. So, otherwise we will have to deal with the real number precision and round off all these things will be coming.

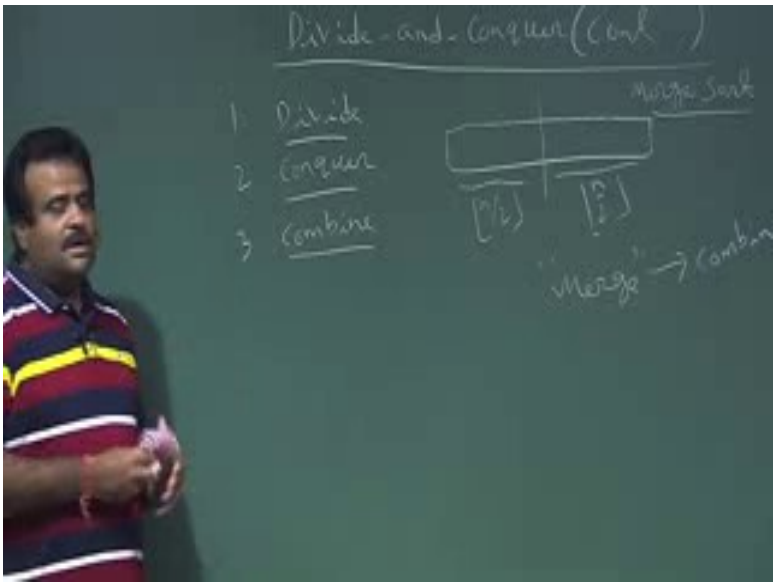
Thank you.

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

Lecture – 08
Divide And Conquer (Contd.)

So we are talking about divide and conquer technique. So, it is a designed technique. So, it basically has 3 steps. One is divide.

(Refer Slide Time: 00:34)

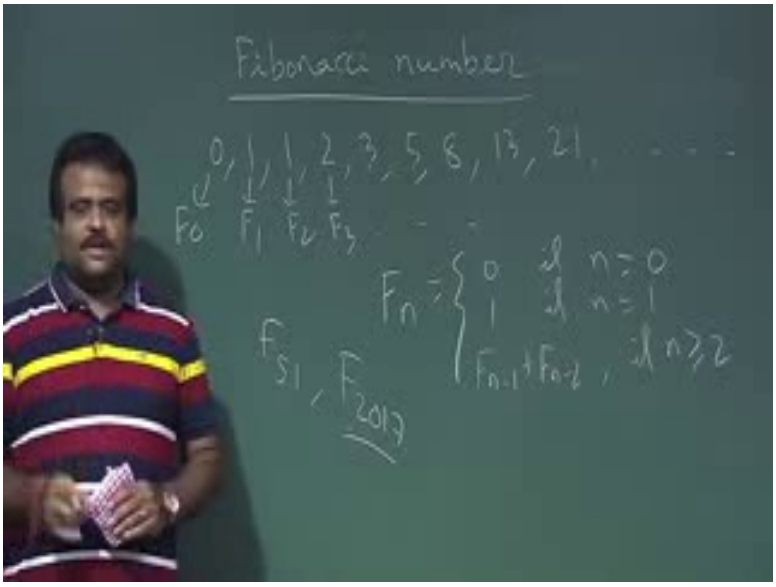


So, we are given a problem of size n, we divide the problem into subproblems. So, that is the divide step and then, we have the sub problems which are of lesser size, which is not now in n. So, which are of lesser size now we solve the sub problems by recursively solving them so that is the conquer step and then once we have the solution of these 2 sub problems then we combine them; we combine the solution of the sub problems to get a solution of the whole problem. So, this is basically 3 fundamental steps of any divide and conquer technique like in merge sort. Merge sort is a example of divide and conquer technique. What we are doing, we have an array of size n which needs to be solved, now we divide this array into 2 subarray with equal size and lower ceiling and upper ceiling n by 2, then this is the divide step. In merge sort this is the divide step in merge sort and then what we are doing; we are recursively sorting.

So, this is a sub problems we reduce the problem. Our problem is sorting problem, we reduce the problem size from n to n by 2 that is the divide step. Now the conquer step; we sort this sub array we sort this sub array recursively. So, that is the conquer step, but when you sort this sub array this is the sub problem; this is also sorting problem, but the size is not n size is n by 2. So, we call same merge sort on this, now that is the conquer step. Now once we have the solution for this sub array and this sub array; that means, once these 2 subarrays are sorted then we call what is called merge; merge subroutine.

So, that is the combine step; this is the combine. So, we have a solution up to sub problems then we solve we get the solution of the whole problem by the combine step. So, these are 3 fundamental steps for any divide and conquer technique and we have seen few examples like binary search, powering a number. So, today will talk about 2 more examples which can be handled by divide and conquer technique.

(Refer Slide Time: 02:56)

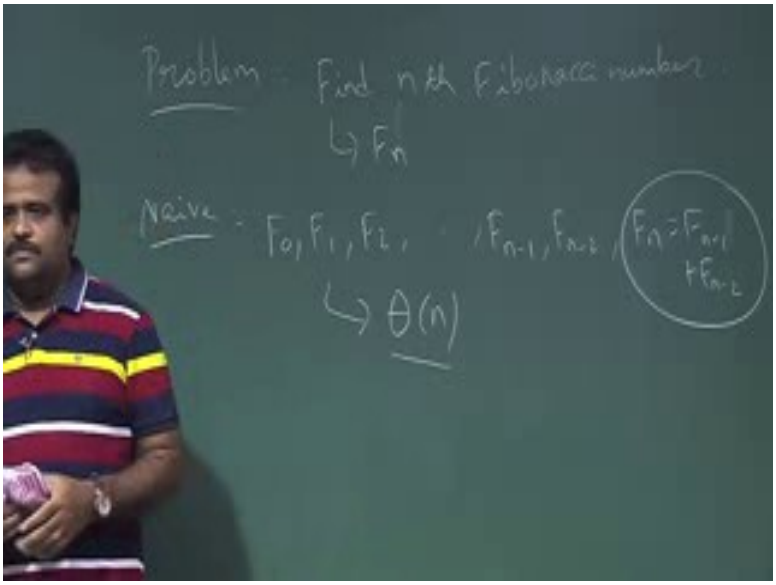


Today's problem is first problem is Fibonacci number. So, problem is to find Fibonacci number. So, what is the Fibonacci number? So, it is start with 0 then next Fibonacci number is one, and then next one is add the previous two, 1, then 2 then 3, then 2 + 3 5, 5 + 3 + 5 + 3 8, 8 + 5 13, and then 13 + 8 21 like this. So, this is the sequence of Fibonacci number, this is called step 0 Fibonacci number F 0 we denote F 1 like this F 2 F 3 so on. So, what is the formula is F n is basically zero if n is 0 this is the are in first Fibonacci number or the 0 Fibonacci number one if n is one and then after one we have the recursive formula like F_n =

$F_{n-1} + F_{n-2}$ {if n is greater than equal to 2}. So, this is the formula for n-th Fibonacci number. So, $F_n = F_{n-1} + F_{n-2}$. So, last few Fibonacci number will give us the n th Fibonacci number. So, our problem is to find the nth Fibonacci number where n is another input. So, we have to find 50th Fibonacci; 51 th Fibonacci number we want to find say 2017 th Fibonacci number like this.

So, this is the problem. So, the problem is to find the nth Fibonacci number. So, how to do that? So, that we can just see how we can solve this problem what is the algorithm will use for this. So, the problem is to find n th Fibonacci number.

(Refer Slide Time: 05:06)



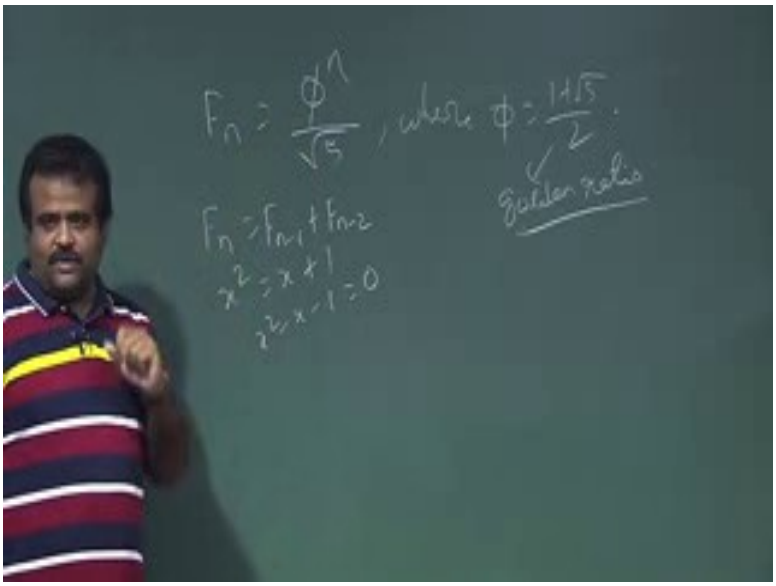
Find nth Fibonacci numbers nth Fibonacci number F n so; that means, we have to find F n where n is also an input. So, how to do that? So, what is the naive approach?

So, it is a bottom up method we start with the first Fibonacci number that is 0 I mean we just keep on calculating F 0 F 1 F 2 and every time we store the last Fibonacci number and then F n-1, F n-2, and then we got F n by adding this 2 F, F n -1 + F n -2. So, this is our nth Fibonacci number. So, this is the bottom up method we keep on calculating the Fibonacci number until we reach to the nth Fibonacci number. So, this is basically a bottom up way and. So, this each time we store the 2 last Fibonacci number to get the next Fibonacci number.

So, what is the time complexity for this? So, this is basically linear time algorithm because we are just calculating up to nth Fibonacci number like is bottom up way. So, this is the linear

time algorithms. So, now, we want to do we want to see whether we can have something better than this linear time whether we can do it in logarithm time like this, so for that we can use some formula which we know about the Fibonacci number. So, that formula is basically telling us the nth Fibonacci number F n can be written as phi to the power n by root 5.

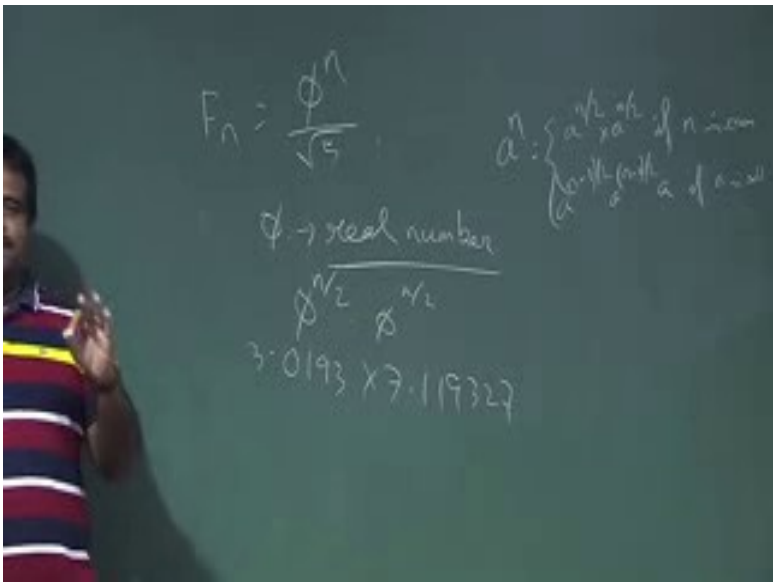
(Refer Slide Time: 07:08)



Where phi is basically 1+root(5/2), and this is basically what is called golden ratio this number is called golden ratio, where from this is coming? This we can prove by induction and this is coming basically by solving this recurrence. So, we know that $F_n = F_{n-1} + F_{n-2}$ now if we take this as x square this is basically x + 1. So, if we have this x*x - x - 1 = 0 now it has 2 roots. Root is “1 +/- root(5/2)”.

So, that is why this phi is coming. So, it can be shown that Fn = phi^(n/root(5)). Now we want to see how this formula can help us to have an algorithm, now this is also similar to powering a number, but here the number is not an integer. We have seen if we have integer like A to the power n we know this is a candidate of divide and conquer algorithm.

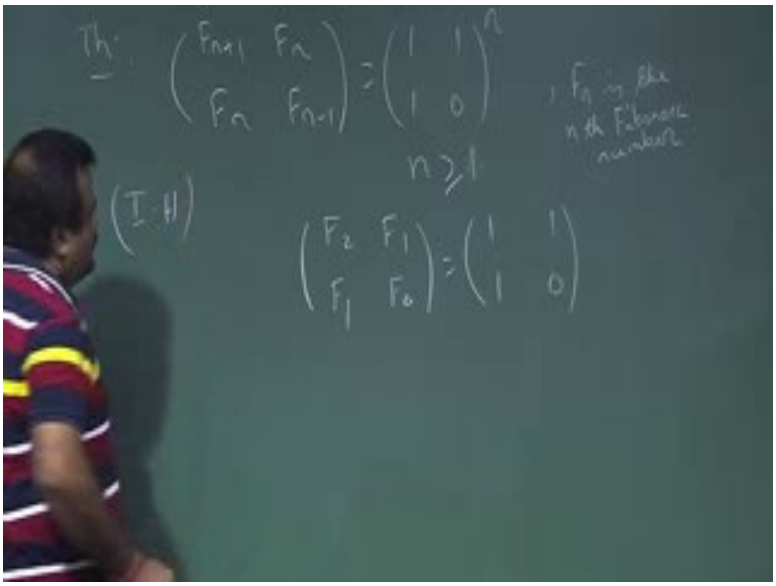
(Refer Slide Time: 08:27)



We can write this as a^n/2 into a^n/2, if n is even else we have seen this is a^(n-1)/2 into a^(n-1)/2, if n is odd this is formula. We know this is basically give us powering a number, but here a is basically a integer, but here also this is also similar to powering a number, but this phi is a real number. So, problem with handling real number when we find out powering a real number then the problem will be in the round off or in the precision. Suppose if we calculate this phi to the power n/2 phi to the power n/2, suppose this is a 3.01932 like this and this is say 7.119325.

So, depending on how much round off. So, this rounding and precision will take some time to fix. So, this algorithm is not as simple as powering a integer. So, that is why this will not give us a logarithm time algorithm because this fixing will take some time more time. So, this is the reason will try to avoid this powering a real number. So, this formula is not helping us. So, now, we try to see whether we can have another kind of formula which can help us to find out the nth Fibonacci number so that formula is basically will have a theorem on that.

(Refer Slide Time: 10:28)

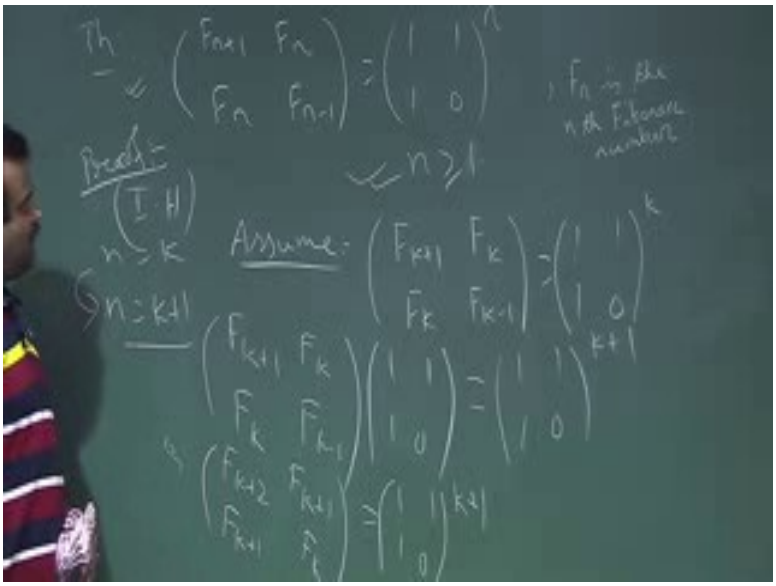


So, this is the theorem on the Fibonacci number this theorem is telling F(n+1), F(n), F(n), F(n-1) can be written as [1, 1, 1, 0] to the power n. So, this is the theorem where Fn is the nth Fibonacci number. So, it is start with F0. So, we know F0 is one. So, what is the base case? Base case means. So, this, 2 this should true for all n greater than equal to 0 this we have to prove. Now what is the base case? Base case is if we put n is equal to one. So, if we put n is equal to one.

So, this is true for n is greater than equal to one because if n is 0 then F_{-1} there is no F_{-1}. Now Fibonacci number is starting from F 0. So, the base case is we put a n is equal to 1 and we see whether it is true the result is true or not. So, this we'll see by the method of induction. So, for n is equal to 1 what is this matrix? This is we put just n is equal to one this is F 2 F 1 F 1, n is equal to 1 this is F 0.

So, this matrix is basically we know F 2 is 1 F 1 is 1 F 0 F 0. So, this is true for n is equal to 1. So, result is true for n is equal to 1. So, the base case is satisfied now we need to take the induction hypothesis. So, induction hypothesis test.

(Refer Slide Time: 12:28)



We assume that result is true for n is equal to k so; that means, we assume F_{k+1} (we put n is equal to k) F_k F_k F_{k-1} is equal to [1 1 1 0] to the power k. So, we assume the result is true for n is equal to k.

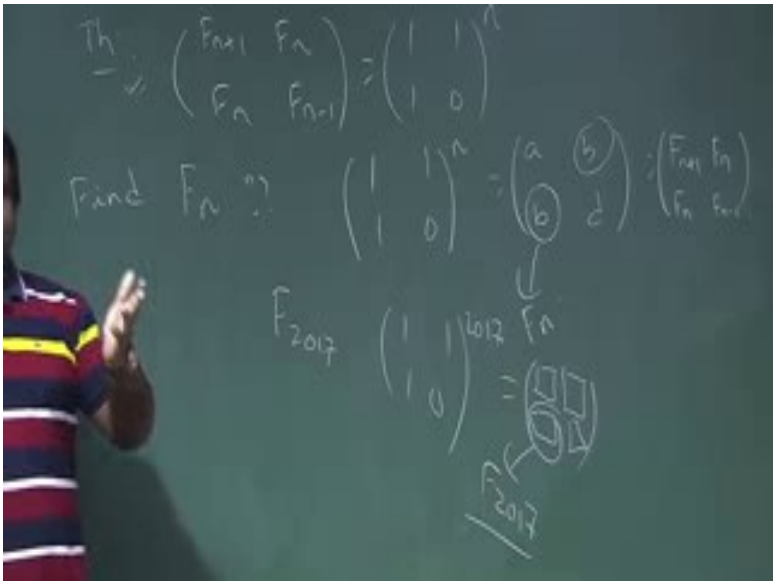
And now we need to prove that we need to show that the result is true for n is equal to k + 1, then it is done for we already put the base case based by the method of induction this is true for all n now how to prove this? So, this is our assumption induction hypothesis now we multiply both side by this right hand side matrix [1 1 1 0]. So, F_{k+1}, F_k F_k F_{k-1} 1 1 1 0 we multiply and then it will be 1 1 1 0 to the power k + 1 ok.

So, now what is this matrix? If we multiply this with this matrix, so this will be basically. So, this with this, this will give us F of k + 2 and so, this is basically F of k + 1 and here F of k + 1 and F of k. So, this is basically this into this F of k. So, this is to be 1 1 1 0 to the power k + 1 so; that means, the result is true for n is equal to k + 1. So, we assume the result is true for n is equal to k. So, from there we can show that the result is true for n is equal to k + 1. So, on we have already prove the base case.

So that means, by the method of induction we can say the result is true for all n greater than equal to 1. So, this theorem is proved. So, this is the, prove of this theorem by the help of mathematical induction. So, this theorem is true. So, if we have this theorem how this theorem will help us to have a algorithm. So, that we'll see. So, for that we just. So, this is a.

So, now, our goal is to find F n we need to. So, this is our goal we need to find F n what is F n ok.

(Refer Slide Time: 15:04)

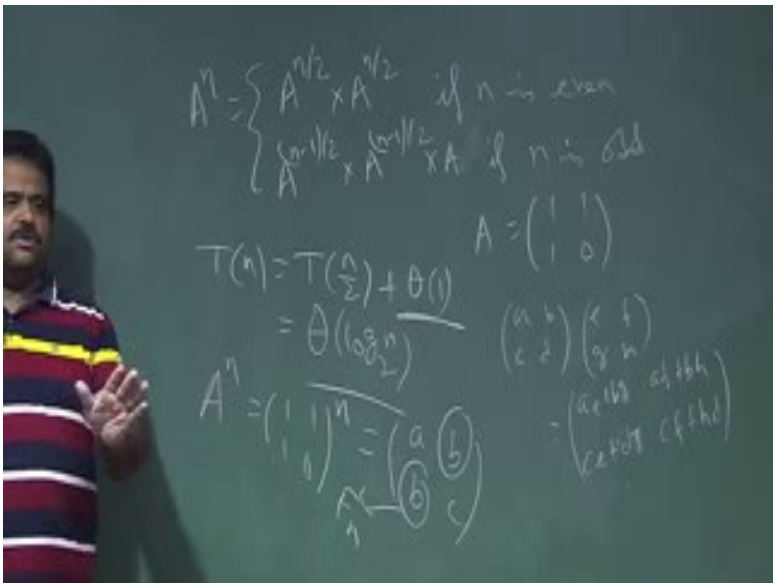


Now, to find F n we have this theorem now if we can have this to the power n, then if we can have this matrix result. So, this is a b c d then our F n is this is basically this will be same matrix, because this is basically by this theorem F n + 1, F n, F n F n minus 1. So, from this base and we can take this as F n, so done, if you want to find F of say 2017. So, what we do we just need to get this 2017?

So, this will give us some matrix 4 by 4 matrix and this will be our F 2017. So, now, the problem is reduce to the problem of powering a matrix, but here the matrix is a 2 by 2 matrix not n by n matrix, we will see in the next problem is the matrix multiplication, but it is just a 2 by 2 matrix. So, it is very similar to powering a number. So, we can apply the divide and conquer technique to have this matrix to the power n powering a matrix how.

So, that is basically the same divide and conquer formula, which we have for powering a number

(Refer Slide Time: 16:51)



So, here we have a 2 by 2 matrix. So, we need to find out a to the power n, A to the power n can be written as A to the power n by 2 into A to the power n by 2, if n is even and if n is odd it is basically n minus 1 by 2 into A to the power n minus 1 by 2 into a if n is odd. So, now, this a is a just a 2 by 2 matrix 1 1 1 0 it is a 2 by 2 matrix. So, now, this is a formula for a divide and conquer approach now.

So, we have a problem of size n. So, we have to find A to the power n. So, we reduce the problem in size n by 2 now we need to find A to the power n by 2 now once we have the solution A to the power n by 2, once we have this matrix A to the power n by 2 that is conquer by conquer step then we multiply by itself to get A to the power n if it is even. Otherwise if it is odd we calculate this I mean we multiply this with again with itself. So, either 2 multiplication or 3 multiplication then what is the recurrence, recurrence is we reduce the problem into subproblems.

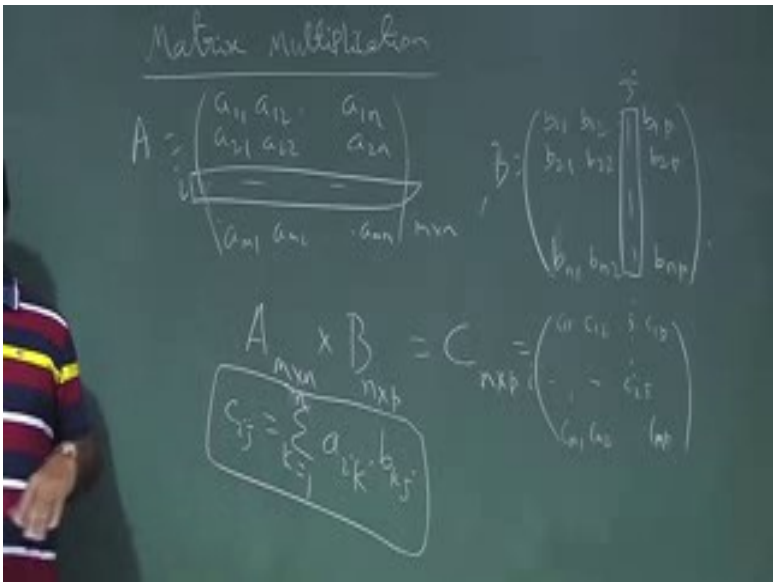
But we have only one sub problems and this is the cost for doing this multiplication either 2 multiplication or 3 multiplication, but all the multiplication on the matrix of size 2 by 2 because once we get A to the power n by 2 this will be again a matrix of 2 by 2. So, if you multiply 2 matrix of size 2 by 2. So, we are multiplying 2 matrix 2 by 2 matrix. So, say a b c d e f g h. So, what is the result? Result is basically this into this, a e + b g this into this a f + b h this into this c e + d g and this into this, c f + d h .

So, this is the result. So, that means. So, we are doing just this is the l number addition. So, we are doing 4 a 4 addition and this is l number multiplication and 8 multiplication. So, all

together will be some constant time. So, that is why it is theta(1) and if it is if odd we have to take another matrix. So, that is also in constant time because it is just either 4 5 or 4 addition and 8 multiplication, but real number multiplication. So, this will all take constant and so this is a theta of one time. So, this will be same as recurrence is same as powering a number.

So, this will give us by master method log n base 2 sorry n log n base 2. So, this is the n th Fibonacci number finding. So, once we got this A to the power n in this time once we got A to the power n a to the power n is [1 1, 1 0] to the power n, then this will be some 2 by 2 matrix. So, this is say some number like a b, b c now this quantity will give us F n. So, to find F n we just need to power this matrix this matrix is 2 by 2 matrix so, this will take logarithm time. So, this is the finding the nth Fibonacci number. So, next problem will deal with the matrix multiplication problem.

(Refer Slide Time: 21:01)



So, this is the next problem will talk about which can be solve using the divide and conquer technique this is called matrix multiplication problem. So, basically suppose we have 2 matrix A B. So, a is a matrix this is general matrix say a11 ,a12, a 21 n suppose this is a m cross n matrix a 2 1, a 2 2, a 2 n now a m 1, a m 2, a n m.

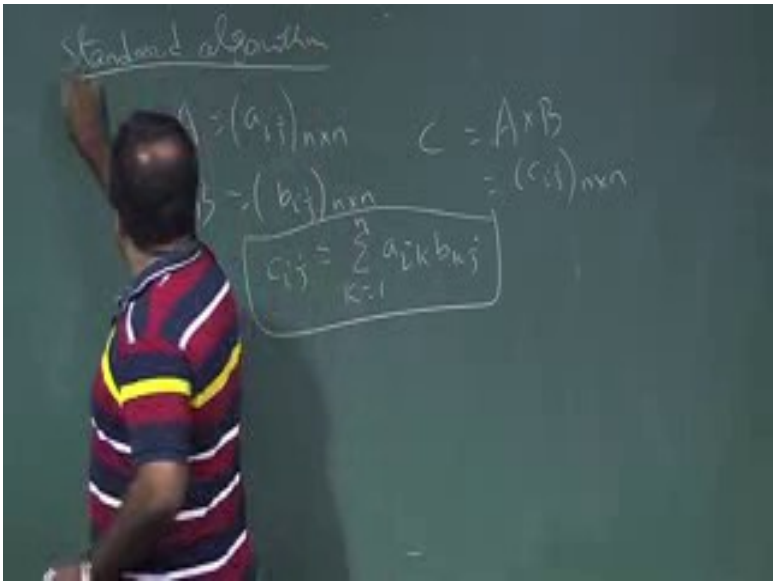
So, any matrix can be written as in this form this is if you want to implement this is 2 dimensional array of size m into n suppose we have a matrix a and we have another matrix b which is say of size say, to have 2 multiplication enhance if we have matrix say which is size m by n and another matrix is b which is size n by p, then we can multiply these. So, this will

give us a matrix C which is of size m cross p. So, what is the formula for that? So, this is basically if we have a matrix n by p.

So, this is basically b 1 1, b 1 2,…. b 1 p; b 2 1, b 2 2,..... b 2 p. So, b n 1 n row p columns b n 2 b n p suppose we have this matrix. So, then the, what is C matrix? C will be again the matrix of size m cross p. So, this is c 1 1, c 1 2,….. c 1 p; c m 1, c m 2,.... c m p and this is basically this is if this is i th row this is j th column and this is basically c i j and what is the formula for c i j? c i j is basically we take the i th row of this and we take the j th column. So, c i j is basically inner product of these two.

So, c i j small c i j is basically summation of a i k into b k j; this k is varying from the column 1 to m. So, this is the 1 to m cross n n cross n. So, this is one to n. So, this is the formula for c i j. So, this we know this is the definition of matrix multiplication. So, now, we want to have a, for this we want to know how we can have the algorithm to solve this matrix multiplication. So, here for the simplicity will take all the m n are same.

(Refer Slide Time: 24:39)



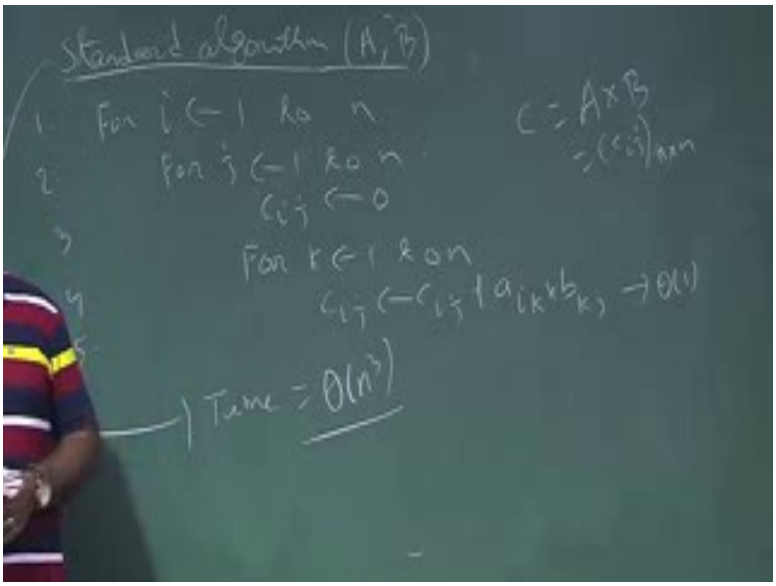
So that means, we take the order of the matrix both the matrix are n cross n and b matrix also n cross n. So, the c matrix will be also n cross n. So, basically we have 2 matrix of order n cross n and n cross n then we have to multiply.

So, this is A matrix a i j n cross n, and we have B matrix b i j n cross n and then C will be the A cross B this is the matrix multiplication. So, this is c i j this will be also n cross n. So, we

are just taking for the simplicity the matrix order a n cross n. So, now, c i j is basically summation of a i k, b k j; k = 1 to n. So, this is the formula. So, now, what is the code the standard algorithm to find this? So, this is the standard matrix multiplication algorithm to find this standard algorithm for matrix multiplication.

So, now how we can get this c i j? Basically we can write in a loop. So, basically we need to find c i j

(Refer Slide Time: 25:55)



So, we have 2 loops i is equal to 1 to j, and we have a j th loop j is equal to 1 to n and then we assign c i j to be 0 initially then we have another loop with k to find the c i j. So, this is basically for k = 1 to n. So, this is basically c i j = c i j + (a i k into b k j) that is it. So, this is the pseudo code for finding c i js. So, we have 2 matrix A B.

So, this is the standard algorithm for matrix multiplication. So, this will give us a c i j. So, this will give us c matrix which is basically A into B. So, this is basically c i j, n cross n because all matrix are all n cross n matrix. So, given two matrices this is the standard code for matrix multiplication. So, what is the time complexity basically we have 3 loops each of size n and this is what we are doing in constant time. So, basically the time for this is of the order of n cube.

So, this is the standard runtime for matrix multiplication. So, this is the naive approach now we want to see whether we can use some divide and conquer technique to handle this problem. So, that we'll do in the next class.

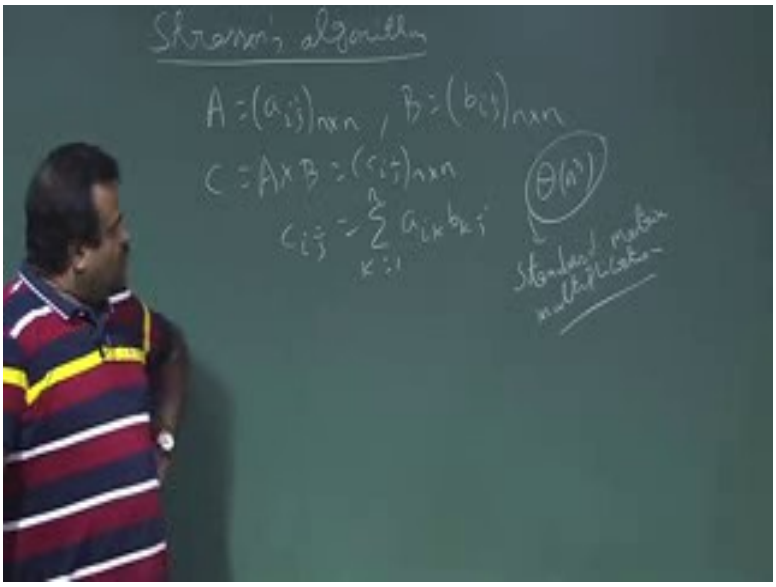
Thank you.

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

Lecture – 09
Strassen's Algorithm

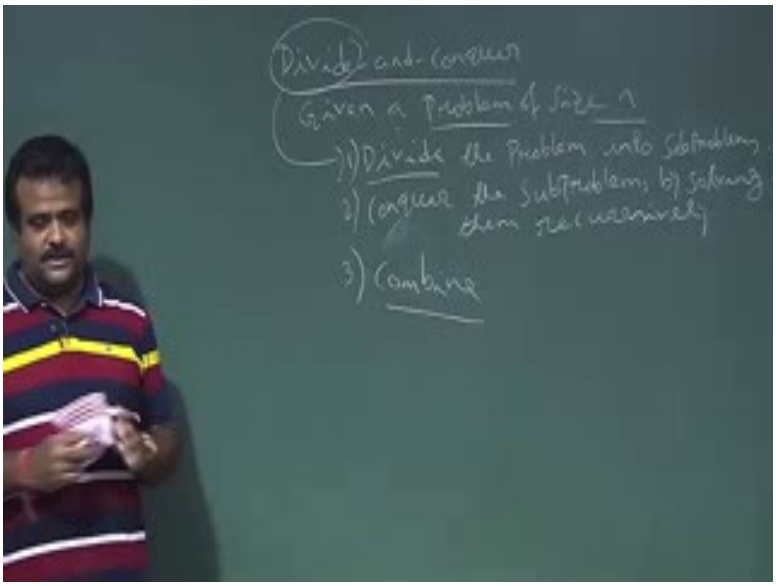
So we are talking about matrix multiplications. So, that we have seen using the standard method of matrix multiplication.

(Refer Slide Time: 00:31)



If we have 2 matrices A B which is basically size n cross n and B matrix is size n cross n and we have seen C is basically A cross B which is of size n cross n, and c i j is basically summation of a i k into b k j and we have seen the standard, using three for loops we can get this value, but that will be of the order of n cube this is by the standard matrix multiplication algorithm. So, now in this talk we want to see whether we can use some divide and conquer technique to reduce it further. I mean whether we can apply some divide and conquer technique to handle this problem. So, for that let us just look at divide and conquer technique means if we have a problem of size n it basically has three step.

(Refer Slide Time: 01:46)

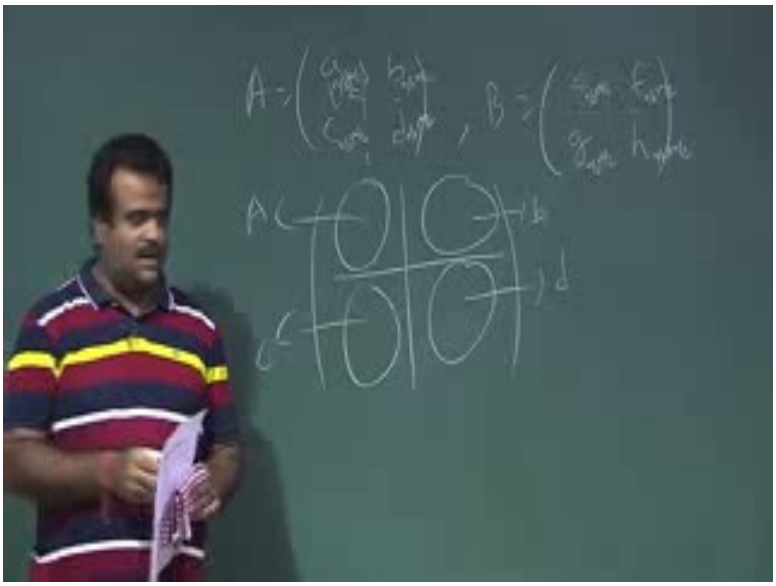


So, this is the just to recap. So, we have a problem of size n now in the first step i.e. divide step we divide the problem. This is the divide step, we divide the problem into lesser size into sub problems and subproblems size should be less than n it could be n/2, it could be n/3. Divide the problem into subproblems and then this is the first step of divide and conquer method then the conquer step we conquer the sub problems by solving them recursively.

We conquer the sub problems by solving them recursively. So, this is the recursive step; again we have a sub problems again we further divide into sub problems like this until the we reach to the size is 1, once the size is 1 you must stop because then it cannot be further reduce to sub problems and then we have a final step which is combine which is the basically for merge sort which is basically merge subroutine.

So, we had a solution for these 2 sub problems or 2 or 3 we have the solutions for the sub problems. Now this merge step will combine the solution of the sub problems to get the solution of the whole problem. So, this is basically the divide and conquer technique. So, now we will see how we can apply this technique for our matrix multiplication method. So, for that, just write the matrix into submatrices. So, we have 2 matrix A B.

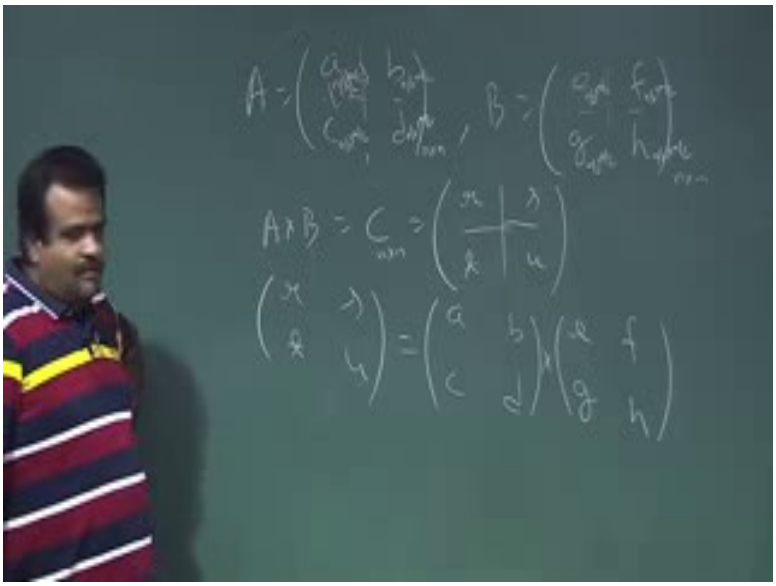
(Refer Slide Time: 04:03)



So, we write the matrix A as a b c d and B as say e f g h. So, these are basically sub matrices. So, these are all size n/2 cross n/2. So, a is n/2 cross n/2. So, b is also n/2 cross n/2. So, if we have big matrix what we do we just partition into 4 parts. So, this is basically A this is this matrix this sum matrix is B like this, this sum matrix is c and this sum matrix is d like this. So, these are basically sum matrices.

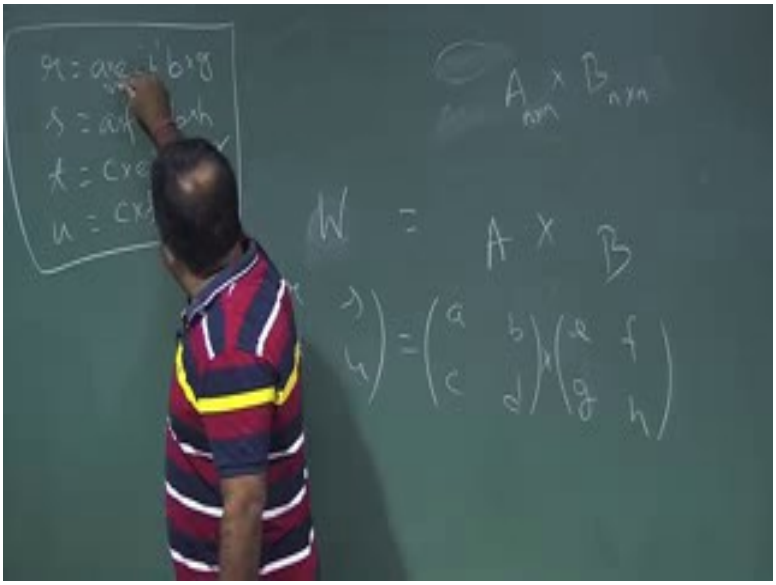
So, we have 4 parts w have dividing provided n is an even number or n n is multiple up to otherwise we have a lower ceiling or upper ceiling, but anyway this we will do the asymptotic analysis. So, we are allowed to do this slopyness. So, these are all sum matrices of size n/2 cross n/2. So, these are all n/2 cross n/2, these are all sum matrices similarly here we have n/2 cross n/2, we have n/2 cross n/2 this is also n/2 cross n/2, n/2 cross n/2. So, this is basically just given a matrix we divide into 4 parts.

(Refer Slide Time: 05:53)



So, that is the way we represent this a b c d and for b matrix e f g h. So, now, we denote this. So, we are multiplying a into b. So, we are multiplying A into B and A into B we where denote by C. So, this is also we are writing in 4 parts. So, like r s t u. So, this is n by n cross n by n. So, c will be also n by n cross n matrix. So, we again partition this into 4 parts. So, these are all n by 2 n by 2. So, basically what we have we have this matrix multiplication r s t u which is basically a b c d then this is e f g h. So this is the matrix multiplication and these are in the sum matrix form. So, now what is r? R is basically. So, we have to write this formula. So, what is r, r is basically r is getting by this into this. So, r is.

(Refer Slide Time: 07:01)



Basically a e + these are all matrix multiplication this is sum matrices + b g. So, these are these are all this is n b/2 matrix this n/2 this is also n/2 cross n/2 this is also n/2 cross n/2 this is also n/2 cross n/2 and this addition is. So, in matrix addition, we have 2 matrix of size n/2 cross n/2 we add it.

So, just addition of the elements corresponding elements. So, similarly we have s equal to s is basically this cross this. So, f + s is a f + b h. So, these are all matrix multiplication these are all matrix addition. So, T is basically this into this. So, c e + d g. So, this c is nothing to do our this A into B is equal to C. So, better we use this in different notation. So, do not confuse with this c and our matrix C they are different basically. So, basically we have 2 matrix A B, A this is a matrix this is B matrix and the multiplication is this matrix.

So, this is capital C. So, these two c and C are not same. So, if there is any confusion we can made it to some another matrix W. So, W is equal to A cross B if there is any confusion with this c. But anyway this is small c that is capital C, c into e + d into. So, this into, this into this d into h, so if T is basically these into this, c into e + b into g. So, and the last one is u, u is basically this into this c into f; c into c into f + d into h, d into h. So, this + are all matrix addition. So, this is the formula we have. So, matrix multiplication is basically.

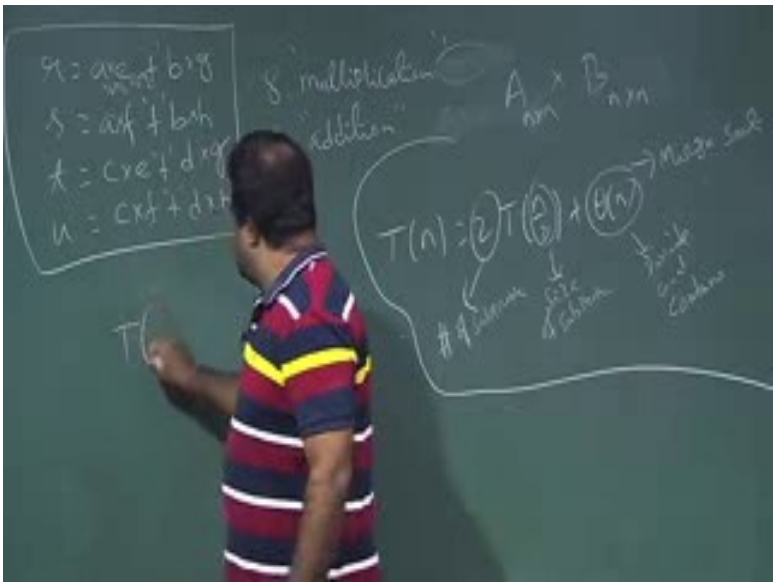
So, this is the divide and conquer step this is the divide step. So, now what is this now we have the matrix. So, we earlier we have to multiplication of matrix this cross. So, we have a A matrix we have a B matrix, and we have to multiply this is of size n cross n and this is also of size n cross n. Now here these are the all matrix multiplication, but these are all lesser size what is the size these are all n by 2 cross n by 2, these are all n by 2 cross n by 2 all the matrix multiplication. So, these are the sub matrices. So, these are all lesser size. So, this is the divide step.

We divide the problem into subproblems earlier our problem was to multiply 2 matrices of size n cross n into n cross n, now we reduce the problem into subproblems like now we have to multiply n/2 cross n/2 and another matrix is also n/2 cross n/2. So, that is the divide step and now in conquer step we must get the solution of this and we must get the solution of this. Once we have this solution of these 2 then we have to combine by adding this to get s so, that is the divide and conquer approach is this.

So, basically we have a we have 2 matrices we reduce the matrix into sum matrices and then we reduce the problem into subproblems and then conquer step we again call the same

formula, now this is of size n cross lesser size again we part the divide into sub matrices like this and again once we have a solution for this once we have solution for one; once we have the result, we add this and that is a part of the combine step. So, how many matrix multiplication we are doing basically we are doing 8 multiplications .

(Refer Slide Time: 11:44)

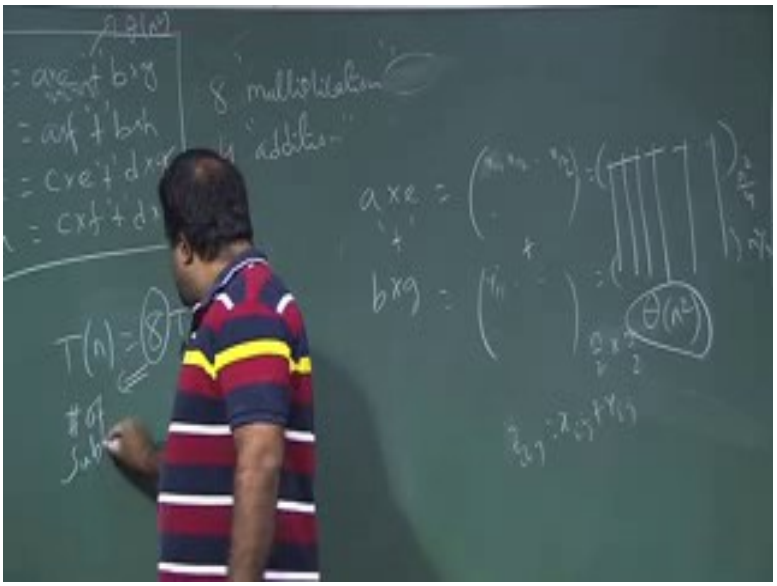


And how many additions we are doing? We are doing 4 additions; i.e. 4 matrix additions. These are all matrix additions and these are all matrix multiplication of lesser size. So, now, what is the recurrence we have? So, basically every divide and conquer technique have the recurrence like T n is equal to something. So, for merge sort we have if you remember for merge sort we have the recurrence T(n) = 2*T(n/2) + theta(n).

So, we have a problem of size n we reduce into the sub problems and these are the number of sub problems. This is the merge sort recurrence this is the number of sub problems, and this is the size of subproblems size is reduced to half. I mean we have an array we divide into 2 part half half, this is the size of sub problems and this is the cost for both divide and combine.

So, now if we use this divide and conquer technique of matrix multiplication. So, then what is the recurrence we have? So, we have T n is basically how many sub problems. So, our problem is basically matrix multiplication.

(Refer Slide Time: 13:25)



So, we reduce this into sub matrices. So, how many sub problems. So, now, here 8 sub problems because 8 matrix sub multiplication is required on the lesser size. So, that will be done in the conquer step. So, that is 8 and this n/2 is the size of the sub problems. Now our problem was n cross n matrix multiplication now it is n/2 cross n/2 and both matrices sub matrices.

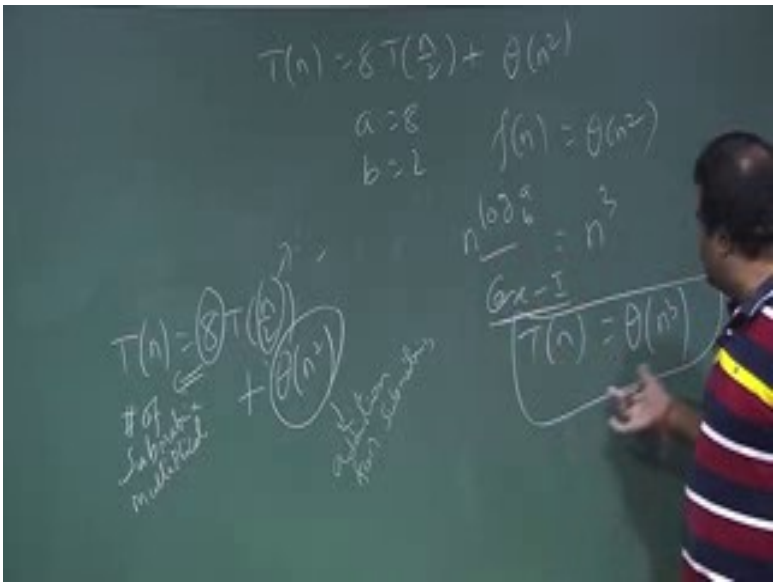
So, size is reduced to half. So, that is the size of the sub problem. So, this is basically n/2 + the cost for combine. So, combine means this additions, this matrix addition. So, how much time it will take to add 2 matrices. So, that is the point. So, that is the time to take the addition. So, how much time it will take? So, 2 matrix addition. So, what is the size of this matrix? So, once we get the result by the conquer step. So, this will be a cross e will be a matrix of size n/2 + n/2.

And then b cross g this is also of size n/2 cross n/2; now this addition means we are just adding; this is added with this like this. So, if this is the matrix like x 1 1, x 1 2, x 1 n/2 like this and this is y 1 1 like this. So, this basically addition is y i. So, addition will be z i j, z i j is basically x i j + y i j. So, we are adding just adding the corresponding entity; matrix addition. So, just we take this value adding with this value adding with this like this even you can think this as a vector.

So, just first row then second row third row like this. So, this is a vector of size n*n/4, and then this is also vector of n*n/4. So, we are just adding 2 vectors like this. So, if we have 2

vectors then it will be this. So, how many additions? We have basically n*n/4 additions. So, this addition will take order of n square. So, each addition will take order of n square. So, how many addition we have each of this addition will take order of n square. So, we have basically 4 such additions. So, if we have 4 additions, this will take order of n square.

(Refer Slide Time: 16:55)



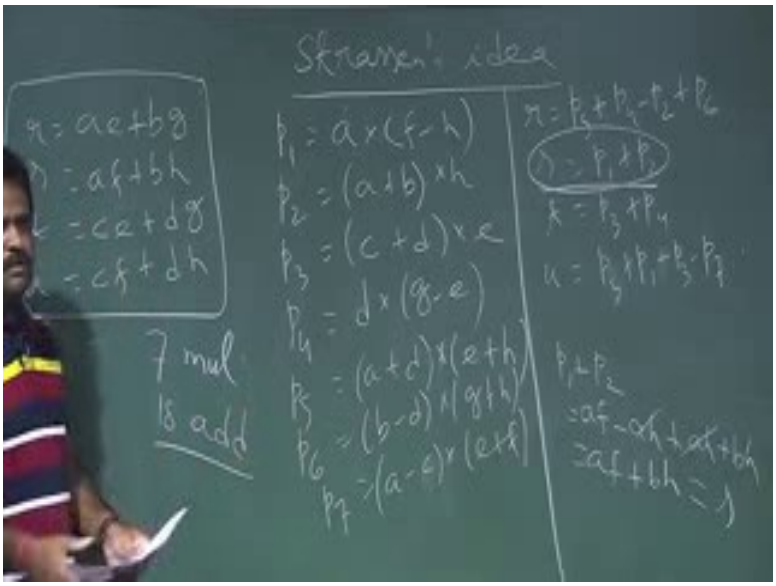
So, this is the cost. So, this is basically number of sub matrices, this is the recurrence for matrix multiplication using this divide and conquer technique this is the number of sub matrices basically we are having 8 sub matrix multiplication, number of sub matrix multiplication, and this is the size of the sum matrices, sub matrix size is reduced to n/2 cross n/2 size of sub matrix and this is the cost for combining the addition for addition for sub matrices.

So, this is the recurrence we have for this divide and conquer techniques. So, now, we want to know the solution for this recurrence. So, we have this recurrence what is the recurrence T(n) = 8*T(n/2) + theta(n*n). So, what is a? We want to fit it into master method. So, a is 4, b is 2 and f n is n square. So, which case we are in? So, $n^{\log_b a}$ this is basically n^3. So, we are in case 1. So, what is the solution? So, solution is this term is dominating.

So, this is asymptotically polynomially faster than this one. So, this is basically order of n^3. So, this divide and conquer technique is giving us order of n cube time algorithm which is not better as the standard method or matrix multiplication because standard method is also giving us order of n cube. So, then the point is how we can modify it further? So, that is the idea of

Strassen's. So, Strassen's gives the idea of how we can modify it further like because this direct divide and conquer technique will give us order of n cube algorithm, which is same as the standard matrix multiplication algorithm. So, the idea of strassen is like this. So, instead of this recurrence suppose.

(Refer Slide Time: 19:39)



So, this is the recurrence coming from this divide and conquer techniques suppose instead of 8 multiplication if we can reduce a multiplication by 1 instead of 8 multiplication, if we can reduce it to 7 with the cost of more addition, because addition anyway will take order of n square, so here we are doing 8 addition what is the formula? Formula is our r is basically a e + b g our s is a f + b h our t is basically c e + d g and our u is basically c f + d h. So, this is r s p u now here we are doing 8 multiplication.

So, the Strassen's idea is if we could reduce 8 multiplication; instead of 8 multiplication if we could reduce it to 7 with the cost of more addition here we are doing 4 addition, but if we could do more addition that will be captured under order of n square. So, that will not affect much. So, that is the idea. So, that is why we want to make it that Strassen's idea is to make it 7 matrix multiplications. So we are going to write how we can do that. So, that is the idea by Strassen's. Strassen's idea to reduce the multiplication by 1 with the cost of more additions.

So, for that Strassen is calculating some intermediate terms like p 1 a into f - h, p 2 into h, p 3 is basically c d into e. So, these are the term we are calculating these are intermediate terms. So, there will be 7 terms so; that means, there will be 7 matrix multiplications, but we will

have more addition. So, we will talk about that. So, d into g - e and p 5 is basically a + d into e + h.

And p 6 is basically (b-d) into (g+h), and the last term is p 7; which is (a - c) into e + f). So, these are the terms Strassen's is calculating these are the intermediate terms. So, there are basically 7 terms so; that means 7 multiplications of lesser size. So, once we get this value how to get r s t u. So, that formula we have to write then we can see r is basically p 5 + p 4 - p 2 + p 6 and s is basically p 1 + p 2, t is basically; we can verify this p 3 + p 4, and u is basically p 5 + p 1 - p 3 - p 7. So, with the help of this pi's we can get this. So, how to verify this we can take one example like this one remaining you can check by yourself, p 1 + p 3, p 1 + p 2. So, p 1 is basically this into this. So, p 1 is a f - a h. Now what is p 2? p 2 is h + b h. So, h + b h, this 2 is canceling a f + b h what is a f + b h a f + b h is basically s.

So, this is verified s = p 1 + p 2. So, remaining term also you can easily verify just putting the value of p's. So, this is the formula, this idea is given by Strassen. So, basically instead of 8 multiplication we have 7 multiplication see there are 7 terms and each term is having one multiplication, but with the cost of more addition. This subtraction is also addition. So, how many addition? Subtraction is also addition 1 2 3 4 5 6 7 8 9 10 then 11, 12, 13, 14, 15, 16, 17, 18 with 18 addition.

So, we have we have 7 multiplication and we have 18 addition, that is because addition can be more because addition will take one addition will take order of n square. So, this will be captured in theta of n square, what is the recurrence of this method. So, recurrence will be we have 7 multiplication.

(Refer Slide Time: 24:56)

So, that is the good news. So, T(n) = 7*T(n/2) + (order of n square), this is the recurrence we have for Strassen's multiplication method; here we have more addition.

But that will be taken care by order of n square and instead of 8 matrix multiplication we have 7 matrix multiplication. So, then what is the solution what is the solution for this. So, here a is 7, b is 2 f n order of n square. So, what is $n^{\log_b a}$ it is basically $n^{\log_2 7}$, it is basically $n^{2.81}$. So, now this is also in case 1 and the solution for this is basically $T(n) = n^{2.81}$.

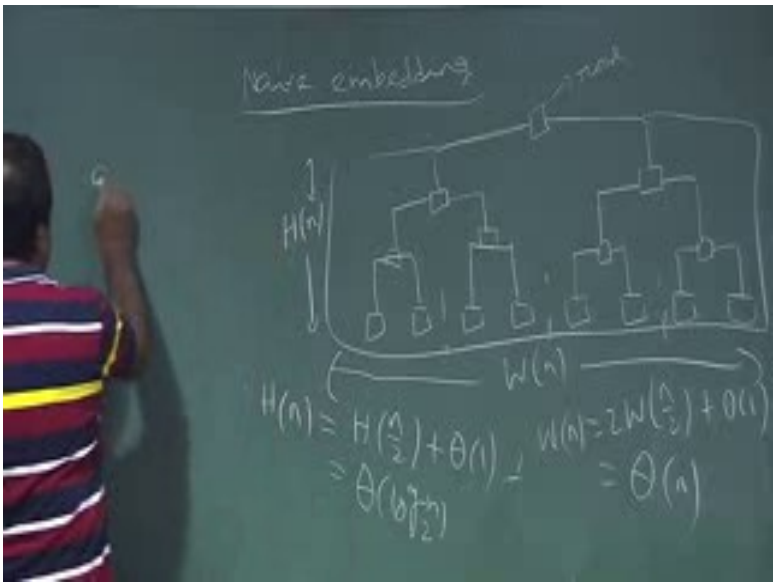
So, this is the time complexity for Strassen's matrix multiplication. So, this is much more better than n cube which was the standard method or just the divide and conquer technique. So, this is also divide and conquer technique, but we could reduce the matrix multiplication by instead of 8 we could reduce to 7. So, this is a good gain in terms of this. So, this is the Strassen's idea. So, next problem we will talk about VLSI layout which is the another problem which can be used which in which we can use the divide and conquer technique.

(Refer Slide Time: 26:54)

So, this is the different problem, this is this is not really algorithm problem this is called how much time 5 minutes VLSI layout. So, the problem is that we have a chip; rectangular chip and we want to embed a complete binary tree, with n leaves in a grid using minimum area that is the problem using minimum area. So, we have a rectangular chip. So, in which we have to embed a tree complete binary tree like this.

So, this is the tree. So, may be once more. So, this tree we have to embed in a chip a rectangular chip. So, we have the rectangular chip in a grid so that we should take the minimum area. So, area means this is the length this is the width. So, the total size will be less.

(Refer Slide Time: 28:46)

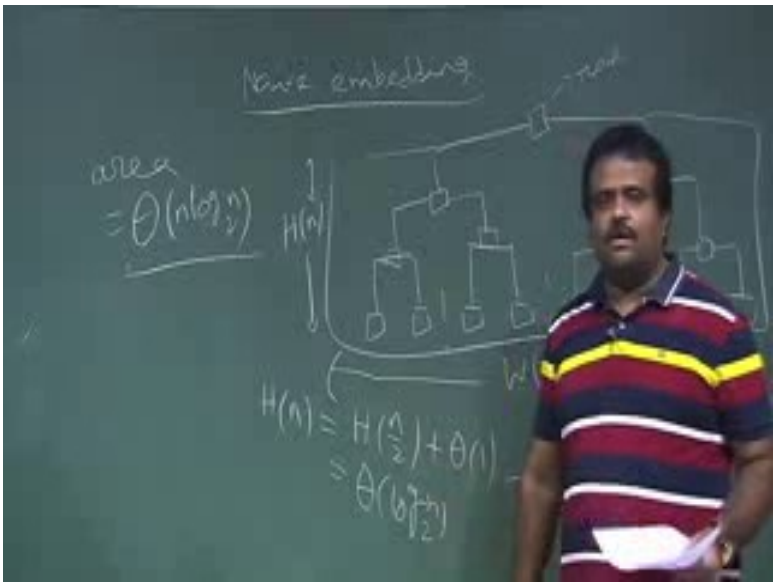


So, this we can do it like this. So, we can just start at this is the naive approach. So, what is the naive approach? So, this is the problem, the naive embedding. So, we start embedding this.

So, this is giving us like this like this then again we take this again we take this like this. So, this is the naive embedding. So, we just start with this leaf node then we slowly this is the root now this is our, this is our chip now what is the length of this how to calculate the area. So, if we denote this by the height of the tree, and if we denote by this width of the w n, now for height what is the recurrence height is basically.

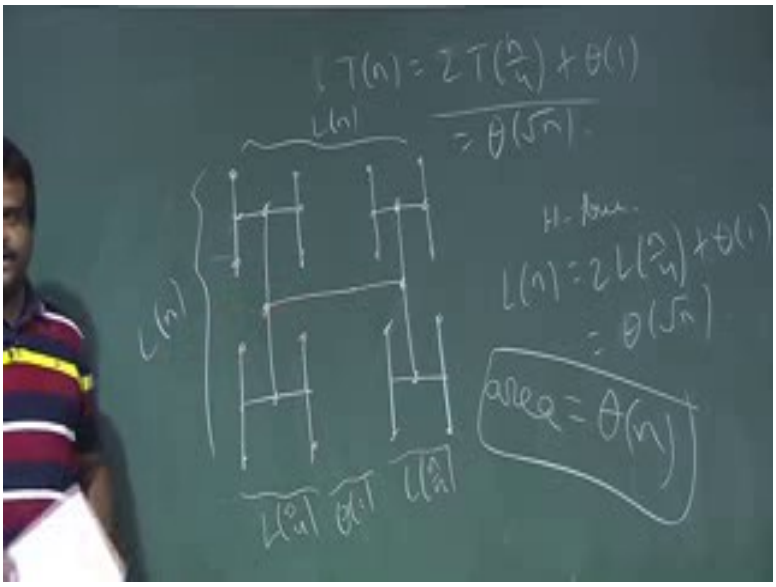
So, we have to embed n nodes. So, to embed n nodes we just. So, these are the running in these are in parallel. So, height for recurrence for height is T n h n equal to. So, h n/2 + theta 1 because these 2 are in parallel, but for width it is not parallel. So, this will count. So, basically for width we have w n is equal to 2 of w n/2 + theta 1. So, if we again use the master method this will give us a solution like log n, but this will give us a solution like n. So, the area is basically.

(Refer Slide Time: 31:05)



So, the area is height into width. So, area is basically order of n log n. So, this is the log 1 now do you want to make it in a order of n times n we want to do it in order of n. So, that should be the area we want. So, for that we will do something which is called h embedding h tree embedding. So, how to get order of n? To get order of n height and width should be order of root n.

(Refer Slide Time: 31:43)



So, how to get root n which recurrence give us the root n? So, if we have a T n is like this T(n) = 2*T(n/4) + theta(1).

So, this type of recurrence will give us root n if we can so, that the height of both are coming like this recurrence then you are done. So, for that you will have h type embedding. So, what is that will do like this. So, these are all h this called h tree. So, we are drawing the, this thing and another one here another one h-tree here. So, these are all the connections. So, these are all h tree, then we have this one.

So, these are all H tree H structure. So, if we have this then this is basically L of n and this is basically this is also same this is also L of n. So, this is we have L of n/4 and this is theta of 1 and this is again L of n/4. So, we have a recurrence for L is like this 2*L(n/4) + theta of 1. So, this will give us a solution theta of n.

So, this is the one side area. So, L(n) and the area is basically root over of h this. So, this is the using the help of h type embedding. So, this is one application of divide and conquer technique totally in different area. So, this is called VLSI layout, we want to fit a complete binary tree connection. So, that is the idea.

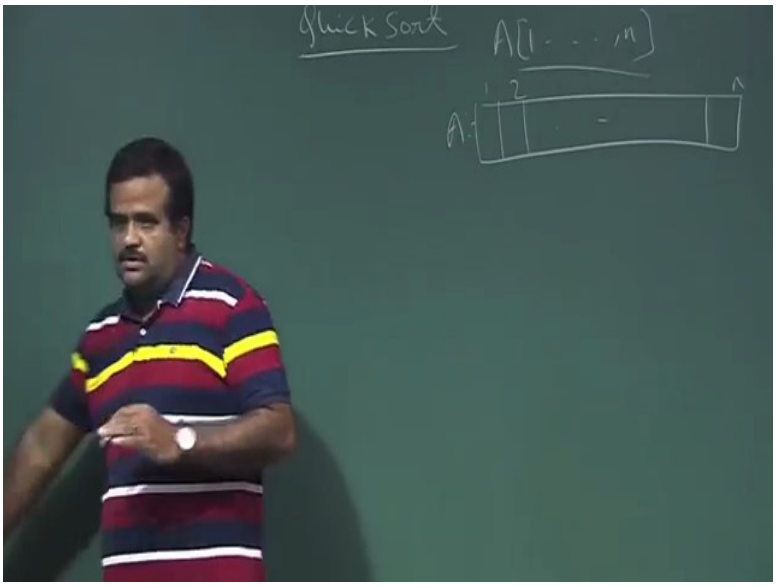
Thank you.

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

Lecture –10
Quick sort

So we'll talk about example of another divide and conquer technique which is basically quick sort.

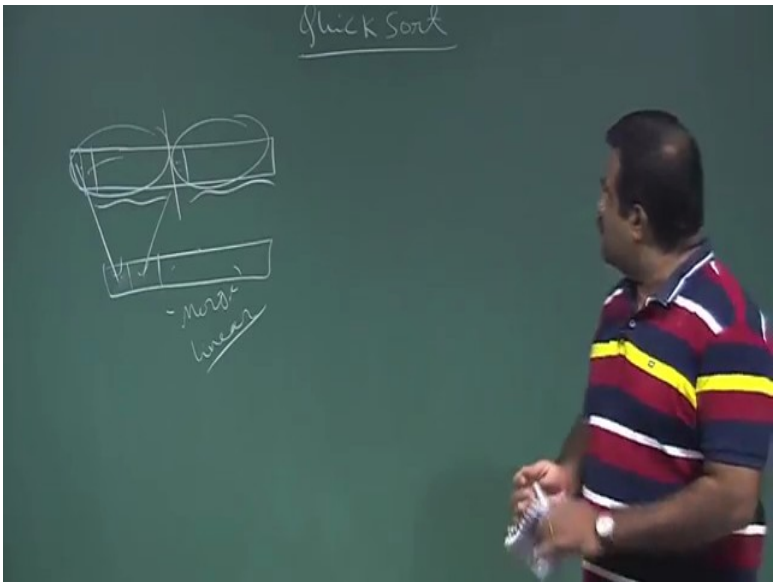
(Refer Slide Time: 00:33)



So, it is basically sorting algorithm where we need to sort an array of yes this is the input we are given n numbers, the numbers are in array what could be any other form now we need to sort this number. So, this is the sorting algorithm and we have seen. So, far 2 sorting algorithm like insertion sort merge sort.

So, merge sort is one of the one of the example of divide and conquer technique now quick sort is another divide and another example of divide and conquer technique. So, it has basically 3 steps. So, every divide and conquer technique has 3 step, we divide we have given a problem of size n we divide the problem into sub problems of lesser size that is the divide step and in the conquer step we conquered the sub problem by recursively solving them, and then once we have the we got the solution of this sub problems then we combine the solution to get the solution of the whole problem so, that is the divide and conquer technique.

(Refer Slide Time: 01:43)

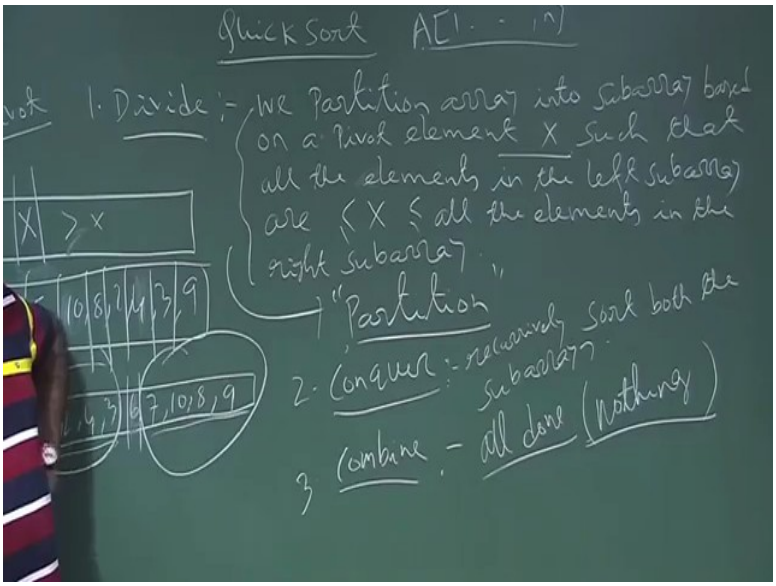


So, in merge sort what we have we did in merge sort we have a array of size n we need to sort we divide into 2 part half half, that is the divide step. So, merge sort divide step is very simple and the conquer step we sort this sub array we sort this sub array by recursively calling the same merge sort, then once we have the solution of these 2 sub arrays then this is sorted and this is sorted then we call the merge subroutine. So, this will take a array extra observable array. So, it compares for the minimum.

Whichever is the minimum it will output that next minimum like this. So, this is the merge subroutine. So, that is the combine step in the merge sort. So, merge sort combine step is crucial than the divide step. Divide step is very straight forward we are going to the middle of the array we divide the array into 2 parts that is the divide step, but combine step we have a merge subroutine and that is of linear time and for that we need to have a extra storage. So, that is why merge sort is not an inplace sort for merge sort we need an extra storage.

So, now we talk about quick sort which is an inplace sorting algorithm, and which is also divide and conquer technique.

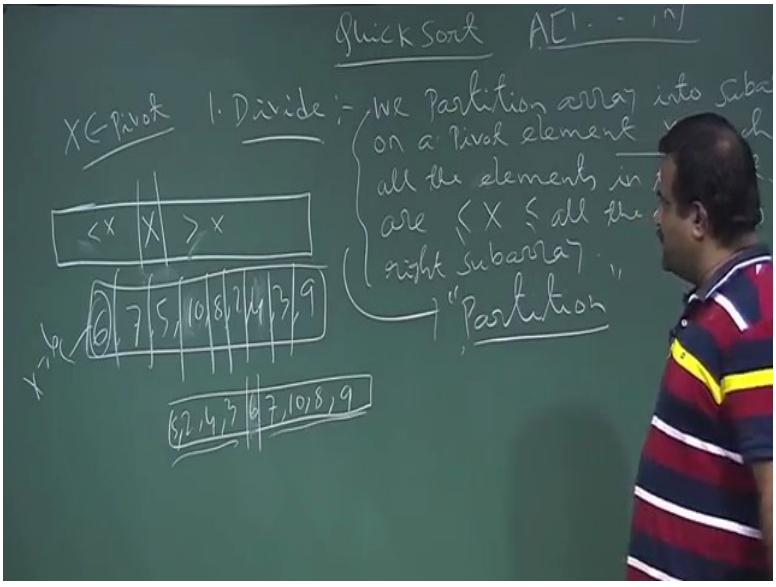
(Refer Slide Time: 03:01)



So, in the divide step what we are doing? Basically we partition the array, we have given an array of size n initially we partition the array into 2 sub arrays based on a pivot element x, x we called an element among this anyone could be a pivot, but for our partition this is done by a subroutine which is called partition.

So, based on a pivot element x which is basically an element from this array such that all the elements in the left sub array are less than x or less than equal to x and then all the elements are in the right sub array are greater than x.

(Refer Slide Time: 04:48)



So, basically we have this is our given array. So, what we are doing and this is done by a subroutine which is called partition.

So basically, we choose among this element we choose one element as a pivot say x is pivot element; for our partition algorithm we take first element as a pivot, but pivot could be any of these any of the element from the given array. So, we choose this pivot now in this divide step what we are doing this partition algorithm, partition sub routine it is partitioning the array into 2 part it is putting the pivot in its correct position correct position in the sense. So, if we sort this array the position of x it should be in the sorted array that is called correct position.

So, x will be sitting in this correct position and all the elements in the left sub array will be less than x, if there are repetition will less than equal to x solve the element and get if there are all the elements are distinct and this will be strictly less than x and this will be greater than x. So, this is what we want in our this divide step. So, they need not be sorted suppose for example, if we have a array like this 6 7 5 8 I say 10 8 2 2 4 3 9 suppose this is our given array this is the input.

This is our given array say now suppose we call the partition algorithm on this array by choosing this 6 as the pivot. So, our x is 6; 6 as a pivot now we want this divide step should give us. So, it should put the 6 over somewhere here. So, this is 6 and it should partition this array into 2 part all the elements less than 6 would be in the left sub array. So, who are the less than 6. So, 2 4 3 should be in the left sub array and the all the element greater than 6 will be in the right sub array.

Who are the greater sorry 2 5 is also there. So, 5 2 4 6 these are the element less than 6 and 7 10 8 9 how many there 1 2 3 4 5 6 7 8 9, 1 2 3 4 5 6 7 8 9 yeah. So, this is the output after the partition call on this array. So, you have only this if you see this is this is not sorted, but only thing 6 is in correct position. So, you put the 6 in the correct position; that means, if you sort this array where this pivot should be x would be x is in correct position. So, this is the sub routine partition will do.

So, it will just divide the array into two parts the left sub array all the elements in the left sub array will be less than x and all the elements in the right sub array will be greater than x. So, this is the job of the partition algorithm and this is the divide step. Now what is the conquer

step then. So, now we have 2 sub arrays now conquer step is what now we have to sort this sub array by recursively calling the same quick sort. So, we recursively sort both the array.

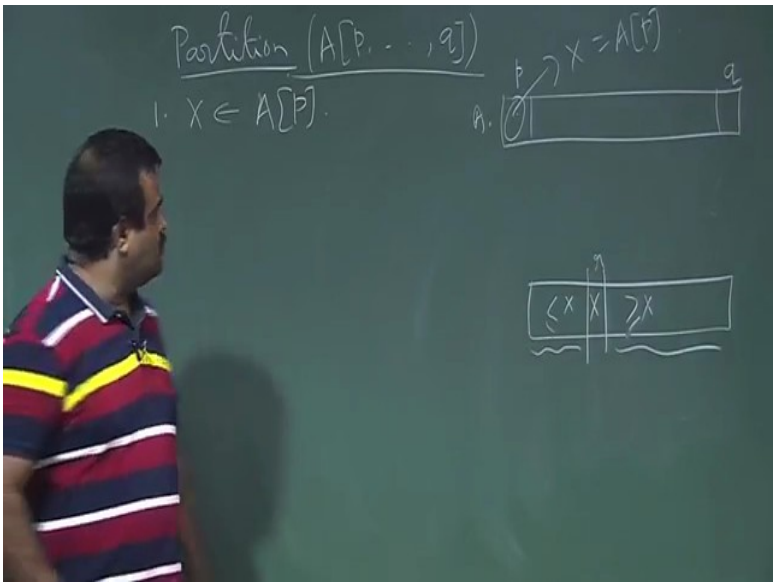
Both the sub arrays this is the conquer step; that means, now again we need to call this quick sort here then in that quick sort call again we need to choose this as a pivot if we choose first element as a pivot then again it will partition into sub array like this. So, you continue and then what is the combine step. So, in conquer step once we call the conquer step, then this after the completion of the conquer step this is sorted array this is sorted sub array and 6 is in correct position.

So, what is the job remaining what is the combine step what need to combined? Nothing has left all done all done. So, combine step is nothing we are done all done it is nothing has left. So, that is the combine step. So, we have we have this combine step is trivial nothing has left. Now the idea is to have a partitions of routine in linear time that is the key idea of the quick sort we need to have a partition algorithm in linear time. So, we will we will discuss that in the in a moment that what is the partition code what is the pseudo code for partition which will give us the runtime linear.

So, that is the key point of the quick sort, and if we just see the quick sort this if you compare the quick sort with merge sort, merge sort the divide step was trivial we just go into the middle of the array and we divide into 2 part and conquer step we sort this part we sort this part, and the combine step was the merge sub routine. So, combine step was more than the divide step in the merge sort, but here divide step is more than the combine, x combine is nothing in the quick sort. So, this is the divide and conquer technique for quick sort now we talk about the partition algorithm.

So, this partition sub routine how this array will be divided into 2 sub arrays such that all the elements which are less than x will go to the left sub array, all the element which are greater than x will go to the right sub array. So, that should be done in linear time. So, that is the partition code. So, let us just write the partition code; pseudo code for partition.

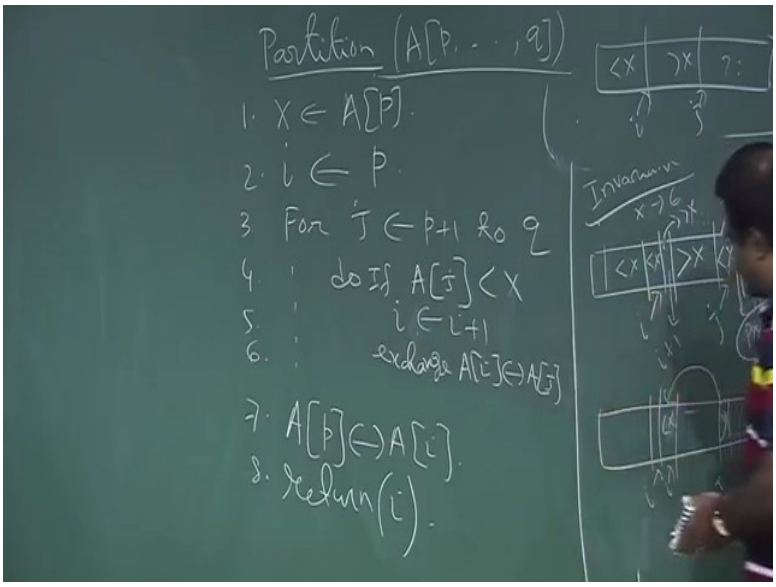
(Refer Slide Time: 11:44)



So, it is basically. So, partition which is taking an array which is p to q. So, initially it will take 1 to 8, but again all the sub array we are calling the partition.

So, it will be index would not be one and throughout the subsequent call. So, index will change. So, that is why we taking the general index. So, initially p is 1 q is n. So, we have an array which is A which is starting from p to q this is an input of this partition.

(Refer Slide Time: 12:24)

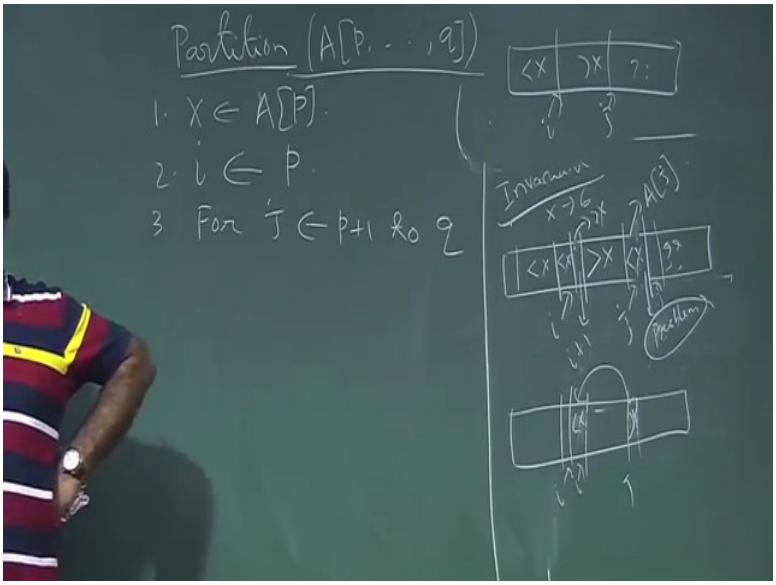


So, for our version of the partition we are taking first element as a pivot. So, A p. So, whatever the value over here this we are taking as pivot and what we want from this? We

want at the end of this code it should partition this array into 2 parts such that x will be putting somewhere here and this index is r, such that all the elements over here will be less than x all the elements here will be greater than x if they are distinct otherwise if there are some equal you can use this.

So, this is what we want for our partition algorithm. So, let us write the code. So, we start with 2 index i is p and we have a for loop for J is equal to p plus 1 to q. So, J is equal to p plus 1 to q and so, this is the loop now if we to write a code it will help us if we know what is the loop invariant we are looking for in this, then it will help us to write a code every algorithm every pseudo code if we know what we are looking for then it will help us to write the code.

(Refer Slide Time: 13:40)



So, what is the loop invariant here we are looking for? So, we are looking for this; this is the loop invariant. So, i is pointing somewhere here and J is pointing somewhere here now what do you want the loop invariant is all the elements here would be less than x less than x, less than equal to or less than if there are identical element.

And all the elements here will be greater than x, this we want all the elements here should be greater than x and this is yet to decide. So, that is the invariant. So, what is the initial step initial step is initially i is pointing here and J is pointing here and initially everything is yet to decide. So, that is the initial condition nothing is decided yet. So, that has to decide that is the initial step, but at some point of time this will be the situation. So, we take this. So, now, if

this one J is varying, now if this is our A J now if A J is greater than x suppose A J is greater than x then you can increase J by 1 and that will be taking care by this for loop this is the for loop in j.

So, it automatically increases if you do not do anything. So, if A J if this is greater than x then cool, no need to do anything here. So, it will be just automatically increase J by 1 now the problem is if this guy if this is less than x if this is less than x, say suppose x is 6 and suppose this is 4 for example,. So, this is less than x now we cannot just increase J by 1. So, you need to do something. So, what we need to do that we have to think. So, we cannot just increase J. So, this is our A j. So, if this then we have a problem then we have a problem then that problem how we can tackle. So, then we cannot just increase J by 1 we cannot just increase J to this point by this for loop in that case we are not evaluating our invariant. So, for that what we want? We know we want to exchange this with somebody which is greater than x. So, what we know, we know the element which is sitting here that i plus 1 this is i plus 1 we know this element which is sitting over here here is greater than x, now if we just in exchange this and this.

And if we increase i by i+1 then we have through then also we have this this was i now the which now this is J, now this is less than x and this is greater than x we know now if we increase i by 1 and if we exchange this guy with this now this x is coming here and now this x is greater than x and now this guy is greater than x and this is less than x then we have through then still the loop invariant is there loop invariant is the point where i up to i. So, this is the loop invariant we want so.

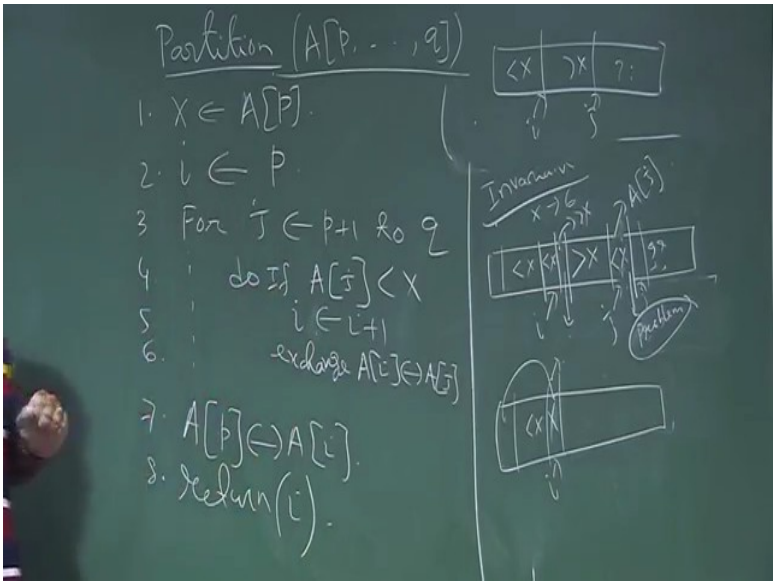
So, if this is the i then all the elements here is less than x if this is J all the elements here will be greater than x this is the way we will handle this situation. If the A J is less than x if A J is greater than x no need to do anything. So, that we have to do if A J is less than x, if the A J is less than x then we have to do this what we do we increase i by 1. So, i will be i plus 1 we increase i by 1 now the A i is some element which is greater than x now we exchange J i with A J now we exchange A i with A J that is it now we exchange A i with A J exchange.

So, if we exchange A i with A J then peacefully we can increase J by this that will be taken care by this for loop. So, this is 5 this is 6 and then. So, this is the loop and this will do for until the J exhaust and then finally, we have 7. So, after exhausting this loop what we do we will put the pivot element in its correct position. So, that will be done. So, J exhaust now i

will be pointing somewhere. So, now, where i is pointing that guy is less than equal to x. So, now, we exchange A p with A i; A p with A i and we return A i. So, that is the, that will give us the position where we have put x.

So, J will exhaust. So, J is gone and now i is pointing somewhere here and all the elements over here is less than equal to x. So, now, J is gone J loop is exhausted.

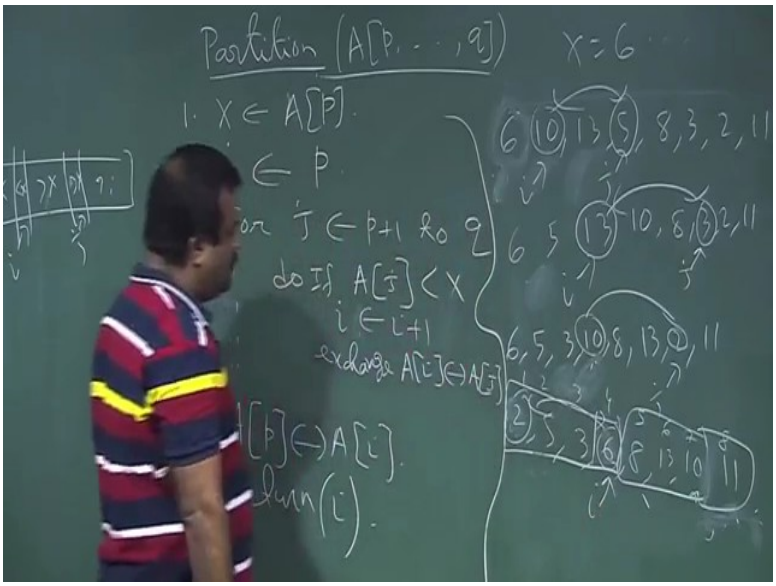
(Refer Slide Time: 19:21)



So, now we put it here. So, x we'll put in here. So, now, we return this value of this element to indicate that where we kept the pivot element. So, that it will identify the left sub array right sub array, because again in conquer step this is only the divide step. Again we in conquer step we have to call the quick sort on this sub array on this sub array.

So to identify the left sub array we need to have the index where have we kept the x. So, that is the point that is why we need to return i, unless we return i we do not know where we have kept x then again we have to call this quick sort on this part we have to call this quick sort on this part. So, this is the pseudo code for partition algorithm now let us have a example then it will be more clear.

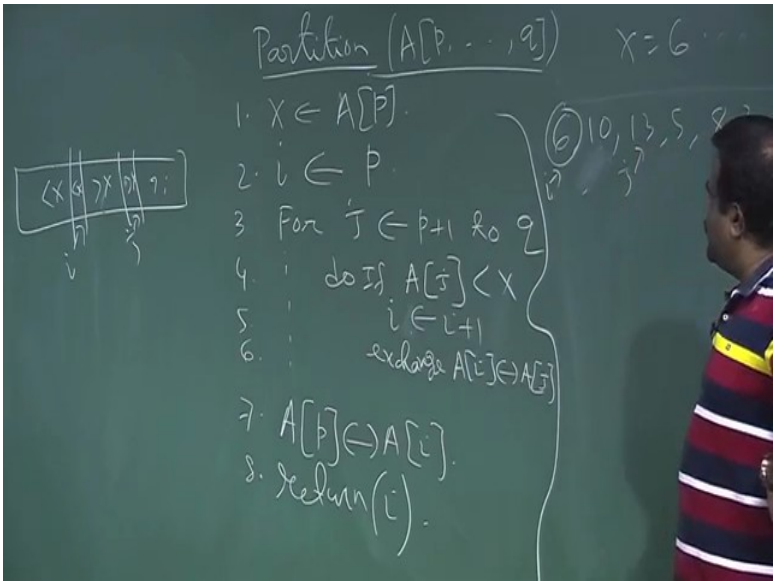
(Refer Slide Time: 20:49)



Let us execute this code on an example on an input. So, say let us take an input say example of partition.

Suppose we have given this input 6 10 13 5 8 3 2 11 suppose this is our input and we want to execute this partition algorithm on this input. So, this is our array. So, this is a p to q, now we choose this as a pivot. So, our x is our pivot is x is 6. So, this is our pivot now we point this to be i this is p plus p and we start our J loop from here, and every time we increase J by 1 by this for loop, now if we now compare and what do we want.

(Refer Slide Time: 21:53)



We want that this is the loop invariant we want; however, this i is pointing and wherever is J is pointing we want all the elements should be less than x or all the elements should be greater than x and this is yet to decide. So, this is the loop invariant you want. So, now, we check this; this is our now A i now if A i is greater than x then they need to do anything we do will not enter into this loop then the J will be increase by 1 just by this for loop, now J is pointing here now again J is 13 A J is thirteen which is greater than x, x is 6.

So, again we increase this by 1. So, this is basically J is now pointing here now if J is pointing here now this is our A J this is our A J now we compare this with 6 now this is less than now we cannot just increase J by 1 then that will highlight our invariant. Now what to do? Now we know that the element sitting here is greater than. So, we just increase i by 1 and we exchange this with this. So, this will be like this. So, 5 will come here 13 10 then 8 3 2 11 and now i is pointing here and J is just this exchange now J will be increase by this now J is pointing 8 .

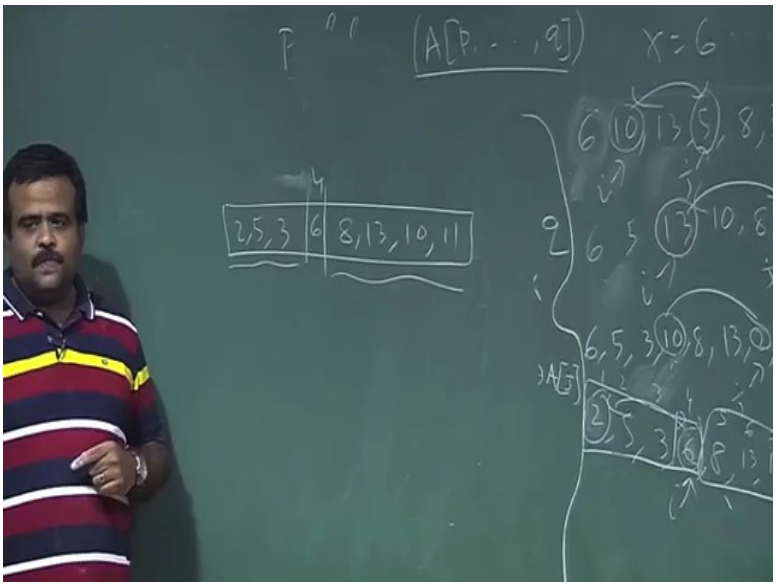
So, now what to do now again we compare A J with 6 now A J is greater than. So, we can peacefully increase A J by 1 and that will be taken care by this for loop. So, now, J has increased by this now A J is 3 now we got a problem now A J is say A J is 3 which is less than pivot element 6, now we need to do something what is that we increase i by 1 and we know the element sitting over here is less than. So, we can exchange this and this and this will give us 5 6. So, this is 3 and 10 8 say 13 will be gain there 1 11 ok now i is pointing this 3, and now J is pointing this. So, 3 and 13 exchange J is pointing this 2 ok.

So, this is the situation now again A J is less than pivot now again we have to exchange this. So, we know the elements sitting over here is greater than. So, i is increased by this and we exchange this and this. So, now, this is the 5 3. So, 2 will come here 8 13 10 will go there and 11. So, now, i is pointing this 2 and J is pointing here, now if you see all the elements up to i these are less than equal to x and all the elements up to J these are greater than equal to x.

So, this is the loop invariant you want for our partition code, this is the loop invariant you want for our partition code and then we increase this J by 1. So, J will be pointing this now this is 11 so, nothing to be done. So, this is one. So, J exhausts. So, J loop is gone. So, we cover everything. So, now this is our i. So, i is basically this this is index if we write 1 2 3 4 5 6 7 8. So, this is basically 4, now i is 4 now what we do we have to put this pivot in its correct position.

So, we exchange this A p with A i. So, we exchange these two 6 and 2 we exchange these two then this will be 2 and this will be 6 and we return i. So, we return 4. So, this will be return as like this. So, this will return this is 6 will be sitting here and this is index is 4 and all the element here is 2 5 3 and 8 13 10 11.

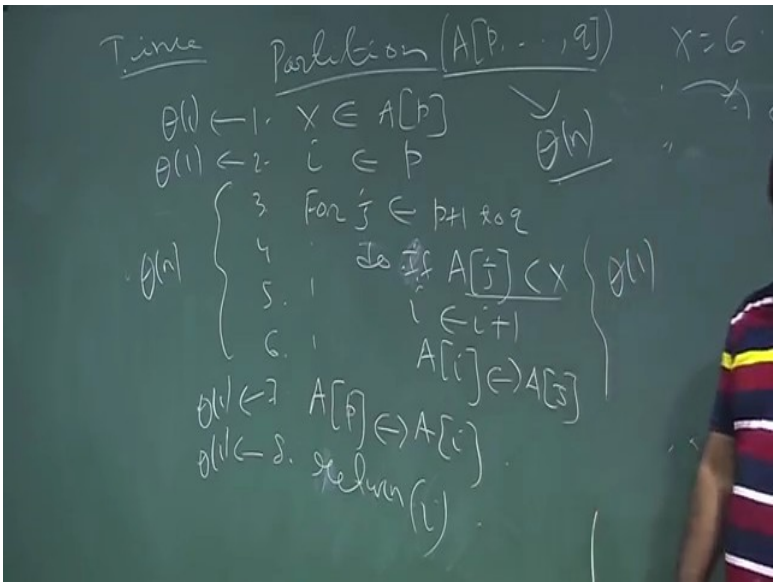
(Refer Slide Time: 26:17)



And this is the index we have to write to identify the left sub array and this is the right sub array. So, this i is the index this is say r. So, this is the i. So, this 4 is returned. So, the partition will return the element 4.

So, this is the partition algorithm now what is the time complexity of this algorithm, because we have a for loop in j.

(Refer Slide Time: 27:07).



So, just if we just erase that. So, just if you write the partition. So, this is basically just quickly I will write this. So, this is the, for J is equal to p plus 1 to q. So, so we want to find the time complexity for this. So, if do if A J is less than x then what we are doing i is equal to i plus 1 and we exchange A i with A J that is it.

So, that will be 5 4 5 6 and then after this we just exchange A p with A i and we return A i. Now we want to find the time complexity for this what is the time complexity for this. So, this will take constant time this will take also constant time now this is a loop in this is a loop in size n because we are assuming order of size of this array is n q minus p is n if we have a input of size n then what is the time complexity.

So, now this is n now whatever we are doing here these are all constant, because we do not care about how much time will be taking by this checking less than then this assignment because we want this analysis to be machine independent analysis. So, that is why we need to bring the asymptotic notation may be in our computer in our branded machine this will take may be microsecond, but if you have old computer it is taking more time. So, we do not bother about that; that is why these are all constant time we are taking theta 1 time for this. So, this is again theta 1.

So, this is basically theta of n this is a linear time algorithm. So, this is a linear time algorithm for this partition if we are given array of size n. Now, we'll talk about quick sort how we will use this partition in the quick sort algorithm, so that will discuss in the next class.

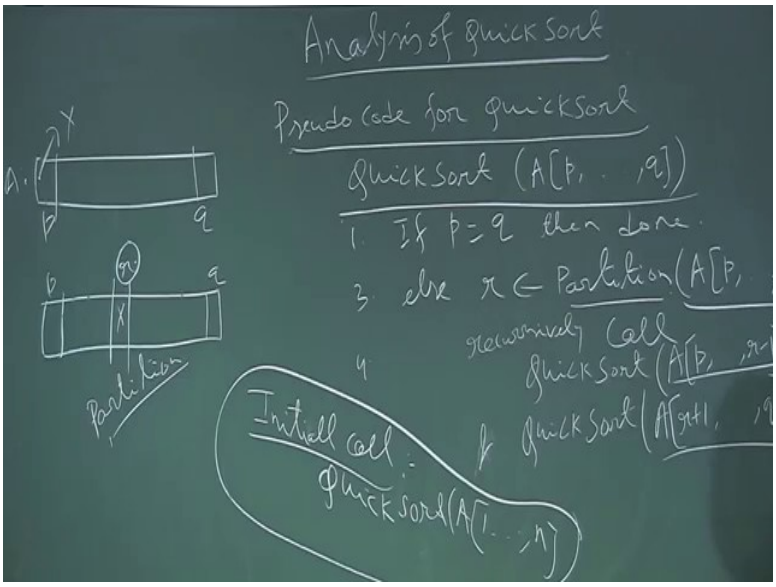
Thank you.

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

Lecture – 11
Analysis of Quick Sort

So, we talk about quick sort analysis of quick sort. So, before that let us just write the pseudo code for quick sort.

(Refer Slide Time: 00:27)



So, for quick sort we need to call the partition algorithm. So, if we just run the quick sort of an array which is starting from p to q, now if p is equal to q then it is done then stop because that is the return step since only one element that is already sorted.

So, we stop, at p this is the stopping condition every branch otherwise else what we do we call the partition; we call the partition on quick sort, we call the partition on this array. Partitions are routine which you have seen in the last class. So, what it will do it will just take this array and it will take this; this is as a pivot first element. So, the array A is from p to q. So, what it is doing it is putting the x in some position and this is the index is p this is r and this is q.

So, it is putting x in the correct position and it is returning the index of this where it is keeping this pivot element. So, this is the partition this is the partition sub routine if doing

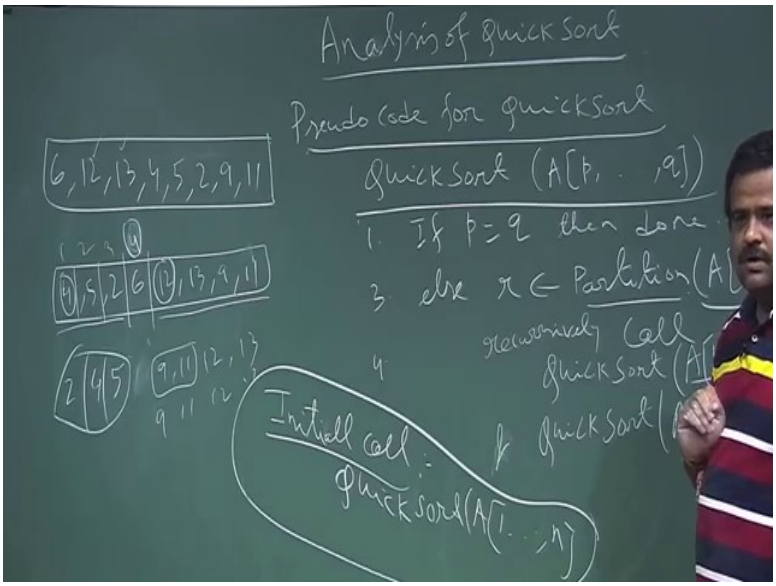
this. So, this we have discussed in the last class and this is the linear time algorithm. So, we call the partition. So, after calling the partition it will divide into 2 part 2 sub array this is left sub array right sub array then again we have to call that is the divide step of the quick sort now again.

We have to call the quick sort on this sub array and this sub array that is the conquer step. So, this is recursive call. So, we recursively call quick sort on left sub array again recursively call quick sort on right sub array so that we have to write. So, this is 4 we call recursively. So, we have quick sort call quick sort on this left sub array. So, that is why we have to return r to know; what is the left sub array. So, if we written r then the left sub array is basically A p to r - 1 this is the left sub array and another.

So and right sub array we have to call the quick sort on the right sub array also. So, right sub array is basically index from r + 1 to q. So, this is the left sub array and this is the right sub array. So, again this is the conquer step. So, we sort this sub array we sort this sub array. So, this is the conquer step. So, this is the sub routine on quick sort and the initial call is initial call is initially our array is 1 to n. So, initial call is quick sort A 1 to n this is the initial call of this quick sort.

So, initially our array is 1 to n slowly it will divide into sub array then sub array. So, the index will be in general p to q. So, that is why we keep the index p to q. So, after the initial call it will not be 1 to n it will be some different index. So, that is why we have; we took the general index p to q.

(Refer Slide Time: 04:34)



So, this is the pseudo code for quick sort. So, now, if we take the take a example. So, if you take a example. So, if you takes say 6 12 13 4 5 2 9 11 suppose this is our input and we have to apply quick sort on this. So, on this input what we do. So, we first apply the partition algorithm if we apply the partition algorithm what it will.

So, it will partition the array into 2 parts. So, 6 it will put 6 somewhere here and it will divide into 2 parts left sub array all the element which are less than x will be left sub array. So, they are basically 4 5 2 and the all the elements which are greater than 6 will be in the right sub array. So, this they are basically 12 13 12 13. So, 4 5 2 are less than 6 12 13 9 11 and this is the index this index is. So, this is 1 2 3 4. So, this is 4. So, 4 will be 4 will be return from this partition algorithm because this; this index we require 2 identify this sub arrays.

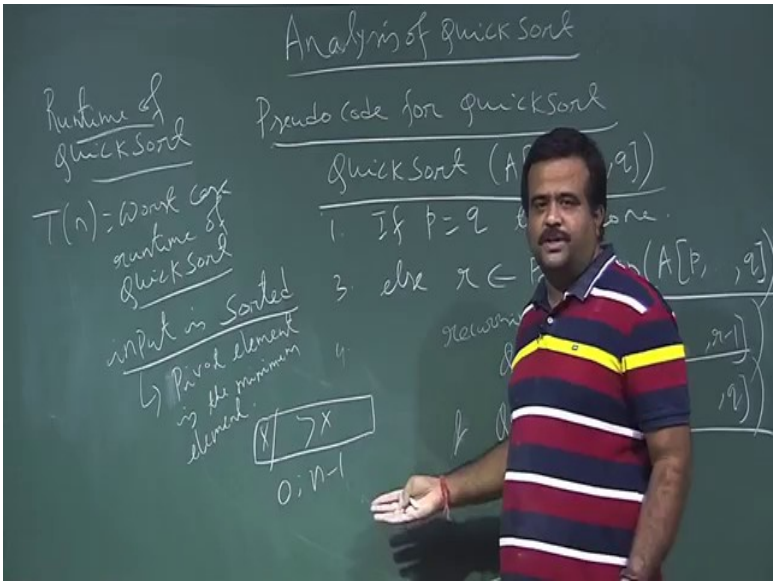
Otherwise we cannot identify this sub arrays if we if we written this for we know that up to three is the sub array and then after this 4 + 1 to q is the right sub array. So, this is the first call now again we have to call the quick sort on this. So, if we call the quick sort on this. So, 6 is in correct position. So, again it will take this as a pivot it will partition into 2 part. So, 4 will be sitting here and all the elements over here is less than x all the elements over here greater than x. So, this is 6 is here and again we call the quick sort on this.

So, this is basically the quick sort call partition on this again we call the quick sort. So, this is the 12. So, 12 will be sitting here all the elements less than 12 will be here 9 11 and greater than 12 now again we call the quick sort on this. So, even we call quick sort on this quick sort

on this there only one element. So, we stop we call quick sort on this. So, this will be again. So, 9 will be there. So, 9 11, so, 12 13, so, done, so, this is basically a recursive call this is basically a divide and conquer approach. So, this is the example of quick sort.

But here in our partition algorithm we are choosing the first element as a pivot so, but we will see we can choose any one of this element as a pivot. So, that will see. So, now, we want to analyze this we want to analyze this means we want to know the time complexity of this algorithm. So, what is the time complexity of quick sort to do the analysis that is quick sort analysis is much more interesting that in the quick sort pseudo code it will it will take much more time to analyze the quick sort than to know the what is quick sort. So, that is more interesting the quick sort analysis part.

(Refer Slide Time: 07:48)



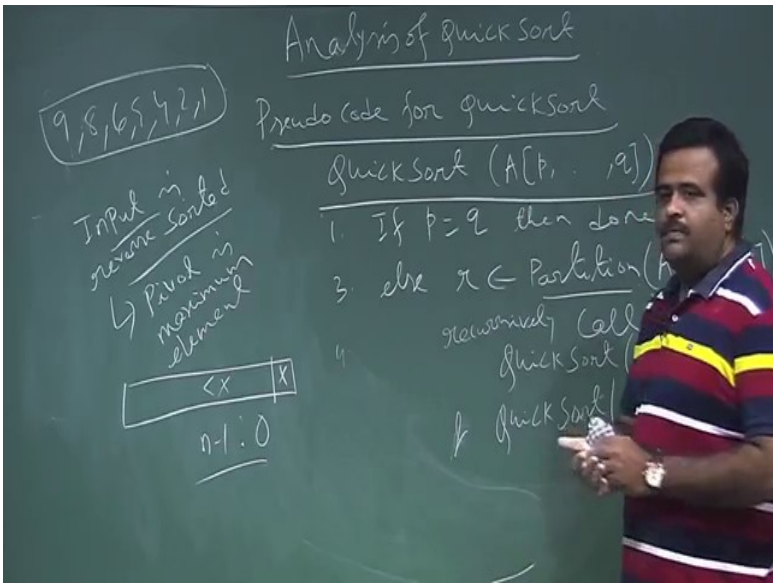
So, we want to know the runtime of this run time of quick sort. So, that is the quick sort analysis and for that we assume that all the elements are distinct in order to avoid this equality to do this we assume all the elements are distinct and T n we take the worst case time complexity for quick sort. So, now, when is the worst case for quick sort when is the worst case for this code. So, if the partition is dividing the array into 0 to n - 1, the worst case will occur if we have an input as a sorted or reverse sorted array. So, if the input if the input is already sorted if the input is sorted then we has choosing the first element as a pivot then always the pivot element in this case pivot is the minimum element. So, pivot element is the

minimum element always. So, if the pivot is minimum element if we choose the pivot is minimum then after calling the partition what will be the situation of the sub arrays.

If pivot is minimum we know the pivot will be sitting somewhere here all the elements greater than x will be here all the element less than x will be here now if pivot is minimum then there is nobody which is less than x. So, if pivot is minimum then the partition will give us this type of structure. So, pivot will be sitting here all the element is greater than here. So, this is 0 is to n - 1 partition. So, nobody is there in the left sub array and n - 1 element will be in the right sub array. So, that is the situation if we choose the pivot to be minimum element among from the given array.

And this will happen if we have a sorted array as input if the input is already sorted and we are choosing the first element as a pivot. So, then it will partition 0 is to n - 1 and again we call this quick sort on this. So, again we choose the first element as a pivot. So, that is the minimum among this. So, again it will be 0 is to n - 1. So, throughout this execution it will be 0 is to n - 1. So, in that case, then what is the recurrence that case. So, if pivot is like this minimum or if the pivot is maximum element so; that means, if the input is say reverse sorted. So, we have to analyze this.

(Refer Slide Time: 10:53)

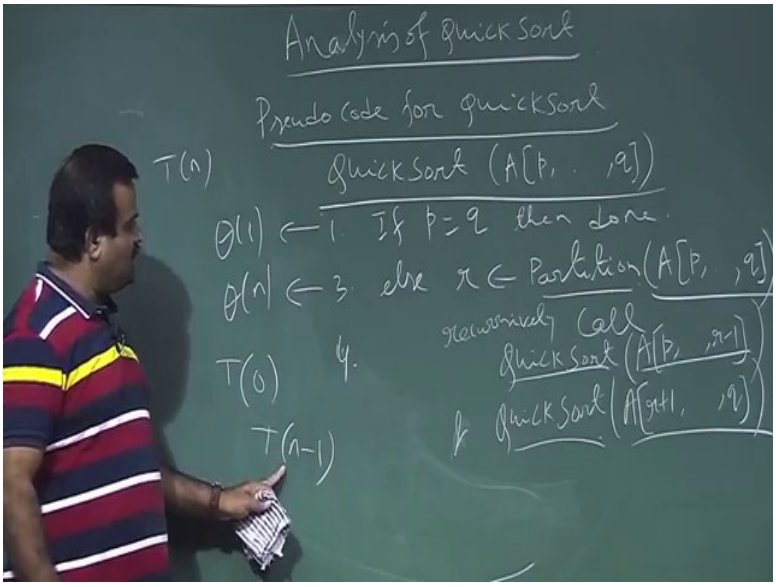


So, suppose our input is reverse sorted means we have say if the input is like this 9 8 6 5 4 2 1 this is the reverse sorted input. If the input is reverse sorted and we are taking the first element as a pivot then in this case the pivot will be the maximum element pivot is maximum

element of array now if the pivot is maximum element. So, how the partition will work then? So, maximum means it has to be here and all the elements has to be less than x.

So, it is basically n - 1 is to 0. So, if the pivot is to be maximum or minimum then it will be either 0 is to n - 1 or n - 1 is to 0 so; that means, there is no one part it is sub array size is 0. So, that is why worst case will come. So, if this is the situation then we want to know; what is the time complexity of this so; that means, we want to know the time complexity of this quick sort when the pivot is minimum or maximum.

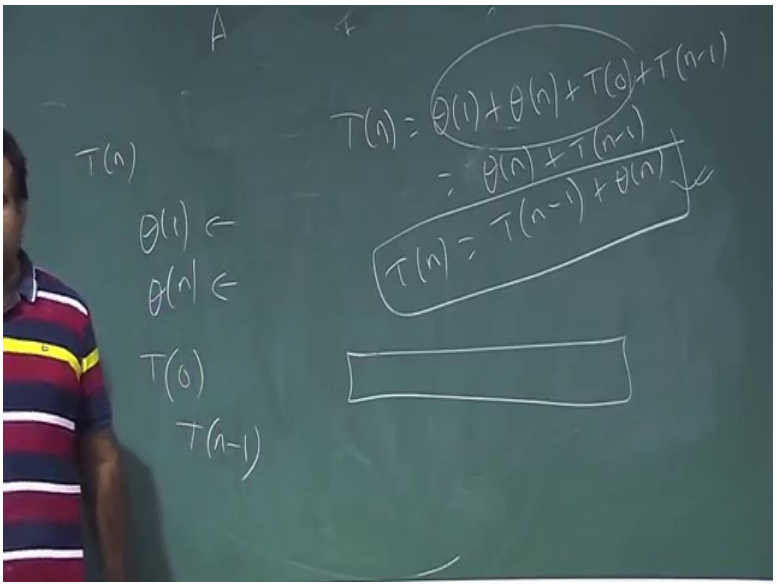
(Refer Slide Time: 12:28)



So, in that case what is T n. So, if T n is the time for this quick sort. So, this is taking theta 1 time and this partition is taking theta of n we know this linear time partition that is the key point of the quick sort and then we have a 2 recursive call of the same function quick sort. So, these 2 call, but we know pivot is here we are taking pivot is minimum or maximum. So, we have one call is on 0. So, this is T 0 and T n - 1 or T n - 1 and T 0.

So, in that case we have this recurrence. So, the time complexity for this, T n is basically sum of this theta of 1 + theta of n + T 0 + T of n - 1 or T of n - 1 + T 0.

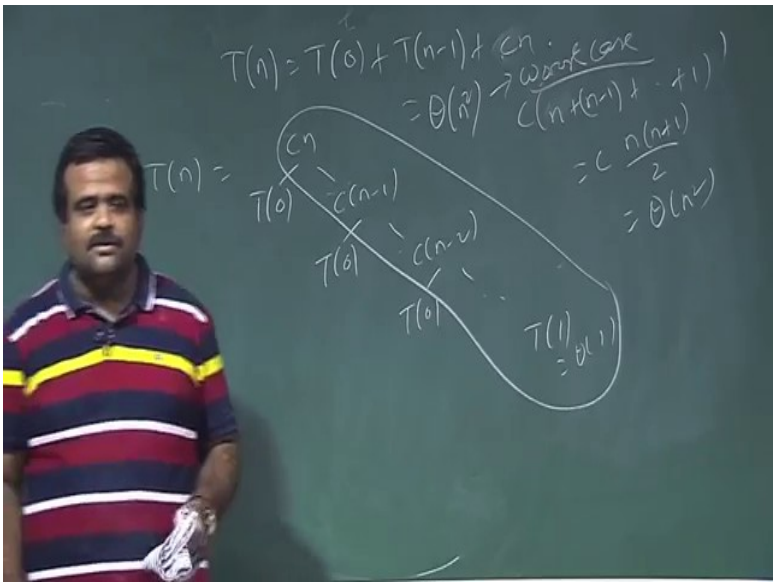
(Refer Slide Time: 13:15)



So, this will all be like theta(n) + T(n-1). So, this is basically T(n-1) + theta(n). So, this is the recurrence for quick sort when the pivot is the maximum or minimum and throughout the execution if the pivot is maximum; minimum in the first level we have given array 1 to n.

We choose the minimum as a pivot for our code we are choosing the first element as a pivot. So, then this is the minimum or if it is reverse sorted then this is the maximum, but for if we choose any element as a pivot element. So, if that element is minimum or maximum always throughout the execution even in the conquer step then this will be the recurrence throughout the execution then we want to know the; what is the time for this what is the time complexity for this.

(Refer Slide Time: 14:56)



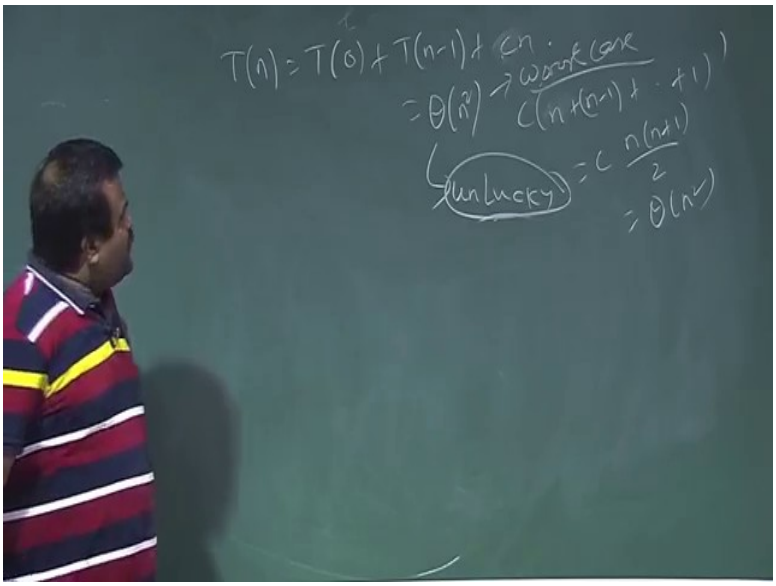
So, to know this time complexity what we do we draw the recursive tree this is the worst case recurrence. So, we draw the recursive tree. So, basically what we have this is T(n) = T(0) + c*n. T (0) is just a symbolic notation. So, if we draw the recursive tree T(n) is basically T(n) this is T(0) this is T(n - 1) now again now this is a problem of size n - 1.

Now, again we have to divide into sub problems, but again in the subsequent step; in the conquer step we assume we are choosing the minimum or maximum as a pivot. So, that assumption has to be made in order to draw the recursive tree in this way. So, again this will be c of n - 1, T(0), T(n - 2) like this. So, again this is the problem of size (n - 1) again we further divide into sub problems, but again in that sub array we are assuming that again we are choosing the pivot to be minimum or maximum.

So, in that case it will be again c of n - 2 T 0 like this. So, this will continue until we stop at T 0 or T 1. So, this is theta of 1. So, if the time is basically sum of this time. So, this is basically arithmetic series n into n - 1. So, this is basically c into (n+1)/2 this is basically order of n square algorithm. So, the solution for this is order of n square algorithm. So, this is the worst case. So, the time complexity is order of n square this is the worst case or we call this is as a unlucky case.

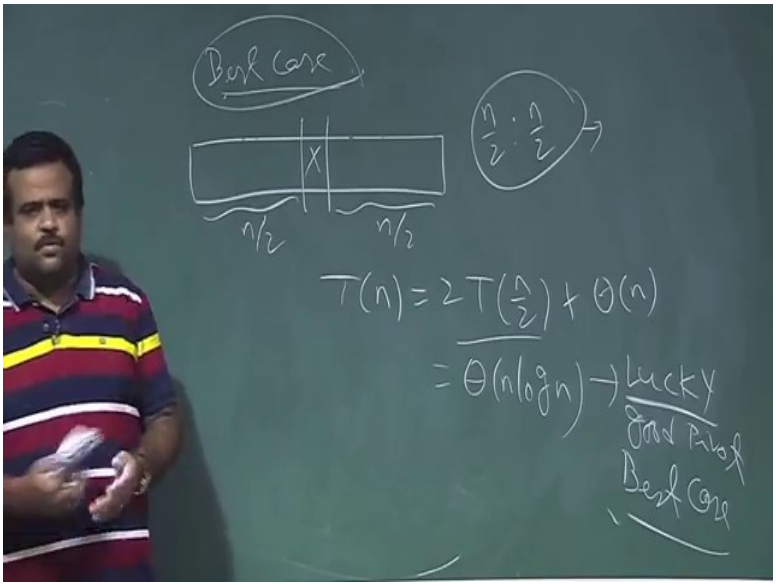
So, what is the lucky case when we get order of n*log(n) algorithm time is order of n*log(n) then we call this as lucky case.

(Refer Slide Time: 16:51)



So, if we choose the pivot to be minimum or maximum then this is a unlucky case we are not lucky we are unlucky if this is the solution of the recurrence now when is the lucky case or when is the best case? So, we know the if the pivot is bad pivot in the sense if it is partitioning 0 is to n - 1 that is called bad pivot. If our pivot is maximum or minimum for the corresponding input then it is a bad pivot. So, that means it will partition. So, then what is the good pivot.

(Refer Slide Time: 17:36)

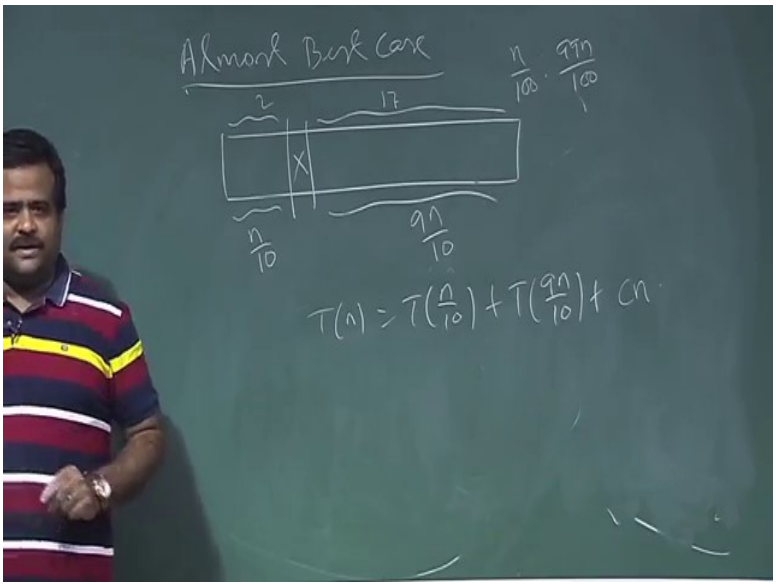


So, that will give us a best case. So, what is the good pivot intuitively if the pivot is partitioning the array into half half so; that means, if the pivot is. So, we are given an input if the pivot is partitioning into half half pivot is sitting here this is roughly n/2 this is n/2 - 1. So, that is puffiness will do. So, that is lower ceiling or upper ceiling. So, this ratio n by 2 is to n by 2. So, then we call say this is a good pivot why good; because this will give us we want to see; what is the time complexity for this. So, if this is the situation then what is our time complexity what is the recurrence T(n) = 2*T(n/2) + theta (n) because in that case pivot is sitting here. Again we have to call quick sort for this quick sort for this, but this 2 have the same size n/2 or n/2 - 1 that will be also n/2 because of this is just a lower ceiling or upper ceiling and this is the cost for partition and those initial steps.

So, this is the recurrence for quick sort. If the partition is n/2 is to n/2 now this recurrence we have seen this is the same recurrence as merge sort. So, what is the solution again if we use the master method we can get this solution n log n. So, this is the lucky case. So, this is the best case scenario when the pivot is partitioning the array into 2 equal part half half n/2 is to n/2 then our recurrence is just because we have to call again quick sort on this quick sort on this.

So, T(n/2), T(n/2) so, that that is why 2*T(n/2) + the cost for partition, so, this will give us the solution order of n*log(n). So, that is the lucky case. So, that is the best case now suppose our pivot is not that much good, but little bit of good in the sense it is not dividing the array into half half it is just and also it is not dividing into 0 is to n - 1 like minimum or maximum it is not the bad one. So, if it is little bit of lucky. So, that is called almost best case.

(Refer Slide Time: 20:17)



So, this will refer as almost best. Why it is almost best? we come to know when you talk about the time complexity for this best case.

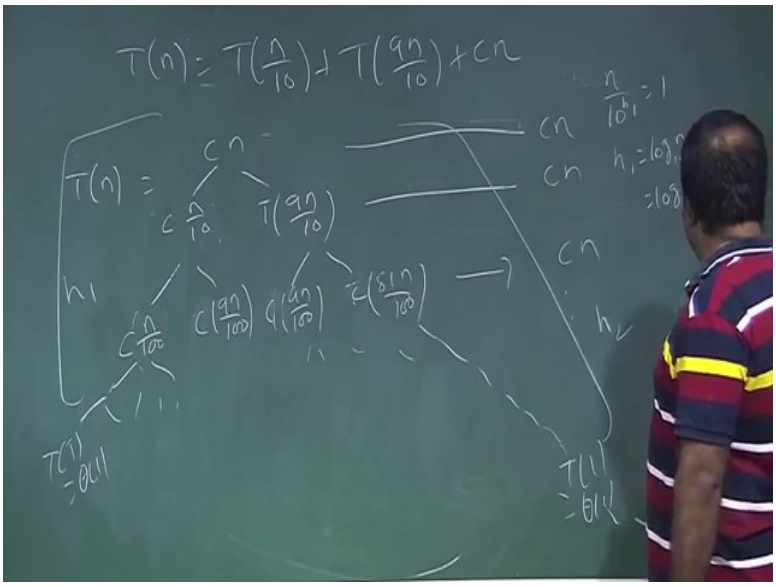
So, almost best case means suppose our pivot is not that much good, but it is not even the worst. Suppose our pivot is such that it is partitioning the array into 2 part 1 is n/10 another is 9*n/10 or may be 1 is n/100 another 99*n/100 for simplicity we choose the former. So, it will work for this also. So, that is the ratio 1/10 : 9n/10. So, this ratio it is partitioning.

So, pivot is such that it is partitioning in this ratio for example, suppose say n is say 20 if n is 20 then how many elements are there this is n by 10. So, this is 2 and this is x if we keep this is 17. So, this is only 2 element. So, if we can assure that pivot is such that it is partitioning some little portion in any side this could be this side also it does not matter which side. So, if we can ensure that it is partitioning into 2 part then 0 is to n - 1. So, if we can assure some little fraction in some of the part then will want to see what is the time complexity of this.

So, if this is the situation 1 part is say then; what is the recurrence then the recurrence will be T n is equal to; so, this is the sub array this is the right sub array or it could be left or it could be right. So, then we have to call this quick sort on this quick sort on this. So, this is basically T of. So, T(n) = T(n/10) + T(9n/10) + theta(n) or basically this is the partition cost c n. So, we want to know where the pivot is?

Such that it will ensure that some fraction may be little fraction some fraction will be there 1 sided and remaining will be all the almost all the elements will be other side. So, it is not 0 is to n - 1 it is better than 0 is to n - 1 now we want to know the solution for this if it is giving us whatever n log n then we say this is a good pivot so. In fact, this will give us order of n log n. So, that we have to see by the help of recursive tree. So, then we want to draw the recursive tree for this recurrence.

(Refer Slide Time: 23:12)



So, our recurrence is T(n) = T(n/10) + T(9n/10) + theta(n) . So, this is the recurrence we have to solve the recurrence using the recursive tree. So, this is the problem of size n we have 2 sub problems 1 is T(n/10) and other T(9n/10) now again this is a problem of size n/10 now again we will use the quick sort for solving this problem again we assume that our partition is portioning this into 1/10 is to 9/10. So, this assumption is required.

We are best case it is half half then we have a different type of tree, but here we are assuming same recurrence will be go through so; that means, now this is the problem of size n/10. So, again we call quick sort on this. So, again it will call the partition again our partition will choose pivot and that pivot is such that again it will be 1/10 is to 9/10 ratio that is important because it is not that suddenly in this step will have half half then we have different type of analysis.

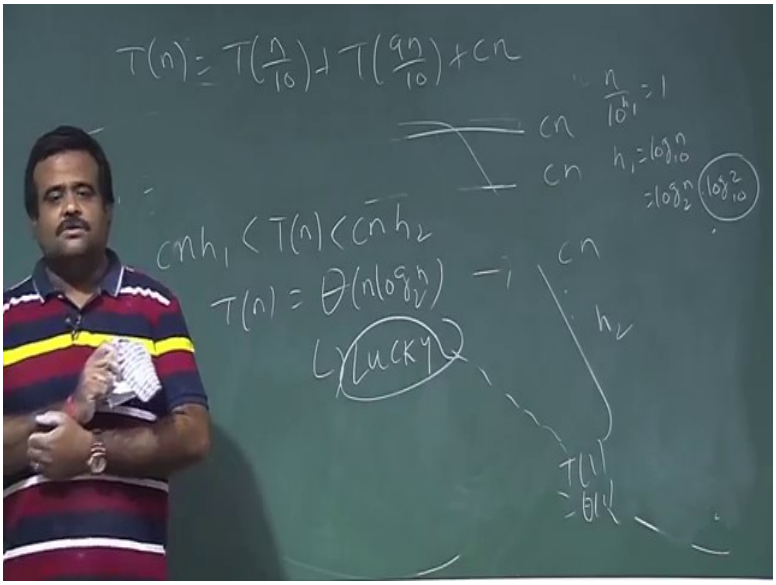
So, again this will be this is now size n/10. So, just put n = n/10 or n/100. So, we put n = 9n/10. So, this way again this is a problem of size n hundred and again we assume this will have the same recurrence again our pivot is such that it will be 1/10 is to 9/10 ratio.

So, if we do that it will be basically c of that c of that c of that c of that and it will further reduce. So, we stop at T 1. So, which branch will end fast, so, this ratio will end this is end fast this will be T 1. So, this will go little more because this is like this. So, we have branches over here every branches will end to the T 1. So, this will stop we stop when we reach the T 1 and this is basically theta 1 and this is basically theta 1.

And our time complexity is the sum of the all the notes to make the sum we want to do the level wise sum this is the first level sum this is also c n this is also c n again we can assume this is c n, but this is not a inductive prove. So, that is why we need to take help of the substitution method to know the solution for this in a mathematical probability. So, anyway, so, this is c n now. So, this is height of the tree is basically order of log n base 2 if the height is h 1 over here and if the height is h 2 over here we can simply say that.

So, what is h 1? H 1 is basically n by 10 to the power h 1 is 1 because that branch. So, h 1 is basically $\log_{10}n$, but we can make it $\log_2 n$ (ignoring any constant multiplication term). So, height is order of log n. So, this side also height will be order of log n. So, if we have a complete tree like this.

(Refer Slide Time: 27:12)



So, it will be T n is bounded by c n h 1 less than T n less than equal to c n h 2. So, this is basically given because h 1 and h 2 are both order of log n. So, this is n log n. This is lucky; n log n is the lucky case.

So, that is why it is called algorithm based. So, that is the thing. So, if our pivot is not half half if we if our pivot is such that it can assure that some little fraction may be in one side remaining on the other side then also we are lucky then also this is the best case. So, that is why so, that we have to guarantee that. So, if it is not even half half if it is just 1/10 is to 9/10 then also we are getting the lucky case then also we are getting the best. So, we will continue this analysis.

So, will talk about we analyze this if we are suppose we are say we have hundred step to execute for a given array now suppose in this hundred step we have 50-50 lucky unlucky. So, first suppose we are lucky then next step we are unlucky then lucky then unlucky. So, 50-50 lucky unlucky situations, then we want to see whether ultimately we are lucky or unlucky. So, that analysis we'll do in the next class. So, there we'll see we are finally, lucky if we can assure some lucky step in the execution then we have eventually we will be lucky. So, that analysis we'll do in the next class.

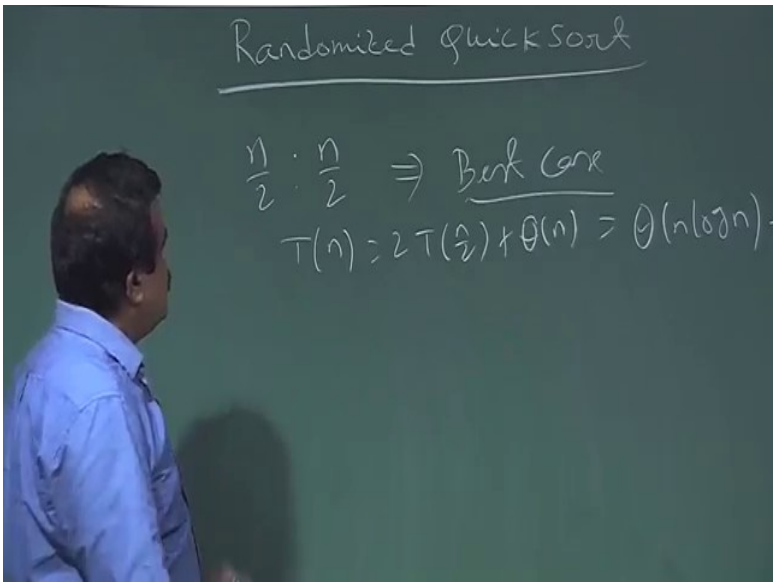
Thank you.

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

Lecture – 12
Randomized Quick Sort

So we are talking about we are doing the analysis of quick sort and we have seen if the pivot is a good pivot then it will partition the array into n/2 : n/2.

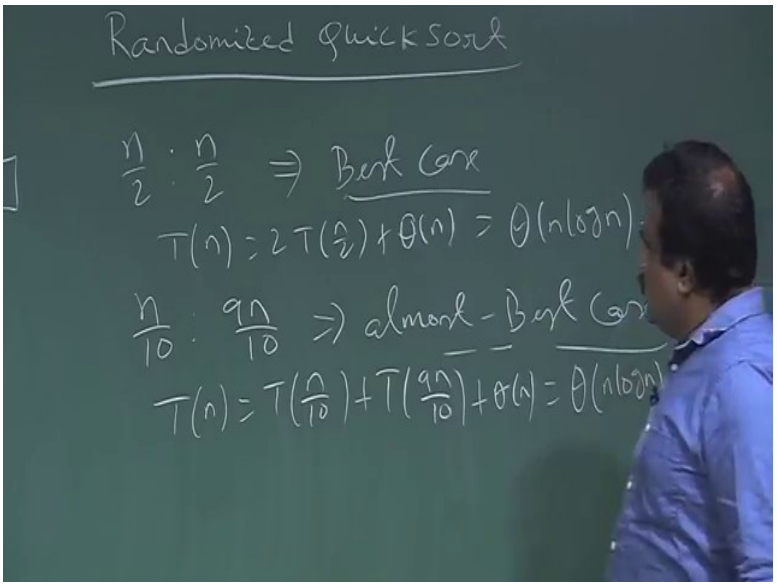
(Refer Slide Time: 00:31)



So, this is the best case. So, if our partition algorithm will partition the array into this way then it is the best case in that case we have the recurrence is 2*T(n/2) + theta(n) and that will give us order of n log n.

So, this is the lucky case order of n log n and even if this is not n/2 : n/2; if just we can ensure the partition will put little fraction over here say 1/10 : 9/10.

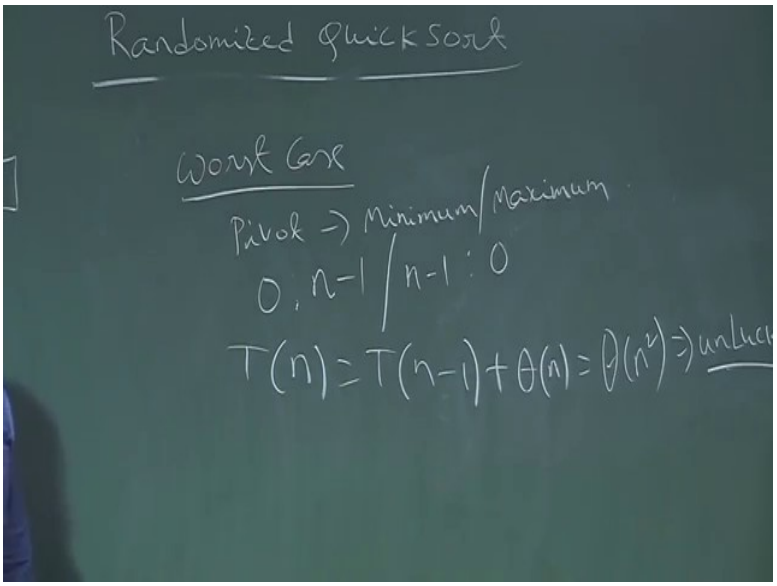
(Refer Slide Time: 01:25)



So that means, if the partition is n/10 : 9n/10 then we have seen this is referred as almost best case. This is also best case because for this we have seen the recurrence is T(n/10) + T(9n/10) + theta(n). This is the recurrence corresponding to this partition and this will give us also order of n log n.

So, this is also lucky case. So, even if one can ensure that it will partition some little portion guaranteed some little fraction in one side and other side the remaining element then also you see it is the lucky case and the unlucky case the worst case is basically 0 is to n - 1; that means, if we choose the pivot element to be minimum or maximum.

(Refer Slide Time: 02:31)

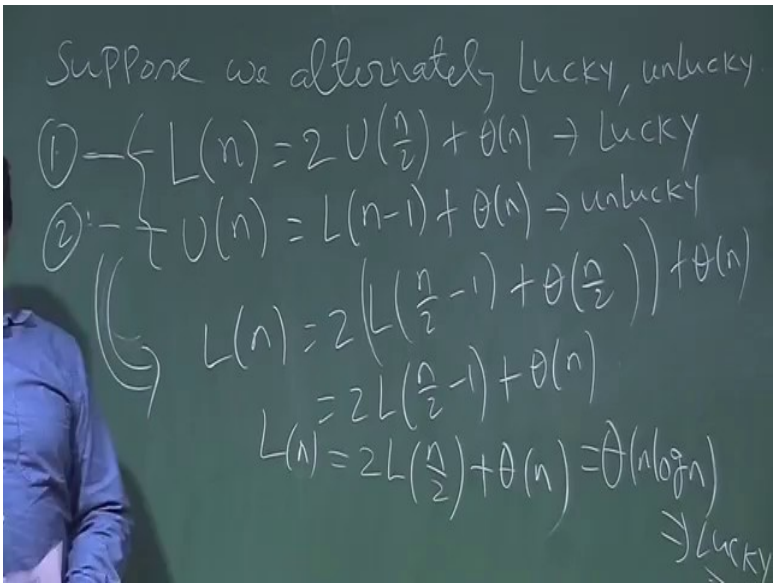


So, the worst case is worst case for quick sort is if the pivot is if the pivot is minimum element minimum or maximum element from the array. In that case the partition will be 0 is to n - 1 if it is minimum or n - 1 is to 0 so; that means, one side nobody is there another side sub array is containing all the elements. So, in that case recurrence will be T(n) is equal to T(n - 1) + theta n and this will give us the solution order of n square.

So, this is the worst case and this is we refer as unlucky case. We are unlucky if the time complexity is coming to be order of n square. So, now we'll do an alternative will do a another type of analysis suppose we are given a array. So, we call the partition algorithm. So, it is partitioning into 2 part if it is good if it is not minimum or maximum then it will partition into nor 0 into n - 1. So, it is a lucky case now suppose in the next time it will be unlucky. So, if we have suppose we are alternatively lucky-unlucky.

So, then we want to see whether eventually we'll be lucky or not. So, that is the analysis we want to do.

(Refer Slide Time: 04:15)



Suppose we are alternately lucky unlucky. So, we are given an array of size n. So we call the partition it reduces the array into 2 sub array and then again we call the partition. So, we have subsequent calls. So, suppose in the first sort we are lucky in the next sort we are unlucky. So, alternatively we'll be lucky unlucky like this sequence and then we want to see whether eventually we are lucky or not.

So, that analysis we'll do. So, that means, now suppose we are lucky. So, this is the lucky recurrence. So, if we are lucky now then it will be partition the array into 2 equal part half half and then the recurrence will be two, but if we are lucky now next time we know we are going to be unlucky. So, that is the unlucky recurrence . So, this is corresponding to lucky recurrence and what is the unlucky recurrence now suppose we are unlucky now if we are unlucky so; that means, it is the worst case.

So; that means, it will divide the array into 0 is to n - 1 so; that means, out pivot is minimum or maximum from the given input. So, in that case the recurrence will be, but now if we are unlucky then next time we know we'll be lucky. So, that is why this unlucky means it will be n-1 is to 0 or 0 is to n-1. So, this is corresponding to the unlucky recurrence. So, now, we want to solve this to get the time-complexity. So, if this is L this is 2 now we put this 2 in L.

So; that means, this will give us L of x 2 of so, now, we can put L(n/2-1)+ theta of(n/2) we are just putting this u u of n u of u of n by 2 u of n is this recurrence. So, we have n we are putting just n = n/2 + theta(n). So, this will give us basically 2 of L(n/2-1) + theta(n) because

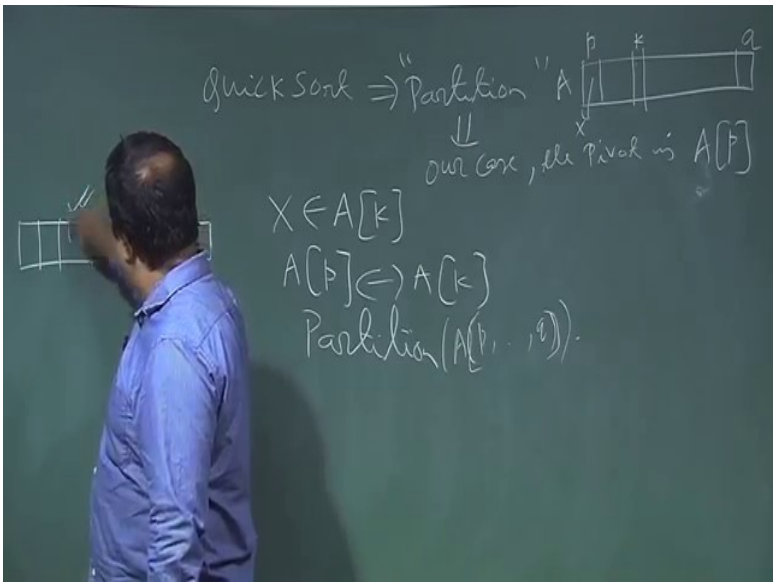
this 2*theta(n/2) will be combined with the theta of n and it together will give us theta(n). So, now, this will play role of like round offing I mean lower ceiling or upper ceiling.

So, this we can easily write as 2*L(n/2) + theta(n). So, this is basically our L(n). So, recurrence for time complexity, this will give us the solution order of n log n. So, lucky case, this is lucky, lucky day. So, we are lucky. So, this is the observation. So, suppose in the whole sequence of execution. So, we have a big array and we are calling the quick sort. So, it is partitioning and then it is dividing into 2 sub array and again we call the quick sort in the sub sequence say it is further partitioning like this.

So, in this whole execution suppose we are fifty percent lucky fifty percent unlucky so; that means, if we are lucky unlucky lucky unlucky in this way then finally, our time complexity will be L log n. So, finally, we are lucky. So, this is more intuition that. So, if we can ensure that throughout our execution we are good amount of time of iteration I mean recurrence we are lucky then it will give us a lucky; lucky case. So, that will give us the idea of randomized quick sort. So, if we can choose the pivot randomly then I mean there is a chance that the pivot will not be minimum or maximum.

So, it may be minimum or maximum, but there is a chance that it will not be minimum or maximum, that way, it may give us the lucky sequence; lucky recurrence. So, that way it will ensure sort of that eventually will be lucky. So, that we'll do in the random way quick sort. So, in the whole sequence of execution if we can ensure at some number of times of this recurrence is lucky recurrence then eventually we will be lucky. So, this is one idea behind this randomized version of the quick sort and another idea is the adversary point of view. So, suppose we have a quick sort version and quick sort.

(Refer Slide Time: 09:55)

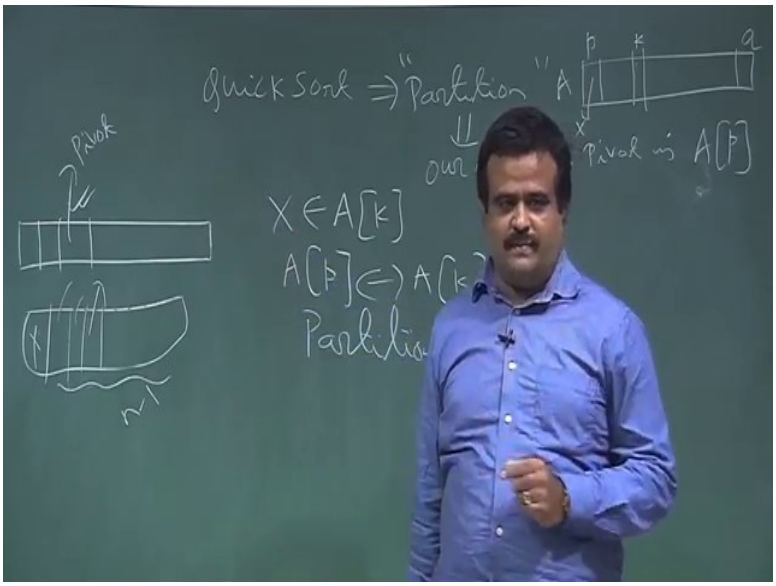


So, we have a; we have corresponding the partition subroutine now in our partition algorithm we choose the first element as a pivot. So, our partition our case we choose. So, our case the pivot is p basically the first element. So, if we have array starting from p to q. So, our case, the pivot is this one; x our partition algorithm. So, we know the position of the pivot suppose we want to take any other position as a pivot say third element from this array.

So, this is the k th element suppose we want the k th element it is not the k-th smallest or something it is the index k th index suppose want to. So, instead of first element if we want to take third element as a pivot in generally if we want to take k th element as a pivot. So, k th element means k th index. So, A k as a pivot say then we can change our partition algorithm or we can do what in the initially we will just change this A p and A k exchange A p and A k and then we call our original partition algorithm.

Because now we have exchange this pivot element in the first element then we call our original partition algorithm anyway that is we can modify the partition algorithm accordingly. So, now, we can call the partition algorithm with the index p to q now this as a if we fix the position of the pivot this as a drawback instead of first element.

(Refer Slide Time: 12:10)



If you fix this third element if we this is our say rule we always choose this is we always choose this element as a pivot in here we are choosing the k th element.

So, if k is three this is the pivot now what is the drawback of this suppose we design a algorithm the sorting algorithm and we give it to the somebody to buy this. So, what that company will do company will take this algorithm and company knows that I knows my pivot is the third element. So, the adversary always can construct a input where putting third element as a minimum or maximum of the given array. So, we can always do that. So, we are given an array. We put the minimum or maximum in the third position then we call the partition.

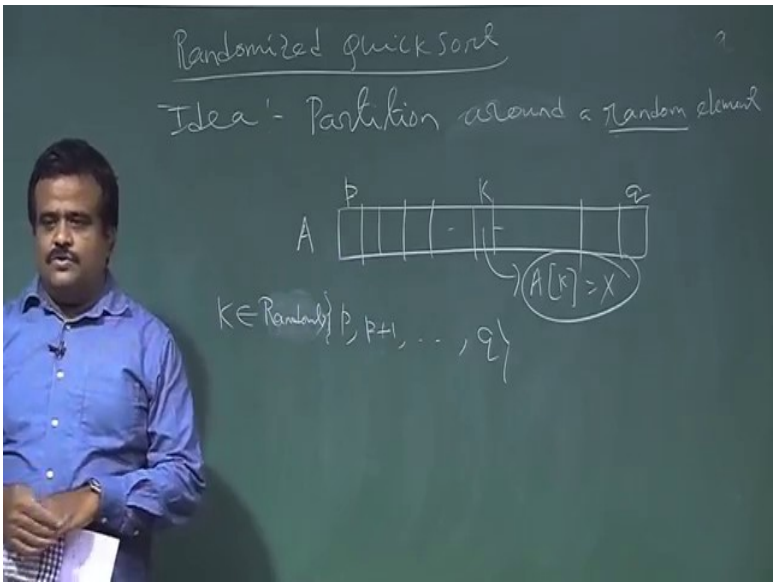
So, if we put minimum again it will divide into just x will be here remaining all the n - 1 element will be here again from this. So, from this sub array this is first element this is second this is third again we this will be going to be pivot again we put the minimum among this array remained here. So, this way one can always construct a input where always we are going to choose the pivot as a minimum or we can choose the maximum. So, that is the drawback. So, if we know before what will be the position of our pivot whether first element or third element then one can always come with an input.

Where by putting the minimum or maximum element in that position always throughout the execution sub sub array then that will perform the worst case. So, that will always give us the 0 is to n - 1 or n - 1 is to 0 partition because that is happen to be a minimum or maximum of

that corresponding array or sub array. So, is this clear? So, if we know the; knowing the position of the pivot is dangerous in that sense. So, we can always come with same input where our code will perform worst case I mean perform badly.

So, to avoid that the idea of randomized quick sort came, so, randomized quick sort idea is we will not let the others know which one which position we are going to choose as a pivot. So, that will determine at the run time. So, that is the randomly we choose the pivot element among this element. So, that will save us this type of constructing of this type of input where our code will perform badly. So, that is the idea of randomized quick sort. So, we talk about randomized quick sort. So, idea is to choose the pivot element randomly.

(Refer Slide Time: 15:09)



So, idea is to partition the array partition around partition around a random element random element. So, we have choosing a pivot randomly. So, we have given a array this is a array starting from p to q initially it is want to a. So, any element could be a pivot element any position. So, we just choose a random number k. So, this is say set index set p p + 1 up to q from this index set we choose A k and k randomly we generate random number randomly. So, we choose a.

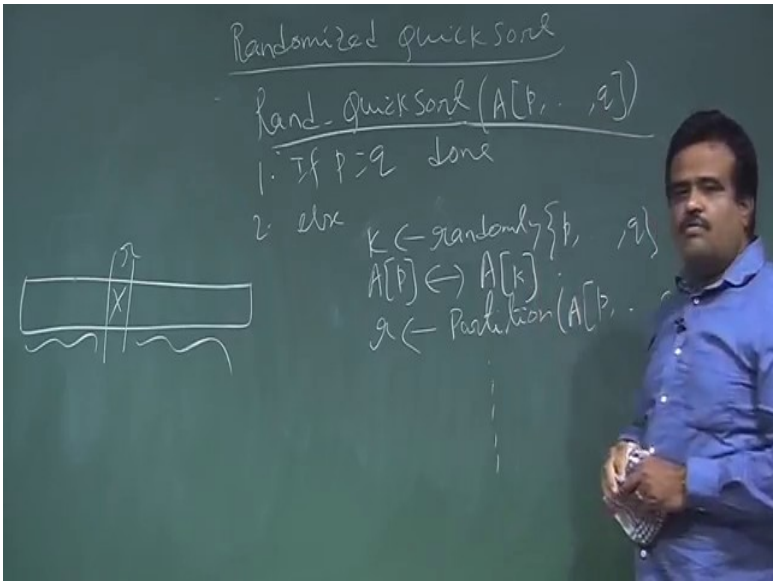
So, if there are n elements, they are equally likely to come as a pivot element. So, this is 1 by n. So, we choose a k. So, from this index we choose A k randomly. So, this is k and this is going to be our A k is going to our pivot this is our x and this is a random choice. So, before running the code we do not know which k will be coming it depends on the random number

generator. May be this A p can be pivot may be A (p + 1) we do not know any one of this. So, any one of this element can come with a probability 1/n if there are n element. So, that is the idea.

So, we choose the pivot randomly from this set at the runtime. So, before running the code nobody knows which is going to be the pivot element. So, that is why adversary or nobody can come with some input where our code is performing bad because we do not know which position is going to be pivot. So, that is why we cannot put in that position minimum or maximum of that array. So, that is the idea. So, we choose the pivot randomly among this set. So, we have n element among this anyone of this can participate as a pivot. So, that will be decided at the rum time. So, run time we generated random number based on that we choose a index and that A k will be the pivot element.

So, this is the idea of randomized quick sort. So, we can just write a code for that. So, we can just quickly write a code rand quick sort.

(Refer Slide Time: 18:12)



So, it is basically A p to k. So, now, we earlier version of the quick sort if A p is our pivot element, but here, again we have the initial condition if p is equal to q then done else what we do we have to call the partition algorithm and that partition should be random partition.

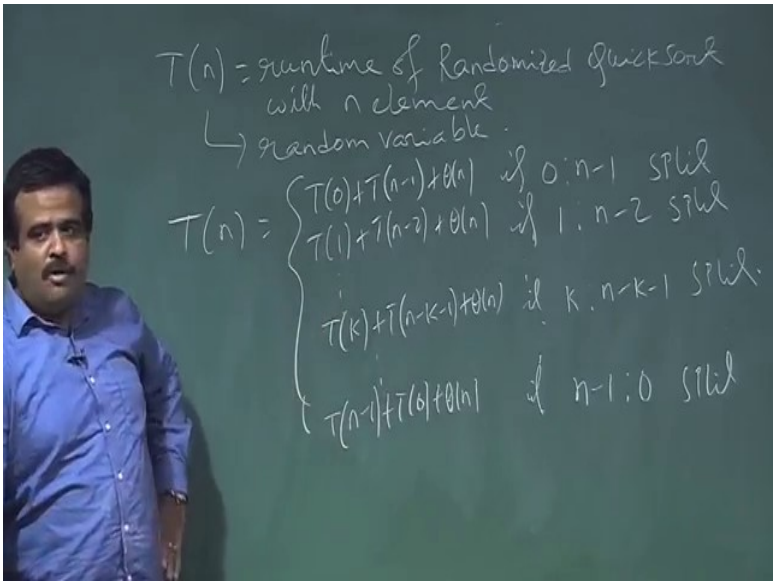
So, we call the else say r is equal to will have to explain what is the random partition from A p to q and even if we want to use our partition then what we do we know that our original

partition is taking first element as a pivot then what we do we just call this we choose A k randomly from this set. So, generated random number we choose A k from this set and then A k is our pivot now we exchange A k with A p because we know the our partition is.

Basically taking first element as a pivot then we call the then we call this partition our original partition algorithm our first element as a pivot and then it will partition into 2 part and then we have that just the remaining call quick sort. So, it will partition into 2 part this is the r x will be sitting here this is the left sub array right subarray again we have to call the randomized quick sort on both the sub array and again on this sub array also we have to call a pivot randomly. So, that is very important.

So, that is very important. So, in a subsequent step also we choose the pivot randomly. So, again for this sub array we have to call A k we have to generate a random index k and that is going to be the pivot. So, similarly here and the initial call is rand quick sort from a 1 to n. So, this is the code for randomized quick sort now we want to analyze this code what is the run time of this. So, to analyze this we need to take help of random variable some probabilities stuff will come.

(Refer Slide Time: 20:51)



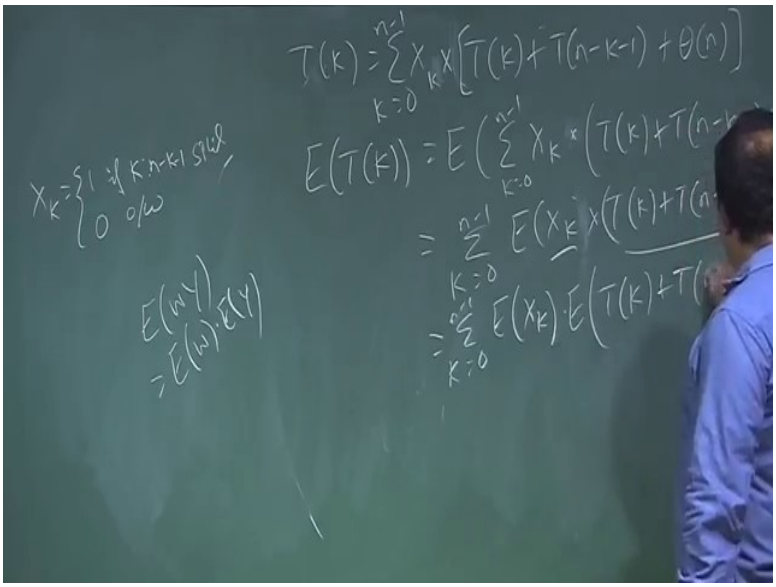
So, what is the time complexity? So, we denote T(n) is the runtime of randomized quick sort with n element.

So, then T(n) is a random variable basically T(n) is a random variable because T(n) will depend on the input pattern. So, that is why it is random variable. So, T(n) can be written as the form which depends on the partition. So, if the partition is 0 is to n - 1 if this is the split then T(n) is T 0 + T(n - 1) + theta(n) if this is the split otherwise if T 1 + T (n - 2) + theta of n if the split is n - 2 and so on...

If the split is change to n - k - 1 split then it is basically T(k) + T(n- k - 1) + theta(n) dot, dot, dot and this will be up to this n - 1 + theta of 0 + theta of n if the split is n - 1 is to 0 split. So, T(n) is in this functional form. So, we do not know which partition will occur we do not know which split will occur, but we know one of the split will occur any one of this. So, if it if the random number is bad then the split will be this if the random number is very good, then the split will be half half and the, depending upon the random number we are choosing the pivot element thus we are choosing the partition as one of this, but before running the code we do not know which one will occur. So, this is the time complexity that is why it is a random variable this is time complexity in the functional forms now we want to make it in algebraic form. So, that we can take the expectation, to make it in the algebraic form we need to take help of what is called indicator random variable.

So, what is that we will just defined x k is 1 if the split is k is to n - k - 1 if this is the split otherwise 0 0 otherwise. So, x k is the indicator random variable which is 1 if the split is this. So, 1 of the split will occur so; that means, 1 of the x k is 1 and then remaining x k is 0 and corresponding that expression will come so; that means, we can write this functional form in algebraic form like this in terms of this x k.

(Refer Slide Time: 24:28)



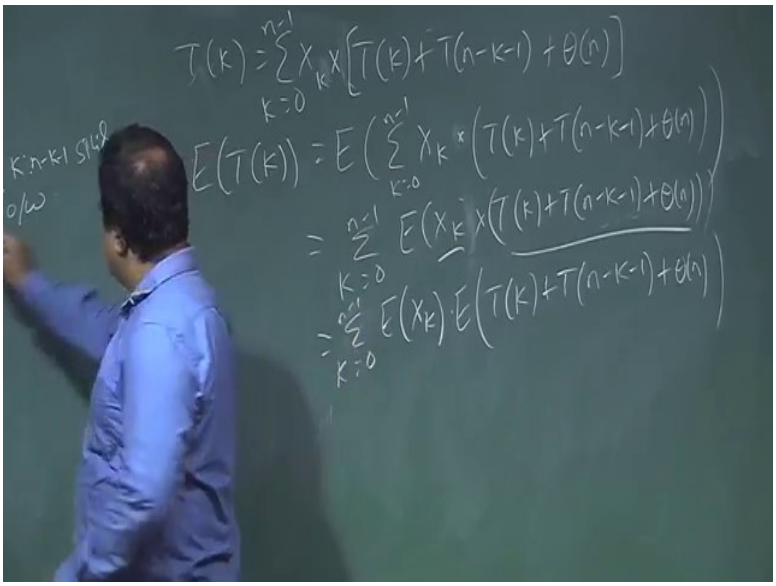
So, T(k) can be written as summation of x k into T(k) + T(n - k - 1) + theta(n) this is x k and k is varying from 0 to n-1. So, this is the expression that functional form we write in algebraic form because we want to take the expectation we want to find the expectation.

So, now we take the expectation on both sides. So, this is basically expectation of this summation of x k into this is into T(k) + T of n - k - 1 + theta of n. So, this is basically k is 0 to n - 1 now expectation is a linear function we can take the expectation inside. So, this is basically summation of x 0 to n - 1 expectation of x k into this form T(k) + T (n - k - 1) + theta (n).

So, now, if we take this to be independent random variable this and this then why from this independent rays are coming. So, we are choosing the random number for choosing the pivot element now every subsequent step we have to choose. So, that choice of random number if they are independent then we can say this is independent. So, if we assume that independence then this product can be written as. So, if 2 random number is independent expectation of w and say y if they are independent then we can write as this. So, this property is there in the probability theory.

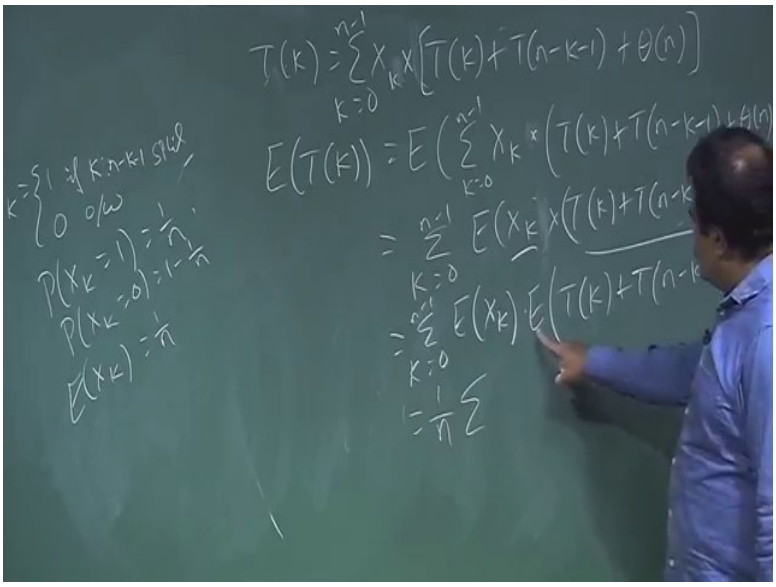
So, that's why that assumption we can write this is k is equal to 0 to n - 1 expectation of x k into expectation of this theta of n.

(Refer Slide Time: 26:54)



So, now this is basically expectation of this is a discrete random variable so expectation of this. So, probability of x k equal to 1 is basically 1 by n because there are n possible split among this split one of they are equally likely.

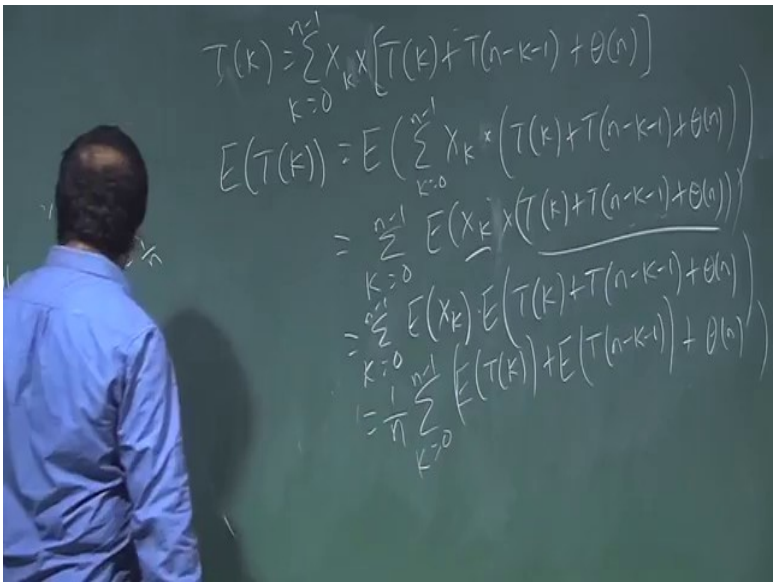
(Refer Slide Time: 27:18)



So, this probability is 1/n. So, the expectation and probability of x k equal to 0 is 1 by this. So, expectation of x k is basically 1 into 1/n and 0 into this. So, it is basically 1/n.

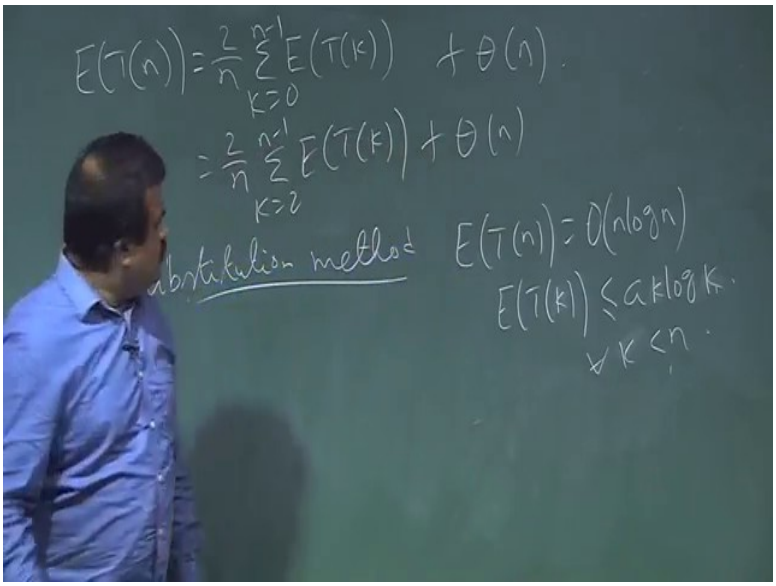
So, this will give us basically 1/n will come our summation of expected value of this now this again we take the expectation inside

(Refer Slide Time: 27:33)



So, expectation of T(k) + expectation of T(n - k - 1) + expectation of theta(n) so, that is nothing to do with random n that is basically theta n, this is term k is equal to 0 to n - 1. So, if we simplify, this will give us basically what.

(Refer Slide Time: 28:17)



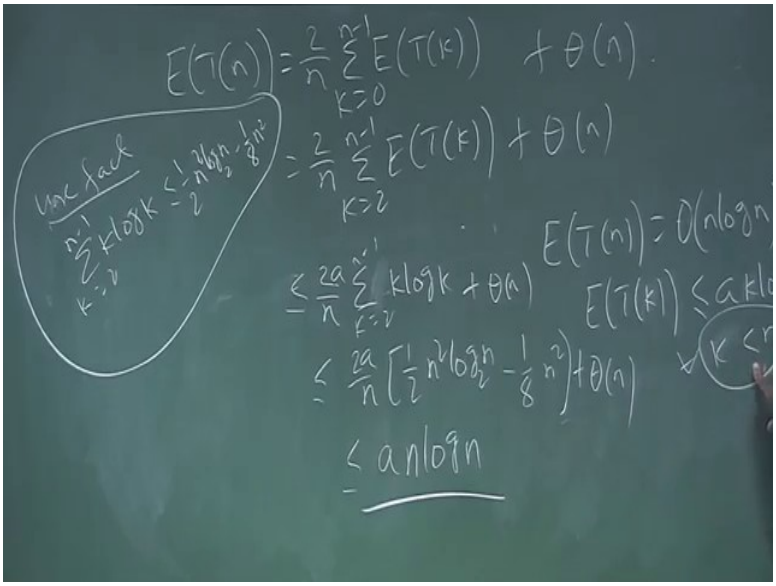
So, we want to show the expectation of T(n) is basically n log n. So, that is our goal we want to show this expected value of T(n) is theta of (n log n). So, this is basically now this 2 term these 2 sum we can take the sum inside. So, this is from 0 to n - 1 and this is for if we put n is equal to 0 it is n - 1 to 0. So, 2 terms is coming out and this will combined with the theta of 1.

So, this is basically 2 by n summation of expectation of T of k; k is equal to 0 to n - 1 + theta of n.

So, you got this now this first 2 term we can combine with this. So, this can be written as 2/n summation of k is equal to 2 to n - 1 expectation of T(k) theta of n why we take 2 to n - 1 we'll use a result which is starting from 2 to n - 1 and for 0 and 1 this term will be combined with theta of n. So, we got this; now we want to show this is to be theta(n log n). So, for that we have to use some inequality.

So, that is we got this now we assume this is the substitution method we prove this using substitution method. So, what we are trying to prove we are trying to prove expectation of T(n) is big theta of n log n. So, now, to prove this we are assuming this expectation of T(n) is less than equal to sum a*n*log(n) where a is a constant. So, this is our assumption. So, this is true for k this we have to prove. So, k and k, this is true for all k less than equal to n.

(Refer Slide Time: 30:36)

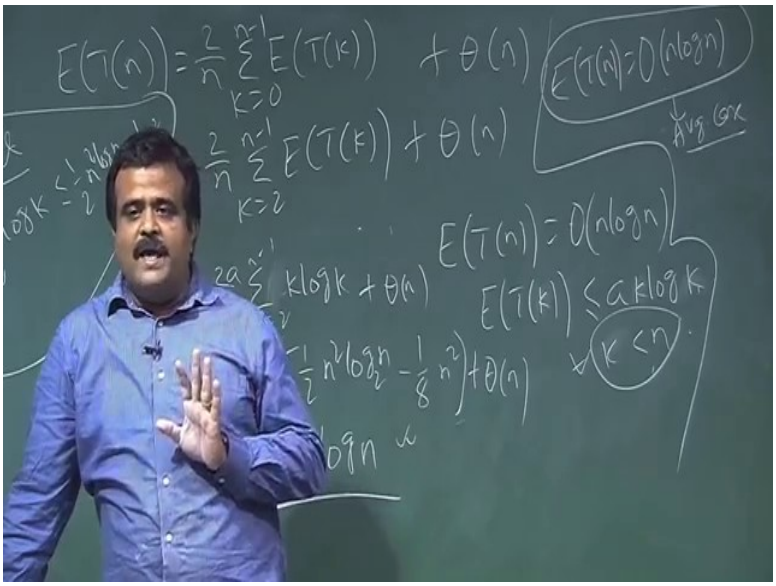


So, this is all less than 2 by n and summation over A k log k + theta of n. So, now, we will use a result use this fact what is this fact and the fact is summation of k log k k is from 2 to n - 1 that us why you take 2 to n - 1 is less than equal to 1 by 2 n square log n base 2 - 1 by eight n square. So, this result we are going to use.

So, this result again we can prove by mathematical induction summation of k log k. So, if you take this a out a will be here. So, this is summation of k log k and this k is starting from 2 to n

- 1 this is basically less than equal to this is also we can prove by induction 2 by a n and this is basically half n square log n base 2 - 1 by eight n square + theta of n. So, now, we can choose this such that this will be basically less than equal to a. So, this is A n log n. So, this is true for n. So, we assume this is true for up to up to n - 1 now we prove that this is true for n.

(Refer Slide Time: 32:16)



So, this is true for all n. So, basically we prove that the randomized quick sort expected run time is n log n, but this is expected run time this is what is called average case analysis but worst case runtime is always order of n square because in the worst case we always choose the pivot in such a way randomly. So, what case pivot will be always minimum or maximum? So, worst case run time is always order of n square whether it is randomized or normal, but if it is randomized version then expected run time is order of n log n.

So, this is the randomized version of the quick sort and quick sort is in place sorting algorithm for quick sort we are doing everything in the array we are for like in merge sort we are taking extra array for doing the merge sub routine for quick sort we do not need the extra array. So, it is an in place sorting algorithm.

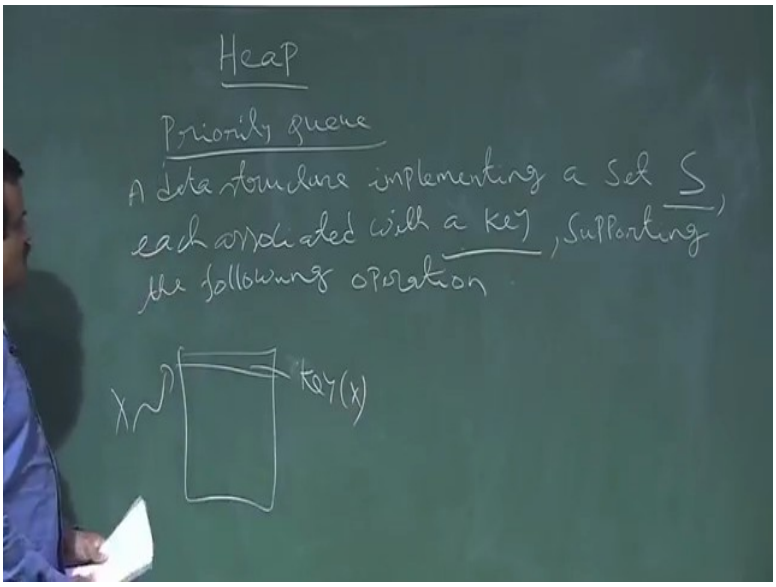
Thank you.

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

Lecture – 13
Heap

So, we talk about very cool data structure which is called heap and we'll basically talk about heap sort, but heap can be totally independent from heap sort. So, basically heap is nothing do with heap sort, but we use max heap to have a sorting algorithm and that is called heap sort.

(Refer Slide Time: 00:57)



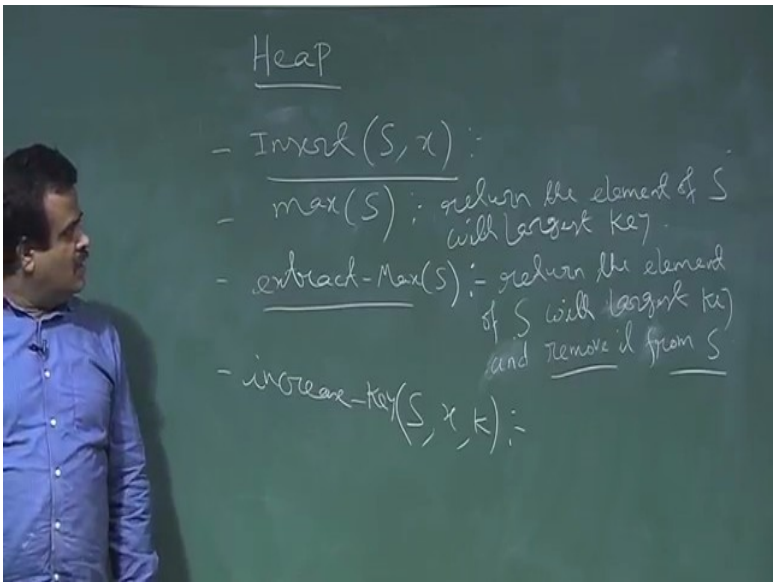
But heap can be taught totally separate way like for specially it is a priority queue implementation. So, basically what is a priority queue?

So, basically a data structure implementation; implementing a set S which is associated with a key each element in S is associated with a key and it should support the following operation, supporting the following operation. So, basically we have a dynamic set S and this S set and each element in the S set is associated with the key. So, key is basically we can say if S is record stage student record, student roll number.

So, roll number could be the. So, if this is a record. So, this is the roll number of the student. So, this could be the unique key of x. So, each of this data or we can say record is associated

with a key. So, key is the unique representation of that set. So, each element of S is associated with a key and it should able to perform the following operation like insert we should able to insert an element in the S.

(Refer Slide Time: 02:51)



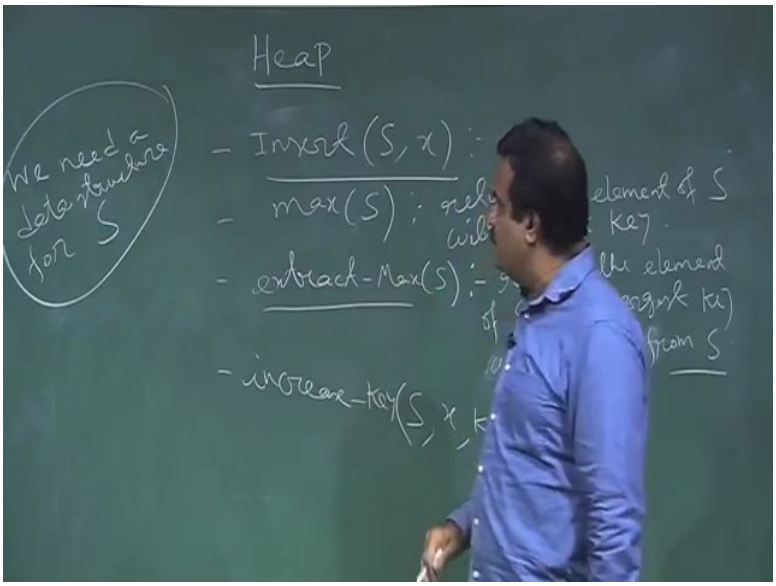
So, this is basically we insert x into f. So, we insert x a new record if the x is the key of that record into S new record with some key value in x this is the insert operation and there is a max operation max of S. So, this is basically the return the element whose key value is maximum return the element of S with largest key value. So, we are looking for maximum element from this set. So, S is a dynamic set every time.

So, we should able to insert a new record new key in this S and we should able to find it out. Another operation is max operation. We should able to get the maximum of this set. Maximum element from this is the largest key value and we should able to do this. Another operation is called extract; extract max it is telling us we should return a maximum and delete from the queue.

We should return the element of S with maximum key with largest key value and remove it from S that is also important. So, extract max is we are extracting the maximum element that is the max S and we should be able to remove that element from S. Remove means delete and remove it from S. So, this is also important.

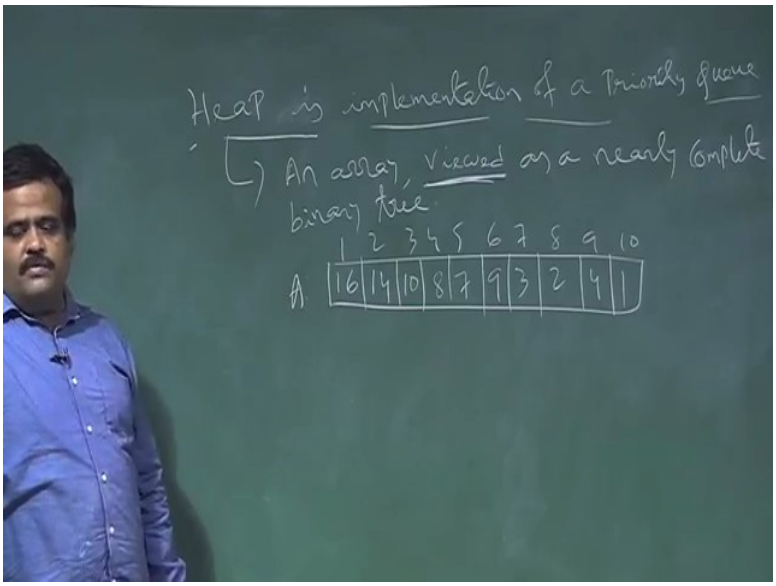
So, extract max means we should able to get the maximum element. We should able to get the element whose the key value is maximum and we should able to remove that element from this set S and another operation is increase key. So, we take a key we take a x and we should able to replace its key value by the new key value k. So, that is the increase operation. So, it should go to that particular value and it should able to increase it.

(Refer Slide Time: 06:10)



So, we need a data structure for this. So, we need a data structure for S. So, that is our goal. So, we need a data structure. This is called priority queue implementation. So, we need a data structure for this S so, that we can perform this type of query so, for that heap is the data structures which are going to introduce for this purpose. So, heap is basically it is basically an array, but it is viewed as a completely binary tree nearly complete binary tree.

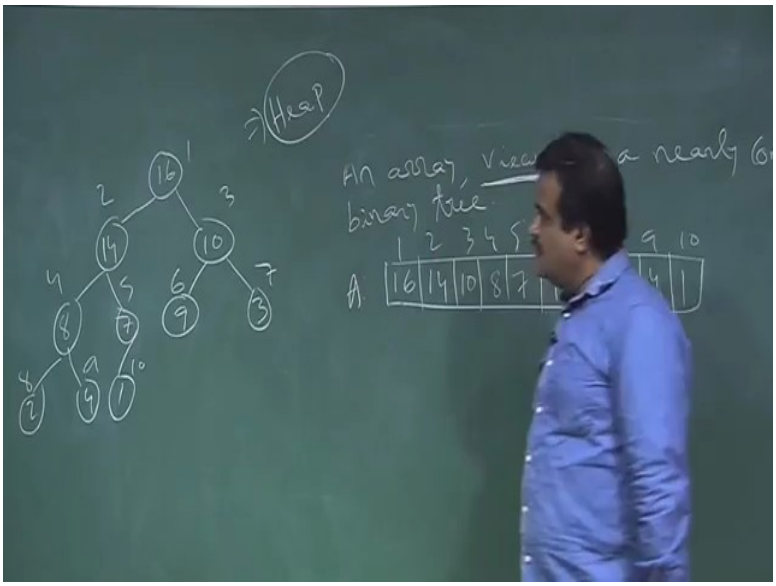
(Refer Slide Time: 06:57)



So, heap is basically implementation of a priority queue. So, heap is basically an array which is viewed as or visualized as A, this is the viewed or visualize as a nearly complete binary tree how we will explain complete binary tree. So, basically it is an array, but we are viewing this array as a binary tree how we can view an array as a binary tree.

So, that we have to explain. So, suppose we have given a array. So, say we have array. So, this is 16, these are the element these are the key value 10 8 7 9 3 2 4 1. So, how many element one 2 3 4 5 6 7 8 9 10, so, we have a array of 10 element now this array we want to view as a we want to visualize this array as a tree.

(Refer Slide Time: 08:48)



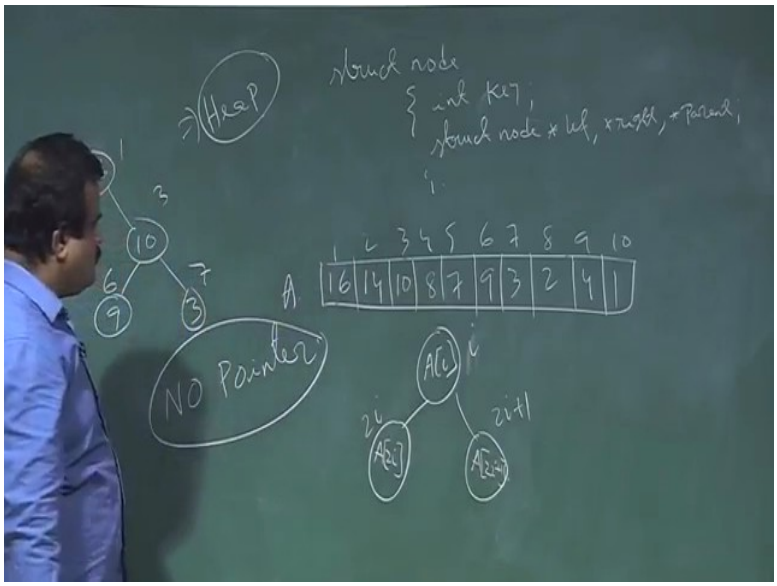
So, how we can do that so, one way to do this is we start with this as a root. So, we start with this as a root.

So, this is A 1 and then A 2, A 3, then A 4, A 5, A 6, A 7 then A 8 A 9 and then we have a 10 because array is up to 10. So, this is the way we view this array as a tree. So, this is basically A 1; A 1 is 16. So, we just put this value this is the key values 10 8 7 9 3 2 4 one. So, this is the visualization of this array as a tree and this is nearly complete binary tree.

Because here number is 10 if the number is say 15 if we have another 1 2 3 another 4 element in the array if the array size is 1 to 15 then it is a complete binary tree, but anyway this height of this tree is log n if there are n element. So, that is why it is called nearly complete binary tree. So, depending on the value of n we will have complete or it is a nearly complete. So, this is a visualization of this array as a tree and this is called heap this visualization this is called heap.

But this is just a visualization in reality we do not have this tree. If we need to do the tree implementation what we need to do suppose in c we want to implement tree.

(Refer Slide Time: 10:41)



So, what we need to do we need to define a structure struct node then we need to have the data or the key value and the we need to have the pointers struct node star left star right may be star parent something like that.

So, this way we just implement a tree in a C language to have a this type, but here we do not need that. So, no pointer is required this is the advantage no pointer is required to have this tree because we know if a node is say I. So, if this 3, 3 node, who are the child for 3 6 and 7; that means, if a node is I, this is a I, this is I, if a node is a I then we know the child's are at A 2 I A 2 I + 1 this is basically 2 I 2 I + 1.

So, this is the formula we use to build A to visualize that tree. This tree is our visualization it is not there actually we are not having this tree we are not having the pointer to have this tree we are just viewing this tree basically we have an array, but we are viewing this tree that this 10 is the child of 10 is the parent of 9 and 3 like this. So, basically we have a array, but we are viewing this as a array as a tree and this is basically called heap data structure. So, this is the A 2 I and A 2 I + 1. So, this is the formula we use for visualizing the tree.

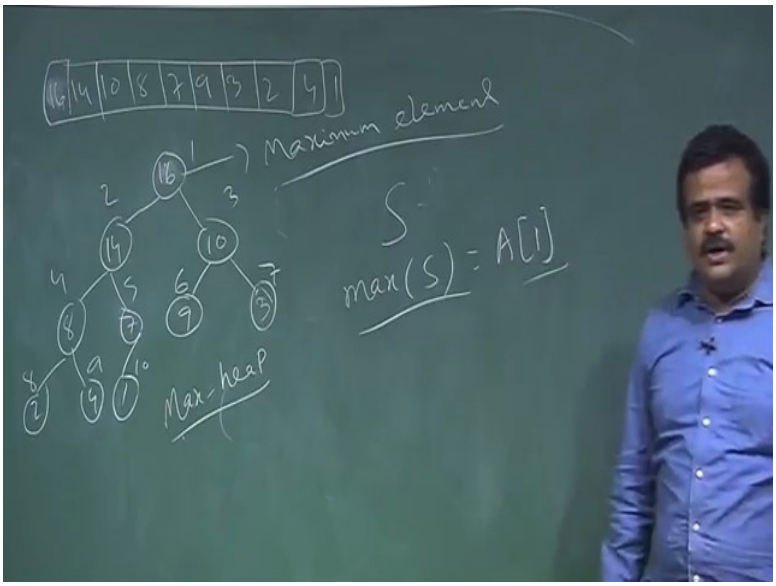
(Refer Slide Time: 12:47)

So, no pointer is required. So, we know where at the child. So, basically left child of left of i is 2i and right of i is 2i + 1. So, this is the formula will give us the tree and the parent of i is basically i/2. So, if by using this formula we will just view this as a tree. So, this is called heap. This array is called heap where we are viewing this array as a binary tree nearly complete binary. So, if there are n element height of the tree is?

So, height is basically order of log n. So, that is why it is a good tree it is a balanced tree. We will talk about balanced tree. So, balance means the height is of order of log n. So, that is why it is a good tree. So, this is called heap this data structure is called heap now when we call this heap as a max heap. So, this is the way we just have child. So if we take this as a i then we know that the children are basically 2i + 1, 2i there is no 2 I + 1.

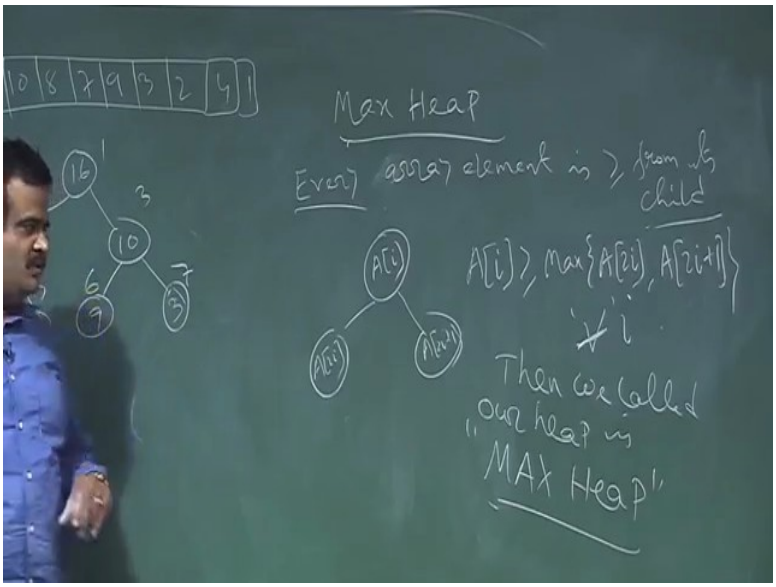
So, this 7 has only one child which is one. So, this is basically the formula to get this to view this tree. So, now, we call this heap to be max heap. So, this is 3 let us have this. So, this is basically the visualization actually we have this array.

(Refer Slide Time: 14:47)



So, let us have the array. So, the array is 16 14 10 then 8 7 9 3 2 4 1 16 14 8 7 9 3 2 4 one this is the array.

(Refer Slide Time: 15:17)



So, when a heap is called max heap the definition of a max heap is if each of the elements is greater than from his child we take an element this is the element say ith element if it is greater than from his child. So, for 8 the children are basically 2i, 2i+1. So, if every array element is greater than or equal to from its child and if this is true for every element so; that means it is a maxheap.

So, if this is A [i] i-th element then the child is basically where A [2 i] and A [2 i + 1] so; that means, if A [i] is greater than equal to maximum of A [2 i] and A[2 i + 1] for all i and if this is true for all i then every element is greater than from his child and if this is true for all i this is this is the symbol for all if this is true for all i then we call our heap is max heap.

So, for max heap this property should be there every element should be greater than from his child. So, is this a max heap array? So, you can verify that.

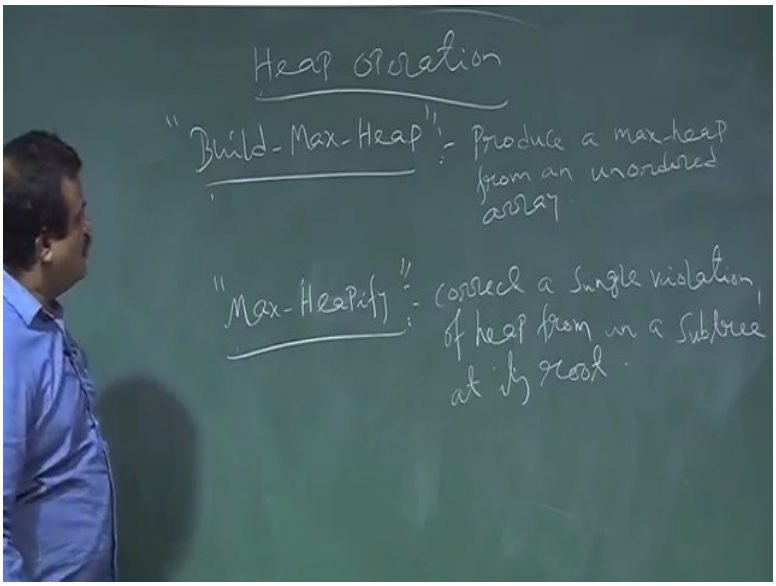
So, this array is a max heap array. So, so this is a max heap array now question is given a array. So, in general we may not have a max heap array. So, given a array suppose we take a input say suppose instead of this say this is 14. So, instead of 14 suppose we have 9 already we have eleven we can put eleven somewhere here. So, instead of 16 suppose this is eleven if this is eleven then this is not a max heap array because this element is not greater than from this.

So, now the question is given a array how we can make a array to be a max heap array. So, what is the advantage of having max heap array? So, advantage is if we have a array which is max heap array suppose this one then we know root content the maximum element. So, that max operation we can easily find out. So, we should able to perform those square is max square is. So, max of S. So, this set is basically S. So, max of S is basically maximum element.

So, this is basically root this is the basically A[1] if the array is a max heap array similarly we can have min heap array in that case the minimum is our priority, min priority. So, we will talk about that. So, if we have min priority then we have the min heap. In the min heap the property will be that all the elements will be less than from his child. So, that will be the min heap property. So, in general we have a unordered array we have given any array how we can make the array to be max heap array or how we can extract the max.

I mean how we can delete it from this array and again make the array to max heap array. So, those operations we have to do. So, we have to perform few heap operations. So, those are basically max heapify and build max heap. So, these are basically heap operation.

(Refer Slide Time: 20:22)

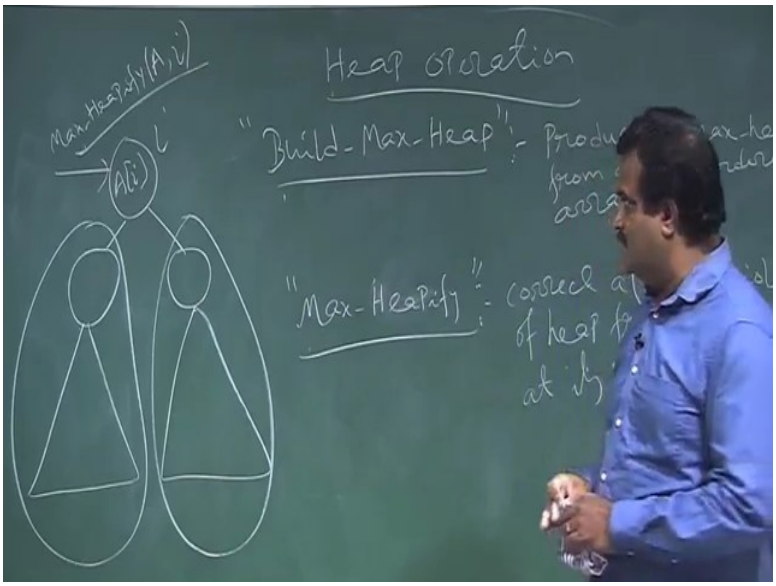


So, basically build max heap. So, this is basically the function which will produce the produce a max heap array from a unordered array.

We are given an array and we have to make it max heap array and that will be done by this sub routine build max heap by this function. So, build max heap is the function or the sub routine which will take the input as any array and it will give it will produce the max heap array. We'll talk about that; this sub routine and in this build for build max heap we need to take help of another function another sub routine which is called max heapify. So, write small or big anyway it should be consistent max heapify.

So, max heapify means it should able to correct only one violation correct a single violation I will explain violation of heap property specially max heap we are talking about it could be min heap also heap property in a sub tree at its root. So, this is a sub routine which we are going to call in this build max heap and this max heapify is basically. So, this is say i th element.

(Refer Slide Time: 22:32)

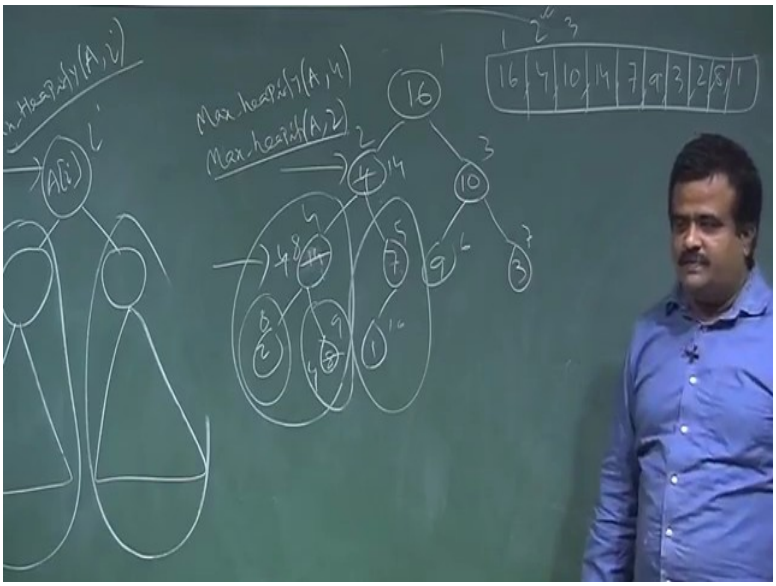


So, maybe we have to call this max heapify of the array at the point i. So, this is the call. So, we will talk about the code for this. So, we will see how this will looks like. So, this is the max heapify. So, we call this max heapify at the point. So, for that what we need to have. So, this is the left child this is the right child. So, this is the left sub array this is the right sub array. So, for this max heapify we assume everything is over here and everything is over here. So, this is this sub array is already max heapify and this sub array is also max heapify only we may have a violation at this point.

So, that is why it can only handle the single violation. So, max heapify can only handle the single violation; that means, everything is fine over here everything is fine over here only we may have a problem with this point. So, then only we can call this max heapify this assumption is very much required to call this max heapify. So, whenever we call the max heapify at that point this assumption should be there that we have to check that whether everything is in the left sub array everything is in the right sub array then only we will call max heapify say this max heapify can handle single violation.

So, that is very important. So, will talk about max heapify and how it will handle this violation through an example?

(Refer Slide Time: 24:24)



Suppose we have this is suppose a given input say. So, we have 4 10. So, 14 7 9 3 and say we have 2 8 and 1. So, what is the array; array is 16 4 10 then 16 4 10 14 7 9 3 2 8 1. So, this is the given array this is the array this is the array and this is the 1 2 3; these are the indexes 4 5 6 7 8 9 10.

So, that there are 10 elements is this a max heap array we can just check whether each element is greater than from his child this is greater than from his child this is greater than from his child this is greater than from his child this is, but this is not greater than from his child. So, we have a problem over here. So, we have to call this max heapify and for calling the max heapify we have to check whether everything is over here this is the left subtree this is the right subtree.

If we just check this is already max heapify this is already max sub max heap sub array this also max heap sub array. So, then only we can call the max heapify at this point. So, that is that is the assumption we need to have. So, we call the max heapify max heapify on at 2 I is equal to two. So, we call this max heapify at this point this is 1 2 3. So, this is the point we call the max heapify remember this tree we are just visualizing mainly we have this array and we can viewing this array as a tree.

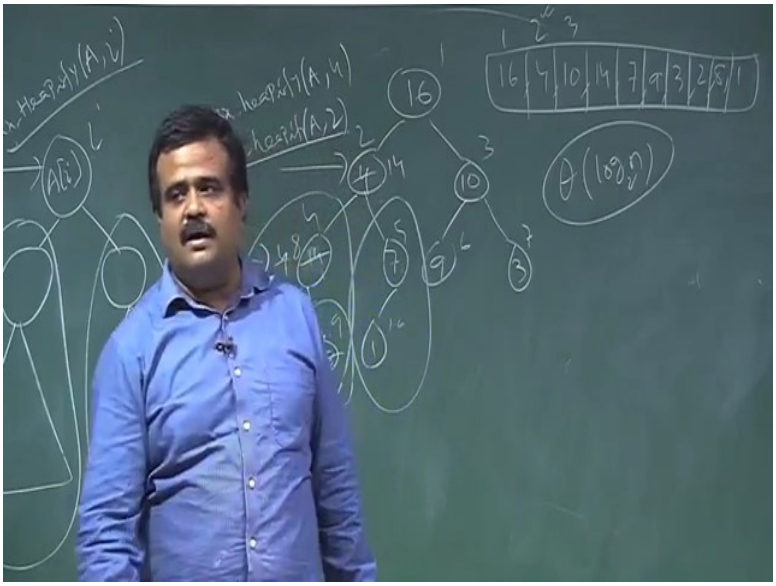
So, now we how we can fix that max heapify sub routine is work like this. So, this has a problem. So, what we do we replace this by the maximum of this two. So, these are the 2 child. So, among this 14 is the maximum. So, we replace 14 with this 4. So, this 14 will. So,

this is 14 of 4 will come here and the 14 will go here now once we put this 4 then it may violate. So, again we have to call the max heapify over here.

So, we call this max heapify it may violate a and where it is going it is going to 4 we call the max heap here at the point 4 now again we check this is already max heapify this is already max heapify. So, we may have single violation yes there is a violation because this is less than 8. So, what we do again we just replace this by maximum of this 2. So, this is maximum. So, this will come 4 and this will be 8. So, now, this is the leaf node. So, we stop, so this is this is the max heapify operation max heapify sub routine.

So, we keep on doing until we reach to a leaf node or it is fixed so, but every time we have to remember this can handle only single violation. So, every time we need to check whether left part is or right part is we may have a violation in that in that corresponding route. So, this is very much required to have and send then what is the time complexity.

(Refer Slide Time: 28:16)



So, it may go up to the height. So, if it is if there are n nodes and if we starting from the root then the time complexity may be the log n for the max heapify.

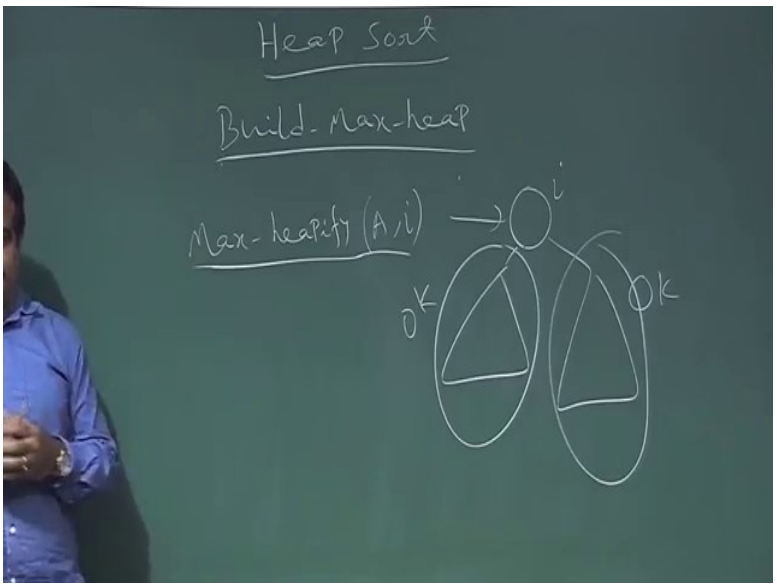
So, it may go up to the height. So, we'll use this max heapify to build a max heap array from unordered array. So, that will be the build max heap. We'll talk about it in the next class.

Thank you.

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

Lecture – 14
Heap Sort

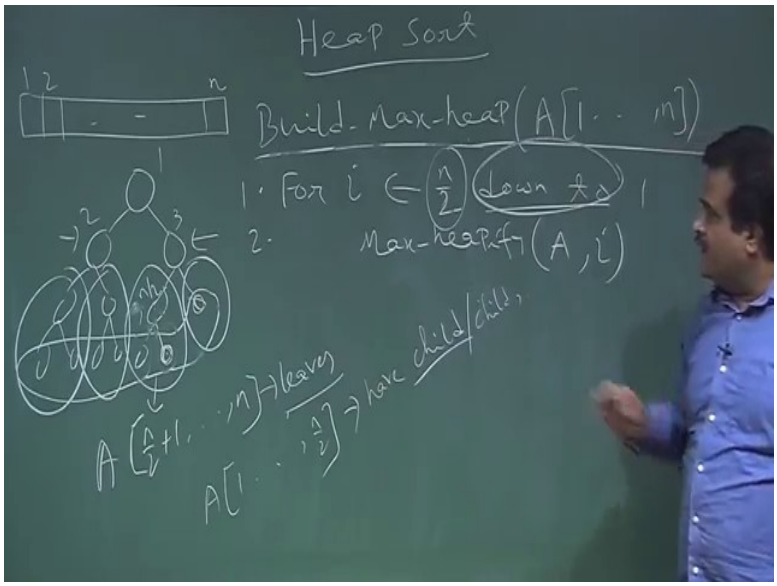
(Refer Slide Time: 00:26)



So we are going to talk about the build max heap that means, so far we have seen the max heapify sub routine. So, this is a sub routine where we call this at i. So, i th node. So, for that we need to have an assumption that everything is ok here and everything is ok here in the left sub array right sub array and only we have a violation over here. So, then we call the max heapify at this point.

So, this will exchange with the maximum of this 2 child and then again we may need to call the max heapify on the exchange node because that may violate the max heap property of that. So, this way we continue. So, we have seen that this is the max heapify code. So, now, we will talk about how we can use this max heapify to build a max heap array we are given an unordered array. So, how to build a max heap array?

(Refer Slide Time: 01:45)



So, that is our next operation which is called build max heap.

So, the input is an array which is basically 1 to n an unordered array and it will output will be a max heap array. So, this code is very simple code this is for i n/2 down to 1 and then we call the max heapify this is for this is for loop for i n by 2 down to 1 then we call max heapify. So, this is the code for build max heap, very simple code only 2 line pseudo code.

So, now this is from n/2 down to 1. So, why it is n/2 down to 1 that we will see. So, why not n + 1 to n; so, basically we want to see who are the element who are the element from n by 2 n by 2 + 1 to n. So, suppose we have array say we are given a array of size n. So, this is our array it is starting from 1 to up to n. So, if you draw the tree. So, this is 1 2 3 like this. So, this is like this, like this.

So, suppose this is the tree corresponding to this array now. So, these leaves are basically nodes from n/2 + 1 to n. So, these are basically leaves these are basically leaf node because they have no child. So, if this node has a child then the child will be a 100 2 i 2 + 1. So, 2 i is basically n + 2 there is no n + 2 or n + so; that means, these are basically leaf nodes.

So, but n/2 has a child because n/2 has if we 2 i. So, n/2 has child at n. So, i is equal to n/2 child has a, so the nodes n/2. So, 1 up to n/2 they have the child all the nodes they have child.. So, that is the reason and if a node is leaf node it is already max heapify leaf node means they

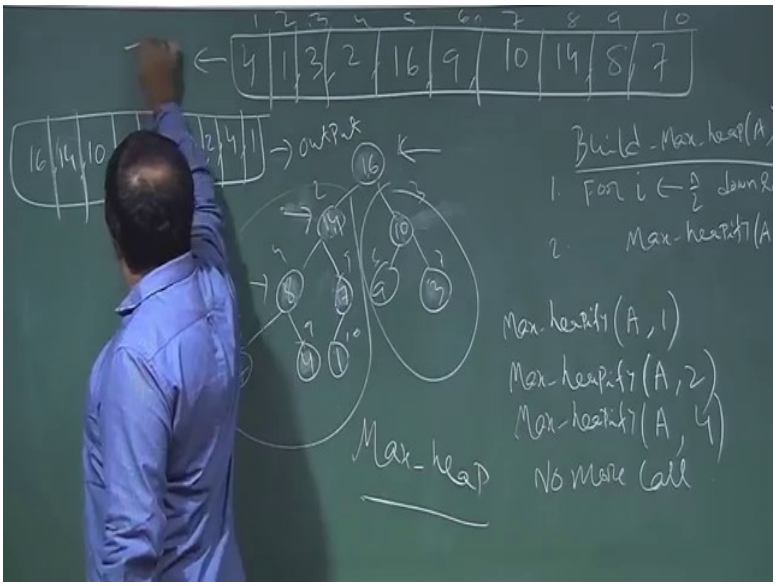
have no child. So, this is already max heapify they are they are already max heapify. So, we do not need to call the max heapify for those.

So, that is why we start this loop from n/2 down to 1. So, we will just start it from this node. So, this is the node n/2 th node. So, we call the max heapify here because other after that these are the old leaf nodes for this nodes; this is already max heapify because these are the nodes only single root there is no child. So, only we may have violation over here. So, that is why we call this loop from n/2 down to 1 why down to 1 because if we call n/2. So, we may have a violation over here so; that means, it may be, but not every time.

So, we'll call max heapify for max heapify we need to have the assumption that left part is right part is. So, that assumption is there because this is the left part this is already max heapify this is already max heapify. So, we call the max heapify here. So, it may only get only max heapify can only handle only one violation. So, that way again we call this like this. So, when we reach here we know that this part we already fixed this part we already fix like this.

So, when we reach here we know this is already fix this is already fix we may got a violation here. So, that we can fix by calling max heapify like this. So, that is why it is down to 1. So, this way we will keep on fixing and we reach to the root. So, this ultimately will be a max heap array of this. So, we'll take an example, this is the basically the code will come to the time complexity of this, but before that let us take an example to execute this code. So, let us take an array this could be any array could be a input. So, let us take an array.

(Refer Slide Time: 06:53)



Suppose we have this array say 4 1 3 2 16 9 10 14 8 7 suppose this is our given array this is the input and we want to make this array as a max heap array, so, now, we just from we just this is our heap. So, we just view this heap as a data as a tree. So, this is 4 this is 1 this is 3 this is 2 this is 16 and this is 9 10 and this is 14 this is 8 and this is 7. So, this is the way we view this as a tree now we call this build max heap.

So, just to build max heap, so this is the code for build max heap. So, for i n by 2 down to 1 here n is equal to 10. So, we have how many element 1 2 3 4 5 6 7 8 9 10 n is equal to 10. So, n by 2 5, this is 1 2 3 4 5 6 7 8 9 10. So, and then we call the max heapify a, i. So, basically these are all leaf nodes. So, this is the first node which is not a leaf node it has its child. So, this is the n/2 th node. So, we start calling the max heapify from this node.

Because after that every nodes are leaf node which is already max heapify. So, no need to call. So, we just start calling the max heapify from here. So, we call max heapify from here we check with this. So, this is get a. So, this is max heapify. So, then we call the max heapify, this is the loop in 2. So, then we once we call the max heapify it will exchange the maximum. So, this 14 will come here 2 will come here then no more call then we start with here. So, 3 will exchange with ten. So, this 10 will come here 3 will be here then we this is our i. So, this will exchange with 16.

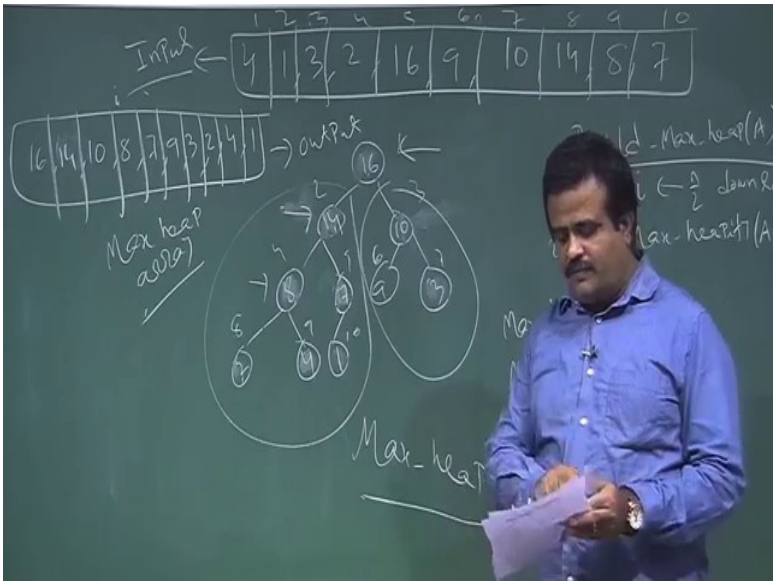
So, 16 will come here and 1 will come here again we have to call then 7 will come here and 1 will be here and then finally, we call this at here now when we calling this if we observe

everything is fine over here everything is fine over here that is why we are going down to 1 because we are keep on fixing and going to ensure that max heapify when you call max heapify we need to have the assumption that left sub array is right sub array is.

Then only we can call this max heapify. So, we call this max heapify here. So, what it will do now it will exchange this maximum with maximum child. So, 16 is maximum. So, 4 will come here. So, 16, this is the call this is the max heapify call a on 1. So, then it will exchange with this then we have to again call the max heapify because it may violate a with 2. Now again it will exchange with the maximum 4 will come here 14 will come here. So, again we have to call the max heapify a comma 4 index 4. So, again it will exchange with maximum.

So, 8 will come here 4 will come here then this is the leaf. So, no more call no more call. So, this is the max heap array. So, then I exhaust. So, these array is a max heap array. So, our array is change now this is the input and the output array will be just 16 14 10 16 14 10 then 8 7 9 3 2 4 1. So, this is the, this is the output.

(Refer Slide Time: 11:56)



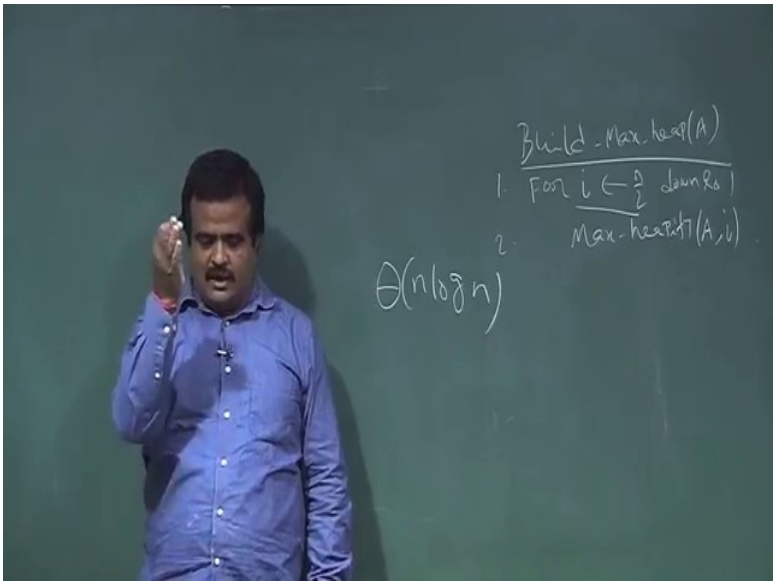
This is the output after build max heap and this is the input array. So, this is unordered array which is not a max heap array.

But we can make this array to be max heap array and this 3 we are viewing this is we are our visualization in reality this 3 does not exist we do not have pointed to implement this tree to view this tree we know that if a node is i then the child will be i + i 2 i 2 i + 1. So, that is the

way we are viewing this tree. So, this is a max heap array because if we check any node i it is greater than from his child. So, any node i is greater than from 2 i 2 i + 1 any node this is the array if we take any node i then it is greater than 2 i 2 i + 1.

So, that is why this heap is a this is a max heap array and this is the output of this build max heap now the question is how what is the time complexity of this build max heap. So, that we have to see. So, that we have to calculate the time complexity. So, how much time we are spending in build max heap. So, so what is the intuition? So, it is for loop it is for loop of size n and each time we are calling the max heap now.

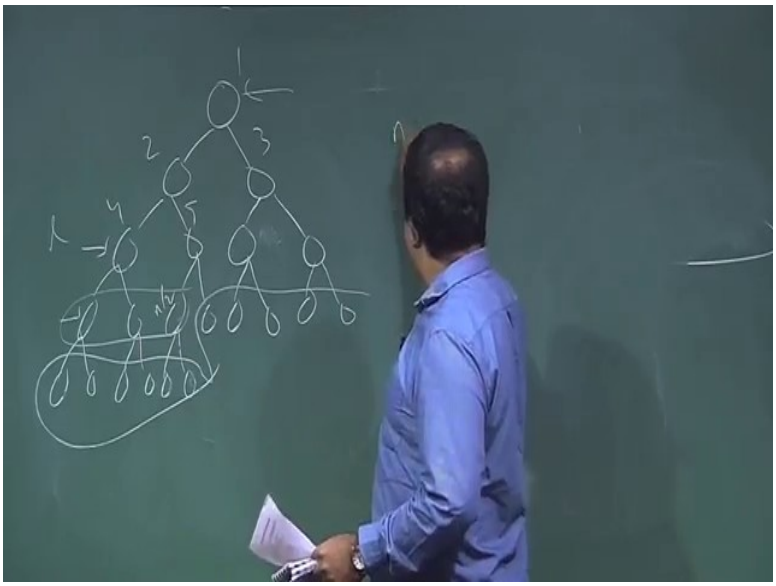
(Refer Slide Time: 13:23)



So, intuitively it should be order of n log n because each time we are calling max heap and that max heap can be again further call an consist of further call this is a recursive can come because it may fix there and come down then again. So, maximum it can go to the root height of the tree. So, that is why it is n log n, but this is the intuition, but we really it will take order of n. So, that analysis we will do.

So, it is not really n log n because this max heap call when you are doing it is really not going to the height of the tree for each node why? Suppose this is our array 2 3 4 5 say 6 7 like this.

(Refer Slide Time: 14:14)



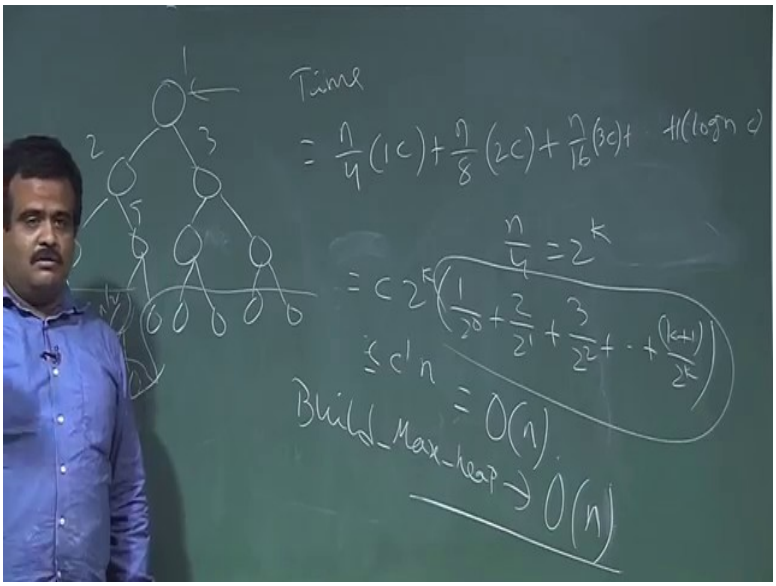
So, this is the say tree and this is the node n/2 i mean. So, let us draw once more now these are all leaf nodes and this is the n/2 th node, now we are calling the max heapify starting from n/2 now for these nodes the max heapify call is only once.

Because this node will compare with the maximum of this 2 then we may have to exchange then we stop there is no further call. So, the node which are in this level 1 level above from the low level 1 level above the leaf level these are all leaves 1 level above the leaf level for them. The number of call is one, this will be one call only because it will exchange with this and then stop because we are reaching to the leaf, but the node which are here for them we call the max heapify it may exchange this and then again we may have to call. So, it may go this for them 2 max heapify call for them.

But 2 if they are 2 level above from the leaf so, that is why it is not log n for all it; it will be log n for the root if we call the max heapify here this max heapify call for this node may come down and consist of log n when we call for with this max heapify. But not for the all I mean not for all the nodes we will have log n many max heapify call in the worst case; that means, if a node is lies in l level from the bottom then it may cause us the order of l.

So, that is the analysis we have to do. So, the number of nodes which are just one level above for them number of call is just one max heapify and what is the number just one level above. So, that number is n by 4.

(Refer Slide Time: 16:39)



So, n/4 into 1c and then 2 level above n/8 into 2c for which are the nodes which is 2 level above from the down from the for them we may have to call twice. So, this may be exchange with this again we may have to call this max heapify like this. So, 2 c + n by 16 into 3 c this way, but for the root we need to call 1 into log n.

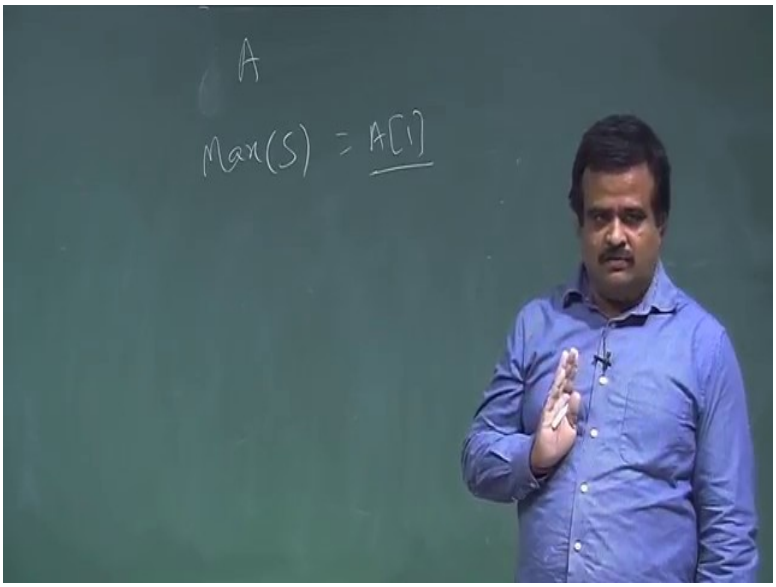
So, that is. So, log n log n c this is for the root. So, this is the time. So, not for every node we are spending log n time. So, this is important. So, this if we have to simplify. So, this is the time complexity for build max heap. So, if you simplify this. So, this will be basically c into. So, if you take n by 4 is equal to 2 to the power k just we take n by 4 equal to then this is c into 2 to the power k in common. So, 1 by 2 to the power 0 + 2 by 2 2 to the power 1 + 3 by 2 square like this + k + 1 by 2 to the power k this way.

Now, this term is bounded this can be proved this is g p mix series I mean finite series. So, this is bounded by a constant. So, this is basically less than equal to c some another constant c prime into n. So, this is basically order of n logarithm. So, build max heap will take build max heap will take sorry n linear time and that is why it can it will give us a sorting algorithm build max heap will take linear time it is not n log n because for not for all node we are spending log n call.

Because the nodes which are just 1 level above for them just a one call and that is 1 c and number of such node is n/4 the node which are just 2 level above and that is 2 c a number of nodes is n/8 like this. So, this is the time complexity for build max heap. So, now, we will

talk about how we can use this 2 have a heap sort how we can use this max heap array or max heapify this. So, this is the way this is basically nothing to do with sorting still now we have not talk about sorting yet.

(Refer Slide Time: 19:44)

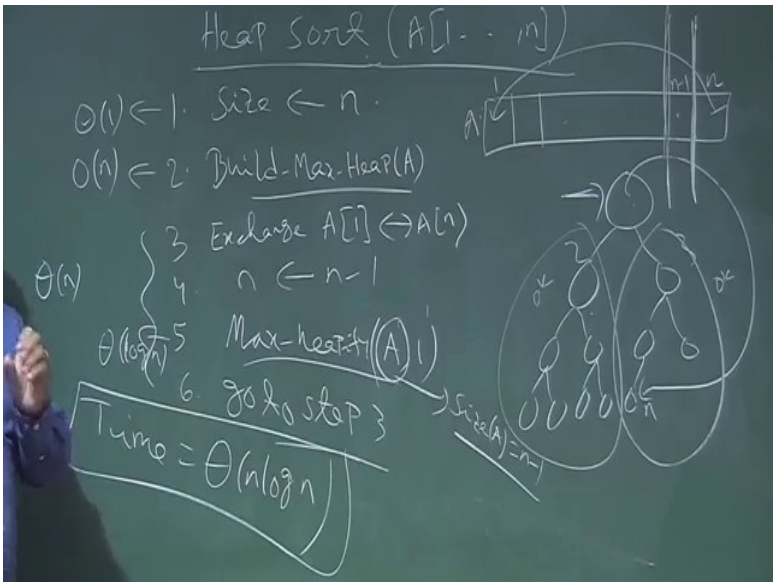


So, this is basically priority queue and this is basically to maintain a set S where we should perform this type of operation like we should be able to return the maximum. So, max of s. So, if we have an array and if we make it max heap array then max of S is basically the first element after making the max heap after calling the build max heap and then the extract max extract max means we must delete it. So, we just extract exchange and then again we call the max heapify.

So, again it will make the max heap array like this. So, that is the operation we can do. So, another operation is increase key. So, we can go to that particular position we can change the value and if we change the value. So, again to make it max heap array we need to call the max heapify at that position. So, that may cost us again logarithm time depending on which position we are calling. So, worst case it may come from root to the height of root to the leaf. So, that is the decrease key operation. So, these are the operation we must able to perform.

So, this heap data structure is used as an implementation of the priority queue now we will talk about how we can make use of this max heap data structure to have a sorting algorithm. So, that is called heap sort.

(Refer Slide Time: 21:19)



So, now, we will talk about the heap sort which is basically using this max heap array. So, the heap sort. This is the sorting algorithm. So, it is taking an input of size n. So, what we are doing we are taking size of this array to be n.

So, we have a array of size n. So, this is our input. So, size is n we are just taking the size now we are calling the build max heap in a first sort build max heap on A. So, this is making the array as a max heap array. So, we have any unordered array and it is making the array to be a max heap array. So, once we have a max heap array we know the maximum is rooted here.

So, this is 1 2 3 like this. So, this is n. So, this is after calling the build max heap and this is max heap array. So, this is the maximum. So, now, this is our situation. So, this array will change to the max heap array. So, this is the situation now what we do we know this is the maximum. So, we just exchange we just send this guy to the end we know this is after calling this build max heap this array. So, we know this is the maximum. So, we send to the end and we reduce the size of the array.

So, we exchange this we just exchange; exchange A 1 and A n. So, whatever value over here we send it here this is the max heap array after calling this build max heap whatever value over here we send it here and we bring this value over here now once we bring this value over here. So, this may violate the max heap property. So, again we mean again to make it

max heap array sub array because we just disrupt this we forget this element because this is the element which if we sort it this is the position where the maximum will be sitting.

So, we just forget that. So, we just do what exchange this and then we reduce the size by 1 size - 1. So, n - 1 basically or we can just say n is equal to n - 1 yeah size - 1 or n n we replace by n - 1 so; that means, we just forget this part now our array is up to n - 1 now again we call max heapify max heapify a at the point.

This, but this a array is now size is size of a is now n - 1 not n that we have to remember we call the max heapify at this point because this as exchange with this element. So, it may violate the max heapify property here and then we can remember this is already this is because this was the max heap array. So, we may have violation here. So, this will keep on fixing and again after this call it will be again a max heap array. So, then again we will do this same thing. So, we can just put n is equal to n - 1 here instead of size.

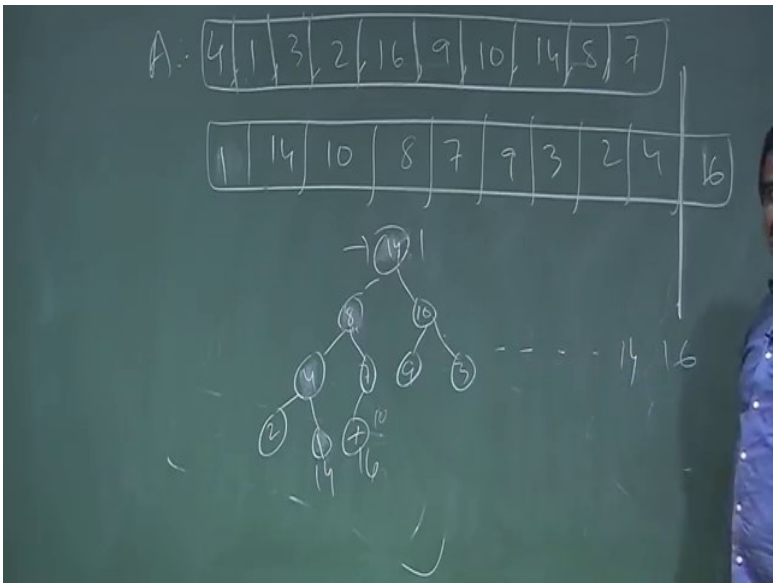
Again we go to this 1 go to step 3 and this will continue until n is equal to 1 once n is equal to 1 we stop. So, basically then again we put this n - 1 here. So, this is the second maximum and then we disrupt this element from here. So, this is our sub array then this way we continue until we reach to the n is equal to 1. So, this is the way we keep on reducing the size and we fix the largest element this way and every time when we exchange this with the last element then that may violate the max heapify property.

Then we have to fix by calling the max heapify we will take an example, but we further let us talk about the time complexity. So, this is theta 1 and then build max heap will take linear time this is once and then we are exchanging this and this is a loop of size n and each time we are calling the max heapify property. So, this is the log n time will take because it may go. So, the time complexity is basically n log n. So, this is n log n time algorithm and this is an in place sorting algorithm.

Because we are not taking help of any extra memory everything we are doing in the array. So, this tree just we are viewing the tree we are visualizing the tree in reality this tree does not exist we are not using any pointer to build this tree. So, we are just viewing this tree. So, everything we are doing in this array given array everything. So, that is that is why it is called in it is a in place sorting algorithm. So, we are not taking any extra storage to have this. So, let us take a quick example quick execution of this.

So, this is a n log n time algorithm and this is called heap sort some quick example on this heap sort. So, suppose we have say this is the input say yeah.

(Refer Slide Time: 27:55)



So, say suppose this is our given input this is we have 14 8 7. So, this example we have taking to get the build max heap. So, this is our given array we need to sort it. So, what we do we first make it a max heap array?

So, we called the build max heap if we call the build max heap it will be like this 16 14 10 this part we have seen 8 7 9 3 2 4 1. So, so this is the build max heap. So, if you draw the tree. So, 14 10 8 7 9 3 2 4 and 1 this is the tenth node. Now what we do after build max heap we exchange this and this. So, we exchange 14 and 16.

So, this 16 will come here and 1 will be here. So, this 16 will come here and 1 will be here and we forget this part now the array is reduce by 1 size 1. So, again we call build max heap here. So, it will exchange this maximum of this 14 will. So, 14 will come here and 1 will be here again we have to call build max heap. So, 1 will come here 8 again we have to call build max heap. So, one will come here and 4 will go here we stop. So, again this is a max heap array max heap sub array.

So, again we exchange this here with the low this is this is again will exchange with this. So, 14 like this. So, 16 14 like this, it will be we continue this until it will end to the all the

exhaust the all the array element. So, this is the implementation, this is the execution of the heap sort.

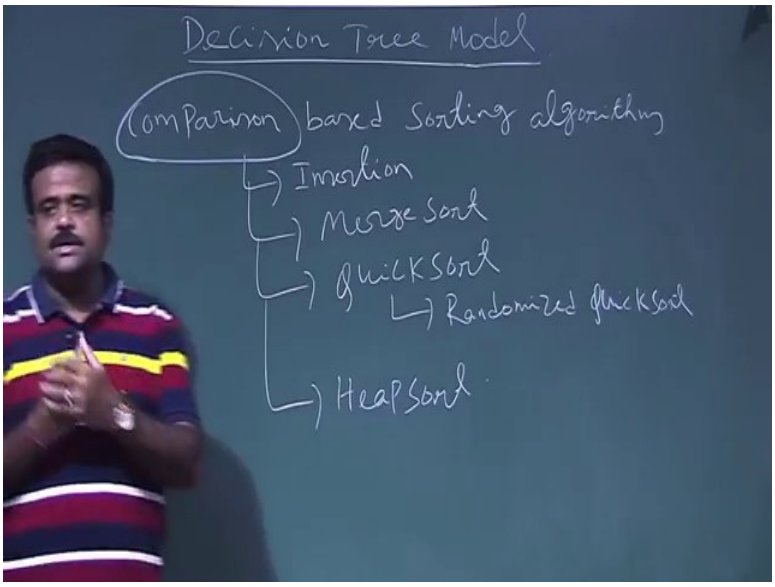
Thank you.

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

Lecture – 15
Decision Tree

So, far we have seen the sorting algorithm which are basically comparison based sort so; that means, we compare 2 elements to know their ordering.

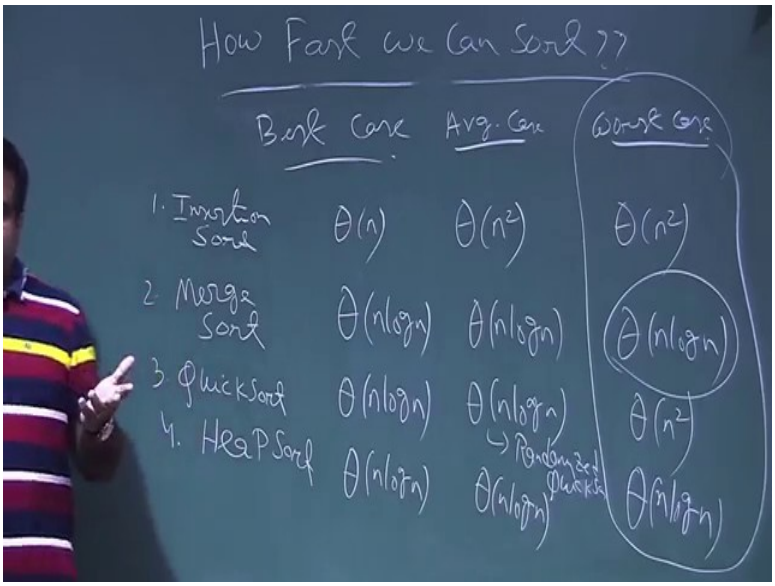
(Refer Slide Time: 00:38)



So, basically we compare the elements to get the ordering. So, those are basically comparison based sort. So, like we have seen insertion sort, then we have seen the merge sort, then we have seen the quick sort.

And a version of randomized quick sort, and also we have seen heap sort. So, all these sorting algorithms we have seen are basically comparison based sort so; that means, we compare 2 elements to get their relative ordering. So, these are comparing the input elements. So, these are comparison based sort. So, what is the time complexity for this sorting algorithm?

(Refer Slide Time: 01:57)



So, the question is how fast we can sort? So, that is our topic today I mean. So, how fast we can sort?

So far these are the sorting algorithms we have discussed. So, let us have best case, average case, and the worst case time complexity of this. So, we talk about we have seen insertion sort. Insertion sort best case we know it is a order of n; that means, when it is when the array is already sorted. And that will give us the best case, and average case we have seen is also order of n square, and the worst case is also order of n square.

And then we talk about merge sort, merge sort we know all the cases it is basically order of n log n. Because it does not matter whether a input is sorted, reverse sorted we go to the middle we just partition, it we just go to the middle we divide into 2 sub array we sort this sub array, sort this sub array then we call a merge sub routine. Then it is a all the cases is order of n log n. And it is not an inplace sort because to call the merge sub routine we need to have a extra array.

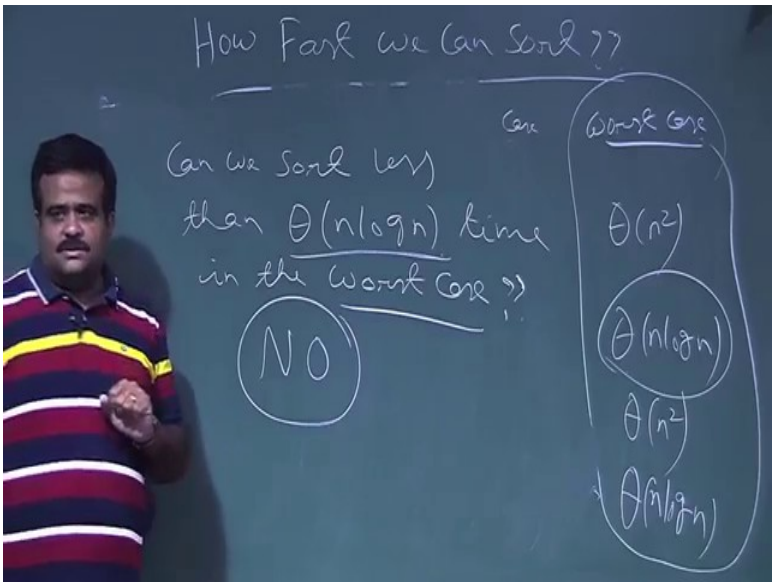
So, then we discuss the quick sort. And the best case we have seen it is n log n, when we choose the pivot is maximum or minimum always. And the average case analysis we did for randomized version of the quick sort which is the expected run time. And then the worst case we know order of n square for quick sort because, in worst case always we choose the pivot, best case we do not choose the pivot as minimum or maximum, but for

the worst case we always choose the pivot to be maximum or minimum. Then it will be partition is 0 is to n minus 1 always.

And then we talked about the heap sort in the last class, which is also all the cases it is order of n log n. So, basically we make use of the data structure heap. So now, the question is, we are basically concerned about worst case. So, our question is, how fast we can sort in the worst case? Because worst case is the most usual case one should go for, because it is give us a guarantee. So, the question is how fast we can sort? So, so far we have seen this is the best way, the order of n log n which we are achieving by merge sort or in the heap sort in the worst case. And merge sort is not an inplace sort, but heap sort is an inplace sort, we are doing in that array we do not need any extra storage for heap sort.

So, now the question is this the faster way or we can reduce the worst case time complexity further? So, the question is, can we sort, sort less than order of n log n time In the worst case worst case?

(Refer Slide Time: 06:03)

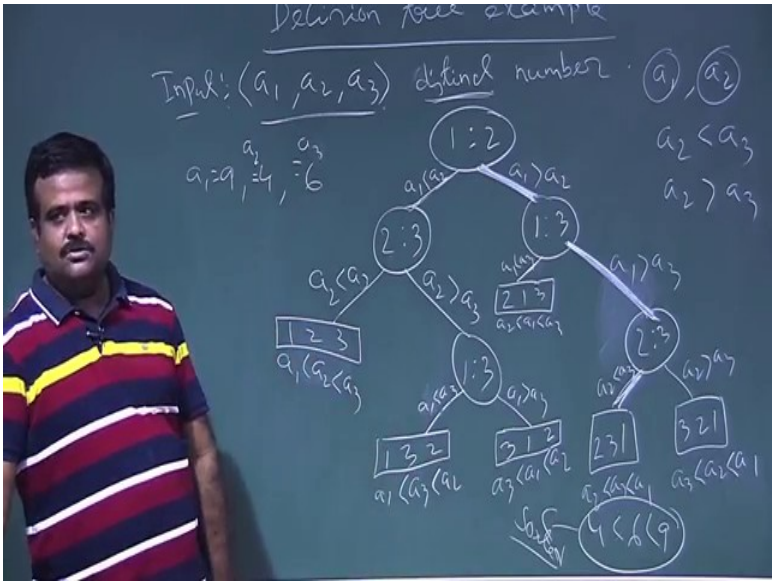


So, that is the question in the worst case. So, so the question is this the best one in the worst case scenario or we can have some better algorithm which will give us the worst case time complexity better than, order of n log n or better than order of n log n means it could be linear it could be log n, it could be something like that. So, it is should be time complexity should be lesser than less than this. So, that is the question.

So, the answer is if we are using the comparison based sort so; that means, if we are comparing 2 element to get their relative ordering then the answer is no. So, there is no comparison based sort which can give us the worst case time complexity lesser than this. So, for any comparison based sort if we consider that will give us the worst case I mean this is the lowest one. We cannot reduce this time further. So, that we have to convince by the help of what is called decision tree. So, we'll take the help of decision tree model to convince this answer that if we use some comparison based sort, any comparison based sorting algorithm we cannot do better than n log n in the worst case.

So, let us convince this with the help of a decision tree. So, what is the decision tree? So, basically, decision tree example.

(Refer Slide Time: 08:03)



Suppose we have given 3 number to sort. a 1, a 2, a 3, 3 numbers are given a 1, a 2, a 3 and we need to sort. So, what we do we will form a tree decision tree. So, for that for making this tree we assume this number at distinct all are distinct. So, we have 3 distinct numbers ok now how to form a decision tree. So, first we compare a 1 with a 2. So, 1 is to 2, this we write in this way. So, we are comparing a 1 and a 2. So, a 1 is a number a 2 is a number and these two are distinct. So, if we compare two numbers which are distinct then what are the possibilities? We have only 2 possibilities, because we are not taking them to may be equal. So, equality possibility is gone.

So, we have 2 possibilities, either a 1 is less than a 2 or a 1 is greater than a 2. So, if a 1 is less than a 2 will go to some branch left branch. And if a 1 is greater than a 2 we will go to right branch and we will do the subsequent comparison again. So, this branch will go if a 1 is less than a 2. And this branch will go if a 1 is greater than a 2. There is only 2 option because, these are distinct. Now if you follow this branch again we compare a 2 and a 3. So, again if we compare a 2 a 3, so there are 2 possibilities either a 1 a 2 is less than a 3 or it will greater than a 3.

So, again we have 2 possibilities. So, this way a 2 is, so we will go for further comparison if a 2 is this is the branch for to follow if a 2 is less than a 3, and this is the branch if a 2 is greater than a 3. Now if we reach to this branch a 1 is less than a 2 a 2 is less than a 3. So, this will reach to a decision. What is the decision? This is telling us a 1 is less than a 2 less than a 3, this is the decision.

So, this we denote by 1, 2, 3. Suppose we have a example such that a 1 is less than a 2 then we will follow this branch again we compare 2 3; a 2 is less than a 3 then this is the branch we follow and this will reach a decision. So, these are all this is the leaf node. Now here if we go for this branch, we have to compare again a 3 with a 1 with a 3 to reach to a decision. So, if you compare 2 number again. So, we have 2 possibilities a 1 is less than a 3 or a 1 is greater than a 3. So, if a 1 is less than a 3, So we know the a 1 is less than a 2 and a 2 is greater than a 3 and here, a 3 is greater a 3 is greater than a 1.

So, this is giving us the decision like 1 3 two; that means, a 1 is less than a 3 less a 2. Because a 1 is less than a 3 coming from here and then a 2 is greater than a 3 and then again a 1 is greater than a 3. So, if we follow this branch it is reaching us to a decision that, a 1 is less than a 3 less than a 2. And similarly if we follow up this branch then it will reach to a decision 3 1 2; that means, a 3 is less than a 1 less than a 2, ok.

So, if a 1 is greater than a 2 then we have to do further comparison. So, we compare a 1 and the a 3. So, if a 1 is less than a 3 and we know the a 1 is greater than a 2, So this will reach to a decision 2 1 3 so; that means, a 2 is less than a 1 less than a 3. Now if we have to follow this branch, so then we have to compare again a 2 and a 3. So, these will reach us to a, so this is basically, this should come down here.

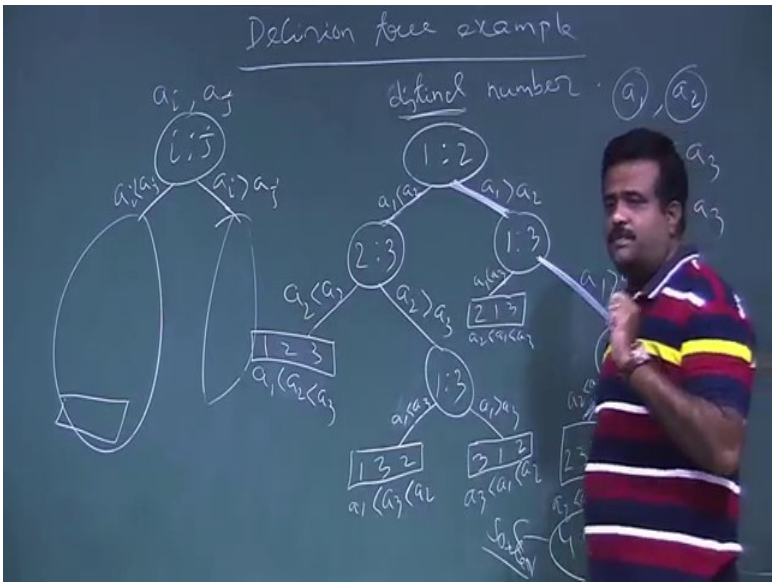
So, this is basically a 2 a 3. So, this is giving us 2 decision. So, if a 2 is less than a 3. So, this is basically a 1 is greater than a 3, and this is a 1 is less than a 3. So, this is reaching

to a decision 2 3 1 and this is reaching. So, this is a 2 is greater than a 3. This is reaching to a decision 3 2 1. So, this is basically a 3 is less than a 2 less than a 1 and this is a 2 is less than a 3 less than a 1. So, this is this is called decision tree.

Now, we can take an example. Suppose we take say 9, 4, 6 suppose this is our a 1, this is a 2, this is a 3. a1 is 9, a2 is 4, a3 is 6. So, we want to follow this model. So, now, we start with here. So, this is the model we have this tree we have. This is the decision tree. Now first we compare a 1 with a 2. So, a 1 is 9 a 2 is 4. Now 9 is greater than 4, we must follow this part, and then because we have to follow this part because 9 is greater than 4. Then we have to again do the subsequent comparison to reach a decision. These all leaves nodes are basically giving us the decision. So, they are basically permutation. So, leafs are basically all possible permutation. So, there are 3 nodes. So, there are factorial 3 permutation, so factorial 3 6, 6 permutation ok.

So, here 9 is greater than 6. So, we follow this path, then again we after reaching here we have to compare a 1 a 3. So, a 1 is 9 a 3 is 6 we compare 9 and 6. So if so, 9 is greater than six so again we follow this path. Now again we have to compare a to a 3 a 2 is 4 a 3 is 6 now 4 is less than 6. So, we follow this path and we reach to a decision, what is the decision? That a 2 is less than a 3 is less than a 1. So, what is a 2? A 2 is basically 4, 4 is less than a 3 is 6 and less than 9 which is correct basically. So, this is sorted. So, every path is if we so for a given input basically we are following a path to reach to a decision. So, to reach to a to get a sorted array. So, this is our input we need to sort it. So, we start with the root and we compare 2 element and then based on the comparison either if these are distinct element.

(Refer Slide Time: 16:35)



So, basically what we have we have say nodes are basically i to j. So, basically here we are comparing a i and a j. And these are distinct nodes; distinct element. So, if we compare 2 element then a i is either less than a j or a i is greater than a j.

So, if a i is less than a j, then we will go to the this sub tree. And we will do the subsequent comparison until we reach to a decision. And then if a i is greater than a j we will go to this sub tree right sub tree until if and we keep on comparing again until it reach to a decision, until it reach to a leaf node and every leaf contain a permutation. And that is the ordering between the elements. So, any comparison based sorting algorithm we have seen is basically, execution of this decision tree.

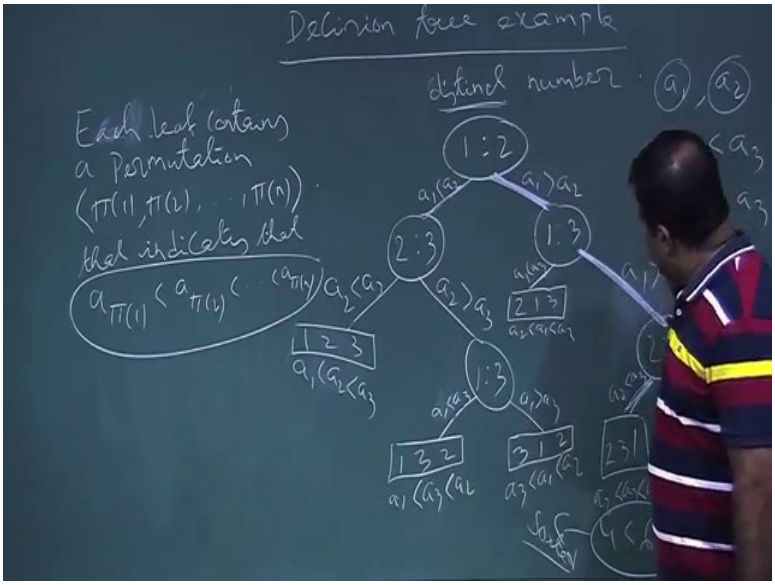
So, we basically every comparison based sort if you compare, if you think like insertion sort. Insertion sort what we are doing? We are comparing 2 elements and then we are taking a decision. So, basically same thing, so any comparison based sort basically we are comparing between the elements and we are going and further comparing, again we are further comparing and finally, we are reaching to a decision finally, we are reaching to a permutation which is giving us the ordering of that element. So, this is basically the decision tree model.

And which is basically the any comparison based sort execution of any comparison based sort is basically the path execution of the path of the decision tree from root to the leaves. So, here any node is basically of this form i is to j. So, this is basically comparing

the a i with a j now if a i is less, than a j we will go for this path and we do subsequent comparison and if a i is greater than a j we go for this path, we do the subsequent comparison. So this is the model of decision tree and any comparison based sort is basically fit under this model, ok.

So, and what are the leaves?

(Refer Slide Time: 19:05)



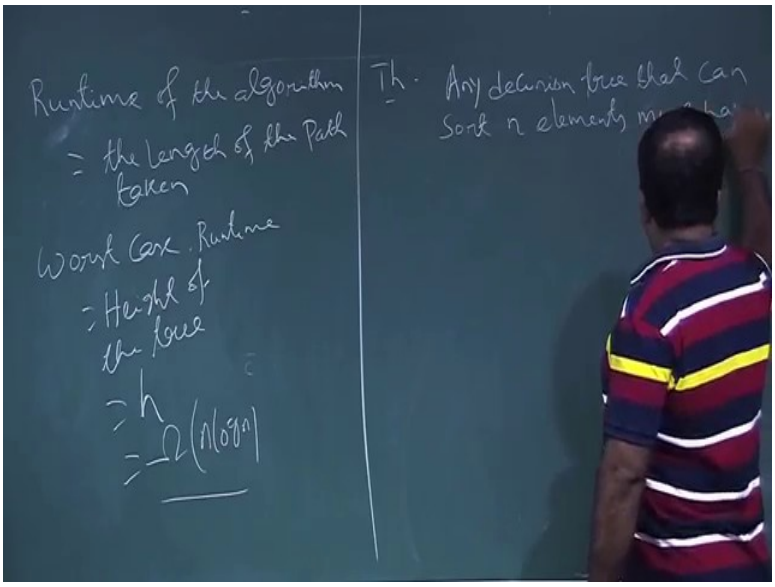
So, each leaf contain a permutation say, pi 1, pi 2, pi n. That indicates that, they are in n that order, that indicates that a pi 1 is less than a pi 2 less than dot, dot, dot a pi n. So that means, they are in the order. So, we reach to a decision. So, this is a decision, this is a decision depending on the input we reach to that particular I mean, one of this part, one of this leaf and that is a decision.

So, if each leaf contain a decision. So, basically any execution of any comparison sort is basically we just execute the root to the leaf, so basically the run time. So run time depend on the length of the path. Now here if the input is say 9, 4, 6 then we follow this path. Now if the input is a instead of 9, 4, 6 if the input is say 9, 6 and then say 10, then what we do? So, we just this is our a 1, a 2, a 3. If the input is this then what is the path? The path is basically; let us erase this earlier one.

So, path is basically we compare a 1 with a 2. So, we follow this path then we again compare a 1 and a 3, a 1 is 9 a 3 is 6 so we follow this path. So, we reach to a decision,

what is the decision? A 2 is less than a 2 is 6, which is less than a 1 and which is less than 10 which is correct. And length of this path is small. So, depending on the input I mean, the time complexity will be the length of the path decision tree. So, let us write this.

(Refer Slide Time: 21:58)

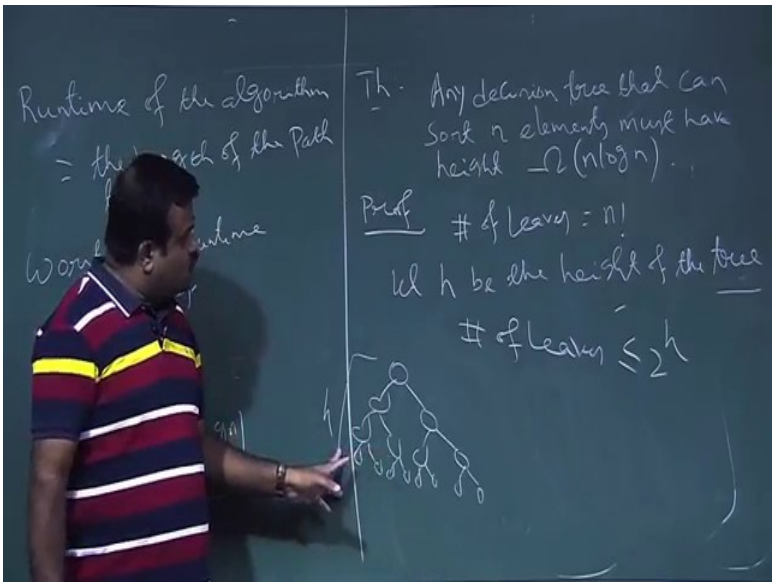


So, the run time of this decision tree model of the algorithm is equal to the length of the path taking.

For this input we go to the less length, but then what is the worst case run time? Worst case runtime will be the height of this tree, because that is the worst case, worst case means maximum path we follow. And that is the height of the tree. In the worst case, worst case means we have to follow the root to the maximum length part. And that is the height of a tree. So, the worst case run time is basically, height of the tree. And this is denoted by h and we prove that height of this tree is n log n.

And that will give us the answer that in the worst case any comparison based sort we cannot have a comparison based sort which can be better than n log n. So, let us talk about how the worst case, how the height of this decision tree is n log n ok.

(Refer Slide Time: 23:52)



So, this we prove by a theorem. So, this theorem is telling any decision tree that can sort n element must have height. So, height must be bounded by n log n lower bound. So, height must be greater than equal to n log n.

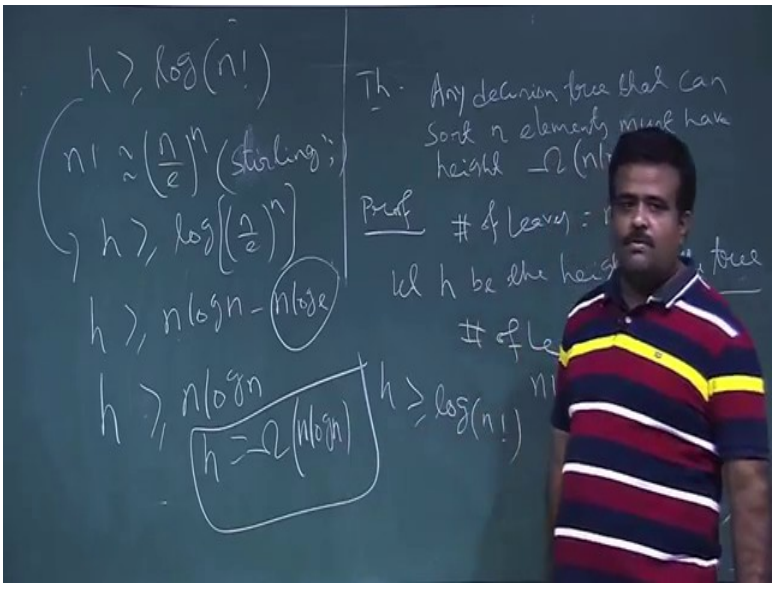
So, these we have to prove. So to prove that, suppose we have n elements. So, in the example we are having 3 elements and we form the tree. Now suppose we have n element and we form the tree. So, so what is the number of leaves? So, number of leaves is basically all possible permutation, basically they have the decision we do not know which input will come. So, the number of decision will be, so we should have a decision for all type of inputs so; that means, and each leaf is basically the decision. So, for a given input we execute the path and we reach to a decision.

So, number of leaves must be factorial n, and this is the all possible permutation I mean permutation of n inputs. So, this is the number of leafs. Now this is the binary tree so; that means, we have each node have 2 children nodes. So, if the height is h, let h be the height of the tree. Then 2 to the power h must be greater than equal to factorial n because, then the number of leaves must be, number of leaves must be 2 to the power h. So, if we have a complete binary tree like this. Then this is the height means maximum depth if the height is h. So, at this level what is the number of nodes? 2, this level 2 square. So, if the height is h. So, if is a complete binary tree then the number of leaves will be 2 to the power h, but it is not a complete binary tree. There are some leaves some

branches are ending fast so; that means, if it could complete then it is 2 to the h so; that means, the 2 to the number of leaves must be less than 2 to the power h

So, from here we know the number of leaves is factorial n it is basically 2 to the power h.

(Refer Slide Time: 27:05)



So, now from here we can just write h is basically greater than equal to log of factorial n. Now, we will use a formula which is stirlings formula. So, let us. So, what we have? We have h is greater than equal to log of factorial n. Now factorial n can be approximated by, (n/e)^n. And this is called stirlings formula.

So, if we use here this, so h is greater than equal to log of (n/a)^n. So, this is basically h is greater than equal to nlog(n) - nlog(e). Now this is a low ordered term we can ignore this. So, height is basically bounded by n log(n). So, height is omega n log n. So, the height of a decision tree cannot be less than n log n it is a lower bound by n log n.

So, big omega(n*log(n)) so; that means, worst case run time of any comparison based sort is bounded by lower bounded by n log n.

So, we cannot, we cannot sort n element faster than n log n if we are using comparison based sorting algorithm. And this we have seen by the help of decision tree. So, next class we will talk about linear time sorting algorithm, but those are not comparison based sort, we are not comparing the elements we are just seeing value of the element and we are putting into the bucket.

Thank you.

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

Lecture – 16
Linear Time Sorting

So far we have seen the comparison based sorting algorithm. And we have proved that by the help of decision tree that if you are using comparison based sorting algorithm we cannot go faster than the n log n. So, now, we talk about linear time sorting algorithm where we will not do comparison between the elements, but we will see the value of the element based on that we will do the sorting.

So, we will discuss 3 such sorting algorithm one is counting sort or radix sort bucket sort. So, let us start with the counting sort.

(Refer Slide Time: 01:04)

So, this is an example of linear time sorting algorithm. So, but this is not a comparison based sort, that means, no comparison between the element. So, we are not doing any comparison between the elements.

So, that is how you could reduce it to the linear time. So, this is, so for this we need to have some assumption like, suppose this is our number.

(Refer Slide Time: 02:11)

So, this is the input, what are the input? Input as say A array of numbers. And here we are assuming the value. So, we have to concern about the value of this number. So, we have to bound this numbers are coming from this range.

So, we have to fix the range of this number. So, that is also part of the input. So, that is basically where A i is coming from k. So, k is the maximum value we are allowing this number to take. So, this is also part of the input so; that means, we are fixing the range of the input. So, that is one of the criteria of this sorting algorithm. So, we have to mention the range of these numbers, otherwise we cannot sort this. Because basically based on that we need to take the auxiliary array.

So, output will be in a B array which is basically sorted. What we need to take a, extra storage or extra or auxiliary storage here. So, these are basically you can say bucket auxiliary storage. So, this is denoted by C and this will be based on this range k. So, based on the value k we need to take this extra memory. So, this is not; obviously, inplace sort. So, this storage also is a part of the input.

So, we need to take extra storage. So, this is the range has to be fixed at the time of the input. So, once we fix the range then we can allocate the memory or the storage for that. So, this is the input output and the auxiliary storage. Now we have to write the pseudo code for this counting sort. So, so basically the basically, we are going to sort this number into the B array which is also of size n, but with the help of a extra another array

which is the auxiliary array C array whose range is from one to k, and k is the maximum value of the input we are allowing the size, not the size the value input can take. Say for example, if we have a array, A array say 2, 4, 5 and then say 11. So, if this is our input. So, now, our k is basically 11. So, basically our numbers are coming from 1 to 11. So, k is the maximum value we are allowing as the input. So, we are having a range of the input. So, we are bounding the input to come from this range.

(Refer Slide Time: 05:56)

.

So, that is one of the restriction, or one of the criteria for this so; that means, if k is 11 then. So, this we are going to sort in B array. So, B is also this B array and, but we need to take a C array of size 11. So, so C array is of size 11, C array will be 1, 2, 3, 4, dot, dot, dot, 11 - 1, 2, 3, 4, 11. So, this is the auxiliary array we are going to, we need to take for this algorithm. So, now, let us write the pseudo code for this counting sort. So, this k is the range of the input. So, counting sort ok.

(Refer Slide Time: 06:48)

So, first of all we have to fill the C array. We have to initialize the C array by 0, C is just a frequency. Or we can say C are the bucket and we can fill the bucket like this. Anyway, so we will come to that. So, for i is equal to 1 to k. So, size of the C array is k. So, we fill it we initialize, we put the count as 0. So, this is the initialization and, then we read the array. So, we have given the A array of size n. So, this is our A array and this is our C array of size 1 to k. So, initialize this all by 0 and then we read this A array and we accordingly if that we put the frequency in C array.

So, for j is equal to 1 to n, this is the A array. So, we read A j. So, this is particular j. So, we read A j. So, depending on the value of A j. So, if this is say 2 then we go to this and we put plus 1 like this. So, this is sort of frequency we are counting frequency of the that particular number to So, what we do? Now we do, we do this A, of A C of A j. C of A j we increase by 1; C of A j plus 1. So, this is just a frequency counting. So, the number of depending on the value of the element, this is the frequency count of that element.

So, after this what we are doing? So, up to this if we just count the frequency. Say suppose we have say input say like, this suppose we have a input. Say suppose, Our A array is 4 1, 3, 4, 3, ok.

(Refer Slide Time: 09:18)

Suppose this is our A array. So, this is 4 is the maximum size. So, C array is 1, 2, 3, 4. This is our C array. Now what we are doing this step we are initializing by 0, and we are just counting the frequency. So, we first read this is 1, 2, 3, 4, 5 there are 5 elements. So, we first 4.

So, we put it plus 1. So, this is 1 then 1, 1, this is 3. 1 this is 4 again. So, this is plus 1 2 this is 3 this is 2. So, this is the execution after this. So now, now we know that there is only one 1. So, we can just pin the 1. So, if you put a bracket. We can just pin the 1, then we know there is no 2. So, we will just ignore that and we know there is 2, 3. So, we can pin this to 3 and we know there is 2, 4. So, we can pin this 2, 4 sort. So, sort it sort it. So, should we stop here, because we just, if we just read the array, read the C, C i mean frequency wise, that is it.

So, we can just get that sorted one, but should we stop that, stop here? No because, we want a extra property in this algorithm. What is that? That is call stability. Stable sorting algorithm or stability or stable sorting algorithm, we want the this sorting algorithm to be stable. Stability means it should preserve the between the equal element it should preserve the ordering of the equal element like, like if we print just this. So, this 3 and this 3, we do not know which 3 come first in the input. We know this 3 came first than this 3.

So, we want in the output this 3 should come first than this 3. So, suppose there is a small tag over here. So, these are very important for satellite data. So, if we see the Google map some if we just capture the images. So, 2 image looks like same, but they are not exactly same. So, for those say suppose this 3 contents is 3 point we put a tag over here, 1 and we put a tag over here, 2 to indicate that this 3 come, this 3 came first than this 3. So, in the output also we want this 3 should appear first than this 3, ok.

So, that is called the stability. So, stability means, the input ordering should be preserve in the output ordering, the between the equal elements. So, that is called stability. So, we want that stability. That is why we have to execute few more, we have to write we have to do few more step for this counting sort in order to get this stability. Let us just complete this pseudo code. So, for this what we do? We first have the cumulative frequency. So, we come back to this example again. So, let us just do the for i is equal to k we will just take the cumulative frequency 2 to k.

(Refer Slide Time: 13:21)

So, we do C i basically C i plus C of i - 1. And then we fill the B bucket like this. For j n down to 1, for j n down to 1 what we do? We just fill this B of, B of So, basically A j we are going to fill in this B of C of A j. So, this A j we are going to fill in B of C of A j. And we decrease C of A j by 1 and C of A j is decreased by C of A j - 1. So, C of A j is decreased by C of A j. So, this is 8, this is 9. C of A j is decreased by C of A j - 1. So, this is the bucket filling. Let us take an example. So, we take the same earlier example.

Suppose we have this input A array there are 5 element 4, 1, 3, 4, 3. So, this is A array. So, this is A array now. So, we take a so, we are going to fill this in to the B array. So, B is also has to be 5 element 1, 2, 3, 4, 5. Now we have a C array. So, this maximum k is 4. So, C is 1, 2, 3, 4. So, this is B array and this is C array. So, now, we just execute this code. Now we fill this is the initialized step. So, we initialize this C by 0 and then we put the frequency.

So, after this there is only 1, 1, no 0 no 2, 2, 2. So, this is after this frequency of C. Now after that we have, we have to compute this C prime or cumulative frequency. So, we can say this is C prime. So, this is basically our new C. So, cumulative frequency means we just. So, it is starting from 2, up to 2, this plus this 1. And then, then we have this plus, this plus this, 2 plus 3, this plus this, so 5. So, this is the after this cumulative frequency we have this position of this array. So, this is the C array after the execution of that loop, 4, 5. Now 1, 1, 3, 5.

So, this is our C array after this execution of this loop. So, this is meaning of this is, up to this how many elements are there? One element. Up to this, how many elements are there? 3 element. Up to this how many elements are here? 5 elements because, that total 5 elements are there this is the sense. So, now, this is our C array. Now we go for the exact bucket filling by this loop. So, we start with n. So, we just look at this and we have to put this into some of the B array. So, how we can put this? So, put it by this way. If we go to the C of A j. So, A j is 4 A j sorry, A j is 3.

So, C of A j, so this is basically 1, 2, 3, 4. So, C of A j is basically 3. C i by A j is 3. So, we will put it into here. So, this 3 we put it into here and we decrease this by 2. I mean this C of A j is decrease by C of A j - 1. So, this now this value is 2. So, what is the meaning of this? Meaning of this is if again we see a 3, and that will going to put in this position and that will assure the stability. Is this clear? If again we see a 3 we are going to put it here in the place of 2.

So, that will give us the stability, and that we are doing a n down to 1. So, we reduce this by now it is now this value is 2. This is the C array. Now this is our next element, this of this loop j loop n down to 1. So, now, it is 4. So, we go to C 4, C 4 is 5. So, we put it here this 4, and we decrease by this one. So, what is the meaning of that meaning of that is if

again we see a 4 we are going to put it in this place. So, that will give us the stability. So, now, this is basically 4, now we are here 3.

So, we go to here now it is telling us we put this 3 in here. And we reduce this by 1 so; that means, again if we see a 3 we are going to put it here. So, now, C 1, C 1 is basically 1 and we put it here, and we reduce this by 0. So, there is no more. So, this is C 4. So, we go to the C 4 it is now 4. So, we are going to put it here this 4. And this is reduced right now 3. So, it is not only sorted this also preserves the stability. So, this is stable.

(Refer Slide Time: 20:56)

Stable means, we preserve the input ordering. If this happen then it is called stable sorting algorithm. So, which is the comparison based sort is stable? Quick sort? Think about it. Now if you hear this stable because this 3 and this 3 are same. These are 2 equal element, but this 3 is coming first then this 3 if you put a little tag over here 0.1, 0.2 So, this 0.1 is coming before then 0.2. Even this 4 are equal, but this 4 is coming before this 4.

So, the input ordering is preserved between the equal elements. So, that is the property we want in our sorting algorithm. And to have this property we need to execute this extra step. Otherwise we could stop at this place; up to the frequency. This kind of bucketing I mean we have the buckets we are filling the buckets and then we have print the frequency of that numbers, but we need this extra property to have achieve this extra property which is stability we have to execute this, and this is a stable sorting algorithm.

So, now we want to have the time complexity of this. So, let us talk about, we want to analysis this code. So, this is the code and now we want to have the run time of this code.

(Refer Slide Time: 22:58)

So, analysis of counting sort. Basically we do the run time analysis. What is the time complexity of this counting sort run time? So, let us look at the code. So, here there are few for loops.

So, this is basically a for loop. So, this will take the order of k this the loop of size k. And here we are initializing this so, we are in asymptotic notation this initialization will take the constant effort. So, asymptotic sense So, that is why it is theta of k and this will give us theta of n the filling of the that frequency, and this will again give us the theta of k and this exact bucket filling will give us theta of n. So, what is the time complexity then? So, time is basically, adding if we add this to it is basically, theta of n plus k.

So, now what is k? K is the range, range of the elements. If we fix some range say if k is order of n then the time is basically theta of n. So, it is linear provided the range is also bounded by the number of inputs. So, if there are say n input.

(Refer Slide Time: 25:10)

And now we are bounding by say some c n or some constant n or something.

But if the k say n square then this is the dominating term if k is n square then the time complexity is basically what? So, the time is basically order of n plus k. Now depending on which is dominating. If k is now the case one if k is big o of n, then the time is linear. Otherwise, if k itself is say n square, or n log n. Then we have work then the time will be dominating by order of k it is basically, order of n square. Which is basically worse than the comparison based sort, ok.

So, that is the condition for this to be a linear time sorting algorithm if the k is order of n. Other than that it is not possible. So, like a say 5 n 100 n something like that some constant into n then we can have this sorting algorithm to be linear. Otherwise it is a, depending on the value of k it will be either order of k or order of n.

So, next class we will talk about radix sort and the bucket sort, those two are also another sorting algorithm, another linear time sorting algorithm.

Thank you.

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

Lecture – 17
Radix Sort

So we talk about another linear time sorting algorithm which is radix sort. It is the idea of radix sort was invented by Herman in 1990, very old sorting algorithm.

(Refer Slide Time: 00:32)

And it was used for US sensor. So, they use a cord for this sorting algorithm yes, sensor and the idea of this radix sort is basically digit by digit sort. And the original version of the Herman was from most significant digit to the least significant digit, but that idea was having some drawback.

So, the original version of the radix sort is most significant bit to least significant bit. But that is having some drawback that idea is not good. So, then it is modified and the current radix sort is basically, least significant to the most significant.

(Refer Slide Time: 01:59)

So, we sort this number digit by digit, by the least significant bit to the radix sort. So, digit by digit sort, least significant digit to most significant. So, least significant digits first then the we come most significant.

And whether we are sorting this each digit will use a help of auxiliary sorting algorithm these are stable sorting algorithm. So, when we sort this each digit I mean number based on that particular digit. So, we will take help of a auxiliary stable sorting algorithm. Maybe we can use the counting sort, stable sorting algorithm. So, let us take an example then it will be more clear. So, suppose we have this number 3 2 9 4 5 7 5 6 5 7.

(Refer Slide Time: 03:32)

These are the input 8 2 9, these are the 3 digit number and each of this is the decimal number, 3 8 3 9 4 3 6 7 2 0 3 5 5. So, how many numbers 1 2 3 4 5 6 7. So, they are the input. We need to sort these numbers. So, this is basically 3 digit 3 bits. And each digit are decimal digit. Now this is a digit by digit, sorting algorithm and we start with the least significant digit.

So, this is the least significant digit. So, we sort this number based on this value, and we will use a stable sorting algorithm, say we are taking help of a counting sort, which is a stable sorting algorithm we know. Which is we are taking a linear time stable sorting algorithm. So, we cannot use the comparison based sort, it is not a linear. So, we take help of a stable sorting algorithm, and we sort this number based on this digit. So, 0 will come first. So, 7 2 0, then what is the next 5, 3 5 5, then we have 6, 4 3 6 and then we have 7. But we have 2 7 and this is a stable sorting algorithm.

So, this 7 is appearing first in the input. So, this should come first. So, basically we should have 4 5 7 and then 6 5 7 because of stability. We are using a stable sorting algorithm. And then the next number is 9 again we are using the stable stability. So, 3 2 9 will come first then 8 3 6. So, this is the sorting after first 7 2 0, 3 5 5 4 3 6, 4 5 7 6. So, this is after sorting based on the least significant digit. Now this is the digit by digit sort here. So, this is the next digit.

And we sort again this number based on this digit. And again we are using a stable sorting algorithm where we can use the counting sort. So, if you do that then just look at the volume that is it. So, the 2 is the minimum. So, there are 2 2s. So, it should preserve the stability. So, these 2 should come first. So, 7 2 0 then we have these 2, 3 2 9 and then we have 3, but they are 2 3s. So, 4 3 6 should come first then the 3 6 8, then we have no 4 we have 5 we have 3 5s, 1 2 3 and among this is first came first stability.

So, then 4 5 7 and 6 5 7. 1 2 3 4 5 6 7. So, now, finally, most significant digit and again here we are using the help of a counting sort. Or any stable sorting algorithm. So, if you do then this is the minimum 3 and here also you need to preserve the ordering. So, this 3 should come first 3 2 9, then this 3 3 5 5 and then 4 there are 2 fours this 4 should come 4 3 6 and 4 5 7. And there are 6 6 5 7, and there are 7 7 2 0 and 8 3 6, 1 2 3 4 5 6 7 sorted, sorted. So, this is basically the radix sort.

So, it is a digit by digit sort, and it starts from the least significant digit to the most significant digit. So, here digit are decimal digit 0 to 9 this is the least significant digit. So, we sort this number based on this and then we sort this number based on the next digit and then we sort this number based on the next which is the most significant digit. So, this is the shortest, now what is the correctness of this? To know the correctness we have to think in terms of the induction method.

Now suppose We are assuming this algorithm is correct up to t th digit, or t - 1 digit. So, suppose t is say 2. So, suppose this algorithm is correct up to this. Now we need to prove that this algorithm is correct after this t f digit to sort. Now we are assuming this algorithm is correct up to this. Now if there are 2 numbers different like here say there are 3 and 4 if they are different then 3 will come first and 4 there is no issue, only issue will come if there are common number this 3 and this 3.

So, among this 2 3 we know the algo is correct up to this, this is the assumption we made this is the method of induction. We are proving we know the algo is correct up to t - 1 steps.

(Refer Slide Time: 10:02)

This is our assumption and we are going to show this is correct after t th step. Now if the algo is correct up to t - 1 steps. Then this will appear before this because this value is less than this. Now once this will appear before this then this is a stable sorting algorithm.

So, we assume the our algo is correct up to t th step, say after second step which is now we are going to show it is correct after up to t - 1 step now we are going to show it is correct after t th step. So, between the different element we do not have no issue, only problem is between the equal elements. So, these 3 2 are equal. So, now, we know this is correct up to this. So that means, this will come before than this ok.

So, since it is coming before than this, and we are using a stable sorting algorithm at the t th step. So, this must come before this, that is what is happening. So, this is correct. So, this is the correctness of this radix sort now we talk about the time, we have to analyze this radix sort.

(Refer Slide Time: 11:58)

So, let us have a quick analysis of radix sort. So, analysis of radix sort and we will see that it is a linear time sorting algorithm. So, to analyze the radix sort suppose there are n, elements; suppose there are n inputs. And say each inputs are say b bits. So, each inputs are b bits. So, there are n numbers each of this are binary bits.

So, if we are given input we convert into binary, 0 on this. So, there are n numbers. So, each number is of total b bit. So, we are given a number any number we convert into binary, and the maximum size is b bits. Now we have to fix our digit now suppose we take our digit as say r bits. So, these are the numbers which are b bits, b bits number n numbers.

(Refer Slide Time: 13:15)

And now we fix our digit by r bit, r could be any value say 3. So, there are 3 bits we are using as a r. So, we are taking r bit as 1 digit, and we divide this into digits.

So, r bit r bit, r bit provided this b is multiple of r. So, these are r digit dot, dot, dot, dot like this. So, this is the r bit, this is the r bit this is least significant digit r bit. So, how many digit are there. So, number of digit will be b/r, this is the number of digit. Now we are going to sort this digit by digit. So, this is the least significant digit then this is next significant digit like this. Now we are going to sort digit by digit. So, we sort this first this we sort this number based on this value, this is the least significant value of this. So, this is r bit.

So, what is the maximum value it can sort how much time it will take. So, if we are using counting sort. We know counting sort will take theta(n+k), what is the k? K is the maximum range of that number. So, if it is r bit the range is basically 2^r. So that means, if it is r bit each can take one value. So, it is order of 2^r. So, basically, it is basically theta((n+2)^r), this is the time complexity to sort each of this digit. We sort this number based on this digit and this is the time.

(Refer Slide Time: 15:49)

So that means, time complexity for radix sort is run time of radix sort is basically theta(b/r). And each digit will take this much time (n+2)^r. So, this is the time complexity of radix sort. Where b is the total number of bits in binary and n is the number of element to be sorted, and r is the size of our digit we are defining that we are fixing some size for our digit. So, let us just talk about that. Now suppose we have to choose r to be log n. Suppose we fix our device size to be $\log_2 n$. Then 2^r is basically n. So, then this will be basically theta(n/r), r is again $\log_2 n$.

So, suppose we know where our numbers are coming from, our numbers are coming from this range, 0 to n^(d-1). Suppose our numbers are coming from this range. So, this is the maximum value. So, this is the maximum value our number can take. So, we have n such numbers each number can take this. Now numbers are basically b bits each number is b binary bits. So, what is the maximum value yeah it can take. So, 2^(b-1). So, basically 2^b = n^d. So, this means if you take the log, b = d*$\log_2 n$.

So, b/d, b/log(n) is basically d. So, this is basically theta(d*n). What is d? We have n numbers and we are assuming the numbers are coming from this range. 0 to n^(d-1). So, that case it is basically theta(d*n). Now if d is constant just like if the numbers are coming say n^(100-1), then n^d = 10. So, this is a constant. So, then this will be theta(n) if d is a constant. So, this is a linear time algorithm provided, d is a constant. So, bucket sort is a linear time sorting algorithm, provided if the number we choose are from this range.

So, next we will talk about another sorting algorithm, which is basically radix sort, which is now we talk about bucket sort. So, for bucket sort we are having some numbers, and here we need to assume the numbers are coming uniformly from the interval 0 1 bucket sort.

(Refer Slide Time: 19:10)

So, here we are assuming that the numbers are coming from the interval 0 and 1, this is open interval this is close interval. The elements are coming the numbers are coming uniformly from this; uniformly means we need to have this assumption in order to get the time complexity for this. So, what we do?

So, basically the idea is suppose there are n numbers, suppose n numbers. So, idea is to partition this into n sub intervals. We divide the interval into n sub intervals. So, numbers are n numbers are coming from this. Now we divide this into sub intervals, and each are called bucket. So, this is a b0 b1….b n - 1. This is starting from 0. So, these are called buckets. So, basically we have n buckets. b0 to bn sub intervals they are denoting by b0, b1, b(n-1) and these are basically bucket.

And then we have this number we put into the buckets, depending on the value of this, and once we fill the bucket, and we sort individual bucket and we report it, that is it. So, let us write the code. So, let us write the pseudo code for bucket sort.

(Refer Slide Time: 21:43)

So, there are n numbers. So, which are giving in a array. So, n is the length of this array. So, what we do for i is equal to 1 to n, we fill the bucket. We insert a i into the corresponding bucket, b of n of a i are lower ceiling.

So, this is the bucket filling we will take an example, and then after that once we have the buckets of numbers, then we sort this. Each bucket for there are n buckets for i is equal to 0 to n-1 because bucket is starting from b 0 to b n. Do, we sort? sorting means we use the insertion sort basically, insertion sorting algorithm we sort this b i, we sort each bucket. Each bucket contain some numbers we sort it. And then we concatenate just the bucket after sorting, we concatenate this b 0 then b 1 concatenated this b n - 1, and this will give us a sorted one ok.

So, let us take an example, quick example suppose our n is 10. And suppose numbers are like this 0.78, 0.17, 0.39, 0.26, 0.72.

(Refer Slide Time: 23:39)

So, we are assuming this numbers are coming uniformly from this, uniformly from this interval 0 1, 0.94, 0.21, 0.21, 0.23 and then 0.68. How many numbers 1 2 3 4 5 6 7 8 9 10. So, we want to sort this into the buckets. So, here n is 10. So, we have buckets like 0 to 10.

So, these are the bucket basically. So, you have 0 bucket. So, 1 2 up to nine. So, 1 2 3 4 5 6 7 8 then 9. We divide into equal. So, now, we put into the bucket. So, one bucket means the 0.1. So, 0.1 is basically we have 0.21 and then we have 0.17, 0.21 and we have 0.17. And then we have 0.2 bucket. Basically, we are putting into this like this. So but it is a basically a linear is 0.21 and then 0.23 and 0.26. And in number 3 we have only one element 0.39, these are the bucket and 6.

We have only one, 0.68 and 7 we have 2 elements 0.72, 0.78. And then we have 9 we have 0.94. So, this is the bucket filling then after that we sort this using the insertion sort individual buckets. So, these are the number falling in the bucket number 2, b 2 then we sort this numbers using the insertion sort. And then we just print the buckets like concatenate the bucket. So, this is basically bucket sort. Now what is the time complexity for this bucket sort? So, we are reading all the elements, and we are filling into the bucket So that will take the theta(n) time.

And then after bucket filling we are sorting the individual buckets using insertion sort. So, that will take the time depending on the number of elements in the i th bucket. So, if the number of element in the i th bucket is n then this is basically summation of, n i theta of n i square i is equal to 0 to n - 1. Where n i; the number of elements fall in b i i th bucket. Because each bucket is sorted by using the insertion sort. So, this is the time, now if we take the expected value of this, then this is basically expectation of this. So, this is basically theta(n) + summation of(big O of (expected value of n i square)). So, i is from 0 to n - 1. Now this expected value of n i square.

(Refer Slide Time: 27:31)

So, the time complexity is basically theta(n) + summation of(big O of(expectation of n i square)) and this i is from 0 to n - 1. Now we claim that this expectation of n i square. Also we have to prove that this is basically (2-1)/n. If we can prove this then it is done. Suppose we assume this is for all i then this will become what? Then this will become expected this will become linear.

So, how to prove this? Expected value of n i square is 2 - (this). So, to prove this we need to take help of indicator random variable. So, this we have to prove. So, what is that? Now we define x ij is 1 if, if a j falls in j i th bucket if a j falls in i th bucket, 0 otherwise. X ij is 1 if a j is false in because this ball this elements are chosen randomly from 0 1.

And we have n buckets if the b 0 to b n - 1. So, we define this. So, what is n i? N I is basically summation of x I. So, n i is the number of element in i th bucket. So, n i is Basically summation of x i and this i is j is varying from 1 to n. So, we take the expectation of ni^{th} square. So, ni^{th} square is basically this square and we take the expectation of this. So, basically the expected value of expectation is a linear function. We will take that x i square x ij square + double summation of x i k x k j x i ij I k, where k not equal to j. This is the form of x square now expectation is a linear function. We take the expectation inside.

So, we take the expectation inside. So, expectation of x i x j square, and this j is from one to n this summation of expectation of this 2, and this is a independent.

(Refer Slide Time: 30:33)

So, this is k not equal to j. So, if we write this, then this is basically expectation of x i square.

(Refer Slide Time: 30:51)

x i x j square is basically 1 into 1/n + 0 is to 1 - 1/n. So, this is basically 1/n. And now this will give us expectation of x ij x i k, if we take the independence of this 2 into expectation of x i k. So, this will give us Basically 1/n square.

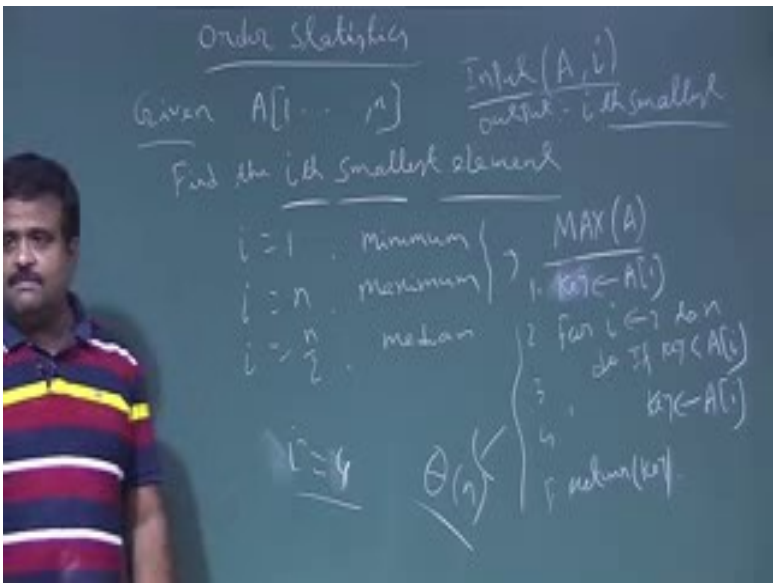
So, if we plug this value over here, we are getting basically (n * 1/n) + (n * n - 1 * 1/(n^2)). So, this will give us basically 2 - 1/n. That is what we are looking for 2 - 1/n. So, that is basically expectation of n i square. So, if we put this expectation then we are getting basically bucket sort is linear.

Thank you.

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

Lecture – 18
Order Statistics

(Refer Slide Time: 00:26)



So, we talk about order statistics. So, this problem is basically we are given some numbers given an array of n elements and the problem is to find the i th smallest element that is the called order statistics finding the i th smallest element we are given n numbers and we need to find the i th smallest element. So, this problem is called order statistics problem or this is called order statistics. now if i is equal to 1; this is called minimum. So, for i is equal to 1 means first smallest element which is basically minimum.

I is equal to n this is called maximum and for i is equal to n/2 it is called median provided n/2 is an even number otherwise we can put a ceiling on (n+1)/2 anyway, so, if i is like that then it is basically the median. So, now, the question is how we can solve this problem in general. So, for any i say suppose for these 2 problems it is quite simple. So, how we can find the minimum or maximum?

So, what we can do we can just scan the array to get the minimum or maximum; so, very simple code. So, suppose this is a maximum finding max of A. So, just few line code. So, we start with i. So, we have we are assigning this key; key value is basically A 1 and then we are

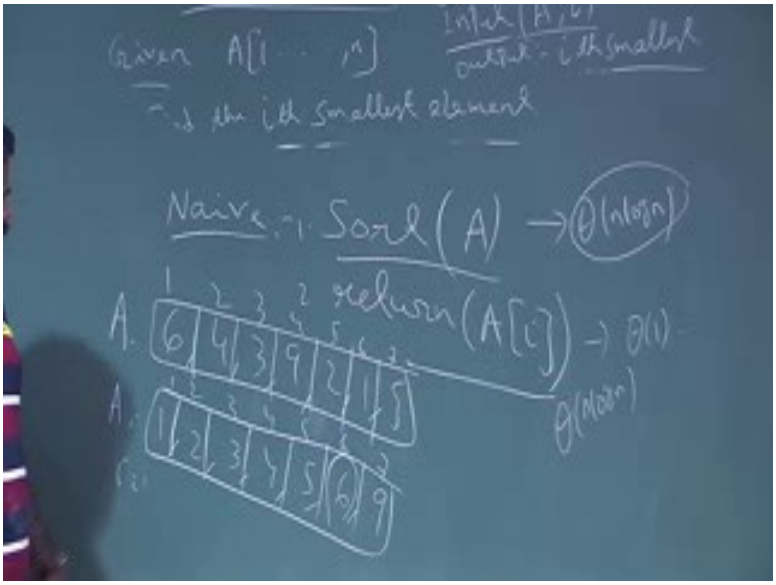
having a for loop for i starting from 2 to n we compare with the key. So, we are finding maximum do if key is less than A i then we replace key by the new A i; that is it and finally, after this loop we just return the key as a maximum very simple code.

So, basically we need to scan the array. Basically need to scan the array and every time if the element is greater than that what we have candidate of the maximum then we replace that. So, this code will take how much time this is a for loop of size n this is a linear time order of n, now this is for maximum. Now suppose we want to find out the second minimum second smallest element say i is equal to 2 then can we extend this code or say it could be difficult to extend this code even for i is equal to 3 or 4.

Because we may need to store these and then again compare these, this simple code will not extended i mean for this for any value of i in general. So, i is also part of the input. So, basically the order statistics problem is to we have input is an array and A i and output is basically i th smallest element. So, now, how we can solve this problem in general like for any i. So, the naive approach is we can sort these numbers.

And then we can go to the i th position and we return that value that is it. So, that is the naive approach what is the naive approach. So, we can just sort this array using any like sorting algorithm. If we have to use the comparison based sort because using that we do not need to use the auxiliary storage for that auxiliary storage means that we do not need to bound our input in a some range. So, yeah, what is the naive approach?

(Refer Slide Time: 05:13)



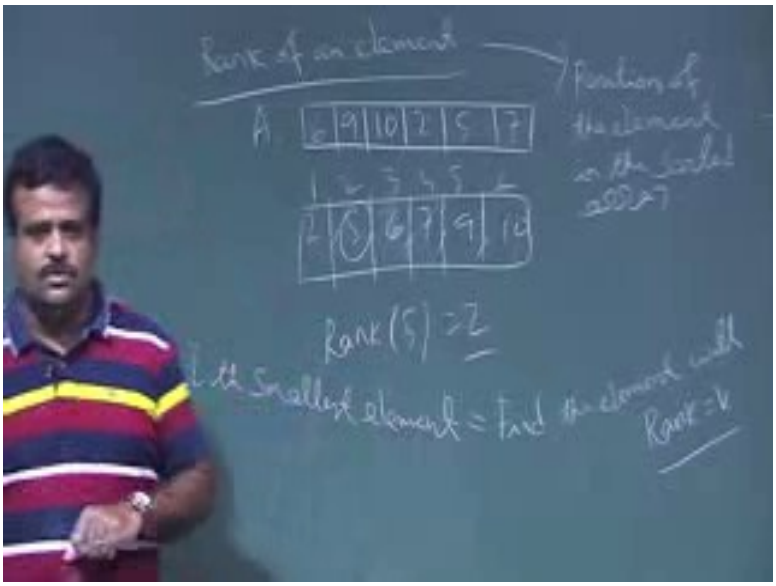
So, we sort it we sort A and then we return A i after sorting. So, that is our i th smallest element. So, suppose this is the input 6 4 3 9 2 1 5 there are 7 numbers. So, now suppose we want to find out say fifth smallest element. So, what is our naive approach we sort this. So, once we sort it the array will be like this 1 2 3 4 5 6 9 are we missing any numbers 1 2 3 4 5 6 9 fine.

So, this is the sorted one after sorting with this 1 2 3 4 5 6 7. So, if we want to find out say 6 smallest element i is equal to 6 then we just return this one A 6 after sorting. So, what is the time complexity? So, this will take linear time sorry this will take order of n log n because to make it linear we need to restrict our self in A range of the input and also we need to take that size of the storage and this is the basically just of return order of one. So, this will take basically order of n log n provided we use a sorting algorithm which is n log n I mean heap sort we can use even merge sort.

But merge sort we need a same size array for the margin now we want to do something better than n log n. So, that is the algorithm we will discuss today how we can do better than n log n because we do not need to sort I mean sorting is something extra we are doing because we just ask for the i th smallest element. So, if we sort then we can find all the smallest element that is i for 1 to n. So, that we do not need really we just need the i th smallest element for a particular i.

So, how to do that? So, let us define rank of an element. So, we have A array given a array of some n element say 6 9 10 2 5 say 7 we have this element.

(Refer Slide Time: 08:23)

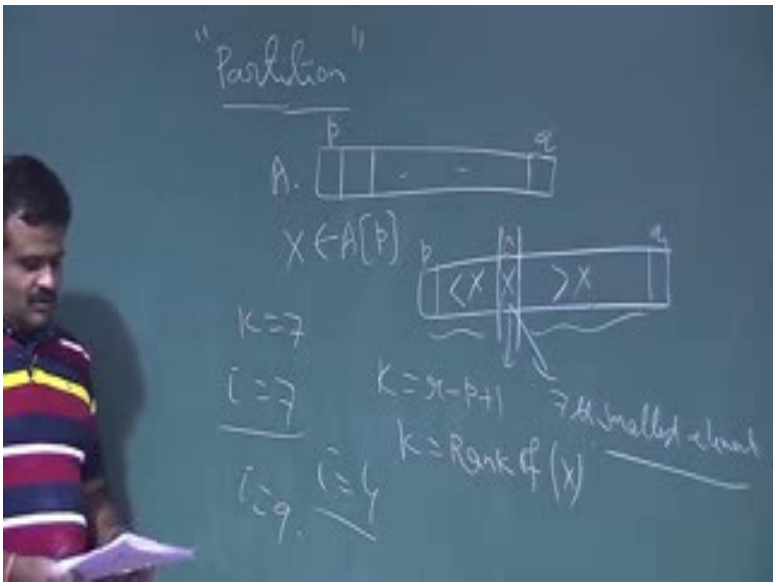


So, this is our input now what is the rank of say if I ask rank of say 10? Rank of 5? Rank of a element is basically position of the element in the sorted array. So, if we sort this then it is basically 2 5 6 7 9 10.

So, 1 2 3 4 5 6 1 2 3 4 5 6, so, if I ask rank of 5 rank of 5 means basically 2 because position of this 5 in a sorted array is basically it is in second position two. So, rank of 5 is basically 2 rank of 7 is 4 rank of 9 is 5. So, basically order statistics problem is basically finding i th smallest element means we are trying to find the element whose rank is i. So, the i th smallest element means to find an element this problem is same as to find an element find the element whose rank is i the element with rank i rank equal to i. So, finding the i th smallest element is same as we are looking for a element whose rank is i.

So, how we can do this other than sorting and going to the i th position and get it. So, that we have to discuss. So any sub routine you can think for which can help us for this I mean we do not need to sort the complete array, but any subroutine from quick sort? Yes! partition-subroutine . So, let us talk about that how we can use the partition sub routine which we use in quick sort to find the i th smallest element, so, our partition; partition sub routine in quick sort.

(Refer Slide Time: 11:41)



So, what we had what is the partition sub routine basically just to recall suppose we have A array; A array say p to q. So, what we are doing in partition. So, we choose this A p as a pivot element first element we choose as a pivot element and after partition what we are doing we are dividing the array into 2 sub array we are putting the pivot element in this position and all the elements over here is less than x all element over there is greater than x and this is this is the position; we are returning the position of this r if this is p to q and in the partition.

So, this is basically partition call we are taking the first element as a pivot our original partition algorithm we can take any element as a pivot, but we are taking first element as a pivot and then after partitioning what it is doing it is dividing the array into 2 sub array and here it is putting x in such a way such that the left sub array all the element must be less than x and the right sub array all the elements must be greater than x and x is in correct position.

So, x is basically p - x is basically in the r th position; now if we define say k is equal to r - p + 1 then can you tell me what is the meaning of this k? we just defined r - p + 1 this is basically our array starting from p to q. So, this is the position of this is the rank of k rank of x. So, k is basically rank of x because if it is sorted in the sorted array x is in right position x is in correct position. So, what is the position of x if this is starting from 1 1 it is starting from p.

So, that means r - p + 1 if it is starting from 1 if p is equal to 1. So, this is basically r th position, but our array is general array our index is start from say p to q. So, if p is 1 if our

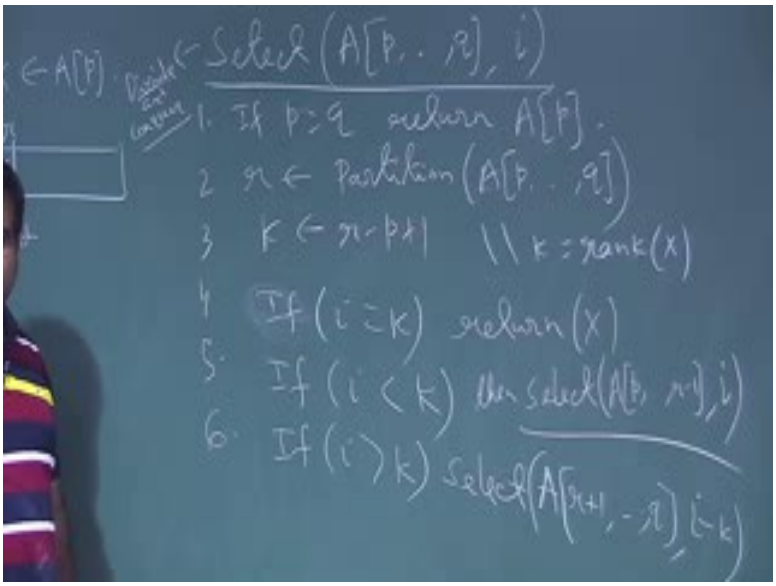
array is starting from 1 to n p is 1 then this is cancel out it is r. So, r is basically the rank of the x because x is in correct position when you sort it x position is not changing in the quick sort we sort this we sort this x is in correct position. So, that is the divide and conquer we use. So, what we do now, now if we are looking for the k th smallest element then we got that is our x.

Now, suppose if k is say 7 suppose our pivot element this is 7 rank is 7. So, suppose this is our seventh smallest element k is 7. Now suppose we are looking for i is equal to 7; seventh smallest element then we got i straightaway, we stop. And now suppose we are not that much lucky suppose we are looking for i = 4; we are looking for fourth smallest element we know this is seventh smallest element so; that means fourth smallest element must be here because this is the seventh smallest.

So, now we call the same function. We look at the same this is also divide and conquer technique we look at the fourth smallest element in this sub array with i is equal to 4. So, this is also divide and conquer technique. We will write the code now suppose we are looking for say ninth smallest element now we know this is the seventh smallest element. So, ninth smallest element cannot be this side this sub array. So, we need to look at the ninth smallest element here.

But we have already reach the seventh smallest element. So, we will look at the second 7 9 - 7 there is a second smallest element in this sub array. So, this is the divide and conquer step. So, we do not need to sort completely, but we will take use of this partition sub routine.

(Refer Slide Time: 16:53)

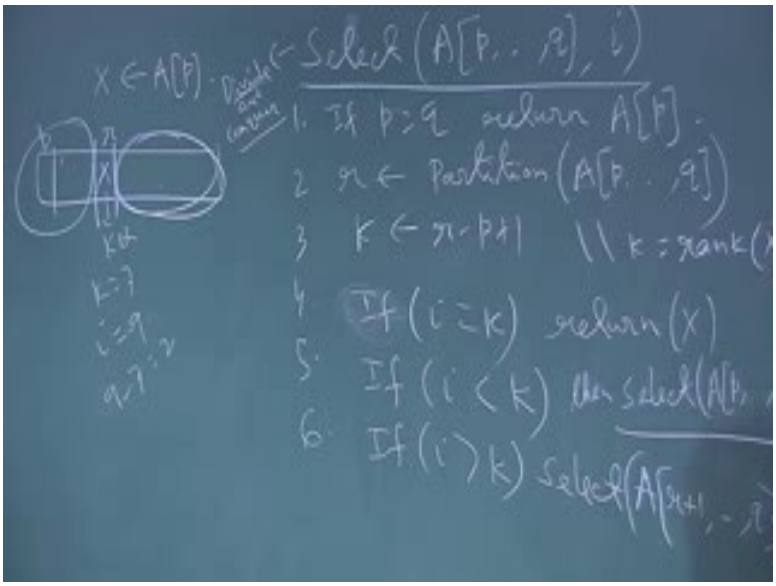


So, let us write the code for this. So, this is we call select algorithm. So, this we call select. So, what is the input; A; which is starting from p to q and i.

So, now, if p is equal to r q; we return; we return A p otherwise what we do we call the partition; we call our partition algorithm p to q. So, it will choose the first element as a pivot and then it will just divide the array into 2 sub array 1 is. So, it will choose. So, x is basically A p and now this is the p th 1 and it will return this r now we take k is equal to r - p + 1 and this k is basically rank of x.

Now, if we are looking for k th smallest I mean if i is equal to k i is equal to k then we just return x p turn x that A p I mean that pivot element basically happy lucky case I mean otherwise if i is less than k then we again call this select on this sub array. So, suppose this is the k th smallest and our i is less than k sub. So, we have to look at this sub array. This is divide and conquer technique. So, this select is a divide and conquer approach. So, if i is less than k, then again we call this select on the left sub array p to r - 1 with this i again we call this; this left sub array with p to r - 1 and with i else if i is greater than k then again we have to call select in the right sub array r + 1 to q with A new i and that i will be i - k because we have already seen the k th smallest and then suppose if k is equal to say 7 and if you are looking for ninth smallest; that means, if this is 7 say.

(Refer Slide Time: 20:01)



And if we are looking for 9 smallest; that means, we have already reached to the seventh smallest here.

Then we have to look first 9 - 7 second smallest in this sub array. So, we call this; this is the conquer step. So, this is the code this is the select code. So, here basically we are using the partition algorithm of quick sort. So, what is the time complexity of this? So, when we will be lucky we will be lucky if we get this sorted in that case we are calling the partition it is order of n that is it because partition algorithm will take order of n. So, this is a linear time now even if we are not lucky here. So, we are calling here.

So, it will depend on this our pivot element I mean it will depend on the partition. So, if our partition is say. So, if our partition is say half half then we are. So, it will depend on the partition or the pivot element.

(Refer Slide Time: 21:19)

Now suppose our partition is say half half suppose x is sitting here n/2 n/2. So, if the partition is n/2 : n/2 then; what is the recurrence? So, we are having the recurrence T(n) = T(n/2) + theta(n). So, this is the recurrence for this now this will give us what this will give us theta of n by again master method.

So, this is the lucky case if the partition is this one so this is the best case now we also have almost best case now suppose partition is not that much good, but suppose the partition is say some little fraction over here 1/10 and this is a 9/10 suppose almost best case this one if the partition is this. This type of analysis we did in the quick sort if the partition is this then also what is the run time complexity then the time complexity is basically in the worst case maybe we have to look for the bigger sub array it is partitioning into 2 sub array.

This is the smaller one; this is the bigger one. So, we are looking for the worst case runtime. So, in the worst case we have to go for the bigger part of the array. So, this will be T(n/10) + this is the cost for partition sub routine. So, this will again give us linear time by the master method, but only this if it is 1/100 : 99/100. So, if we can ensure that little fraction in one side then also we are lucky. So, this is almost best case we have seen this type of analysis in the; this is the lucky case, but when is the unlucky case what is the worst case of this algorithm select.

So, worst case is basically if we choose our pivot is minimum or maximum then what is the partition then the partition will be 0 is to n - 1 that is the worst case and we are choosing our

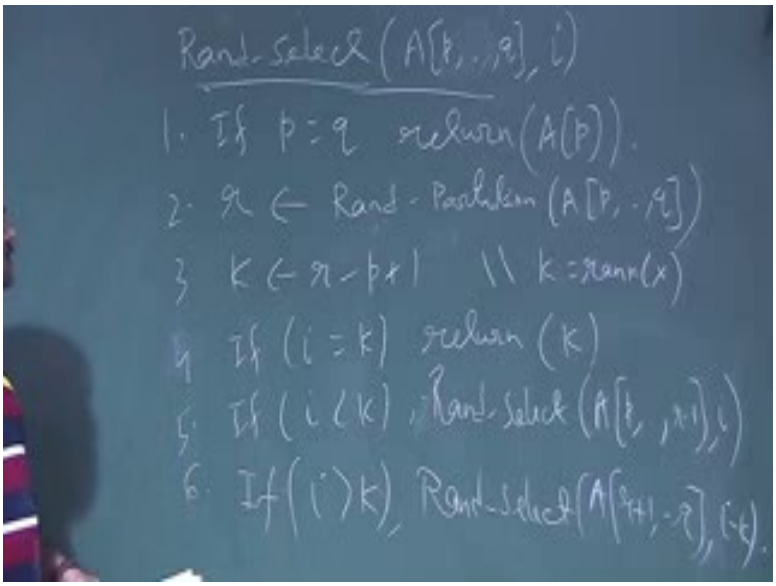
pivot minimum or maximum then what is the recurrence; recurrence is T(n) = T(n - 1) + cost for partition.

(Refer Slide Time: 23:31)

So, this is again basically theta of n square which is worse than sorting in n log n and going for the i th smallest element. So, this is the worst case run time for this select algorithm.

So, now, how we can ensure that like we did this type of analysis in quick sort randomized version now how we can ensure that on an average we will be lucky? So, if the partition is sometimes lucky sometimes unlucky we have seen also the idea is to choose the pivot randomly. So, this is the randomized version of this select. So, that is basically rand select.

(Refer Slide Time: 24:44)



So, what we are doing here we are basically the same code p is equal to q we return a p and then we are just basically using a randomized version of the quick sort sorry randomized version of the partition where we are choosing the pivot element randomly. So, that may give us with a hope that it may we may be lucky sometimes it may choose the good people basically. So, let us talk about this rand partition a comma. So, this is say a is starting from p to q and then i. So, randomized partition a comma p to q and then the remaining part is sent r - p + 1.

This k is the rank of x and then we just do the same thing if i is equal to k then we return k otherwise if i is less than k then we have to call again this randomized select or rand select from p to r - 1 comma i. So, same code, but every time we have to call the randomized version of this if i is greater than k then you have to call rand select a to r + 1 to q, but now the index is i - k. So, this is the randomized version of that. So, in the next class we will analyze this code and we will see the expected runtime of this code is order of n linear time.

But in the original version of the select we have seen the in the worst case the partition will be 0 : n - 1 in that case it is the order of n square algorithm, but this we will do in the next class the analysis.

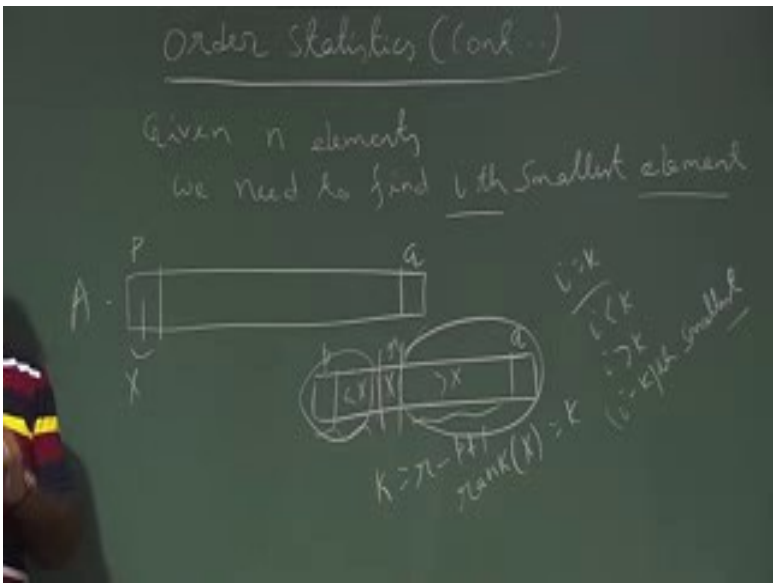
Thank you.

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

Lecture – 19
Order Statistics (contd.)

So, we are talking order statistics problem the problem is...

(Refer Slide Time: 00:27)



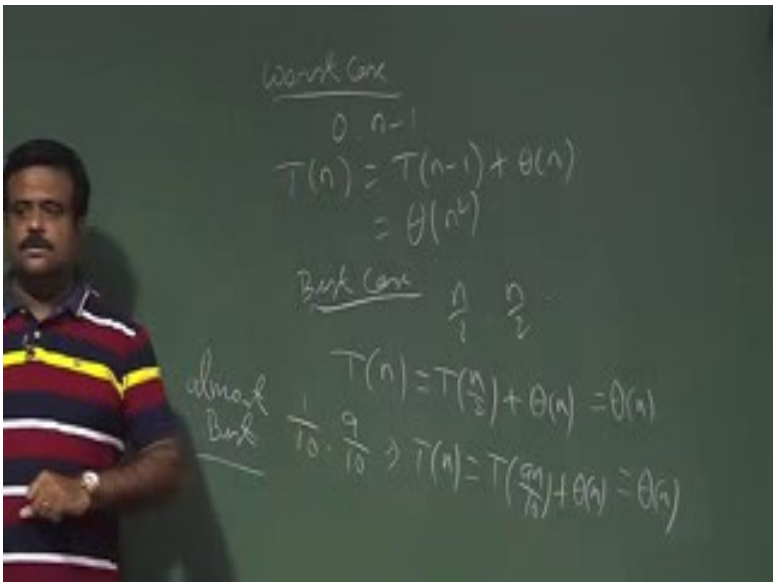
If we are given n numbers, we need to find out i th smallest element. So, this is the problem for order statistics. So, i is any index from 1 to n. So, if i = 1 this is minimum if i is equal to n this is maximum in if i is equal to n/2 order then it is called median. So, we have seen the select algorithm basically we are using the partition algorithm.

We are given this array of numbers. So, we choose this as a pivot element x. So, this is say p to q and then we call the partition then x will be sitting somewhere here and this is the index of x where it is sitting and this is p this is q and then. So, all the elements over here are less than x all the elements over here are greater than x. So, now this is k if we choose k is equal to r - p + 1 then k is basically rank of x.

Now, if i is equal to k then we got the i-th smallest element otherwise if i is less than k this is the divide and conquer technique if i is less than k then we know our i th smallest element is here then again we call the same function this is the conquer step on this sub array otherwise if i is greater than k, then we know the i-th smallest element will be here, but we already release the k th smallest element. So, we need to look at this sub array with i - k th smallest element.

So, this is the new i for if you have to look at this. So, this was our select algorithm now the time complexity of this will depend on the partition. So, how the partition will be over this array I mean if the partition is good partition I mean rather if the pivot element is minimum or maximum then it is a bad partition then it is.

(Refer Slide Time: 03:01)

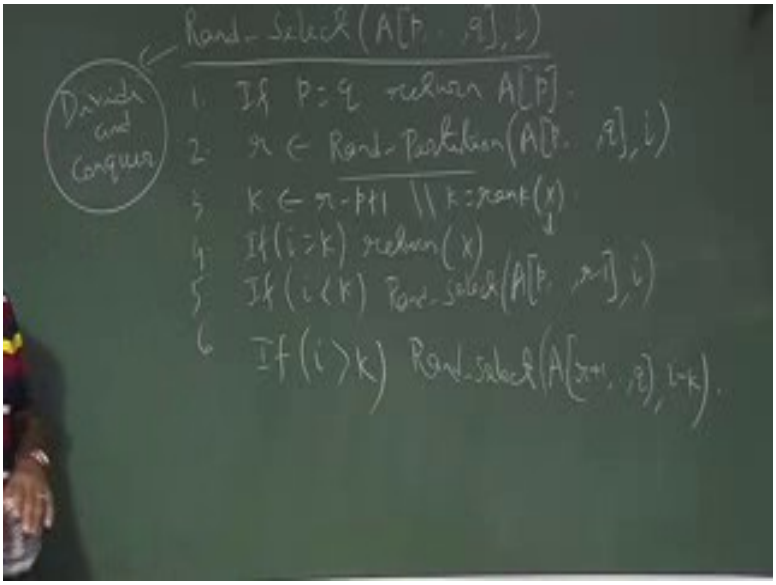


If the pivot we choose always minimum or maximum then the worst case will be 0 is to n - 1 and in that case the recurrence is theta n. So, this is our recurrence and this is the arithmetic series. So, this will give us order of n square and what is the best case? Best case is basically if the partition is n/2 is to n/2 then we have seen the recurrence T(n) = T(n/2) + theta(n) where theta(n) is the cost for the partition algorithm and this by master method of case 3 this is order of n and even if we have almost best case right, If we have partition like 1/10 is to 9/10 then also we have seen the recurrence is T(n) = T(9n/10) + theta(n). So, this is again will give us theta of n. So, this is the almost best case.

So,that was a lucky case. So, now we will talk about a randomized version of this algorithm this is our select algorithm. this is divide and conquer approach we choose we divide the array into 2 sub array and we put the pivot in some by its position and then we call the conquer by calling the same function select either on the left side or right side depending on the value of i.

But this is the analysis of that code depending on the partition it will give us the worst case or best case lucky case or the unlucky case. So, let us talk about randomized version of this. So, if we know that.

(Refer Slide Time: 05:19)



So, in the randomized version of this, so, this is called randomized rand select. So, we are given an array and we have to find out the i-th smallest element.

So, here the idea is to choose the pivot element randomly. So, that there is a chance that the partition will be the good partition in some of the subsequent cases. So, in that case we can say we will. In fact, we will prove that the expected run time will be order of n. So, what is that code; code is if p is equal to q then we just return A p else; what we do we just call a partition; now here we call the randomized partition A p to q.

So, what is this randomized partition? So, this is the array A p to q. So, we choose A p as a pivot element this was our original partition algorithm, but in the randomized version of the partition algorithm any one of this index will be the pivot element. So, like pivot. So, any one

of this index we can choose as a pivot element if there are n elements. So, they are equally likely to come.

So, we choose a index randomly from this p to q and then we choose that element as a pivot element. So, that is the randomized version of the quick sort and it will return the r r is the r is basically the index where x is putting after calling the pivot and then we choose k to be r - p + 1; k is basically rank of x; x is the pivot element which is chosen randomly now the code is similar than the select code we discussed in the last lecture. So, if i is equal to k then we return x that is our i-th smallest element else.

If i is less than k then again we have to call this randomized select. So, less than k means we have to look the i-th smallest element in this sub array. So, we call again randomized select on this sub array left sub array A p to r - 1 with the index i else if i is greater than k then we have to call the rand select on the right sub array, but we already explore the k th smallest element. So, now, we have to get the i - k th smallest element a r + 1 to q with i - k.

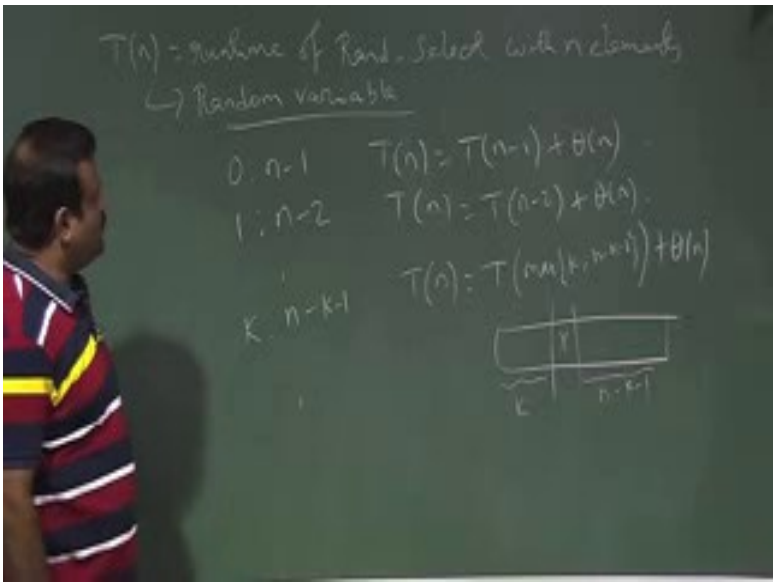
So, this is the divide and conquer approach divide and conquer. So, basically we divide the problem into 2 sub problem, here it is one sub problem we either look at the left sub tree or left array or right array.

(Refer Slide Time: 09:26)

Now what is the time complexity? So, time complexity will depend on the partition. So, worst case we have seen. So, this will take the order of one time this partition will take linear

time and. So, it will depend on the nature of the partition if we are lucky if the partition is good partition if the pivot is good pivot then we know this is either n/2 or it is some portion of the some fraction of the n, but if the worst case it is n - 1. So, depending on the partition it is basically give us the time, but worst case time complexity is always n square, but this is the randomized version. So, we want to look at the average case analysis. So, expected run time. So, that we are going to do now.

(Refer Slide Time: 10:32)

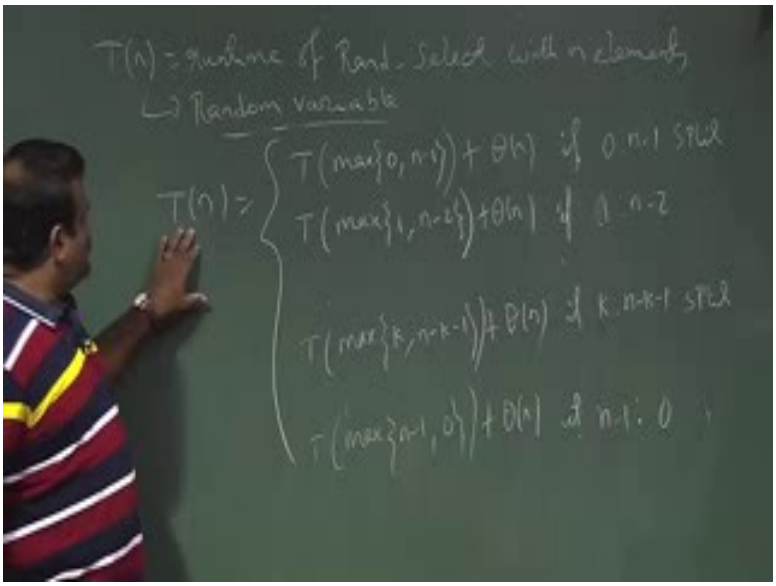


So, for that let us denote the time complexity as T(n). T (n) is the run time of rand select with n element now this is randomly chosen pivot. So, the partition we do not know which partition will come. So, that is why this is basically a random variable this is basically a run time variable. So, if the partition is say 0 is to n - 1 then we know T n is basically T 0 + I mean T n - 1 + theta of n.

So, if the partition is 1 is to n - 2 then we know T n is basically T of n - 2 + theta of n and if the partition is k is to n - k + 1 n - k - 1 then the dot dot dot then the T n will be basically T of maximum of this 2. So, it is basically partition is this means we have x is sitting here and there are k element over here and k - n - k - 1 element over here. So, depending on the value of k we are talking about worst case run time. So, you go for the maximum size.

So, maximum is depending on the value of k it will be a max; max of k comma n - k - 1 + theta of n is the time for partition sub routine.

(Refer Slide Time: 12:45)



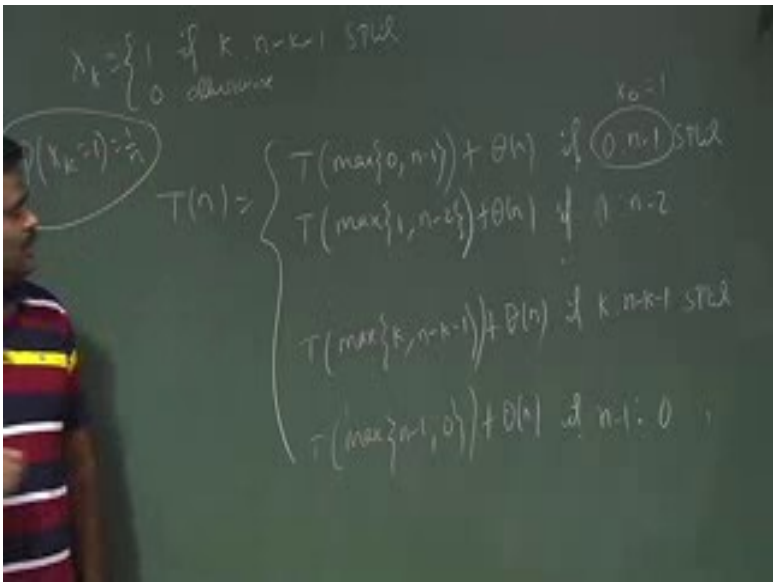
So, this way we continue. So, we can write T n in the functional form like this. So, depending on the partition, it will be of that particular form. So, T n is basically in the functional form it is basically T(max(0, n-1)) this is for symmetric we are writing in the max is n - 1.

But still for the symmetry we are writing this if the split is if the split or partition is 0 is to n - 1 split or it is T of max of 1 is to n - 2 if the split is 1 is to n - 2 like this dot dot dot T of max of k comma n - k - 1 + theta of n for the partition. So, if the split is k is to n - k - 1 dot dot dot T of max of n - 1 comma 0 + theta of n if the split is n - 1 comma 0 split.

So, this is T n in the functional form.T n is a random variable because we do not know which partition will occur because our pivot is randomly chosen. So, before running this we do not know which one is going to be a pivot. So, that is may; that means, we do not know which partition will occur, but at least one of this partition has to be occur, but we do not know which one. So, that is why T n is in this functional form now we want to take the expectation of this T n we want to calculate the expected value of this T n and that we are going to show it is linear.

So, for that we want to take this functional form to algebraic form. So, for that we need to take help of the indicator random variable. So, now, we define indicator random variable. So, x k, so, let us just.

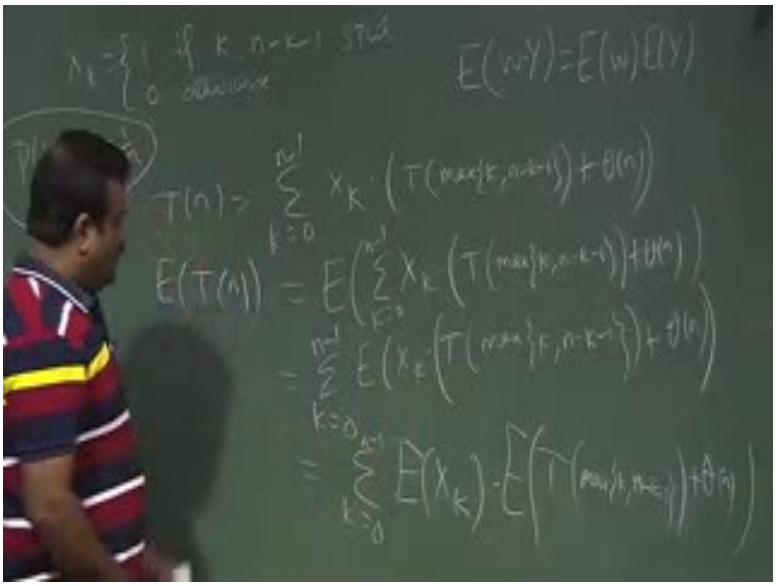
(Refer Slide Time: 15:17)



Let us just define the indicator random variable x k x k is 1 if the split is k is to n - k - 1 split 0 otherwise. So, one of the escape value is 0. So, if the partition is this then this is x 0 x 0 is one then remaining all other x value is 0. So, now, one of the partitions will occur and they are equally likely to occur.

So; that means, probability of x k equal to 1 is 1/n because there are n partitions. So, that is why it is 1 by n. So, now, we want to write this in terms of algebraic form because now we have a indicator random variable. So, 1 of this x k 1 of this partition will occur; that means 1 of that x k is 1 remaining as 0.

(Refer Slide Time: 16:26)



So, basically we can write T n as the summation of x k that particular one of the split is one into that corresponding t.

So, this is basically T(max(t , n - k - 1)) + theta(n). So, this is basically our x algebraic expression and this T k is running from 0 to n - 1. So, one of the partition will occur and that corresponding x k is 1 remaining x k's are 0. So, that is that is why we can write this in terms of this. So, now, we want to take the expectation on both side.

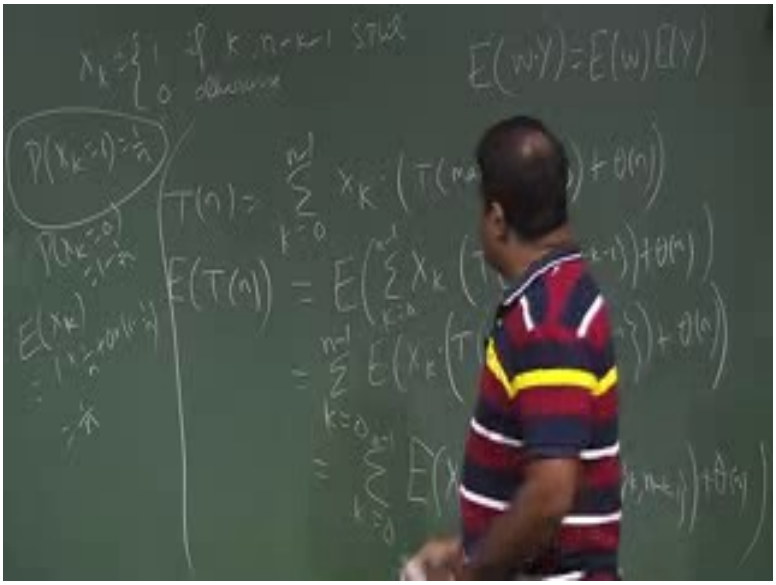
So, expectation of this now summation of x k into T(max(k, n-k-1)) + theta(n) . So, this k is from 0 to n - 1 now expectation is a linear function we can take the expectation inside, so, summation of expectation of x k into T of max of T comma n - k - 1. So, this is the closing bracket of this + theta(n). So, this will come with x k into this n this and this k is from 0 to n - 1.

So, we have something like that w y we have 2 random variable w and y expectation of w into y. So, if they are independent then we can written as expectation of w into expectation of y. So, if y and w are independent random variable independent why independent? Because here we are choosing the pivot element randomly. So, for that we need to generate the random number in the in the subsequent steps also.

So, this all choice of random numbers generation are independent. So, that sense it is coming the independence is coming. So, if we assume that independence then this can be written as

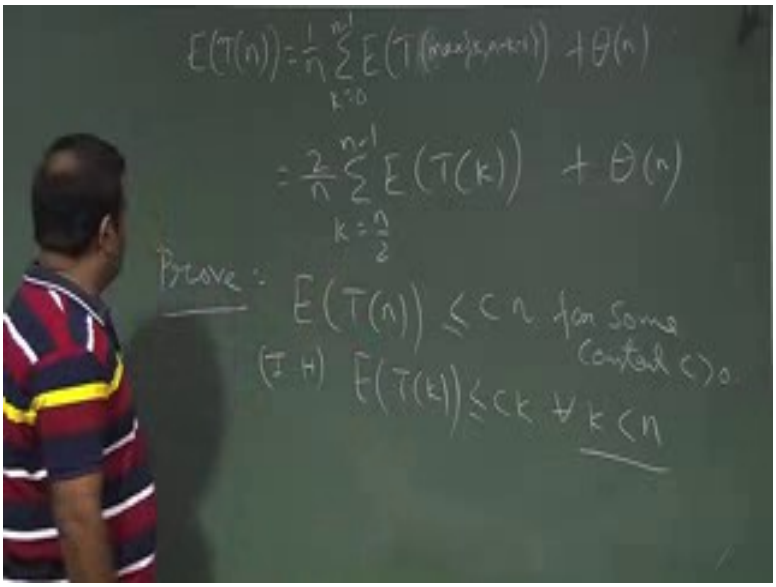
summation of expectation of x k into expectation of T(max(k, n-k-1)) + theta(n) and this k is varying from 0 to n - 1 now what is the expectation of x k.

(Refer Slide Time: 19:47)



Now, probability of x k is equal 1 this and probability of x k is equal to 0 is basically 1 - (this). So, this is a 2 point; this is a binary random variable. So, expectation is basically 1 2 value 1 and 0 1 into its probability + 0 into its probability. So, this is basically 1/n. So, expectation of x k is basically 1/n. So, we put this value over here 1/n.

(Refer Slide Time: 20:28)



So, this will give us expectation of T(n). So, this is 1 by n we take outside summation of this thing. So, expectation of this + this and this is nothing to do with the expectation. So, this will basically give us again theta of n. So, this is basically summation of expected value of n T of T of max of k comma n - k - 1 + theta of n this is from k from 0 to n - 1. So, now, how we can simplify this further. So, this is basically.

So, let us just say try to simplify this further now this is basically we are taking the expectation of T of maximum of this, now k is varying from 0 to 0 to n - 1 now if k is up to n by 2 then this maximum will be n by 2 i mean. So, these can be written as. So, these can be written as this is 2 by n summation of expectation of T of k where k is varying from n by 2 to n - 1 + theta of n because see when k is equal to 0 then the maximum is n - 1 when k is equal to 1 then maximum between 1 and n - 2 is n - 2.

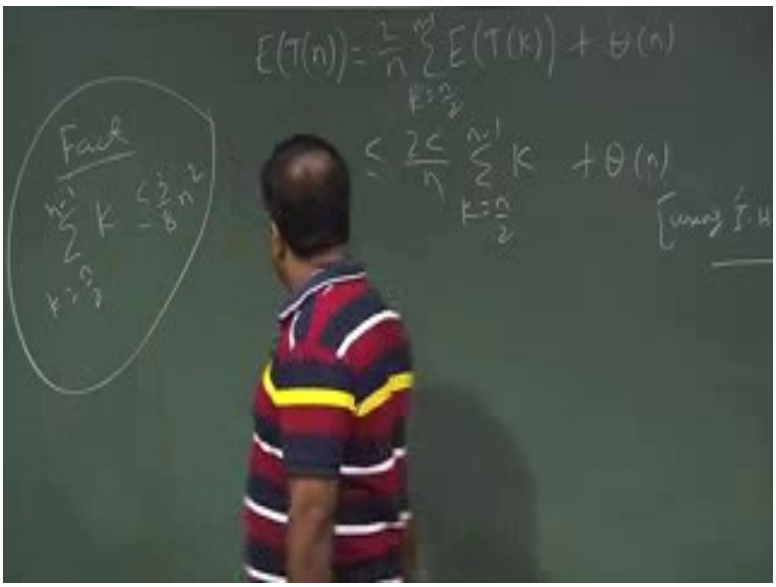
So, basically, so, if k k is varying from 1 2 like this n by 2 then n by 2 + 1 to n now if k is 0 then maximum is n - 1 k is 1 maximum is this like this. So, k is if k is n n by 2 then the maximum will be n by 2 again from here also we have maximum n by 2 to n. So, basically we have twice term this sum is basically we have 2 of this is this clear. So, now, we have to simplify this mode. So, this is basically the expression we got for the expected run time now how this is linear we have to show that.

So, we want to prove expected value of T n is basically big O of n. So, for that we want to take this as c of n some constant for some constant c greater than 0. So, this how we can prove this. So, now, will prove this by method of induction the substitution method now we

assuming that this is true this result is true this is the induction hypothesis. So, we assume this result is true for up to n.

So that means, expected value of T of k is basically less than equal to c k for all k less than n this is our assumption and then we prove that this is true for n also. So, this is the way this is the substitution method.

(Refer Slide Time: 24:36)

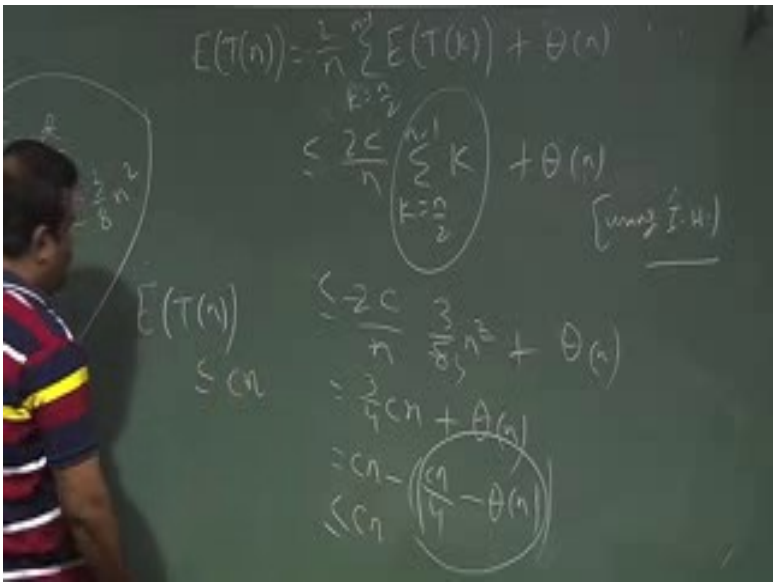


So, this is basically the expression we have 2/n summation of k is equal to n/2 to n - 1 expected value of T(k) + theta(n) outside. So, now, we will use this induction hypothesis. So, we will use this induction hypothesis.

Because this is the all the k value n by 2 to n - 1 these are all less than n. So, this will give us 2 c by n summation of summation of k k is equal to n by 2 to n - 1 + theta of n this is by using induction hypothesis. So, this is by using the induction hypothesis now we have to prove that this is less than c of n. So, to show that we need to take a inequality which is which is basically telling us this is a fact.

So, summation of k; k is equal to n/2 to n - 1 is basically can be shown as 3/8 n*n. So, you can prove it again by induction. So, this expression this inequality we are going to use.

(Refer Slide Time: 26:11)

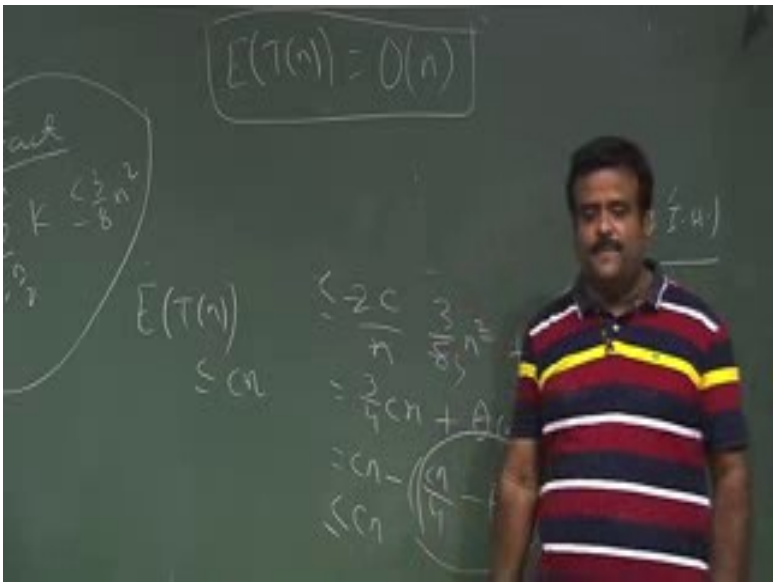


So, this is again 2 c/n into 3/8 n*n + theta(n). So, this is basically giving us what. So, this is 4 this n n cancel n.

So, 3/4 c n + theta(n), so, now, we have a control on c; c is the constant we can play with. So, we choose c in such a way that this should be less than. So we want to write as this c n - c n/4 - theta(n). So, if this expression is positive then we can write this is less than equal to c n, but to have this positive we can choose c is large such that it will be positive because this is theta of n means it is some expression in n some function in n.

So, we can choose c in such a way this will be the positive. So, this is basically less than n. So, this is the proof.

(Refer Slide Time: 27:36)



So, expected value of T(n) is basically less than c(n). So, this is basically telling us by the method of induction the expected value of basically big O of n. So, this is the average case analysis of randomized version of the say randomized select, but this is the average case. But the worst case is always n square because even we are choosing the pivot randomly it may happen that always that pivot is coming as minimum or maximum all though this is random choice, but that chance is there.

So, the worst case is always order big O worst case is always 0 is to n - 1 partition, but this is the average case analysis for this randomized version of the select.

Thank you.

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

Lecture – 20
Worst Case Linear Time Order

So we have seen the select algorithm and randomized select algorithm where basically the problem is to find the i th smallest element. Now basically we are using the quick sort partition algorithm. So, basically depending on the partition the pivot element we have to partition the array into two parts. So, depending on this it will give us the time complexity see, but in the worst case always it is 0 is to n - 1 in the worst case.

So, now we in this lecture we want to talk about a guarantee for the worst case linear time for finding the i th smallest element. So, this was invented by this people Blum Pratt I think Rivest 4 scientists and Trajan.

(Refer Slide Time: 01:20)

So, in this in the work we have. So, that was in I think 1973. So, this algorithm is proposed by this 4 scientists in 1997. So, idea is to basically their idea is to generate a good pivot. So, idea is to generate a good pivot recursively.

So, what do you mean by good pivot good pivot. So, bad pivot means it is always minimum or maximum so that means, it will be always partitioning all this 0 is to n - 1. Now good pivot

means if we can ensure that some portion of the element will be less than x, some portion of the element will be greater than x. Maybe it is very little portion 1/10 is to 9/10, then also we have seen we are lucky then also we have seen we have linear time. So, if we can ensure the pivot is good in the sense that if you can ensure there are certain portion may be very small amount, even it could be work for 99/100.

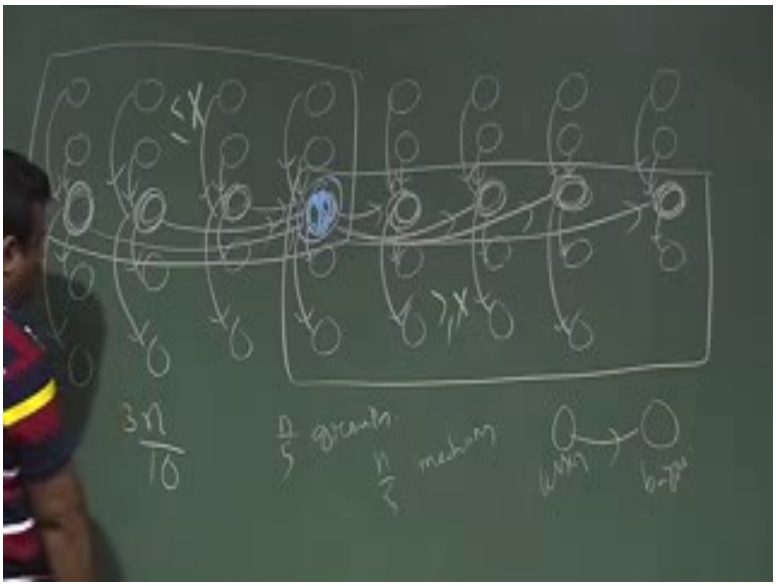
So, if we can ensure that a certain portion is less than x, and certain portion is greater than x then that choice of pivot is good pivot. So, the idea is to get such a pivot. If we can get such a pivot from this array, so we will use that pivot and then we call the partition and then obviously, if we know that there are some portion guaranteed, there are some portion will be left side and some portion will be right side. If this thing is guaranteed then if we call the partition then obviously, it will be array will be like this. So, it is guaranteed that some portion will be here some portion will be here.

So; that means, when we call again in the recursive call, they this then will go for here again in this subsequent recursively we are calling this divide and conquer technique, even in this call, even if we can choose again a good pivot. For this sub call then also again it will partition into certain portion over here should not. So, we want to avoid the 0 is to n - 1 case, and that can be avoid only by choosing a good pivot that will guarantee that there are some portion over left side some element less than x and some element greater than x.

So, the idea is to choose how one can choose a good pivot. So, if we can choose a good pivot which ensure that and choice has to be recursively and if it can ensure that guaranteed there will be some elements which are less than x, there will be few elements which are greater than x. Then we choose that x as our pivot and then we call partition, and then that will be that partition will be divide the array into like this portion not that 0 is to n - 1. So, how we one can choose a good pivot?

So, we have given array. We are dividing these array into 5 element groups.

(Refer Slide Time: 05:03)



So, we have given the array of n elements. So, we are dividing into 5 element groups like this. So, we have we are grouping the elements into 5 element groups like this, may be in the last one we have only 3 4 element because this n may not be multiple of 5, but anyway. So, this is the this is our we have given a array A array is form say we have a array say 1 to n or p to q. So, we take first 5 up to 5 as a first group, second group like this. So, we are grouping this array. So, we divide the array into 5 element groups.

Now, now what we are doing? Now we are finding the median of this group. So, this is only 5 elements. So, we can sort it up, we can just sort the element and we can get the median. Suppose this is the median of this group and this is the median, this is the median, this is the median this is the median. So, we sort this and got the middle point of the group. So, these are the medians of this individual group. So, we have how many groups are there. So, there are n elements. So, roughly n/5 groups are there.

So, among this, now, we have n/5 medians. Now we find the medians of the medians and that we will find recursively by the same function call. So, now we want to find the medians of medians. So, suppose this is the medians of median and this is the medians of medians and these we are going to choose as a our x. So, this is basically our pivot element we are going to choose, this is our x. Now if we can choose this as a pivot then why it is a good pivot so that means, what is the guarantee.

So, good pivot means it will guarantee that some fraction of the element will be less than x, some fraction will be greater than x. So, this we have to argue. So, this is the medians of medians. So, now, we will follow this notation. So, this is median of this. So, this is less than this, this is less than this. So, we will use this symbol as less than. So, this means this is lesser and this is bigger, we can use this symbol lesser by this. So, this is the medians of medians. So, this is lesser now there this is lesser than this.

Now this is the median of this group, so that means, this will be lesser than this, this will be lesser than this and this is like this. So, like this we have. So, this is the median of this group. So, this is the relation we have less than relation. So, this is the median of this group. So, this is lesser this is lesser like this. So, we continue with this like this like this like this. So, we can leave the last one like this. So, now, if we look at all the elements over here say this element, now this element is less than this because this is the median of this group this element is less than this.

Now, this is the median of then now this element is less than this so; that means, this element is less than this. So, if you take any element over here they are basically less than equal to x. So, all the elements over here are basically less than or equal to x, and that is guaranteed. Because if you take any element this element this element is less than this, this is the median of that group and this element is less than this because this is the median of the medians so; that means, this element is less than this. So, this is true for any element less than equal to. So, if you take this element this is the median of this group. So, this element has to be less than this. So, this is the less than symbol we have use.

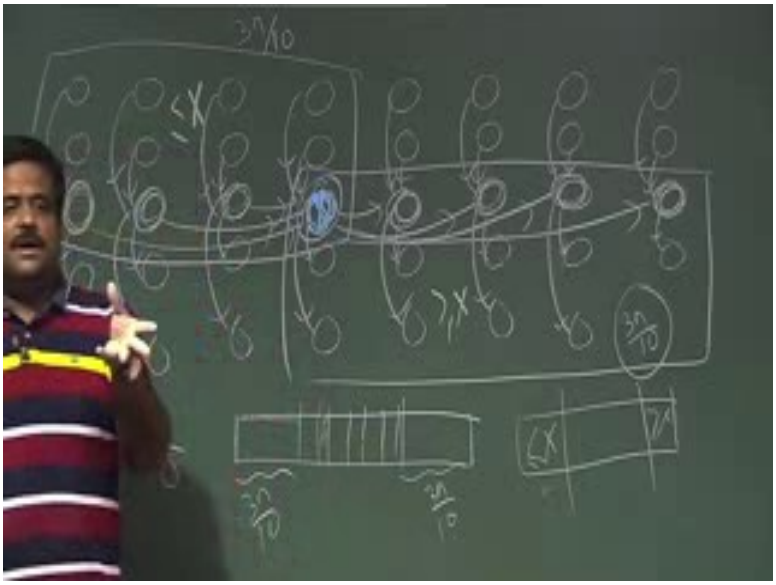
So, this is the portion where all the elements are less than or equal to x, and similarly this is the portion. So, this is the portion if you take this element say, this element is greater than this and this is greater than this. So, these element is greater than this. So, this is the portion where all the elements are greater than or equal to x, because by this relation if you take this element, this element is less than from this group median and this median is less than from the medians of the medians. So, this element is less than this.

So, we have a portion guaranteed we have a portion which is less than x, we have a portion which is greater than x. Now what is the size of this portion? Now size of this portion is basically. So, there are n/10 medians, n/10 groups now this is almost half of their. So, size of this is basically sorry n/5 size of this is n/10. So, among this n/10 there are 3 elements each

from each group. So, there are n by 10 such groups are there, which are basically i mean have a portion of which is basically less than x and how many of them? 3 of them.

So, basically 3 n/10 is the size of this portion, where elements from that portion is less than equal to x, and similarly here also the size of this portion is 3 n/10. So, basically x will be sitting somewhere here.

(Refer Slide Time: 12:35)



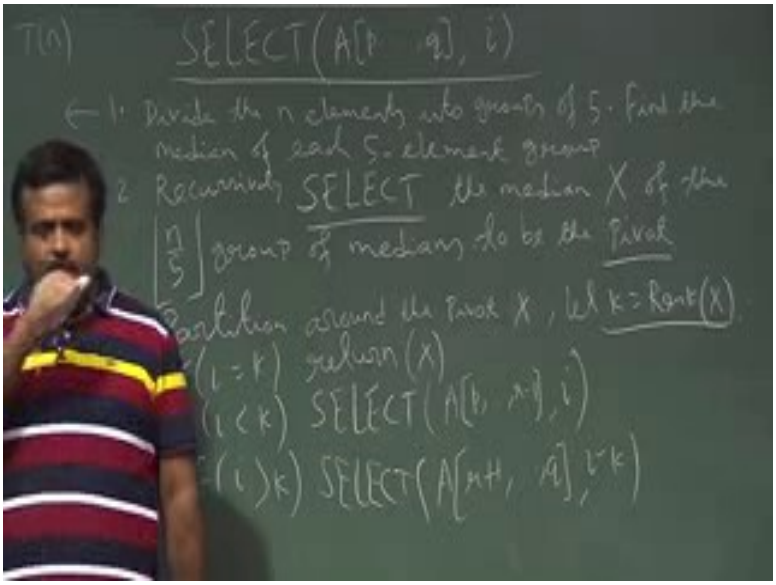
So this is 3, so 3 n/10, and this is also 3 n/10. So, x will be somewhere in this here I mean we do not know where will be x, but this is guaranteed. So, this is the good pivot because we are guaranteeing that there will be some portion which are less than x, 3 n/10 that portion size is 3 n/10 and there will be some portion which is greater than x, 3 n/10 and x will be sitting somewhere here. So, if we take that x as a pivot and then if we call the partition, then it will be like this. So, x will be sitting somewhere here and this. So, this is the guaranteed that the partition would not be 0 is to n - 1. So, we are ensuring that there will be some portion which is less than x, there will be some portion which is greater than x.

So, that if we can ensure then if we call the partition algorithm by choosing that x, then it will not give us the worst case scenario like 0 is to n - 1. So, it will give us some fraction left side some fraction right side. So, that will give us the recurrence. So, we will talk about that. So, this is the idea to choose a good pivot. So, here also we are involving some time. So, this should be taking consider into the algorithm. So, that will write the pseudo code for this, but let us just recap this. So, this is what we are doing we have a n element.

We are partitioning we are dividing the element into 5 element groups, and then we choose the median of each groups maybe there are 5 elements. So, we can just simply sort this and we can find the median. And then we find the medians of the medians. So, we will write the pseudo code for this and then that we are going to choose as a pivot element and then we call the partition by taking this x as a pivot and then this will partition into it will avoid the worst case scenario and then we have the same select.

So, we will find the rank of this. So, this is the idea behind to choose a good pivot recursively. So, let us write the pseudo code for this algorithm.

(Refer Slide Time: 15:31)



So, this algorithm we refer as say this is also SELECT. So, we have n elements say 1 to n. So, what we are doing let us write this in English description. So, we divide the array divide the n element into group of 5 element; groups of 5 and then we find the median of each group of each 5 element group then.

So, we got the median of the each group then we find the medians of the medians by using the same SELECT algorithm. So, that is the recursive call. So, use the same select to find the medians. So, we have how many elements how many medians. So, we have n/5 groups so. So, now, we have n/5 medians. So, now, we want to find the median of the medians. So, we call the same select to find that. So, we recursively call the select same function to find the median and that is our pivot x of the n by 5 group of median.

So, we have medians of medians; I mean here we can use the lower ceiling because n may not be a multiple of 5 median to be the pivot. So, x is this medians of medians is our pivot and that we are finding using the recursive call of this same select function. So, we will talk about time complexity. So, now, this is our pivot element X. Now we use this pivot and then we call the partition algorithm and so, we partition the array around the pivot sorry we partition around the pivot X and then let k be the rank of X. So, if we call the partition it will return the position. So, suppose it is we call the partition.

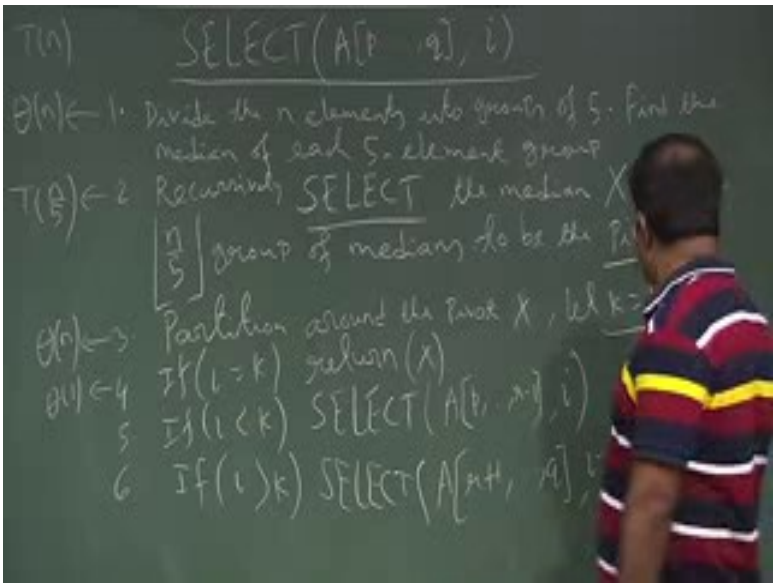
So, suppose this is this is 1 to n, but in general it is p to q. So, it will return the position of x and then r - p + 1 is basically our k. So, k is the this is the i th r th this is the index of the pivot element. So, r - p + one is basically this is the rank of x, our x is sitting in the sorted array. So, now,. So, this is the r will be the r is coming from this partition algorithm and now. So, we know the rank of x now the code is same, if i is equal to k then we are lucky we return x, because we got the i th smallest element otherwise if i is less than x then we have to call the select again.

So; that means, r is less than x means we have to look at the. So, it is dividing a array into two parts, we have to look at the left part of the array. So, A recursively call select from a. So, if it is p to q. So, here it is say p to q and if the partition is returning r. So, this is basically p to r - 1 comma i else, if i is greater than k then we call the select. So, we have to look at the right part of the array. So, this portion of the code is similar to the select algorithm. So, only thing here difference is the choice of this pivot element.

But only thing this is the way we choose a good pivot. So, that is the idea. So, we have to choose a good pivot in order to avoid the that partition 0 is to n - 1. So, this is the way we choose the good pivot. So, let us write the time complexity of this. So, how much time we are spending here. So, for that let us just write the T n be the time complexity for this run time. So, we divide the element into 5 element groups and find the median. So, how much time we are spending there. So, basically, we have n elements.

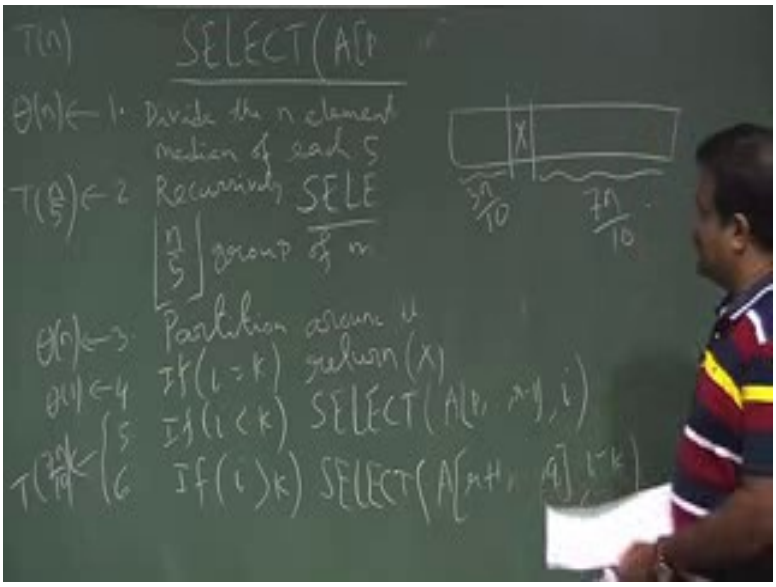
So, we are divided into 5 element groups. So, how many groups? So, n/5 and each group we are finding the median. So, we can just sort this 5 elements.

(Refer Slide Time: 21:51)



So, this will take constant time. So, there are n/5 such groups. So, this is a theta(n). So, now, recursively we call. So, now we need to find the medians of this medians. So, that for that we are again calling the same select. So, that will take us theta of n/5 a T(n/5) because we are calling the same function recursively T(n/5). So, then this step will take the partition, partition will take the linear time theta(n), now if you are lucky enough we will get this as x, but otherwise we have to go for either left part of the array or right part of the array this is the conquer step. So, we have to call either left part of the array right part of the array. So, for that what is the size of that maximum size?

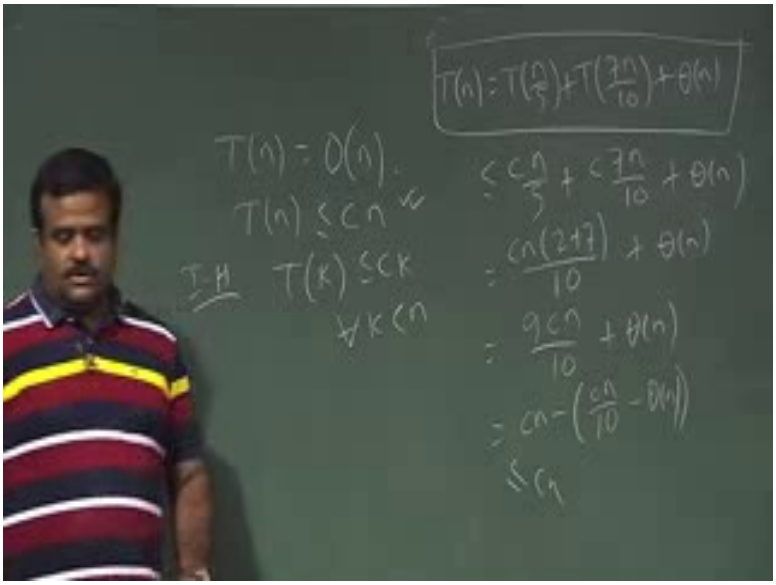
(Refer Slide Time: 22:57)



So, it is basically dividing the array into two part, we know that this part is 3 n/10 and in the worst case suppose x is sitting here, and we are going to call the right part of the array.

So, this is roughly 7 n/10. So, this is basically in the worst case this is basically T(7n/10), because in the worst case we have to go to the right part which is the biggest one this is the worst case analysis.

(Refer Slide Time: 23:38)



So, the T n is basically sum of this. So, T (n/5) + T 7 (n/10) + theta(n). So, this is the recurrence we got for this select algorithm. So, now we have to get the time complexity for this, we need to solve this.

So, how to solve this? So, we can try for recursive tree to get the solution otherwise we can try for substitution method to get the solution. So, let us try for substitution method. So, we are thinking that T n will be big O of n. So, this is our assumption so that means, we are thinking that T n must be < or = some c n some c constant. So, now, we need to take a substitution method. So, we need to take an induction hypothesis. So, we assume T of k must be less than c of k, for all k less than n and then we need to prove that T of n is less than this.

Now, we have this recurrence. So, this is n/5 which is less than n. So, we use this induction hypothesis this is c n/5 + we have again 7 n/10, which is also less than c n by which is also less than n. So, we have this n by induction hypothesis we can just write c 7 n by 10 + theta

of n. So, c n we take common. So, this is 2 + 7 + theta(n). So, this is basically what? So, its nine c n by 10 + theta of n. So, this is basically you can write is at.

So, we want to write this we want to show this less than c n. So, to show this we have to write this to be c n - (some quantity) and that quantity has to be positive. So, what is that quantity? So, basically c n/10 - theta(n). So, we choose c in such a way that this will be positive. So, c is in the our hands we can choose c such a way this will be positive and then if once this is positive this can be written as c of n so; that means, this is established. So, this is substitution method.

(Refer Slide Time: 26:48)

So, we can just write T(n) is less than c*n. So, this imply by the substitution method theta of n. So, that is the worst case runtime for our select algorithm and this is not a randomized algorithm, this is just to get the pivot as a good pivot, we choose the good pivot in that way. So, this is the guaranteed. So, guaranteed worst case run time is linear by choosing that even we can establish this by using the recursive tree. So, one can draw the recursive tree and can see that the run time will be also linear.

So, this is the worst case linear time order statistics finding the i th smallest element, and this is the guarantee this is not the expected this is not the average case this the worst case linear time algorithm for finding the i th smallest element.

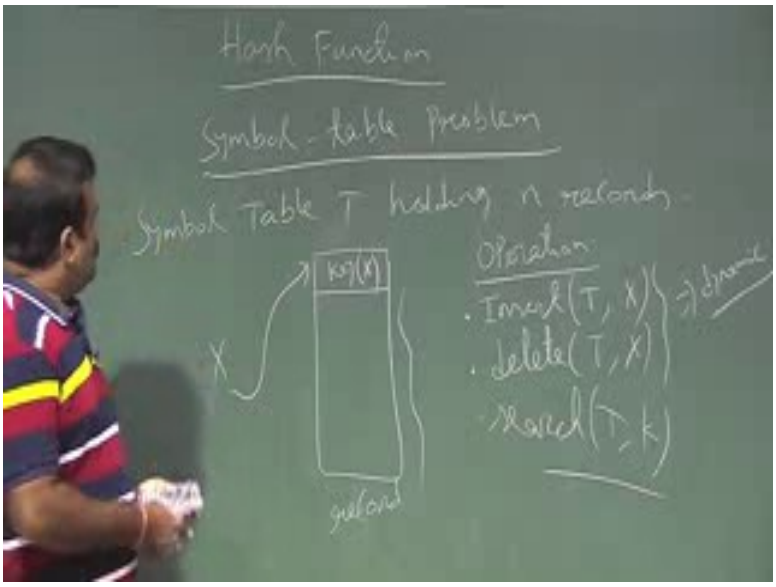
Thank you.

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

Lecture – 21
Hash Function

So we will start the hash function will talk about hashing. So, the problem comes for the what is called symbol table problem.

(Refer Slide Time: 00:29)



So, what is the symbol table problem? Basically we need to store. So, symbol table T basically holding n records. So, we need to store the n records. So, what are the record basically? So, record is having few fields and which is say x is the pointer pointing to this record, and among this field there is one field which is referred as key of x which is basically unique and remaining are some other data field may be satellite data or something, but one field is unique identification of this record. So, this is one record. So, maybe this is a student record. So, student has roll number, name, age, cgpa, sgpa, address.

So, these are all fields, but we need to find out one field which would be use for unique identification of the student maybe student roll number or student pan card number. So, that is the key of x. So, the problem is to maintain n records. So, we need to store n records in a table. So, that is the problem. So, that is called symbol table problem. So, now we need to

have a data structure to store to maintain this record, such that we should able to perform few operations.

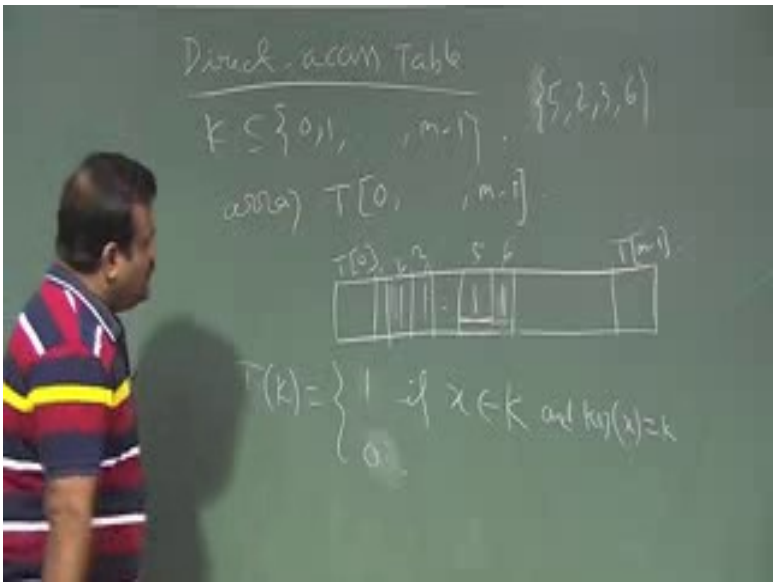
So, what are the operations on T? So, basically there are three basics operation insertion. So, we should able to insert a record in the table and we should able to delete a record from the table. So, this two is the dynamics, this two will make the record dynamic. So, basically we have dynamic state of records. So, we have already seen one dynamic set where in the priority queue when you talk about heap data structure to which is basically the implementation of the priority queue.

So, there we are having a set a which is a dynamic set. So, any point anytime anybody can join and anytime anybody can leave. So, we should able to have that query maximum of that set, if it is max we take max heap or minimum of that set or we should able to decrease something. So, this way, it basically makes this set dynamic. So, we need to maintain set of n records, and this set is dynamic in a sense that any point of time anybody can join into that table or anybody can delete and we should able to search a record that is very important query.

So, we should be able to search a record. So, key value is k. So, given the key value k we should able to find whether that record is there or not. So, given a student id we should be able to find out if the student record is in our data base or not. So, this is the problem. So, this is the problem called symbol table problem. So, we need to have a data structure for this. So, we need to store the n records in such a way that we should able to perform these operations.

So, we should be able to insert a new record, we should able to delete a record. These two operation makes this say dynamic and we should able to search and this should be in a faster way. So, we need to have a data structure for this. So, let us just think about what is the data structure we can use for this. So, let us start with the very simple data structure, but very powerful- array. So, this is called direct access table. So, here we are assuming the keys are coming from some set of values.

(Refer Slide Time: 05:06)



So, we are assuming that the keys are coming from this set 0 1 up to m - 1. So, this is the. So, maximum value of the key is basically m - 1. So, this assumption is required. So, then basically we set an array it is simple array m dimensional array like this. So, we basically have an array. So, this is T 0, T n - 1 and this is direct access. So, now this array, if a record is there in the table, then we put that value on other way sort.

So, T(k) is basically the record we got if x belongs to K and key(x) is basically k. and otherwise it is nil. So, we put this 1 if the corresponding say this is 5, if say suppose at some time our record set is this say 5, 2, 3, 6 suppose this is our record set. So, 5 then 3 must be here.

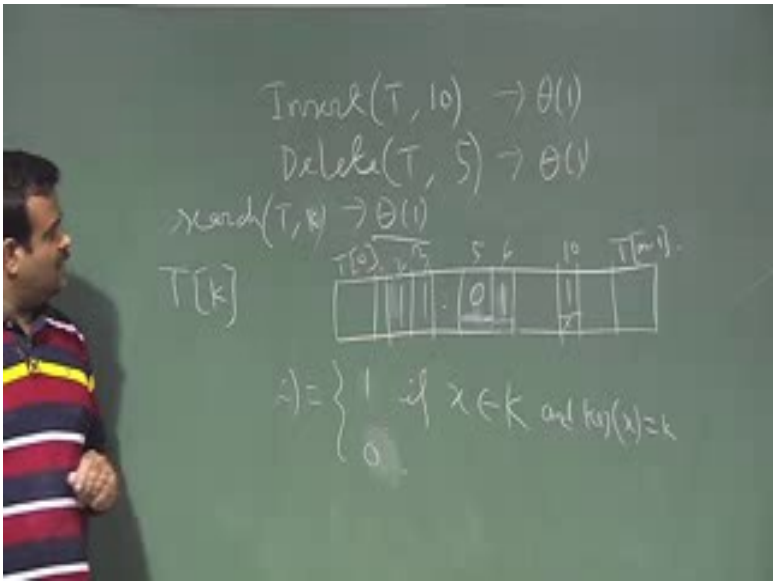
So, these are all 1 then say this is 2 2 2 1 0. So, these are all 1. So, this is just a bit vector 0 1. So, if that particular record is there key value then we put it one otherwise we put it zero. So, that is it very simple data structure this is called direct access table. So, 0 1 bit we can just maintain this array by 0 one bit if the record is, but somehow we need to have some information about the record where from we can get. So, we can store some pointer of that record. So, this will indicate the record is there are not.

If the record is there we can have some pointer to access that record, but anyway those are implementation issue, but we are just concerned about the presence or absence of the record. So, if the key value is the key value 5 is there; that means, we have a record present whose key value is 5 so; that means, this is one and then we can maintain a field over here, which

will give us the exact address or the pointer where we will get the face the record I mean the whole record whole data ok.

So, this is the idea, this is the idea of the direct access table. Now if we use this simple data structure, now what is the time complexity for those operation how to insert a element, suppose we want to insert say 10, we want to insert a record whose key value is 10.

(Refer Slide Time: 08:31)



So, what will we do? We go to this 10 and we put it and somehow if we are maintaining that we will put a x over here, that will corresponding to this record. So, that is basically linear time operation I mean not linear constant time operation.

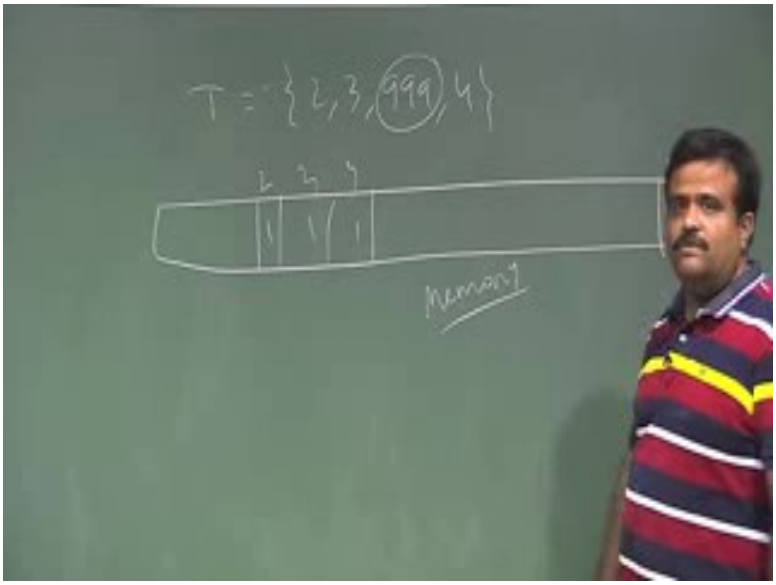
We are going to that particular field and we are putting the switch on and that is it. And deletion is also similar way if we want to delete say 5. So, what we are doing? We are going there we are putting 0 we are making this empty. So, deletion is also very constant time and searching a record suppose we want to search a record whose key value is say 3. So, what we do? We go to that position; go to that array position. So, search k say T(k) now if T(k) is O(1).

Then we got the record and somehow if we have the information about the physical record, we go to that position and we get the record. So, T(k) depending on the value of T(k), if it is 0 then the record is not there. So, it is simply say no record is not there or if the T(k) is one we

got the record. So, this is the search. So, search will also take theta 1 time. So, this is all are constant time operation this operation can be done in constant time.

So, this is very simple data structure, but very powerful this is just a 0 1 bit vector, but this has a problem this data structure has a problem in the sense that the memory problem; because suppose say we know the number of records will be less.

(Refer Slide Time: 10:36)



So, may be maximum we can have 6 7 records or something more, but the size of the record is more. Suppose we have at some point of time we have this size, so 2 3 9 9 9 and 4.

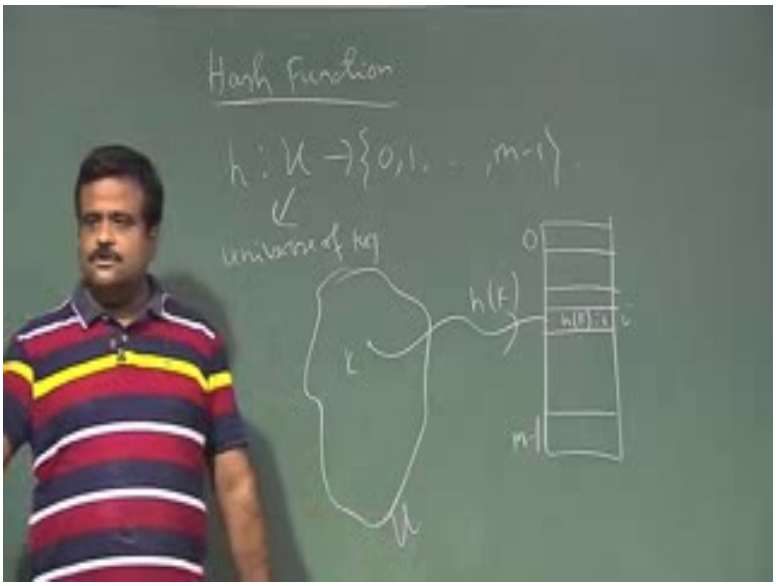
So, at some point of time this is the snapshot of the dynamic set. Now we are allowing the record size to be this. So, for that we need to maintain the array of this many long size although we have only. So, 9 9 9 may be longer than that, but we have only using few bits. So, that is the memory problem because if the value is more if the value of the record key value of the record is more although the number of records is less then it is the wastage of the memory.

So, this is the major drawback of this simple array data structure, although this is very powerful this is just a 0 1 bit switch on off simple very simple data structure, but this is a problem with the value we are allowing for this key value. If the key value is we are allowing more then we need to maintain a this is statically static allocation this array. So, we need to

have this data structure, we need to have the array size up to this the maximum value we are allowing for this key value.

So, this is the drawback although our number of record is less. So, to avoid this drawback the solution is to have a function which is called hash function.

(Refer Slide Time: 12:18)

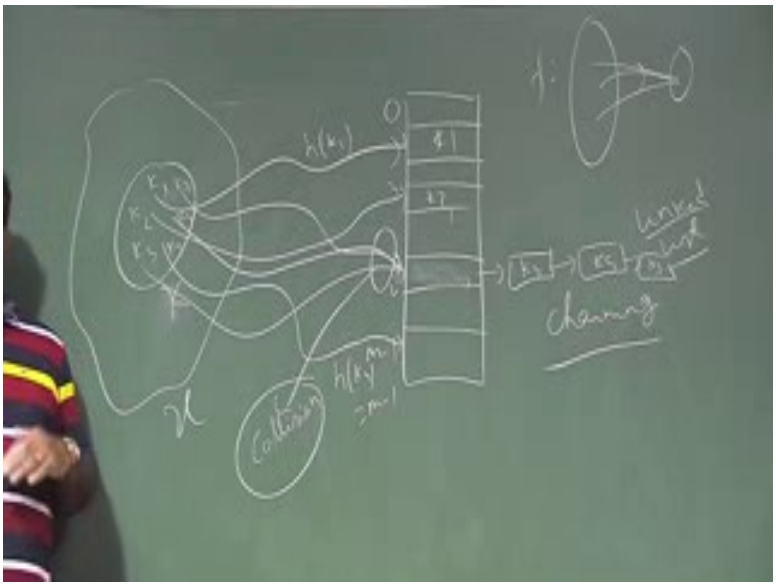


So, basically if hash function is a function from u to this set. So, this is the universe of the key, universe of key and this is the slots I mean this is the table size. So, we have say table of. So, there are m slots.

So, we have a table or this is a simple array and this is the universe of the key. So, set of all possible keys is u. Now hash function is a mapping from u to this set. So, if you take any key from here and if we apply h(k) it will reach towards a slot over here say i, i th slot. So, this basically h(k) is i. So, any such function is called hash function. So, it is basically taking a key and it is giving us a slot basically, it is giving us a value from 0 to suppose we have given.

So, we are allowed to have table size up to m. So, 0 to m - 1. So, our hash function will be we take a key from here and it will map to a particular slot from 0 to m - 1. So, any such function is called hash function. Now suppose we have a hash function then how we can maintain a record. So, so let us draw this ok.

(Refer Slide Time: 14:30)



So, this is a function from key to this say 0 to m - 1 and this is the set of all possible keys and this is the set of keys of our interest.

So, this the key set we have so far. So, here we have some k 1, k 2 some keys are there, k 3, k 4 like this. So, now, we have to maintain that stable. So, what we do we apply the hash function on k 1 suppose it is mapping here. So, h of k 1 is basically one say this is 1. Now k 2 say h(k2) is basically say three something like that. So, this way now h(k3) is basically say some slot here i th slot say i. So, this is basically h(k3) is i now suppose h(k4) is basically say some slots here.

So, h(k4) is say m - 1 something like that. Now suppose we have a k 5 whose has value is say this. So, suppose h(k5) is also mapping to the same slot i. So, then we have a problem and this is what is called as collision, this situation is referred as collision. Collision means suppose we have a 2 key which are going to map into a single slot and that is quite possible because if the slot is if this set is small and if this set is bigger.

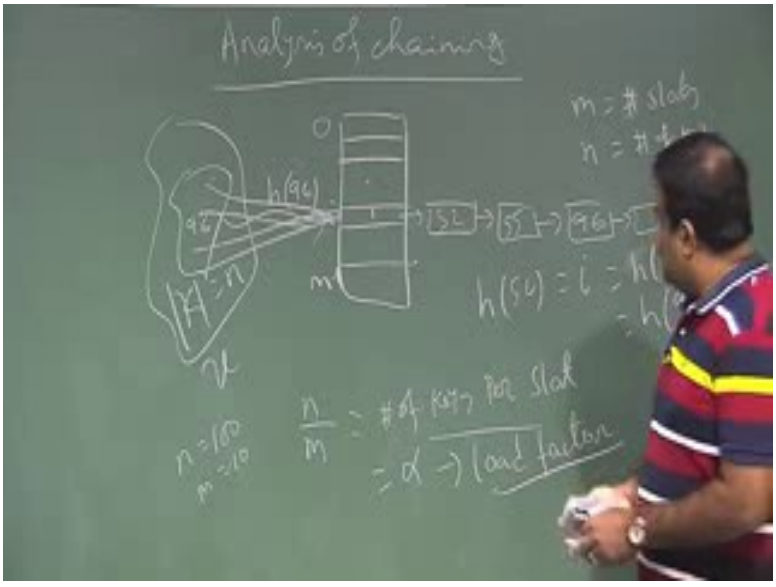
So, if we have a function say h is a function if we have a function from a bigger set to smaller set. So, then there has to be collision. So, collision is quite natural in the situation of the hash function because usually this set is smaller set. So, basically hash function is basically as a compression function. So, we have a big length input. So, we convert into small than output. So, that is the compression function. So, since this is a smaller size this co domain, this is the domain this is the co domain this smaller. So, they are has to be collision ok.

But now the question is how we can handle this collision. So, collision will be there. So, how to handle this collision because here h, so h 3, the k 1 is here k 2 is here k 4, sorry k 2. So, k 3 and k 5 both is colliding to the i th slot, but they cannot sit in the single slot here. So, then what is the solution. So, we have to do some sort of chaining over here. So, because there are because this is a position for only one guy this is a room for one.

So, we can have a something what is called chaining or linked list. So, k 3 then we can have k 5. So, this is called chaining method to handle the collision. So, this is basically linked list, if a slot is containing more number of keys then we will put that outside the table will put a chain linked list, this is basically linked list. We will put a linked list outside the table. So, this is the way we just handle the collision ok.

So, if there are another say k 7 is also if say k seven is also colliding here if the k 7 then we have a k 7 over here like this. So, if a slot is containing more than two keys I mean more than one keys, then we have to use this chain we have to use the linked list for this. So, this is the collision. So, now this chaining is a method to handle the collision. So, now we want to analyze this chaining method how good this is ok.

(Refer Slide Time: 18:55)



So, this is the analysis of chaining. So, chaining is the method to handle the collision. So, now, suppose there are m slot 0 to m - 1, and suppose there are this is the k the suppose there are k keys n keys there are n slot m n keys. So, so m is the number of slots and n is the

number of keys. So, now, it is a k 1 k 2 k n. So, now, among this if there is a collision suppose this slot i th slot having collision say 52 say 55 96 like this.

So, all of this h(52) = i = h(55) these are the key value h(96) all are mapping to same slot. So, this is basically the chain in that slot if there are say 3 keys are colliding there. Now what is the worst case of this, now how to search key? Suppose we want to search a key suppose we want to search say 96 is there or not. So, what we do? We apply the hash function on it. So, it will first map to the i th slot then we know there is a chain.

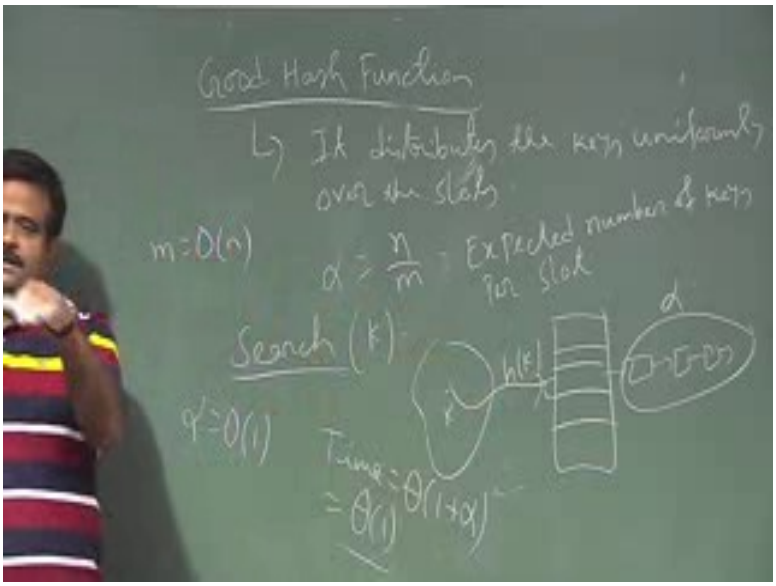
So, we have to read the chain basically. So, basically we are just scanning the chain. So, that is the way we search a key. Now what is the worst case for this chain? So, worst case is now suppose all these elements are colliding in a single slot. So, there are n keys all are colliding here then this is the worst case then the search time will be order of n because if it is colliding in to this slot then we have to because all are colliding in the same slot. So, there is a chain of size n ok.

So; that means, when you search that key is colliding here we need to search this whole list and this list is not sorted we are not going to sort this list then it will take some more time to sort. So, it is just a unordered list linked list. So, we need to search our key in this linked list. So, that will take linear time that will take the time of the depending on the size of the list if the size of the list is linear the time complexity is linear. So, that is a bad hash function. Bad hash function in the sense that everybody is colliding in the same slot.

So, there are n keys all are colliding in the same slot that is a bad hash function. So, what is the good hash function? If we have uniform distribution of the keys over the slots, so, there are n slots. So, if n slot is distributed over this and if there are n keys, if n keys are distributed over the slot uniformly. So, then n/m is basically what n/m is basically. So, there are n keys n slot if there are 100 say keys and if there are 10 slot. So, n/m is 10.

Basically then 10 is the number of keys per slot expected number of keys I mean. So, this is basically the number of keys per slot. So, this will happen if our hash function is such that it is distributing the key over the slot uniformly, it is not that all the keys are going to a single slot it is just we have n keys it is distributing the keys among the m slot uniformly. So that means, each slot will get n/m keys. So, this is called load factor this is referred as alpha this is called load factor ok.

(Refer Slide Time: 23:48)



So, this is a condition this is a criteria of a good hash function. So, a good hash function should such that it should distribute the key. So, it distributes the key distribute the keys uniformly over the slot. So, there are n keys m slot. So, that means, each slots should get same number of keys i mean the distribution is. So, given a key, it will be in one of this slot its probability is 1/m it is equally likely.

So, then this alpha is the load factor n/m, alpha is the expected number of slot or expected number of keys per slot. So, in that case if we have such a hash function in that case search time will be how much. So, you want to search a key. So, to search what we do. So, this is a say key we are going to search. So, we first apply the hash function on this. So, this will map to some slot i th slot ok.

Now, we know in the i th slot there is a linked list or chaining and this size of this chain is alpha. So, we have to just scan this. So, what is the time complexity for this? So, time is basically 1 + alpha. Since one is the time to apply the hash function and then this alpha is the basically the load factor; that means, the number of keys per slot now if our hash function is a good hash function then this is the scenario and now if alpha is order of one.

So; that means, if the n is order of m or n is order of n then alpha is 1 then this will be constant time. So, if our hash function is a good hash function in the sense that it distribute the key uniformly over this slot then alpha will have a alpha is the expected number of keys per slot then this is the time for searching a key or insert a key because we first apply the hash

function this will take one, and then + alpha is the size of the list we have in that particular slot.

So, this is the idea of the hash function and this is collision is there and collision can be handled by the chaining, and next class will talk about how to construct such hash function. So, that it will be distribute the key uniformly over the channel. So, while we construct the hash function this should we should keep this in our mind that it should distribute the keys uniformly over the slots.

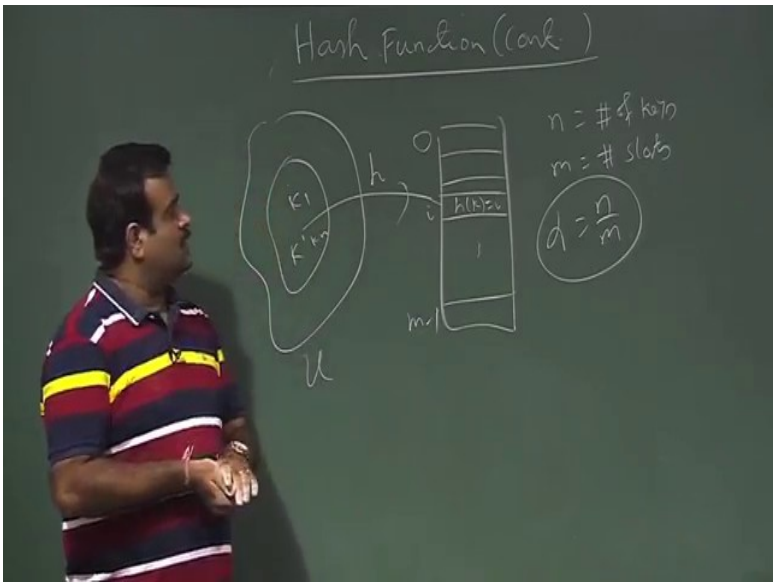
Thank you.

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

Lecture - 22
Open Addressing

So we have seen the hash function, which is the function from universal set of the keys to a given slots.

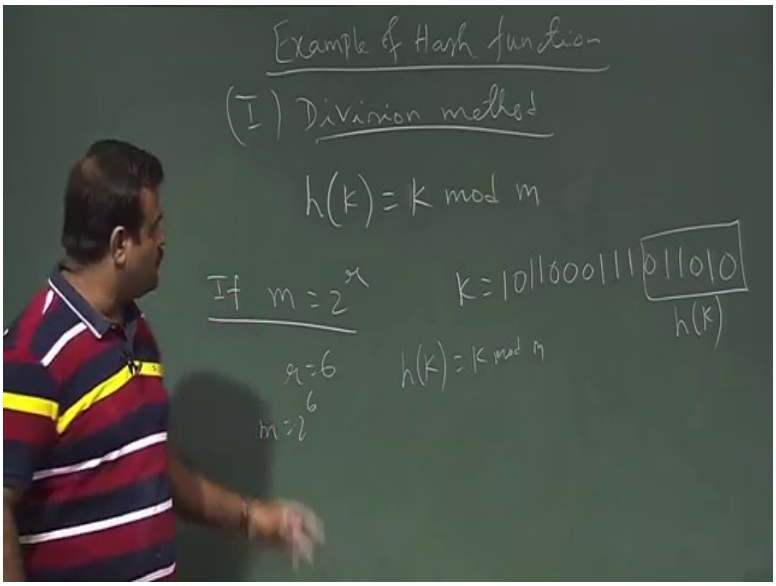
(Refer Slide Time: 00:25)



So, suppose 0 to m - 1 slots. So, it is a function from u 2. So, you are given a key and if we apply this hash function it will map to a particular slot. So, h(k) is a i. So, now we say that we know hash function is good hash function, if there are same keys n is the K 1, K 2, K n and there are m slots.

So, if the keys are distributed in uniformly over this slot. So, n is the number of keys and m is the number of slots, then alpha n/m which is basically expected now this is called load factor, which is basically expected number of keys per slot. And then we call this hash function is a good hash function, because it is distribution in the key uniformly over the slot. So, now in this lecture we will construct few hash function and we will see whether this is a good hash function or a bad hash function. So, let us talk about some example of hash function, how we can choose a hash function.

(Refer Slide Time: 01:53)



So, example of hash function. So, first one is the division method. So, here we are assuming our keys are basically integer. So, we have integers; all the keys are integer basically and we have a slot of size m which is also an integer. So, we take an integer. So, we need that after applying the hash function it should be 1 of this slot. So, what is the function we can use for that what is the obvious function we can think of that is modulus function yes so; that means, h(k) is basically a k mod m, were k is a integer m is also an integer. So, if you take the mod, mod mean if you divide this k/m k is key, key could be any big integer.

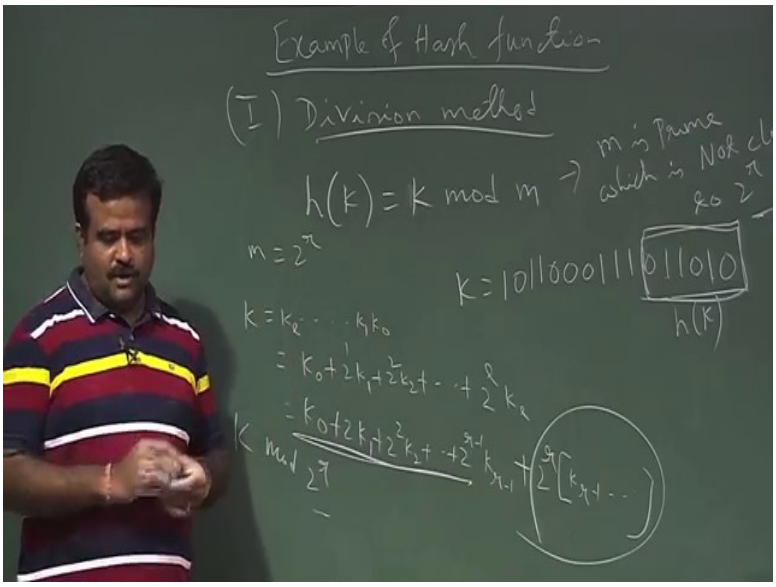
So, any big integer we take and we try to divide this by m and this is the reminder. This is the reminder and this reminder will be form 0 to m - 1. So, this is the hash function, now we will talk about how this hash function is. So, whether this is the good hash function or bad hash function; that means, whether it is distributing the keys uniformly over the slots or there is huge number of collision for this. So, if our m is 2^r, then we have a problem what is that problem.

So, if n is 2^r. So, then suppose we have key say k which is a integer and we convert in to binary say. So, suppose this is 1 0 1 1 0 0 0 1 1 1 0 1 1 0 1 0 suppose this our key, we have an integer we convert into binary. You know how to convert into binary we try all we divide by 2 and then again 2 like this. So, we will get all this component. So, this is basically the binary representation of our integer. Now suppose k is basically 2 to the power r say r is 6 here say.

So, for this r is a 6. So, m is basically 2 to the power 6. So, your if r is 6 then what is h(k)? H(k) is basically k mod m. So, 1, 2, 3, 4, 6. So, this is basically our h(k).

If we choose r to be 2 to the power m to the 2 to the power r, I mean here r is 6 then this is basically h. So, does not matter what the values are so; that means, if we take any bit differ over here. So, they all are colliding to the same. So, if we fix this, why this is happening because if you take mod of this, mod of 2^r then we take up 2.

(Refer Slide Time: 05:38)



So, this is basically we can just try to write this. So, this is basically if k is say, k 0 k 1 dot dot dot k l. So, this is basically k 0 + 2 k 1 + 2 k 2 this is binary dot dot dot 2 to the power sorry 2 square.

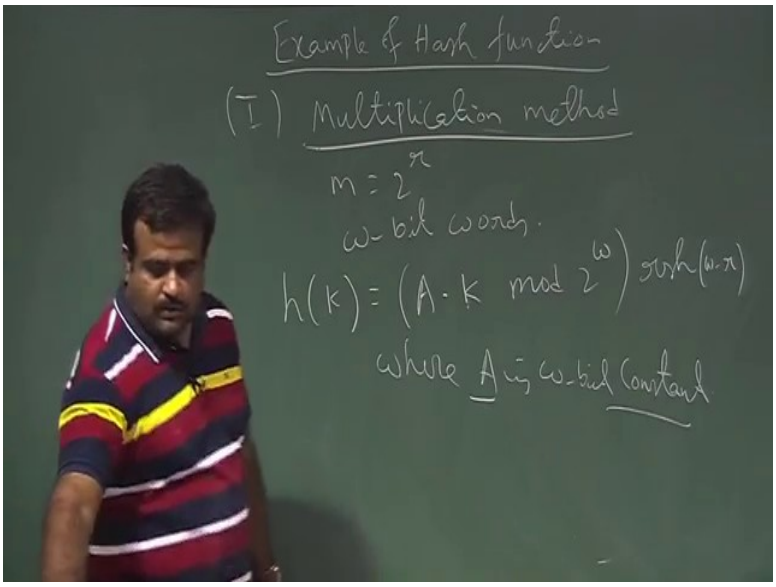
2 square k yeah 2 to the power l k l. So, now, this can be written as k 0 + 2 k 1, + 2 square k 2, + 2^r - 1 k, k r - 1 + 2^r then we take the common here then kr + …. like this. Now if you take if our m is 2^r now if you take this k mod 2^r then; that means, this portion will be vanish and only these values will come. So, that is why if r is 6 this is the key mod 2 to the power 6 does not matter what are the values these are. So, if we change these values any one of these values, by fixing this values all are colliding into the single slots.

So, there are huge number of collision is happening. So, it is not distributed uniformly over the slots. So, this choice of m is not a good choice. So, this is not a good hash function in that sense if we choose m to be like this. So, usually for this we choose m to be prime, which is

not close to some 2^r or 10^r; because we may consider into the decimal system also. So, that is the choice of m 1 should be considered if we are having this division method. So, division method have this issue that we should not choose m to be 2^r then we have a huge number of collision ok.

The second example is the multiplication method. So, there we are allowing m to be 2^r. So, multiplication method this is another example of hash function ok.

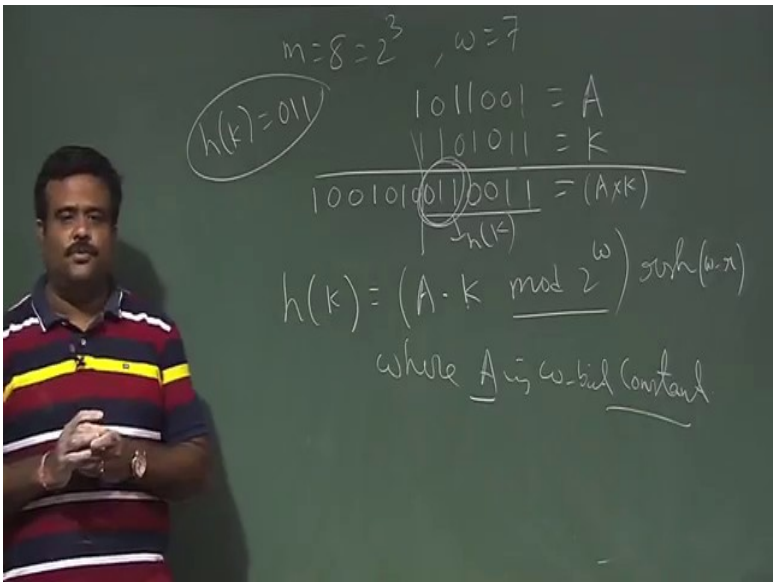
(Refer Slide Time: 08:30)



So, what we are doing here. So, basically we are allowed to choose m to be 2^r and here we are our numbers are said all are say w bits. So, our word at w bit word. So, our key is w bits. So, our computer is w bits. So, 64 bit computer nowadays. So, 60 w could be 64 ok.

So, our key is w bit. So, h(k) is basically a into k. So, a is the constant again up w bit we have to choose that a, mod 2 to the power w this is into and then left right shift how many time w - r times, right shift w - r time. So, where A is a w bit constant. So, if our computer is 64 bit, we choose a 64 bit constant A and then we choose a key which is again a 64 bit, w is 64 say then we multiply these 2 then it will 2 w bit now we want to make it again w bit. So, that why we take mode 2^w. So, it will again become a w bit. So, the last w bit and the we will do a right shift w - a t times. So, this is basically called the multiplication method, we will take an example for this suppose.

(Refer Slide Time: 10:42)



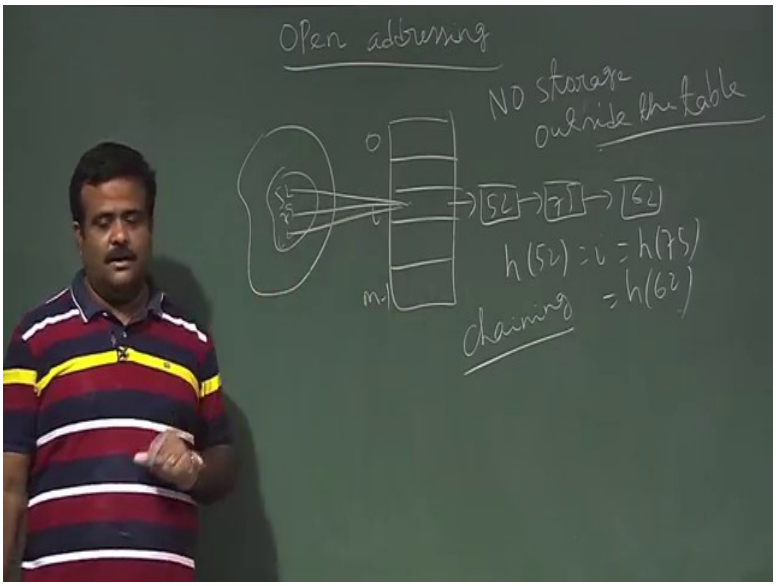
So, let us take an example of this suppose say m is 8, 2^3. So, we have 8 slot. So, 0 to 7, and our w is say 7 bit suppose our computer is 7 bit computer I mean whole computer ok.

So, now we choose a, a is also 7 bit we have to choose a say a is constant we choose 1 0 0 1. So, this is our A set. Now we choose a e which is also say which is also 7 bit and we need to apply the hash function by this formula. So, we let us take a key. So, 1 1 0 1 0 1 1. So, this is our k set. So, now, we apply this we multiply this A into K, if we multiply this this will be this is 7 this is w bit, this is this w bit, then the result will be 2 w bit this is 7 bit this is 7 bit. So, result will be 14 bits. So, if you do that it will be like this 1 0 0 1 0 1 0.

Then 0 1 1 0 0 1 1. So, this is again fourteen bit now we have to take mod of 2^w. So, this is up to 7 8 1 2 3 4 5 6 7. So, this is the mod of 2^w, and then we will do the right shift w - r times. W is 4, r is 3 where w is 7 r is 3. So, fourth time if we sweep these 4 times, then what then this is gone this is gone, this is gone. So, this is our basically h(k) this 3 bit. So, h(k) is basically here 0 1 1 this is our h(k). So, this is the multiplication method.

And here we are allowed to take m to be 2^r and we have seen that people have seen this is sort of distributing the key uniformly over the slots. So, One can have some more example of hash function, but this is the way we choose the hash we have discussed 2 method one is division, another one multiplication method ok.

(Refer Slide Time: 13:48)



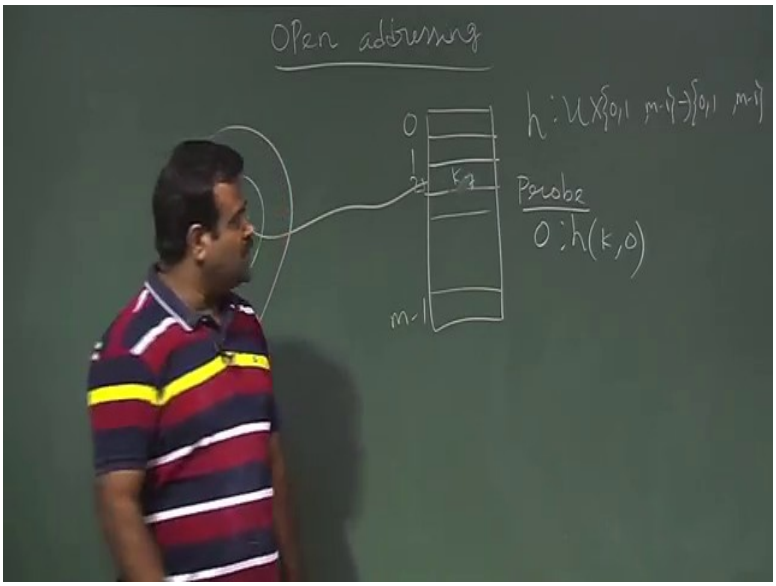
So, now we will talk about a strategy to handle the collision which is called open addressing. We have seen a strategy for handling the collision that was basically chaining. So, now, we discuss another way we can handle the collision open addressing ok.

So, in the chaining what we did? We did what. So, if we have a if we have this slots 0 to m - 1 and if we have some keys and if some keys are colliding in a slots, then we have a chain say 75, 6. So, this all this h of 52 is equal to i is equal to h of 75 is equal to h of 62. So, these are all that keys which are colliding into the. So, 52 75 62. So, these are the keys all are mapping to the same slots ith slots, and then then to this is the collision and we handle this collision by having a chain outside the table. So, this is a extra storage. So, we have to maintain a linked list outside the table.

So, suppose this is not allowed, suppose no storage outside the table suppose this is our restriction, then we cannot do the chaining? This is chaining. So, if you are allowed to do any extra storage outside the table, then we cannot have a linked list outside the store outside the table. So, the chaining cannot be applied there. So, in those extra storage is available outside the table. So, everything we have to do in the table. So, we have area we have table or area of size m, 0 to m - 1 and everything we have to do there. So, that problem we have to address by open addressing. So, no extra storage.

So, chaining is not allowed for chaining we need to have a linked list outside the storage outside the table, but that is not allowed here ok.

(Refer Slide Time: 16:16)

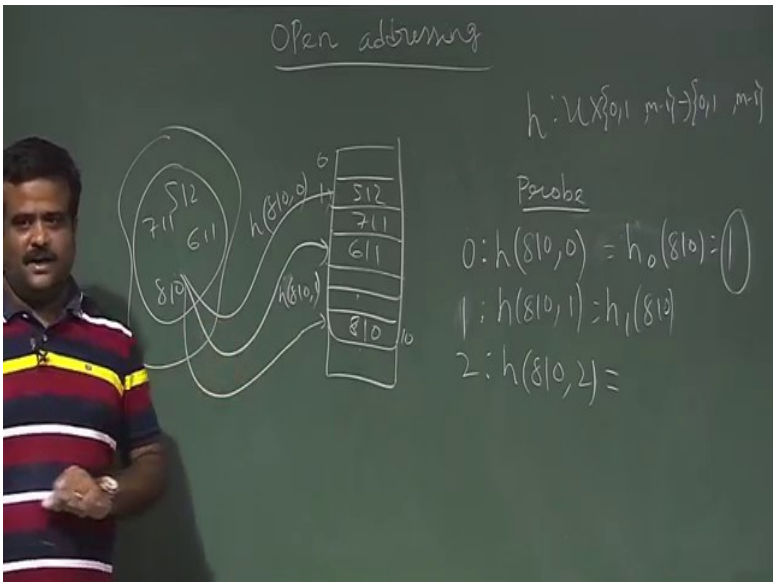


So, what is the idea? Idea is. So, basically what we have a table 0 1 2 m - 1. So, we have a key, we are mapping over here. So, now suppose it is hitting to an empty slot and there is no issue, but suppose it is hitting to a. So, suppose this is a k 10, suppose it is hitting to a slot which is already occupied then what we have to do we have to search for a next slot, because earlier method we have a had a chain, but here chaining is not allowed here no outside storage is available.

So, linked list is not allowed in the outside the table. So, then we have to find a slot which is free. So, to find that slot we cannot apply the same hash function. If you apply the same hash function, then it will again hit here so; that means, we have to apply a second hash function and then again if that second hash function hitting some occupied slot then we have to apply a third hash function. So, this is the probe sequence. So, you have to apply sequence of hash function or sequence of probing. So, we have probe sequence. So, basically here our hash function is for open addressing it is basically function for u cross 0 1 to m - 1 to 0 1 to m - 1 ok.

So, basically we first. So, this is the probe sequence. So, we first zeroth probe. So, we apply the 0th hash function and suppose it is hitting to. So, so suppose it is hitting to some occupied slot over here.

(Refer Slide Time: 18:30)



So, this is our k. So, suppose. So, let us draw some example it will be more clear, suppose this is the scenario. So, somebody is sitting here 5 1 2 6 1 1 say 7 1 1 like this suppose this is this are the key we have where 7 1 1 6 1 1, suppose we have a key say 81 0.

Now, suppose you want to insert 8 1 0 here. So, what we do we apply the. So, we try for the zeroth probe we apply the hash function on 8 1 0 ,suppose this is hitting here 0 1 suppose this is hitting here. So, this is this is say 1 which is not empty slot. So, then we have to go for a next probe. So, this is basically the hash function which is giving us the value 1 which is not available which is not available. So, this is the first hash our original hash function maybe then we go for the next probe, 8 1 0 this is basically h 1 of 8 1 0 as if we have sequence of hash function. So, if we have apply because if you because if you say. So, same hash function it will be here. So, we use. So, suppose it is hitting some slot which is. So, this is basically h of 8 1 0. So, this is h of 8 1 0 1.

Suppose again it is hitting some occupied slot, then we have to go for the next probe and suppose it is hitting some slot which is available. So, this is the ten slot then we put 8 1 0 over here. So, this is the idea. So, we are basically sequence of hash function. So, we have sequence of probe sequence, first attempt the zeroth probe or maybe original hash function if it is hitting to some occupied slot then we have to basically the idea is to search for a empty slot by this probing probe sequence, then we will go for the next hash function or the first

next probe and if still hitting some occupied slot, then we will go for the next probe like this. So, this is the way we insert the keys into the table ok.

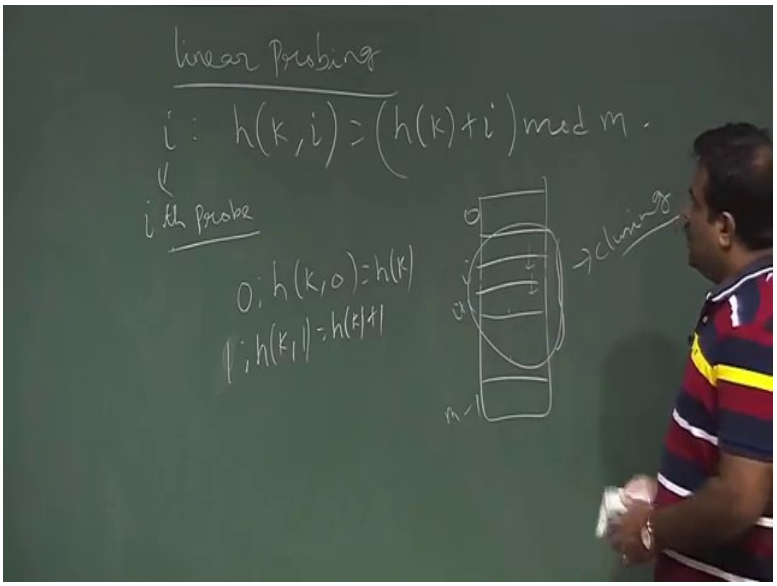
So, then our hash function is basically this u to the set of universe of the key and this is the probe sequence and we will do up to m - 1 times, and this is the slots available this this is the slots 0 to m - 1. So, this is our function basically we have sequence of hash function k 0, k, k 0, k 1 like this. So, if one function is given to some empty slot a some occupied slot we use the second hash function with the hope that we will get a empty slot and then until we continue like this we continue this probing until we reach to a empty slot. Once we reach to a empty slot we will put that. So, this is the way we insert a key ok.

Now, how to search? So, suppose we want to search say 8 1 0. So, what we do? So, suppose you are searching 8 1 0. So, what we do? So, we need to follow the same probe sequence as they have inserted. So, we first apply the zeroth probe. So, it will hit here. So, we will see whether these values 8 1 0 or not it is not. So, we will go for the second probe then we will hit here. So, we will see whether this is 8 1 0 or not it is not, then we will go for the next probes then we will see it is the 8 1 0. So, we got the value. So, this is the way we search. Now deletion is little has to be tricky here because suppose we delete this 6 one 6 1 one suppose and then after the deletion suppose we are searching 8 1 0 then what will happen.

So, we will do this probe sequence in the zeroth probe we will hit here. So, we check 5 1 2 is not is is not matching with 8 1 0 then we will go for the next probe, it is hitting to a empty slot then we stop, but this is correct because it is not originally empty somebody was there, but it got deleted. So, that information has to be kept somewhere here in a single bit. So, that we can go further otherwise if we see a empty slot we'll stop. So, in deletion we have to take care this. So, if we delete somebody then we have to put a tick over here, that this room was not originally empty somebody was sitting there and it got deleted. Otherwise we cannot find 8 1 0 if we found this is the empty the we stop, but it is not the correct one.

So, we if we see this bit it is wrong; that means, somebody was there then we go for the next probes and we will found 8 1 0. So, one has to be careful in the deletion. So, this is called open addressing. So, basically we have sequence of probes and by there we will find the empty we will try to find the empty slot in the table ok.

(Refer Slide Time: 24:37)



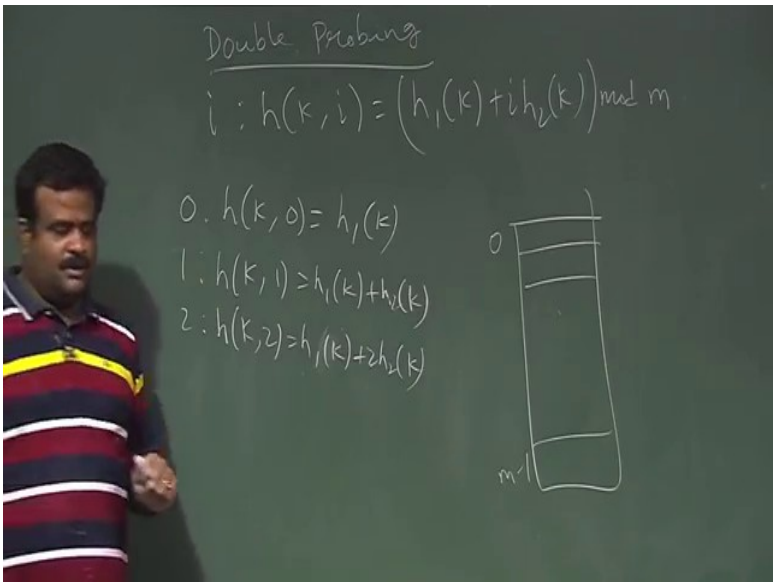
Now, we take some example of this probing there are basically we will talk about 2 example 1 is linear probing. So, 2 example of open addressing linear probing. So, in the linear probing the h(k, i). So, this is the ith probe is basically h(k + I) mode m.

So, this is the ith probe. So, this is our slot this our slot m - 1 0 to m - one. So, suppose it is hitting some occupied slot then this basically is looking the next 1 where next 1 is available or not. So, if it is available they are putting there. So, this is because we have to everything we have to fix into the table we are not allowed to have the chain. So, we just try to see the next 1 is available or not this is the way we see whether the next 1 is available or not so; that means, the zeroth probe is basically our h(x) h(k). So, we will go there and if it is not available then we go for the first probe.

That means, h(k + 1). So, suppose it is hitting the ith slot then we check the ith + 1 slot is empty or not. So, we have to take a mod because it means then we have to come back here. So, this is the linear probing, but this has a drawback because it is sort of clustering if it is hitting somebody here and if this is empty then it is basically clustering in some portion. So, it is not uniformly distributing the keys over the slots it is sort of clustering if it is hitting some occupied slot then after that it is basically making a cluster it is not distributing over this.

So, to avoid that there is another example of open addressing is called double probing.

(Refer Slide Time: 27:02)



So, in double probing basically we use 2 hash function h 1 and h 2. So, in the ith probes is basically. So, h1(k+i) h2 (k mod m). So, basically this is our table. So, in the zeroth probe what we do. So, h(0) is basically h1(k). So, we say h is our original hash function. So, we apply the hash function if it is hitting to some occupied slot then we go for the next probe then it is basically h1(k) + h2(k).

So, we have another hash function we just add it and; obviously, mod m check whether again anything a occupied slot or empty slot or not if it is not available then we go for the next flow which is basically h1(k) + 2 *h2(k) since we have already calculated h1 h2 we can use this value to calculate this we did not need to calculate again h 1 h 2 like this. So, we check this; obviously, mod m we check this whether we are getting a occupied orders. So, this way depending on the is h 1 and h 2 are good hash function distributing then this is shown to be a good hash function over the… I mean it distributes the keys in the uniformly over the slot. So, this is slightly better than the double hashing or 1 can think of it as triple hashing or some quantity hashing over here.

So, in the next class we will analyze the open addressing method and we will see how good or how bad this is.

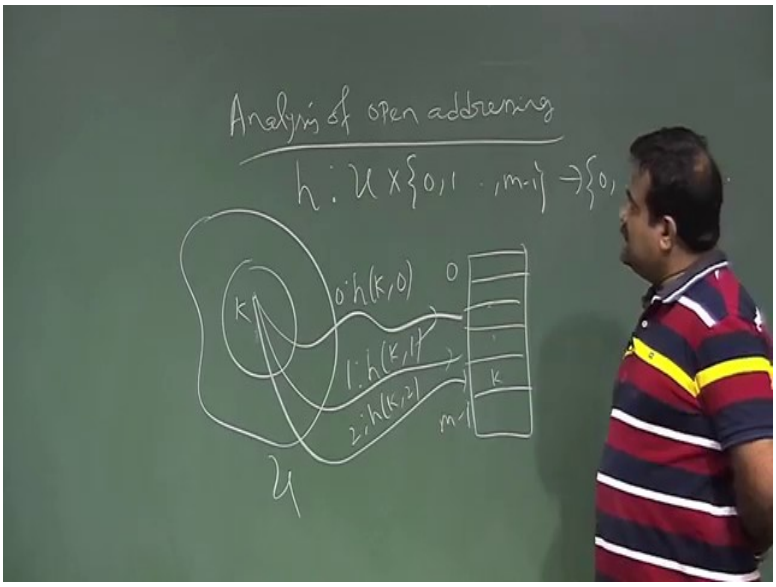
Thank you.

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

Lecture – 23
Universal Hashing

So we talked about universal hashing. So, before that let us just complete the analysis of open addressing, so analysis of open addressing.

(Refer Slide Time: 00:28)

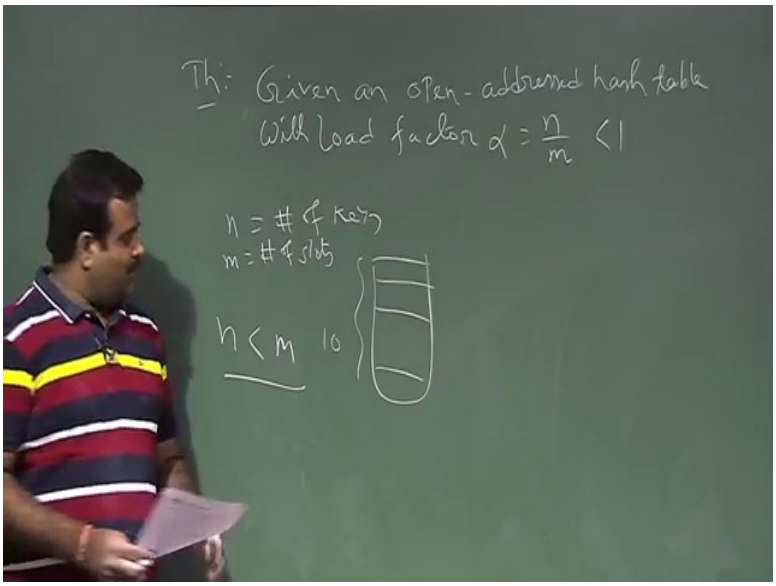


So, basically just to recap, so in the open addressing method we are just having the probe sequence here hash function is a function from u cross this is the sequence number of probes too. So, we have a hash table of size m 0 to m - 1. So, I mean this is the set of keys now if you insert a key. So, what we do we just apply the 0th probe. So, so we apply the 0th probe. So, h(k,0) and if it is hitting some occupied slot then we go for the next probe h(k,1) this is the 0th probe, this is 1th probe and if it is still hitting some occupiers slot then we go for the next probe h of k comma 2 and these way we continue until we get a empty slot. Also you get a empty slot we will insert that value k inside the k over here.

So, and we have seen to example of probing linear probing and the double probing in the last class. So, now, we will analyse is this is called open addressing. So, we analyse the, so everything we have to be feed in the table. So, there is no storage outside the table. So, we are not allowed to have a linked list outside the table. So, changing is not allowed. So, we have to

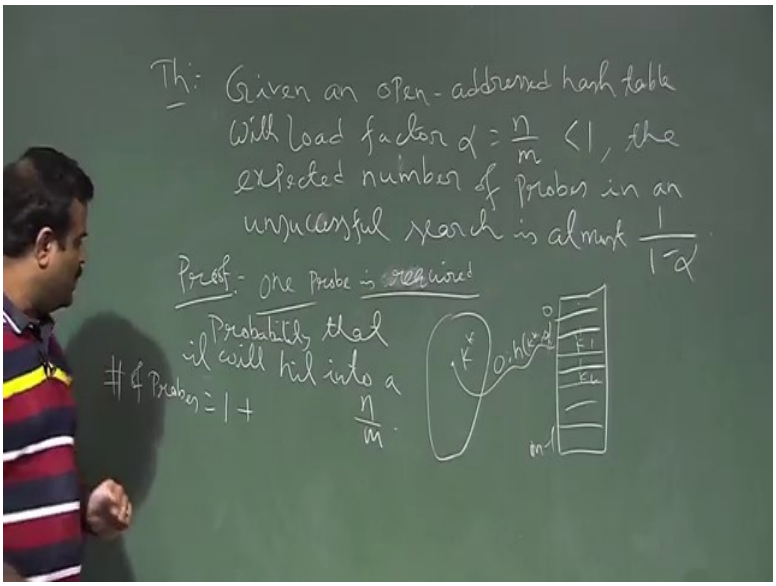
feed everything inside the table. So, basically the idea is to search for a empty slot by this probe sequence. So, we have a sequence of hash functions. So, we have applying first hash function if it is hitting some occupied slot next expansion like this. So, this is the recap of what is open addressing now let us analysis this open addressing by a theorem.

(Refer Slide Time: 03:00)



So, this theorem is telling, given an open address hash table with load factor alpha which is basically n/m and this has to be less than 1. Why it is less than 1? Because if there are, so if there are n keys, if there are m slots if the keys number of keys is more than the slot then there is no way we can feed the keys, if there are say if there are ten slots and if there are 20 keys then we cannot fit 20 keys in the 10 slot. So, we need to have the extra storage there so that assumption is mandatory. So, n is the number of keys and m is the number slots. So, n has to be less than m otherwise we cannot fit the keys in the table by the probing because if it is so, this assumption is quite.

(Refer Slide Time: 04:35)



So, m n has to be less than m then the theorem is telling the expected number of probe n for around successful search the expected number of probes in an unsuccessful search. unsuccessful search means we are searching a key which is not there in the table unsuccessful search is at most 1 by 1 - alpha. So, this is theorem what this theorem is telling. So, we have n keys we have m slots and we have a open address hash function and we have fill up these keys into the slots by using this open address hash function and now suppose we are trying to find out the key which is not there in the table. So, now what is the number probes we should have we should do this for this search. So, that is the theorem.

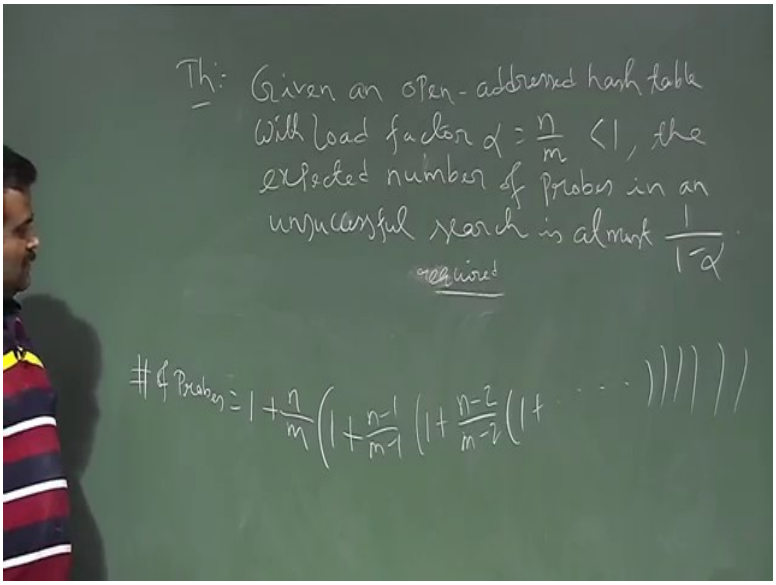
And that it is telling 1 by 1 - alpha. So, how to prove this? To prove this we will just use some probability staff. So, there are say basically there are m slots 0 to m - 1 m slots and in this m slots there are k keys I saw n keys are sitting there now 1 probe is mandatory. So, one probe, probe is required, required. So, one probes is mandatory because. So, suppose we are finding a key k a star which is not there in the table. So, for that we need to try the probe that is 0 of, 0 that is the 0th probe. So, 1 probe is required. So, if we denote this is the number of probes. So, one probe is required.

Now, when we go for the second probe if this is hitting to a occupied slots if this is hitting to a occupied slots then only we go for the second probe I mean next probe. So, now, what is the probability that it is hitting to a occupied slots. Now what is the probability that it will hit to a occupied slot now total number of slot is same. Now there are n keys live in this here, so just

the classical definition of the probability. So, if we choose one of this if it is hit one of the slot where this n keys live then that is the probability. So, basically probability that it will hit to an occupies slot is n/m.

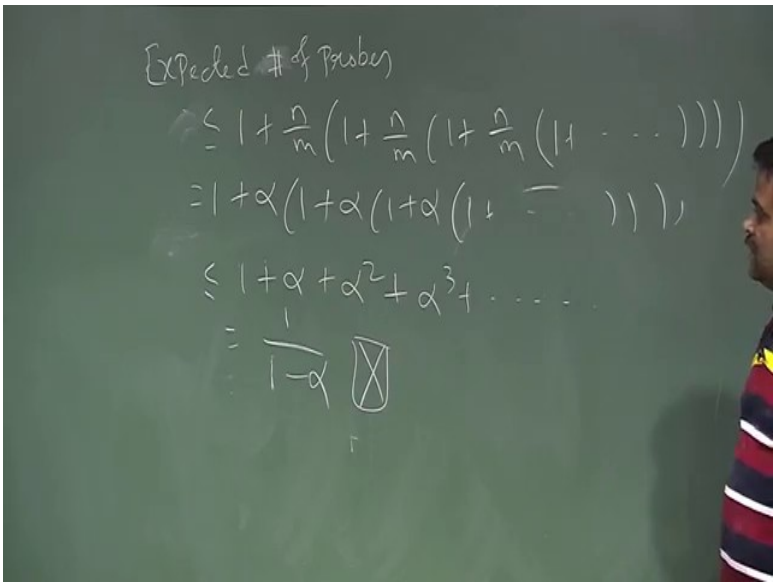
So, the probability that it will to a occupied slot is basically n/m because there are a m slot which is favourable case and there are m slots are there n keys are there in that slot and there are total m slots. So, if we can hit anyone of this n. So, that is the just classical definition, so n/m. So, with this probability n/m then we will go for the next probe so that means, we will go for next probe with this probability 1/m and then again next probe.

(Refer Slide Time: 09:14)



Then again we will go for the next probe if it is again hitting to a occupied slot and that probability is basically n-1 / m-1 and then we will go for the next probe like this. And again we will go for the next probe. So, this is the number of probes or we can say expected number of probes in the probability calculation. So, now, we want to simplify this.

(Refer Slide Time: 10:03)

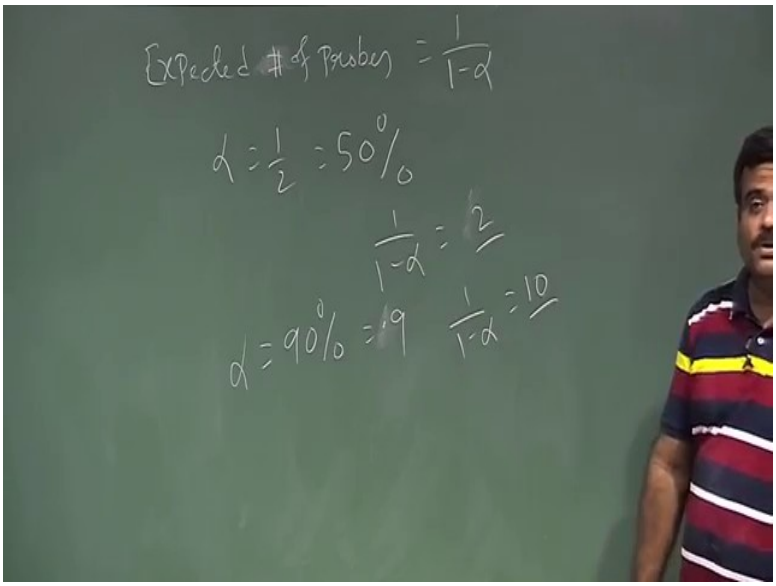


So the expected number of probes for an unsuccessful search is basically this. Now we can simplify this just by using this fact n - i by m - i is less than equal to n by m for all i this can be easily probe.

So, if we use that. So, this is basically less than equal to 1 + 1 + n/m 1 + n/m into like this. So, this n/m is basically alpha this is 1 + alpha into 1 + alpha into 1 + alpha into 1 + ... So, now, we have the expression like this is less than basically 1, so this is basically 1 + alpha + alpha square + alpha cube I mean we can say less than now this basically 1/1 - alpha since alpha is less than 1. This is the power series, this is the some infinite some, this is 1 by, so this is the probe.

So, expected number of proof for a unsuccessful search in a open address hash table is basically 1/1 - alpha. So, what is the meaning of this? So, this is the 1/1 - alpha.

(Refer Slide Time: 11:59)

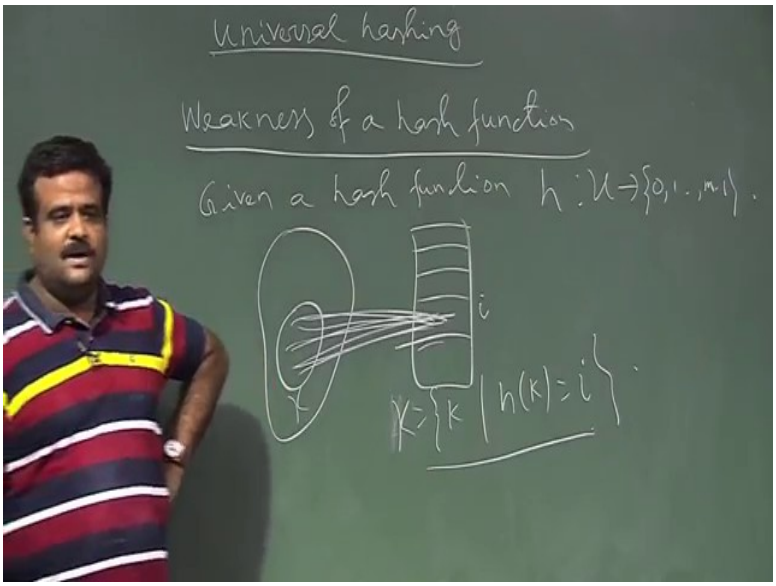


So, now suppose our table is half full. So, suppose our alpha is 50 percent so that means, our table is half full. So, if we have hundred slots suppose there are 50 keys half full. So, alpha is 1/2. So, then what is this value than 1/1 - alpha is basically 2 so that means, we need just two probes for an unsuccessful search, but what happens if the table is 90 percent full, I mean this is 0.9, so 9/10, if the table is 90 percent full then what is 1/1 - alpha this is basically 10. So, if the table 90 percent full then we need to have 10 attempt 10s probes for an unsuccessful search.

So, this is quite obvious because if we have 90 percent full means if there are 100 slot and if there are 90 keys and which are distributed about this slots then it is almost loaded. Then for a search key which is not there in the table we need to have the 10 probes for this. So this is the analysis of this open addressing and it is good in the terms of handling the collision because collision will be there in the hash function we cannot say you can construct a hash function there will be no collision because hash function this doming size is big then the co doming size. So, there has to be collision. So, collision this is the one way we can handle the collision by using the open addressing.

So, now we will start what is called universal hashing or universal hash function.

(Refer Slide Time: 14:07)



So, to start this we will talk about a weakness of a hash function. So, suppose there is a computation you have team a team b suppose IBM came to your place and announce a competition to build a hash function. So, they are going to use this for their or for their often hash function is having use in compiler operating system. So, suppose they announce a competition to have a hash function, so you develop hash function and b team develop another hash function and you both submitted the hash function to IBM.

So, now, what IBM will do IBM will give your hash function to b and your hash function to team a and ask for a, ask to find set of keys where it is colliding. So, it is always possible to have this because we know there will be the collisions.

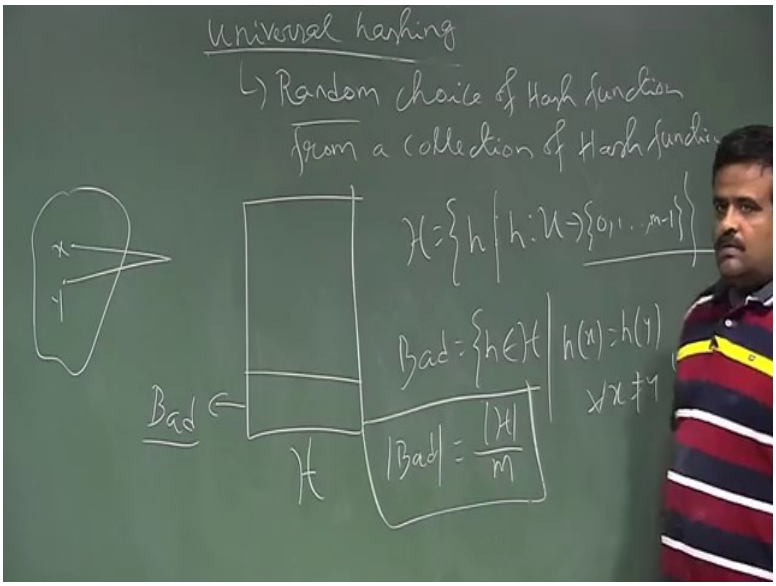
So, given a hash function, given a hash function h which is basically function form, so one can always construct a portion of the key where it is colliding a because if you have a time you can try for all possible keys and then you can just have this key which is basically for all values it is colliding into same slot. So, you obviously, will try for all possible key for your opponent code and you will come out with the such a key and you will submit this to the IBM that this is the set of keys were my friends code is not working that and your friend is not sitting ideal. So, they are also trying to find out a set of keys where your code is performing bad. So, that is can be possible because if you know the hash function in hand then you can try for all possible keys and then you can come with some portional where you your code is performing badly.

So, this is a fundamental weakness hash function. So, to avoid this weakness, so one can always construct this, this h such that h of k is always going to be a fix slot, so each number of collision. So, now to avoid that how we can avoid this scenario? How our friend cannot our friend or adversary cannot come with some I mean cannot come with some input where my code is performing bad. So, this type of thing we did in a quick sort if you remember in the quick sort if you know the position of the p board element the say in our original version of the partition we are choosing the first, had a first element as a p board element.

And then if we put the minimum or maximum in the first one then we are always getting the worst case. So, if we know the position is the third element third means not third smallest third position index third then also one can put the minimum maximum in the third element always and can get a input can construct a input where it will give a 0 is to n - 1. So, if we know the position of the p board element then we can have a example where our code is performing bad. So, to avoid that what we did in the quick sort we did a random choice of the p board element. So, we choose the p board element randomly position of the p board element randomly from the given adding. So, here also we will do the same thing.

So, to avoid this we have to choose a hash function randomly from a collection of the hash function. So, this is the way we can avoid this weakness because if we choose the randomly hash function then our friend is not knowing which hash function we are going to choose at the wrong time because this choose is at the wrong time. So, nobody can come with some set for which it will perform badly because we do not know which hash function we are it is going to choose randomly. So, we have a collection of a hash functions we are going to choose hash function randomly. So, that is the idea behind this universal hashing.

(Refer Slide Time: 19:30)



So, let us talk about this little more. So, this is to overcome this weakness. So, random choice of hash function from a collection of hash function, so that collection has some property what is that suppose we have a collection of hash function, this is a H. So, H is basically collection of hash function it is the function such that u cube. So, we have, so we considered set of all hash function I mean the collection of hash function such that among this collection suppose there is a portion which is referred as a bad portion that in the sense now if we chose a hash function from this portion then there will be collision; that means, if h is coming from this. So, bad is basically, so if we choose two keys.

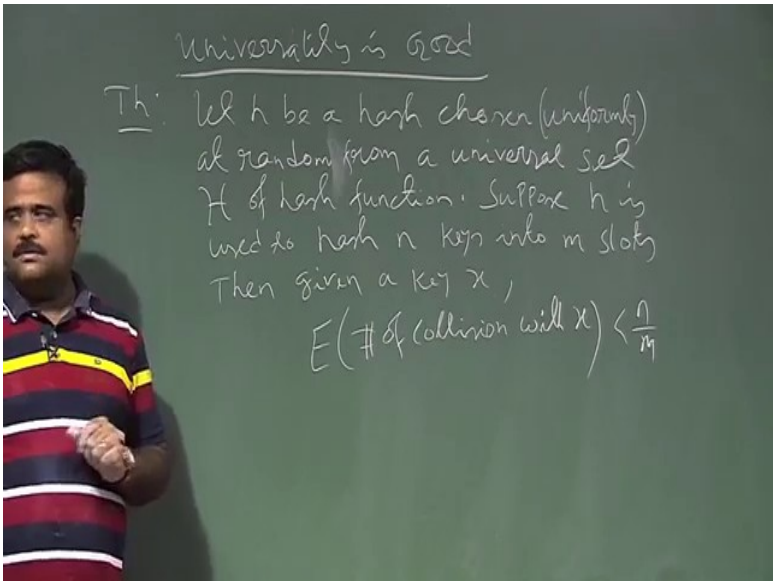
So, this is the keys set if we choose two key x y, x and y then if we choose a hash function which is x not equal to y if you choose a hash function from the form this bad potion then there has to be collision so that means, bad means it is basically subset of H such that h of x equal to h of y for all x not equal to y. So, this is called bad portion; that means, if our hash function is coming from this portion then there has to be collision for any two key any two distinct key, if we choose any two distinct key then definitely there will be collision. So, this is called bad portion I mean I refer this as a bad portion because it is giving as a collision for sure.

Now, if the cardinality of bad portion is basically cardinality of H/m. If the cardinality of this set that means number of element in this set is basically 1 by m fraction of the total then this collection is called universal collection. If the cardinality of this bad portion is 1/m fraction of

the total n is the number of slots, 1/m fraction of the total then this collection is called universal collection of hash function and if we choose a hash function from this collection and that hash function is called universal hash function. So, now, we will talk about why this universal hash function is good. So, the bad potion is the 1/m fraction of the total then this collection is called universal hash collection and if we choose a hash function randomly from this collection then it is called universal hash function or universal hashing.

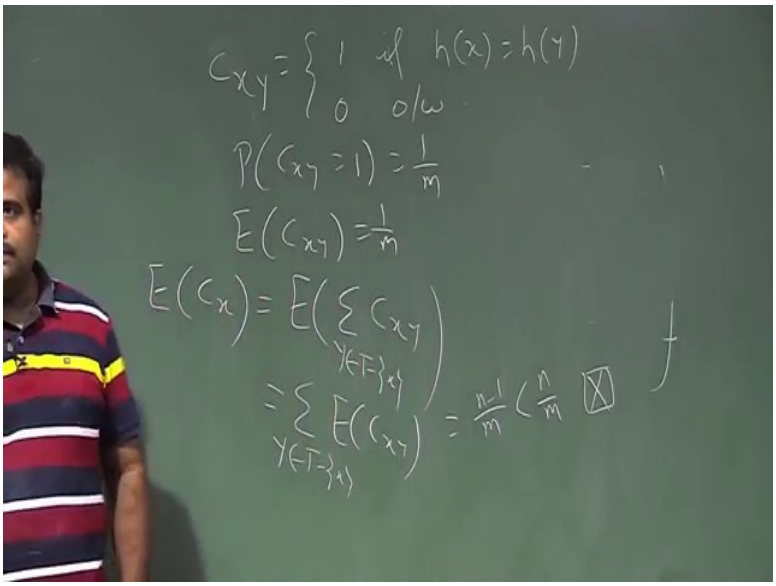
So, next we will talk about why this universality is good why you should take this hash function randomly from this collection.

(Refer Slide Time: 23:20)



Next we will talk about why universality is good. So, this we will prove by a theorem again. So, this theorem is telling let h be a hash function chosen uniformly at random equally likely basically at random from a universal set H of all hash function and suppose H is use to hash in arbitrary key n keys into m slots. Then given a key x we have expected number of collision with x is less than n/m which is basically the load factor and this is good because if we even cannot expect we are expecting the number of collision is less than n by m which is the load factor and that is good.

(Refer Slide Time: 25:49)



So, how to prove this? So, to prove this we need to use some indicator random variable. So, we choose the hash function from this universal hash function set uniformly at random. So, we denote this random integral random variable C xy is basically 1 if h of x equal to h of y and 0 otherwise.

Now what is the probability of C xy? Is 1, is basically 1 by n because we are fixing x and it will colliding to this, so we are choosing the hash function from this universal set. So, this is our H and this is the bad portion and the cardinality of the bad portion is cardinality of H by n. Now if we have a x now if we choose a y which is not equal to x now what is the probability that collision. So, collision will happen if we choosing the hash function from this. So, size of this is H by m and size of total is H. So, the probability that there will be the collision is 1/m basically because H/m divided by H. So, it is basically 1/m. So, this is the probe, so expected value of c of x is basically 1/m.

Now, we are looking for expected value of C of x and C of x is basically summation of what, C of x y where y is t - x. So, this is what we are looking for and this we want to show is less than 1/m. So, this is basically we can take the expectation inside. So, this is basically expectation of C of x y and y belongs to; now this is basically 1/m, this basically 1/m. So, this is (n-1)/m this is less than n by m. So, this is the probe. So, expected number of collision is 1 by m. So, that is good. So, expected number of collision is basically n/m. So, this is the load factor. So, the universal choice is good. So, what is the universal hash function? We have a

collection of the hash function along this collection if the bad potion is just 1/m fraction of the total and then if from this collection total collection if we choose a hash function randomly, uniformly equally likely and if we use the hash function for our hashing and then this choice of a hash function is called universal hash function.

So, in the next class we will construct some universal hash function.

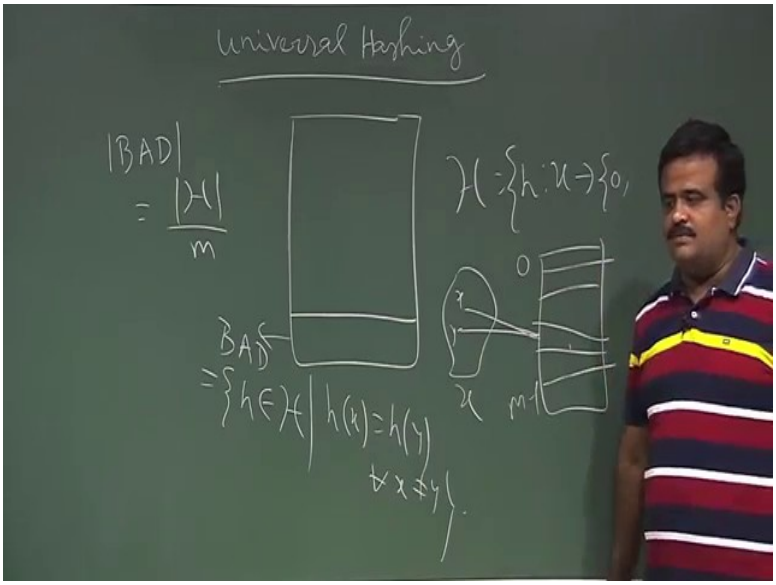
Thank you.

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

Lecture - 24
Perfect Hashing

So, we talk about perfect hashing. In the last class we have discussed the universal hashing. So, today we will start with a construction of a universal hashing an example of a universal hashing. So, let us just recap what is universal hashing.

(Refer Slide Time: 00:36)

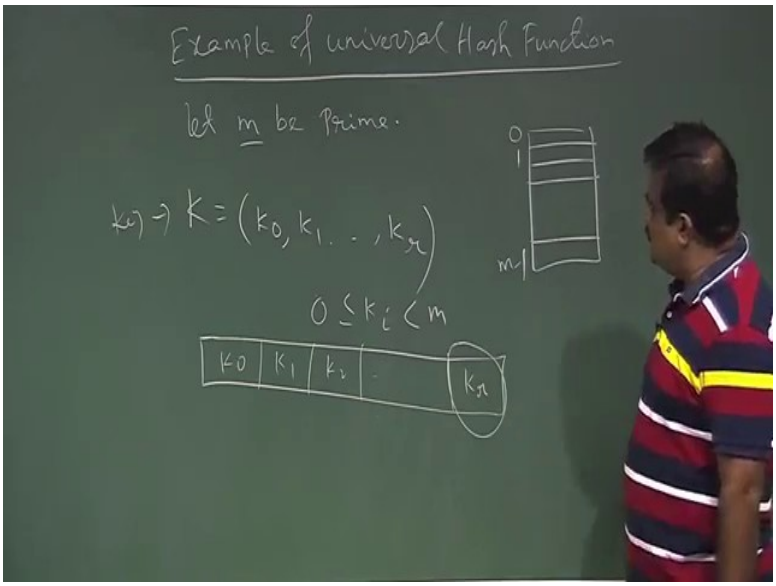


So, basically it is a collection of hash function H is basically hash function from u to 0 1 up to m - 1. So, our table size is 0 to m - 1 this is our hash table size and we consider a collection of hash functions such that among these collection there is a bad portion and this bad portion is basically set of all hash function such that given any 2 key x y they will collide for all x not equal to y if we choose any 2 key, if we choose any 2 key x y which they are not same then if we apply this hash function then they have to collide.

So, if our hash function is coming from this portion then there is a guarantee that there will be a collision. So, if you choose any 2 keys then this will collide. So, that is why it is called bad, bad portion and if the size of these set is basically 1/m fraction of total set m is the table size basically, then this collection is called universal collection of hash functions and if you choose a hash function randomly from this collection, suppose we have such a collection and

if you choose a hash function randomly from this collection then that is called universal hashing. So, now we talk about an example of how to construct such a universal hashing or such a collection. So, that is an example or we have to construct such hash function or such collection so construction of or example of universal hashing function.

(Refer Slide Time: 03:03)

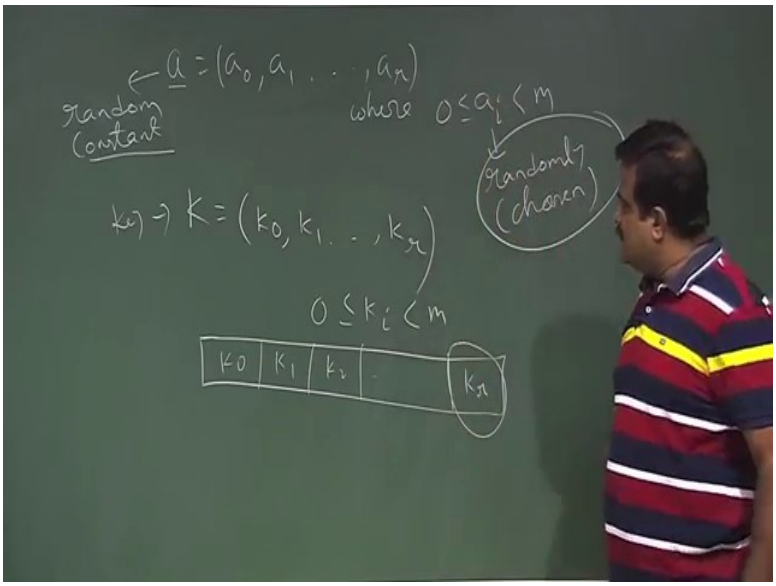


So, we are given key suppose keys are basically, so this is a random choice. So, suppose let m be a prime which is the table size, so our table size is m so we have a 0 1 up to a m - 1 this a table size and this is a prime number, that means, a number which is divisible by only itself or 1. So, there is no factor, factor of that number this is an integer. So, now, we have a key K we decompose this key into r bits, r digits.

So, k 0 k 1 I mean r + 1 digits. So, k r where each of this k's are coming from this so, the value is maximum value is m. So, if we are given a key, usually very long so what we are doing, we are dividing this key into digit like k 0 this is k 0 k 1 k 2 like this. So, last one is k r and each of this is basically bounded by each of this value is less than m. So, this is the decomposition on we are doing on the key.

So, now what is the random strategy here? So, now, this is the key now we are choosing a constant a vector.

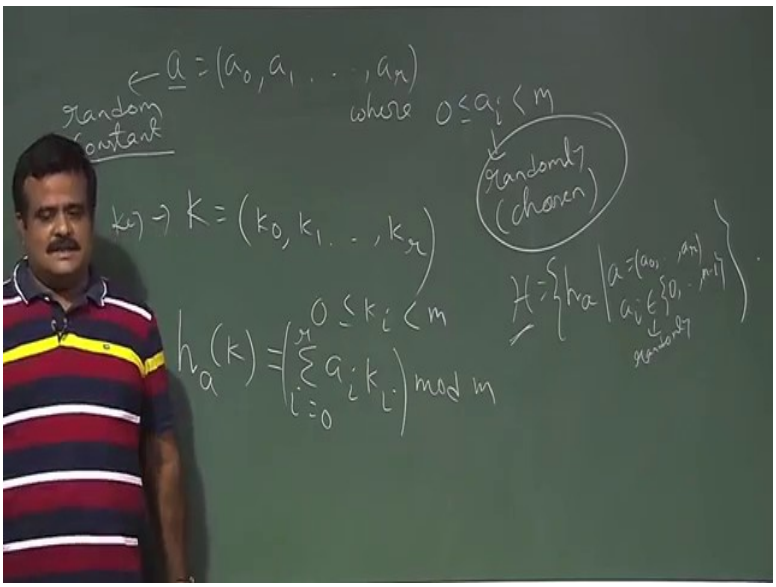
(Refer Slide Time: 05:13)



So, we are choosing a constant a which is also a 0, a 1, a 2 a r where a i is a i is also form 0 to m and a i are chosen randomly. So, this is the random choice of a i. So, this this vector this is basically a vector which is a basically random numbers and each digit of these. So, these are all random digits and these are coming from the value is coming from 0 to m.

So, this is a constant which is chosen randomly it is a random constant. So, we are choosing a a i randomly from this then we construct this we have this a, then how to define this hash function? So, hash function will depend on this constant.

(Refer Slide Time: 06:30)



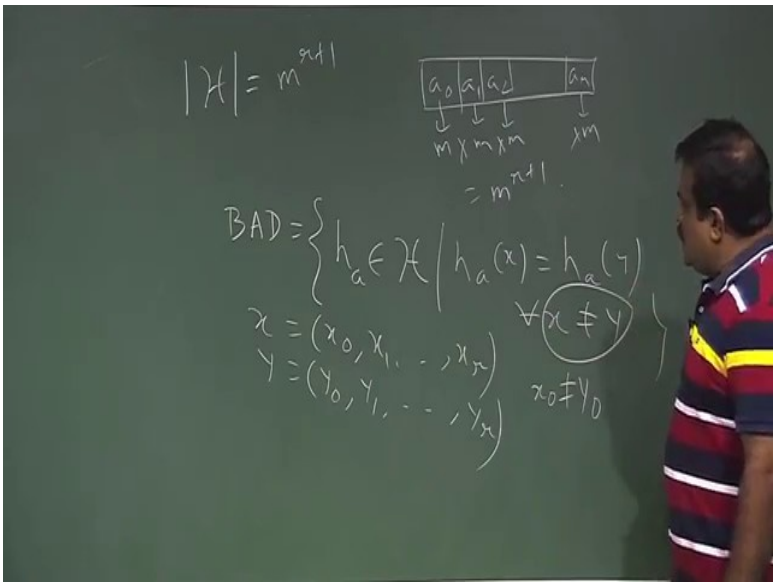
So, it is denoted by h(a), a is this random vector h(a) on k. So, k is the vector it is basically the inner product of this and this. So, it is basically the summation of a i k i, i is equal to 0 to r then mod m. So, this is basically our hash function. So, it is basically a 0 k 0 multiplication.

These are all integer so, a 0 k 0 + a 1 k 1 . . . + a r k r and then this inner product may be more than. Then we have to take the mod m to fit in to the table. So, this is the hash function. So, this is called this hash function is depending on this choice of a. So, if we choose a randomly then we have for different values of a, we have different hash function. So, this a, we are going to choose at the run time. So, now, so if we considered this collection like H h is basically this collection a where a is basically a 0, a 1, a a r and a i are basically coming from this set.

This, this is 0 1 . . up to m -1and this choice is randomly so this collection, this collection is a hash function collection. So, where we are choosing this a i is randomly from this set 0 to m - 1 and then once we choose a i randomly we have this a vector random vector a then use that random vector a we define the h of a by using this formula in our product formula. So, this is our hash function and this is our collection. So, we want to know whether this collection is a universal collection of hashing. So whether this collection is a universal collection. So, to prove that we have to see the bad portion I mean what is size of the bad portions. So, let us just have the cardinality of H first and this, the random strategy because we are choosing the hash function randomly at the run time because we do not know which a i we are going to choose this a i we are going to choose at the run time.

So, once we got the a i then we have the hash function using the formula ok.

(Refer Slide Time: 09:23)



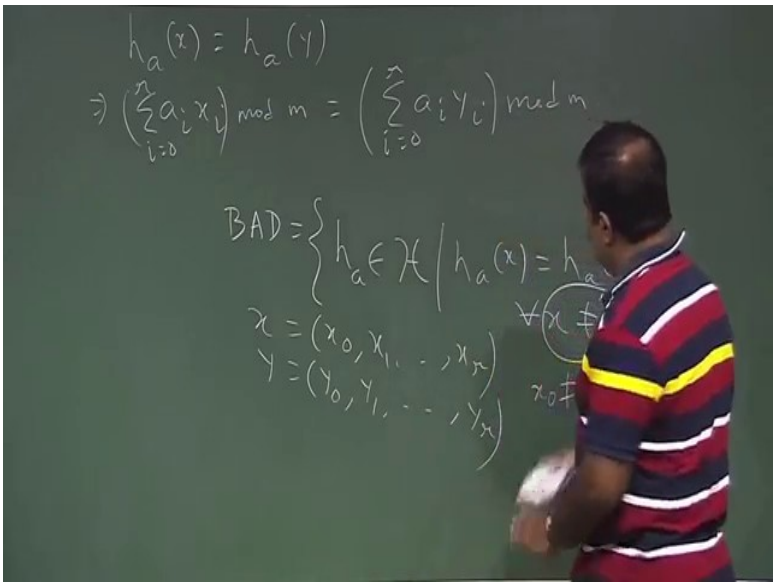
So, what is the cardinality of this set? So, what is the size of this? So, this is basically wearing this a a digit. So, then how many there are r + 1 a a i and each of then take value m. So, size of this is m to the power r + 1 because this is basically the choice of a is so a 0 a one a 2 like a r. So, each of a i can be chosen m ways this is m ways, this is m ways, this is m ways. So, the total possibility is m^(r + 1).

So, this is the cardinality of this collection H now we want to see whether this collection is a universal collection or not. So, for that we need to consider the cardinality of the bad portion. So, let us just talk about cardinality of bad portion so for the bad portion what we have, we know the bad portion is such that this is the set of all function h from this such that h of x equal to h(a y) where x y not equal. So, if given any 2 key and this is to for all x y, given any 2 distinct key if they are colliding and if this is true for all such keys, such pair of keys then that collection is called bad portion.

Now we want to have cardinality of this bad portion, to have the cardinality of the bad portion let us take x is a key so, x is also in this form x r and y is a another key y 0, y 1, y r and since x is not equal to y. So, any 2 digit of this at least any 2 digit of this will be not equal to y. So, for the simplicity we are taking x 0 is not equal to y 0, without loss of generality for the simplicity we can assume because we want x is not equal to y.

So, now we take this is equal so we actually want to find out the cardinality of this. So, h of a x equal to h of a y.

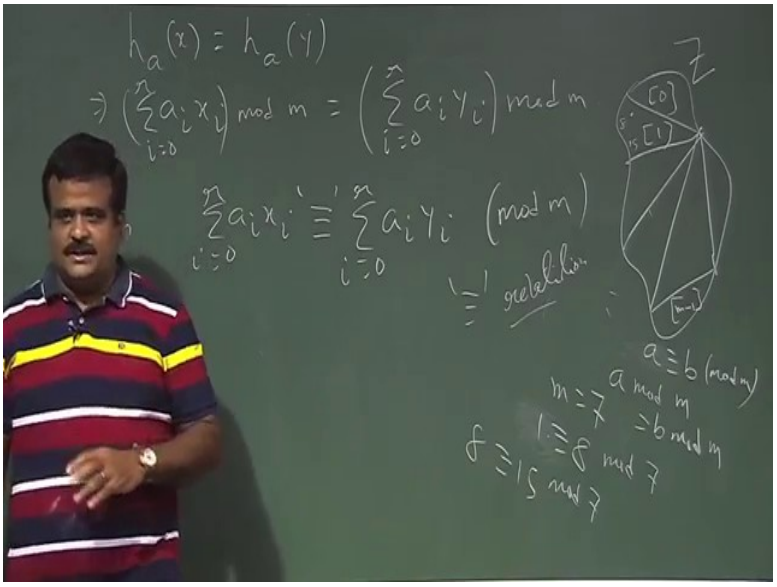
(Refer Slide Time: 11:55)



So, this means, this implies summation of a I, x i is equal to summation of a I, y I so, mod m and this is also mod m so, this is also mod m. So i is from 0 to r, mod m is equal to summation of a i, y I, 0 to r mod m. So, now, this is the scenario now this is basically so this 2 expression is equal under mod m so; that means, once we have 2 expressions equal under mod m.

Then we can say these 2 are under the congruence relation.

(Refer Slide Time: 13:10)

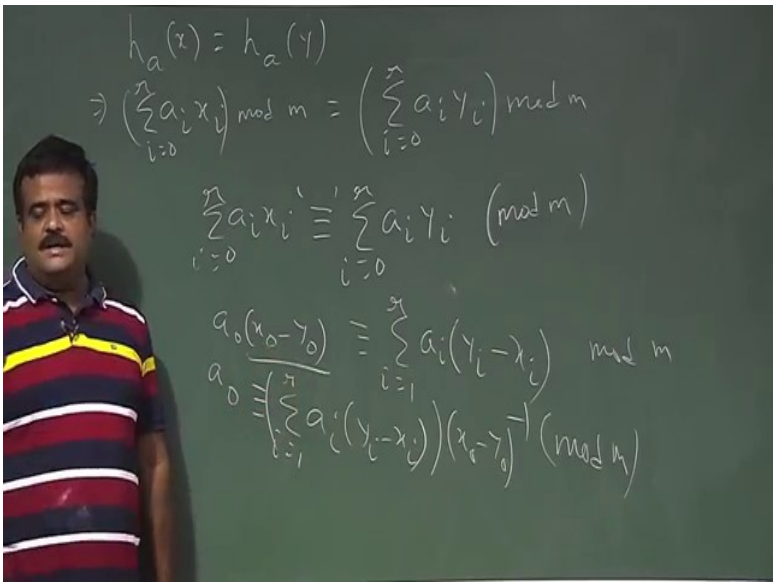


So, we can write this as summation of a i xi, i is equal to 0 to r is congruence to summation of a i y i i is equal to 0 to r mod m. So, this congruence is a relation . So, how this is coming it is coming basically suppose this is a equivalent relation suppose we have a set of integer, this is a integer set now we take a m, now we just take a any integer and we divide we try to divide this by m. So, then what are the reminder? Reminder will be 0 to m - 1 if you take a any integer if you divided by m then we are the reminder 0 to m - 1.

So, that will be basically the equivalence by the cross section. So, this is 0 class, this is 1, class or dot dot this is m - 1 class. Now, we say 2 integer say x y, we said 2 integer a b will be in the same class or they are related. So, a b a is congruence to b mod m, when you say this if a mod m is same as b mod m so; that means, if we divide a by m the remainder will same as if we divide b by m, suppose m is 7 set then 1, then 8, 1 and 8 they are congruence under 7 1 is congruence to 8 mod 7; mod 7 or 8 is congruence to 15 mod 7, because if we divide 8 by 7 the remainder is 1, if you divide 15 by 7 then remainder will be one so; that means, 8 and 15 will be in the same class this class.

So, here if any 7 so 8 and 15 will be seating here. So, this is the equivalent classes and this relation is equivalence relations and it will form a partition on this set and this is the equivalent classes. So, basically since they are under mod they are equal; that means, there in same class so; that means, they are are related by this relation this is called congruence relation congruence modulo m. So, anyway this is basically giving us this relation now we want to take this this side.

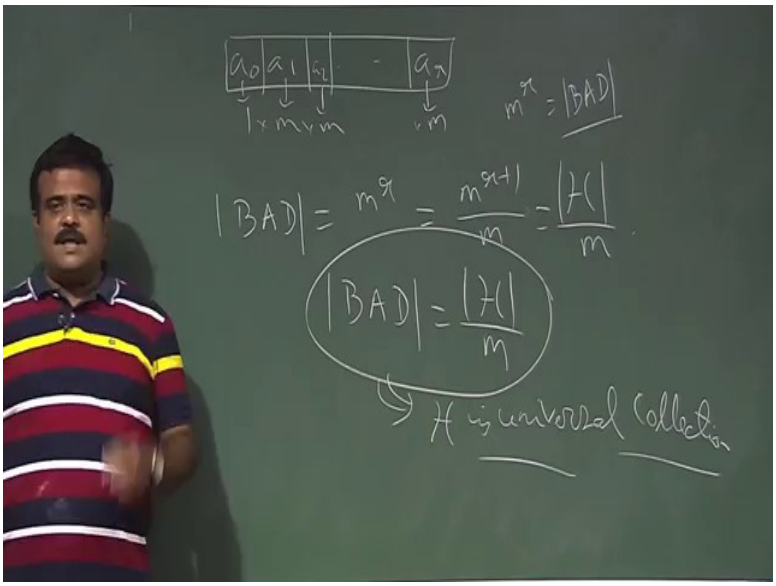
(Refer Slide Time: 16:04)



So, this basically so, we want to take a 0 x 0 - a 0 y 0 1 side and then remaining are other side. So, summation of a so a i will take common a i, y i - so x i will come this side, so this this from 1 to r because we have taken a 0 x 0 here so we can take common of a 0 x 0 - y, so this is under mod m so; that means, what now we are assuming x and y are not are same. So, we are assume x 0 is not equal to y 0, if x 0 is not equal to y 0; that means, this is non 0 x 0 - y 0 is not equal to 0 if x 0 - y 0 is not equal to 0.

Then they have a inverse under mod m. So, summation of a I, y i - x i and this i is from 1 to r multiply with x 0 - y 0 inverse under mod m so; that means, we are not having choice of a 0 a 0 will be determine by this and this is possible because x 0 - y 0 is not a is a non 0 quantity. So, it as a inverse we can multiply the inverse both sides so; a 0 is basically coming from this expressions. So, then, that means what? That means, we have a choice for this a i's other than a 0, if you just write this, basically the a i's. So, this is a 0 a 1 a r.

(Refer Slide Time: 17:56)

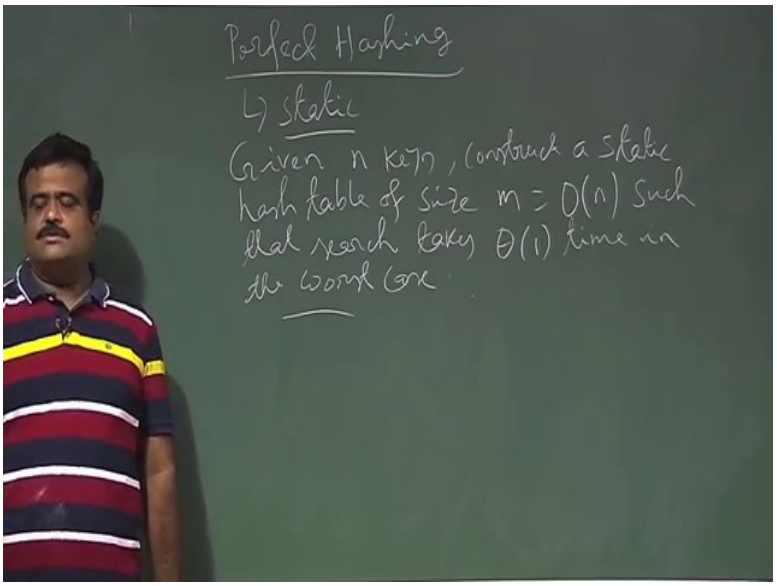


So, this can be chosen m ways this is a 2 this can be chosen m ways and this can be m ways all this, but we do not have choice for a 0 because is coming from this expression. So, a 0 can be chosen only one way because this is coming from this expression so; that means, what is the cardinality of bad portion is basically multiple of this, so this is basically m^r. So, this is the cardinality of the bad set. So, bad set means where there coming to be equal. So, this is basically m^r and m^r is nothing, but what? So this is the cardinality of bad set; m^r which can be written as m^(r+1)/m and m^(r+1) is the cardinality of h/m.

So, the cardinality of the bad set is basically 1/m fraction of the total. So, this implies this collection is h is universal collection universal hash function. So, this is a now h is universal collection now if you choose a hash function from this collection then that hash function will give us the that hash function is called universal hashing. So, this is one example of universal hashing. So, it is also it is basically the random choice. So, if we choose hash function random depending on the value of this vector a so; that means, nobody come with some key the input set where it is colliding because we do not know which way we are going to choose at the wrong time.

So, this is a random choice. So, this is one example of universal hashing.

(Refer Slide Time: 20:23)

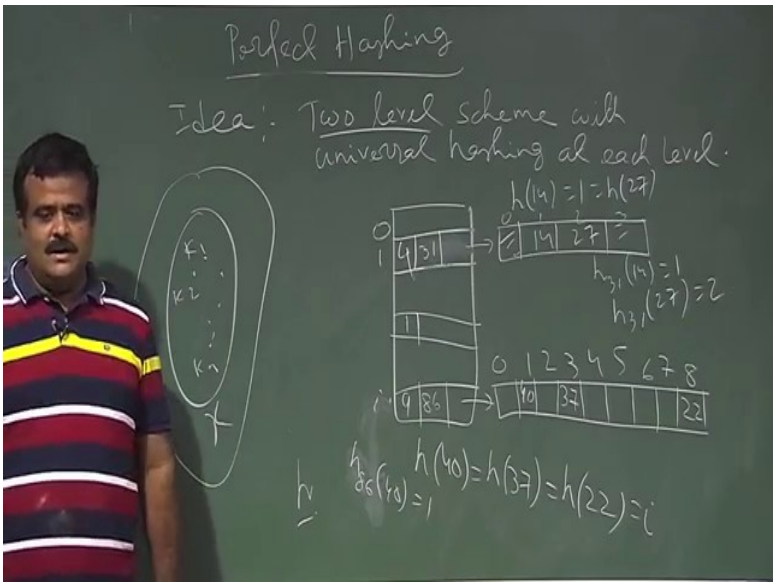


So, next we will talk about perfect hashing, perfect hashing so, it is basically a static arrangement. So, it is a static arrangement, so what we are doing here we are basically we have given, problem is we have given n keys and we need to construct a static hash table.

That means, we are not allowing anybody to join or anybody to delete from this table. So, the dynamicity we are not allowing here this set is static set, so given this we need to construct a static hash table of size m which is also order of n such that search will be taking in constant time, such that search takes theta 1 time in the worst case in the worst case. So, this is our problem, so, our problem is we are given n keys and we have given a, we need to construct a table and this set is static.

That means, we are not allowing anybody to joint in a set or anybody to delete from this set and this set is static. So, given way we need to construct a hash table and the table size is of order m such that the search will be faster ok.

(Refer Slide Time: 22:28)



So, this will be done, this will be done using the 2 level hashing 2 level hash table so, the idea is to, idea is to 2 levels scheme 2 level hash function with universal hashing at each level, with universal hashing at each level. So, we have basically 2 level hash function.

2 level hashing so, what is the idea? So, we have given we have a hash, function hash table 0 to m where m is of size m and we have given our key set where are basically k 1, k 2 up to k n we have n keys. So, what we have doing we are basically apply the first level hash function so, we have a hash function h which is first level hash function. So, we will apply this hash and this is also universal hash function this is also coming from universal hash hashing. So, every level so, we just apply the hash function and it will store it will mapped into this slots and there will be come collision, if there are collision suppose here some n i suppose so, we are n n keys we are mapping into this table.

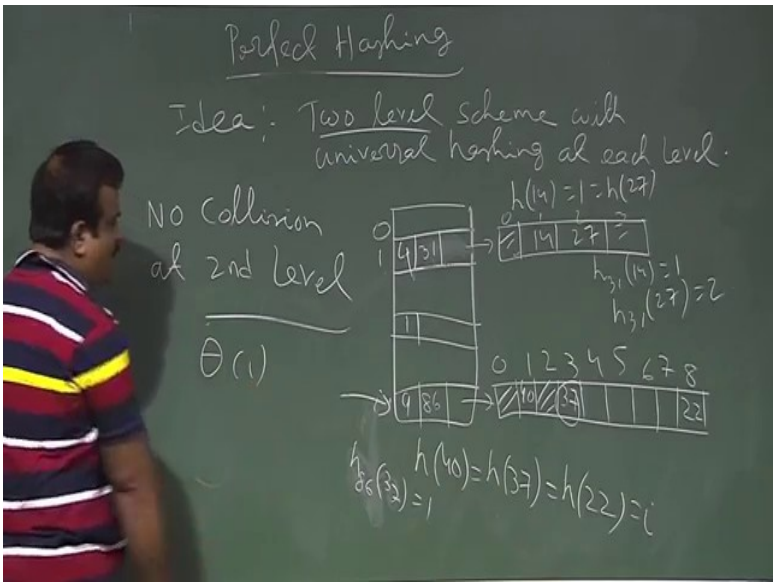
So, there will be some collision suppose in this slot there are say 2 keys which are colliding, so, these 2 keys are say 14 and 27, 2 keys are colliding in this slot so, what we do we cannot fit it here. So, we will have a second level hash table, but in the second level hash table what we are doing. So, since 2 keys are colliding in this slot we are having 4 slot for the second level. So, second level table size is 4 for 2 keys so, if it is 3 keys it will be 9 if it is n i key it will be a ni square in the second level. So, second level so, this is 14 and 27. So, this is the second level so, we have to apply the second level hash function. So, for that again this is coming from universal hashing. So, we have to choose a, a to have this hash function.

So, this is means that vector h(a) is our hash function. So, this is the first level hashing and this is the second level hashing so; that means, what? So, this is say 1 so, h of h is the first level hashing h of 14 equal to 1 is equal to h of 27. So, under first level hash function they are mapping to slot 1 and since they are colliding and there are only 2 keys are colliding. So, we need to have a second level hash table and i say in the second level will have the table size square of that so, 2 key; that means, 4 table size so; that means, and for that we need to have a hash function that hash function again we are choosing from universal hashing for that we need to have a so; that means, h of 31, 14 equal to so, this is 0 1 2 3 is equal to 1 h of 31 27 equal to 2 so this is the way.

So, similarly if there are only 1 is here so then we do not need to have any second level, but there are say 3 keys are colliding here so second level. So, this will indicate second level how many what is the table size and , this is the second level hash table and the second level since 3 keys are colliding in the second level we have we need to have 9 is the table size. So, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 so 9 means up to 0 to 8 so, this is the second level and suppose these are the keys are colliding. So, say 40 say 37 and say 22. So, we know in the first level these are colliding so; that means, under h(40) h(30) there colliding into the same slot and this is a ith slot. So, there are 3 keys colliding in the i th slot in the first level hashing.

So, for the second level we need to have 3 keys of 3 square so 3 square means 9, 9 slot we need to have and for the second level we need to use a different hash function so that hash function we have getting from h of a. So, a is 86 So, basically h of 86 of 40 is 1. So, h of 86 of this is this, so this is the way so this is the 2 level hashing. So, now, we have to, so since we are taking a in the second level we do not want any collision. So, the second level hashing should be collision free and to guaranty that we have taking the size of the table is square of that.

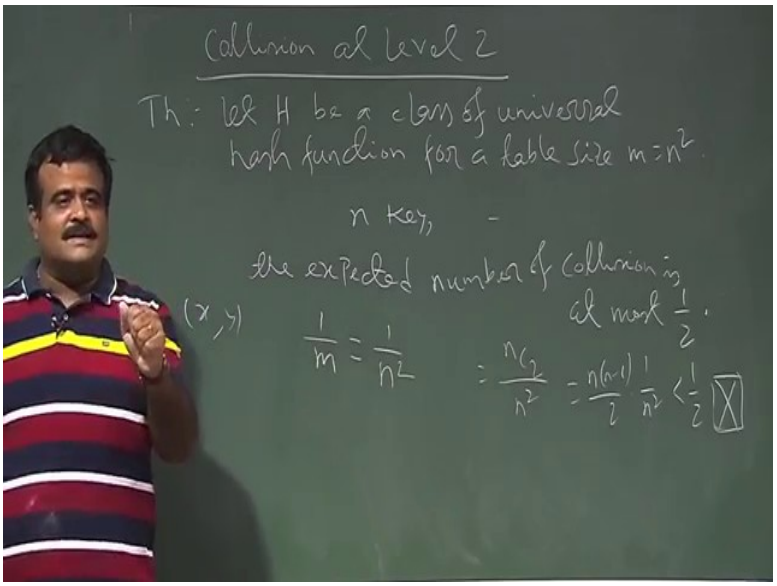
(Refer Slide Time: 28:04)



So, no collision at second level so, this we have to ensure and for that reason we are taking the table size for the second level is double. So, we have to analysis that, so before that how to search how the search is in constant time suppose we want to search 37, so what we do? We apply the first level hashing so it will map to here, so we know the there is the second level hashing and we need to apply the second level hash function. So, where from we can get the second level hash function that information has to store here. So, if a is given we know the hash function, so this a is given in this table, so we know h of a, so we know the hash function. So, we apply h of that on 37 we will reach here and we got 37. So, searching is just 2 hash function. So, searching is theta of 1 time because just a 2 hash so theta 1 time is the search time.

So, now, searching is now let us have a quick analysis of this why this is, 2 types of analysis we need to do why there is no collision in the second level and we have to bother about storage also because we are using intuitively it is where we are maybe using lot of storage, but that also we need to analyse.

(Refer Slide Time: 29:33)



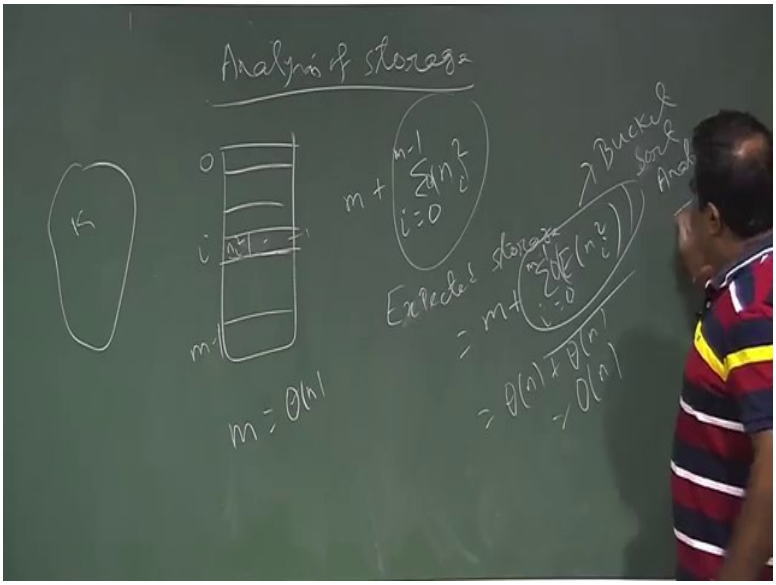
So, let us analyse the collision at level 2, so why it will ensure that there will be no collision is level 2. So, this is coming from this theorem let h with the be a class of because every level we are using the universal hashing universal hash function and in the second level the hash function table size, universal hash function for a table size.

So, table size we are using that square so, if there are m keys and we are using if there are n i keys we are using n i square. So, suppose there are n keys in the second level and so the, our table is n square for the n keys. So, n is the keys and the table size is n r square. So, now, the theorem is telling the expected the expected number of collision number of collision is at most half; that means, we cannot expect even 1 collision also. So, how to prove this? So, prove this, so if we take 2 x y 2 keys so what is the probability that they will collide is basically 1/m m is the table size it is basically 1/n square because we are choosing the universal hashing now how many pairs are there.

So, expected number of collision is basically n c 2 / n square. So, it is basically m into n - by,i 2 into one by n square this is basically less than half, so this is the proof. So, even we cannot expect more than 1 collision this number of collision is at most half so; that means, because we have say we have 3 keys and we have 9 slots. So, it is quite obvious now the expected collision will be no collision because there are 9 slot and 3 keys has to be fitted and our hash function is good hash function it will distribute the key uniformly over the slots. So, the chances of collision is less, now let us do a quick storage analysis.

So, this is the second level that is why second level there will be no collision because number of size of the slot we are taking n square.

(Refer Slide Time: 32:11)



So, analysis of storage, so what is table size? So first level we had this is the first level we had m is order of m we have m tables now suppose we have n keys and suppose at the i th slot there are n I, n i keys are colliding in the slot in the first level. So, in the second level table size is n r square now. So, what is the size in the second level table second level table size is n i square summation of n i square i is equal to m to n - 1 and. So, what is the total size?

So, m + this this is the total storage so this is the storage for the second level where n i is the number of keys colliding in the ith slot after the first level now if you take the expectation. So, expected storage is basically m + summation of expected of n i square or it is order of order of n i square. So, this is order of n i square 0 to m - 1. So, these expression we have seen for the bucket sort if you remember we have some keys and we have throwing into the bucket and this we have proved there it is order of n. So, this order n this is also order of a n, so this is the this is coming from bucket sort analysis. This storage is also order of n. So, we are not using really extra storage although its look like that n i is the number of keys colliding. So, in an square it look like we are using new storage, but this is the analysis that we are not really using new storage . Total storage is order of m.

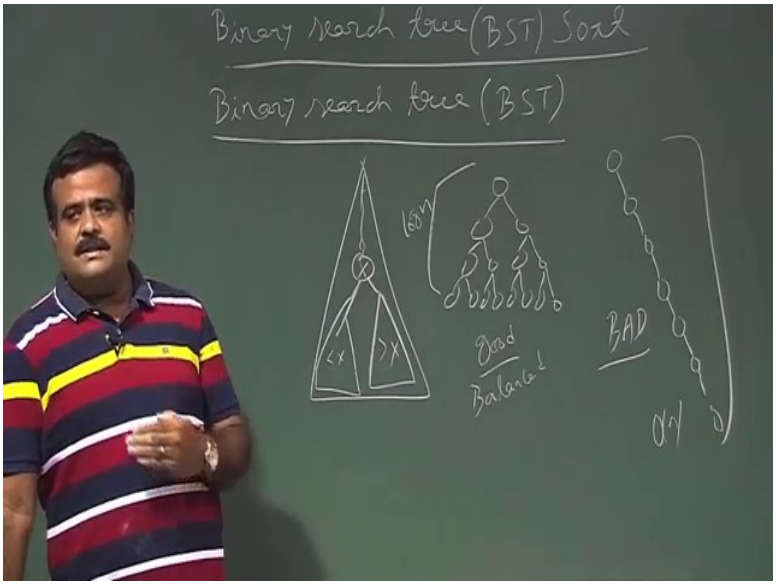
Thank you.

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

Lecture - 25
Binary Search Tree (BST) Sort

So we talk about binary search tree sorting I mean how binary search tree can help us to have a sorting algorithm. So, before that let us talk about what is the binary search tree, recap the binary search tree, binary search tree or as it is called BST.

(Refer Slide Time: 00:30)



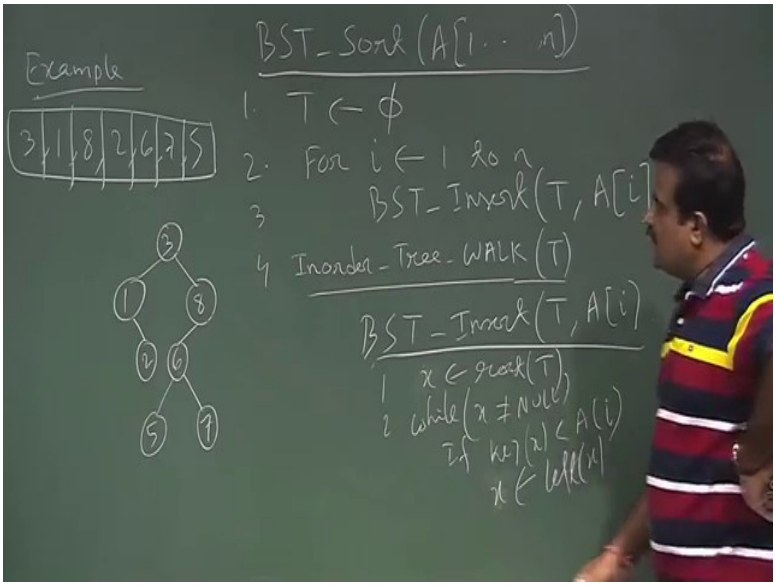
So, it is basically a tree and it has some property. What is that? It is suppose this is a tree suppose you take any node over here in this tree, key value is X. Now we considered the left subtree rooted at this, right subtree rooted at this. Now binary search tree means all the key value over here must be less than X all the value must be greater than X. So, this is the property. So, if they are distinct then it is must be strictly this. So, this is the property of binary search.

So, a binary tree which is having this property is called binary search tree. So, if the all elements in the left side of the tree is less than X and all the element in the right side of the tree must be greater than X. So, there are some good tree and bad tree. So, what are the good tree? So, suppose we have a tree like this, this is a good binary tree why because it is balanced this one is good tree, good in the sense it is balanced as height is log n. So, if there

are n nodes height is log n. And what is the bad tree? Bad binary tree like this if we have this. So, say height is n. So, this is a bad tree because it is not balance tree.

So, why balance tree is good because we can do some search or we can do some query in a height time and if height is log n then it is a log n time. So, that is why if it is good to have a balance tree. So, today now we will talk about how we can use the binary tree to have a sorting algorithm. So, let us just write this as BST sort. So, we have given n numbers we can sort them by using the binary search tree.

(Refer Slide Time: 03:07)



So, this is BST sort we have given n numbers or we have given a array of size n. So, what we do? First our t is empty we are initialising a t tree binary search tree empty and then for i is equal to one to n we insert this node into the tree.

So, this is BST insert basically we insert this A i into the tree and then we will do the inorder traversal of the tree. So, after this we form the tree, form the BST. So, we will talk about how the BST insert will work we form the BST and then so to from BST basically we start with the root until we reach to a nil or the leaf node, if tree is initially empty then the first node will be the root and in the subsequence step. So, if we start with the root if the element is less than that we go to the left side of the root left subtree otherwise we go to the right subtree and this way you continue until we reach to a nil or leaf node once we reach to a nil we insert that node there. So, this is the basically BST insert.

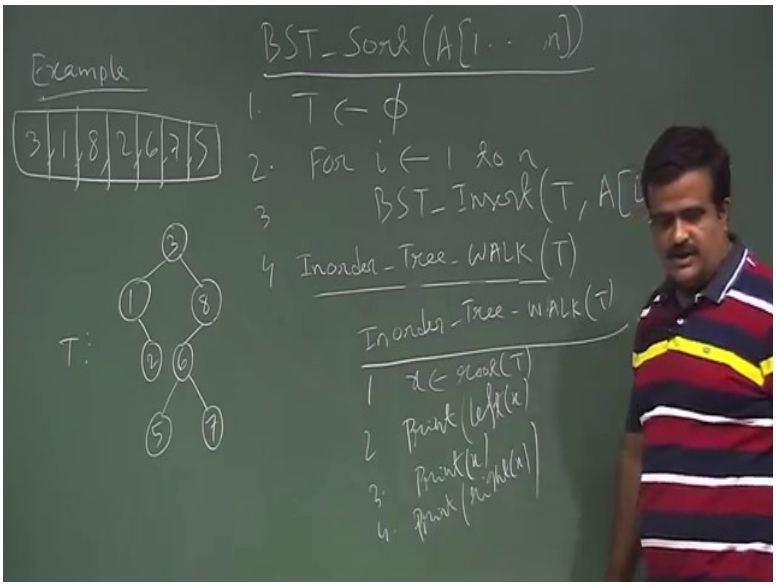
And then we do the inorder tree walk inorder tree walk of this tree inorder tree walk means we first print the tree traversal. So, we first print the leaf subtree then the root then the right subtree. So, that way we just visit the all the element of the tree. So, this is one. So, there are inorder, pre order, post order traversal.

So, this is the way. So, inorder, if you do the inorder traversal then we will get the sorted one. So, let us take an example - suppose we have some number say 3 1 8 2 6 7 5 suppose this is our a array this is our given numbers, there are how many numbers 1 2 3 4 5 6 7, there are 7 numbers is given. So, now, how to form the tree? So, initially tree is empty now we insert this 3, 3 will be the root and then we then we insert slowly, so 1. So, this we have to insert 1. So, we start with the root 1 is less than 3. So, we will go to the left part of the tree left part is empty. So, you will insert there 8, 8 is greater than 3, then 2 we start with the root 2 is less than 3.

So, we will go to the left part of the tree then again 2 is greater than one. So, we will put it here and then 6, 6 is greater than 3, but less than 8. So, we will put it here then 7, 7 is greater than 3, but less than 8 again greater than. So, we will put it here then 5 5 is greater than 8 say greater than 3, but less than 8, but again less than this. So, this is the structure of the tree. So, basically this is the BST insert.

So, if you just write the BST insert code; so, we first start with the root. So, we take this X as a root, root of the tree initially root is empty i.e. root is null. So, while we are not reaching to a null while X is not null. So, we do this if the key of key of X; that means, key means here the values if the key of x. So, we are going to insert some number A i, if key of X is less than A i then we call the this is the recursive call. Then we go to the, then X is this is while loop then X is left of X else X will be the right of X, if key is greater than this. Ao that means, we start with the root; if the root is empty that is the initial condition t is empty then we insert that; that is why 3 is inserted has a root and after that we start with we say 8.

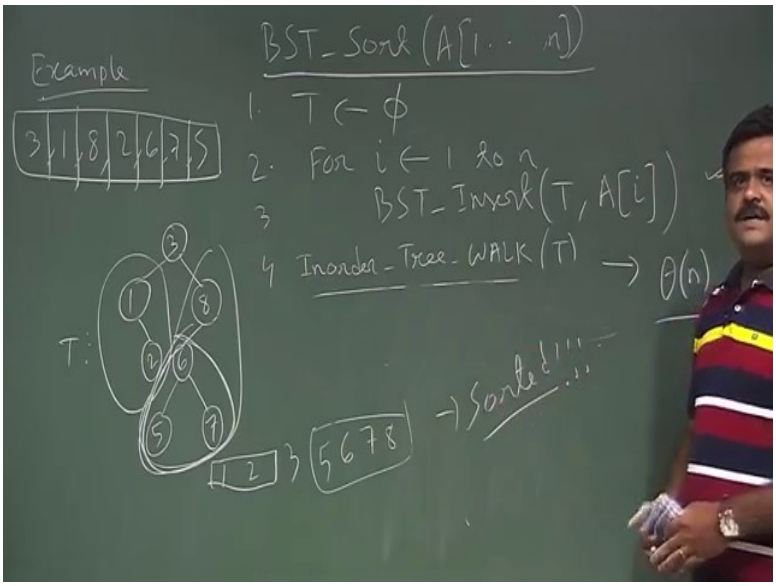
(Refer Slide Time: 08:44)



So, we know 8 is greater than this. So, we call this in this then again 8 is sitting here. So, like this. So, this is the BST insert now what is the inorder tree walk inorder tree walk basically. So, inorder tree walk. So, inorder tree walk is basically we first, so X is the root, root of t. So, we first print the we first traverse the left part of the left of a X then we print the root and then we print the right of X print means we again call the inorder tree walk. So, we do until we go until it search no child then we will print it print right of X print means this is basically inorder tree walk again.

So, for example, here we start with we want to call. So, this is our tree after this is insert BST tree insert. So, we start with the root we first call this inorder tree walk of this and then we print 3 and then we print the right part of the tree. So, again for the left part so this is the tree now, now we again call the inorder. So, this has no left part, this is one then we print the right part.

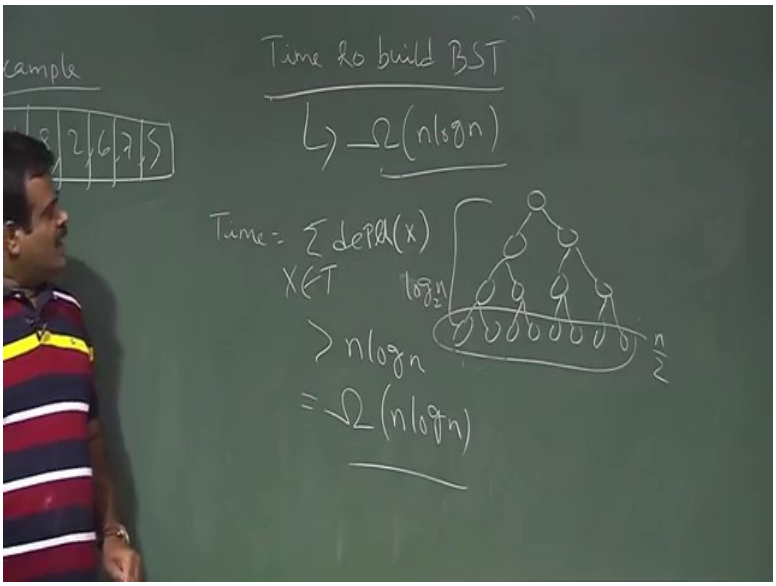
(Refer Slide Time: 10:06)



And similarly for this tree, so 8 is the root for this subtree now we call the inorder tree walk. So, there is no right subtree, so 8 will be printed here and this has to be print the left subtree. So, left subtree is this one. So, again for this we call this is the root this is the right sub right and this is the left. So, 5 6 7 sorted. So, inorder tree walk; if we do the inorder tree of on this BST we are getting the sorted array. So, this called BST sort. So, now, we have to analyse this how much time it is taking. So, how much time it is taking to do this inorder tree walk it is order of n because basically we are just seeing the node only once we are just print we are just travelling the tree. So, we are visiting the nodes only once. So, there are n nodes. So, it is basically order of n we are just printing the nodes. So, order of n.

Now the question is how much time it will take to build the tree build the BST this is the build the BST. So, how much time it will take, so time to build the BST, so that is the question. So, that time will depend on this the total time for the BST sort. So, how much time it will take to build a binary search tree for a given n input for a given n array. So, how much time it will take to build a BST? So, we will come back to this example again.

(Refer Slide Time: 12:23)



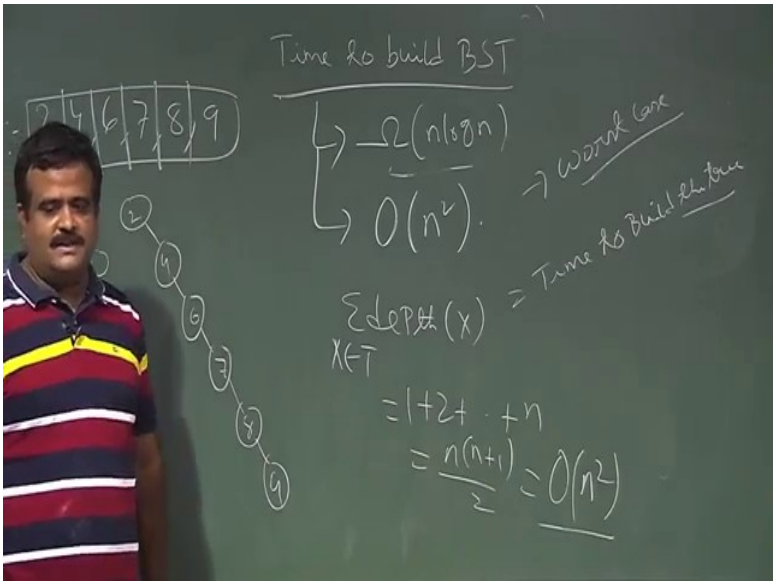
So, time to build BST binary search tree. So, any lower bound upper bound. So, any upper bound how much time it will take. So, how to build a BST? So, basically time to build a BST is basically summation of the depth of X, while X in the tree. So, every time we are comparing and we are reaching to a position and that position is the depth of that node. So, depth means that many times we compared to insert this node. So, this the time complexity to build the BST basically time of the depth of X where X is in the tree.

So, this will be basically big omega of n log n this is the lower bound we cannot do better than big omega of n log n why because the best case for this is when we have a balance tree; it dependence on the nature of the tree. If the tree is balanced; suppose there are n nodes this is the balance tree if the tree is balance now how much time it will take to insert how much time it will take to build such a tree. So, how many node are there here in the leafs? So there are basically n/2 nodes there are total n nodes and this is the height, height is basically $\log_2 n$. Now this is the depth of this node this depth is log n. So, for each of this leaf node we have to compare root then these then these with these then only it came here. So, that is the depth of this node, depth of this node is log n.

So, at least for each of this node we have to compare log n time. So, log n/2 n is the comparison for all these node and even we have these node. So, that is why time must be greater than n log n. So, this is the depth, this must be greater than n log n because there are n/2 nodes. So, at least for this node the depth is log n for this node. So, depth is log n means

they have to compare with this node this node. So, how many comparison at least log n comparison each of this leaf at to compare with the log n many nodes. So, that is why we inserted in that height level. So, that is why this time is basically big omega of n*log(n) this is the lower bound. We cannot do better than this. And any upper bound whether this is any big O. So, this we have to think. So, this is the best case.

(Refer Slide Time: 15:48)



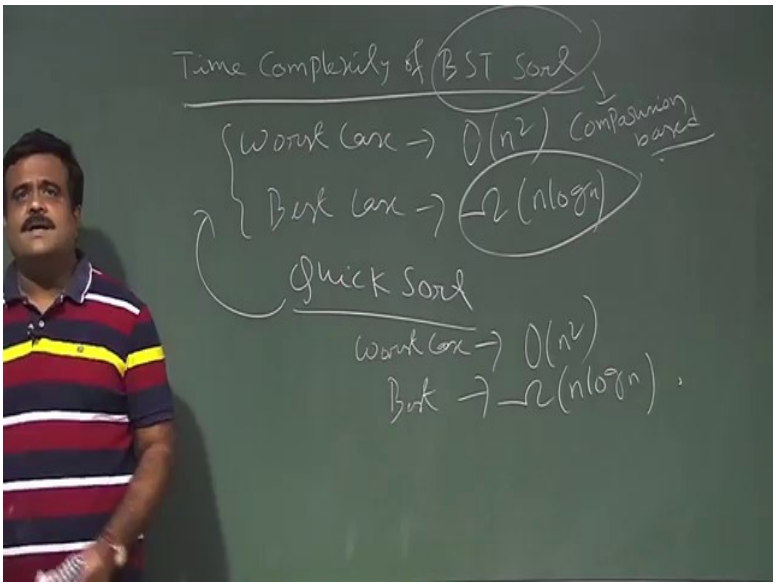
Now can we say this is big O of n square. So, when is the worst case of this scenario? Worst case is if the tree is bad tree like if the tree is like this if there are n nodes resulting in maximum height. So, then what is the summation of depth of X? X is in tree, so this basically arithmetic series, so this is n*(n+1))/2. So, this is order of n square. So, if the tree is bad tree so that means, if our numbers search is ascending order or descending order suppose we have given numbers are like this say 2 4 6 7 8 9 suppose this is our input. Suppose this is our input sorted, the our array is sorted already then what is the tree, tree will be like this. So, 2 then 4 6 7 8 9 see. So, this is basically to from this tree we need order of n square because everybody has to compare with the that previous all the element.

Even if it is reverse sorted also this will be like this if it is reward sorted the tree will be looks like this. So, this is the worst case. So, worst case is order of n square to build the tree. So, this is the time complexity or, time to build the tree. So, this is the worst case so that means so this will the time for BST sort. So, this will be the time for BST sort because inorder

traversal will take linear time and this will take the measured time will take by here to build BST to build the binary search tree and that itself is taking the time of order of n square.

So, the time complexity for BST sort in the worst case it is order n square and in the best case it is order of n log n.

(Refer Slide Time: 18:27)

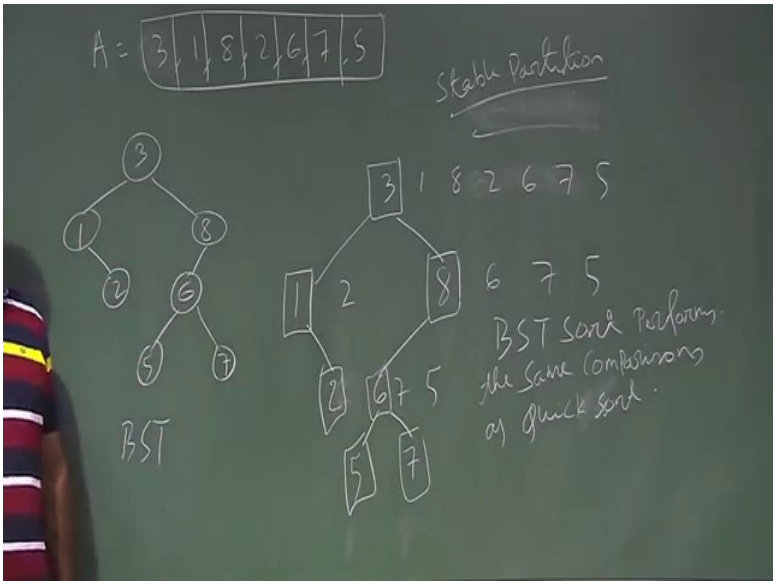


So, the time for runtime or time complexity of BST sort. So, worst case it is order of n square and the best case it is n log n. So, BST sort, this is the sorting algorithm which is taking worst case n square and best case n log n. So, is it remind us some sorting algorithm we know who performance is like this, like worst case is n square and the best case is n log n. So, we know any sorting algorithm we have studied comparison based sort because here this is also the BST sort is also comparison best sort because this is also a comparison best sort and that is why lower bound has to more than n log n; I mean we cannot do better than n log n we have seen by the this is entry method module that we have seen any comparison best sort worst case time complexity is more than cannot be faster than the n log n.

So, this is also comparison based sort because we are comparing the element and then we are forming tree and then we are inorder traversal. So, any comparison based sort we know having this time complexity yes quick sort. So, quick sort is having same type of time complexity worst case is order of n square and the best case is order of n log n. So, quick sort is having the same time complexity has BST sort. So, that is the our next point to see whether is there any relationship between BST sort and the quick sort why they are giving us the same

time complexity. So, that we want to see that what is the relationship between BST sort and the quick sort. To see that let us take the example again and we will see that they were having same number of comparison we are doing in quick sort and the BST sort. So, let us take that same example A array, 8 sorry 3 1 8 2 6 7 5 this is our given input and in the BST sort we form the tree 3 then 1 then 8 then here 2 here 6 and then 5 7.

(Refer Slide Time: 21:14)



So, this is the BST binary search tree and then we do the inorder traversal to have this thing. So, have the sorted one to have the sorted one array. So, this is BST sort. Now let us talk about quick sort performance on this input. So, in the quick sort what we are doing we are choosing a pivot element suppose we choose and we choose the first element as a pivot if we choose the first element as a pivot. So, it looks like 3 is a pivot. So, what pivot will do, pivot will partition this array into 2 sub array all the element less than X must be left sub array all the element greater than X possible right sub array. So, that way it will do like this. So, 3 this is the array 3 1 8. So, let us just 3 1 8 2 6 7 5. So, this is the given input now we choose this as a pivot element.

So, it will partition this into two part all the element will be less than X must be in the left sub array all the element greater than X must be in the right sub array. So, who are the element? So, 1 2 must be sitting here and 8 6 7 5. So, this is quick sort partition and here we are assuming one thing this partition must be stable partition because we are assuming this ordering will not change. So, we know this is 1 2 will come in left sub array, but it is not

guaranteed that 2 will come. So, which ordering, but here we are using a partition which is call stable partition and it is very tough to get the code of stable partition. So, stable partition means we are assuming the ordering of this input ordering same has this in the array. So, there are 8 6 7 5 there in the same order.

Now again for this we are taking this as a pivot now all the elements. So, 2 will be sitting here and we are taking this as a pivot. So, who are element are greater than all the element are greater than less than 8. So, 6 7 5 then 2 then nobody is there then we call 6 is here, so this is basically 5 and 7. So, this is the recursive call of the partition. Now if you just form the tree like this, like this, so like this if you look this 2 tree as same, so that means, we are doing the same number of comparison. So, in the recursive call of partition the number of comparison we are doing is same as to form the BST because to form the BST this 6. So, when you insert 6, 6 has to compare with 3 6 has to compare with 8 here also.

So, to come here 6 as to compare with 3 because this will partitioning into 2 part again 6 as to compare with this 8. So, the same number of comparison we are doing for the both the cases. So, this is the idea. So, BST sort. So, this is the observation BST sort basically to form the BST performs the same number of comparison may be different order, but we are doing same number of comparison as in quick sort. So, that is the reason they are giving the same time complexity, the time complexity is coming out to be same because of this fact that we are basically doing the same number of comparisons in both the cases to form the BST and for the quick sort.

Here we are assuming the partition is stable partition because to and that is also not tough to achieve to have this 2. So, the same tree we are having this. So, this is the observation and that is the reason it is giving us the same time complexity for the BST sort and the quick sort.

So, in the next class we will talk about randomized version of the BST sort and there we will see the height of the randomly build BST. So, the expected height we will see to be balanced log n. So, that we'll cover in the next class.

Thank you.