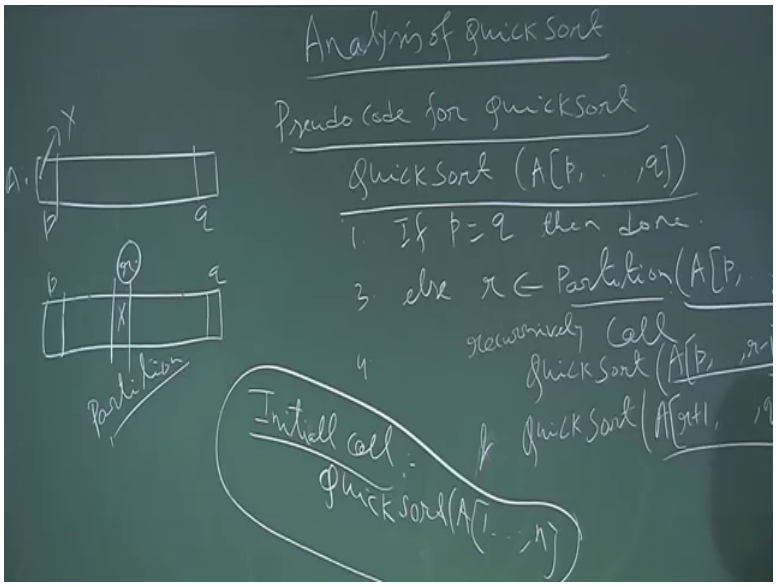
An Introduction to Algorithms
Prof. Sourav Mukhopadhyay
Department of Mathematics
Indian Institute of Technology, Kharagpur

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 1 sorry; p is equal to q then done then stop because that is the return 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 have 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, this is A 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 (Refer Time: 01:56) it is keeping this pivot element. So, this is the partition this is the partition sub

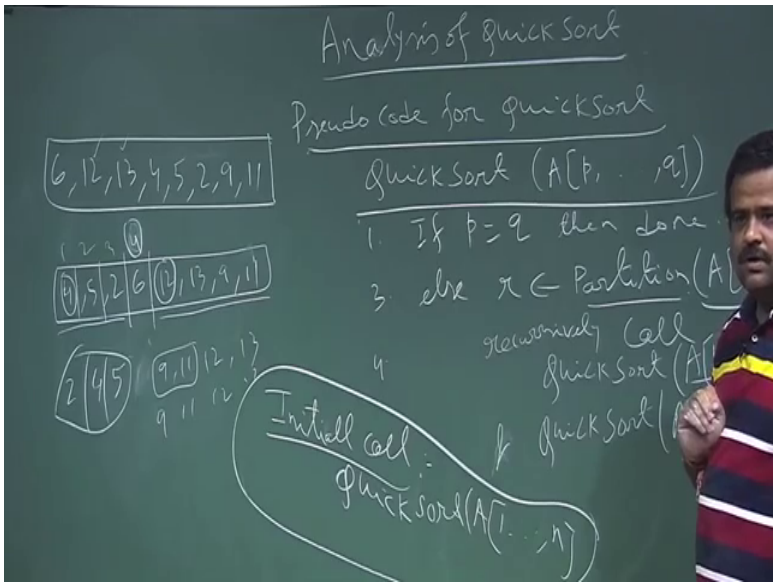
routine if doing this. So, this we have discussed in the last class and this is the linear time algorithm. So, we call the partition. So, after calling the partition it will divide into 2 part 2 sub array this is left sub array right sub array then again we have to call that is the divide step of the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

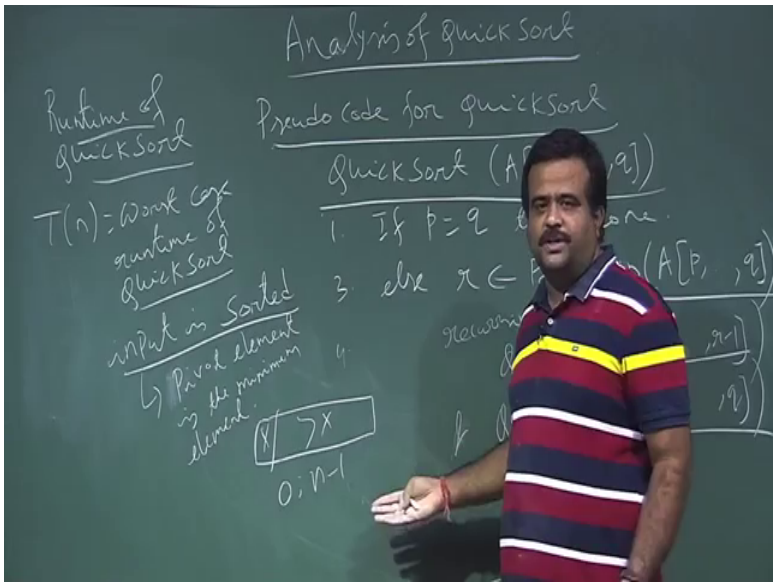
So, it will partition the array into 2 parts. So, 6 it will put 6 somewhere here and it will divide into 2 parts left sub array all the element which are less than x will be left sub array. So, they are basically 4 5 2 and the all the elements which are greater than 6 will be in the right sub array. 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 plus 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 partitioning to 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 run time 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 run time for quick sort worst case time complexity for quick sort. So, now, when is the

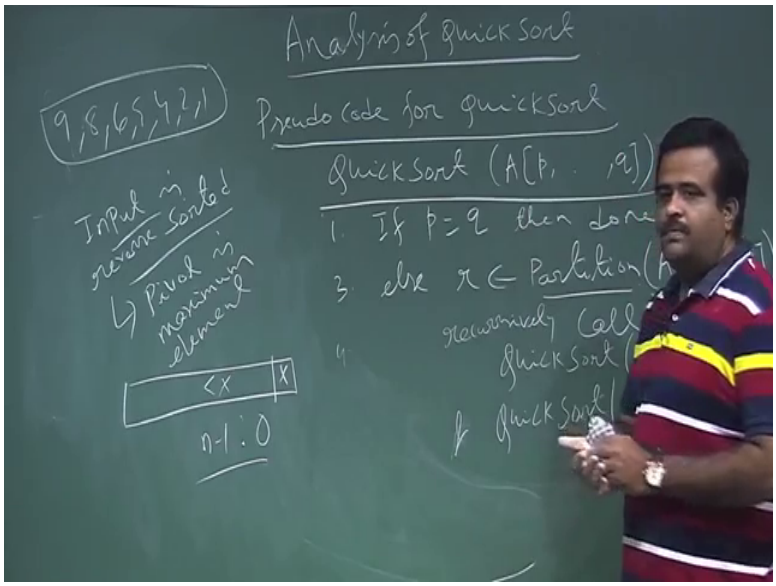
worst case for quick sort when is the worst case for this code. So, worst case means. So, if the partition is dividing the array into 0 is to n minus 1.

So, worst case will occur if we have a 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 minus 1 partition. So, nobody is there in the left sub array and n minus 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 if we have a sorted array as a 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 minus 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 minus 1. So, throughout this execution it will be 0 is to n minus 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.

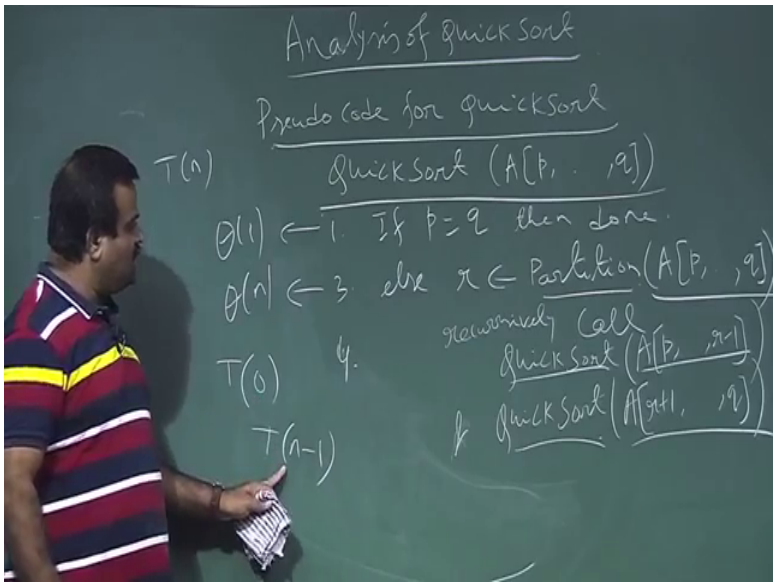
(Refer Slide Time: 10:53)



So, suppose our input is input is reverse sorted 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 the partition will partition this into. So, maximum means it has to be here and all the elements has to be less than x.

So, it is basically n minus 1 is to 0. So, if the pivot is to be maximum or minimum then it will be either 0 is to n minus 1 or n minus 1 is to 0 so; that means, there is no one part it is sub array size is 0. So, that is why that is the worst case will come to that. 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.

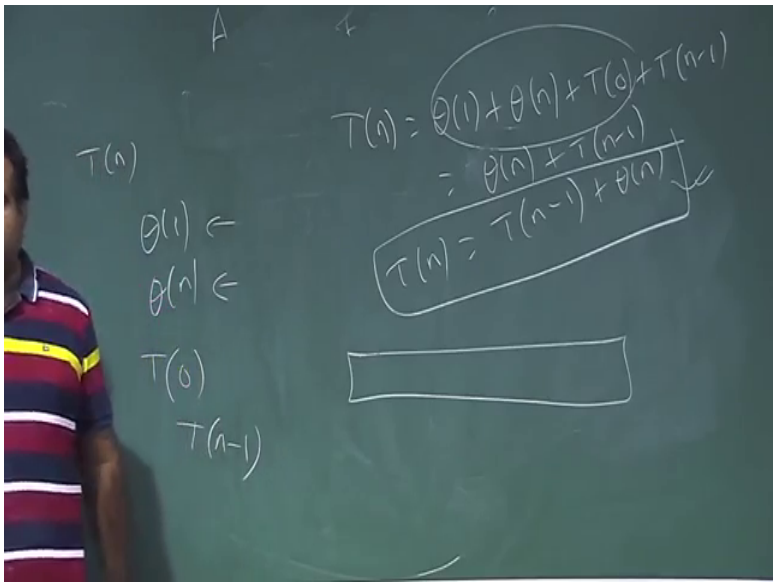
(Refer Slide Time: 12:28)



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, this 2 call, but we know pivot is here we are taking pivot is minimum or maximum. So, we have 1 call is on 0. So, this is T 0 and T n minus 1 or T n minus 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 plus theta of n plus T 0 plus T of n minus 1 or T of n minus 1 plus T 0.

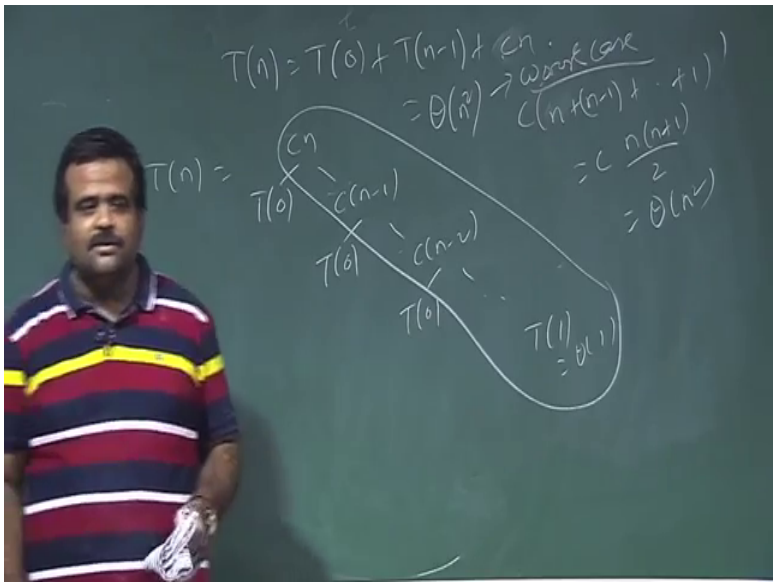
(Refer Slide Time: 13:15)



So, this will all this like theta of n plus T of n minus 1. So, this is basically T of n minus 1 plus theta of 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, we put this is sorted in input. So, then this is the minimum or if it is reverse sorted 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)



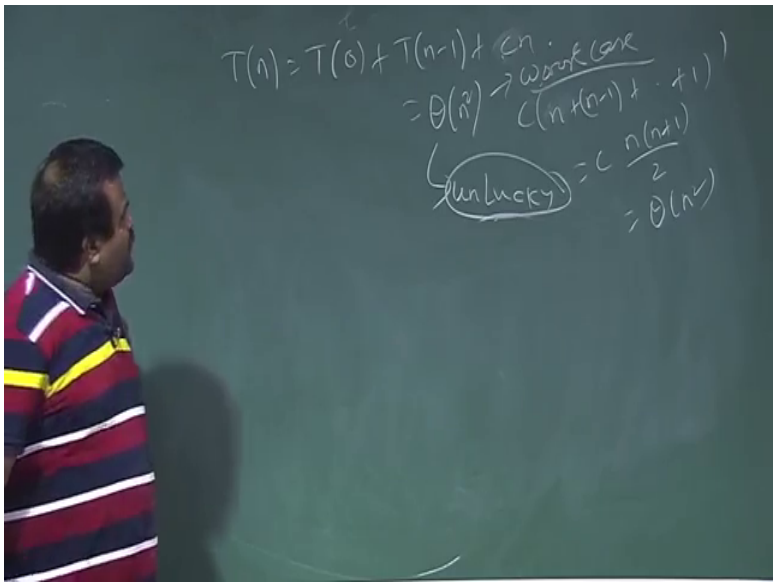
So, to know this time complexity what we do we draw the recursive tree this is the worst case recurrence. So, we draw the recursive tree. So, basically what we have this is T n is basically T 0 plus c n theta of n the T 0 is just a symbolic notation. So, if we draw the recursive tree T n is basically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

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

So, in that case it will be again c of n minus 2 T 0 like this. So, this will continue until we stop 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 1 n into n minus 1. So, this is basically c into n plus 1 by 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.

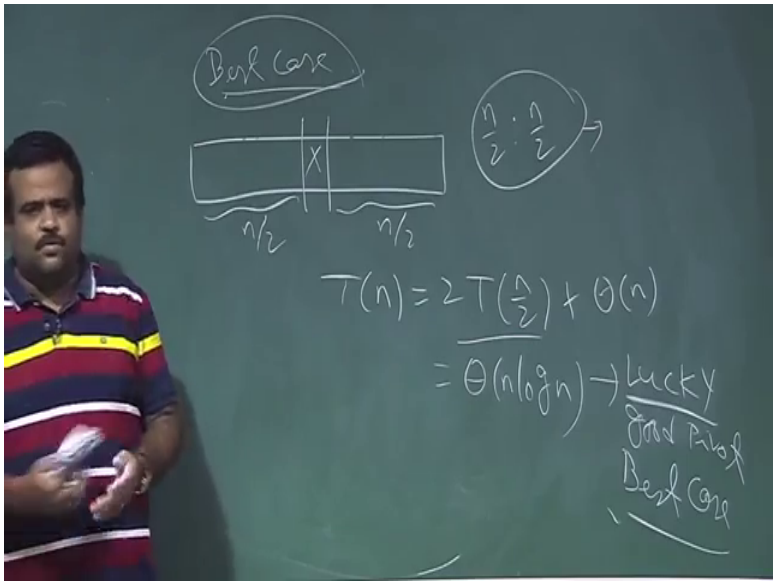
So, what is the lucky case when we get order of n log n algorithm time is order of n log n then we call this is a lucky case. So, this is a unlucky case.

(Refer Slide Time: 16:51)



So, if the 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, best case may be. So, we know the if the pivot is bad pivot bad pivot in the sense if it is partitioning 0 is to n minus 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 is; that means it will partition. So, then what is the good pivot.

(Refer Slide Time: 17:36)



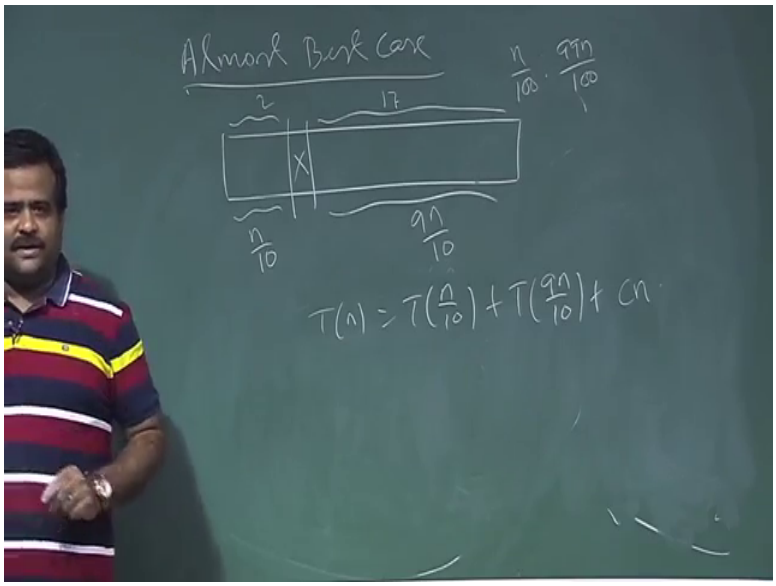
So, that will give us a best case. So, what is the good pivot intuitively if the pivot is partitioning the array into half half so; that means, if the pivot is. So, we have given a input if the pivot is partitioning into half half pivot is sitting here this is roughly n by 2 this is n by 2 n by 2 minus 1. So, that is puffiness will do. So, that is lower ceiling or upper ceiling. So, this is 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 equal to 2 T n by 2 plus theta of n because in that case pivot is sitting here again we have to call quick sort for this quick sort for this, but they this 2 have the same size n by 2 n by 2 or n by 2 minus 1 that will be also n by 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 by 2 is to n by 2 now this recurrence we have seen this is the same recurrence has merge sort. So, what is the solution again if we use the master method we can get this solution n log n. So, good, so, this is the lucky case lucky. So, good pivot best case anything you can name. So, this is the best case scenario when the pivot is partitioning the array into 2 equal part half half n by 2 is to n by 2 then our recurrence is just because we have to call again quick sort on this quick sort on this.

So, T of n by 2 T of n by 2 so, that that is why 2 T n by 2 plus that is 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 minus 1 like minimum or maximum it is not the bad 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 not worst than 0 the 0 is to n minus 1 suppose our pivot is such that it is partitioning the array into 2 part 1 is a n by 10 another is 9 n by 10 or may be 1 is n by 100 another 1 99 n by 100 for simplicity n by 10 9 n by 10. So, it will work for this also. So, that is the ratio 1 by 10; 9 n by 10. So, this ratio it is partitioning.

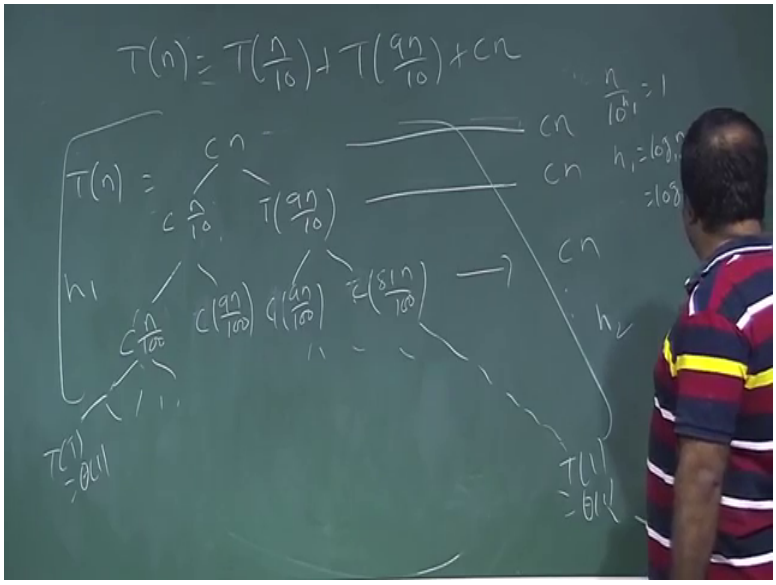
So, pivot is such that it is partitioning in this ratio for example, suppose say n is say 20 if n is 20 then how many elements are there this is n by 10. So, this is 2 and this is x if (Refer Time: 21:24) this is 17. So, this is only 2 element. So, if we can assure that pivot is such that it is partitioning 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 minus 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 is basically T of n by 10 plus T of 9 n by 10 plus theta of n

or basically this is the merge this is the partition cause c n. So, we want to know the; this is the recurrence for this case almost best case 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 minus 1 it is better than 0 is to n minus 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 is basically T of n by 10 plus T of 9 n by 10 plus c 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 by 10 T 9 n by 10 now again this is a problem of size n sorry size n by 2 n by 10 now again we will use the quick sort for solving this problem again we assume that our partition is portioning this into 1 by 10 is to 9 by ten. So, this assumption is required is not that suddenly we assume here for this subsequent step.

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 by 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 by 10 is to 9

by 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.

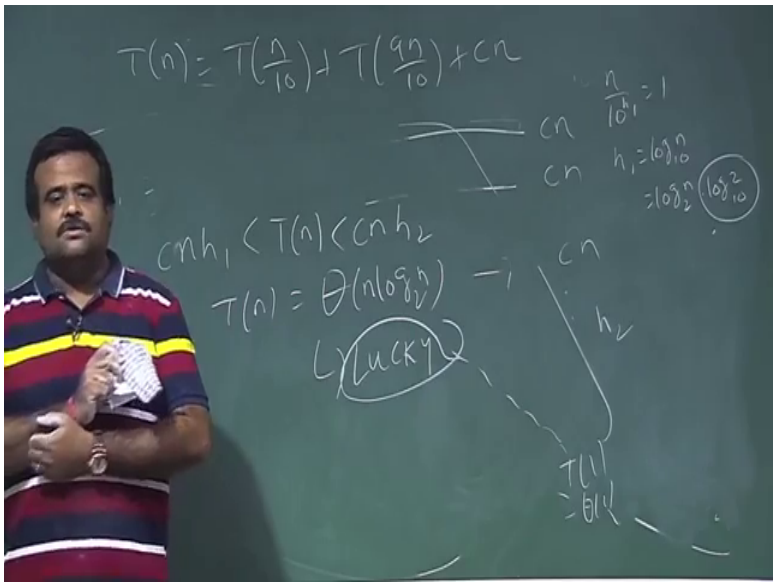
So, again this will be this is now size n by ten. So, this will be T of. So, it just put n is equal to n by 10 n of hundred and this is basically T of 9 by hundred and again this will be T of. So, we put n is equal to 9 n by 10. So, this is 9 n by hundred and this will be 81 n by 100. So, this way, again this is a problem of size n hundred and again we assume this is will have the same recurrence again our pivot is such that it will be 1 by 10 is to 9 by 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, when we stop 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, soon, all the branches that this 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 recursive tree sorry substation 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 n base 10, but we can make it log n base 2 and 2 base 10. So, this is just a constant. So, height is order of log n. So, this side also height will be order of log n. So, basically time is. 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 lucky case this is lucky n log n is the lucky case.

So, that is why it is called (Refer Time: 24:34) based. So, that is the thing. So, if our pivot is not half half if we if our pivot is such that it can assured that some fraction 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, that is that is why. So, if it is not even half half if it is just 1 by 10 is to 9 by 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 will do in the next class. So, there will 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 will do in the next class.

Thank you.