

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.

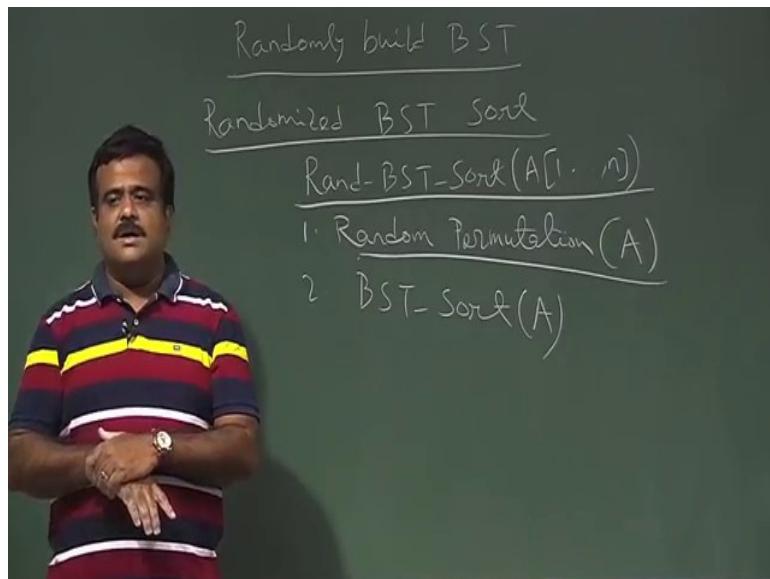
Thank you.

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

**Lecture - 26**  
**Randomly Build BST**

So we will talk about Randomized BST Sort. We have seen the BST sort and we have seen the relationship of BST sort and quick sort they have basically the same number of comparisons. So, the time complexity of BST sort is same as the time complexity of quick sort.

(Refer Slide Time: 00:52)



So, we will see that the idea is to randomly built BST; so for that let us look at the randomized version of the BST sort.

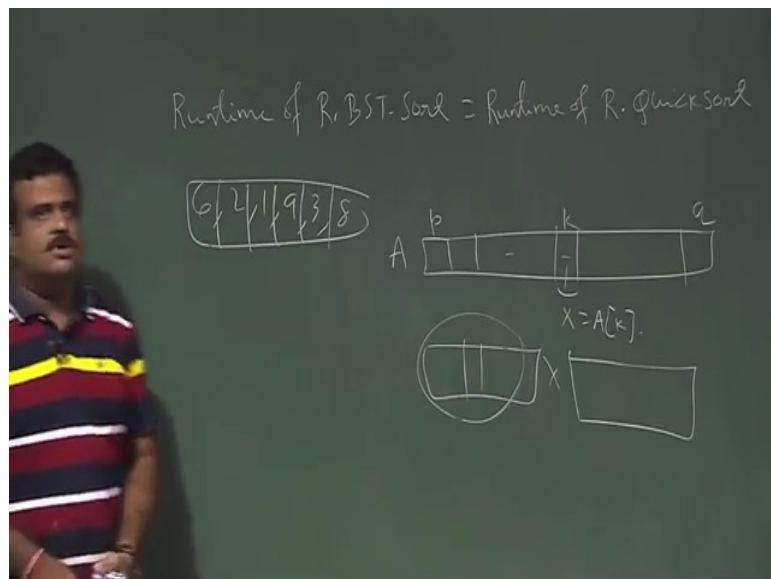
So, for this; what we do? So, this is we can denote by rand BST. So, we are given a n numbers we are given a array of n element. So, how we can have a sorting algorithm?

So, randomized BST sort means before doing anything we just randomly permute this. So, we will do the random permutation we apply random permutation on this numbers and then call the BST sort. So, basically we have n number we are given a n numbers and we are just randomly permuting these numbers and then we are building the tree BST; and then we are doing the inorder traversal. So, this is the randomized version of the BST sort. So, just we are

doing the random permutation before we form the BST so, that BST is called randomly build BST. So, given n numbers before building the BST we just have a random permutation on this number. So, then we form the BST and then we do the inorder traversal and then that will give us a sorting algorithm.

So, this is referred to as randomized BST and this is same as the if we just look at consider the randomized quick sort in the randomized quick sort also what we are doing we are choosing the pivot element randomly and then we are partitioning into 2 parts so; that means, the time complexity of the BST sort we have seen the BST sort and the quick sort having same time complexity. So, the time complexity of the BST sort is same as time complexity of randomized.

(Refer Slide Time: 03:32)



This BST sort is same as time complexity of randomized quick sort. So, this means the runtime of randomized BST sort is same as runtime of randomized quick sort.

Because in randomized quick sort we are choosing the pivot element randomly and then we are performing the partition and that will divide into 2 parts again we are choosing the pivot randomly. So, what is in the randomized fixed we have given array. So, what we have doing we are choosing the pivot randomly. So, suppose this is our p to q. So, we are choosing an element index A k randomly and this is going to be our pivot and we are partitioning this array into 2 parts x is sitting here like this. So, again we will call randomized quick sort on this. So, again it will be choosing a element randomly from here. So, this forms a tree and this

tree is same as the randomly build BST because randomly build BST means we have given some numbers. So, this is the number we are given.

Now, we are doing a random permutation on this. So, there are how many permutation? If there are  $n$  numbers there factorial  $n$  permutation. So, among these we are choosing one permutation randomly. So, equally likely and then we are forming the tree. So, this is basically same as this randomized version of the quick sort. So, that is why their time complexity is same. So, now, this is a randomized algorithm. So, we need to talk about the average case analysis.

(Refer Slide Time: 05:45)

The chalkboard shows the following content:

- A list of numbers in a box:  $\boxed{3, 5, 6, 9, 10, 11}$
- $E(\text{Runtime of R.BST.Sort}) = E(\text{Runtime of R.QuickSort}) = \Theta(n \log n)$
- $E\left(\sum_{x \in T} \text{depth}(x)\right) = \Theta(n \log n)$
- $E\left(\frac{1}{n} \sum_{x \in T} \text{depth}(x)\right) = O(\log n)$
- A curved arrow points from the term  $\frac{1}{n} \sum_{x \in T} \text{depth}(x)$  to the text "Avg depth of a Randomly build BST =  $O(\log n)$ ".

So, you take the expected value of this. So, expected runtime is same as expected runtime of randomized quick sort now we know this expected runtime of randomized quick sort is order of  $n \log n$ .

So, now what is the runtime for BST sort? Runtime of BST sort is basically the time it required to form the tree BST insert. So, this is basically summation of depth of  $x$  that  $x$  is  $t$ . Or we can say big o or big theta ( $n \log n$ ). So, now, we take the same so, expected. So, this is basically  $1/n$ . So, this is giving us order of  $\log n$ . So, this is basically average depth. So, this is basically average depth of a randomly build BST is basically order of  $\log n$ .

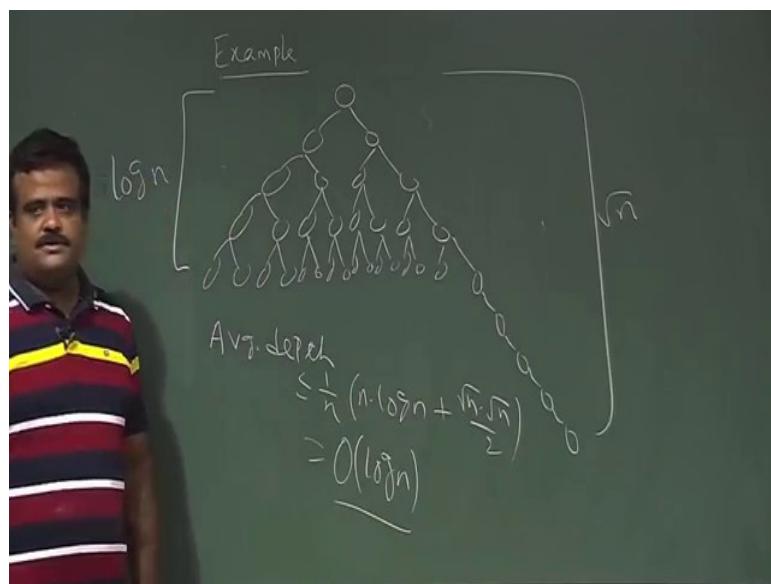
Actually our goal is to know the expected height of this randomly build BST and we have to show that is basically the  $\log n$ . So, expected height of this be a randomly build BST

randomly build BST means we just perform a random permutation on these numbers and then we form the BST and magic that it will be a balanced tree usually the tree is not balanced depending on the input suppose input is say sorted say 3 5 6 9 10 11. So, for this input if we form the tree it is not balanced it is just like this.

Now, the magic is if you do a random permutation on this and then after that if you form the BST that is what is called randomly build BST and that will be expected balanced tree. So, the height expected height of that tree is  $\log n$ . So, that we are going to establish. So, for that we need to bring this randomized version of this BST sort and the comparison between the quick sort they are same, but all these things we are doing to establish that if we have the randomly build BST that is expected that will be the balanced tree I mean on an average expected. So, this is the average case analysis. So, that we are going to show.

Now so, far what we get? We get the average depth of a randomly build BST as  $\log n$ . So, does it mean the height will be  $\log n$  if the average depth is  $\log n$ . So, let us see if we can have a tree wherever is depth is  $\log n$ , but the height is not  $\log n$  height what is the height of a tree height of a the tree is the maximum depth of the tree that is the height of the tree. So, this does not mean the height is  $\log n$ . So, what is the example?

(Refer Slide Time: 09:42)

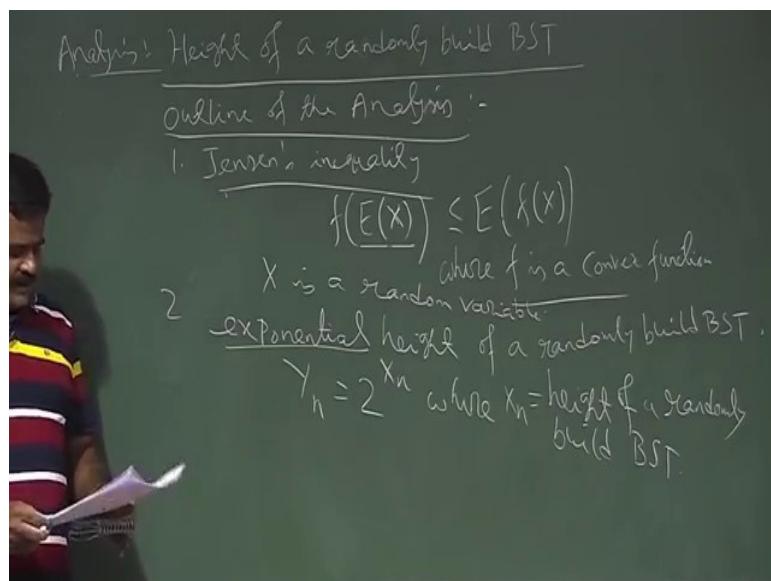


Let us take an example where average depth of the tree is  $\log n$ , but the height is not  $\log n$  suppose we have a tree like this. So, almost all the path it is balanced.

Except some branch now suppose this is basically this branch is root over of n and suppose remaining all the elements are balanced. So, only we have one branch which is more. So, now, all other elements are like this; so, only one branch. So, what is the average depth over here? Average depth is less than equal to  $(1/n)*n \log n$ . So, there are n elements which are of height  $\log n$  and there are root over of n element which are height root over of  $n/2$  sort of. So, and this is basically  $\log n$ .

So, average depth is  $\log n$ , but the height of this tree is root over n; height of the tree is not  $\log n$ . So, even the average depth is  $\log n$  height may not be  $\log n$  by seeing this example, but I mean randomly build BST we have to prove that the expected height is  $\log n$ . So, that need a little mathematics to prove that. So, let us try that. So, let us try to prove that randomly build BST expected height is  $\log n$  so; that means, we have given some input we just do a random permutation on it we just apply random permutation then we form the tree BST and then expected that BST will be a balanced tree. So, let us try to prove that.

(Refer Slide Time: 11:56)



So, this is the height of a randomly build BST and this is expected value of this is  $\log n$ . So, this analysis we'll do. So, this is the outline of this analysis. So, we are going to show this expected value of this analysis. So, what is the outline? So, outline is basically we first prove a Jensen inequality. This is about the convex function what it is telling it is telling expected value of or  $f$  of  $f$  is a convex we will prove that expected value of this is less than equal to  $e$  of  $f$  of  $x$ . So, this is basically where  $f(x)$  is a convex function. So, we will prove this convex

function and we will defend the convex function this is the first part of the analysis. So, for any function  $f$  which is a convex function for that function  $f$  of  $e$  of  $x$  expected value of  $x$  is basically  $e$  and  $f$  will be exchanged less than equal to  $e$  of  $f$  of  $x$  and  $x$  is a random variable.

So, we analyse the exponential height of this randomly build BST. So, that is basically we denote by  $Y_n$  to be  $2^x n$  where  $x_n$  is the height of a randomly build BST expand  $x_n$  is the height of a randomly build BST with  $n$  elements. So,  $x_n$  is the height of a randomly build BST with  $n$  element when we define we are taking the exponential height. So, we define  $Y_n$  to be  $2^x n$  and that will help us to get the expression in the next step.

(Refer Slide Time: 15:24)

$$3. \text{ Prove that } E(X_n) \leq E(2^{X_n}) = E(Y_n) = O(n)$$

$$\Rightarrow E(X_n) = O(\log n)$$

$X$  is a random variable whose  $f$  is a convex function

exponential height of a randomly build BST.

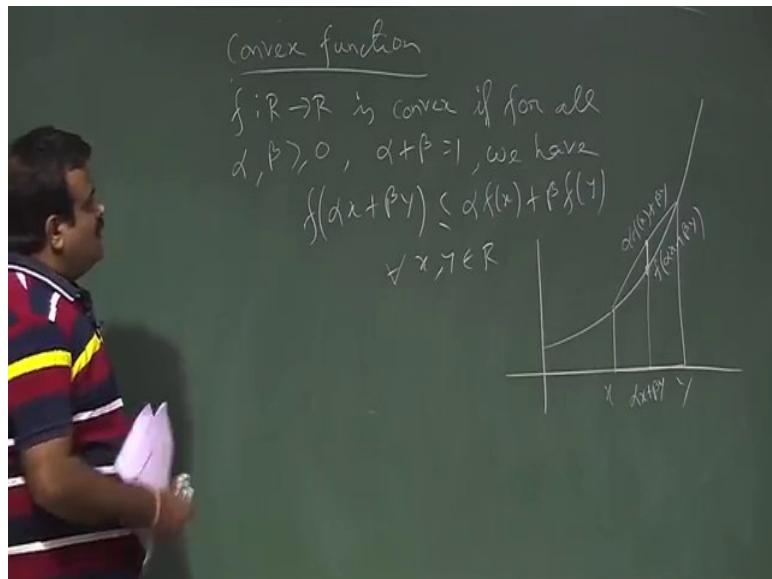
$Y_n = 2^{X_n}$  where  $X_n$  = height of a randomly build BST.

In the final step we prove that  $2^x$  is a convex function basically. So, we will use this Jensen; Jensen inequality is basically expectation of  $2$  to the power  $x_n$  which is basically  $Y_n$  we prove this  $Y_n$  to be order of  $n$  cube.

If we can show this then this implies expected value of the height is basically order of  $\log n$  this we have to establish finally, to establish this we first prove expected value of  $Y_n$  is order of  $n$  cube if we can prove that then by using the Jensen inequality we can say  $2$  the power expected value of  $x_n$  is less than equal to expected value of  $2$  to the power  $x_n$  because this  $2$  to the power  $x$  is a convex function and then this is basically our  $Y_n$  and if we can show this  $Y_n$  is bounded by  $n$  cube then this is telling us expected value of  $x_n$  is  $\log n$  so; that means, it is a balanced tree. So, this is the outline of the proof outline of the analysis now let us talk about the Jensen inequality.

So, for that let us define; what is the convex function. So, convex function 2 to the power x is a convex function and for convex function we are using that Jensen inequality the theorem.

(Refer Slide Time: 17:15)



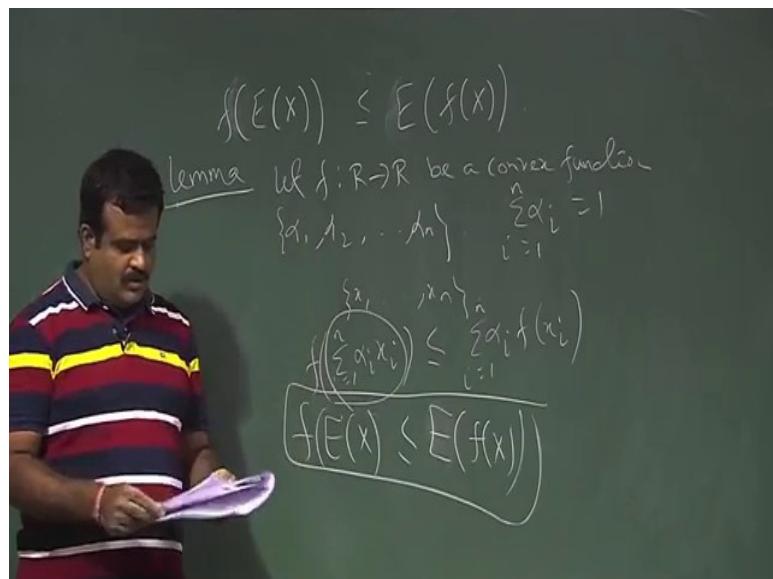
So, if function  $f$  from  $\mathbb{R}$  to  $\mathbb{R}$  (real number set to real number) is convex function this is the definition is convex if for all alpha beta greater than 0 such that alpha plus beta is one then we have  $f$  of alpha  $x$  plus beta  $y$  is less than equal to alpha  $f$   $x$  plus beta  $f$   $y$  where for all  $x$   $y$  belongs to  $\mathbb{R}$ . So, this is the definition of convex function.

A real valid function  $f$  is called convex function if for a given alpha beta if for all alpha beta greater than 0 such that their sum is one we have this. So, like if we have a example. So, if we have this function this function is convex. So, if you take 2 point  $x$  and  $y$ . So, now, this is basically. So, if we take alpha; alpha  $x$  plus beta  $y$  where alpha beta is one. So, this is this point is basically here alpha  $x$  plus beta  $y$ . So, this point if we take the value of this function and this is the; if we take this straight line.

And this is basically alpha  $f$   $x$  plus beta  $f$   $y$ . So, this is basically should be less than and this is basically  $f$  of alpha  $x$  plus beta  $y$ . So, the curve is like this. So, this must be less than this value. So, curve will be like this. So, this type of function is called convex function. So, if you take any 2 point and the curve. So, if you take any 2 point  $x$   $y$  and if you take any other point then it is the all the point inside must be inside of that curve. So, this type of function is convex function and the Jensen inequality is telling and e to the power can be shown to be convex function and the lemma which is the Jensen inequality this is telling us expected if we

have a convex function  $f$  then the expected value of  $f$  of expected value of  $x$  is less than equal to  $E(f(x))$ .

(Refer Slide Time: 20:02)

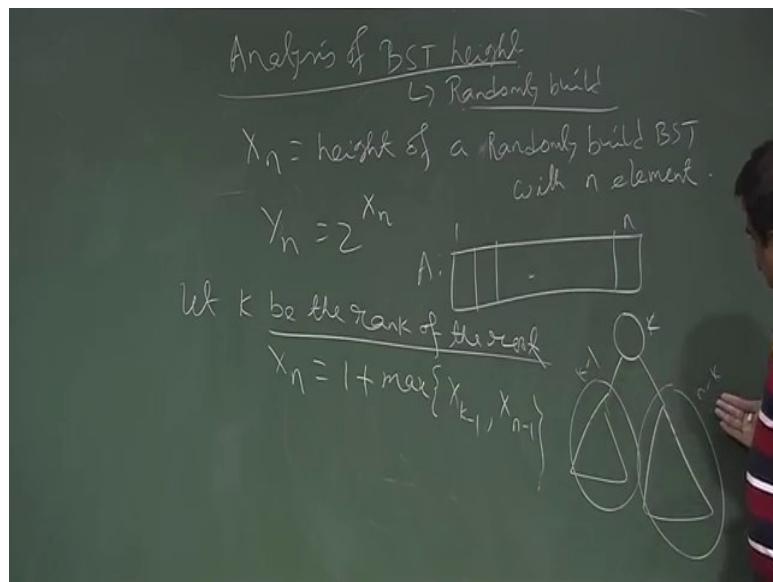


So, how to prove that; so, to prove that we have to use this lemma. So, let  $f$  be a convex function  $f$  be a convex function and then we have a set alpha one. So, this is generalized function alpha one, alpha  $n$  there we have alpha beta, but if you have  $n$  elements such that summation of alpha is equal to  $n$  then if we have the point  $x_1 x_2 x_n$  this is generalized version of the; which we have seen for 2 points.

This prove can be extended by using the earlier one  $f$  of summation of alpha  $x_i$  less than equal to and this can extend for infinity also provided this sum is absolute convergent summation of summation of alpha  $i f(x_i)$ . So, this  $i$  is from one to  $n$  and this  $i$  is from 1 to  $n$ . So, this is basically giving us this now if  $x$  is a discrete random variable if  $x$  is a discrete random variable with this points then this is nothing, but and these are the points and this is the probability then this is basically expectation. So, this is basically  $f$  of expectation of  $x$  is less than equal to summation of this is the probability. So, this is basically expectation of  $f$  of  $x$ .

This is the definition. So, this is basically our Jensen inequality and this is true for discrete random variable. So, this is the proof for Jensen and we are not this; this can proved using the induction. So, we are not going to the details of this proof now let us go to the height analysis.

(Refer Slide Time: 22:34)



And there we will use this Jensen inequality for  $2^x$  case. So, analysis of BST height randomly builds BST height. So, we have taken  $x_n$  to be height of the of a randomly build BST with  $n$  element.

With  $n$  element; so, we have given an array of  $n$  elements. So, what we are doing we just permuting these array randomly and then we are forming the BST and  $x_n$  is the height of that BST after permuting and forming the BST of that BST and we are taking  $Y_n$  to be  $2$  to the power  $x_n$  and our goal is to if you recall the outline of the our goal is to. So, that expected value of  $Y_n$  is equal to order of  $n$  cube. So, now, now what is the proof now suppose? So,  $x_n$  suppose this is the we are forming the BST now we do not know the we are doing the random permutation we do not know the we are doing the random permutation we do not know the which will be the root we have given  $n$  number.

We are given  $n$  numbers and we are just permutating it. So, we do not know which one will be the root. So, we do not know the rank of the root, suppose if the rank of the root is  $k$  and that is also random variable let  $k$  be the rank of the root rank means the position of that element in the sorted array of the root then  $x_n$  can be written as basically:  $1 + \max(x_{k-1}, x_{n-k})$  because root is sitting here. So, if rank of this is  $k$  then we know how many elements are there how many there are  $k-1$  elements and there are  $n - k$  elements over here because rank is  $k$ .

So; that means, so, the height of this tree will be we do not know which is maximum which is minimum height of this tree will be  $1 + \max(x_{k-1}, x_{n-k})$ .

(Refer Slide Time: 25:33)

The image shows a person from behind, wearing a red, white, and blue striped polo shirt, standing in front of a chalkboard. The chalkboard contains the following handwritten notes:

$$Y_n = 2 \cdot \max\{Y_{k-1}, Y_{n-k}\}$$

$$Z_{nk} = \begin{cases} 1 & \text{if the root has rank } k \\ 0 & \text{otherwise} \end{cases}$$

$$P(Z_{nk} = 1) = \frac{1}{n}, \quad E(Z_{nk}) = \frac{1}{n}$$

$$Y_n = \sum_{k=1}^n Z_{nk} (2 \cdot \max\{Y_{k-1}, Y_{n-k}\})$$

$$E(Y_n) = 2 \sum_{k=1}^n E(Z_{nk}) \cdot E(\max\{Y_{k-1}, Y_{n-k}\})$$

So, now we will take the  $y$ . So,  $Y_n$  will be what. So, from here we can say  $Y_n$  will be  $2^x n$ . So,  $2$  of max of  $x_n$ ; so, again it will be  $2$  to the power of that. So,  $\max(Y_{k-1}, Y_{n-k})$ . So, this is of the form now we will take the help of indicator random variable.

So, we define  $Z_{nk}$  is one if the rank of root is if the root has rank  $k$  otherwise it is 0 then the probability. So, these are equally likely probability of this is 1 by  $n$ . So, the expected value of  $Z_{nk}$  is basically 1 by  $n$  and. So,  $Y_n$  will be basically any one of this because rank is  $k$ . So,  $k$  can take value one to  $n$  we do not know which  $k$  will occur. So,  $Y_n$  is basically summation of  $Z_{nk}$  into this expression  $2 * \max(x_{k-1}, x_{n-k})$ . So, this type of analysis we did. So, this is our  $Y_n$ . So, this is from one to  $n$  now we take the expectation both sides. So, expectation is a linear function expected value of this and this now again if we take the independence of these two. So, this will give us expectation of  $Z_{nk}$  into expect 2 will come out here. So,  $k$  is 1 to  $n$  expectation of max of  $Y_{nk}$  comma  $Y_{n-k}$  this 1. So, let us just give this. So, these we want to further simplify.

So, now this is basically max of this. So, max of 2 numbers is less than equal to sum of their number.

(Refer Slide Time: 28:09)

The chalkboard contains the following derivation:

$$\begin{aligned}
 E(Y_n) &\leq \frac{2}{n} \sum_{k=1}^n E(Y_{k-1} + Y_{n-k}) \\
 &= \frac{4}{n} \sum_{k=0}^{n-1} E(Y_k) \\
 &\leq \frac{4}{n} \sum_{k=0}^{n-1} C k^3 \\
 &\leq \frac{4C}{n} \int_0^n x^3 dx \\
 &= \frac{4C}{n} \left( \frac{n^4}{4} \right) = C n^3
 \end{aligned}$$

Annotations on the right side of the board include:

- $E(Y_n) = O(n^3)$
- $E(Y_n) \leq C n^3$
- $E(Y_k) \leq C k^3$
- $\forall k < n$
- I.H.

So, expectation of  $Y_n$  can be written as 2 off. So, expectation of this is 1 by  $n$ . So, 2 by  $n$  and this is basically expectation of summation; summation expectation of  $Y_k + Y_{n-k}$  because max of  $a + b$  is less than equal to. So, this must be less than and this  $k$  is what one to  $n$  and this is basically 4 by  $n$  summation of expectation of  $Y_k$  and this  $k$  is now adding from 0 to  $n-1$  because this term is coming twice. So, this is the expression we got. So, how we can further simplify this?

So, we will prove this with the help of substitution method. So, what we are expecting we are showing that we want to show the expected value of  $Y_n$  is order of  $n^3$ . So, this is our goal to achieve so; that means, expected value of  $Y_n$  is less than equal to  $C n^3$  this is this want to show by the use of substitution method. So, for this we are taking the induction hypothesis  $C k^3$  for all  $k$  less than  $n$  this is the induction hypothesis this table now we put it here. So, this will give us less than equal to 4 by  $n$  summation of  $C k^3$  for  $k$  from 0 to  $n-1$  by the induction hypothesis step.

Now, this we can again using the integration  $\int_0^n x^3 dx$ . So, this will be less than of this integration 0 to  $n$  improper integral  $x^3 dx$  now this if you simplify this will give us  $\frac{4}{n} C n^4$ . So, this is basically giving us  $C n^3$  so; that means, expected value of  $Y_n$  is less than  $C n^3$ . So, by the method of induction or substitution method we can say that this is true. So, how do we establish? We establish that expected value of  $Y_n$  is order of  $n^3$ , this we have proved using the substitution method.

(Refer Slide Time: 31:02)

The chalkboard shows the following derivation:

$$E(Y_n) = O(n^3) \cdot \dots$$
$$2^{E(X_n)} \leq E(2^{X_n}) = E(Y_n) \leq c n^3$$
$$\Rightarrow E(X_n) = O(\log n) \quad \boxed{\text{J.H}}$$

Below the main derivation, there are two additional equations:

$$\frac{4c}{n} \int x^3 dx$$
$$= \frac{4c}{n} \left( \frac{n^4}{4} \right) = cn^3$$

To the right, there is a circled note:

$$E(Y_k) \leq ck^3 \quad \forall k < n$$

I.H

So, now we take the 2 to the power expectation of  $y_n$  which is basically less than equal to expectation of  $2^x n$  and this is the by Jensen inequality. So, this is basically expectation of  $y_n$  and this is basically order of  $n$  cube. So, from here we can say expected value of  $x_n$  is order of. So, order of  $n$  cube. So, expected value of  $Y_n$  is basically order of less than equal order of  $\log n$ . So, less than we can use the less than less than equal to  $c n$  cube then we take the log both side then it will give us  $c \log n$ . So, it will be order of  $\log n$ . So, that is the expected height of the randomly built binary search tree is height is  $\log n$ . So, it is balanced. So, expected it is balanced.

But the worst case is always like this. So, worst case is always like this because it may happen that we are doing the random permutation the random permutation will give us the sorted array or even sorted array. So, once we from the tree it will be the like this or this. So, the worst case is always quality height, but average case is expected; expected height is  $\log n$ . So, that is the average case analysis. So, in the next class we will discuss how we can guarantee that we have a balanced tree no; no randomization. So, guaranteed balanced tree. So, they are called balanced tree the like red black tree b tree a real tree.

So, we will talk about red black tree in the next class.

Thank you.

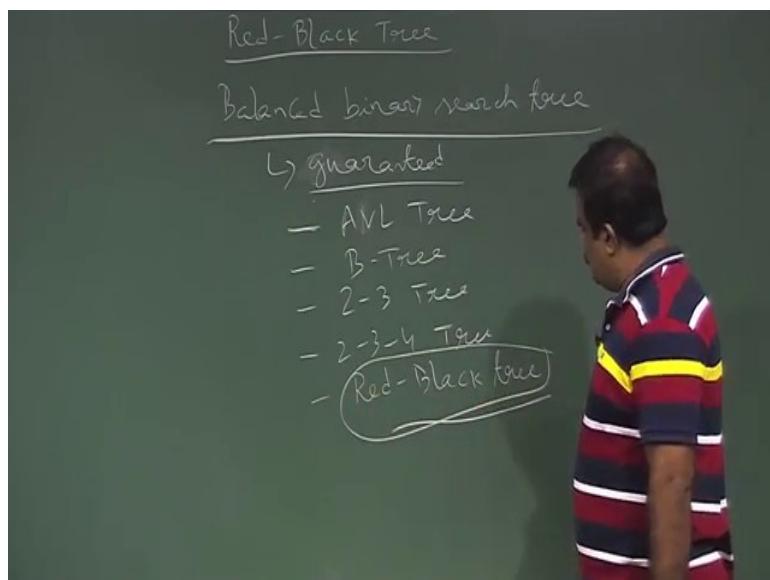
**An Introduction to Algorithms**  
**Prof. Sourav Mukhopadhyay**  
**Department of Mathematics**  
**Indian Institute of Technology, Kharagpur**

**Lecture - 27**  
**Red Black Tree**

So we talked about balanced search tree. So, we have seen that if we have a input and if we can do a random permutation before forming the tree, then after doing the random permutation if you form the tree then that tree will be expected to be balanced, but there is no guarantee because that is the average case analysis, but the worst case it may be quadratic it may not be balanced in the worst case.

So, now we want a guaranteed balanced binary tree. So, that is our class. So, red black tree is one example.

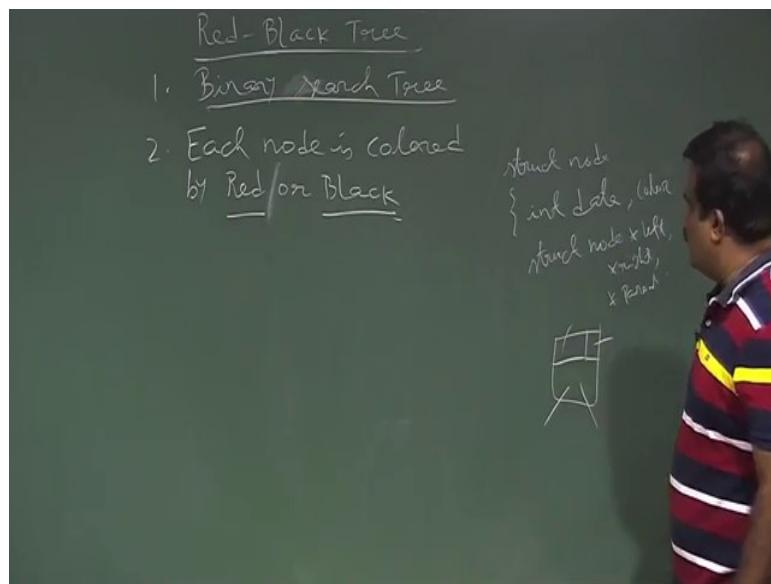
(Refer Slide Time: 00:58)



So, we want balanced search tree binary search tree and this should be guaranteed, no randomly build nothing. So, we want guaranteed binary search tree because we want the balanced tree that is a good tree because we can perform the query in a logarithm time if we have the balanced tree if we can store our data in a tree which is balanced then we can just do the query in a logarithm time or many things we can do in  $\log n$  time. So, they are few example of a balancing and this is the guaranteed.

This is the no randomization this is guaranteed. So, we need guarantee we need the balanced binary search tree. So, there are some example of balanced binary search tree like AVL tree B tree then 2 3 4 tree and same red black tree. So, these are balanced binary search tree. So, we will talk about this red black tree in more details. So, what is the red black tree? So, what is the definition? So, red black tree is a binary search tree with some properties what are those properties.

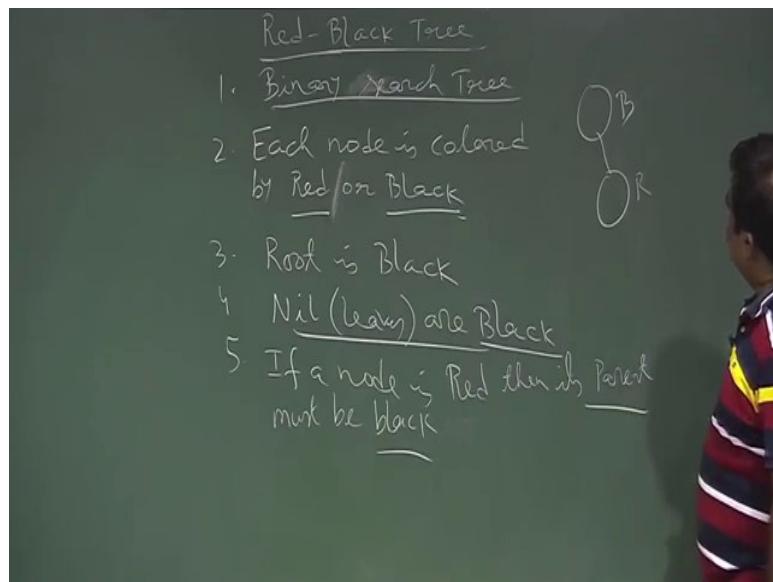
(Refer Slide Time: 02:49)



So, first of all it is a binary search tree.

So, binary search tree means we have that property. So, if you take any node in the tree  $x$  the value of this key value of this must be the all the left subtree has value less than  $x$  all the right subtree has value greater than  $x$ . So, this is binary search tree property. So, this is a binary search tree, that is the first condition. So, now, each node is coloured and coloured by 2 colours, coloured by either red or black. So, we give the colour to each node. So, and we can only use the red colour or black colour. So, for that if you want to implement this. So, usually we define the node like this struct node. And then we have the pointer for the left child and maybe the right child and maybe we need to have a pointer for the parent. So, now, we have a colour bit also. So, we use one bit as a colour either red colour or black colour. So, this is if it is one we can use it that is the convention we can fix by ourself, but each node has to be coloured by 2 colour either red or black. So, this is one property another property is roots are black.

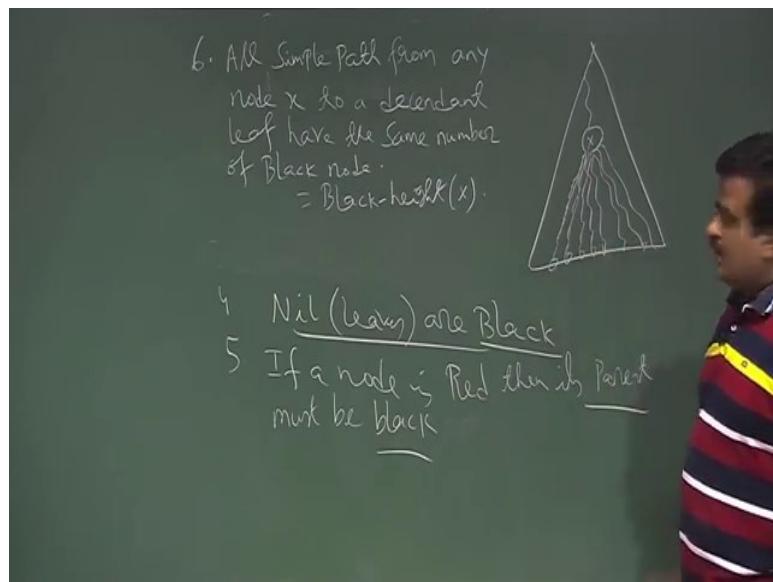
(Refer Slide Time: 04:53)



Root is black and nil leaves nil are basically new leaf will come to that. Nil leaves are black. Nil are basically the leaves. So, we have a tree we have the leaves. Now we introduce some new leafs which are basically nil to make the tree to be complete binary tree because a node may have only one child. So, to make it complete we need to introduce this node; nil. So, nils are black and the next property is if a node is red then the parent must be black.

So, if a node is red if this is the red node, then the parent has to be black the parent cannot be red; so this one property. So, if a node is red then the parent has to be black. And the last property which is very important property which is give us the black height. So, this is the next property.

(Refer Slide Time: 06:40)



So, if you take a node. So, suppose this is our tree, now if you just visit all the path from this node to the leaves so, simple path we are not considering the loop.

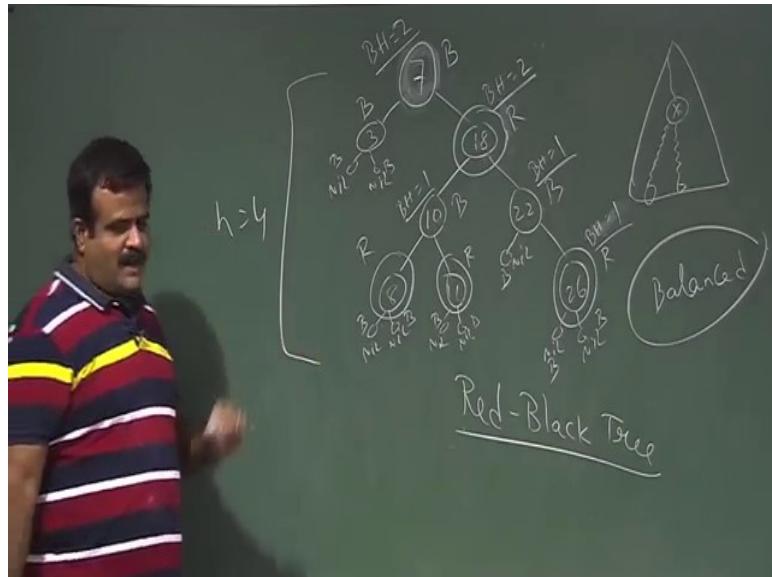
Now for each part if we count the number of black node. So, this is a path from this node to the leaf this is another path from this loop to the leaf it is ancestor. So, if you just count number of black node in the path. So, each path should contain same number of black node and that number is referred as a black height of this node. So, let us write this property all simple path from any node  $x$  to a.

Descendant leaf has same number of black node have the same number of black node and that number is referred as a black height of  $x$ . So, if you take any node in the tree and if you consider a path from this node to a leaf, and we count the number of black node if and then we take another path from this node to a leaf, and the same number of black node must be same if there is four black node in this path, there has to be four black node in this path there has to be four black node in the path and that why it is the unique.

So, this is called black height of that node. So, this is just; number of black node in the path from that node to the leaf; it is called black height of this node. So, if the 6 properties are satisfied, then we call the tree is a red black tree and we will see because of this properties this tree will be a balanced tree guaranteed. There is no randomly permutation nothing this is guaranteed, this is a this if our tree is satisfying this property then the height will be  $\log n$  that we will prove. Now let us take an example of a red black tree let us give an example of a red

black tree, this is the definition of red black tree these are the property our tree must satisfy. Then the tree is called a red black tree.

(Refer Slide Time: 10:08)



So, suppose this is our nodes here 7 say 3 then 18, 10, 22, 8, 11 and we have 26. Suppose given these are the numbers, and this is a binary search tree if you look at any node 10 18. So, if you look at 18 all the subtree rooted here are less than eighteen also subtree rooted here greater than 18. So, this is a binary search tree. Now we want to make it a complete tree. So, we will introduce nil nil are basically new leaves.

So, we will put nils because this is having one node. So, we introduce the nil. So, from each leaf node or form each node which is having one child or no child we will put the nil and these our new leafs. So, this we nils we introduced. So, now, this is the nils and now we need to give the colour of this tree now we give this root to be black, because that is one of the property root is black. So, this is 7. So, root is black and these nil are also black this is one of the property we have nils are black.

Now we need to give the colour of this nodes so, that we must satisfy that six properties i mean those properties. So, which colour we give to this node. So, we can give to this node black and we can give to this node red, and maybe this node red, this node red this is black this is also black and this is red. This is we can try we just suppose we assign this colour each node now this is a tree binary search tree and each node is coloured and each node is coloured by red or black. Now root is black leaves are black. So, that is the one of the properties now if

a node is red we know parent has to be black. So, this is red node. So its parent is black this is a red node parent is black this is a red node parent is black. So, that is the property is also satisfying if a node is red that the parents are black. So, now we have to see the final property which is called our black height. So, if we look at all the nodes. So, if you look at this nil, nil black height is basically zero now if you look at this node. So, this node what is the black height. So, if you look at this node so this node if you form this node to a leaf how many black node only one this node to a leaf. So, black height of this node is basically BH. Black height of this node is one now what is the black height of this node; if you take this path. So, there is one if you. So, that black height in the black height we must exclude the colour of this x.

So, let us just recall; black height is basically we take x and we consider all the nodes black nodes in this path and if excluding the colour of x if the colour of x is black we are not counting that in because excluding the colour of x. So, it is that that has to be noted we are not counting the colour of x if the colour is black we are not adding that we are excluding the colour of x. So, that is; that means, black height of this is basically also one because it has if you check it now black height of this node. Now this consider this path from this to this how many two black node 1 2. Consider this path this to this 1 2 consider this path 1 2, consider this path 1 2, consider this path 1 2.

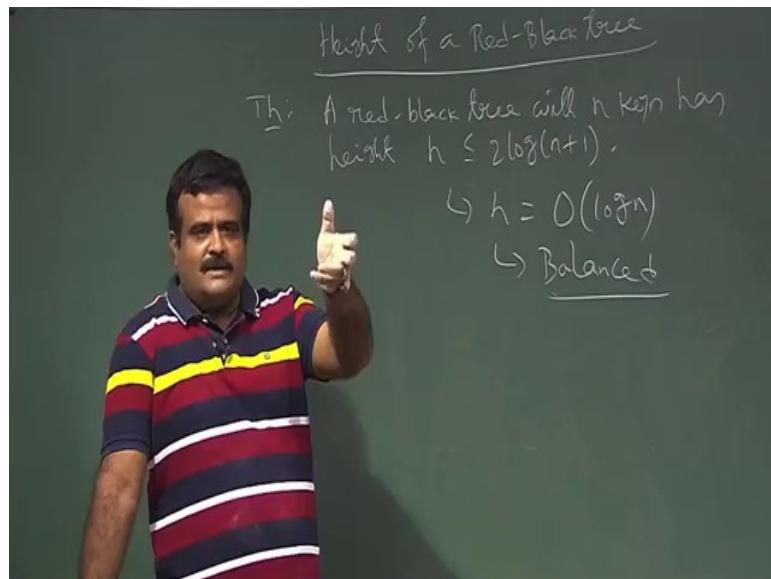
Consider this path. So, from this node if we look at all the path simple path from this node to the leaf all are having same number of black node and which is basically two. So, the black height of this node is basically 2 and this black height of this node is now what? Now black now black height of this node is also consider this path it has one considered this path this is one considered this is point because we are not counting the black of this because that is excluded. Now what is the black height of this node? So, if you take a path from this. So, this is one two one two if you take path; so from this node to this leaf if you take the path.

So, we are not counting this black; so 1 2. So, we can just check the black height of this node is basically 2. So, this is the red black tree because it is satisfying the all the properties we have listed. So, every node is coloured this is a binary search tree every node is coloured by either red or black, and if a node is and the roots and nil are black and if an node is red then the parent is black and the from a given node from a node x if we consider the path from x to a leaf, and all the paths all such path having same number of black node excluding the colour

of  $x$  and that is called black height. So, we have that properties also satisfy. So, this is a black tree.

And this is balanced what is the height of this tree height of this tree is 4 and how many elements are there? 1, 2, 3, 4, 5, 6, 7, 8. So, but this is a eight we in general we have to prove that this is the balanced tree. So, this tree is balanced, but we need to prove why this is balanced for  $n$  nodes, but this is one example of a red black tree now let us prove why this tree is balanced by a theorem. So, this theorem is telling. So, height of red black tree we will come to this example again and we will prove this theorem. So, let us talk about height of a red black tree.

(Refer Slide Time: 17:59)

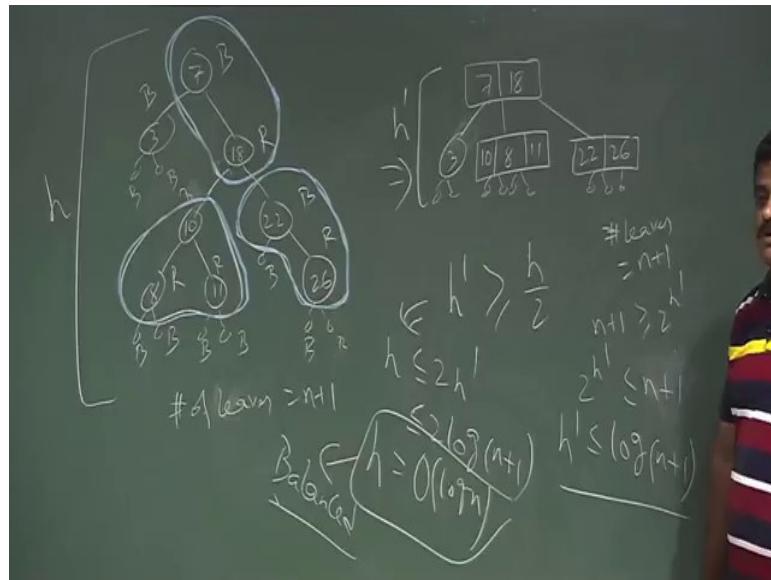


Let us establish that this is a balanced tree height is  $\log n$  height of a. So, this is we are proving by this theorem. So, this theorem is telling a red black tree with  $n$  nodes  $n$  keys has height  $h$  which is less than  $\log 2 \log n + 1$ . So, this is the theorem and this is telling us height is. So, this means height is order of  $\log n$ .

So, this is telling us balanced. So, red black tree balanced if you can prove this. So, how to prove this? So, to prove this we will use some intuition, that is we want to remove the red nodes. Now the question is how we can remove a red node from the tree. So, to remove the red nodes we know if a node is red then the parent is black. So, basically what we can do we can merge the red node to its parent because is black. So, that is the one way you can remove that red node. So, that we are going to do. So, how we can do that? So, let us just take that

same example. So, to prove this will use that same example and we will see how we can remove the red nodes. So, basically we merge the red node use its parent because we know if a node is red the parent is black.

(Refer Slide Time: 20:06)



So, let us draw inside. So, 7, 3, 18, 10, 22, 8, 11 we have 26 and then we have the nil and the nil are black and this is red this is black this is also black nil are black and this is basically red and these two are red this is red this is black this is black. So, this was our example of a red black tree.

Now we want to remove the red node. So, to remove what we do we know if node is red its parent is black. So, we merge a red node with its parent because we know parent is black. So, if you do that, this is our original tree, if you do that then what will be the situation. So, this is red. So, this we are going to merge with this, this is red. So, this we are going to merge with the parent and this two is red. So, this we are going to merge with the parent. So, this is the merging we are going to do so.

So, this is red node this we are going to merge with the parent every red node is merged with the parent. Now if you do that then we will have this tree combined into this tree. So, this 7 and 18 is merged and it has one child tree and we have another child which is basically having 3 nodes because this three node has merged. So, 10 8 11 and we have another 22, 26 and then the nil are there. So, this will having 4 nils and this will have 3 nils like this.

So, this tree will convert into this tree and this is nothing, but 2, 3, 4 tree why. So, what is the number of leaves in this tree for each node? So, number of leaf is either 2. So, the number of leaves is either 2 or 3 or 4. So, that is the reason it is called 2 3 4 tree and these tree is balanced with this tree is very nice structure. So, this tree is converted into this tree. Now we want to talk about height of this two tree. So, what is the height of this tree suppose height of this tree is  $h$  and height of this tree is say  $h$  prime. So, suppose height of this tree is  $h$  and height of this tree is  $h$  prime. Now, what we can say about relationship between  $h$  and  $h$  prime.

So, can you say that can you say  $h$  prime is greater than equal to  $h/2$  because. So, we are merging the red node. So, if we consider a path which is giving us the height. So, if you consider a path now when it will be equal, when there are it is merging with half. So, when there are equal numbers in that path when there are equal number of red node and black node then it will be equal, but we know if a node is red then the parent has to be black. So, if you consider any path then the number of red node will be less than the number of black node because that is one of the property if a node is red then the parent has to be black. So, number of red node is less than the number of black node.

So, for these to be less than we need to have one red node more which is not possible because consider a path if a node is red than the parent has to be black. So, in any path the number of red node will be less than number of black node. So, that is the reason this height is the new height will be greater than  $h$  by 2 and it will be equal when the exactly same number of black and red node is there in the tree. So, this is the relationship we have. So, this is the height of this tree this is the height of this tree. So, now, what is the number of leaves over here in this tree? So, number of leaf is basically  $n + 1$  and this is same as here also because we are not merging the nil because nil are black.

So, number of leaf will be same number of leafs will be same  $n + 1$ . Now this is having each node is having either 2 child or 3 child or 4 child; now two is the minimum number of child. So, from here we can say that  $n + 1$  is greater than two to the power  $h$  prime because if every node has 2 to the power if every node has 2 child then two to the power  $h$  prime is the minimum number of leaf it should have. So,  $n + 1$  must be greater than  $2^{h \text{ prime}}$ . So,  $2^h$  or  $n + 1$  is less than  $4^{h \text{ prime}}$ , but we do not need this. We'll use this one. So, basically from here what we can say  $h$  is less than  $2 h$  prime.

So,  $2^{(h \text{ prime})}$  so; that means. So, from here  $2^{(h \text{ prime})}$  is less than equal to  $n + 1$ . So,  $h$  prime is less than equal to  $\log n + 1$ . So, these we are going to use here. So,  $h$  is from here what we can say  $h$  is less than equal to  $2^{h \text{ prime}}$ , which is basically less than equal to  $2 \log n + 1$ . So, a height of this tree is basically  $\log n$ ; so balanced. So, a red black tree is balanced. So, this is guaranteed there is no randomness, nothing. So, this is guaranteed balanced tree. So, this is good news. So, if we can have such a tree balance tree then we can perform our query search in a height time and height is  $\log n$ . So, the search or any other query can be handled in  $\log n$  time.

So, in the next class we will talk about the modifying operation how we can insert. So, this is a dynamic set how we can insert an element in a red black tree. So, modifying operation that we will discuss in the next class.

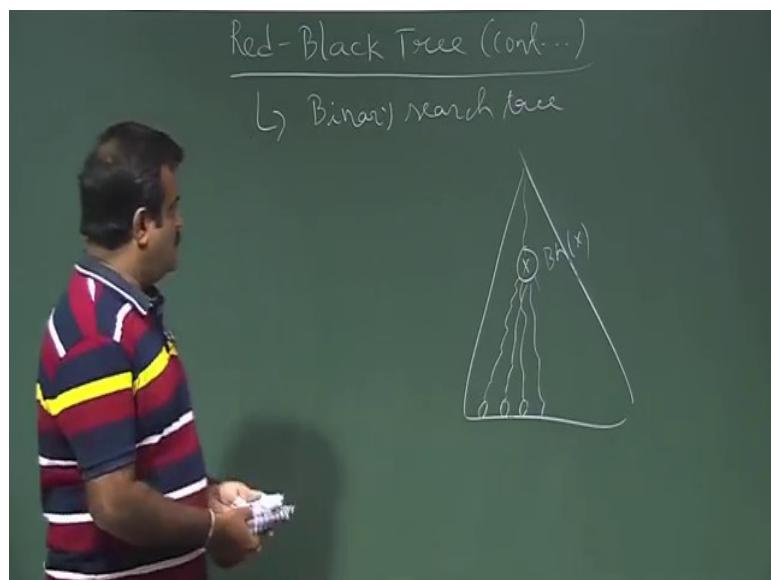
Thank you.

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

**Lecture - 28**  
**Red Black Tree (Cont.)**

So we are talking about balance binary search tree and we have seen the red black tree; one of the example of balance binary search tree guaranteed balance binary search tree, so basically red black tree today we will talk about the red black tree insertion and the modifying operation in the red black tree how we can insert a node in a red black tree, I mean.

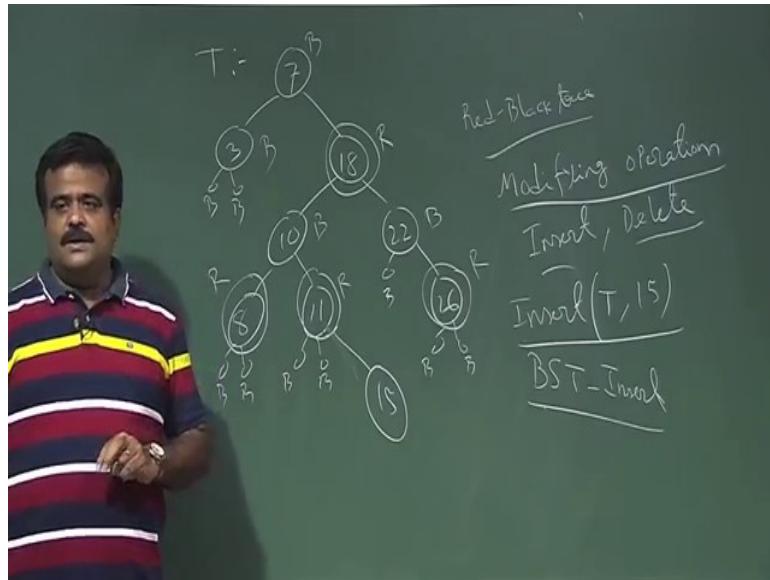
(Refer Slide Time: 00:49)



So, basically to recap red black tree is a binary search tree with some properties and there are properties means every node is colored either red or black and the root is black and the nils are basically the new leafs they are black. And another property is if a node is a red the parent must be black and the most important property is the black height. So, if you take a node, suppose this is the tree if you take a node X and if we consider all the path from that node to leaves then if you count the number of black nodes then the number of black nodes are same and that is basically black height of X and that exclude the color of X, if the color of X is black then we will not count that black we not count that. So, excluding the color of X we just count number of black node in the path from X to any leaf and that has to be same. So, that is one of the property for this binary search tree to be red black tree.

So, if all these properties satisfied then we have called our BST or binary search tree is a red black tree and we have proved that in the last class that height of the red black tree is balanced, I mean it is  $\log n$  if there are  $n$  nodes then the height is order of  $\log n$ . So, today we will discuss some modifying operation in red black tree and those are required for insert a; inserting a node in a red black tree.

(Refer Slide Time: 02:49)



And one of the examples we had is like this. So, we have 7 3 18 10 22 8 11 26 and we have the nils the new leafs to make this tree is complete binary tree and then we need to color it. So, this root has to be black and the nils are black, so these are all blacks. This are all black node and then we have to color it. So, so if you count the black height of this from this to this node 2.

So, now, we considered some modifying operation in red black tree, modifying operations like insertion: we want to insert a node; deletion: we want to delete a node. So, this set is dynamic set. So, in this set anybody can join and anybody can leave suppose we have a database and every record is having this key and based on this key we make this red black tree and the database set is dynamic. So, in that case we have to insert a node any record can get deleted: in that case we have to delete a node.

So, what is the advantage of this type of balanced tree? Advantage is suppose we maintain this red black tree for our database now if you want to search a node. So, searching will just take the height of this tree because if we search some node. So, we check with the root if it is

greater than we will go to the right part of the tree if it is less than we will go to the left part of the tree, so like this we will just make the comparison up to the height of this tree and height is  $\log n$ . So, the search will take  $\log n$  time. So, this is the major advantage searching, finding, minimum, maximum, successor, predecessor we talk about those. So, those will take the height time when height is  $\log n$  that is the major advantage.

So, let us talk about how we can modify such operation. So, deletion we will not cover in this class. So, for insertion what we do? We first insert a node in this tree by using the binary search tree insert like if you want to insert say 15 here, so this is our tree now suppose we want to insert, insert in this tree 15. So, what we do? We first do the binary search tree insert. So, how to do the binary search tree insert we will just compare 15 with 7. So, we will go to the right part it is greater than right then again we will compare 15 with 18 we will go to the, so it is less. So, we will go to the left part again 15 with 10 right again 15 is greater than. So, we will put this 15 here now 15 is inserted by the BST insert binary search tree insert. So, we have inserted this new node, now this is the red black tree, so every node has to be colored. So, we want to give the color of this node we want to we have to give the color on this node new node.

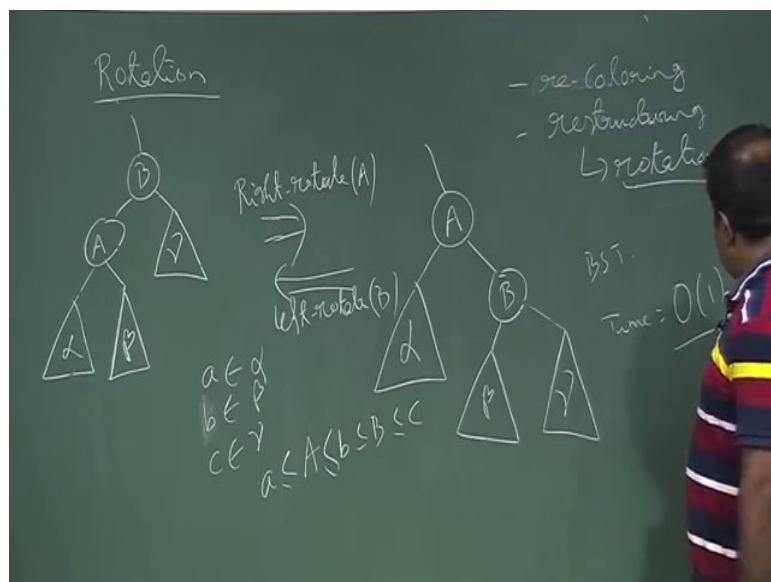
So, which color we should give - we have two option red or black. Now the issues is if we give a black color then there is a danger because if we give a black color and if we go to a upper node which is having two branch then the black height is hampering I mean then for that node we have the black height is not same in the both side. So, giving black color is that way it is not good idea. So, instead of that if we have to give the if afford to give the red color in that case if the parent is red we give the red color for this new node now if the parent is black of this new inserted a node then there is no problem because we have inserted node which is red. So, everywhere the black height is preserving. So, only issue may be its parent may be red, now if the parent is black there is no problem still the tree is red black tree.

Now the issue is if the parent is red like in this example. So, so we give this color red now the parent is also red. So, we have a problem over here. So, we need to fix this problem. So, if this is black then we have no issue we can just state away stop here insertion I mean insertion positive stop if this is black because now this got red so we have a trouble here. So, we need to fix that, but we have to give red here because otherwise this red black height will be hampering for the upper node which have another branches. So, we give the red color and if

we give the red color then there is a problem with one of the property that is if a node is red the parent must be black.

Now if parent is black then you are done, but here parent is red so you have to do something. So, to fix that what we can do we can just try to recolor it. So, to fix that we can do this operation like recoloring like we will try to see whether by recoloring we can solve the problem.

(Refer Slide Time: 10:32)



Now, sometimes it is not possible to solve this problem just by recoloring. So, now, in that case what we need to do we need to restructure the tree and restructuring means we need to perform some rotation operation this is called restructuring, restructuring means we need to perform the rotation operation. So, there are basically two types of rotation operation we will perform. So, we will talk about that. So, first, so let us talk about rotation operation. So, rotation means suppose we have a tree like this we have a node B we have a node A and then we have a sub tree rotate here which is denoted by gamma and we have another sub tree alpha we have beta.

So, now, this is suppose this is a part of the tree. So, we have other portion. So, this is a node B this is a node A and this is left sub tree rooted at a right sub tree rooted at A and this a right sub tree rooted at A and this are two nodes. Now how to perform rotations? So, this is basically rotation on A. So, we want to have A up B down so that means, this side rotation this is right rotation on A. So, right rotate right rotate on A we can denote. So, this means

want A up and we want B to be down and we want this. So, this is the alpha, this is beta and this is gamma.

So, this is basically restructuring the tree. So, this is the rotation operation we are performing. So, we make A up, so the left sub tree with a was alpha still it is with A and now A has a right sub tree beta the node beta now the beta B and now beta beta is the representative of the any node in the, it was the right sub tree of A now it is a left subtree a now it a left sub tree of B and gamma is still the any node in the right sub tree of B. Now if you do this still we are preserving the binary search tree property so that means, if we take A belongs to alpha B belongs to beta and C belongs to gamma then still we have less than equal to A, less than equal to B, less than equal to C sorry this is B. So, this is the binary search tree.

So, any node alpha is a representative. So, any node in alpha must be a which is less than less than A any node beta must be greater than A, but less than B still it is preserving here also. So, this is still a BST and the color also will be same, so this is the rotation operation we are performing. And now how much time it will take to perform this, and the other way round suppose we have this structure we want to make B of A down then this will be the left rotation left rotate on B. So, we want B up A down. So, if this is our portion of the tree and we want to make like this. So, we have to perform left rotate on B. So, this is the rotation operation.

So, how much time it will take to perform this rotation? If basically what we are doing we are changing some pointer. So, so parent of parent of B is not parent of A and right child of A is now B. So, just we are changing three four pointers. So, it will take time is basically order of 1, order of 1 to perform this rotation operation because we have just changing the few pointers. So, this is the rotation operation and we may need to perform this rotation operation in order to perform this modifying operation like insert deletion. Let us go back to the insertion 15 in that sub tree, in the tree so let us just draw this picture again.

(Refer Slide Time: 16:12)



So, this is 7 3 18 10 22 8 11 26 and we have inserted 15 so we have inserted 15 and we have the nils over here, the nils over here and let us give the color also this is black this is black, this is black, this is black, now this is red, this is red and this is also red now we just make this is to be red.

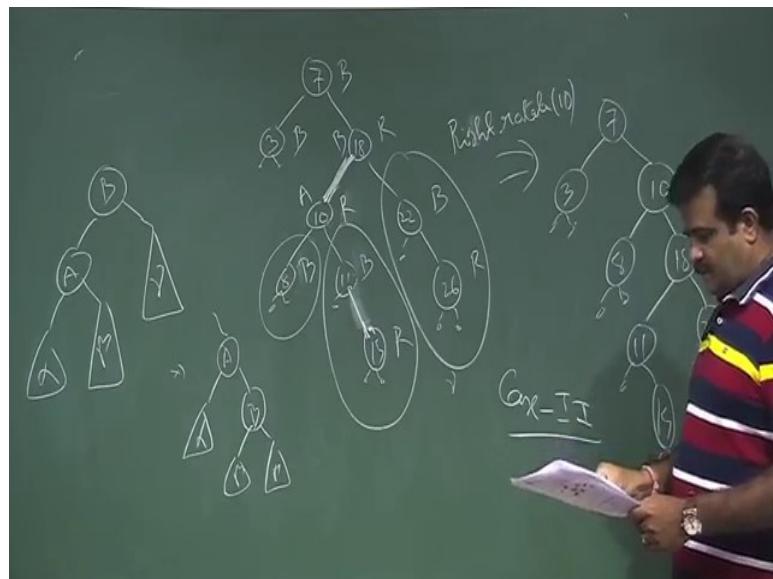
So, now there is a problem because if a node is red the parent has to be black, but here the node is red parent is also red. So, we have to fix that now we first we try to fix using the recoloring. So, now, this is what is refer as case I, this situation case I is basically, if we look at here parent is red. Now parent that grandfather or grandmother of this node is black and this is the aunty or uncle aunty or uncle is red. So, this situation is called case I. So, if the aunty or uncle is same color with the parent and grandfather or grandmother is black then what we do we just change the color of this. So, we will make this to this is black and this is red and we have this is red, this is black this is red now we solve this problem. So, this is called case I.

Now we solve this problem, but we still have problem over here. So, it propagate this problem up, but this is which is refer as case I. So, case one means like this. So, if we have a situation like this. So, if we have a node A and then we have a node B, there is some tree hanging over here, some tree hanging over here and we have a node D some tree hanging over here some tree hanging over here. So, if this is X, now if this is, so this is red and this is red, this is red, this is this side we have taking example of other side and this is black. Now

this is refer at case I; case I means the parent is red aunty or uncle is red, but the grandparent is black. So, in that case what we do - we change the color of these two we make it red and we make it black now it will solve the problem locally, but problem is if this still red, then we have a problem the situation like here. So, problem still remain because this node is red and the parent is also red.

So, now, we will see whether still we are in case I or not. So, here parent is red grandparent is black, but the aunty or uncle is not red aunty or uncle is different color black. So, this is not case I. So, we have to do we have to perform case II. So, case II means we need to perform the rotation operation. So, we want to make this up and 18 down.

(Refer Slide Time: 20:23)

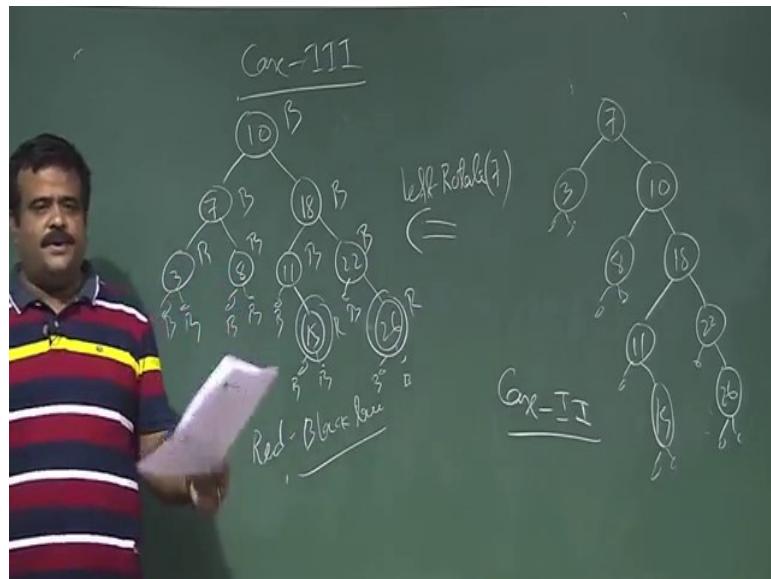


So, what we do, this is A node this is B node if we recall. So, if we recall this is B node this is A node and we have a sub tree rotate over here alpha beta and here we have gamma then if we perform this operation. So, this is basically A node, so we will just do the right rotate on 10. So, we want to make 10 up. So, in that case this is alpha, this is beta and this is gamma. So, after this operations situation will be like this A will be up, B will be down, and alpha will be still hanging over here and beta will be here and gamma will be here like this. So, let us write this after this rotation. So, 7 will be here. So, this will be unchanged now this we want 10 and here we have 18 come down and here alpha will be remain same so 8 and then we have this leaf nils and from this 18 we have this is basically beta. So, beta is this 11 and 15 and we have still gamma is with 18, so 22 and 26 and we have the nils over there.

So, this is the situation if you rotate this on 11 if you 10. So, if you want to make ten up 18 down. So, it will be like this. Now this is called case II. Now if you have to do case II then we have to perform the case III. So, case III means now we want to make this down this up. So, we want to make left rotate on 7 to make 7 down 10 up, so that we will perform now.

So, now, we perform case III, case III means which should be followed by the case II.

(Refer Slide Time: 23:09)

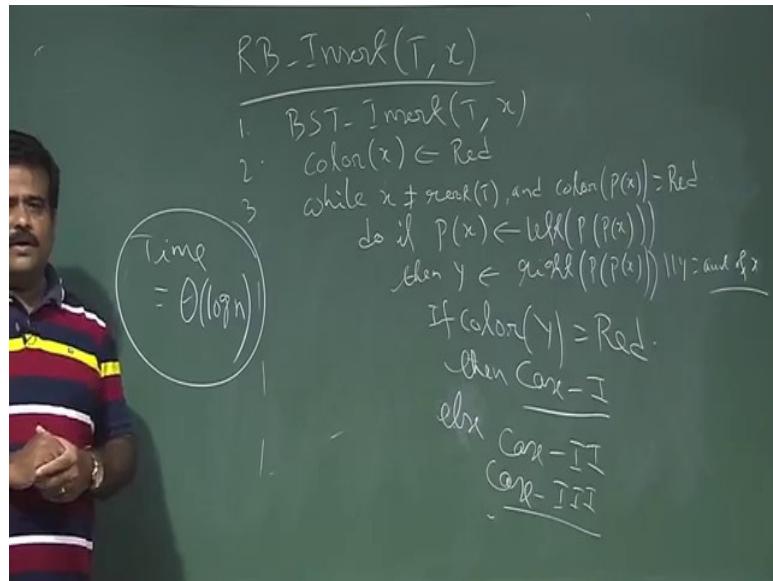


So, here we want to perform left rotate on 7. So, if you do the left rotate on 7. So, 7 will come down 10 will come up, so the situation will be like this. So, 10 will come up 7 will be down and then it is 18 and 3 8, just the left rotate we have seen the operation of left rotate 22 and then we have 15 over here, 26 and this are the nils basically.

Now we do the coloring this is black, the nils are all blacks and so this sorry this is black nils are all black. So, this is basically a red black tree and this is basically operation is case III, if you have to perform case II then case III will be followed by the case II. So, this is a red black tree.

So, this is the insertion operation on a red black tree. So, if we have a red black tree if we want to insert a node then we have to perform this operations. So, let us just write the pseudo code for this, let us just quickly write the pseudo code for this.

(Refer Slide Time: 25:26)



So, red black tree insert, we have tree and we want to insert a node  $x$  in the example it was 15 the key values is 15. So, first of all we have to do the BST insert then we have to color the node. So, if affordable to give the red color then give red. So, if we give the red color it may violate only one property the parent is also red. So, if that is happening, so while  $X$  is not root if it is not the only node and the color of parent of  $x$  is red then we have a problem then we have to perform this.

Now case I means if the aunty or uncle is red and the grand grandparent is black. So, that is the case I. So, if  $p_x$  is the if the parent is the left child of the grandparent  $p_p$  of  $x$  then  $y$  is the other child right child; that means,  $y$  is the aunty or uncle of  $x$ ,  $y$  is the aunty or uncle of  $x$ . And now if the color of  $y$  is red then it is case I and then if its then we perform the else if it is not if it is not case I else we have to do case II and followed by case III anyway this is the pseudo code for this insertion. So, how long time it will take. So, basically time complexity is order of  $\log n$  because we have just inserting a node and we have just performing the operation and we have maybe we are doing rotation this recoloring will take constant time and rotation also take constant time. So, we have just rotating and we are just going to the up of the tree. So, basically we are going to the height of the tree and this is already balance tree, so height is  $\log n$ . So, it is this insertion will take  $\log n$  time.

So, similarly deletion also if you want to delete a node from a red black tree, so we deletion is not we are covering in this course, but if you are interest you can read the text book deletion

is little TDS we have to find the success array and then. So, first of all we have to do the BTS deletion binary search tree deletion. So, that is also we will take  $\log n$  time because that is basically height of the tree. So, red black tree if you want to insert a node then again to make the red black tree we need to spend  $\log n$  time, if you want to delete a node after deletion again if we want to make this set red black tree it will take  $\log n$  time, so deletion and every; so deletion and insertion will take  $\log n$  time.

Thank you.

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

**Lecture - 29**  
**Augmentation of Data Structure**

So we talk about augmentation of the data structure. So, the problem is suppose we have a problem, now we need a data structure for that problem.

So, instead of being a completely new data structure from the sketch, what we can do, we can make use of the old data structure which you know and we can do some augmentation there. We can have some extra field of information for our problem. And it will help us to solve our problem so that is called augmenting of data structure. So, we will take 2 example for this problem. First one is dynamic order statistics.

(Refer Slide Time: 01:08)

So, we have seen the order statistics problem. Order statistics problem means, we are given a array of  $N$  numbers, and we need to find out  $i$ -th smallest element. So, this is the problem of order statistic we need to find out the element whose rank is  $i$ .

So, rank of an element means, rank is the position of that element in a sorted array. So, if you sort it the position of this element. So, this is the problem of order statistics. And we have seen some algorithm to solve this problem like SELECT then the randomized

version of the SELECT. And then we look for a good pivot so that in worst case it will be guaranteed linear time.

So now, if this set is dynamic set so that is called dynamic order statistics. Suppose we have a set  $s$ , which is dynamic. So; it is changing any time the anybody can join. So, dynamic set means insertion and the deletion is allowed. So, these 2 operation make a set dynamic. So, we are allowing anybody to leave from this set at any point of time that is deletion. And we are allowing the join a num join a element in this set. So, this set is dynamic. Now suppose we have a dynamic set and the problem is to find the  $i$ -th smallest element in that set.

So, for this problem we need to have a data structure. So, we need to have data structure for this dynamic order statistics. So, for this we will just do the augmentation of the data structure, like we know we have a good data structure like red black tree which is balanced. So, we will use the underlying data structure as red back tree, and then we will do some augmentation there like we will keep some more information in the data structure. And that may help our problem. So that is called data structure augmentation.

(Refer Slide Time: 05:08)

So, basically idea is to use the red black tree because it is a balance data structure, balance tree red black tree for the set  $s$ , because  $s$  set is dynamic. So, in the red black tree we can also perform the dynamic operation modifying operation like insertion deletion.

And we have to do some augmentation, but keep the sub tree size in the node in the nodes. So, this is called this is the augmentation of the data structure.

So, we will just use our red black tree data structure. So, we will make the red black tree. But we will keep this extra bit of information, extra information so that will help us for solving our problem. So that means, each node is having this field this is the key and this is the size of the tree, size of the sub tree rooted at that node. And this is called augmentation. So, in red black tree we have key based on this key we perform this and with the key each node we are having this we are keeping this information also, for this will help us for our problem our red black our dynamic order statistics problem.

(Refer Slide Time: 07:19)

So, let us take an example. So, this is called order statistics tree we can say. So, this is the example of OS tree. This is basically red black tree, suppose these are the nodes. So, C P A F N Q D H.

So, then we have the leaf nodes to make this So, this is our red black tree. And these are the nodes these are the key. So, C is less than M. So, this M N, ABCD are in alphabetical order. So, the order they are coming in the alphabet here there are I mean, A is less than C in the alphabetical order. So that is why A is in the left side of the C. So, this is a, this is a red black tree we can make the color of it also. So now, we need to put this extra bit of information that is the So, this is the key value and this is the size. So, size of the tree rooted at this.

(Refer Slide Time: 09:53)

So, size of a node is basically size of the left plus size of the right plus 1. So, size of x is basically size of left of x plus size of right of x plus 1.

So, size of left of x is 5; that means, size of the left sub tree is 5 for this node size of the right sub tree is 3 and the node itself. So, this is the formula for making the size. So, we can easily get the size; now suppose we maintain this size, now how it will help us to find the i-th smallest element? So we want to perform OS select.

So, suppose this is the red black tree and this is the x. So now, we want to find the i-th smallest element that is called OS select i. So, order statistics, but here the dynamic on the dynamic set; so i-th smallest element.

(Refer Slide Time: 11:42)

So, what we do So, basically we start with this if you write  $k$  is equal to size of left of  $x$  plus 1, just if we do  $k$  is equal to size of left of  $x$  plus 1, can you tell me what is the meaning of this  $k$ ? So, basically before that let us take the root as  $a$ .

(Refer Slide Time: 12:16)

So, OS select  $t$ , i. So, we just put  $x$  is equal to root of  $t$  and then we take  $k$  is equal to size of left of  $x$  plus 1. So, what is this  $k$  means? So,  $k$  is basically denoting the rank of  $x$  ok.

Why  $k$  is rank of  $x$ ? Because see I mean this is the red black tree, now red black tree is a binary search tree. So, if it is a binary search tree then the inorder traversal will give us

the sorted array. If you recall, in the BST sort what we are doing we are forming the tree. So, if we perform the inorder tree walk that will give us a sorted array, I mean inorder tree walk basically it is the left subtree then root then the right subtree. So that means, size of the left. So, after the left subtree the root will be printed. So that means the rank of that node.

So, we'll come after the printing all the all left sub tree node. So that is the size of the left subtree. So that means, position of  $x$  is basically in the sorted array is basically size of all nodes I mean size of the left subtree plus 1. So, this means  $k$  is basically rank of  $x$ . Basically, this is coming from inorder tree walk. Now we got  $k$ . So, suppose we are looking for  $k$ -th smallest element. So, if  $i$  is equal to  $k$  then we are happy we return  $x$ , otherwise if you are not So happy I mean if we are just get the  $k$  which you are looking for  $k$ -th smallest element, that is the root itself.

So that means, we know this root is the say  $k$ -th smallest element and we have. So, this is the 7 th smallest element say this is 5. So, this is the 6 smallest element, suppose we are looking for third smallest element, then  $i$  is less than  $k$ . So, what we have to do? We have to look at the left part of the tree. So, then if  $i$  is less than  $k$  then we just look at, then we call OS select with left of  $x$  comma  $i$ , otherwise if  $i$  is greater than  $k$ , if  $i$  is greater than  $k$  then we have to.

Look at the right part of the tree then we have to call OS select again right of  $x$ , but we have already seen the  $k$ -th smallest; so  $i - k$ -th smallest in the right part of the tree. So that is the OS select now let us take an example, suppose we have for this case, suppose we all looking for say fifth smallest element in this example. Suppose we are looking for fifth smallest element we have a Q over here. So, this is the complete tree I mean, now suppose this is our set  $s$  and we have this structure we have the this data structure which is basically a red black tree, but we did some augmentation there by adding this extra field that is the size we are keeping the size of the tree. Now suppose we are looking for fifth smallest element. So, we just execute this code OS select.

So, we come here this is  $x$ . So, what is the  $k$  value  $k$  is basically six, but we are looking for fifth smallest element. So  $k$  which is not the case over here So that means, we have to go further. Now  $i$  is less than six. So, we have to go for the left part of the tree, and this is

our  $x$  now. And here also you are looking for fifth smallest element with this, now what is the rank of this? Rank of this is basically  $k = \text{left subtree} + 1 = 2$ .

So now,  $i$  is greater than this So, you have to go for the right part of the tree and this is our  $x$ . But now we have already seen the second smallest. So, we are looking of fifth smallest now our  $i$  is basically this  $i - k$ . So, third smallest element in this subtree we are looking for. Now what is the  $k$  over here; what is the rank of this node? Rank of this node is again 2. So now,  $i$  is greater than so, we have to go for right part of the tree with  $x$  is this, but with  $i$  values is basically 1. So, this is our So, what is the  $k$ ?  $K$  is basically 1. So, we are getting. So,  $H$  is basically fifth smallest element is basically  $H$ . So,  $H$  we return. So, this code will return basically  $H$ .

So, this is the OS select. So, this set is dynamic; that means, we can insert a node over here, suppose we can in insert a node over here we can delete a node over here. And So, what is the running time of this code? Running time is basically order of  $\log N$  because, we are just going to the height of this tree, when you are searching.

(Refer Slide Time: 19:35)

So, that height is this is the balance tree, so that is why time for OS select is  $\log N$ . So that is the advantage of having the balance data structure. So, this is the balance tree and So, the height is  $H$  and we are executing this we are going at most the height comparison. So that is why it is order of  $n$ . So now, the question is how we can perform this insertion and deletion operation, and now the.

So, how to perform insert and delete operation here? So, this is the dynamic order of statistics. So, suppose we want to insert a node say which is basically k, suppose we want to insert a node k we want to insert k.

(Refer Slide Time: 20:45)

Now once you insert k we need to change this field like. So, this is now became 1, and this will become 2 now, and this will become now 4, and this will become 6, and this will become 10. Now all the path it will be changing. So, this operation, and not only that then again we have to make this tree as a red black tree. So, for that we need to do the red black tree insert operation. So, in that case we need to maybe perform the rotation operation. So, we have to see in the rotation operation how we can handle the rotation operation, whether it is easy to handle basically we need to handle this extra information. So, if we can handle this extra information in a constant time or in the same time with the red black tree rotation, then there is no issue ok.

So now, we will talk about how to handle this rotation operation for a this extra information.

(Refer Slide Time: 22:19)

So, suppose we have, we have a tree like, this E then we have C, and we have a tree hanging over here with 4, 4 7 3. Now we want C up E down. So, this is basically right rotate, right rotate on C. So, C will be up and E will be down sorry, C will be up E will be down and the left part. So, this 7 will be hanging here and 3 will be hanging here, 4 will be hanging here. Now how to make these changes? Now this is basically this plus this 11, and this is basically 15 and this 16. Now here also we can just change this. So, this is 3 plus. So, this is 8 and this is also 16. So, this is also similarly I mean 2 to maintain this extra bit of information, we are not really using we are just using constant time. Because just we are change because these are not changing these are alpha beta gamma, alpha beta gamma. So, these are not changing we are just changing these bits. So, these bits is or can be maintained in constant time. So, if this is maintained in constant time then insertion or deletion can be done in logarithm time.

So, this is most important. So, this dynamic set we can preserve this set again this red black tree by logarithm time that is important. So now, the question is instead of storing the subset size. So, our augmentation is we have the key value; here we are storing the subset size.

(Refer Slide Time: 24:32)

Now, if we just store the rank itself, if we do this type of augmentation, then what is the issue? If we can store the rank then we can just look at the this looks like easy to search the  $i$ -th smallest element basically, we need to search the element whose rank is  $i$ , but problem with showing the rank is, now suppose we insert a node which is a minimum then all other node which we have had then we need to change all the elements, the rank will change because we have inserted minimum or maximum element, then the rank of each node will change. So in that case it will not take height time; it will take the order of  $N$  times.

(Refer Slide Time: 25:38)

So, the keeping rank is not a good idea. So, let us just write the methodology for this data structure augmentation. In the data structure augmentation basically we have seen that we have to choose a data structure and in that data structure we have to do some augmentation. So, basically we have to choose a underlying data structure. So, here we are taking the red black tree in our example. and then based on our problem we have to decide the augmentation things; that mean, we have to determine the additional information. For our case for our OS select we stored the subset size, sub tree size. We did not store the rank of a node, because it will not help us. Then we have to verify the modifying operation, verify how we can maintain this additional information.

This additional information can be maintained for modifying operation, maintaining the same timing. Modifying operation means insert: insert a node, delete: a node. So, this is basically the methodology behind the our data structure augmentation, basically we take a underline data structure. Then we put some extra bit of information based on our problems. So, in the next class we will do another problem where we have to use the data structure augmentation, which is called interval trees. So, we will discuss in the next class.

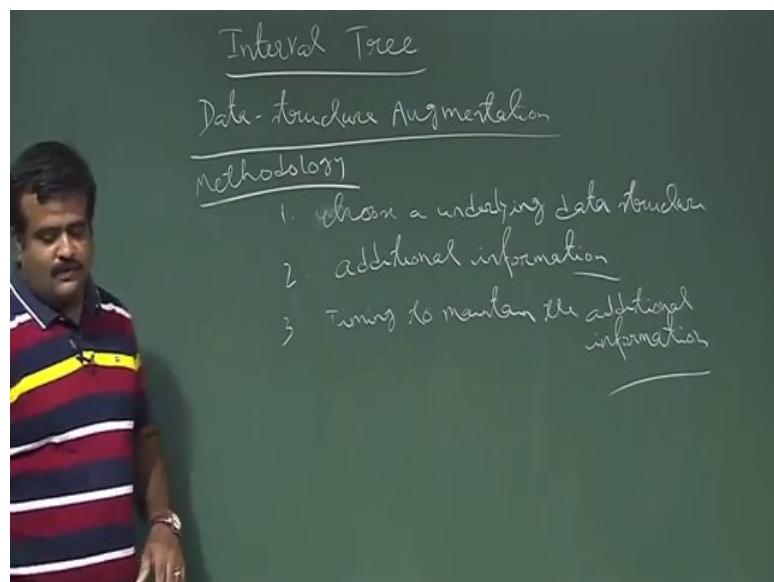
Thank you.

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

**Lecture - 30**  
**Interval Trees**

So we are talking about data structure augmentation. So, we have seen the dynamic order statistics problem how we can use the data structure augmentation. So now, in this lecture we will talk about interval tree. So, before the problem let us just recap the data structure augmentation methodology.

(Refer Slide Time: 00:42)



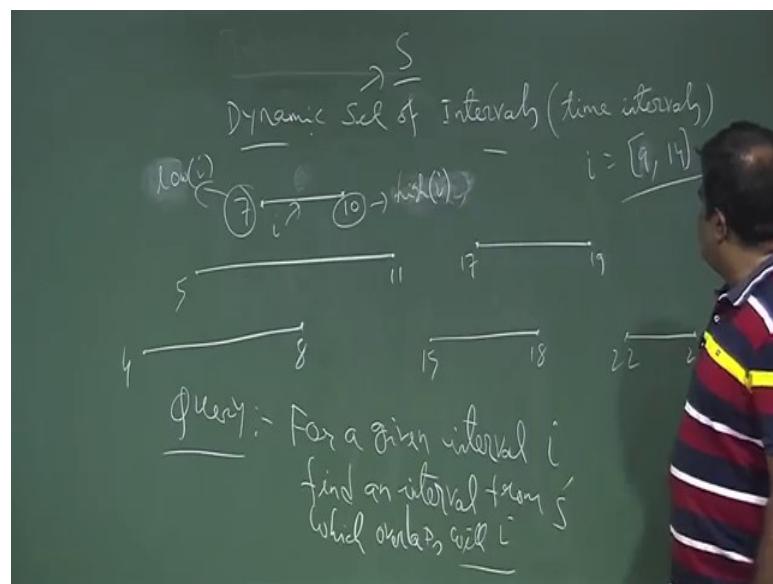
So, the idea is we have to choose a underlying data structure. So, we have to choose a underlying data structure and then after that we have to determine the additional field of information we are going to keep for these data structure for our problem. So, additional information we want to keep. And then we have to see how this additional information can be maintained, when we do the modifying operation like insertion deletion. If we choose the underlying data structure as the red black tree like we did for dynamic order statistics.

And then the additional information we kept as the size of the subtree rooted at that node. And we have seen that modifying operation insertion is similar time, same time as we do it for the red black tree. So, the additional information can be maintained with the same

timing as the original data structure. So, this is also crucial because if we are spending much more time to maintain this additional information field then there is no use of these. So anyway so, this is the methodology behind the data structure augmentation.

Now, the interval tree problem is we have given some intervals there are basically time intervals and that set is dynamic set. So, any interval can join at any point of time and any interval can leave from any point of time.

(Refer Slide Time: 03:21)



So, we have a dynamic set of intervals, basically time intervals and this set is dynamic. We have a dynamic set of intervals. And then now query suppose we have this interval say  $7 \ 10$ ,  $5 \ 11$  and say  $17 \ 19$  and we have  $4 \ 8$ . So, these are the say  $15 \ 18$  say  $22 \ 23$ . So, suppose we have a dynamic set of interval. So, this is a collection of intervals, each is the interval their time interval this is a  $I$ ,  $I$  is a interval and this is this is left of  $I$  left endpoint. And this is  $10$  is the right of  $I$  right endpoint ok.

So, this is basically low or high. So, low endpoint or this is the high end point. So, this is low endpoint and this is high end point. So, any interval has 2 endpoints low and high. So, this set is dynamic set. So, any point of time any interval can join and any point of time any interval can leave. So, this set is  $S$  set. So, this  $S$  set is basically  $S$ , the collection of interval. And this set is dynamic set now we need to do this query like this, then for a given interval, for a given interval, say we have given interval  $I$ , we need to find an interval from  $S$  which is overlapping with  $I$  find an interval from  $S$  which

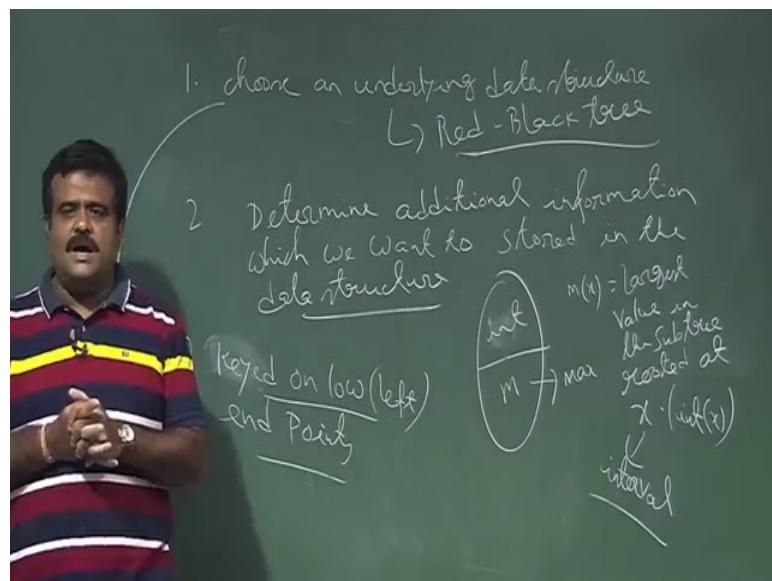
overlaps with I. So, our problem is to the query is we have given a interval I. So, I has a low and high endpoint and then we need to find the interval which is overlapping with this intervals.

So, for example, if I is say we have say interval say 8 or 9 14. If I is 9 14 then we need to find an interval which is overlapping with 9 and 14 so.

9 and 14 is this 7 and 10 is overlapping with 9 and 14. So, we can just return 7 and 10. So, we thought this problem we are just looking for an interval which is overlapping with this. So, this is the problem. So, for this problem we need to maintain a data structure, we need to have a data structure to maintain this set S. So, we need to do the data structure augmentation. So, for that we will use a instead of having a new data structure or data structure from the scraps, we just use a underlying data structure and we will do some augmentation there to act some new field. We store some I mean here new information.

So, basically this is the methodology we choose and underlying data structure.

(Refer Slide Time : 08:00)



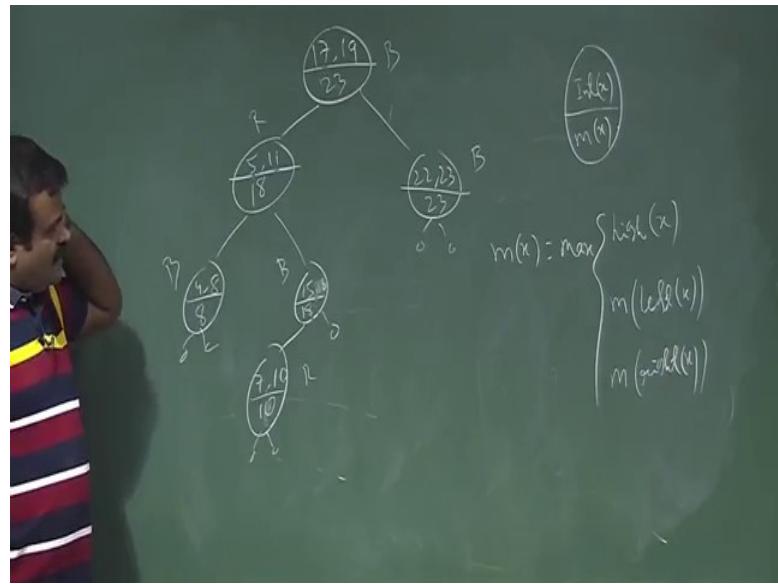
And here we are going to choose the red black tree, because red black tree is a balance tree. So, any operation can be done in logarithm time. And then this is the methodology of augmentation we are going to use for our problem. Then we need to have the additional information. So, we need to determine additional information which we want to store in the data structure.

And which will help us to solve our query. So, our problem is the interval search problem. So, we have given time intervals and we have given a query intervals, and we have to find out the interval which is overlapping with this query intervals. So, that is the problem. So, we need to determine the additional information which will help us to solve our problem. So, this is additional information we will keep, this is the key this is the interval basically. So, we will put this intervals over here and here will use the max m is the so, basically we keep this interval in a red black tree and using the key value as the left in point. So, we will keep the interval each interval is a node and the key on low or the left endpoint. So, each interval is a node. Based on the key value we are making the binary search tree if a node key value is less it should go to the left part if is more right part like this.

So, we will make this structure the key using the left endpoint or the low of this I. So, each interval is a node which is keyed on the low endpoint, and we are keeping this field we are keeping the maximum value rooted at that subtree. This is the basically m of x is the largest value in the subtree rooted as x, x means interval x I into x, x is an interval.

So, x is an interval. So, it has two endpoint low and high. So, based on the low endpoint, we make the tree the binary search tree not only binary search tree is the red black tree. And then this additional field we are keeping to store the maximum value of the maximum largest value which is stored in the interval rooted at x. So, we have the example.

(Refer Slide Time : 12:45)



So we just have this intervals 5 11 this are the low and high endpoints.

So, each interval is a node, and which is keyed as a low endpoint 8 9 sorry, 4 8 15 18 15 comma 18 and we have 7 comma 10 yeah. So, these are the intervals we have given and this is the tree we construct and based on the low endpoint, if you see 5 is less than 17. So, 5 is here a 4 is less than 5 like this. So, 15, 15 is less than 17, but greater than 5 it is here. So, 7, 7 is less than 17 7 is greater than 5 or 7 is less than 15. So, 7 is the left. So, this is the binary search tree. Only binary search tree we can make it a red black tree by giving the color this we can put black, these 2 black, these 2 black, this is black and this is red and this is red.

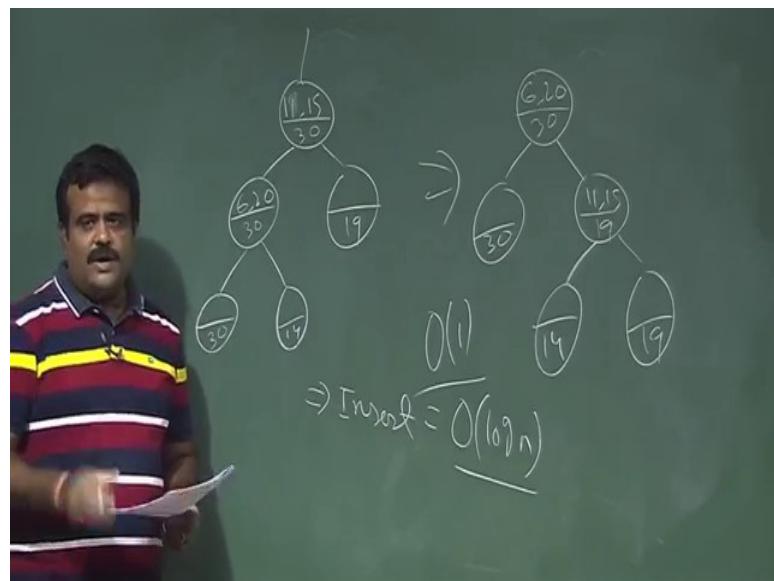
So, we can put these nils. Now we need to so, this is the red black tree this is our underlying data structure red black tree. Now we have a additional bit of information which is basically max bit. So, this is the interval x and this is the max of x. So, max bit means the maximum. So, largest element rooted at that tree now what is the. So, this the nils. So, they are 0 basically what the largest element rooted at this tree 10 because the that is the high endpoint. This is 8 this is 23 now what is the largest. So, this is the 10 is the largest in the left subtree and there is nothing in the right subtree and this is the largest here.

So, this is basically 18. Now what is the largest over here? 8 is the largest in the left subtree, 18 is the largest in the right subtree, and the largest in this interval is 11. So, this is basically maximum of these 3. Similarly here 18 is the largest in the left subtree, 23 is

the largest in the right subtree and the largest in this interval is 19. So, maximum of these 3 is 23. So, formula for m of x basically maximum of these 3, high of x high of x then maximum of left of x and maximum of right of x. So, this is the formula for this additional information ok.

So, this is a red black tree with this additional information. Now we need to maintain t. How this additional information can be maintained when we will do the modifying operation like insertion and deletion. So, for that coloring will not be affect much. So, we need to look at this rotation operation, how it will affect in the rotation operation. So, to look at that let us just try to see whether we can maintain this additional information, with the same time while we are performing the rotation.

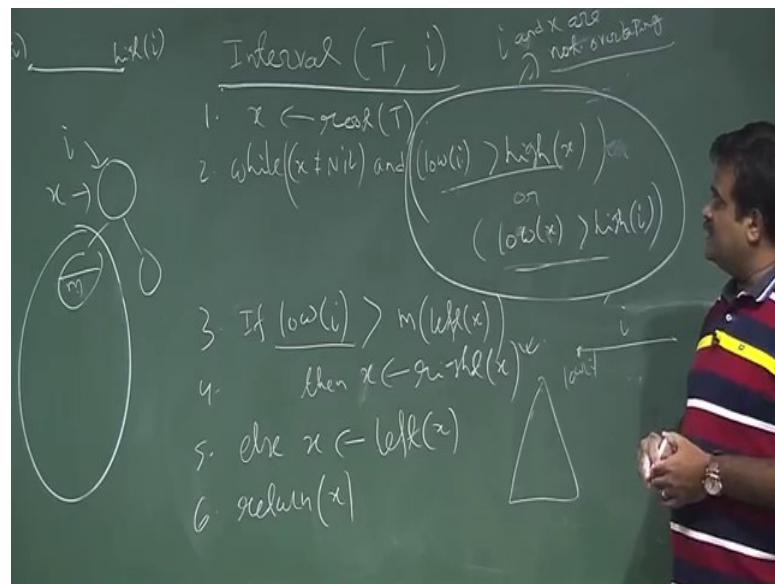
(Refer Slide Time : 16:47)



So, like this suppose we have this interval 11 15 6 20 and this is the say so we have a left subtree we have a right subtree over here. So, this is 30 say we have 30 over here. Now suppose we make this up, this down this is alpha beta this is gamma, this is a subtree rooted at this are the subtree. So, this is part of the tree we want to see the how we can perform the rotation operation, and while performing rotation operation how we can fix this extra additional information in the same time. So, this is the right rotate on this. So, you want to make this up and this down. So, so 11 15 will come down and then this will be hanging here 30 and alpha beta. So, alpha this is beta this is gamma ok.

Now this will be maximum of this 3 19 and this will be maximum this 3 30. So, this can be fixed; order of one time with the same timing with the rotation operation. Because rotation also need to change the point of view pointer. So that means, this imply insertion and deletion will take the same time as red black tree original red black tree insertion and deletion. So, that is what augmentation is. So now, we will see how this extra bit of information this augmentation will help us to have the search or interval query. So, that is the interval search ok.

(Refer Slide Time : 18:51)



So, we have a set of intervals we have a set of intervals which we are maintained in that augmenting red black tree. And we search a query interval which is over we want to see which is overlapping with this interval. So, we have a tree over here which is based on our S. This tree is coming from S set. So, what we are doing we are taking the root of this tree. So, basically we have this tree which is basically the intervals like this like this So on. So, this is our tree and this is the root, this is the on S set we are maintaining this thing. So given an interval I, I is basically we are given lower endpoint. So, basically we need to find an overlapping interval with this. So, how we do so now, if there is element. So, until see if x is not null; that means, if there is only the root and, and if this low of I is greater than high of x. So, this and or low of x is greater than high of I. So, we have this we have this interval. So, this is our x this is the root.

Now we check whether I is overlapping with this. So, this is the check whether I is not overlapping. So, how to check? So, we have a x interval. So, x is having here. So, we have a x interval. So, x is basically having 2 endpoints this is the low of x this is the high of x and we have an interval I, which is the query interval now when it will not overlap with this x, x is the root. Now we check whether it is overlapping with the root, if it is not overlapping then we will go for the left part or right part recursive call basically. So, before that we have to check whether it is overlapping with the x or not. So, how to check that? So, basically, when it will not overlap, either I is completely this side. So, this is low of I and this is high of I; that means, if high of I is less than low of x this is either this or if it is completely that side. So, completely that side means if low of I is greater than high of x. So, either of these 2. So, this means the interval is completely that side and this means interval is completely this side. So, this means they are not overlapping, this means this condition means this whole condition means I is I and x is not overlapping, not overlapping not overlapping.

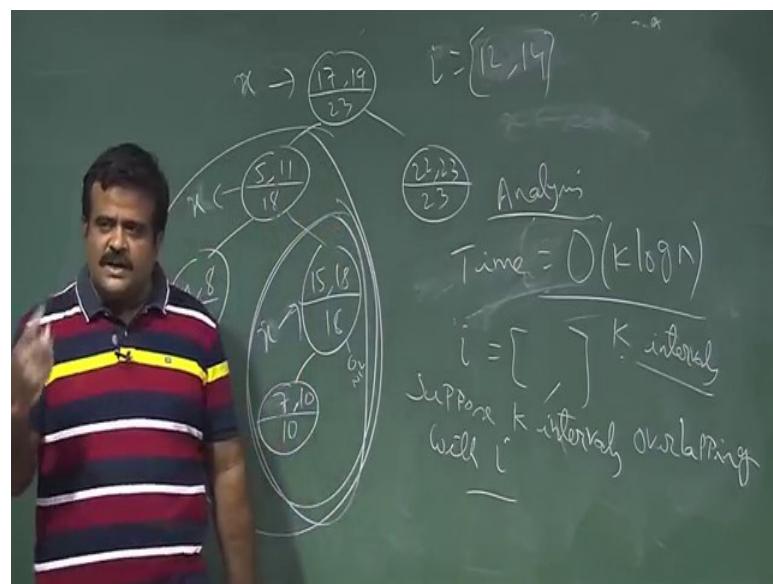
If they are overlapping then we can return x, if this is false either it is reached to a null. If it is not overlapping, and it is not nil then we have to go further left or right part. So, this that overlapping then we have to go further. So, how to go for that? So, for that So, I and x is not overlapping by this condition then what we do? Then we do we just check if left of x is not null.

And low of I, if low of I is greater than max of sorry, if low of I is if left of x left of if low of I just a minute. So, how to check this? If low of I is greater than m of left of x; that means what? So, m of left of x so that means,. So, this is the m of left of x, So that means, if m of left of x is less than low of I So that means, there is nothing interesting in the left part. So, we are looking for a interval which is overlapping with I, if the low of I is greater than m of left.

If low of I is greater than maximum largest value rooted under this part then there will be no interval in the left part which is overlapping with I then we must go for the right part, then x is right of a x else x is left of x that is it. So, if either then we return finally, we returned x. So, either x is nil so that means, there is no interval overlapping with x or x is this. So, this means is this clear. So, this means if. So, if the low of I if the interval we are looking for. So, this is the low of I this is ith interval.

If the interval we are looking for low value is greater than the left part of the x the maximum value of the left part of the x, then there is nobody interval will be there which can overlap with I. So, we have go for the right part of the I. So, that is the idea. So, this is the pseudo code for inter interval search and the what is the time complexity for this? So, time complexity is basically  $\log n$ . So, we will do just quick example. So, we will just do some example on this. So, let us have the interval (17,19) (22,23) (5,11) (4,8) (15,18) and we have (7,10).

(Refer Slide Time : 26:45)



And we have the nils and we have the color. Now suppose we search we search the I is equal to we want to search the interval 14 and 16. So, low of I is 14 high of is 16. So, we will just execute the interval search we have seen now. So, we start with the root x and we check whether this is overlapping with this. So, 14 16 is completely this side I means it is not overlapping. So that means, and it is not null also. So, we then we have to fill this part also. So, this basically 8, this is 10 this is basically 18, 18, 23, 23 ok. Now what we do we just take x to be root.

And this is the not overlapping, now this is basically 14 is basically less than 18. So that means, we have to look at this part of the tree, if 14 is greater than 18 then there is nothing interesting over here. Then we have to go for the right, but here 14 is less than 18. So, now, our x is this. Now we check whether this is overlapping with this 14 this is not overlapping with this, now we check 14 left part of this now left part of this

maximum value is 8, but 14 is greater than 8. So, there is benefit to going to the left. So, we will go to the right of the tree. Now this is our x. Now we compare this and this will be overlapping. So, it will return as this return 15 18 this interval.

So, 15 18 will returned. Now suppose we want to search for interval which is not there. So, suppose we want to search say 12 and 14. So, how will do? So, we will start with x over here. So, it is not overlapping. So, now, this for 12 is less than this. So, we will go to this part now this is our x again we check it is not overlapping. So, again we compare 12 with these 12 is greater than we have to come this part this our new root, and then we compare with 12 and this it is not overlapping, and we check 12 with 12 is greater than we go to the right is nil. So, basically we stop and it is returning us nil so that means, there is no interval which is overlapping with this. So, the time is basically order of  $\log n$  to report one interval ok.

But the question is if suppose there are k intervals overlapping with this and we want to retrieve all of them. So, what then how we can do that? Suppose there are k intervals which are overlapping with this interval I, which the given interval I and we need to return all these k intervals.

So, what we can do we can first get the interval which is overlapping with this and then we delete it from this, augmented red black tree and that deletion can be done in logarithm time, again we make the search after deleting that it will give us the second interval which is overlapping with this. Again after getting that we delete it. So, basically the time will be  $k * \log n$ . So, this is the time to report the k interval which are overlapping with this. Because every time once we got the interval which is overlapping, we will delete it because this is the red black tree deletion will be in  $\log n$  time and after deleting that note that interval we again perform this interval tree operations, searching.

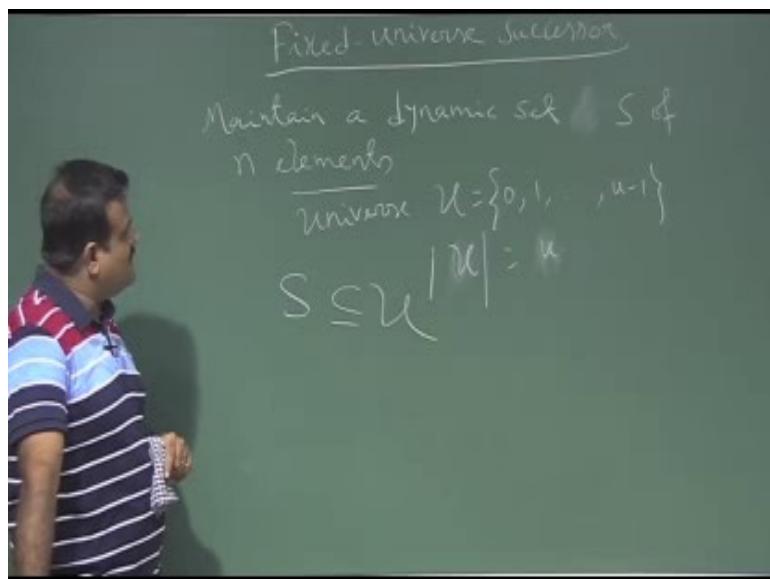
And then again it will take logarithm time. So, this way it is the  $k \log n$  algorithm, this is the output sensitive algorithm ok.

Thank you.

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

**Lecture - 31**  
**Fixed Universe Successor**

In this lecture, we will be talking about the fixed universe successor problem, which is also a problem to deal with a dynamic set. The primary goal is to maintain a dynamic set  $S$  of  $n$  elements.

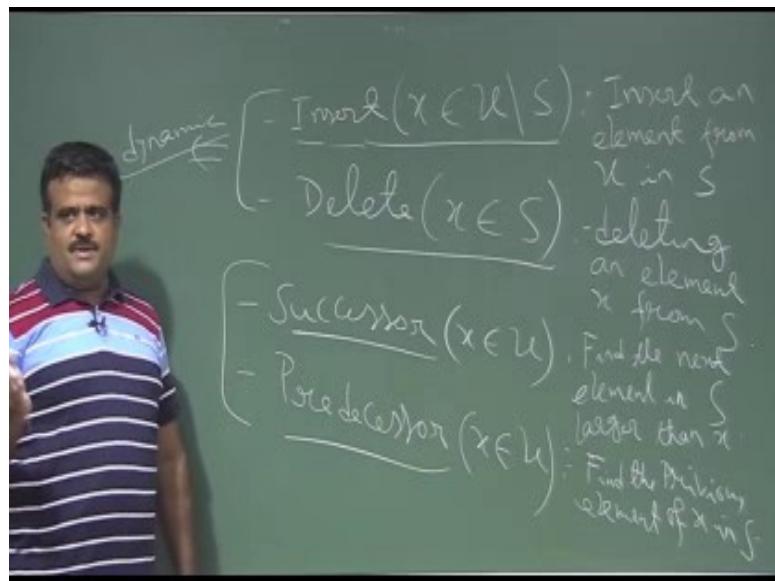


At some point of time, there are  $n$  elements in this set  $S$  and this set is coming from a fixed universe. We have seen many dynamic set problems like heaps, balanced binary search trees. But here, problem has a criteria that the elements are coming from a fixed universe,  $U$ , which is basically

$$U = \{0, 1, \dots, u-1\} \text{ where } |U| = u$$

And,  $S \subseteq U$ .

The goal is to perform dynamic operations like insert, delete, successor and predecessor on this set  $S$  efficiently.



Insert operation is to insert an element  $x \in U \wedge x \notin S$  into  $S$ . Delete operation is to delete an element  $x$  from  $S$ . Being able to perform these operations make this set dynamic.

Successor operation (also dynamic) - we want to find the next element(larger than  $x$ ) of  $x$  in  $S$  where  $x \in U$ . Another query is finding the predecessor of  $x$ , which is the previous element of  $x$  in  $S$ .

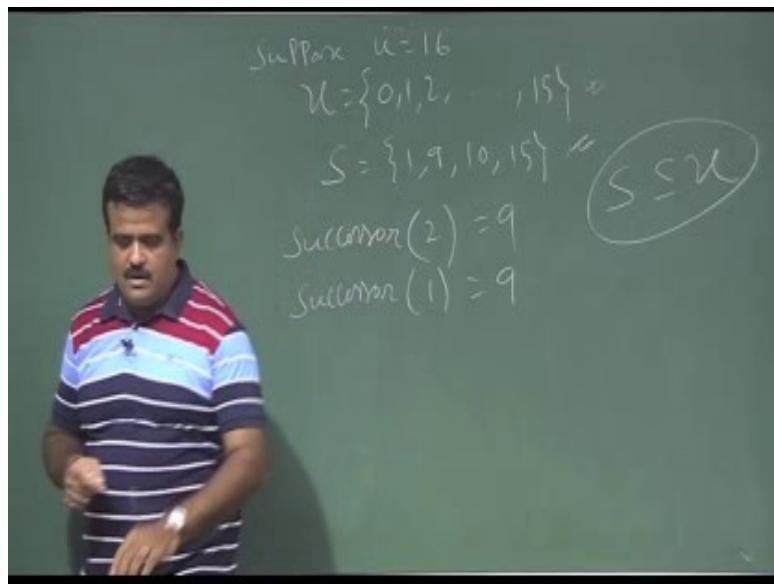
We need to think of a data structure so that we can perform these operation in an efficient way.

Consider an example. Suppose,

$$U = \{0, 1, 2, \dots, 15\}, \forall U \vee u = 16$$

Suppose, at some point of time, the dynamic set  $S$ ,  $S \subseteq U$ , is

$$S = \{1, 9, 10, 15\}$$



What will be the successor of 2 in  $S$ ?

We have to find the next element of 2 in  $S$ , which is 9. So, successor of 2 is 9, similarly successor of 1 is also 9.

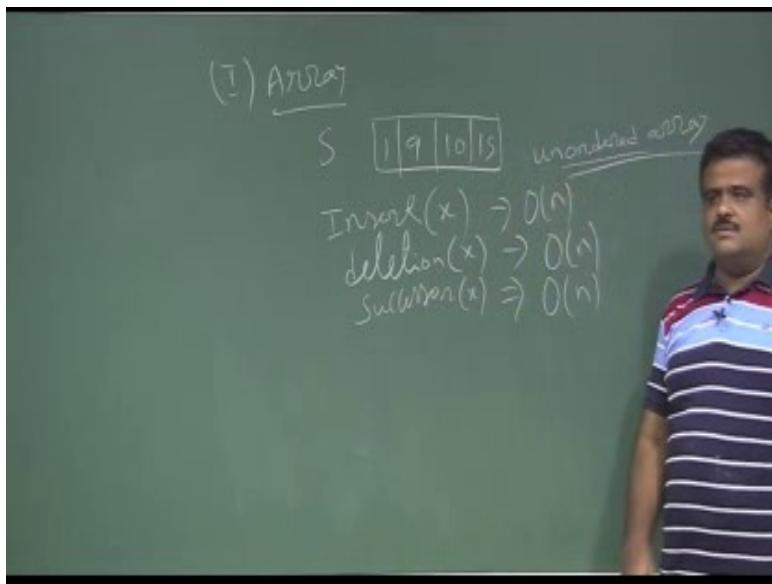
Hence, We have to find the next element of  $x$  in  $S$ ,  $x$  may or may not be in  $S$ , which is the successor query. (predecessor query is similar). We need to think of a data structure for this dynamic set  $S$  so, that we can able to insert an element  $x$ , delete an element  $x$ , and perform the successor and predecessor query for an element  $x$ . Also, this is fixed universe,i.e., universe is fixed. So,  $x \in U \wedge S \in U$ . So, this is called the fixed universe successor problem.

What type of data structure can you think for this problem?

(I) Array

Suppose we maintain an n-dimensional array for  $S$ . Thus, for our example,

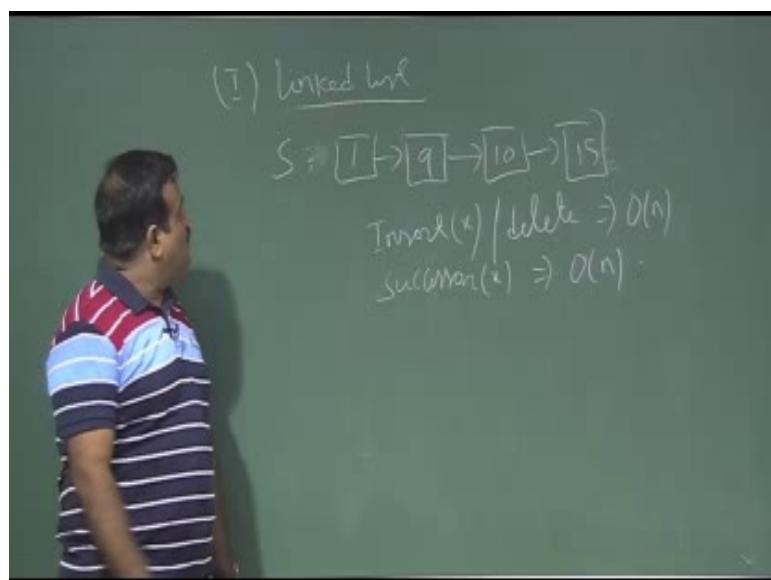
$S=\{1,9,10,15\}$ is an array of size 4



Time complexity for various operations -

Insert and Delete. For a sorted array, insertion would need addition of an element and then shifting the array contents. This would take  $O(n)$  time.

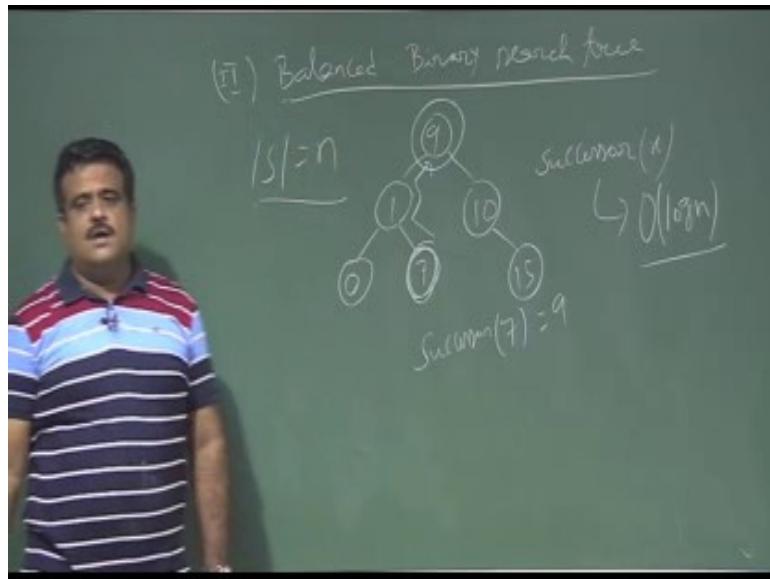
Successor. For an unordered array, this would need a complete traversal of the array, thus the complexity is  $O(n)$ .



(II) Linked list

$S=1 \rightarrow 9 \rightarrow 10 \rightarrow 15$

If the list is unordered, all the operations(insert, delete, successor) will take  $O(n)$  time



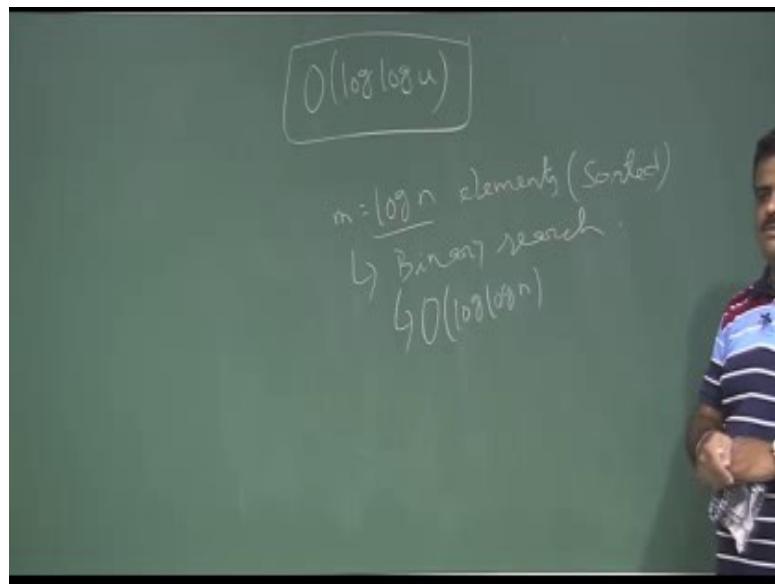
(III) Balanced Binary Search Tree (eg - Red black tree)

(Refer to image for the BST)

Insert will take  $\log(n)$  time because you have to insert the element in the BST, and then we may need to make it balanced. So, if the BST is a Red black tree, it is just a Red black tree insert/ deletion.

Here, how do we find the successor of 7. The algorithm will basically involve an inorder traversal, which will give the successor of 7, i.e, 9. So, We need to traverse the height of this tree. Thus this method will take  $O(\log n)$  time because this is a balanced BST, whose height is  $\log n$ .

Now, Our goal is to perform the successor query(and insertion, deletion) in  $O(\log \log u)$  worst case time, and for the same, we need to find a data structure which does exactly this.



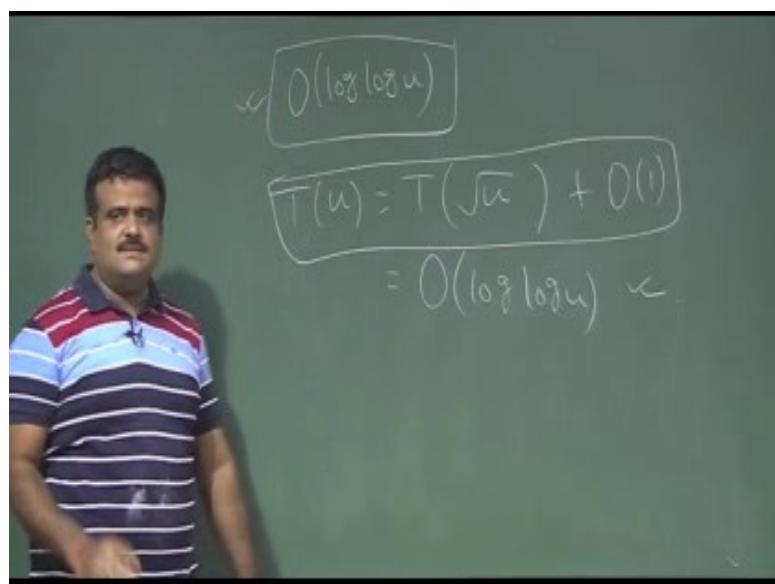
Suppose we have  $\log n$  sorted elements. Binary search on this set will take  $\log \log n$  time. But we want the same for  $n$  elements.

Can you think of any recurrence which can give this solution of order  $\log \log n$ ?

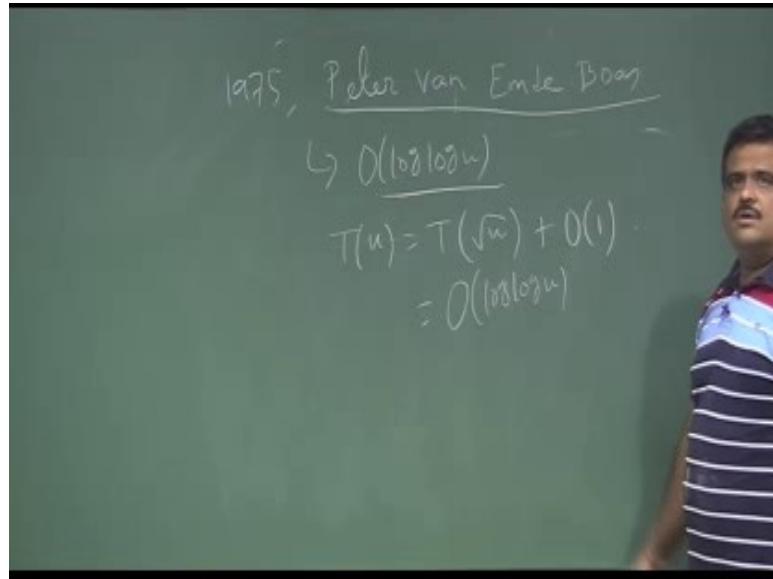
$$T(u) = T(f(u)) + O(1)$$

Let  $f(u) = \sqrt{u}$ ,  $T(u) = T(\underline{u})$

Thus,  $T(u) = \log \log n$



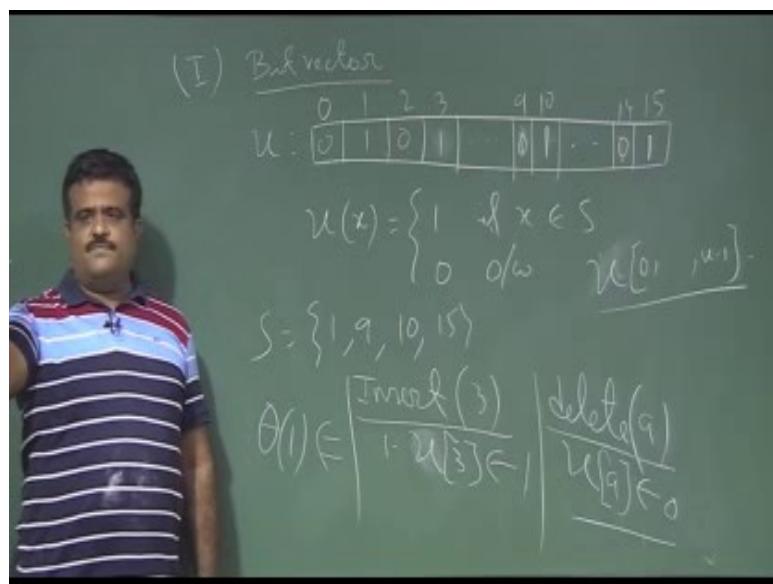
Somehow we need to achieve this recurrence in order to get the desired time complexity.



*Peter Van Emde Boas* discovered a data structure in 1975 for the same which solves the successor query in  $O(\log \log n)$  time.

So, Firstly, think of an array data structure, some other type of array for this problem.

(I) Bit Vector.



An intuitive way to think about a bit vector is a switch in a room. It is on if a person is there in the room and off otherwise. So, for our  $S = \{1, 9, 10, 15\}$  the corresponding elements in the vector are 1 whereas all other elements are 0. It is a very naive, very simple data structure and it is very effective.

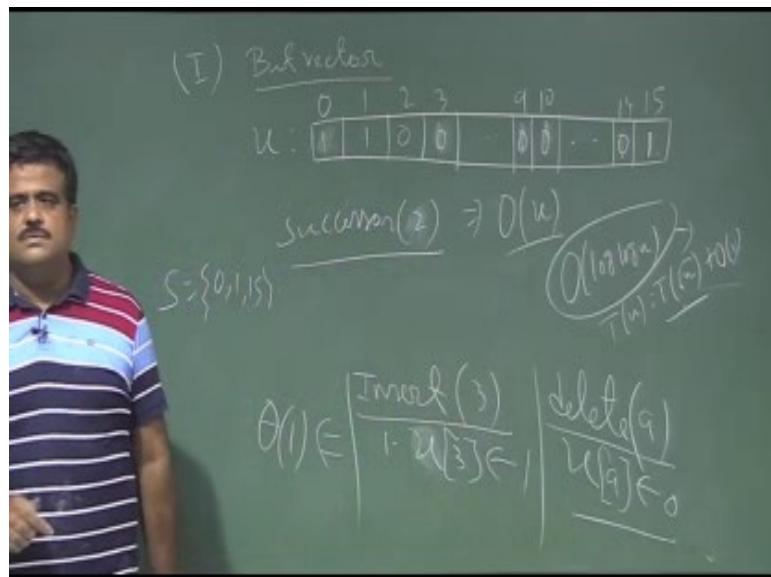
For the universe, this is an array of size  $u$ . And, for the example,  $u = 15$ . such that,

$u[x] = 1 \text{ if } x \in S, 0 \text{ otherwise}$  (Refer figure)

Operations -

$\text{insert}(x) \rightarrow u[x] = 1$ , time complexity =  $O(1)$

$\text{delete}(x) \rightarrow u[x] = 0$ , time complexity =  $O(1)$



$\text{successor}(x) = ?$  time complexity =  $O(u)$

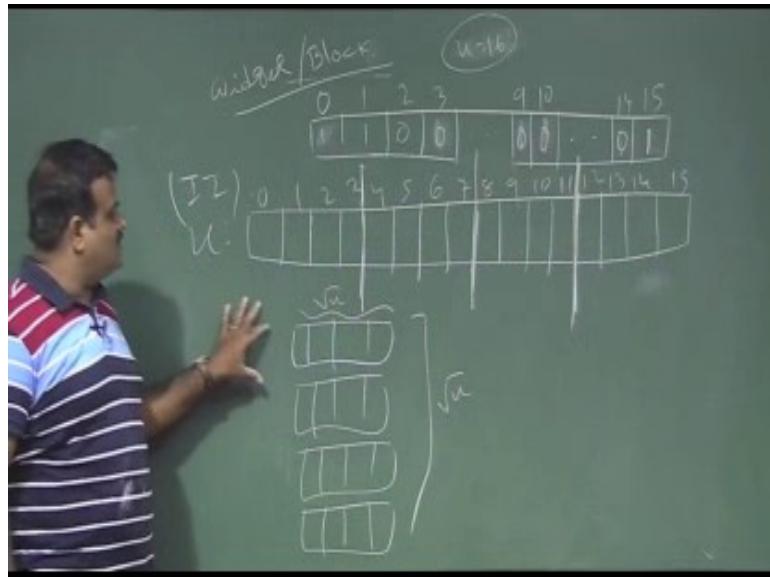
For the successor query, we need to scan the whole array(size  $u$ ). This will take worst case time  $O(u)$  because we may need to scan the whole array. Example,  $S = \{0, 15\}$ , successor of 0 is 15, but we will need to scan the whole array from 0 to 15 to find 15. So, how do we go about achieving  $\log u$  let alone  $\log \log u$ .

Although the complexity is  $O(u)$ , this is a good start nonetheless, this is very simple data structure with very powerful deletion and insertion in constant time, but the only problem is

with the successor which we need to achieve  $O(\log \log u)$  from  $O(u)$ . We need to achieve the following recurrence.

$$T(u) = T \textcolor{red}{\lfloor \sqrt{u} \rfloor}$$

In order to get this we need to somehow realize this  $\sqrt{\square}$ . We have an array of size  $u$ . How can we make it into  $\sqrt{\square}$ ? This is the next step, i.e., step 2.



Method - We break the array into blocks of size  $\sqrt{\square}$ . And we make  $\sqrt{\square}$  such blocks or widgets.

In our example, we have an array of size,  $u = 16$ . We can convert it into a  $4 \times 4$  square matrix.  
(Refer figure for conversion example)

$$U = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]$$

can be converted into

$$U =$$

$$[[0,1,2,3]$$

$$[4,5,6,7]$$

$$[8,9,10,11]$$

12,13,14,15]]

Thus, a  $u$  sized array is converted into a square matrix of size  $\sqrt{u}$

Thus, we now have a  $\sqrt{u}$  term. We have augmented our bit vector data structure from a 1-D data structure to a 2-D data structure. In the next lecture, we will discuss insertion and deletion in this data structure.

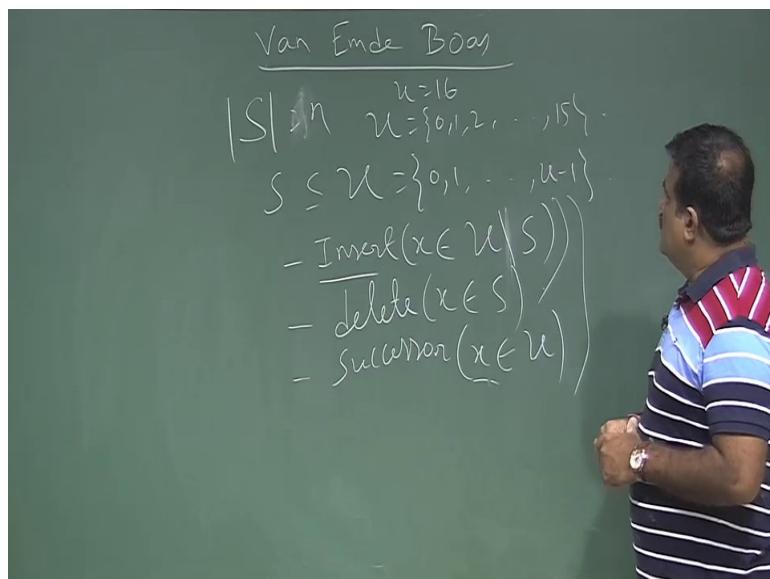
Thank you.

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

**Lecture – 32**  
**Van Emde Boas Data Structure**

In the last lecture, we started talking about the fixed universe successor problem. To recap, the goal is to maintain a dynamic set  $S$  of  $n$  elements ( $|S| \leq n, S \subseteq U$ ) where  $S$  is coming from a fixed universe,  $U$  ( $|U| = u, U = \{0, 1, \dots, u-1\}$ ), so that we are able to perform several operations such as -

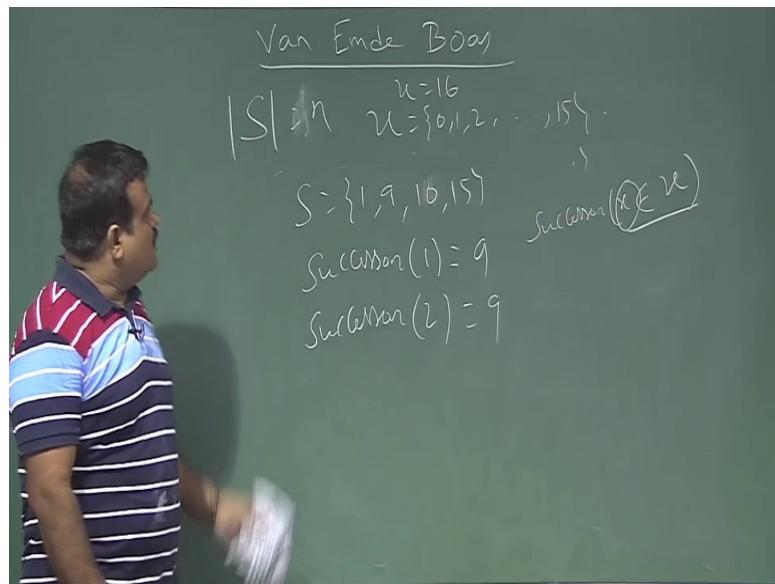
1. We should be able to insert an element  $x$  which is not in  $S$ .
2. We should be able to delete an element  $x$  from  $S$ .
3. We should be able to find the successor of an element  $x \in U$ . So, we should be able to get the next element of next element after  $x$  in  $S$ .
4. Similarly, the predecessor query.



The first 2 operations make this set dynamic.

Suppose,  $u=16$ .

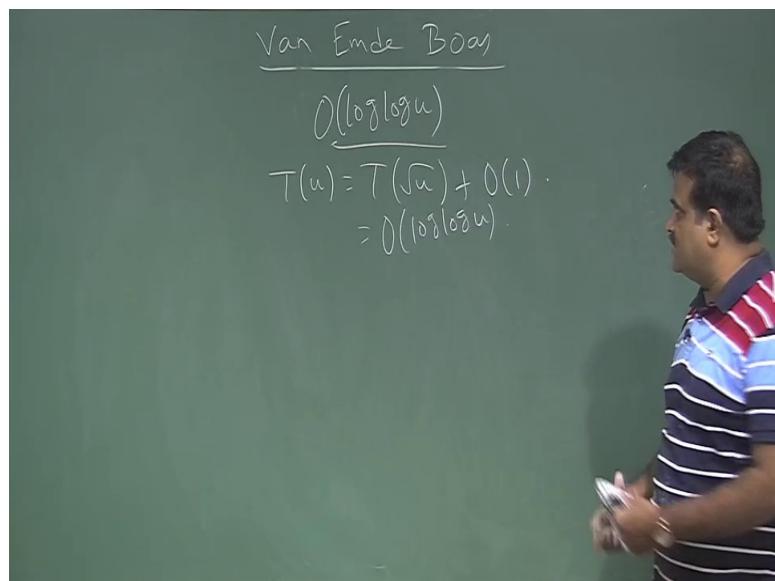
And at some point of time, say,  $S=\{1, 9, 10, 15\}$



Here, successor of 1 is 9 and the successor of 2 is also 9. So, successor of  $x$  means the next element which is in  $S$  after  $x$  where  $x$  need not be in  $S$  ( $x \in U$ ).

We want to solve this problem of performing these operations in  $\log \log u$  time.

(Refer Slide Time: 03:14)

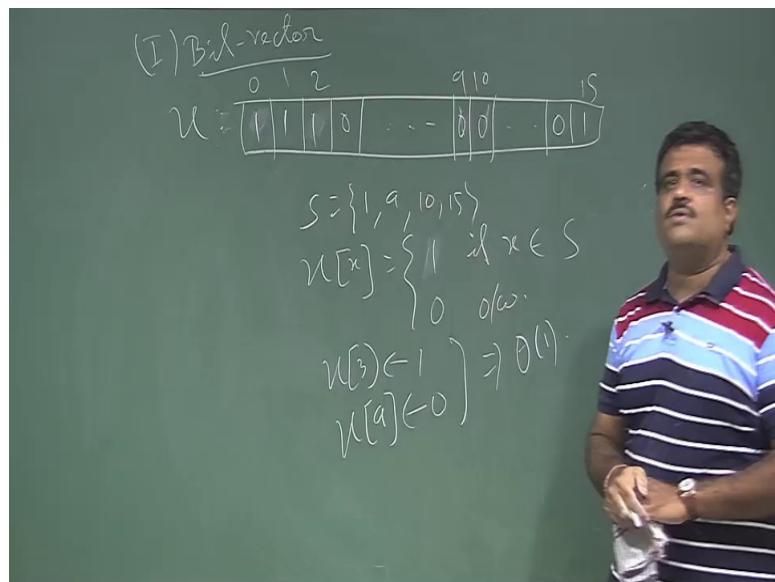


We have seen in the last class, to get  $\log \log u$ , we have to achieve this type of recurrence -

$$T(u) = T(\sqrt{u}) + O(1)$$

So, the idea of the solution is from Peter Van Emde Boas who developed the following data structure for the problem.

(Refer Slide Time: 03:57)



So, We started with a bit vector, which is a new array of size  $u$ . (Light off light on data structure). To recap,

$$u[x] = 1 \text{ if } x \in S, 0 \text{ otherwise}$$

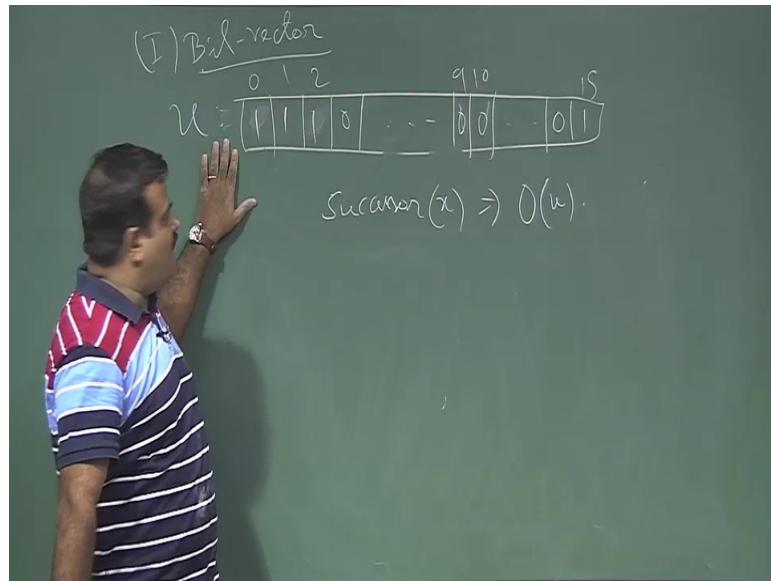
Operations -

$$\text{insert}(x) \rightarrow u[x] = 1, \text{ time complexity} = O(1)$$

$$\text{delete}(x) \rightarrow u[x] = 0, \text{ time complexity} = O(1)$$

$$\text{successor}(x) = ? \text{ time complexity} = O(u)$$

(Refer Slide Time: 05:49)

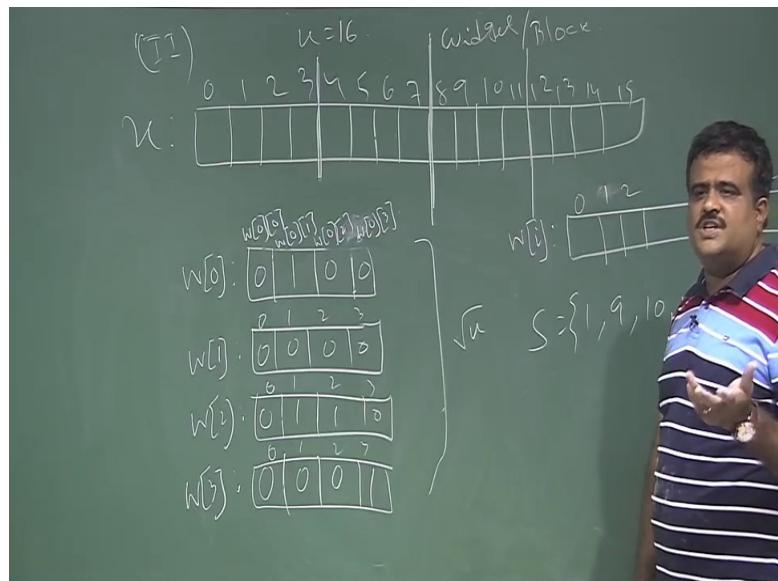


For the successor query, we need to scan the whole array(size  $u$ ). This will take worst case time  $O(u)$  because we may need to scan the whole array. Example,  $S = \{0, 15\}$ , successor of 0 is 15, but we will need to scan the whole array from 0 to 15 to find 15. So, how do we go about achieving  $\log u$  let alone  $\log \log u$ .

In order to get this we need to somehow realize this  $\sqrt{u}$ . We have an array of size  $u$ . How can we make it into  $\sqrt{u}$ ? This is the next step, i.e., step 2.

Method - We break the array into blocks of size  $\sqrt{u}$ . And we make  $\sqrt{u}$  such blocks or widgets.

(Refer Slide Time: 06:23)



In our example, we have an array of size,  $u = 16$ . We can convert it into a  $4 \times 4$  square matrix.  
(Refer figure for conversion example)

$U = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]$  can be converted into

$U =$

$[[0,1,2,3]$

$[4,5,6,7]$

$[8,9,10,11]$

$12,13,14,15]]$

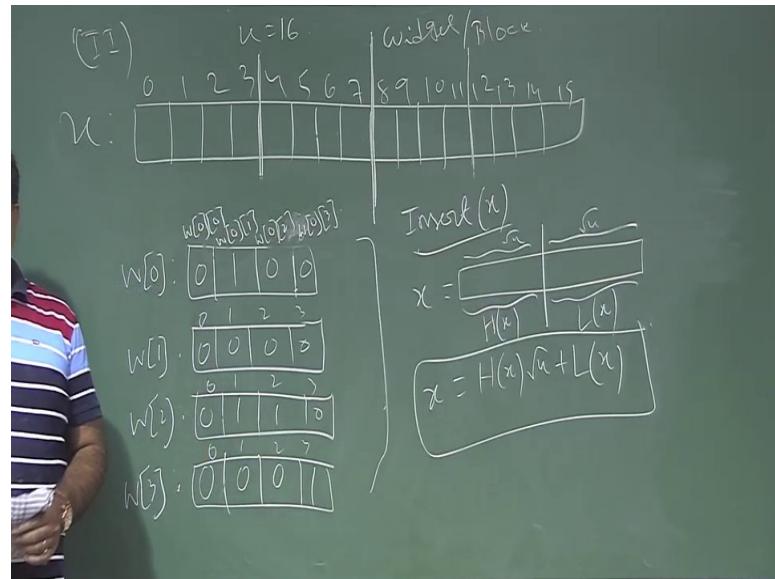
Blocks are named as  $W[0], W[1], W[2]$  and  $W[3]$ .

So, there are  $\sqrt{4}$  blocks and size of each block is also  $\sqrt{4}$ . Blocks are named as  $0, 1, 2, \dots, \sqrt{4}$

So, we've broken a 1-D array into 2-D array.

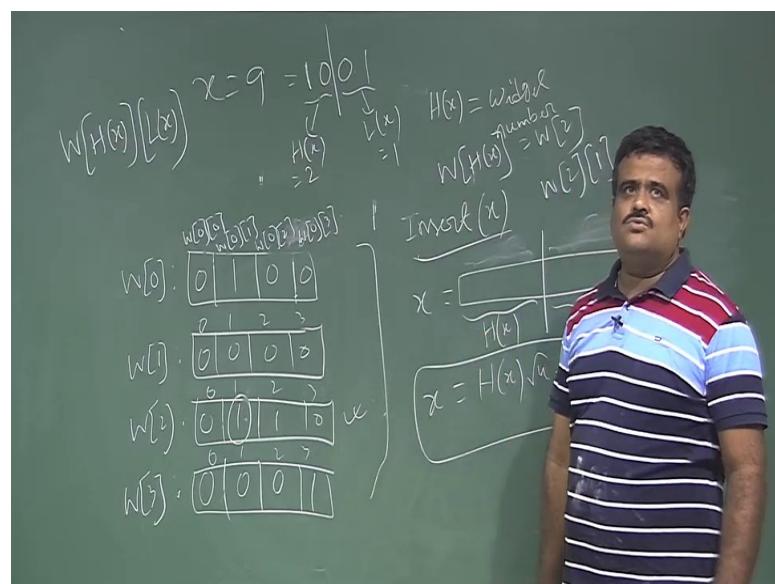
For our example, the  $u$  matrix is given in the figure.

(Refer Slide Time: 10:32)



Now, Suppose we want to insert x. We convert x into binary and we write x into 2 parts. So, we just convert x into binary and it will be of size root u and we take first half as  $H(x)$  and second part as  $L(x)$ . So,  $x = H(x)*\sqrt{u}$

(Refer Slide Time: 11:44)



For example,  $x=9$ . We know that,  $u=16$ .  $\sqrt{u} = 4$ .  $9 = (1001)_2$ .  $H(x)=(10)=2$ ,  $L(x)=(01)=1$ . Thus,  $x = 9 = H(x) * \sqrt{u} + L(x) = 2 * 4 + 1$

To get the position of the element in the array,  $W[H(x)][L(x)]$ .

1. Insert x. set  $W[H(x)][L(x)] = 1$
2. Delete x. set  $W[H(x)][L(x)] = 0$

(Refer Slide Time: 14:30)

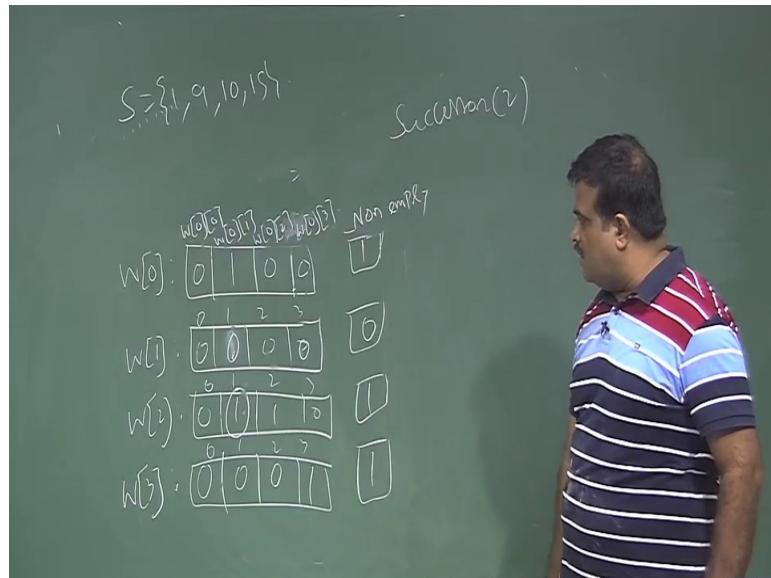
eg:  $5 = (0101)$ ,  $H(x) = (01) = 1$ ,  $L(x) = (01) = 1$ . insert 5  $\rightarrow W[1][1] = 1$ , delete 5  $\rightarrow W[1][1] = 0$

(Refer Slide Time: 16:22)

Suppose we want to find the successor of 2. We go to the position of 2 and then find the first non empty bit after that. This will take  $O(u)$  time.

Can you think of some augmentation of this data structure to avoid search of blocks where all entries are 0. We can put a single bit of information, say summary bit, which denotes if the block is empty(no 1 entries) or not. If for a block, all of its entries are 0, the summary bit is 0, else it is 1.

(Refer Slide Time: 18:51)



Now, suppose you want to find the successor of 2. So, we first find the successor in that block where 2 belongs. If we do not get the successor here, we look at the summary bits of the next block, which is 0, so no point in searching this block, so we move to the next block whose summary bit is 1. We find the first non 0 entry in this block which is 9, which is the successor of 2.

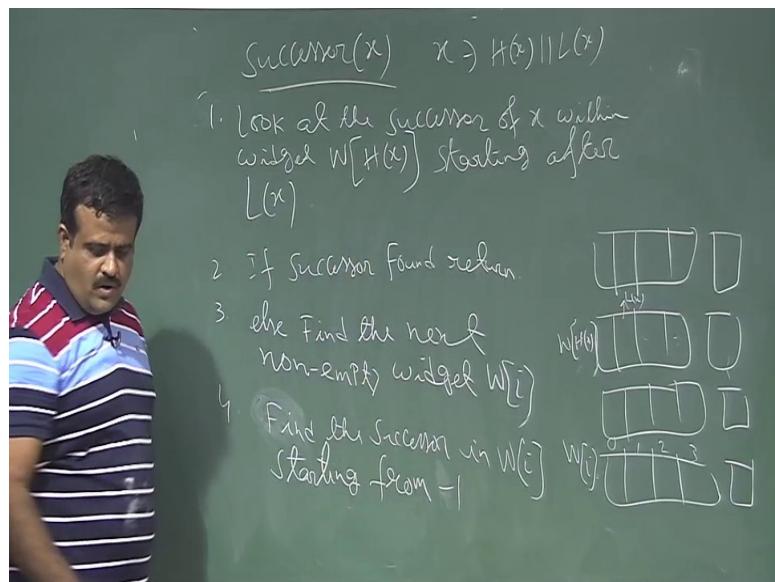
Pseudo - code:

$\text{Successor}(x), x = H(x)|L(x)$

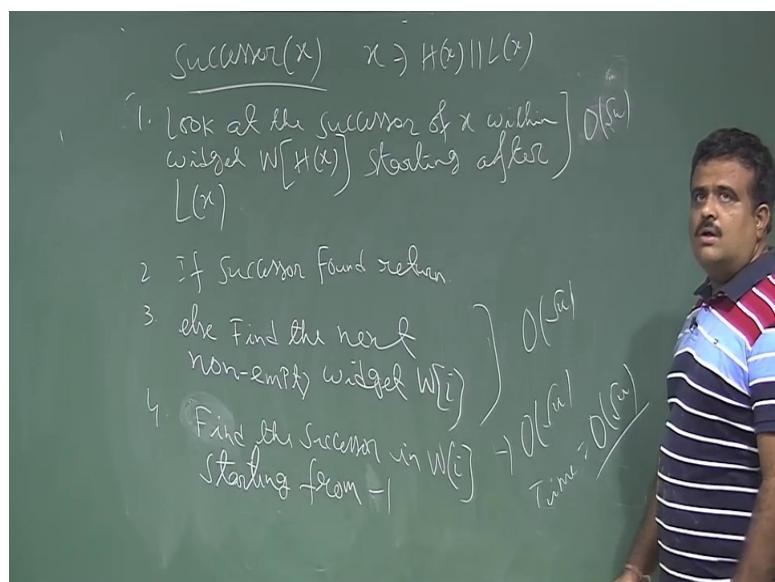
1. Look at the successor of  $x$  within the widget  $W[H(x)]$  starting after  $L(x)$ . // Takes  $\sqrt{\square}$  time
2. If successor is found, *return*.
3. else find the next non empty widget  $W[i]$ . // Takes  $\sqrt{\square}$  time
4. Find the successor in  $W[i]$  starting from -1 // Takes  $\sqrt{\square}$  time

Time complexity =  $O \textcolor{red}{i}$

(Refer Slide Time: 20:34)

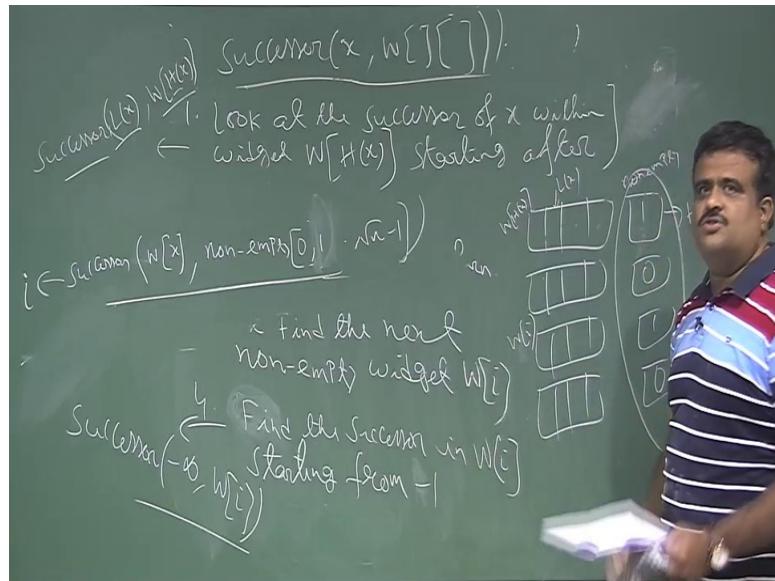


(Refer Slide Time: 24:26)



But we want to do it  $O(\log \log u)$ , we need to get a recurrence relation for the same. Thus we need to convert this algorithm into a recursive one.

(Refer Slide Time: 26:13)



step 1 is a successor call,  $\text{successor}(L(x), W[H(x)])$ .

step 3 is a successor call,  $\text{successor}(H[x] \text{ in summary\_array})$ , the summary array has size  $u$ .  
And elements numbered from  $0, 1, \dots, u - 1$

step 4 is a successor call,  $\text{successor}(-\infty, W[i])$  where  $i$  is the first non-empty widget after  $H(x)$ .

Thus, in the worst case, there are 3 successor calls. But we want to achieve the goal of having only 1 successor call.

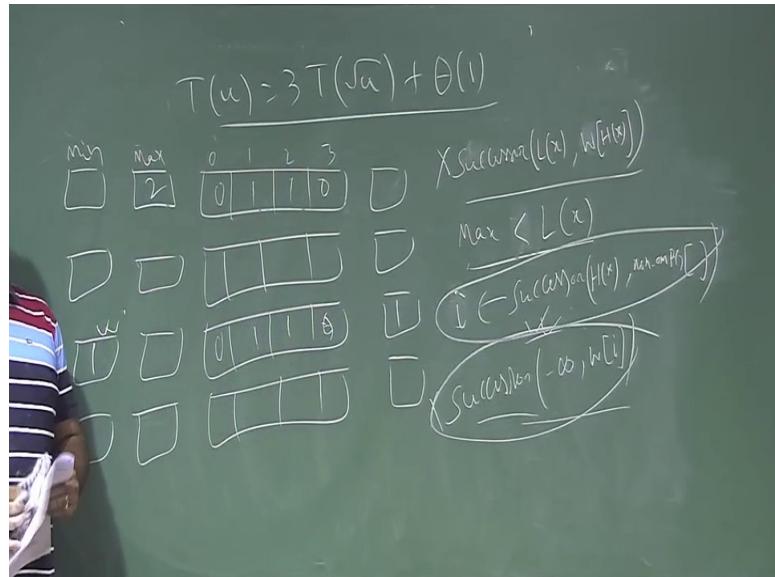
Using the idea by Van Emde Boas we need to do some more augmentation in the data structure. To avoid these calls, we can just have a maximum with each block. Thus we now have a max array. Max array stores the maximum index up to which we can find a 1 in that particular widget.

Thus, in the step 1, We first check if max is greater than x, if yes, then the successor is in the current widget (So, only 1 successor call), if not, the successor is not present in the current widget. Thus, successor call is not needed. And a successor call is saved.

But, in the worst case, 2 successor calls will still be required. If the successor is not present in  $x$ 's widget, we need to first call the successor method to find the first non empty widget

where the summary bit is non-zero, and then we need to perform a successor call in that widget for  $-\infty$ . Thus in the worst case, 2 successor calls are required.

(Refer Slide Time: 29:04)



We further augment the data structure to store the minimum index of the non-zero bit for each widget. So, this will save the last successor call, because, as we find the first non-empty widget, the minimum array will give us the correct position, which will be the required answer.

Thus, the new algorithm with time complexity  $O(\log \log u)$  is:

$\text{Successor}(x \text{ in } S), x = H(x)|L(x)$

1. Check if  $\max(H(x)) > x$ , if yes, return  $\text{successor}(L(x), W[H(x)])$ , else go to 2.
2.  $i = \text{successor}(H(x), \text{summary\_array})$ , gives the next non-empty widget.
3. return  $i * \sqrt{\square} + \min(i)$  as the required successor element.

Time complexity.

$$T(u) = T \text{ } \checkmark$$

$$T(u) = O(\log \log u)$$

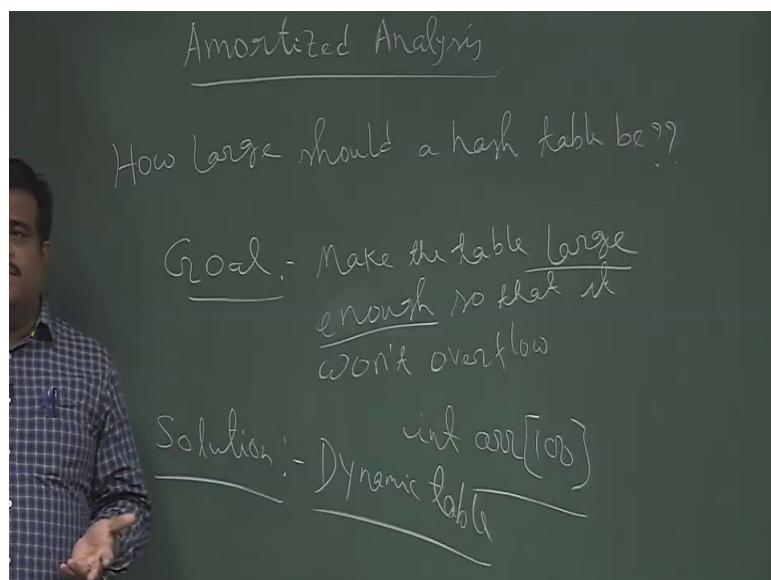
Thank you.

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

**Lecture – 33**  
**Amortized Analysis**

So we talk about amortized analysis, amortized algorithm. So, what do you mean by amortized analysis of an algorithm? So, for this let us start with the first question which we want to discuss. The question is how large should a hash table be?

(Refer Slide Time: 00:36)

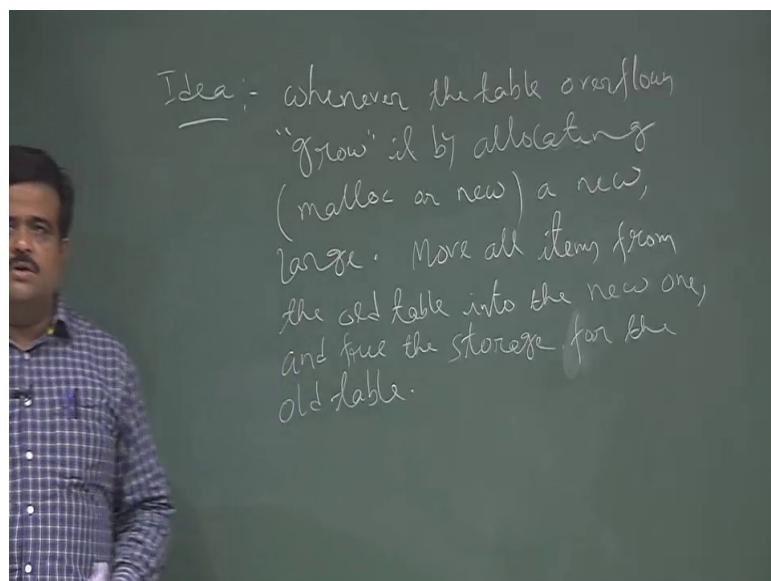


So, this is the question. How large should a hash table be? So, I mean how large you should make our table. So, what is the idea? So, we want to make the hash table as big as we can. So, that there will be no overflow. So, the idea is, make the hash table large enough so that it will not overflow, otherwise it will be inefficient. And it should be try to make it as small as possible, but the question is how large? So, this is a what we want. So, how large? Whether suppose in C language we sometimes we do not know the size of the array. Now if you want to allot some statically then sometimes, we do like this int, array, so 100. So, in C language sometimes we declare the our array like this.

We declare up to 100 because we think 100 will be the maximum size we are going to give the input, but that is the large enough, but suppose we are going to input say only 5. So, that is the wastage and moreover suppose we are going to input say more than 100, at the runtime

we decided our data size is more than 100, soon there is a issue. So, that is the question I mean. So, statically this has no solution I mean we cannot make it so small because in that case overflow and we cannot make it so large I mean large enough so that it will not overflow because in that case we are just wasting the memory. So, this it has no static solution, but there is a dynamic solution, the solution is the dynamic table. So, solution of this problem is dynamic table. So that means, whenever there is a overflow we allot a new memory. So, that can be done by using Malloc or Calloc. So, the solution is dynamic table allocation.

(Refer Slide Time: 04:03)

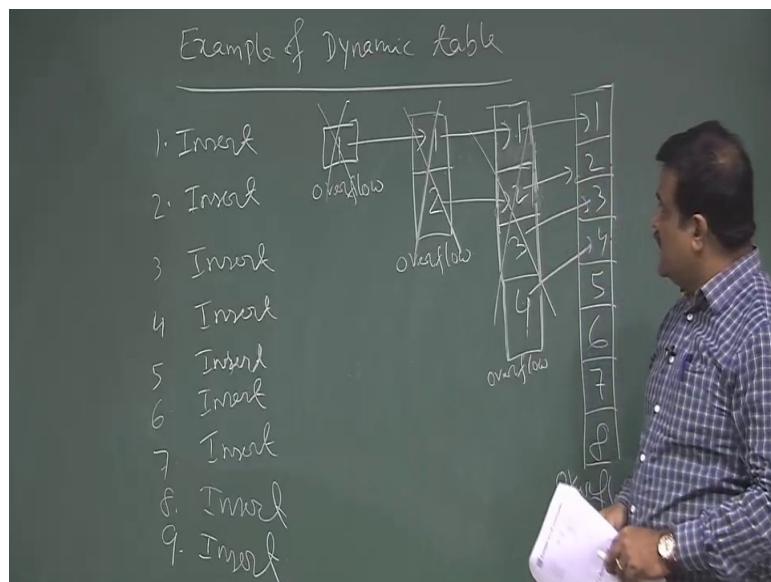


So, the idea is, whenever table overflow, we just grow the table. We grow it by allocating a new larger table, by allocating and this allocation we can do using say in C we know the Malloc or Calloc. So, Malloc call or in java new is the call. So, we are allocating new larger table.

So, whenever that our table is having overflow then we allocate a new larger table which is larger than the current table. And then we shift all the current content to that table. This way we continue and then again we can start inserting and again if we have a overflow then again we will allot a larger table than the current table and again we move all the content to the this table and so this way we continue. So, this is a dynamic allocation to a new larger table and then what we do? We move all the items from old table to new table. From the old table into the new table and we free the storage of the old one. We have to free it otherwise it will be gangling difference. So, we have to free that storage for the old table.

So, this is the idea, this is the dynamic allocation. We want to analysis this. So, what is our insertion time or so let us take an example. So, let us take an example of dynamic table. So, we start with a smaller table, I mean reasonable size table then we start inserting the item or inserting the element and then when we found the overflow then we have to allot a new table which is larger than the old current one and then we shift all old item to this new table and still new table has more space to accommodate more items then we keep on insert the new items in the new table. And then again if we have a overflow in this new table we again allot another table which is larger than the current table like this we continue.

(Refer Slide Time: 07:53)



So, let us take an example of this dynamic table strategy. So, let us take an example of dynamic table. Suppose we have a table with size 1 and suppose we insert a item. So, this is a first item and then suppose, we insert the second item. Then this is overflow because there is only one space, then what is our suggestion our dynamic strategy? Then we have to allot a new table which is larger than this. So, suppose we allot a new table of size 2 now and now we have to shift the content of the old table to new table. So, we shift this 1to here and then what we do? Then we insert this item number 2 here. So, this way we continue. Now again suppose we are going to insert another item. So, again there is overflow here. So, since there is overflow then we have to allot a another table. So, another table must be larger than this table. So, our strategy is we will just have a double of this. So, this table is 1. So, double this table is 2. So now, we are going to have a table of size 4. So, this is a table of 1, 2, 3, 4. So, this is a table of so 1, 2, 3, 4. So, this is the new table we allot. So, this can be done in if we

are using C language we will just use Malloc or Calloc and if we are using java we will have to use the new to have this space.

Then what we do? Then you have to shift the old table content here and you have to free this. So, this old table needs to be freed so that this memory space can be used later on. So, this is the, now suppose this insert we have to do. So, this is 3. So now, we are going to insert another item, insert now we can put it here. So, we are thinking that every time we have to have a new table, it is not. So now, we have a space to insert this item, but now if we are going to insert 5, number 5 item then we have a problem because now this will be a basically overflow, this will be a overflow. So, what is our strategy? Our strategy is to allot a new table with the table size double of that so; that means, the table size will be just 8. So, this will be just 8; so 1, 2, 3, 4, 5, 6, 7, 8 maybe. So, let us check 1, 2, 3, 4, 5, 6, 7, 8. So, we have a new table with table size 8. Now what is our next job? Our next job is to move the content of the old table to the new table. So, we just move this content 1 here, 2 here, 3 here, 4 here and now we are going to insert this new item 5, fifth number item here, there is no issue.

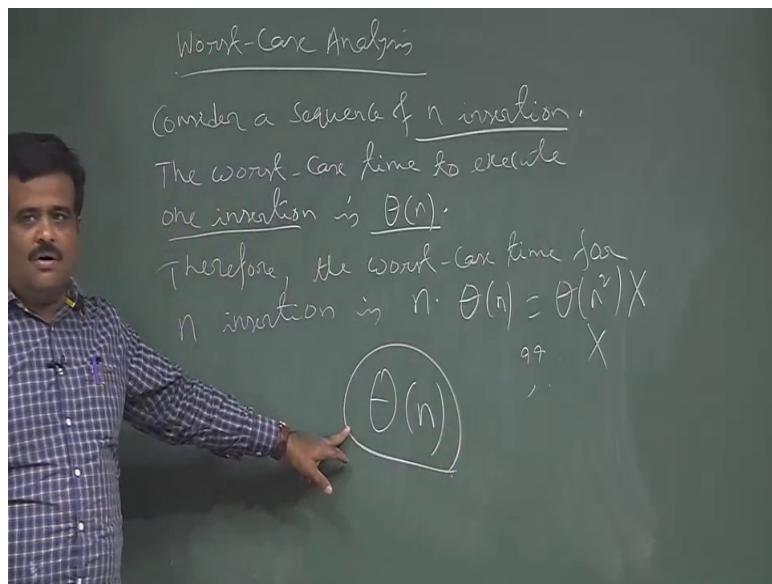
Now, we will keep on inserting the item up to say tables item number 6 we can insert. So, 6 will be inserted here even item number 7 can be insert. So, here we can keep on insert. So, up to 8 we can insert in this bigger table, but again at the ninth stage there will be overflow and once there will be overflow then we need to have a table of size we need to allot that. So, this has to be free after creating this after shifting this item. So, this has to be free, I mean free means we have to free this space.

So, that we can use this space for other purpose, so now, this is the strategy. So, after 8, 8 will be here, 8 insert is no issue there will be no overflow; so 8. So, what is the insert time? Insert time is, so for 6 just 1, for 7 if you are inserting that number 7 item 1, number 8 item 1, but what about number 9 item? If you are going to insert the number 9 item, when we are going to insert number 9 item then there will be overflow. So, once there will be overflow then, we have to have another table of double size. So, there has to be a table of size 16 and then we have to copy all these item there. So, that will that will increase our time complexity, but that is not for all the elements all the items, that is for few items. Most of the items are we are just inserting, we are not creating the table for most of the item, when n is equal to. So, we will come to that analysis for most of the item we are just having only one order of operation, but few item are creating that is for 9.

So, we have to create another table and we have to move each item there from the existing table. So, that will cost us if there are n items that will cost an order of n times.

Anyway let us analyze this method. So, what is the time complexity of this method? So, the idea is we just dynamically allocate the space memory. So, this is not a static allocation. So, we have to go for dynamic allocation because we do not know the size in hand, we do not know how big we should take the table size because we do not know the number of item beforehand. So, static allocation is not possible in that case. So, in that case the suggestion is to go for dynamic allocation or dynamic table, but there we have a issue with the overflow. So, if the item size is more than the table size then that is called overflow. So, once there is a overflow then you have to create a larger table and once you have a larger table you have to move the all existing item on the old table to the largest table and then we have to free this old table.

(Refer Slide Time: 15:43)



So, let us talk about worst case analysis of this. So, worst case analysis of this dynamic table. So, consider a sequence of n insertion, suppose there are n items at some point of time. So, that means, we have n insertion. So, consider a sequence of n insertion. So, in that case, what is the worst case time to execute? Worst case time to execute one insertion is order of n. Why? Because if there is an overflow, if suppose there is order of n items suppose n/2 items are there. Now suppose there is an overflow then we have to shift all the n/2 items into the new table. So, that will cost us order of n. So, that is why the worst case time to execute one

insertion will be order of  $n$  because in the worst case we have to copy or we have to create a table and you have to copy all the existing item to the new table. So, that will cost us order of  $n$  times if the size of that old table is order of  $n$ .

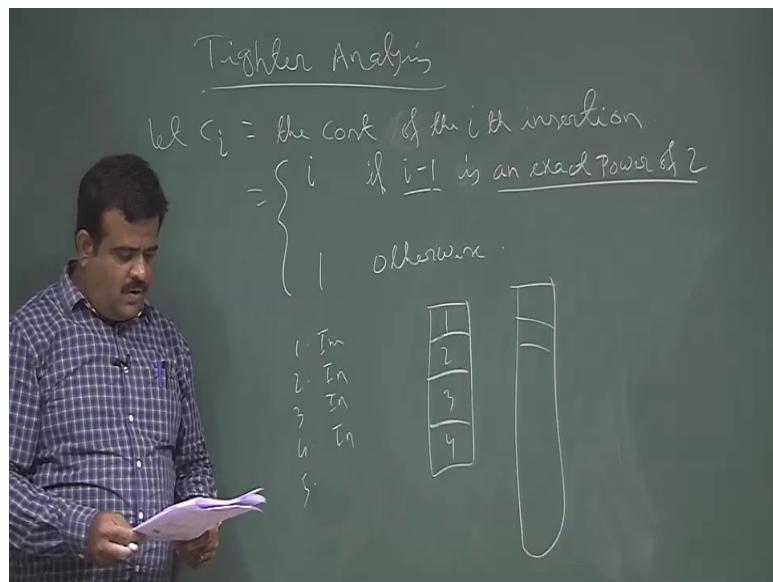
So, from here can you write this? Therefore, the worst case this is for one insertion, one insertion cost us this therefore, can we write this? Worst case time for  $n$  insertion because there are  $n$  insertion, is  $n$  into theta. This is going to be a theta of  $n$  square. So, is the question clear? We are in the worst case we are assuming that when we are going to insert an element we are assuming we want to have a new table. So, we are creating a new table. So, once you have a new table then the old item has to be moved into the new table. So that means, all the old item, suppose there are order of  $n$  old item that has to be moved into the new table. So, that will cost us order of  $n$  times. So, there are  $n$  such insertion.

So, if we theta  $n$  for one execution then there are  $n$  execution so  $n$  into theta  $n$ . So, theta of  $n$  square; no, this is wrong. Why? Because these analysis, we are assuming that every insertion is sort of uniform every insertion is taking the worst case time. It is not because see some of the items are inserting directly, there is no overflow. Only we have to work more if there is a overflow otherwise, we are peaceful. I mean we do not have to worry about that, we just insert the item there if there is no overflow. So, only thing was there is overflow then only there will be more work otherwise there will be less work. Just order of 1, only the if there is a overflow then we have to create a new table and we have to shift the all the old item from this old table to the new table. So, that will cost us theta of  $n$  times, but that is not for all the items. Most of the items will be fitted by not creating the overflow, when the item will create overflow then only we have to do more work.

So, that analysis we have to do and that is called amortized analysis because it is average analysis; on an average, but there is no probability I mean it is an average analysis. What is our insertion time? We want to see and that will be going to the linear time, if there is  $n$  insertion then we can prove this will be this is not order of  $n$  square this will be order of  $n$ . And this analysis is called amortized, we will come to that. So, let us analysis this why it is order of  $n$ ?

So, this is little tighter analysis. So, this is also a worst case analysis, but it is little tighter a little more careful way. So, when there is a overflow then only we have to do the more work otherwise here life is not that much tough.

(Refer Slide Time: 21:07)

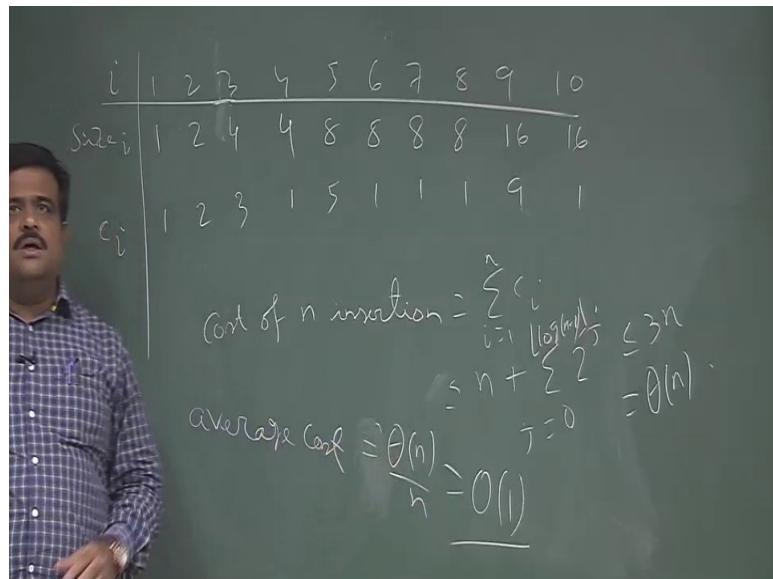


So, g h; tighter analysis, so if we denote the  $C_i$  as the cost for the  $i$ 'th insertion. Let  $C_i$  is the cost of the  $i$ 'th insertion. So, for example, if  $i$  is 1 then, we have one insertion. For example, if we just recall this table of size 4; suppose you have a table of size 4, we have 1 insert, we have 2 insert, we have 3 insert and we have 4 insert then, once you have 5 insert. So, up to 1 insert, 2 insert, 3 insert, 4 insert, there is no issue. So, for these are the insert if  $i$  is  $C_i$  is basically just a 1, just for this 3, but now suppose the our table size is this now. If we have to insert 5 then, it is overflow and once there is overflow then we have to work more. Then what is the strategy? Strategy is we have to make a table size double 8 and we have to shift all the elements over here. So, there are  $i$  elements over here. So, it is order of  $i$  basically it should be  $i - 1$ .

So, 5 is basically  $i$ . So,  $5 - 1 = 4$  this is a exact power of 2 then, there is a overflow. So, similarly for 9, if we have a table of size 8 then we go. So, after 8 table we go up to 8 no problem till then, but once we reach to the nine then we have a issue. What is the issue? Then basically  $i$  is 9, So,  $i - 1$  is power of 2,  $2^3$  basically,  $i - 1$  is 8 then there is a overflow. So, for those  $C_i$ 's for those  $i$  which is basically  $i - 1$  is a power of 2 then we have a overflow. Once you have overflow then we have to have a new table and then you have to copy this old content to the new table. So, that is will cost us order of  $i$  whole item has to copy there including the new item. So, that will be the order of  $i$ . So, this is the exact cost for  $C_i$ ; the cost for the insertion.

So, if we just write it into the table. So, not for all  $i$  we are doing the copying. We are copying when the  $i - 1$  is the power of 2.

(Refer Slide Time: 24:36)



$i$	1 2 3 4 5 6 7 8 9 10
size $_i$	1 2 4 4 8 8 8 8 16 16
$c_i$	1 2 3 1 5 1 1 1 9 1
$\text{Cost of } n \text{ insertion} = \sum_{i=1}^n c_i$ $\leq 3n$ $\leq n + \sum_{j=0}^{\log(n)-1} 2^j = O(n)$	
$\text{Average Cost} = \frac{\text{Total Cost}}{n} = \frac{O(n)}{n} = O(1)$	

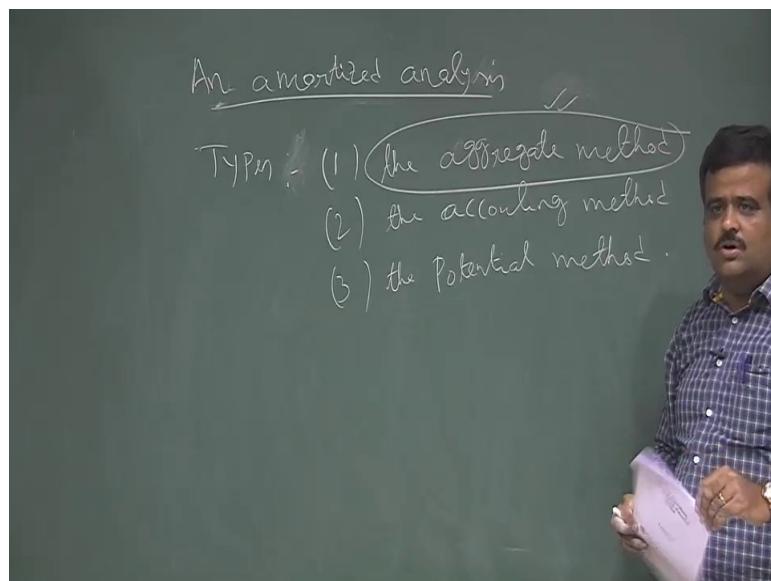
So, let us have the table. So,  $i$  is starting from 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 say then, this is the size of  $I$ , size of the table and this is the  $C_i$ . So, if  $i$  is 1, table size is 1, if  $i$  is 2 table size is 3 then, if  $i$  is 3 there is overflow. So, you have to have a table size of double. Now once we come to the 5 then, there is overflow. So, table size will be 8. So, table size is 8 up to this and then again you have overflow. Table size will be 16, 16 like this and  $C_i$  will be 1, 2, 3 this is the time complexity basically the time for insert  $i$ th item,  $C_i$ ; 3, 1, 5, 1, 1, 9, 1.

See in the most of the cases it is 1, but in the few cases where  $i - 1$  is a power of 2 then we have to have a new table and we have to copy every item there. So, if cost of  $n$  insertion, so it is not for theta  $n$  for all is basically summation of  $C_i$ 's,  $i$  is equal to 1 to  $n$ . Now some of the  $C_i$ 's are 1. So, it is basically less than equal  $n$  plus summation of  $2^j$  and  $j$  is varying from 0 to  $\log(n - 1)$  universally. So, basically why it is  $2^j$ ? Because for this case we are just doing, so for  $i = 5$ , for  $i = 5$  we are just doing 5 times. So, this is the way. For these values we are just having this huge cost otherwise every time here 1. So, that is why it is called amortize analysis because it is not uniform it is not same for all the cases, for some  $i$  it is just a theta 1 that is why this, but there is some  $i$  for which we have more cost. So, we want to average out it. So, that is the amortized way we do. So, if we calculate this will be less than this. So, this is basically theta of  $n$ .

So, this is basically theta of n, cost for n insertion so; that means, what is the average cost? Average cost for one insertion, is basically theta of n/n. So, this is basically theta of 1. So, on an average cost is theta 1, although the n insertion cost is theta n. So, this is not probabilistic analysis there is no probability here. So, this is the exact analysis we did. So, let us write this. So, to insert one item average cost is theta 1 because most of the cases basically is with theta(1), but some of the cases it is basically we need to spend more time because we have to copy the old item previous table to the new table. So, that will cost us more, but that is for few i's, not for all i. So, that is the amortized analysis.

So, let us just write it. So, an amortized analysis.

(Refer Slide Time: 29:13)



Amortized analysis is a strategy for analyzing a sequence of operation to show that average cost per operation is small even though the single operation within the sequence might have the expensive cost because some of the i's has the expensive cost. So, this is an amortized analysis. So, there are three types of an amortized analysis. Types of amortized analysis; one is aggregate method which we have discussed just now. Aggregate method and second one is accounting method and the third one is the potential method. So, the analysis we have discussed for our dynamic table is the method of aggregation. So, we are averaging out. So, this is slightly different from the normal type of analysis we usually discuss because here we are averaging where, in a sequence of execution we are averaging. Average cost is less although few items cost is expensive. So, this type of analysis is called amortized analysis.

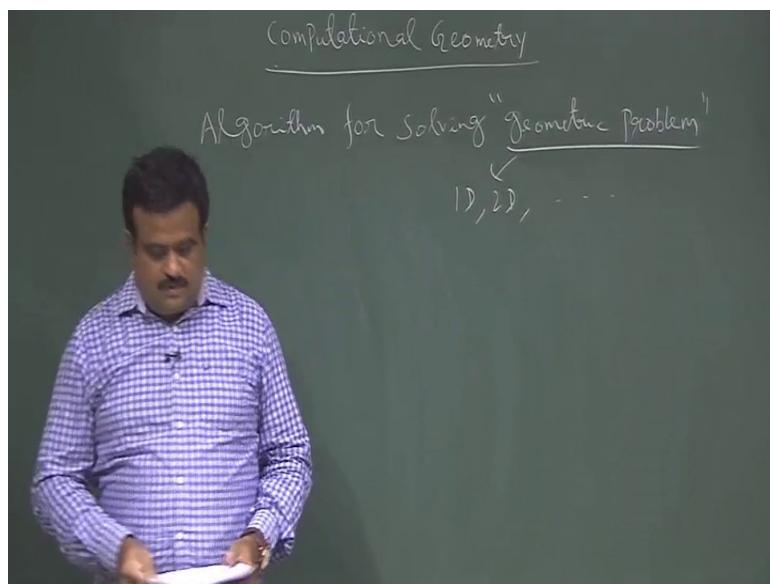
Thank you.

**An Introduction to algorithms and analysis**  
**Prof. Sourav Mukhopadhyay**  
**Department of Mathematics**  
**Indian Institute of Technology, Kharagpur**

**Lecture – 34**  
**Computational Geometry**

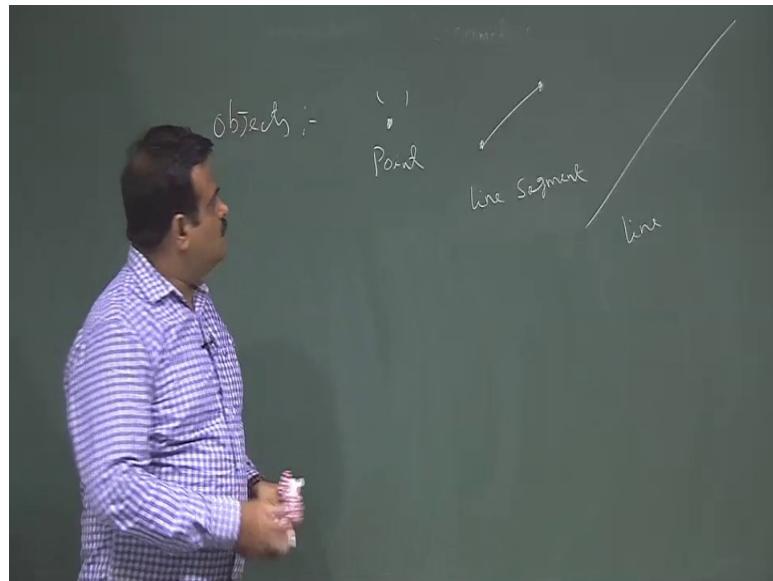
So we start with the computational geometry. So, we will not talk much on this area, just maybe two lectures. So, this is the neat field for geometric problems.

(Refer Slide Time: 00:42)



So, this is the area the computational geometry is basically algorithm for solving the geometric problems, this is the area where we propose the algorithm for solving any geometric problem. So, any geometric problem if we want to solve then if you have to propose some algorithm for that, that comes under this area which is called computational geometry. So, it could be geometric problem in any dimension. It could be 1D it could be 2D or it could be some high dimensional also ok.

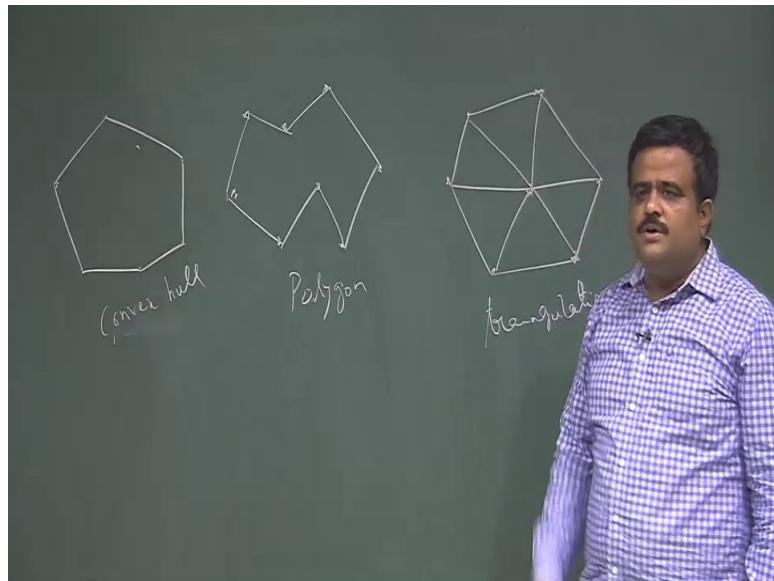
(Refer Slide Time: 01:41)



So, now, to start with let us talk about what are the fundamental objects in geometry. So, what are the objects fundamental objects in geometry. So, point means it could be any dimensional pair point here if this point is on two d plane, but it could be in any higher dimensional point it could be a 3D then we have three compound three dimension x y z it could be some d-dimensional place. So, the point and a line segment this is a line segment and it could be a line. So, these are the six fundamental object in geometry a point a line segment and the line ok.

So, what are the basic structures we can take suppose you have some geometric problem we can discuss suppose our input is some points. So, here we have given some points.

(Refer Slide Time: 02:42)

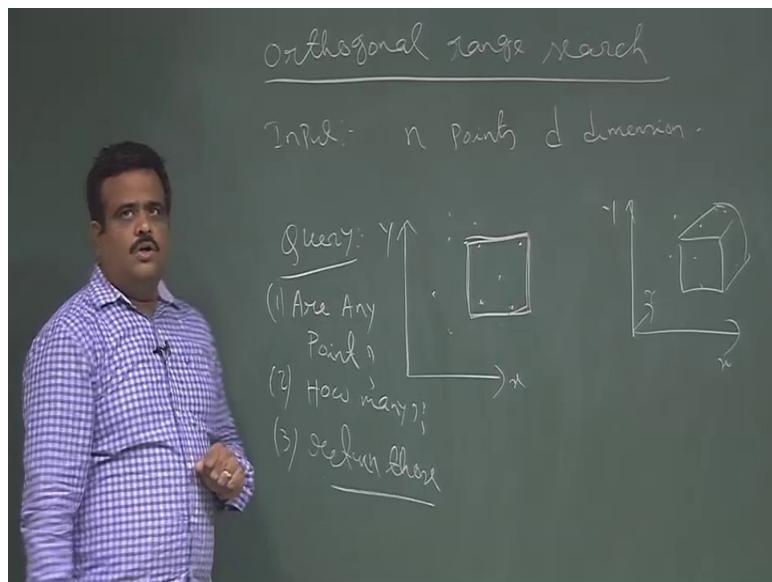


So, these are the points it could be any dimensional points. So, now, maybe we have to think for finding a say polygon using this point this is the smallest intersection which is covering all the points. So, this is basically polygon. So, given a point set, maybe we need to find out the smallest region covering all the points.

So, this is basically polygon problem or maybe we have a triangulation problem like we have these points and we have to form the triangle using these points. So, this is the triangulation or maybe given a points we have to find the say convex hull. So, this is a closed region in which if we take any two points then the line should be inside the region. So, this is called convex hull. So, these are some geometric structure. So, these are some problems we have given the points we need to construct this.

So, this book contained a few more algebra a few more structure on this. So, this is the book for this today's lecture we will follow this books and tomorrow's lecture the other next lecture will follow our textbook. So, let us start with the problem which is called orthogonal range search. So, this book contains some more applications on geometric problem. So, this is the very good book on computational geometry. So, if you have interest you can go through the details on this book. So, our today's problem is orthogonal range searching ok.

(Refer Slide Time: 05:31)



So, the idea is suppose we have  $n$  points. So, we are given  $n$  points. Input is  $n$  points it could be any dimension say  $d$ -dimension and we have given a range orthogonal range. So, the query is. So, if it is a two dimensional point, there we have two dimensional plane  $x$  axis  $y$  axis. So, we have some 2D points and we have some range.

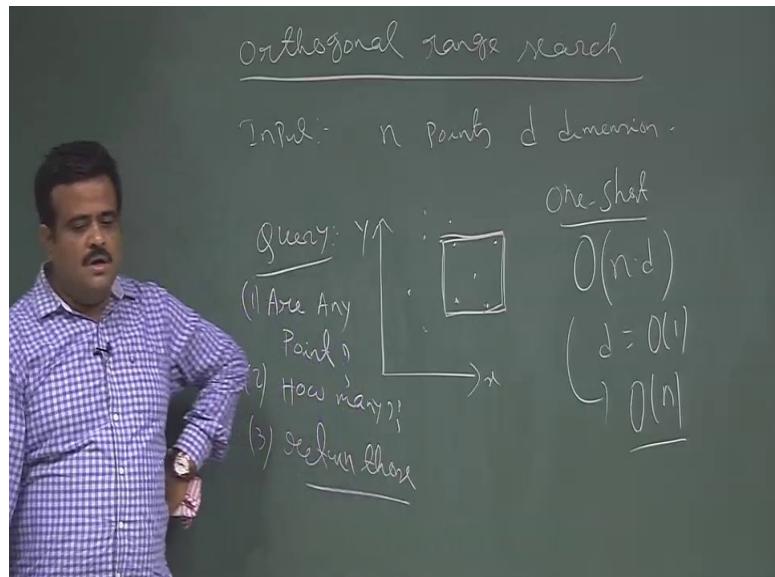
So, we are given a range. Range means the rectangle here in 2D and in 3D it will be like box. So, it will be like this. So, we have a box like this. So, so this is  $x$   $y$   $z$  axis. So, basically we have given a box parallel to the range; parallel to the axis and it could be higher dimension. Also we have 3 types of queries; Is there any point in this range this is the first query. If the answer is yes then how many are there this is the second query. Then tell us the points, if there are  $k$  many we want  $k$  points which are in the range.

So, these are the three types of query one can get. So, is this problem clear? Now, we have given a orthogonal range; that means, it is parallel to the axis  $x$  and then you have to find out the points which are in this range. So, this is called orthogonal range search. So, how we can handle this. So, first query is- “Are there any point” this is the one query then second one is called- “How many” if the answer is yes then return those.

So, now how we can handle this problem what is the net solution of this problem suppose if it is  $d$  dimension. So, how we can check if point lies in this range or not. So, one method is sort. So, we take a point we just see the points dimension. If it is 2D points we check the  $x$  axis  $x$  coordinate  $y$  coordinate and for 2D points we have this rectangle. Then we check

whether this x coordinate by y coordinate fitted in this box or not; lying in this rectangle or not. So, 3D points we have three axis then x y z we check the three values if it is there.

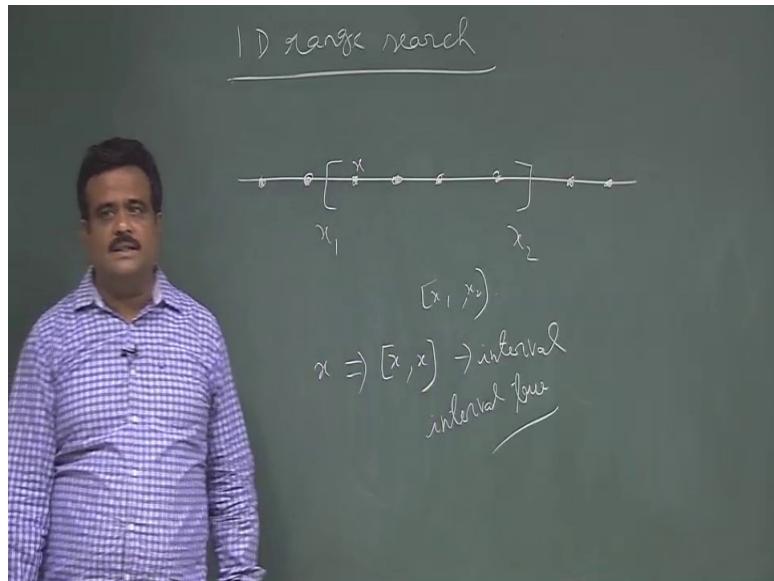
(Refer Slide Time: 09:22)



So, this is the one shot solution. So, what is the time complexity for this it is basically order of we have n points order of  $n * d$ . So, d is the dimension if d is order of one if d is constant then this will be order of n. So, this is the order of n times algorithm ok.

So, now we want to do something better like we want to do some p processing because this safety statistic now we are not changing this. So, we want to have a static data structure on these n points. So, that we can do the search in faster way. So, that is the todays lectures on this orthogonal range search. So, we want to have a static data structure for this. So, let us start with 1D range search.

(Refer Slide Time: 10:27)



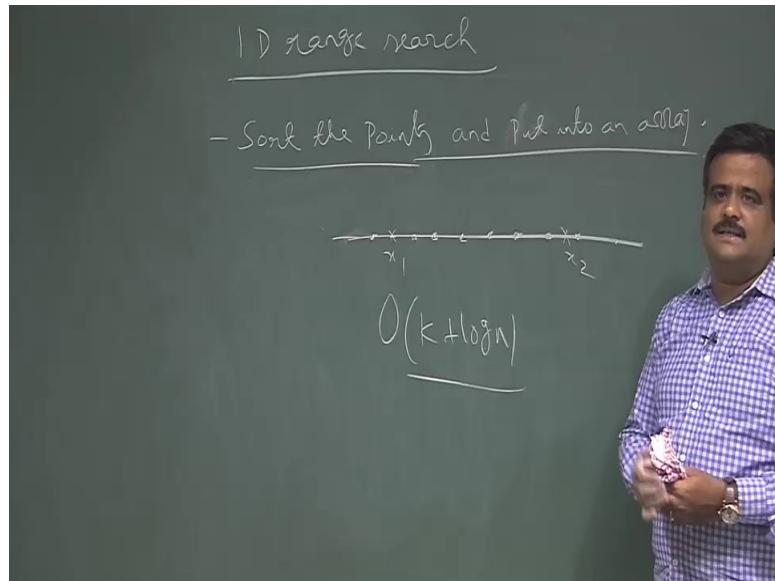
Then we try to extend this for 2D and high dimension. So, for 1D what we have? We have some points on the line one dimensional points.

So, this is the real line. So, you have some points on this line. So, these are the points we have. So, this is a set of n points and this set is say suppose static set n points are there now suppose we have a interval. So, that is the range search only; we have interval say  $x_1$  to  $x_2$  and now we want to find the points lies between here. So, that is the problem. So, how we can do that? Can you use the technique which we know? The Interval Tree. So, can you use that interval tree idea; that is the augmentation of the data structure, can you use that idea?

How we can think of interval search, like how? We are given interval  $x_1$   $x_2$  then we can see how many intervals are overlapping with this; that is the idea. So, how a point can be represented as interval. So, suppose this is  $x$ . So, this is basically interval. So, basically problem is interval search or interval tree. So, if we can find interval tree then this problem will be solved. So, if there are k answers then it will take order of  $k * \log(n)$  to get all the points k points, but this idea cannot be extended for 2D; that is the problem. So, we want to extend this for higher dimension but this cannot be extend for 2D.

So, can you think other technique where we can do some processing and then the query will be much more faster. So, what if we sort this point? We just sort this point and then store it into an array. So, this is the second idea. So, first idea is the interval tree and the second idea is the sorting.

(Refer Slide Time: 13:19)

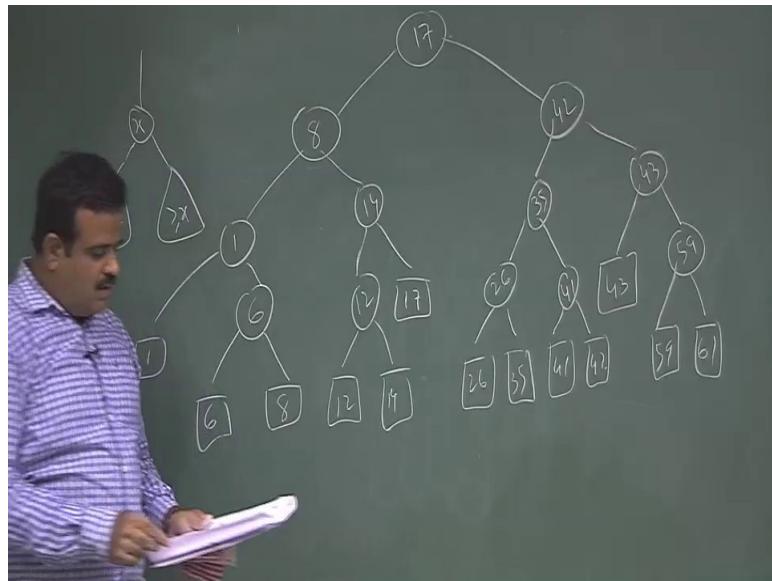


So, what if we sort the points because these points are static points. We can easily sort the points and we can store into an array. So, we have some sorted points. So, we are given two points  $x_1$   $x_2$  say we are given an interval. So, here range is the interval. So, we do the binary search on this point and we do the binary search on this point then we get the position and then all the intermediate points are basically our output. So, this is the idea. But the sorting will take the order of  $n \cdot \log(n)$ , but each binary search will take  $\log(n)$  time. So, we will be basically doing binary search twice.

But we are checking these points. So, it is basically  $k + \log(n)$  where  $\log(n)$  is the time for binary search, but the problem is this idea cannot be extended for 2D because for 2D we have x axis and y axis. So, it is based on which axis we will sort. So, that is the problem. So, this cannot be extended for 2D. So, now we will think about some pre structure whether we can have a tree or if we can have a static data structure which is basically balanced tree then we can try to solve this query. So, that is called 1D range search tree. So, this is the third solution 1D range search tree. So, this is static data structure. So, if you are given  $n$  points, how can we form this tree. So, the idea is if we distribute all the points over the nodes; the nodes starting from the root to the leaf.

Now, here what we are doing instead of that we are putting all the points in the leaf at the leaf level then we form a tree that is the idea. So, how to do that let us have a picture.

(Refer Slide Time: 16:07)

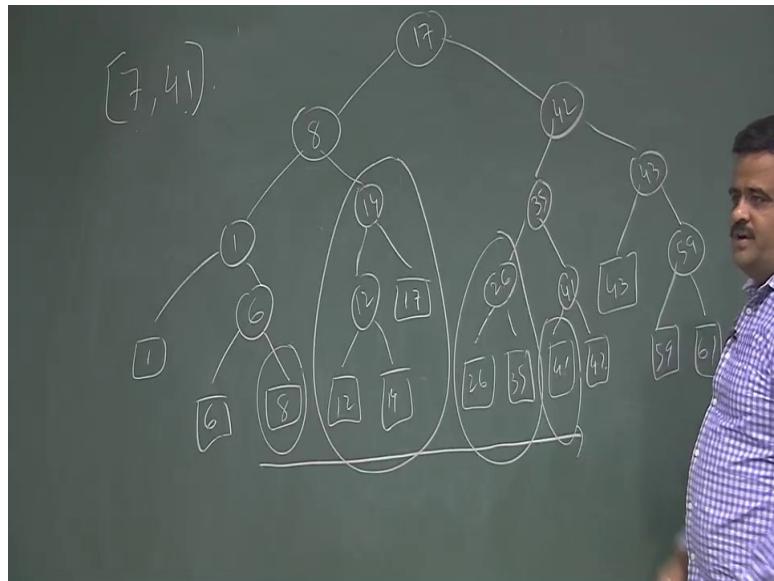


So, suppose these are our points 1 6.. some arbitrary points... 12 14 17 ...then we have say 26 35 41. We are storing into the leaf then here try to form that tree and this tree we want to be balanced search tree.

So we keep merging the nodes one with the other to form a tree and finally we want to have the root. So, this is the way we are going to form the tree. So we are putting every inputs every n nodes in the leaf level and you want this to be a binary search tree. So, for that we need to put a value over here such that it will be a binary search tree; that means, it will be greater than 6 and must be less than 8.

So, we can put any value between 8 and 6, but we are going to put the maximum value of this node. I mean like the idea is to put the maximum node value in the left subtree. So, that is basically we will put it 16 then this is 1, this is 12 this is 14 and so on to form a binary search tree. So, this is a binary search tree. If you take x, now left subtree right subtree. So, this is less than equal to x this is greater than equal to x this property is satisfying. So, this is a binary search tree. In fact this is a balanced binary search tree. So, this is a good news. So, this is called 1D range tree. So, this is a static data structure. So, we want to use this for our query. What are the intervals or what are the points overlapping with a given interval? Suppose given interval is (7 , 41) suppose we are given (7 , 41) interval. So  $x_1=7$  and  $x_2=41$ .

(Refer Slide Time: 19:29)

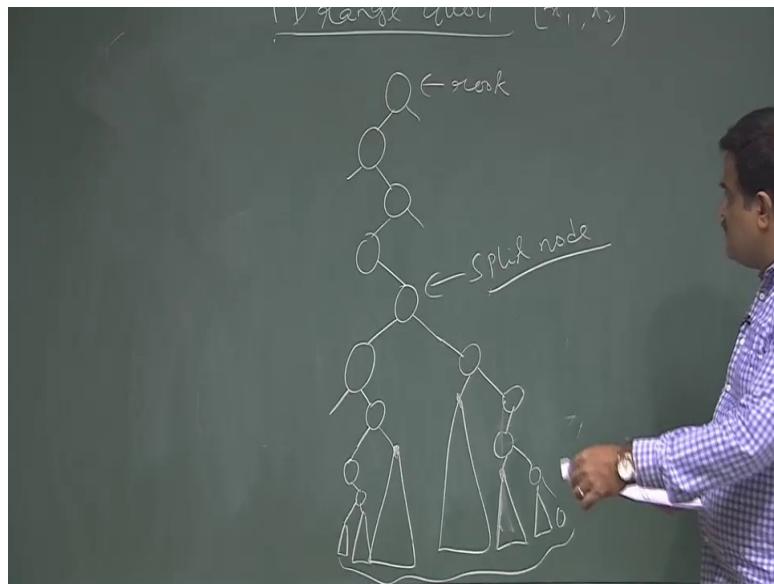


Now, we want to see what are the intervals overlapping with this. So, the leaf in all these subtrees where overlapping occurs is important to us. So, if we just return these roots and if we say that my result is all the leaf nodes in these subtrees where roots are basically 8, 14, 26, 41. So, the leaf of 8 is 8. Leaf of 14 is basically 12, 14, 17. Leaf of 26 is 26, 20, 37 and leaf of 41 is basically 41. So, this is the output. So, instead of returning all the elements separately we just tell that these are the subtrees and this is my answer.

So, the elements are intersection points of basically in the subtree all the leaves. So, you have to go to the leaves. So, how many such subtrees will be occurring? So, basically we are looking for the subtree which we can output and we can tell our answer is our basically leaves in this subtree. So, instead of returning all the leaves we just output the subtrees and then we go to the leaves of this subtrees to get our points which are intersecting with the intervals that is the idea. So, that is what is called 1D range search.

So, we are going to return the required subtrees. So, after getting this tree we just take the leaves of the tree that's it. So, this is the general 1D range query.

(Refer Slide Time: 22:20)



So, we are given an interval  $(x_1, x_2)$ . So, the idea is we start with the root. So, we may find out there is nothing interesting in left. So, you may have to go right. So, we go to the right and then from here we see there is nothing interesting in the left. So, then we go for the right. So, like this we continue. So, we see nothing interesting here and like this we continue until we reach to a node where we find something is interesting in left and something is interesting in the right; that means, there are some nodes which are in the left subtree which are relevant and there are some nodes which are right subtree which are also relevant; that means, which are in the range. So, that node is called split node. So, this is called split node. So, that means, if both the subtree has some nodes of our interest; that means, both the subtree have some nodes which is in the range  $(x_1, x_2)$ .

So, now we have to travel both the way now we start with this the left edge. So, you go to the left edge. So, now, again say we see there is nothing interesting in the left subtree we go to the right now suppose again we go to the left; that means, everything is interesting over here if we are going to the left; that means, this is interesting. So, then again from here suppose we go left and suppose here we go right then all the there will be interesting like this. So, these are the subtrees which are interesting similarly we have to go like this.

So, like this if we again go left then if we go right then all the subtree here will be interesting and like this we continue. So, basically our answer is this subtree. We are going to return this subtree as the answer. So, once we return this subtree then we go to the individual subtree and we get the leaf nodes and that is in the range. So, this is a general idea of range query. So, if there are  $k$  nodes which are overlapping with this, then there will be basically  $\log(k)$

subtree or  $\log(n)$  intuitively it should be  $\log(n)$ , but we can show that  $\log(k)$  subtree will be there. Anyway we will talk about that. So, this will be our output of the range subtree algorithm. So, this is the range subtree algorithm.

So, in the next class we will see the pseudo code of this algorithm. So the general idea is, if nobody is interested in the right then we go to the left then from here if you see nobody is interested in the right we go to the left we go to the right then this way we reach to a point where we will see there is some nodes which are in the range and there is some node in the right which are in the range so; that means, this is the node called split node so; that means, there is something interesting in the left path there is something interesting in the right path.

So, then we continue like this. So, we will talk about the cxvpseudo code in the next class.

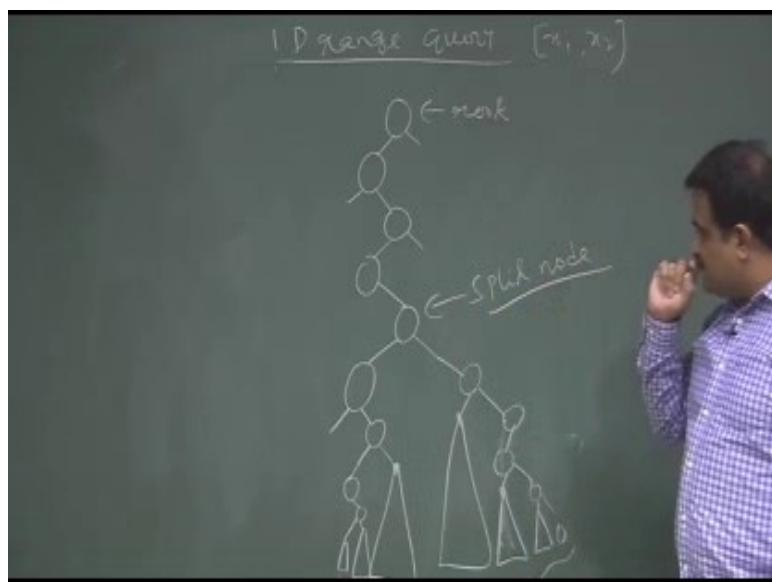
Thank you.

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

**Lecture – 35**  
**Computational Geometry (Contd.)**

So, we are talking about the 1D range search tree. So, this is the continuation of the last class.

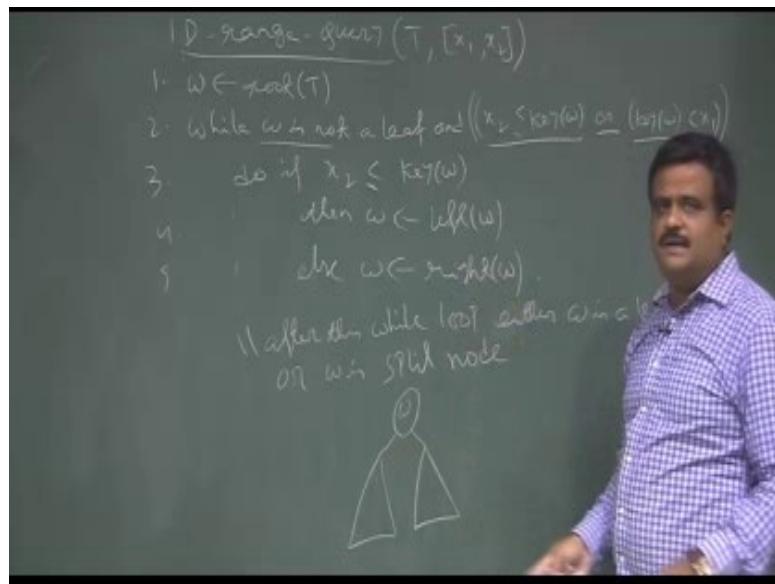
(Refer Slide Time: 00:31)



So, basically the idea is to start with the root. So, we start with the root and then we go until we reach to a split node.

So, split node means there is something interesting in the left path and there is something interesting in the right path; that means, there are some nodes which are in the range and there are some nodes in the right path also which are in the range this is called split node. And then after the split node we will keep on continuing the left and right both sides. So, in the left path, we again check and if there is nobody interested in the left path we go to the right and if we go to the left then everybody will be interested in the right path. So, we have to output this root this whole subtree. So, let us write the pseudocode for this.

(Refer Slide Time: 01:40)



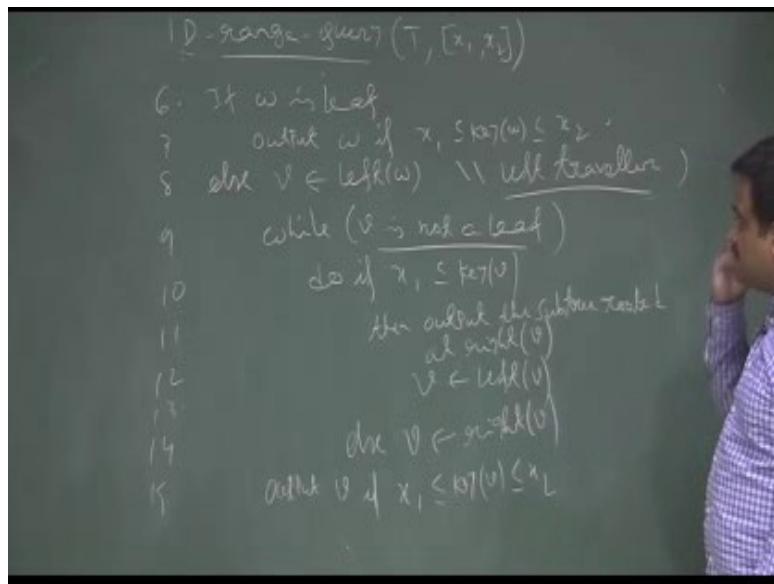
So, this is basically 1D range search or 1D range query. So we have the range search tree which is a static data structure which we form by the given  $n$  points and an interval query. So, this  $x_1$  and  $x_2$  is the interval. So, the root of the tree  $w$ , then we will continue until we reach to a split node then while  $w$  is not a leaf and  $x_2$  is less than key of  $w$  or key of  $w$  is less than  $x_1$ .

So, either  $x$  is leaf and if it is leaf we stop otherwise, we check whether this is a split node or not. So, that means if this is the root, and we are looking for  $x_1 x_2$  then we are looking for the points which are overlapping with  $x_1 x_2$ . Now if  $x_2$  is less than key of  $w$ ; that means our interval is completely in the left side then we go to the left part.

Otherwise if  $x_1$  is greater than key of  $w$  then our interval will be on the right side, so we have to search for this part. So, that means, it is not a split node. So, either way you have to go so that is the idea. So, if this is the case then we have to go either to left or right depending on whichever is the matching.

So, this will continue until we reach to a leaf and we stop otherwise it reaches to a split node. So, after this while loop we reach to a split node. So, after this while loop, so either  $w$  is a leaf node or  $w$  is a split node. So, if it is a split node then we know. So, if  $w$  is a split node; that means, we know something is interesting here. So, then we will do travel both the way but in the algorithm we will show the right part only; 1 part and left part is the similar.

(Refer Slide Time: 05:49)



So; that means, w is a split node now if w is a leaf then we will check the key of w and if it is matching then output the leaf; that is, we output the w if  $x_1$  is equal to key of w is less than equal to  $x_2$ . Otherwise if w is not a leaf; that means, w is a split node which is not a leaf implying that w has some child; left child or right child or both. So, suppose then we take its v is basically left of w. So, this is the left traversal.

Similarly, we have to do the right traversal also. So then we continue, while w is not a leaf, do if  $x_1 \leq$  key of v sorry v is not a leaf if  $x_1$  is less than equal to key of v then. So, if  $x_1 \leq$  key of v; that means, this is our v. So, we are looking for  $x_1 x_2$  now we know that something is interesting over here now if 1 is less than equal to key of v then we have to go to the left part and we will output on the right part, that is the idea.

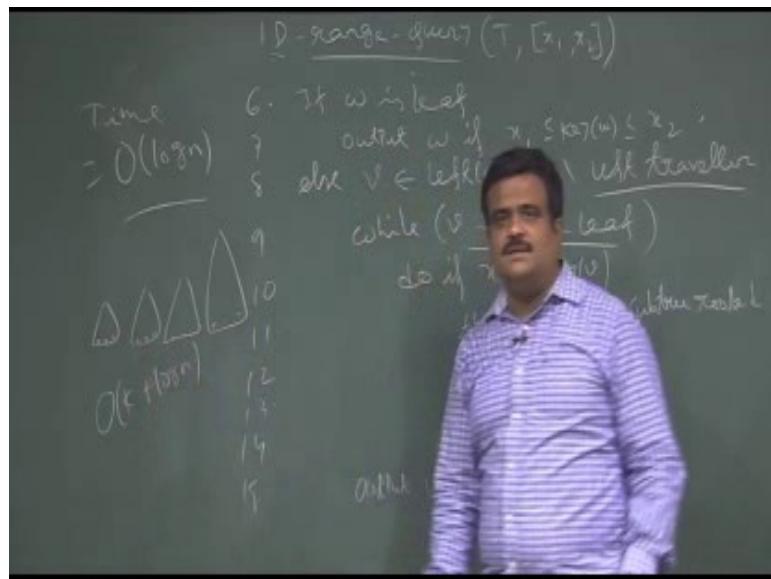
So we output the whole subtree rooted at right of v and we go for left again; left of v else we go for right of v until we reach to the leaf node or we reach to the situation where we have something interesting in the left then we have to output on the right subtree. So, that is the idea. So, finally, once we reach to the leaf then we check. So, this is 9 10 11 12 13 14 15 maybe.

So, finally this means will be released to a leaf then we check the output to see, if  $x_1 \leq$  key of v  $\leq x_2$ . So, this is the left traversal similarly you have to travel in the right part of the subtree. So, right traversal you need to do. So, now, we have to analyze this. So, what is the time complexity of this? So, basically our tree is balanced. So, there are n nodes

and it is basically  $\log n$  height. So, basically what we are doing- we are traversing the tree, for each, and we are outputting the subtree.

So, in any case we are just start from the root and we are going to the leaf. So, it is basically taking  $\log n$  time.

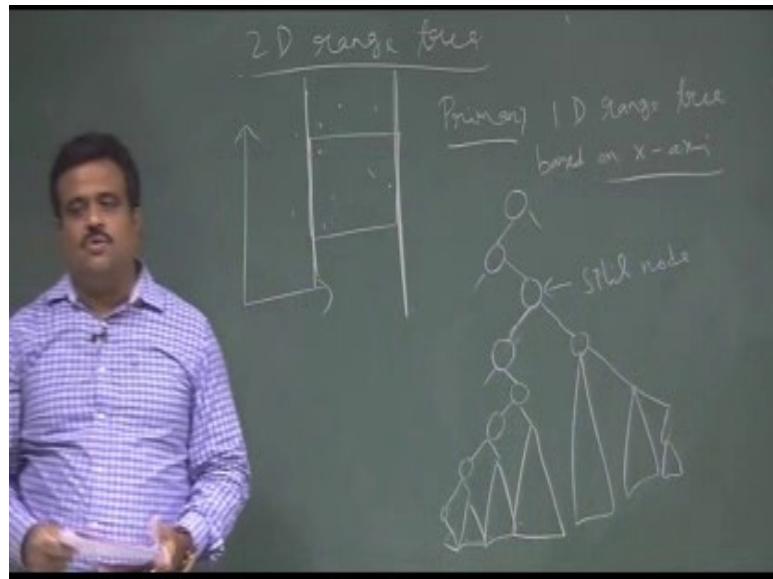
(Refer Slide Time: 10:04)



Time is  $\log n$  for this traversal to go from the root to the leaf and for each time. So, now, after getting the output (the subtree) we need these leaves; they are the answer. So, to get these leaves we need to spend order of  $k$  again. So, this is basically the time complexity for requiring output; that means, we need to get these leaves. So, basically  $k + \log n$  is the time complexity for this. So, to store this we need to have order of  $n$  is the space and to form that tree we need to have order of  $n \log n$  that is the space complexity and this is the time complexity here.

Now the advantage is we can extend this for 2 dimensional or higher dimensional. So, that is the idea. So, that is the 2 D tree, 2D range tree.

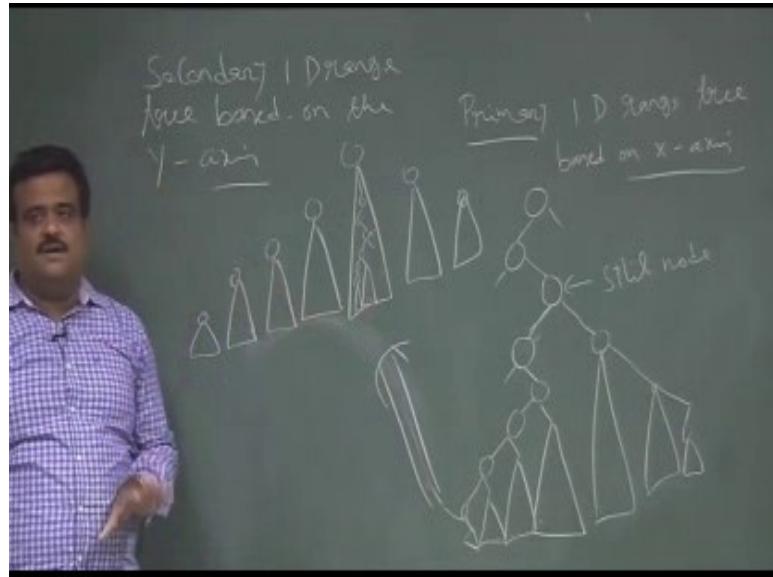
(Refer Slide Time: 11:25)



So, the question is how we can extend this for 2D. So, basically now we have seen the 1 D range tree now how we can extend this for 2 D. So, what is the way? So, for 2D range search tree we have a 2 dimensional points and 2D range search means we have a rectangle. So, what we do? We first take a 1D range search. So, basically the idea is we store the points. So, we store the point in 1 D range search tree. So, that is the primary tree. So, primary 1D range tree is the primary tree based on x axis value. So, we start with the root, this is our 1D range tree based on the x coordinate. So, we have this root and we start with the root. We go left then we go right until we reach a split node.

So, once we reach to a split node then again we continue. So, another secondary range tree will be stored and that will be based on the, secondary 1 D range tree. This is also a pre processing space based on the y coordinate.

(Refer Slide Time: 14:18)



So, we have  $n$  points each point is having  $x$  coordinates  $y$  coordinate. So, on the  $y$  coordinate on the  $x$  coordinate we have this range tree and we got this subtree and on the  $y$  coordinate also we have the range tree. So, lots of data structure is going on. So, on the  $y$  coordinate also we make the range tree and there. So, we have similar kind of structure for the  $y$  coordinate.

So, these points are based on the  $y$  coordinate. So, we have the same size tree, but these are based on the  $y$  coordinate. So, this is the part of the range tree in the  $y$  coordinate, so their size is same because this number is same. So, now for each of these we do the 1D range search query. So, we now we have a  $y_1$   $y_2$ ; we have  $y$ -interval. So, we do the range search. You start with this root. We go until it is a clear split node. So, this is the split node. Then we continue as before. So, these are the points where  $y$  axis is also matching. So, these are the points we are looking for. So, these are the points where  $y$  axis is also matching. So, that is the idea.

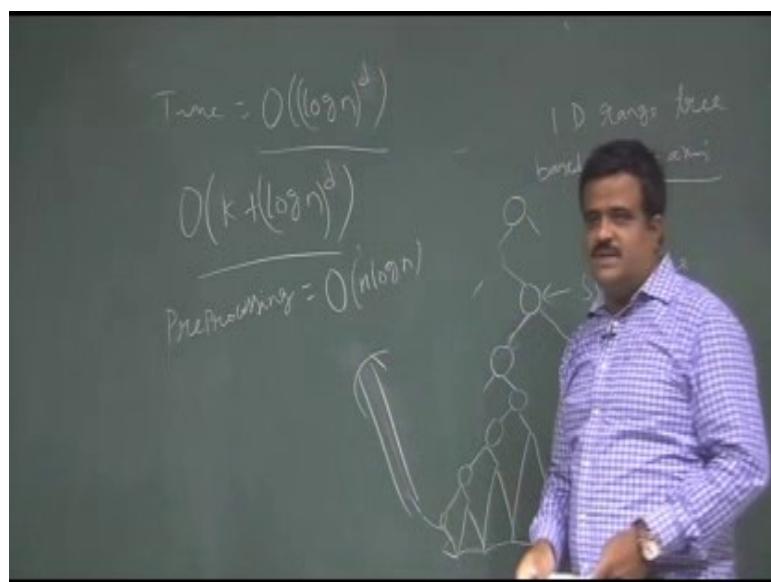
So, basically we have 2 ranges. So, basically we are having two data structures; 2 range trees one is based on primary range tree on the  $x$  axis. So we store in a tree and then the second one is based on the  $y$  axis we store energy for all the points because we do not know which points will come into this. So, lots of data structure is going on.

So, we first apply our 1D range search query on this  $x$  axis and we get some subtree. So, this is the similar data structure we have with the same nodes here based on the  $y$  axis and then we have to do the 1D range search based on the  $y$  coordinate and that will give us basically

the matching of the y coordinate also. So, that will give us the points which are in the rectangle. So, that is the idea. So, this can extend for further dimension. So, on this if we have x y z axis we have another primary range tree and there we have to search. So, this will match x axis y axis z axis. So, this is the idea.

So, what is the time complexity for this? So, for this we are basically  $(\log n)^2$  because this is the  $\log n$  and here again if we need to go for  $(\log n)^2$  for each suppose there are  $\log n$  subtree. So, time is basically  $(\log n)^2$ .

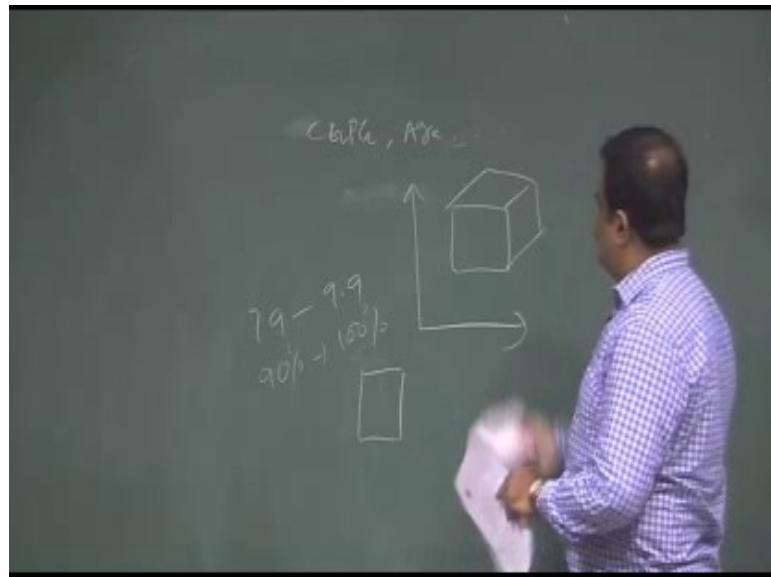
(Refer Slide Time: 18:13)



So, after knowing the subtree if we have to report all the elements then it will be  $k + (\log n)^2$ . So this is output sensitive if  $k$  is the number of points which are lying in that rectangle then it will be  $k + (\log n)^2$  and pre processing time is basically same as earlier. It is basically  $n \log n$ . So, now, if you want to generalize this for  $D$  dimensional then query time will be taking 2 to the  $D$  because we have  $D$  many of such dimension and in each dimension we have to search it.

So, we will just quickly look at another problem; geometric problem and this has lots of application in the database system. So, this range search tree has lots of application. Suppose we have  $D$  dimensional data ( $n$  data samples which are  $D$  dimensional). Suppose student record is the data. Each student has many attributes, so  $D$  dimensional marks or CGPA like this. So, there are many dimensions like the marks in algorithm marks in some mechanics or something. So, now these are the dimensions. So, based on these we store it.

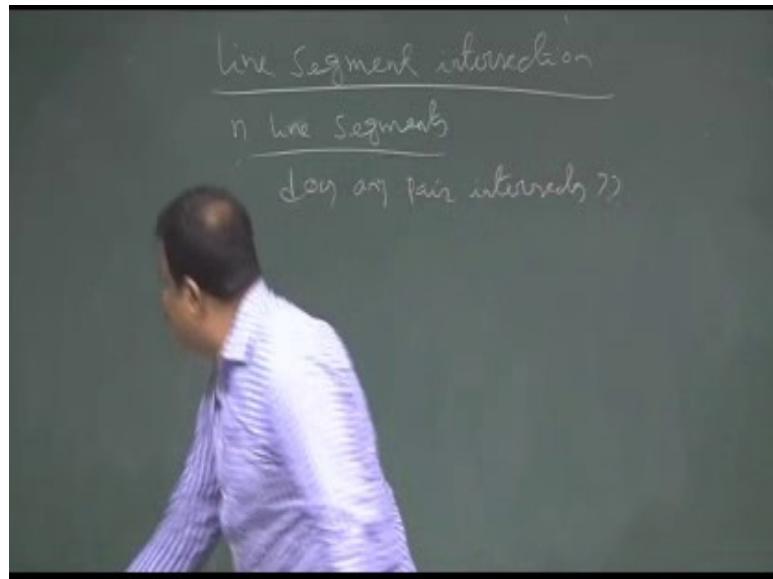
(Refer Slide Time: 19:50)



Now suppose we have a query like this we just have points like this  $x_1 x_2$ . Then, suppose it is say 3D then we have a range like this and we want to see how many points belong to this so that means, we want to fix that. So, Microsoft wants database of the all students name of the students whose CGPA lies between 7.9 to 9.9 and whose algorithm marks basically lie between 90 percent to 100 percent is the 2 dimensional range search.

So, we have to get all the students who are matching with this range. So, this is a rectangular search 2 D search. So, then there are lots of application of range search tree. Anyway let us have a quick look at another problem which is called a line segment intersection problem.

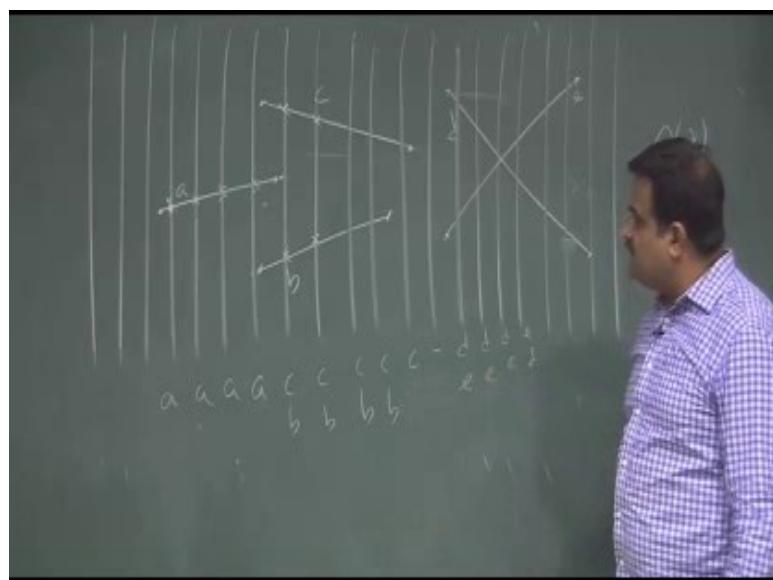
(Refer Slide Time: 21:17)



So, we have some lines, we have few line segments and we need to find out any 2 segments are intersecting or not. So, does any pair intersect that is the public.

So, for example, if we have a line segment say this is the line a and say we have a line like this. Suppose we have line segments a b c d line segments. This is line segment not line, line has no starting point and ending point, but segment has starting point this is e line.

(Refer Slide Time: 22:08)



So, suppose these are our inputs. there are n line segments now we want to know is any pair of this segment intersecting. So, that is the idea. So, what is the near approach? So, we can

take any 2 segments and we can just check by coordinates wise whether they are intersecting or not. So, that will take order of  $n^2$  because there are  ${}^nC_2$  pair. So, just take any pair. So, we will check, but we want to do something better that is called sweep line algorithm. So, basically what we are doing in sweep line algorithm, we are having a vertical line. So, we just move this vertical line this way.

So, while moving what we observe. So, we observe the ordering of this number ordering of this intersect. So, this is the intersect point. So, we are interested in these intersecting points and if we maintain a dynamic data structure with this intersecting points and this we are putting in the order of the value of the x axis y axis.

So, we maintain a dynamic set ace for this storing these numbers this y value. Now, so when this value will change? It can change in the 3 condition this value will change either when a new line is entering; that means, a line segment has joined so that way and when a line is finished then it will change.

Otherwise it will change when there is a intersection. So, either this change will occur either a new segment is joining or a segment is leaving or there they are crossing they are intersecting. So, we have the interest here. So, whenever a change is occurring we will see the neighbouring node; we will see the neighbouring lines and we will check whether they are intersecting or not. So, for this dynamic set we need to maintain a data structure. So, what we can do we can use the red black tree which is balanced and that will be done in order of  $n * \log(n)$  time. Anyway this is the rough idea of this sweep line algorithm. So, if you have interest you can look at the textbook for the details of the pseudocode.

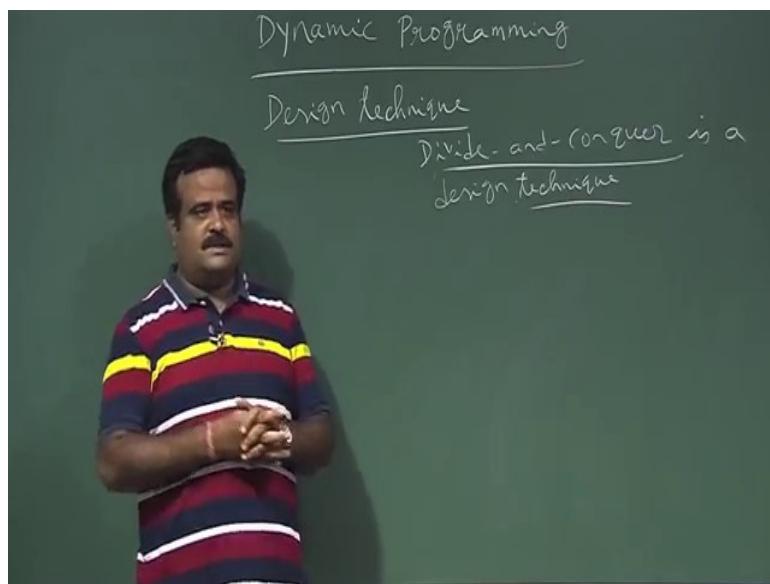
Thank you.

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

**Lecture – 36**  
**Dynamic Programming**

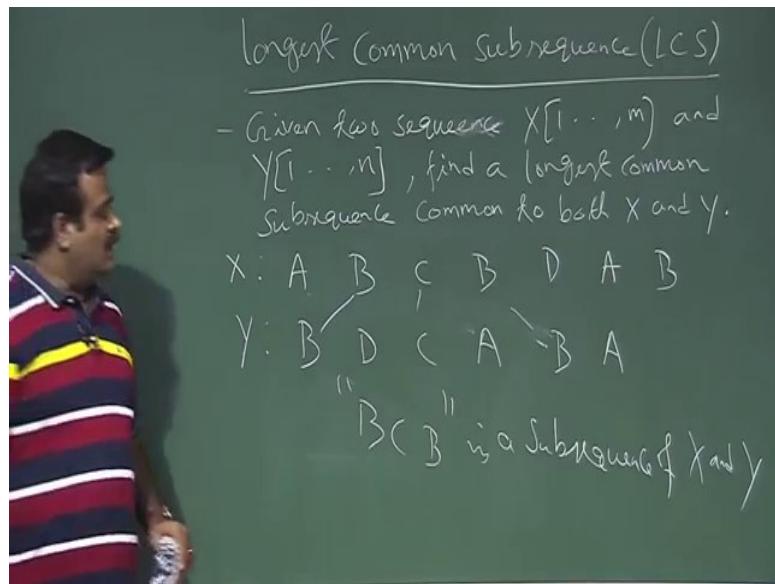
So, we talked about dynamic programming. So, it is a design technique like divide and conquer.

(Refer Slide Time: 00:29)



So, we know one designing technique is divide and conquer technique. This is a design technique we know. So, we have seen merge sort, quick sort all that divide and conquer technique where we have a problem of size  $m$  and we reduce the problem into subproblems of lesser size and that is the divide step and in the conquer step, we solve recursively this subproblems and once we get the solution of the sub-problems then we combine the solution of the sub problem to get the solution of the whole problem. So, this is the divide and conquer technique. Dynamic programming is also a design technique. We will discuss this through an example through a problem which is called longest common subsequence problem.

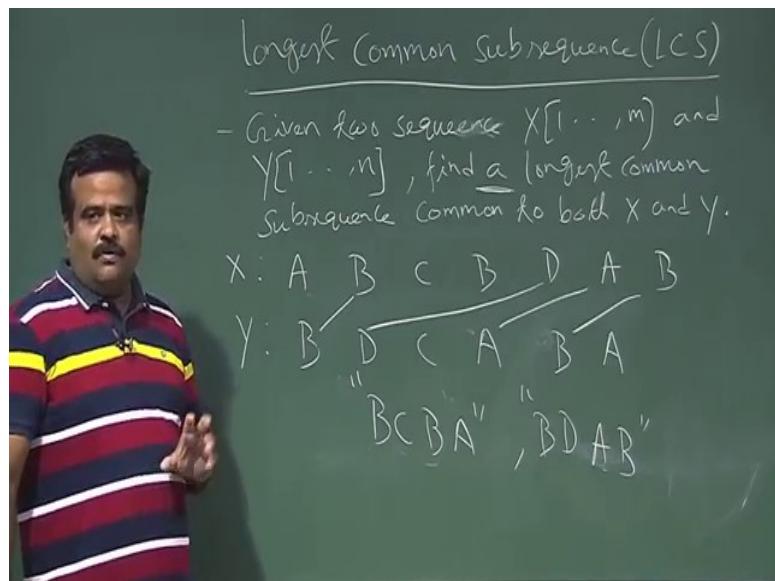
(Refer Slide Time: 01:54)



So, we will learn this technique through an example which is called longest common subsequence problem, LCS. Problem is we have been given 2 sequence X and Y. X is of size m and Y is of size n. So, we need to find the longest common subsequence, subsequence which is common to both X and Y. Let us take an example suppose X is a sequence ABCBDAB and Y is a subsequence BDCABA. So, suppose these 2 are the given sequence.

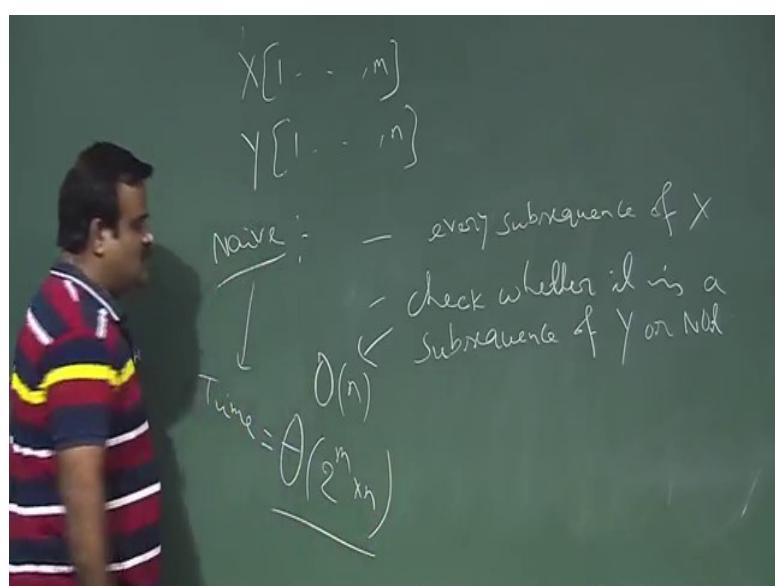
Now, we need to find the longest common subsequence. First of all, we need to find a subsequence. So, for example, AB is a subsequence of both X and Y of length 2. BCB is also is a subsequence of X and Y, but this is of length 3. now we want to see whether we have any subsequence of length 4.

(Refer Slide Time: 04:45)



So, let us try that. B, C, B, A; this is a subsequence of length 4. Now we can check is there any subsequence of length 5. We can check that there will be no subsequence of length 5; so that means, the length of the longest common subsequence is 4. We may get another subsequence of length 4. That is why it is a longest common subsequence not the longest common subsequence. So, there could be many subsequence of line length 4, but the longest length is unique.

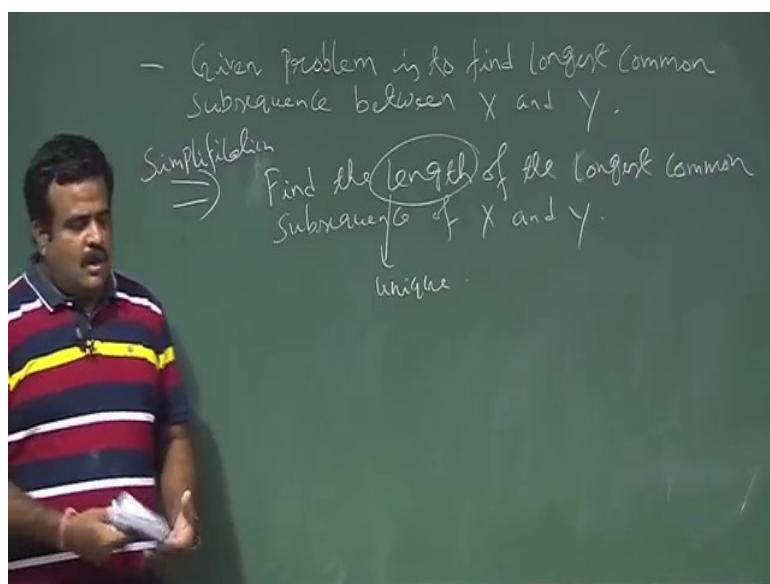
(Refer Slide Time: 06:37)



So, how to find a longest common subsequence from a given 2 sequence X and Y. Naive approach could be, we take every subsequence of X and check whether it is a subsequence of Y or not. So, to check whether it is subsequence of Y or not, it will take order of  $n$  time because this is the length of the Y sequence. Now how many subsequences we can have from X. That is the power set basically.

So, basically this approach there could be  $2^m$  subsequences. So, this approach will take order of  $2^m * n$ , exponential time algorithm. This is not acceptable as this is very expensive and not even polynomial time. But this is the brute force method where one can try for all possible subsequence of X and can try to see whether this is a common subsequence of Y or not. So, now, we want to do something better. We want to simplify this problem. So, basically our problem is to find the longest common subsequence between X and Y.

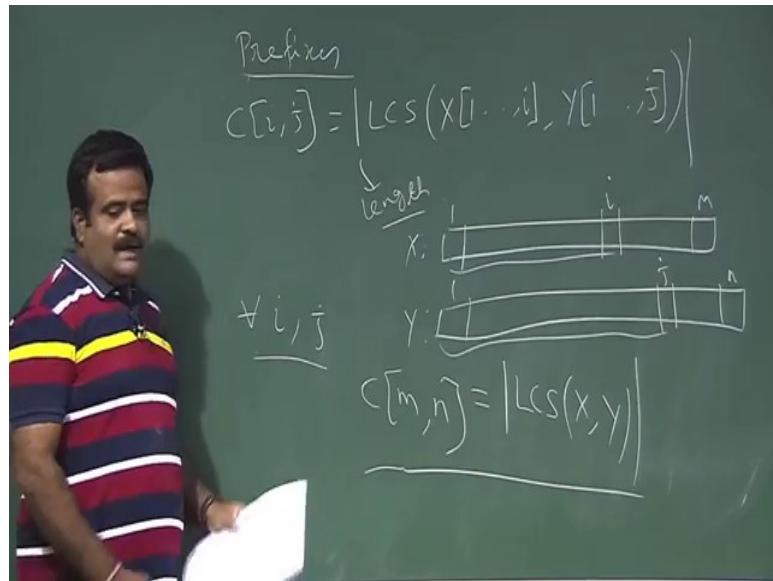
(Refer Slide Time: 09:05)



The given problem is to find longest common subsequence between X and Y. We just simplify this problem. We want to find now the length of the longest common subsequence.

So, we want to get the length of the longest common subsequence of X and Y and that length is unique. So, this is something simplification of this problem. So, we first try to get the length of the longest common subsequence which is unique and then after getting the length from that algo, we will use the dynamic programming technique then from there after getting the length, we will try to find out the longest common subsequence. So, this is the problem. So, now, we want to find the length of the longest common subsequence.

(Refer Slide Time: 11:38)



So, for that we will use some prefix notation. So, we will use some notation on X and Y. So, prefixes. So, we denote the  $C_{ij}$  which is basically length of the longest common subsequence between  $X_1$  to  $X_i$  and  $Y_1$  to  $Y_j$ . This is basically the length. We have a X sequence which is from 1 to m and we have a Y sequence which is from 1 to n. So, basically we take a subsequence of X up to i and Y up to j. So, now, if we can find out this  $C_{ij}$  for all i and j then we are done why because we want to find the length of the longest common subsequence of X and Y.

So, basically we are looking for  $C_{mn}$ . This is basically length of the longest common subsequence of full X and full Y. So, if we can find out  $C_{ij}$  for all i and j then we are done because we can get the  $C_{mn}$  also. So now, this is the definition. We want to find out a recursive formula for this  $C_{ij}$  and that formula will help us to have a algorithm design technique which is called dynamic programming technique. Every dynamic programming technique we have such kind of recursive formula.

(Refer Slide Time: 14:09)

Th: (Recursive formula)

$$C[i, j] = \begin{cases} C[i-1, j-1] + 1 & \text{if } X[i] = Y[j] \\ \max\{C[i-1, j], C[i, j-1]\} & \text{if } X[i] \neq Y[j]. \end{cases}$$

So, this is the recursive formula for  $C_{ij}$ . So, let us write this in a theorem. So, it is telling us  $C_{ij}$  is basically  $C_{i-1,j-1} + 1$  if  $X_i$  equal to  $Y_j$  otherwise, it is maximum of  $C_{i-1,j}$  and  $C_{i,j-1}$ . So, this is the recursive formulation for  $C_{ij}$ . This formula will give us a design technique and that is the dynamic programming technique we will come to that so, but before that let us try to prove this formula. We will prove the first part and remaining part will be similar. So, we want to prove that  $C_{ij}$  is equal to  $C_{i-1,j-1} + 1$  if  $X_i$  equal to  $Y_j$ .

(Refer Slide Time: 16:27)

Bread

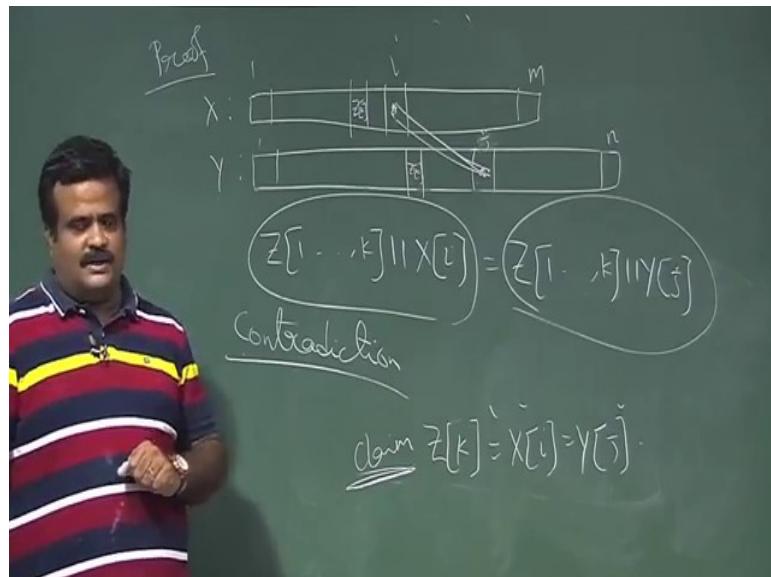
$$\text{let } k = |\text{LCS}(X[1:i], Y[1:j])|$$
$$\text{let } Z[1:k] = \text{LCS}(X[1:i], Y[1:j])$$

Claim  $Z[k] = X[i] = Y[j]$

So, let us prove this. Suppose  $X$  is starting from 1 to  $m$  and  $Y$  is from 1 to  $n$  and  $X_i$  is basically equal to  $Y_j$ . So, if  $X_i$  is equal to  $Y_j$  we have to prove  $C_{ij}$  is equal to  $C_{i-1,j-1} + 1$ . So, let  $K$  be the length of the longest common subsequence till  $X_i$  and  $Y_j$ . Let LCS of  $X_1$  to  $X_i$  and  $Y_1$  to  $Y_j$  be denoted by  $Z$ . So,  $Z_1$  to  $Z_K$  is basically LCS of  $X_1$  to  $X_i$  and  $Y_1$  to  $Y_j$ . So, the length of the longest common subsequence is  $K$  till now and suppose  $Z$  is the one such common subsequence.

We need to prove that that  $Z_K$  is the last common alphabet between  $X_1$  to  $X_i$  and  $Y_1$  to  $Y_j$ . So, how to prove that suppose it is not; suppose the  $Z_K$  is not that equal to  $X_i$ .

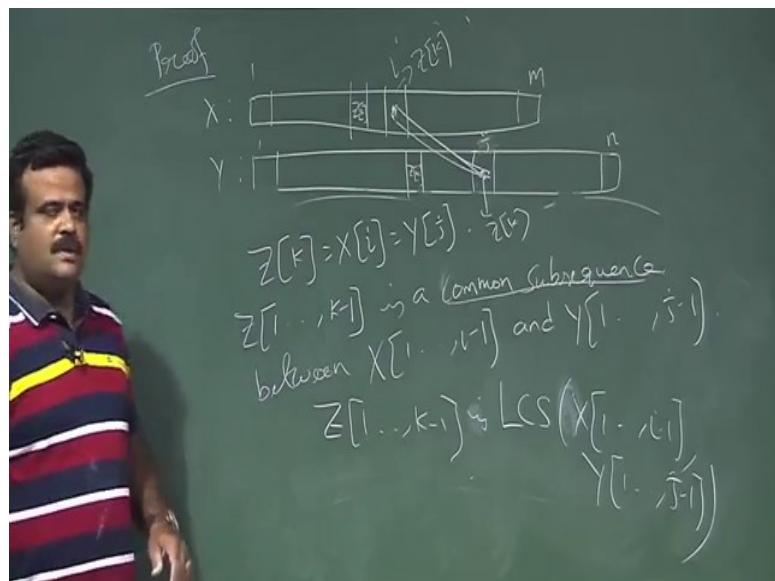
(Refer Slide Time: 19:22)



Since  $Z_K$  is not equal to  $X_i$ ,  $Z_1$  to  $Z_k$  must be the longest subsequence till  $X_{i-1}$  and  $Y_{j-1}$ . Now since  $X_i$  equals  $Y_j$  we have a common subsequence of length  $k+1$ .

So, this is telling us the length of the longest common subsequence is  $K+1$  which contradicts that length is  $K$ . So, this is the contradiction because we assume length is  $K$ , but here we are getting length is  $K+1$ . So, that means,  $Z_K$  is basically the last term.

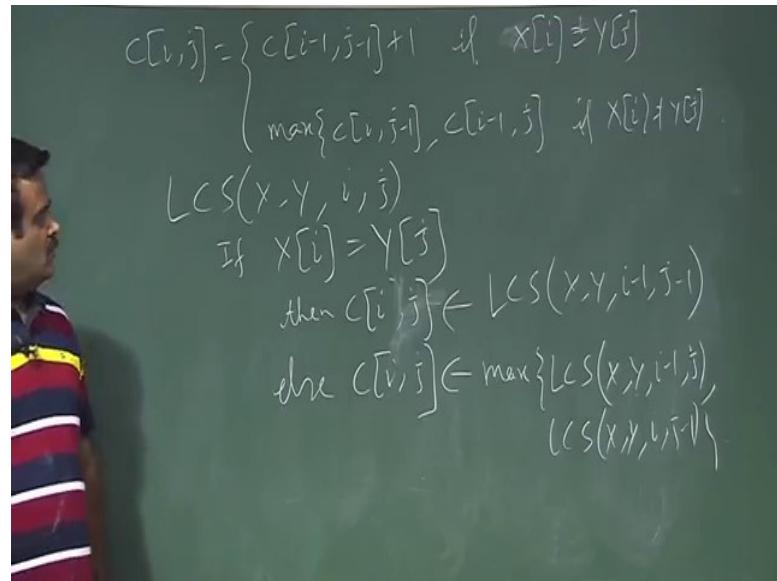
(Refer Slide Time: 20:53)



Now  $Z_K$  is basically  $X_i$  which is basically same as  $Y_j$ . So, now, we take  $Z_1$  to  $Z_{K-1}$ . So, we just remove the last term. So, this is a common subsequence between  $X_{i-1}$  and  $Y_{j-1}$  because we are just removing the last part. This last part is basically our  $Z_K$ . So, we are removing this last part, then this is a common subsequence of  $X_1$  to  $X_{i-1}$  and  $Y_1$  to  $Y_{j-1}$ . If this is not a longest common subsequence, this is of length  $K-1$ ; if there is a subsequence of length  $K$ . So, which will give us a common subsequence of up to  $X_i$  up to  $Y_j$  which is of length  $K+1$ . So, again which contradict the fact that  $K$  is the length of the longest common subsequence between  $X$  and  $X$  up to  $i$ ;  $Y$  up to  $j$ . So,  $Z$  is a longest common subsequence.

So, this is the first part of the theorem and the second part will go similar way. Now, this will give us a recursive formula. So, let us start with the; so, this is one of the hallmark of dynamic programming technique.

(Refer Slide Time: 25:24)


$$C[i,j] = \begin{cases} C[i-1, j-1] + 1 & \text{if } X[i] = Y[j] \\ \max\{C[i, j-1], C[i-1, j]\} & \text{if } X[i] \neq Y[j] \end{cases}$$

$LCS(X, Y, i, j)$   
If  $X[i] = Y[j]$   
then  $C[i, j] \in LCS(X, Y, i-1, j-1)$   
else  $C[i, j] \in \max\{LCS(X, Y, i-1, j), LCS(X, Y, i, j-1)\}$

So, this will give us a algorithm recursive algorithm. So, it is telling us  $C_{ij}$  is basically  $C_{i-1,j-1} + 1$  if  $X_i$  equal to  $Y_j$  otherwise, it is maximum of  $C_{i-1,j}$  and  $C_{i,j-1}$ . So, this is the recursive formulation for  $C_{ij}$ . So, this is a recursive formula recursive algorithm. So, the worst case will be when  $X_i$  is not equal to  $Y_j$ , in that case, we need to compute both term and the size is only in the one index reduce.

So, we will continue this in the next class, where we will talk about how we can use this recursive formula to have a dynamic programming technique. So, you have a hallmark for dynamic programming technique. So, we will talk about this in the next class.

Thank you.

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

**Lecture – 37**  
**Longest Common Subsequence**

So we are talking about longest common subsequence. So, we are given 2 sequences  $x$  and  $y$ .

(Refer Slide Time: 00:24)

You are given 2 sequences one is of length  $m$ , other one is of length  $l$ . And you need to find a longest common subsequence between this. So, for that if we simplify this problem to finding the length of the longest common subsequence, and we have defined  $c_{i,j}$  which is basically prefix is length of the longest common subsequence of  $x_1$  to  $i$  and  $y_1$  to  $j$ . So, this is the length of the longest common subsequence up to prefix 1 to  $i$  after prefix 1 to  $j$ .

Now, we can find out  $c_{i,j}$  for all  $i, j$  then we are done because  $c_{m,n}$  length  $c_{m,n}$  is basically length of the longest common subsequence of  $x$  and  $y$ . So now, in the last lecture we have seen a recursive formulation for this  $c_{i,j}$ .

(Refer Slide Time: 01:36)

So, that is basically  $c_{ij}$  is  $c_{(i-1, j-1)} + 1$  if  $x_i$  equal to  $y_j$  and it is basically maximum of  $c_{(i-1,j)}$ ,  $c_{(i, j-1)}$  if  $x_i$  is not equal to  $y_j$ . And we prove this part and to prove this part we have taken a longest common subsequence  $z_1$  to  $k$  which is a longest common subsequence between  $x_1$  to  $i$ ,  $y_1$  to  $j$ .

So, this is also  $z_1$  to  $k-1$  is a longest common subsequence between  $x_1$  to  $i-1$   $y_1$  to  $j-1$ . So, this is one of the hallmark of dynamic programming problem. So, for dynamic programming problem basically you have 2 hallmarks this is the first hallmark. So, this is telling us a solution of a problem contains the solution of the sub problems and optimal solution to a problem contains the optimal solution of the sub problem. So, this is the longest common subsequence up to  $k$ . So, it is up to  $k-1$  is longest common subsequence of this ok.

So, this is a hallmark of dynamic programming problem.

(Refer Slide Time: 03:30)

So Let us write this dynamic programming. So, this is hallmark number 1. So, it is telling the optimal substructure. So, it is telling that an optimal solution to a problem contains optimal solution to sub problems. So, like if  $z$  up to  $k$  is a LCS then  $z$  up to  $k - 1$  is the LCS of  $x$  up to  $k - i - 1$   $y$  up to  $j - 1$ .

So, this is one of the hallmark for dynamic programming problem. So, if we have such a hallmark then maybe you have to fix our mind that maybe you have to go for dynamic programming approach. So, this is one of the hallmark. So, you have second hallmark also to reach that let us just write a algorithm for this. So, this is the LCS algorithm, recursive algorithm.

(Refer Slide Time: 05:52)

Recursive algorithm for LCS. So, basically we have finding the LCS of  $x$   $y$   $i$   $j$  this is basically length of the longest common subsequence  $c$   $i$   $j$ .

So now if  $x$   $i$  equal to  $y$   $j$  then we know it is basically then it is  $c$   $i$   $j$  is basically LCS of I mean length of LCS of  $x$   $y$   $i - 1$   $j - 1 + 1$  this is the this case else. So, else 2 3 else we have  $c$   $i$   $j$  is max of the length of this 2, LCS of  $x$   $y$  1 index less and other way  $i$   $j - 1$ . So, this is the algorithm for finding the  $c$   $i$   $j$ . I mean now what is the worst case for this algorithm? You want to analyse the worst case of this algorithm.

So, worst case will happen if  $x$   $i$  is not equal to  $y$   $j$ . Because in that case we have 2 call then you have to take the maximum. And also if  $x$   $i$  equal to  $y$   $j$  then we have only one call and size is also for both the index reduced by 1, but if we have if  $x_i$  is not equal to  $x_j$  then you have to take we have to get we have to have 2 call recursive call, and then we need to get the maximum of that. And even the both the call we have only one index less. So, this is the worst case. So now, draw the worst case recursive tree for some example values of  $m$  and  $n$ .

So, let us draw the worst case recursive tree.

(Refer Slide Time: 08:50)

So suppose  $m$  is 3  $n$  is 4. And we are in worst case. So, worst case means we have to go for  $x_i$  is not equal to  $y_j$ . So, give a call with  $m \ 3 \ m \ 4$ . So, since it is worst case we have to we have 2 calls which one index less. So, 2 4 then 3 and then after calculating this  $c_2$  then we take the maximum. Again we are in worst case. So, this is index is 1 1 4 and this is 2 3 again here 2 3 3 2. So, like this, again here 1 3 2 2 again here 1 3 2 2 like this ok.

[Please follow the video lecture for better understanding the example.]

So now this tree is common. So that means, we have same sub problems over here. Because they are common I mean. So, unnecessary we are doing the work double basically. So, too said that we need to have to memorize that I mean whether we have calculated, that if you have calculated corresponding  $c_{1 \ 3}$  then we will not calculate again for this 3. So, there are many repetition over here. So, this is the what is the length of this height of this trees  $m + n$ . This is the height of this tree. So, this is another hallmark of the dynamic programming problem. This is the many repetition of the overlapping subproblems. So, this is the repetition. So, let us write that second hallmark of dynamic programming problem.

So, this is basically telling us overlapping subproblems.

(Refer Slide Time: 11:24)

So dynamic programming this is hallmark number 2. And this is telling us overlapping sub problem. So, what it is telling us? It is telling that a recursive solution contains a small number of a distinct sub problems repeated many times. So, this is one of the hallmarks for dynamic programming problem. So, we have many overlapping subproblems.

So, that is one indication that now we must go for the dynamic programming technique. So, this is the second hallmark. So, if we see that we have some recursive formula and there we have this type of overlapping subproblems. So, many repetitions are happening, then we must be ready for now it is time for go for the dynamic programming technique. So now, let us modify that algorithm of recursive algorithm to avoid this repetition. So, that is the memoization. So, we will memorize the value which we have calculated. So, if you have calculated some value we will memorize that we will not recalculate again.

So, that is the memoization algorithm of that finding LCS.

(Refer Slide Time: 14:11)

So, let us just Memoization not memorization, memoization algorithm for finding the LCS. So, the remaining part is same all the thing if we have already calculated some value will not recalculate again so; that means, if  $c_{i,j}$  is nil; that means, if it is not calculated then only we go for calculating it.

So, this part is same as earlier only thing we have a checkpoint over here to avoid the recalculation. If we are calculated the value will not recalculate again, that is why it is memoization. So, that you will give us the dynamic programming technique. So, it is basically the bottom up technique. So, it is a tabular technique. So, we will just try to find out this  $c_{i,j}$ 's.

(Refer Slide Time: 16:31)

So, it is basically dynamic programming approach. So, idea is basically computing that table bottom up.

So, let us just take that example. So, you have the x value and y value. So, A B C let us take a x sequence and y sequence D A B. So, this is our x sequence and we have a y sequence B D C A B A B D C A B A. So, we just take this table like this. So, this is a tabular method. So, this we put in a table. So, this is our x sequence starting from here, and this is our y sequence. And we want to calculate this  $c_{i,j}$ 's  $c_{i,j}$  is basically the bottom up way we will calculate this.

So, these are all 0 because this y is not started yet, and there is no common thing. So, these are all 0. Now we will calculate say this field. So, for this field we need to take these and these value. So, if you take these and these value. So, this is basically  $x \ c \ 1 \ 1$ . This is basically  $c \ 1 \ 1$ . So, it depends on  $x \ 1 \ y \ 1$ .  $X \ 1$  is A  $y \ 1$  is B. So, if you use the formula they are not same. So, it is basically maximum of  $c \ 0 \ 1$ ,  $c \ 1 \ 0$ . So, these two are both 0. So, it will be 0. So let us fill up the table in a bottom up way.

[For detailed explanation on how to fill up the table , please follow the video lecture.]

Now, the question is from here how we can find the longest common subsequence because we have simplified that problem. We need to find the longest common

subsequence. So, we reduce the problem into finding the length of the longest common subsequence. Now after getting the length you are interested to find a longest common subsequence. So, for that what we need to do? We need to follow this path.

So, if I follow this path we can get the longest common subsequence. So, we have to find a longest common subsequence we simplify this problem to find the length of the longest common subsequence. So, by this dynamic programming method we got the length and after getting the length we just follow this path to get the longest common subsequence. So, what is the time complexity for this algorithm for this recursive algorithm? So, basically we have a if the length of this is m.

So, the time complexities. So, if x is of size 1 to m, and y is of size 1 to n. So, basically to compute this the time is basically order of m into n. So, this is the time complexity for p p at the table. So now, also p p at the table we can get the length of the longest common subsequence. So, this is the dynamic programming technique and we illustrate this technique by an example which is called longest common subsequence problem. So, for any dynamic programming technique we should have such kind of recursive formula, and we should have 2 hallmarks.

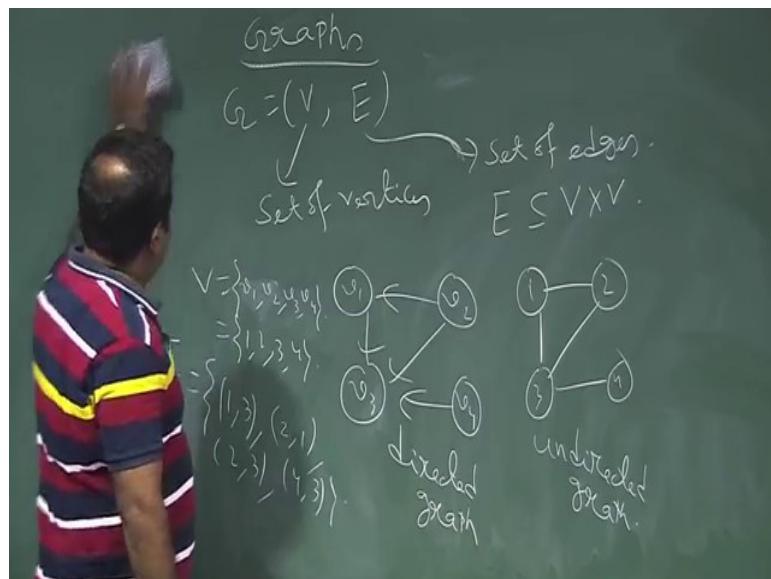
So, in the recursive formula we should have optimal substructure hallmark. So, that is telling us if we have a solution of a problem, that should contain solution of the sub problems. That is the first hallmark of the dynamic programming technique. And the second hallmark is overlapping subproblems. So, if we can see that many subproblems are repeating basically, then we must think that now it is time to go for dynamic programming technique.

Thank you.

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

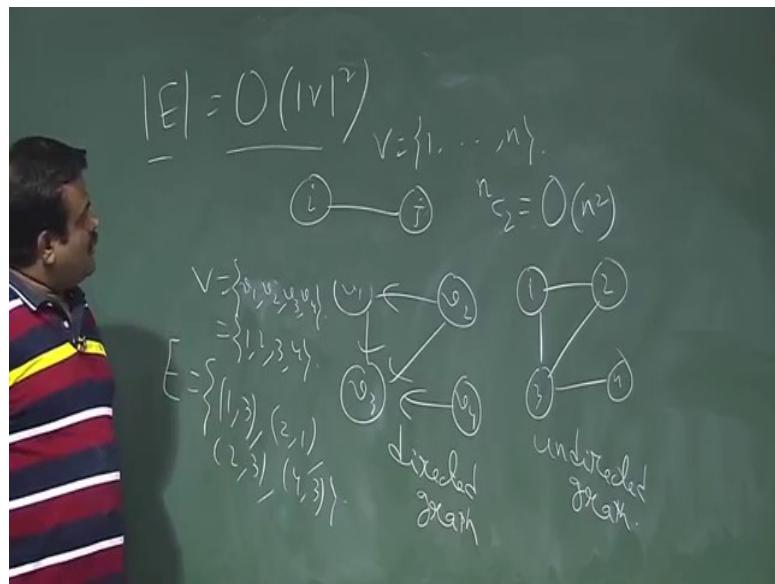
**Lecture – 38**  
**Graphs**

(Refer Slide Time: 00:28)



So, we'll talk about graphs. So, basically for graph we have  $V$  and  $E$ . So,  $V$  is basically the set of vertices or it is called nodes also, and  $E$  is the set of edges. So, basically  $E$  is a subset of  $V \times V$  and there are 2 types of graph directed graph and undirected graph if there is no ordering of the edge then it is called undirected graph and if there is a ordering in the edge then this called directed graph or digraph. Now for undirected graph we are not having the direction.

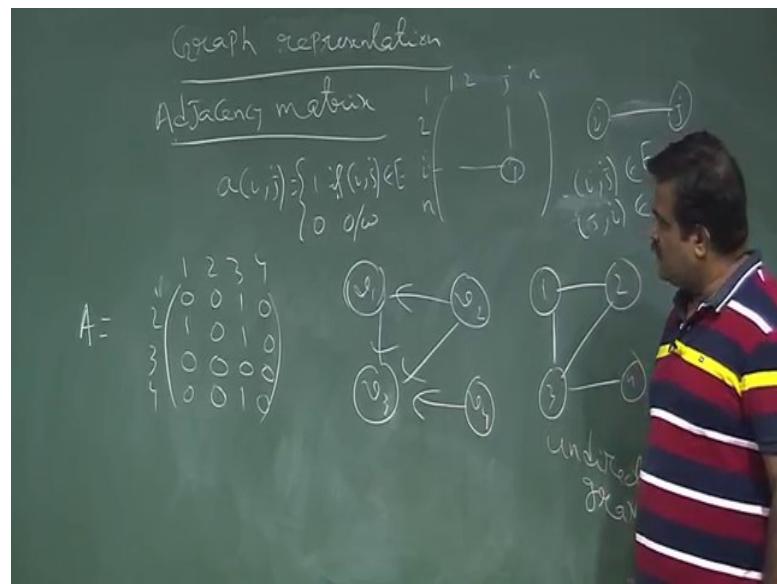
(Refer Slide Time: 03:08)



So, now in a graph, what is the cardinality of  $E$ , so  $E$  is basically the order of  $V$  square big  $O$  of  $V^2$  because it can be at most connected from every vertex to every vertex. So, any vertex to any vertex there is an edge. So, if we have  $V$  say  $v_1 v_2 \dots v_n$ . So, if it take any 2 vertex  $i$  and  $j$  if there is a edge. So, then there will be  $nC_2$  possible edges. So, this is basically order of  $n*n$ , so that is why order of  $V$  is upper bound by order of  $n*n$ . So, this is for a complete graph. So, this is the upper bound. So, order of  $V$  is bounded above by order of  $n*n$ .

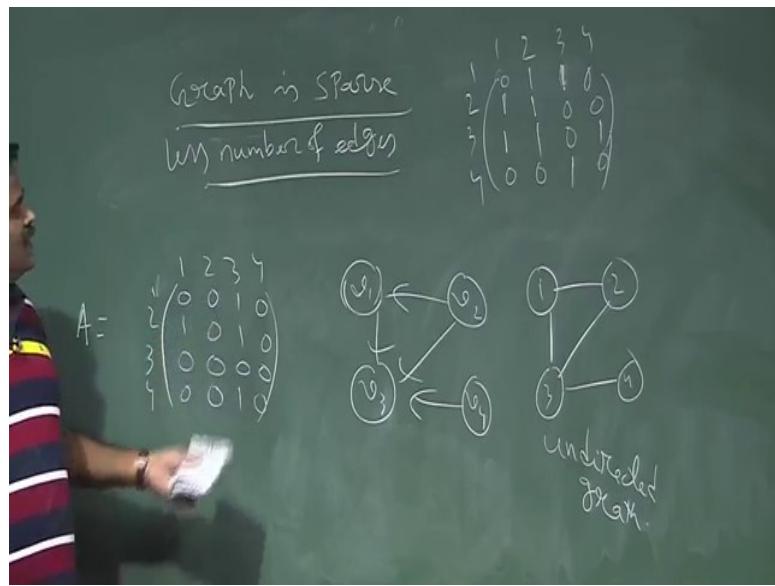
So, now the question is how we can represent the graph. Suppose, we want to give the input of this graph to a computer and we want to have a algorithm where our computer need to take a graph as a input. So, for that there are two methods basically adjacency matrix and adjacency least. So, what is the adjacency matrix representation.

(Refer Slide Time: 04:43)



So, let us talk about graph representation. So, adjacency matrix is basically suppose you have m vertices. So, this is a matrix of size m by n. So, this at the vertices so up to n, v 1, v 2 up to n. So, matrix will be 0/1 matrix. So, if this is an i th vertex, this is a jth vertex. So, i j element will be 1, if i j is an edge and 0 otherwise. So, if you have an edge from i to j, this is the i th vertex this is the jth vertex. If there is a edge from i to j then that corresponding field will be 1 in the adjacency matrix; otherwise it will be 0, that means there is no edge between i to j. So, if a graph is undirected graph then this matrix will be the symmetric matrix because if undirected graph then there is no direction. So, this matrix will be symmetric matrix for an undirected graph.

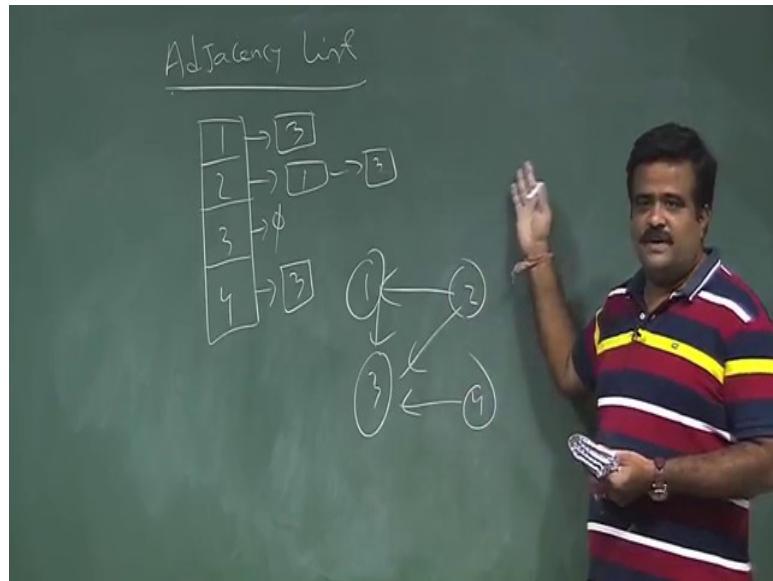
(Refer Slide Time: 07:59)



So, now, the question is whether this is a good way of representing a graph. So, what is the drawback of this. Now, suppose the our graph is sparse graph that means, suppose there is less number of edges suppose the graph is sparse graph so that means, less number of edges. Then this matrix will be a sparse matrix that means, no. of vertex is more. So, this matrix size will be 100 by 100 and suppose we have only 10 edges. So, in this matrix 100 by 100 matrix we have only 10 fields where we have 1, remaining has 0.

So, this will be sparse matrix so that means, we are wasting the memory to have represent base we need to have a array two-dimensional array if you are implementing this in c, we have two different two-dimensional array to have this matrix. So, basically for sparse graph this presentation is not good. So, for that we just have what is called adjacency list that is also one way of representing a graph.

(Refer Slide Time: 10:41)

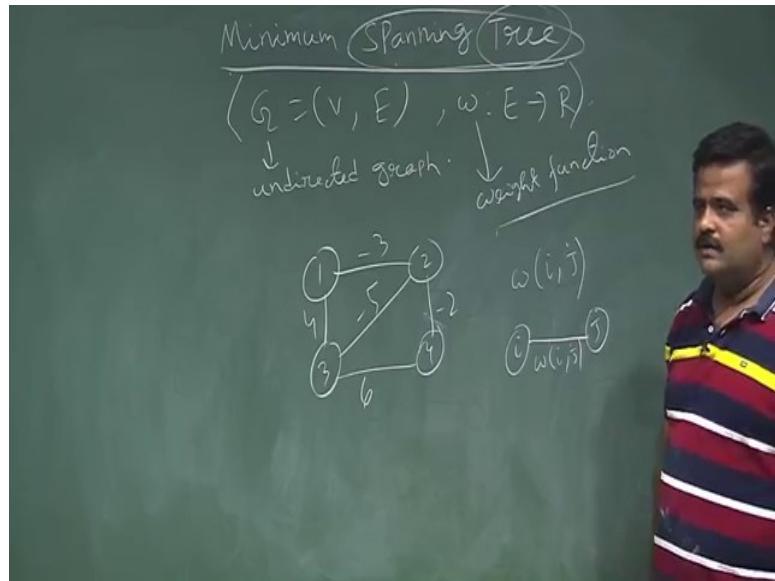


Adjacency list, so adjacency is list is basically so we take a vertex and we have a link list corresponding to each of the vertices. So, this list contains the vertices which are connected to that vertex. So, for example, if we had that the graph 1, 2, 3, 4, so we have this v 1, v 2, v 3, v 4. So, adjacency list means 1 is connected with 3. So, 2 is connected with 1 and 3. So, for 3 there is no connection, this is empty; and from 4 we have a 3. So, this is the adjacency list. So, from each vertex we have a link list kind of thing where that contains the vertices which are connected with this. So, this is basically the adjacency list representation.

So, this is good if our graph is sparse graph because in that case we have only few vertices so that means, we have the list containing few number of edges, a few number of nodes so that way it is good. But if the graph is dense graph if there are more edges than this will be a huge list, then it is good to go for a adjacency matrix, because that is a 0, 1 representation. just the bit vector 0 or 1, so that is very nice data structure just a 0, 1 representation.

So, if it is dense graph that means, if the number of edges are more then we will go for the adjacency matrix representation; otherwise if it is a sparse graph if the number of edges are less one can go for the adjacency list. Because in adjacency list you have to go for the linked list linked implementation, but matrix is very easy to handle. So, this is basically 2 OA we can represent graph by the adjacency matrix or by the adjacency list. Now, we will talk about some graph problems. So, first graph problem, we talk about minimum spanning tree problem.

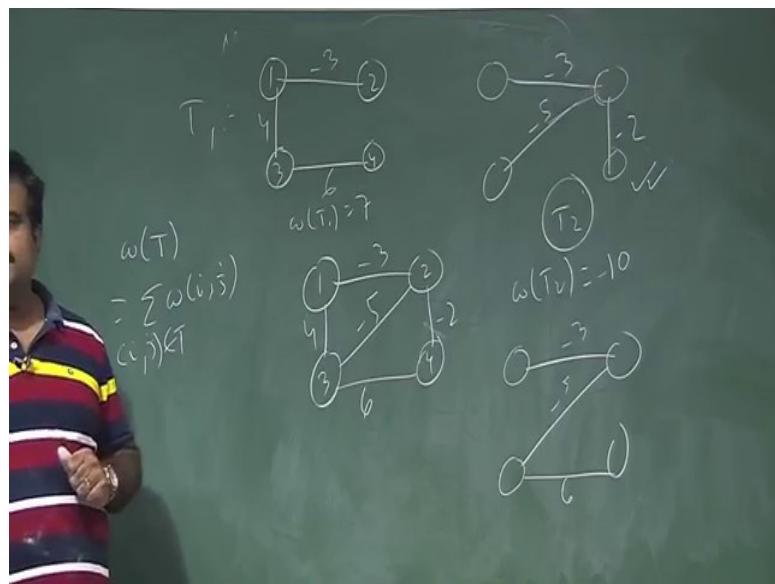
(Refer Slide Time: 13:38)



So, let us talk about minimum spanning tree problem, minimum spanning tree problem. So, for this what are the input, input is basically we have an undirected graph. We have a graph  $G$ , which is basically an undirected graph. And we have an edge  $OA$ . So, we have a weight function  $w$  which is coming  $E$  to  $R$ . So,  $w$  is a weight function. So, each you have an edge weight. So, this is the weight function. So, if  $i$  and  $j$  are two vertices and if there is an edge, so  $w$  of  $i j$  is the weight on that edge. So, weight function is also another input.

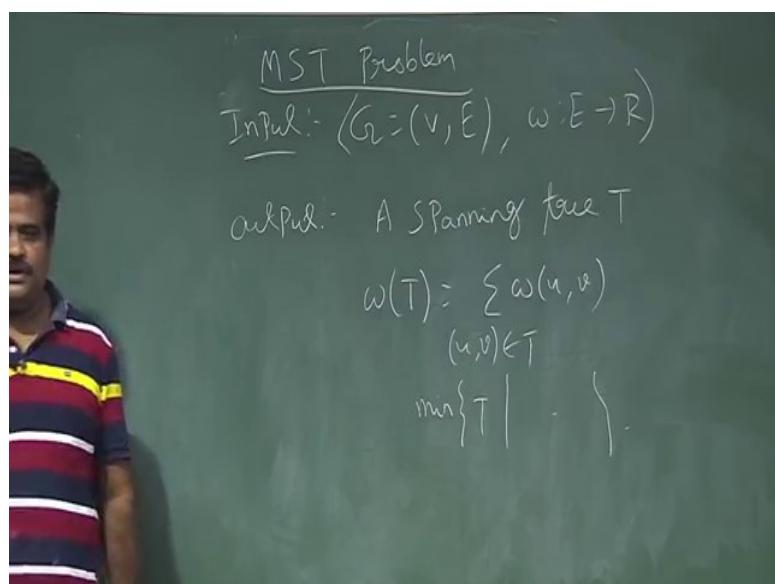
So, this is the input of a minimum spanning tree problem. And we have to find out a minimum spanning tree. So, basically you need to find out a spanning tree which of minimum weight. So, what do we mean by spanning tree? So, it is basically a subgraph it is a tree. So, tree means it should not contain any cycle. So, it is a subgraph it should not contain any cycle and spanning meaning it should cover all the vertices. So, you should get a subgraph which first covers all the vertices. It should cover all the vertices that way it is called spanning tree, and the weight should be minimum among all other. So, what are the spanning tree we can have from this graph. So, if this is the input what are the spanning tree we can have. So, tree should not contain any cycle.

(Refer Slide Time: 16:24).



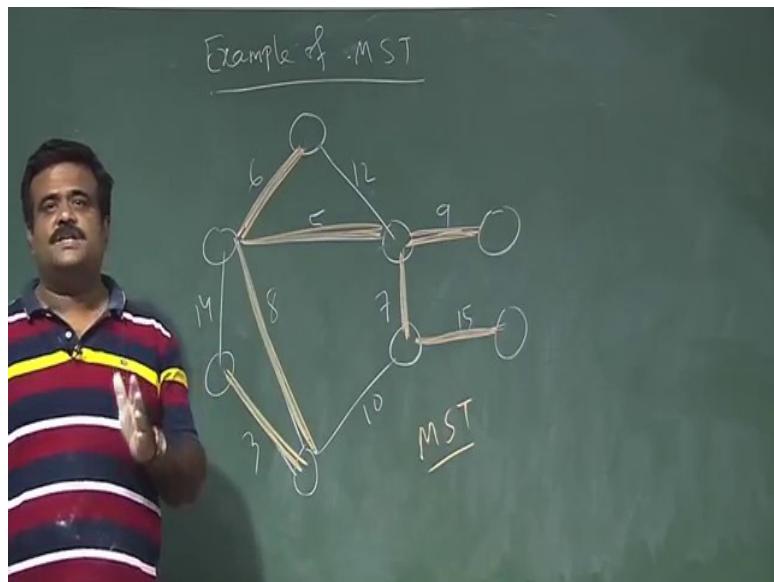
So, we can have a lot of spanning trees; among these I think this one is the minimum one. So, this should be our output of the algorithm, the minimum spanning tree algorithm. So, this is the problem. So, basically we need to find the sub graph which is basically tree, not only tree it should be the spanning tree so; that means, it should cover all the vertices and it should give us the minimum edge weight, so that is the problem. So, given a graph given a directed graph we should find out this.

(Refer Slide Time: 18:59)



So, this is the MST problem - minimum spanning tree problem. So, input is a graph, G directed graph and we have edge weight. So, this is the input. And the output will be a spanning tree which is a sub graph which is a tree no cycle tree means no cycle and that contains all the vertices such that the weight is minimum w t is the summation of w and this is the minimum amount all such t. So, we consider all such spanning tree among these whichever gives us the minimum weight that is the minimum spanning tree.

(Refer Slide Time: 20:25)



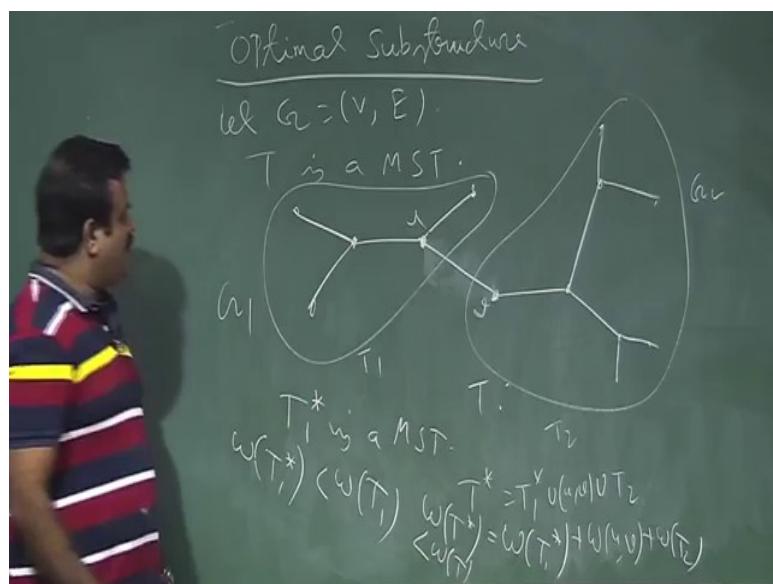
So, let us take an example. So, we have taken a small example let us take a bigger example how we can have a spanning tree of a graph. So, example of a, let us take a bigger graph. So, let us take a little, suppose this is the graph. Suppose, this is the graph with these are the vertices and let us say edge weight 5, 14, 8, 3, 10, 7, 9, 15. Suppose, this is the input and this is the graph with this many vertices and these are the edges and this is the w the weight function on the edges. Now, we want to find out the minimum spanning tree.

Now, can you tell me which edge should be there in the minimum spanning tree? These two edges must be there in the minimum spanning tree because this is the only edge which is connecting these nodes with the remaining nodes and the minimum is the spanning tree. So, if we have to cover all the vertices. So, to cover these two vertices there is no other way which could result in a lower weight.

So, after taking this, we try to cover all the vertices. So iterating this process will result in the MST, the minimum spanning tree. So, we just use the intuition to have this. So, this is the MST.

Now, we want to know the algorithm OA, how we can construction such MST. So, for that we want to see whether we can use the dynamic programming technique which we know for this. So, for that we need to check the hallmark for this. So, there are two hallmark for dynamic programming problem.

(Refer Slide Time: 25:31)



So, optimal substructure, we need to check whether we have this hallmark structure. So, here optimal sub-structure is suppose we have a minimum spanning tree let  $G$  be a graph and suppose we have a min and  $T$  is a MST. So, this is  $u v$  like this, suppose this is a minimum spanning tree which is covering all the vertices. So, we have a graph which has same number of vertices, but in the graph this is still minimum spanning tree and in the graph we have some more edges but number of vertices are same. Now, for optimal sub-structure we need to say that if you have a solution of the whole problem then it contains the solution of the sub problem. So, we remove this edge.

So, you remove this edge, and then we consider this tree as  $T_1$  and this tree as  $T_2$ . Now, we claim the  $T_1$  is the minimum spanning tree of the  $G_1$ , that means,  $G_1$  contain all the edges which are basically connecting with the vertices from  $T_1$ . And  $G_2$  is the all the edges which is basically connecting the vertices of  $T_2$ . So, basically  $G$  is  $G_1$  union  $G_2$ . So, we have to

prove that  $T_1$  is the minimum spanning tree of  $G_1$  and  $T_2$  is the minimum spanning tree of  $G_2$  then that will be sufficient to show that optimal substructure is there.

So, to prove that let us suppose  $T_1$  is not a minimal spanning tree of  $G_1$  so that means, there is another tree which has been minimal spanning tree of  $G_1$ , so that is the  $T_1$  prime is a MST. So that means, weight of  $T_1$  prime must be less than weight of  $T_1$ . So, what we do we take that  $T_1$  prime and we take this edge and we take  $T_2$ . So, we just take this new  $T$  star which is basically  $T_1$  prime union  $T_2$  and this  $T$  star is a spanning tree of the whole graph  $G$ .

And the weight of  $T$  star is less than  $T$  because weight of  $T$  star is basically weight of  $T_1$  star + weight of  $uv$  + weight of  $T_2$  and this is less than  $T_1$ . So, basically weight of  $T$  star is less than weight of  $T$  which contradicts the fact the  $T$  is a minimum spanning tree of  $G$  so, that means, this  $T_1$  has to be the minimum spanning tree of the induced graph  $G_1$ . And similarly we can say  $T_2$  is the minimum spanning tree of  $G_2$ . So, this is satisfying the first hallmark of the dynamic programming technique. And the second hallmark is optimal substructure. With that also we can verify that there are optimal repetition of this. So, that means, one can thing for dynamic programming technique for finding the minimum spanning tree. But in the next class, we will talk about another powerful technique which is called greedy technique or greedy approach to the find the minimum spanning tree.

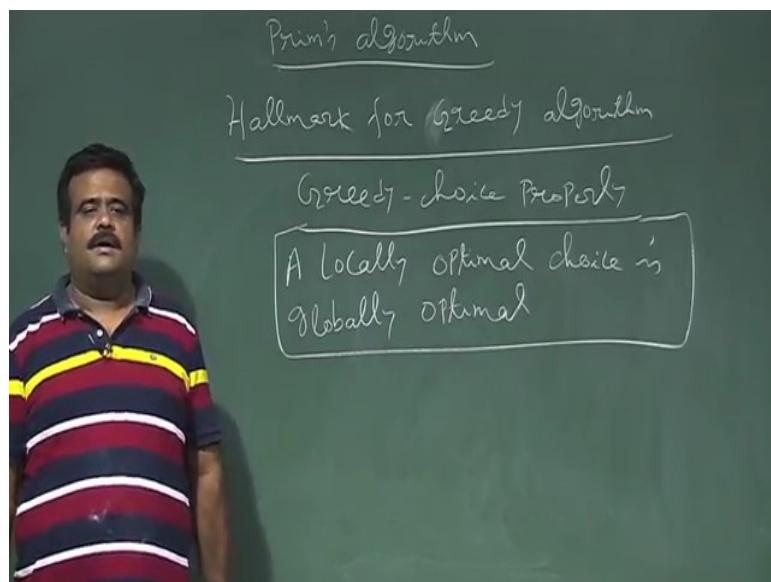
Thank you.

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

**Lecture – 39**  
**Prim's Algorithms**

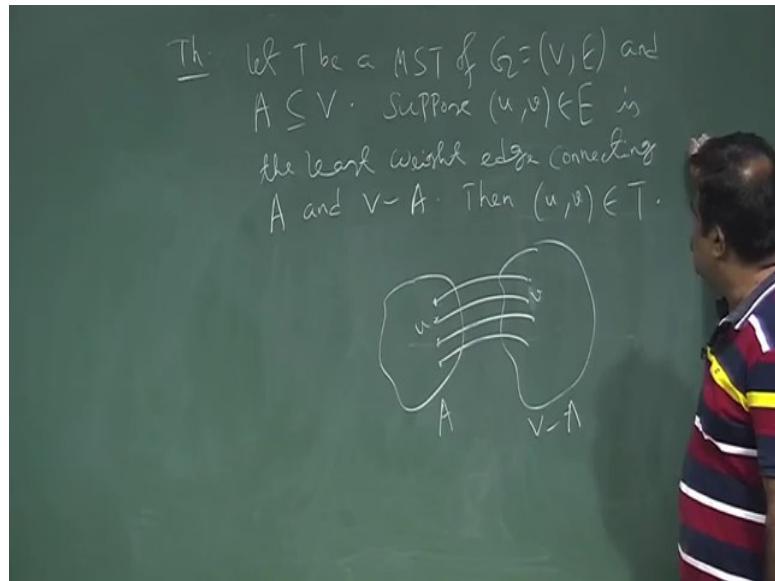
We talk about Prim's algorithm to find the minimum spanning tree.

(Refer Slide Time: 00:29)



So, this is a greedy approach, so let us talk about Hallmark for Greedy Algorithm. So, a locally optimal choice is globally optimal; this is the hallmark for greedy. So, it is telling us a locally optimal choice greedy choice basically is globally optimal. So, Prim's algorithm is a greedy algorithm; so Prim's algorithm is based on this theorem which is basically greedy choice or which is basically coming from this hallmark.

(Refer Slide Time: 02:03)

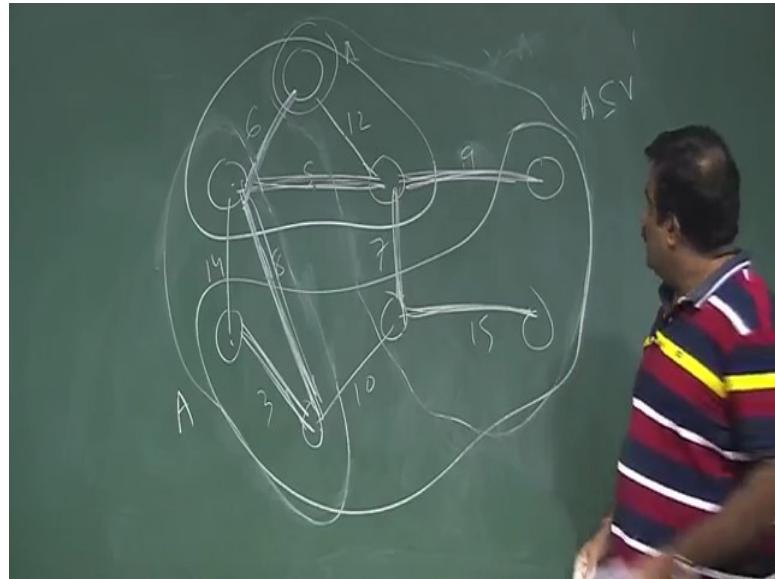


So, this theorem is telling: let  $T$  be a MST of a graph  $G$  which is basically  $(V, E)$  and  $A$  is a subset of  $V$ , any subset of  $V$ . And suppose  $u, v$  belongs to  $E$  is the least weight edge connecting  $A$  and  $V-A$ . Then this theorem is telling this  $u, v$  must be in this minimum spanning tree. This theorem is telling this  $u, v$  must be in this minimum spanning tree. So, what is the meaning of this? Suppose do we have  $A$  S edge, we have some vertices and  $V-A$ . So, there are some bridge edges; bridge edge means the edge which is having one vertex in  $A$  and another vertex in  $A$  compliment. So, they are called as bridge edge.

So, among this bridge edge suppose this is the edge  $u, v$  which is minimum among all this bridge edge, then this theorem is telling this  $u, v$  must be in the minimum spanning tree. And we have to prove this theorem, but before that let us understand this theorem. So, this is telling us this  $u, v$  the bridge edge which is the minimum among all the bride edges, must be in the minimum spanning tree.

So, let us take the earlier example which we have and we will prove this theorem.

(Refer Slide Time: 04:31)



Prim's algorithm is based on this theorem. So, let us just draw the graph we had in the last class. So, suppose this is our graph and these are the weights. So, we have the edge weights as 5, 12, 14, 8, 3, 10, 7, 9, 15.

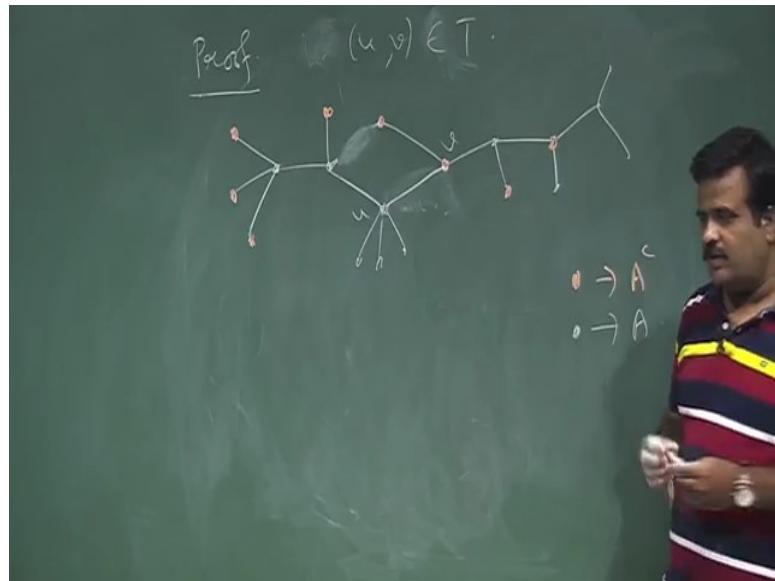
Now we take some vertices A, and the remaining is V-A or A-compliment. Now we consider all the bridge edges. So, bridge edge means, one part is in A another part is in A compliment. So, we can find bridge edges in this manner. So, among this the theorem is telling to choose the minimum so minimum is 5, so 5 is in minimum spanning tree.

Similarly, if we proceed with the remaining bridge edges, we find the minimum among this which is minimum 7. So we put this corresponding edge into the spanning tree as well.

So, this theorem is telling if we take a any set A; subset of the vertices; so if we take these as A and the remaining are in A-compliment then we choose the minimum bridge edge. So, this has to be in the minimum spanning tree and that is the greedy choice. And that is the locally optimal. So, that is the greedy hallmark. Locally optimal: a locally optimal solution is giving us the globally optimal. So, if we start with this vertex we just take this minimum edge. So, this is the greedy choice. So, that is the locally optimal choice and this happens to be globally optimal as well. So, that is the hallmark for greedy.

Let us try to put this theorem. Then we will use this theorem to have an algorithm which is called Prim's algorithm.

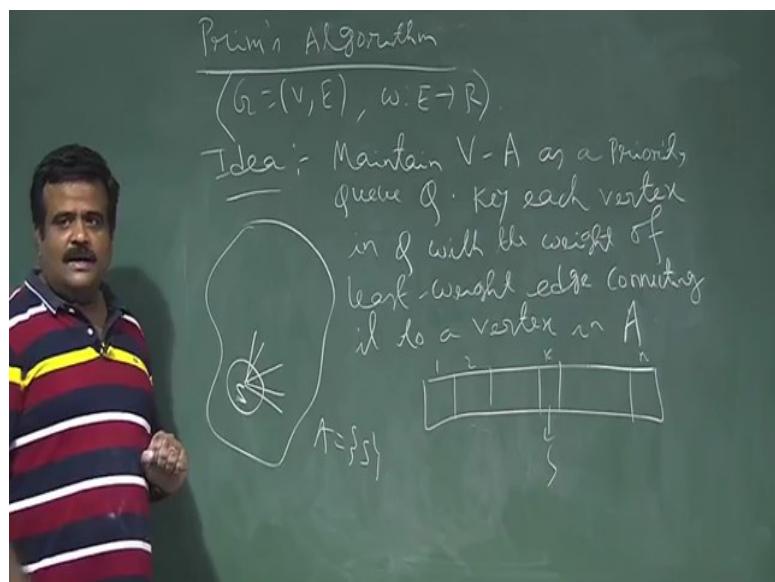
(Refer Slide Time: 08:03)



Now using contradiction approach, we can prove that the minimum bridge edge(locally optimal) will result in the minimum spanning tree(globally optimal).  
[For detailed proof please refer to the corresponding section in the video.]

Now, based on this theorem we have an algorithm which is called Prim's algorithm which is basically greedy algorithm to find the minimum spanning tree.

(Refer Slide Time: 11:26)



So, let us write the Prim's algorithm. So, what are the inputs? Input is a graph, it is an undirected graph and we have weights on the edges and the output will be a minimum

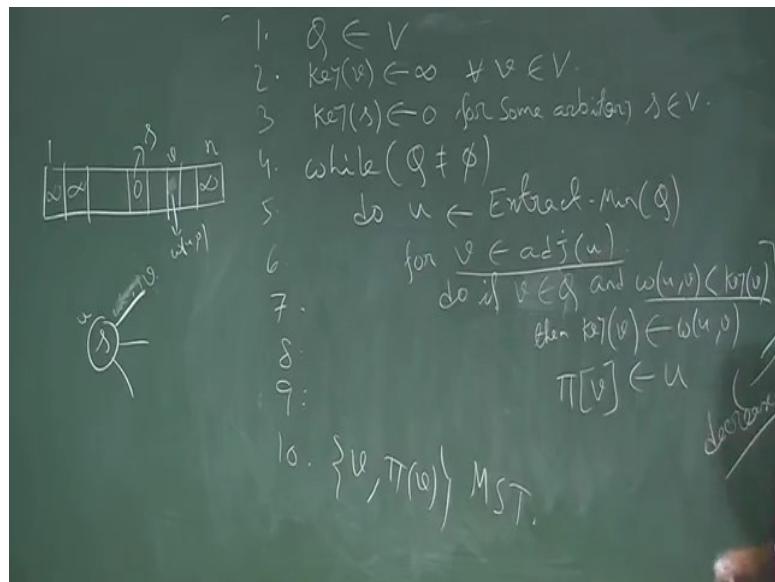
spanning tree. So, that is the output. So, what is the idea? So, idea is to maintain a priority queue. So, we maintain V-A as a priority queue Q and we key each vertex in the queue with the weight of the edge connecting it to a vertex in A.

So, connecting it to a vertex in A. So, basically what we do? So, we basically start with the vertex S. So A is constant initially as S. So, that is the starting vertex it could be any vertex. Now we consider all the bridge edge which is connecting from S to A-compliment basically. And that is the key value of the vertices. And we put everything into the Q and the Q is basically maintaining this as the priority queue.

Now we keep the weight of the each vertex as the minimum of the weight which is connecting from that vertex to A. And now we choose the minimum weight edge and that must be in those minimum spanning tree and that is the greedy choice. So, we capture that vertex in A and slowly we grow the tree. So, we start with vertex S and slowly we grow the tree. So, that is the idea, ok.

Let us write the code then it will be more clear. So, let us write the code for Prim's algorithm; pseudo code.

(Refer Slide Time: 15:09)



So, initially everything is a queue. So, this Q is a priority queue, we can just have a array or the heap implementation for that. And the key value of V is infinity for all v in V and the exception vertex which is the starting vertex which we put the key value as 0 for some

arbitrary  $S$  from  $V$ . So, we choose  $A$  vertex to start, that is the starting vertex. Then, while  $Q$  is not empty; obviously,  $Q$  is not empty initially. So, we just extract the minimum from this  $Q$ ; extract min from this  $Q$ . And then after extracting min, so then we just update this  $Q$  value of the vertex which are adjacent to  $u$ .

For each  $v$  in the adjacency list of this  $u$ ; so what we do? Do: if  $v$  is a  $Q$  and the  $u v$  is less than  $Q$  of  $V$  then we change then the key of  $V$  is basically  $w u v$ . And you make it a responsible vector. Now  $u$  is responsible for the degree of  $V$  change. So, this is sort of responsibility vector. So, this is 6, 7, 8, 9, and finally in 10 at the end this  $V$  comma  $\pi$   $V$  will give us the MST.

So this is the code and this operation is basically this operation is called decrease key. So, if this key value is less we are going to that particular position and we are making the value changing the value. So, this is the decrease key operation. So, this is the pseudo code for prince algorithm. So, basically what we are doing we are starting with the vertex and we are initially we have putting vertex to the key value is infinity, because initially nothing has explore; except a vertex  $S$ .

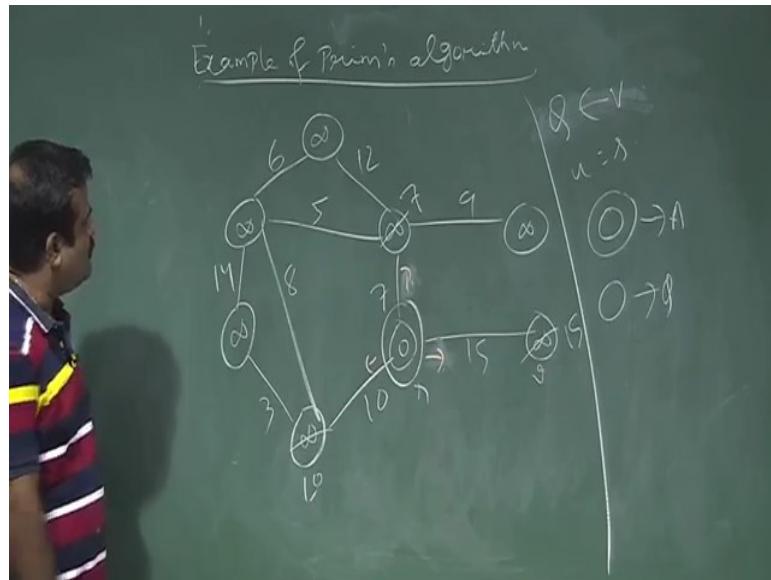
So, if we just implement this using an array. So, we have  $A$  vertex this is the array implementation suppose we have  $n$  vertices and we choose a vertex  $S$  any of this vertex will be  $S$ .

Now, once you extract the minimum since this is 0 so this will be extracted and this we are going to put in basically extract min means- it is deleted from the  $Q$  and it is added in  $A$ . So, it is deleted from the  $Q$  and it is added in  $A$ . So, everything is infinity and 0 is the minimum, so  $S$  will be the first  $u$ . Now  $S$  is any arbitrary vertex and now we consider all the adjacency vertex of  $S$ . So, this is our  $u$  is the first vertex.

So, now it was having that is key value is infinity initially. Now this has a weight, now if the weight is less than the key value(initially it is infinity), if that weight is not infinity, so we have to change this key value. So, if this vertex is  $V$  then if this is infinity, now if this has some value, so this value we are going to put over here. So, this way we update all the adjutancy vertex of  $u$ . And we make a responsibility vector like for this change, for this degree change this vertex is the responsible. And then we will repeat this until the  $Q$  is empty, then we will choose the next vertex. So, basically we start with a vertex and slowly we grow the tree; that is the idea.

Let us take an example how we execute this. So, this can be implemented using the heap also in that case we have to decrease. So, we will come to the time complexity of this when we analyze the Prim's algorithm. So, let us take a quick example of the Prim's algorithm.

(Refer Slide Time: 20:50)



How is Prim's Algorithm working? So, suppose we have the same graph as earlier. And these are the weights we have 5, 14, 8, 3, 10, 7, 9, 15. So, this is the input, this is the graph we want to execute the Prim's algorithm on. So, we put everything into the priority queue it could be array- simple array. Now, we put every key value key of V is to be infinity except some starting vertex; suppose this is the starting vertex. So, this is the S starting vertex. And we put everything to be infinity the key value, because nothing has been explored this is the initialization.

Now, everything is in Q with the key value infinity except this vertex. Now we extract the minimum, so extracting minimum means. So now, Q is basically all the vertices; now extracting minimum means it is getting the minimum it is deleting from the Q and it is adding in A. So, this vertex is added in A. So now, u is basically s, ok.

So, now we consider all the vertices adjacent to this. So, these are the vertices. So, this is V for this V key value was infinity, now this is 15 so 15 is better than infinity. So, we have to decrease this key. So, this will now be 15. And we have to put a mark that this vertex is responsible for this change. So, similarly this will be 7 and this vertex is responsible for this

change. So, this mark is needed. So, let us use another color for this mark. So, this is now 10 and these vertices responsible for this change, ok.

So, now this is in A and this is in P-A. Now so Q is not empty, now again we will do the extract min if you do the extract min who is the minimum everything is infinity except this is 15, this is 15, this is 7. So, this is our next u. So, we choose this u and this will be extracted from the Q. So, this is our next u, now if this is u now we consider all the vertices adjacent to you. So, now, this is infinity now this will be 9 and we put an arrow there because for this change this vertex is responsible. Now this is another vertex, this is infinity, now this is 12 and we have to put a arrow for this, and also this is infinity, now this is 5 and we have to put an arrow for this. And this is already in Q so we do not need to do anything, ok.

So, we started with this we captured this vertex. Now which is the minimum? Now 5 is the minimum, so you extract 5 form the Q and put it in A. So, once we extract 5, so we check all the vertices which are adjacent to 5. So, this is vertex, this was infinity, now it is 14. So, 14 is better than infinity so you need to perform the decrease key operation. So, this will be 14 now and this is the responsible vector.

And now this one, this one is 12. Now, we have a better one which is 6. And now this was responsible for this change. Now who is responsible? Now this is the responsible for this. That is why this sign is important, so this responsibility vector. Who is the responsible for? The final change and that will give us the minimum spanning tree.

So, basically the idea is we start with a vertex S and slowly we will grow the tree. So, we start with a vertex S and we slowly capture all the vertices in a greedy way. So, that is the locally optimal choice and it will become a globally optimal; that is a greedy hallmark. So, we start with the vertex S then we capture this vertex in a greedy way. So, slowly we capture all the vertices.

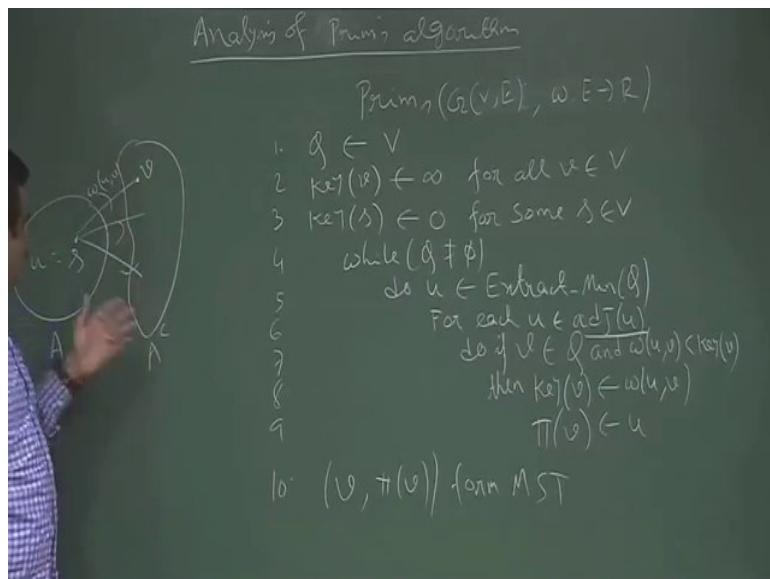
So, this is the example of Prim's algorithm. In the next class we will discuss, we will analyze the Prim's algorithm and the time complexity basically. So, we will discuss the time complexity in the next class.

Thank you.

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

**Lecture – 40**  
**Graph search**

(Refer Slide Time: 00:39)



So, we talk about graph search, how you can explore a graph. So, before that let us just finish the analysis of Prim's algorithm from the last class. So, last class, we have described the Prim's algorithm and we have taken an example, to see how it is working. So, now we will talk about the time complexity of this algorithm so that is called analysis of Prim's algorithm. So, as we know Prim's algorithm helps to find the minimum spanning tree. So, input is a graph, we have a undirected graph  $V, E$  and we have a edge weight which is basically associated with a weight.

So, the algorithm goes like this. So, we have assigned each vertex to a priority queue. So, we put everything into the priority queue. We maintain this queue based on the degree of each vertex, and we assign the degree of each vertex to infinity for initialization because initially nothing has been explored, So that is why you put degree of vertices to infinity. So, nothing is exploring yet, but we have to have a starting vertex.

So, if we recall, the Prim's idea is a greedy choice. We start with a vertex  $S$  and we capture all the vertices which are connected with edge to other vertices whose edge weight is minimum,

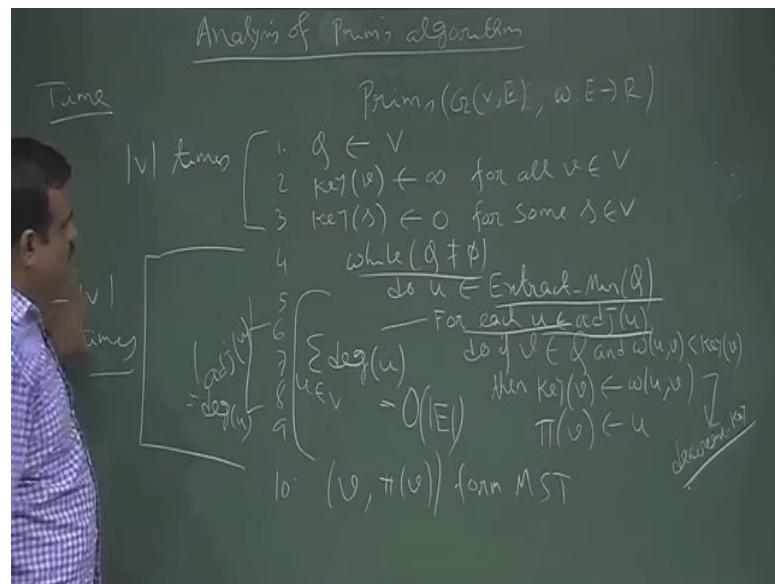
so that is the greedy option. So, which we have to start with a vertex and this vertex is we can chosen arbitrarily. So, for some arbitrary vertex  $S$ , we have to have a starting vertex, so that we can decide. So, any vertex could be the starting vertex in Prim's algorithm, so that we have to choose arbitrarily. And now this loop is for while  $Q$  is not empty. Also  $Q$  is starting with all  $v$ .

Now, we extract the minimum from this  $Q$  and the minimum is based on the key value. So, initially minimum will be the  $S$ . So, this  $S$  will be extracted and this will be  $u$ . And when we extract something from  $Q$ , it will be deleted from  $Q$  and it will be added in a set called  $A$ . So, it will be deleted from  $Q$ .

Now, we check this weight by  $v$  vertex if this is in  $Q$ . So, initially everybody has to be in  $Q$  because initially  $u$  is basically  $S$  other than  $u$  everybody is in  $Q$ . So, their degree was infinity, now we have to change the degree by this weight  $w$  of  $u,v$  because this was infinity and in the later stage this is  $u$ . So, we check the degree of this and we change the degree. So, we check the degree, if the degree is this, this step is called decrease key, decrease key operation.

So, basically we are decreasing the key like this. So, because these are having a degree which is more than the degree what we have right now. So, we have to update that. So, we update that I mean put a. So, who is the responsible for this update this guy. So, we put a link to that to have the minimum spanning tree at the end. And now again we extract the minimum, so that minimum will be the minimum among the bridge edge. So, this is our  $A$  set. A set means everything in  $A$  and  $A$ -compliment is  $Q$ ,  $A$  is basically  $Q$ -compliment. Now, there is a theorem we have prove in the last class that all the bridge edges if we consider  $A$  to  $A$ -compliment whichever is the minimum has to be in minimum spanning tree, and that will reflect in their degree.

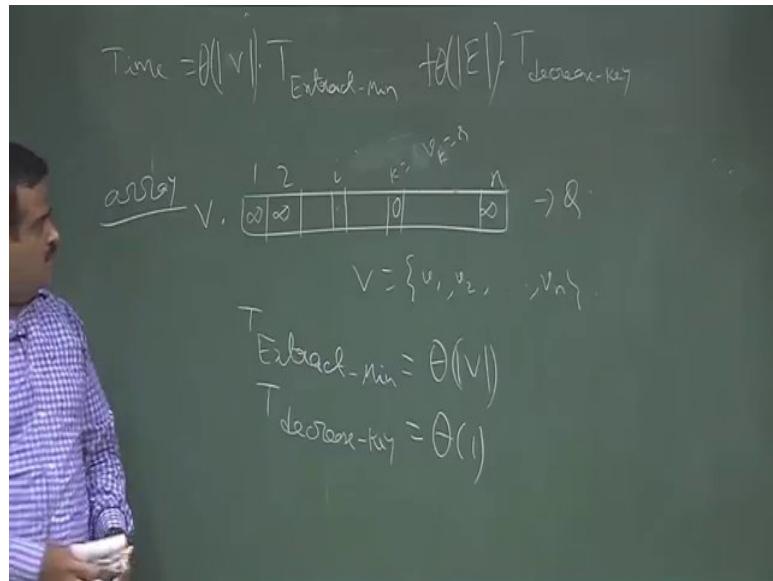
(Refer Slide Time: 05:23)



Now, we talk about time complexity of this algorithm. So, this initialization step is for order of  $V$  times ( $V$  is the degree of the vertices). So, we can just write order of  $V$ , actually this should be order of cardinality of  $V$ , but anyway we are just denoting this  $V$  means anyway to be very specific we have to have this. And order of this inner loop is  $V$  times by this initialize some step and this outer loop while  $Q$  is empty because  $Q$  is initialized by  $V$ . So, this is also  $V$  times. So,  $V$  times we are doing the extract minimum from the  $Q$ . So, it depends how much time our  $Q$  is taking to extract minimum. So, depends which data structure we are using for the  $Q$ , we will come to that analysis.

Now this step, the inner loop, we have to go for the adjacency list of  $u$ . So, this step will take adjacency list of  $u$ . So, this we are doing for all  $V$ . So, this is basically summation of this. So, is called degree of the cardinality of this is basically degree of  $u$ . So, this step will be take summation of the degree of  $u$ ,  $u$  belongs to  $V$ . So, this is basically order of  $E$  by handshaking lemma summation of degrees order of  $v$ . So, this time we are doing the decrease key. Again decrease key will depend on the data structure we are using to have this. So, what is the total time complexity, time complexity is "(order of  $V$ ) + (order of  $V$  times extract min) + (order of  $E$  times decrease key)". So, the time complexity for Prim's algorithm is further explored below.

(Refer Slide Time: 07:50)



So it is basically order of  $v$  times time taken to extract the min. So, it depends how we extract the min. So complexity is basically (order of  $v$  times time taken to extract the min) + ( $E$  times time taken for the decrease key operation) because we are decreasing the key value of a node. So, we have to go to that particular node and we have to decrease this. So, this is the time complexity for Prim's algorithm in general.

Now, suppose depending on which data structure we are using to have this queue, suppose we are using array for our queue. Suppose, we are using array data structure simple array, annotate array so that means, we are putting everything. Suppose, our  $V$  is say vertices  $v_1, v_2, v_n$  suppose there are  $n$  vertices. So, we have an array, so  $V$  array is  $1, 2$  up to  $n$ . So, this is our queue say so we are having an array for our priority queue. So, this is the key value each is insulated by except some value  $s$ ,  $s$  is say  $v_k$ . So,  $v_k$  is basically starting vertex. So, except this everything is infinity. Now, this is changing because every time we are checking. So, how much time, will we take them find the minimum from this array. So, how much time will it take for the extract min? So, we have to scan all the element. So, this is basically order of  $V$  times.

And now decrease key. So, decrease key means suppose we have to decrease these values. So, we have to go to this position and we have to change the value. So, decrease key will take constant time. So, decrease key will take constant time, because we are going to that particular position and then we are changing the value. So, in that case what is the

complexity, this is basically order of V and this is order of 1. So, this time is basically order of  $V^2$ .

(Refer Slide Time: 10:56)

The image shows a man in a blue and white checkered shirt standing next to a chalkboard. He is facing away from the camera, looking at the board. On the chalkboard, there is handwritten mathematical notation related to algorithm analysis:

$$\text{Time} = \Theta(|V|) \cdot T_{\text{Extract-Min}} + \Theta(|E|) \cdot T_{\text{Decrease-Key}}$$

Min-heap

$$\begin{aligned} \text{Time} &= \Theta(|V|^2 + |E|) \\ &= \Theta(|V|^2) \end{aligned}$$

$$\begin{aligned} T_{\text{Extract-Min}} &= \Theta(|V|) \\ T_{\text{Decrease-Key}} &= \Theta(1) \end{aligned}$$

For array implementation time is order of  $V^2 + E$ . I mean this is the cardinality basically. Now we know this order of E is bounded by order of  $V^2$ , so this is basically order of  $V^2$ . So, order of  $n^*n$  if there are n vertices in this array. Now, suppose we are using a min heap for this priority queue. We know the max heap, similarly min heap is basically the heap we know heap is a data structure which is basically an array which is viewed as a binary tree. So, it is a array i-th element node if i th element than the child set 2 i to i + 1, so that way we viewed the heap. So, if we use heap for this priority queue then minimum is in the root. So, for min heap property is every node is less than from his child, so that is the property of min heap. So, we use heap data structure for this priority queue; min heap particularly.

(Refer Slide Time: 12:20)


$$\begin{aligned} \text{Time} &= O(|V|) \cdot T_{\text{Extract-min}} + O(|E|) \cdot T_{\text{Decrease-key}} \\ \text{min-heap} \Rightarrow T_{\text{Extract-min}} &= O(\log V) \\ T_{\text{Decrease-key}} &= O(\log V) \\ \Rightarrow \text{Time} &= O(E \log V) \end{aligned}$$

Then what is the time for extract min, so we know that min is in the root. So, for min heap we know for min heap we know, this is the minimum. Now, we need to extract this. To extract this we can change with the last element here and that will destroy the heapify property, then we have to make it a heapify again, so that will take  $\log n$  time. So, it is basically order of  $\log$  of  $V$ , so that is the extract min.

Now, what about the decrease key? Decrease key means suppose we are going to decrease this key value. Now, suppose it got decreased due to the algorithm the whatever value it was having is more than the direct link from  $u$  to that vertex. So, this has to change. So, if you change that then it may violate many properties. So, we may have to call min heapify here, so that will again take order of  $\log n$  because we do not know the position. So, both the operations decrease key and extract min will take order of  $\log V$ . So, for min heap it will take time basically, order of  $\log V$  and order of  $E * \log V$ . So, basically it is order of  $E * \log V$ . This is if we use the heap implementation of the priority queue.

(Refer Slide Time: 14:27)

$\varnothing$	$T_{\text{Extract-Min}}$	$T_{\text{Decrease-Key}}$	$T_{\text{Total}}$
array	$O(V)$	$O(1)$	$O(V^2)$
binary heap (min)	$O(\log V)$	$O(\log V)$	$O(E \log V)$
fibonacci heap	$O(\log V)$ amortized	$O(1)$ amortized	$O(E + V \log V)$ Worst Case

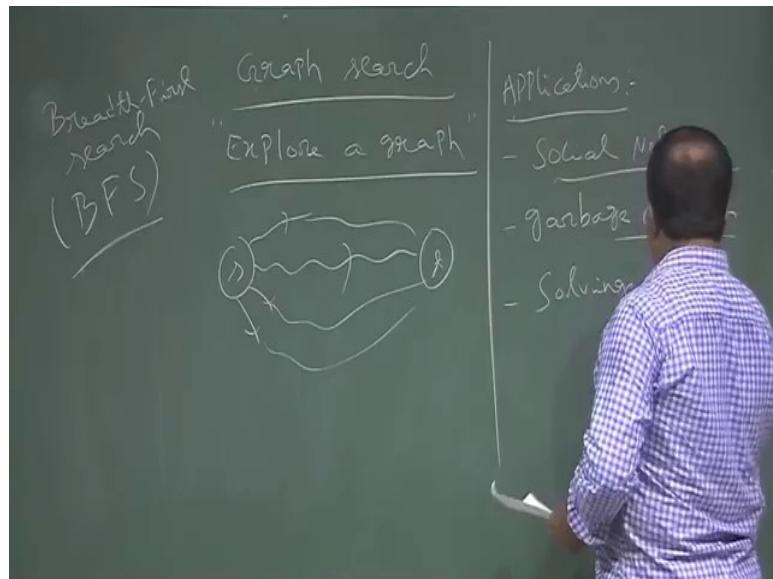
So, now, there is another data structure which is called Fibonacci heap. So, let us write this in terms of priority queue  $T$ . So we will write this in a table extract minimum time and then decrease key and this is the total time. Suppose we are writing this in a table. Now, suppose we are using an array for this priority queue, the data structure we are using here. So, this will take order of  $V$ , this will take order of 1; this will take order of  $V^2$ , so cardinality of  $V^2$ . And if we take the binary heap in particularly min heap then we have seen this will take order of  $\log V$ , this will also take order of  $\log V$  and this will take order of  $E \cdot \log V$ .

Now, if you take another data structure, which is called Fibonacci heap. So, this is there in the textbook Fibonacci heap. So this will take order of  $\log V$  (amortized). So, this analysis is amortized analysis and this will take order of 1, so this is also amortized. So, the total will be order of  $V$ . So, order of  $V \cdot \log V$ , and order of one means decrease key. So, this is  $E$ . So, basically it will take  $E + V \cdot \log V$ . So, total time complexity is  $E + V \cdot \log V$ . So, this is the best among these. So, this is also worst case amortized analysis. So, this is the best data structure for this analysis. So, this is the analysis of Prim's algorithm. So, basically depending on the data structure. So, if we use the Fibonacci heap, the amortized cost in the worst case is order  $E + V \cdot \log V$ .

So, now, we will start the next topic. So, this is the Prim's algorithm time complexity. So, this time complexity will be same when we will talk about Dijkstra's algorithm for finding the

shortest path. So, will not discuss this whole details. So, we have to recap we have to come back here to have the analysis for this table.

(Refer Slide Time: 17:50)



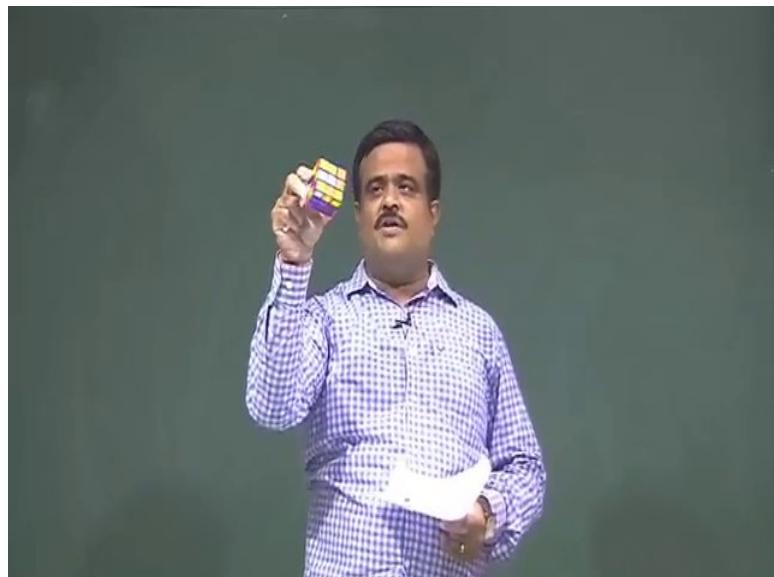
So, let us start with the new topic, which is called graph search. So, how we can explore a graph. So, this is called basically graph search. So, this is to basically explore a graph. So, what do we mean by exploring a graph, so that means, suppose we have a vertex say source vertex and we have another vertex t, we want to find a path from s to t. So, that is one way of exploring a graph. May be we want to find all the path from s to t, so this is one way to explore a graph. Or maybe we want to find the shortest path from s to t, we may or may not have weight there in a graph. So, this is a given graph, we want to explore it. So, this is one way we can find out the path from a source vertex to another vertex, we may need to find out all the paths from source vertex to other vertex.

So, we will talk about breadth first search or BFS, this is one way to exploring a graph, breadth first search this is in short called BFS. So, this is one way to exploring a graph. So, there are many applications of exploring a graph. So, let us just jot down some of the applications. So, one application may be social network like Facebook. So, Facebook we can consider each user has ignored, and suppose I want to find my all of friends. So, we have to explore the graph. So, or suppose there is a friend, there is a node user I want to see whether I can reach to that user through my friends. So, this is one application.

Another application is maybe garbage collection. So, most of the opening system is having this garbage collector like if we use some memory and if you no longer using that suppose we use calloc or malloc and then if you stop using that without freeing that suppose we are no longer using. So, it needs to be free, so that next time we can use this. So, basically we have to see whether we can reach from this to that space. So, if we cannot reach that means, that is not being used, so that is one application then there is web crawling.

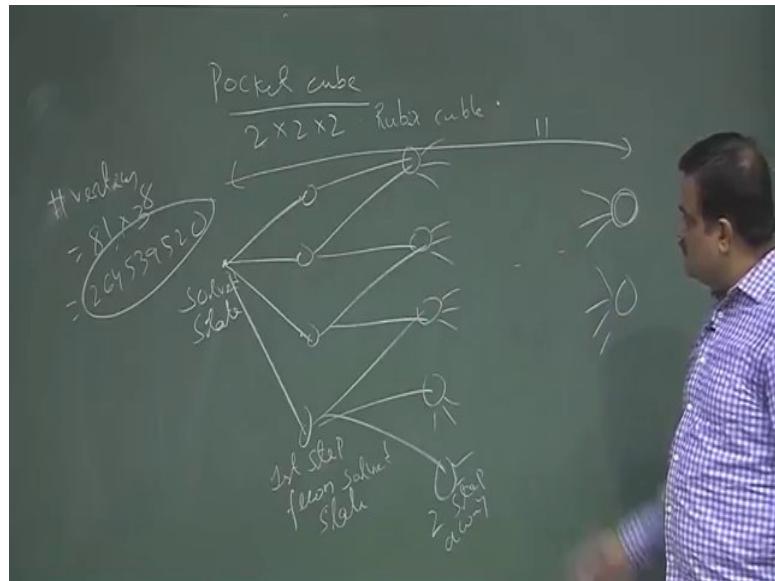
So, if we have a data, if we have a new web page I want to have access from the other. So, Google has to access all the new web pages, so that is also exploring a graph. And solving puzzle like rubix cube. So, I think I have a rubix cube let us bring the rubix cube. So, this is the rubix cube or this is called pocket cube. So, these are also some of the applications.

(Refer Slide Time: 21:54)



So, let us start with the rubix cube or the pocket cube, how this can be applied there? So, how we can associate this graph search with the pocket cube? So this is a 3 cross 3 cross 3 rubix cube, but suppose we take the lower version suppose we have a 2 cross 2 cross 2 pocket cube.

(Refer Slide Time: 22:01)



Suppose, we have a 2 cross 2 cross 2 pocket cube or this is rubix cube. So, how we can think, this rubix cube as a graph? We take each position as a node of the graph. If we change one little cube then it also changes another vertex in a cyclic manner. So, each position is one state, each state is a vertex of a graph and suppose we are in solve state.

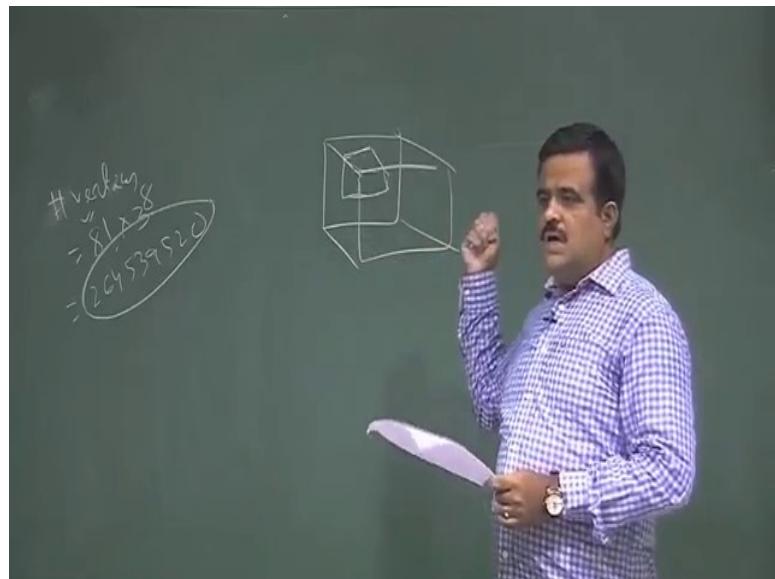
Now, if we take just one move then there will be another state, so that is the basically suppose we are at solve state; solve state means all the colors are matching. And at the solve state if we just have one change 90 degree change then that will be another state suppose these are the vertices which are basically these are the vertices which are basically one step away. So, these are the basically one step away from the solve state. That means, from this we can have one move to go to the solve state. So, some solve state we did one change to get this state. So, this is basically one step from solve state.

Similarly, if from there we can have two moves. So, this is basically two step array from the solve state like this. So, this is may be here. So, like this. So, this is the two step away from solve state. So, this way we will continue we will reach here A. So, this is the last state like this. So, this is an example of a breadth first search. So, this tree is called breadth first tree.

So, now, how many states are there in this a rubix cube 2 by 2 by 2, the number of vertices are basically factorial( $8 \times 3^8$ ). So, this many vertices we have, so this many possible states. So, this is huge number this is basically 264539520 this is the number of state of this graph, but

dimension is 11. There is a proof that one can use to show the dimension is 11 (this is called got number or the dimension of this 2 by 2 by 2 rubix cube.

(Refer Slide Time: 26:11)



Now, how to show this graph has this number of vertices? So, just try to have a 2 by 2 by 2 rubix cube. Now, if you take this one state like this and basically there are basically eight ways we can choose this, so eight ways. That means, so eight are the possible choices here, so factorial eight this is the possible ways and each way there are 8 dimensions and three ways we can change that. So, this is basically  $3^8$ , so that is why it is this. So from solve state, we have to reach to the another state the one step away, two step away like this and so on. So, if we do the other way round and that is called breadth first search. So, we will formally have the algorithm for breadth first search, so that we will do in the next class.

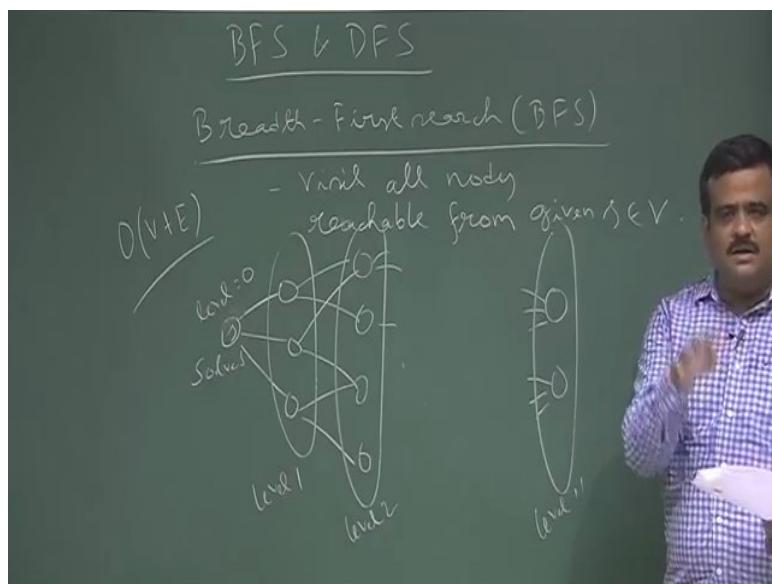
Thank you.

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

**Lecture – 41**  
**BFS & DFS**

So, first is BFS or Breadth-First Search. So, you are given a graph. Idea is to visit all the vertices reachable from given vertex ‘s’.

(Refer Slide Time: 00:20)

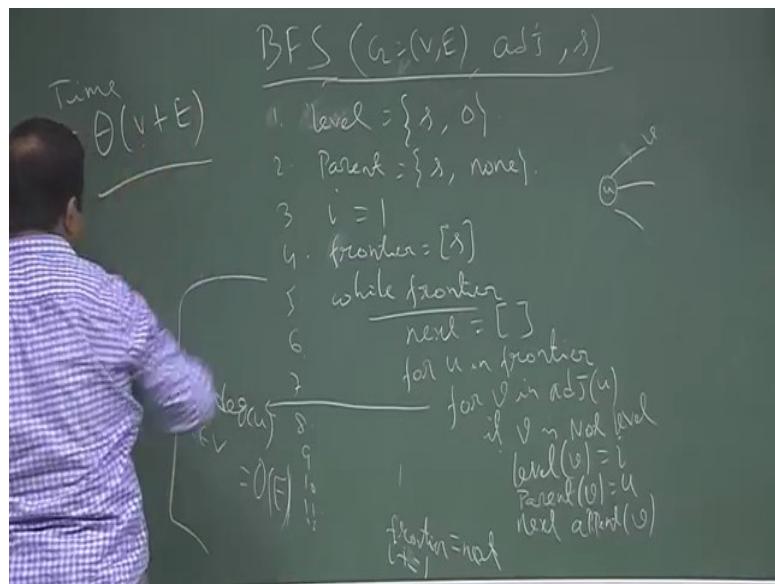


So, BFS is basically level wise search and this we want to do it in  $V+E$  times where  $V$  is the number of vertices and  $E$  is the number of edges.

So, basically we will look at all the vertices reachable from  $S$  and we will look at the adjacency list of each vertex. So, we start from ‘ $s$ ’ we first look at the adjacency list of ‘ $s$ ’. So, that will assigned as level 1. ‘ $s$ ’ is level 0. Then again when we capture all the vertices in level 1 then we go for level 2. Those are basically adjacent to the vertex which is in level one. So, those will be in the level 2. But we need to careful to avoid repetition.

If we have a vertex which is already marked that should not be marked again. So, let us write the code for this. So, the idea is basically level by level search which are reachable from ‘ $s$ ’.

(Refer Slide Time: 03:57)



So, the input is a graph  $G=(V,E)$  and we have a adjacency list and we have a starting vertex.

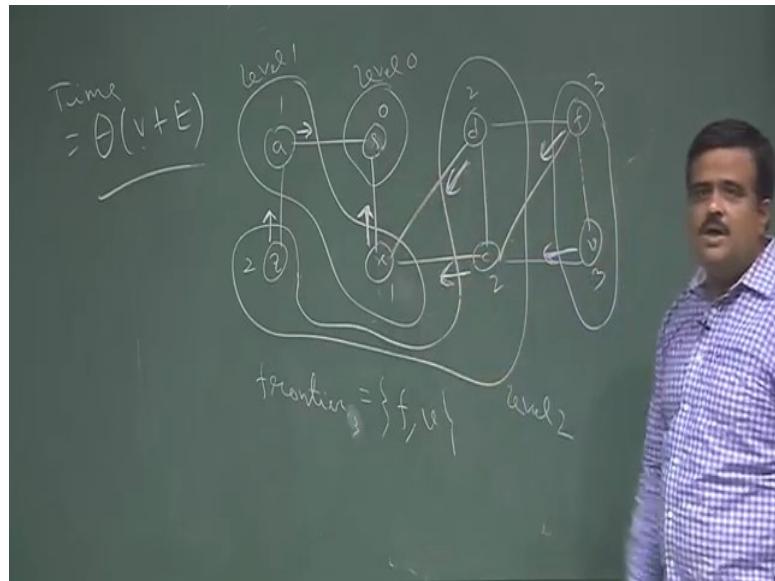
1. We level the vertex 's' as 0.
2. Parent of s as none because that is the starting vertex.
3. We assign i to be 1, i will indicate the current level
4. We assign another set called frontier, Initially it is 's'
5. Then we have a loop 'while frontier' that means, while frontier is not empty.
6. next is assigned empty (next stores the next level vertex)
7. Then we take an element from the frontier for u in frontier
8. We take the all adjacency vertex of 'u' that is 'v'
9. If v is not level; that means v is not mark or not seen; then level of 'v'= i and parent of 'v'=u and we have to append this v in the next.
10. After this the frontier will be next and i = i++

This is the pseudo code.

What is the time complexity?

We are visiting all the vertices and edges only once. So, the time complexity is  $V+E$ .

(Refer Slide Time: 10:01)



Now, let us take an example(refer above image) of this algorithm. So, suppose we have a graph ‘a, s, x, d, c, f, v’.

We have to run BFS on this input. So, it is a level by level search.

So, we start with the vertex ‘s’. Frontier is ‘s’ initially so; we level this vertex as 0. Now we check frontier is empty. No, frontier is not empty. So, we extract ‘s’, this is the u vertex now we check all the adjacency vertex of this. So, adjacency vertex of this is ‘a’ and ‘x’. So, this is the next level, i.e 1.

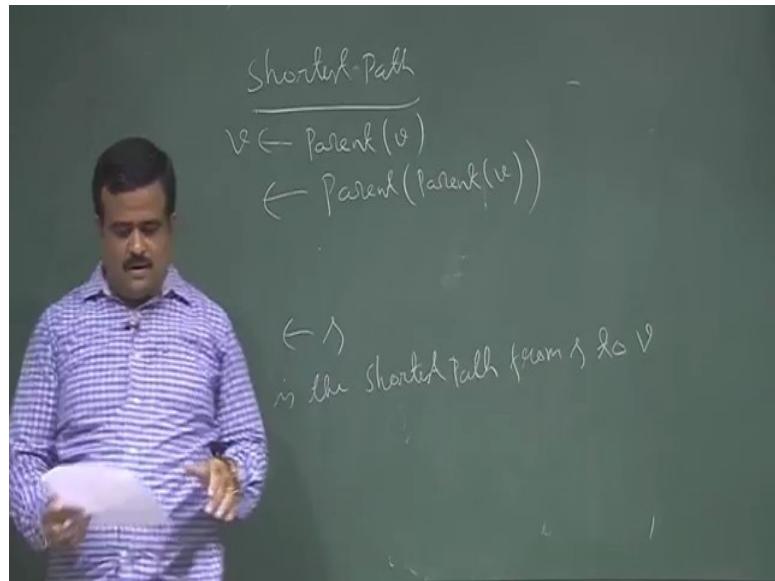
Frontier of level 1 is ‘a’, ‘x’. Now we have to check all the adjacency vertices of ‘a’ and ‘x’. We want to avoid the duplicate. So, ‘z’, ‘d’, ‘c’ are in level 2 whereas ‘s’ is ignored.

Now we see level 2 vertices. So, ‘f’, ‘v’ are in level 3 now, other adjacent vertices have been seen.

So, all vertices are done.

We have also maintained the parent pointer. Now, this parent pointer has a importance. So, what is that? Now parent pointer we will give us the shortest path from that node to ‘s’. So, how to find the shortest path? So, basically if we want to find the shortest path from f to s. So, we have to follow the parent pointers.

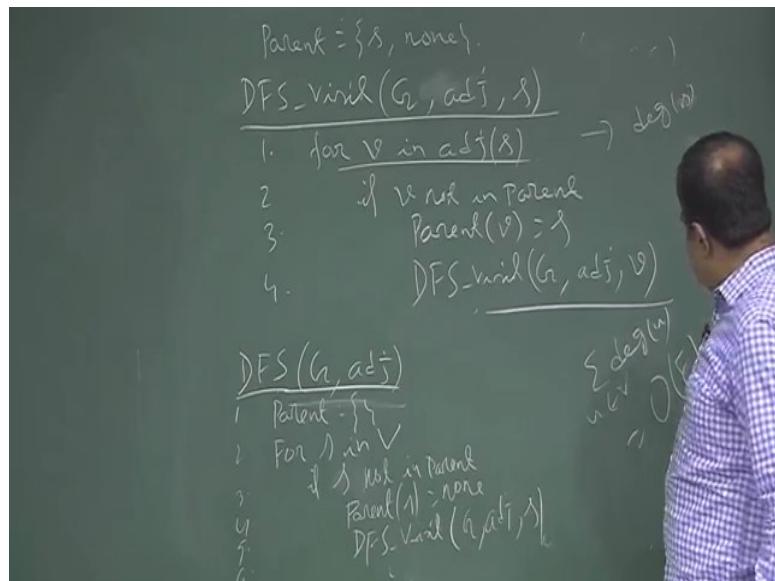
(Refer Slide Time: 15:55)



Now, this is the shortest path from 's' in the sense of number of edges visited.

Now we will talk about depth first search. So, this is also another way to explore a graph. So, this is called DFS.

(Refer Slide Time: 17:16)



So, DFS has many application like edge-classification and topological sorting. The idea is to use backtracking when we are stuck, we start from a vertex we follow all a path we just keep on going until we are stuck.

Now, once we stuck then we will backtrack to previous position and then we choose another path and we continue until we are stuck again. We have to be careful that there should not be any repetition.

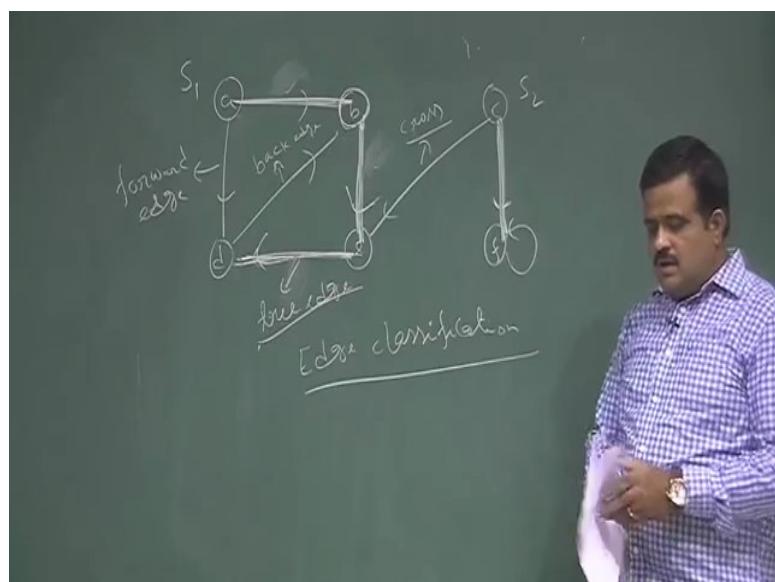
So, let us just write the pseudocode for this. So, it has 2 functions:

1. DFS\_visit. So, DFS\_visit is having an input as source vertex('s'). We check all the adjacency list of 's'. So, if v not in parent; that means, if v is not seen before then we will label parent of v as 's' and we call DFS\_visit on this 'v'. So, this is the recursive call, very simple code.

2. Main code is DFS which is taking the graph and the adjacency list. So, no starting vertex. So, we have parent as empty and then we choose an arbitrary vertex as s, for 's' in V if s is not in parent; that means, s is not explored then we just assign parent of s as null and we call DFS\_visit on s.

What is the time complexity for this? Basically summation of degree of  $u \in V$ . So, this is basically  $O(E)$  and since we are visiting all the vertices. So, this is basically order of  $V+E$  because we have to visit all the vertices.

(Refer Slide Time: 24:38)



Now, let us take a quick example. We have a graph 'a, b, c, d, e, f'.

And then suppose we have these edges, this is a directed graph. We want to explore the DFS on this. So, now, we have to choose vertex.

Let 'a' be the first vertex. So, this is our starting vertex say  $s_1$ . We call again DFS\_visit on this. So, this is having adjacent vertex 'b', we go there. So, again 'e' is adjacent with 'b' we go there. Again 'd' is adjacent to 'e', we go there. But 'd' is having a vertex adjacent to it which is already in the parent. So, we stop here.

Once we stop here, we come back to DFS. So, that is the backtracking.

Now we choose another vertex from this graph which is not marked. Let that vertex be 'c'. So, we call DFS\_visit on 'c'. Again, 'f' is adjacent to 'c' and not visited. We stop at 'f' as it does not have any adjacent vertex which is not visited. So, this is the DFS search.

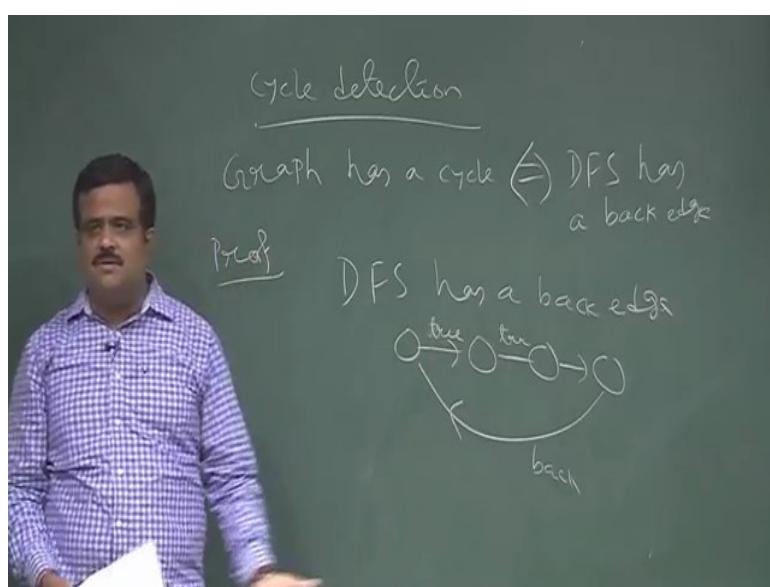
Now, we can label the edges. So, we can classify the edge. Edges which were found in DFS traversal are called tree edges. Non-tree edge is either forward edge, back edge or cross edge.

Forward edges are which go from an ancestor to a descendant, in our example edge (a,d) is a forward edge as 'a' is an ancestor of 'd'.

Back edges are which go from a descendant to an ancestor, in our example edge (d,b) is a forward edge as 'd' is a descendant of 'b'. Edges other than these are called cross-edges.

So, this will help us to have some application like how to find the cycle detection.

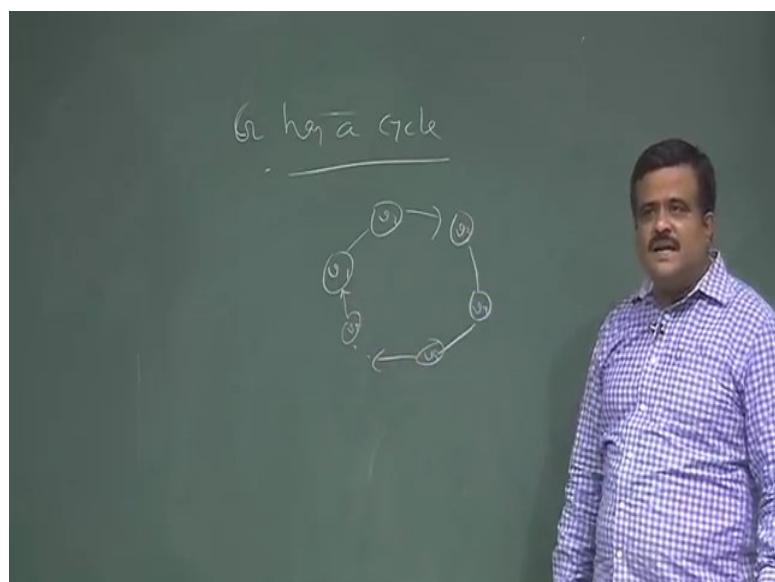
(Refer Slide Time: 30:55)



So, one application is cycle detection. So, this is the theorem. If there is a back edge then there is a cycle. So, given a graph has a cycle implies if we run the DFS, the DFS has a back edge. This is also true vice versa. So, if DFS has a back edge then it has a cycle.

Let us try to prove this. So, first we assume the DFS has a back edge. That means, we have tree edges and we have a back edge. So, it will form a cycle(refer above image). So, that is quite clear. So, if the DFS has a back edge then there is a cycle. Now, we have to prove if there is a cycle then the DFS has a back edge.

(Refer Slide Time: 32:45)



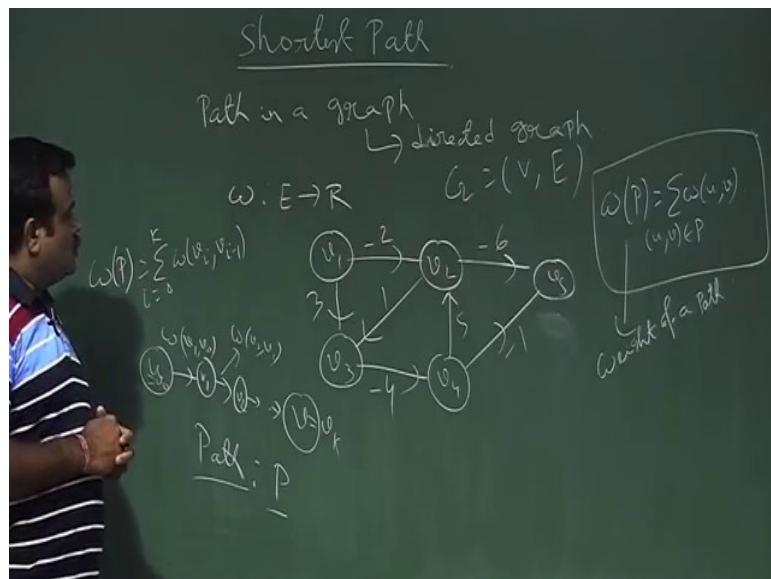
Suppose a graph  $G$  has a cycle. Suppose this is a cycle say we have  $v_1, v_2, v_3, \dots, v_k$ . Suppose there is a cycle. Let us start DFS at  $v_1$ . So, in that case there will be tree edges till  $v_k$ . But,  $v_k$  to  $v_1$  can not be a tree edge as  $v_1$  has been visited so it is a back edge. So, this is one application to find the cycle in a graph there are other applications also like topological sorting and job scheduling algorithm. So, those are in the textbook.

Thank you.

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

**Lecture - 42**  
**Shortest Path Problem**

(Refer Slide Time: 00:23)



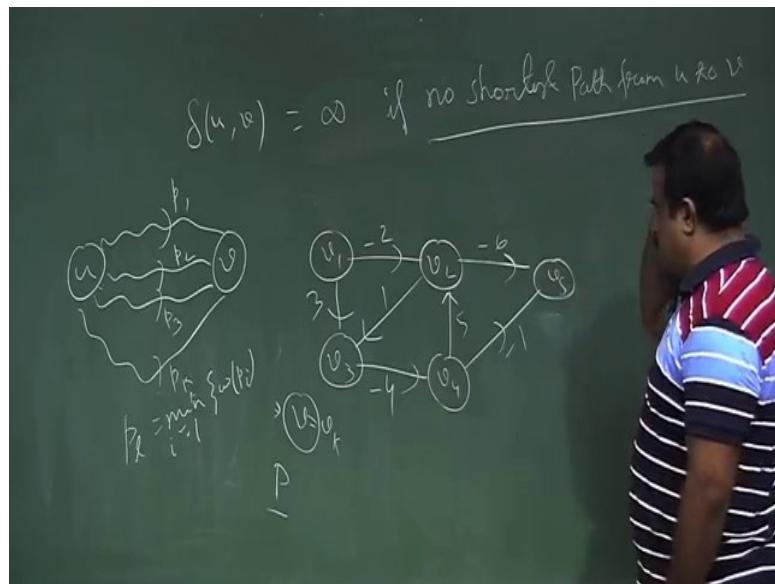
We talked about shortest path problem in a graph.

We have a graph  $G = (V, E)$  and a weight function. Weight function, means that we have a weight of each edge in  $G$ .

Weight of a path means the sum of the weight of edges on that path.

Now, the question is of finding the shortest path. When can we have a shortest path from a vertex 'u' to 'v'. First of all there has to be a physical path from 'u' to 'v'. There may be many paths from 'u' to 'v'. Among these path whichever will have the minimum weight that will be the shortest path from 'u' to 'v'.

(Refer Slide Time: 05:48)

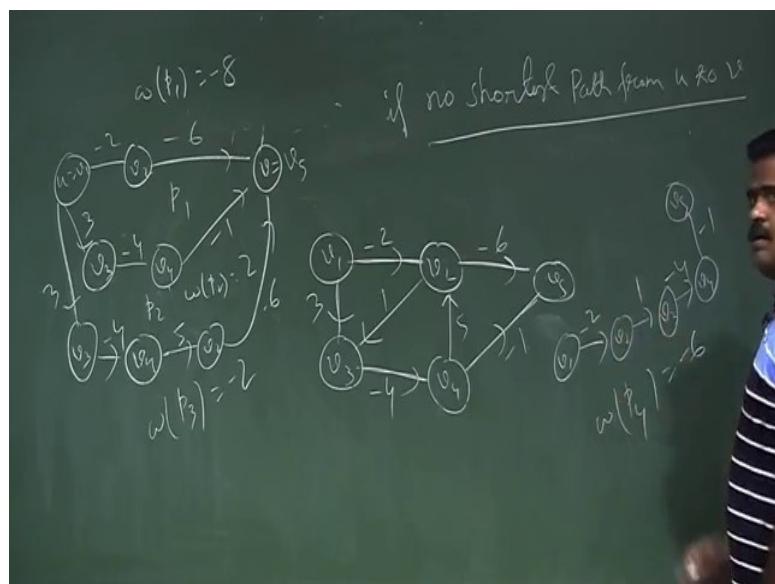


Weight of a path( $P$ ) is:  
 $w(P) = \sum w(u,v)$ , where  $(u,v) \in P$ .

If there is no shortest path from 'u' to 'v' then we denote this by infinity.

Another option such that no shortest path will exist is to have negative cycle in the graph.

(Refer Slide Time: 08:29)

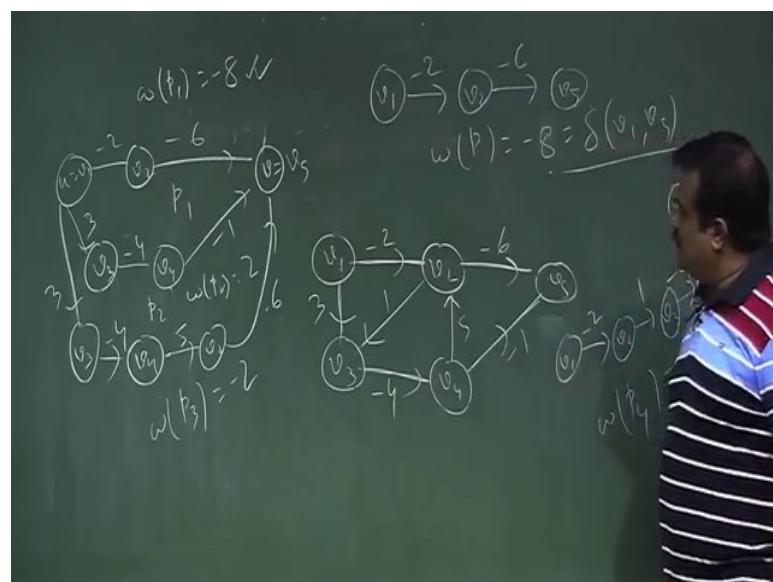


Let us take this example(refer above image). We need to find shortest path between  $v_1$  and  $v_5$ .

Let us explore some paths:  
 $P_1 = v_1 \rightarrow v_2 \rightarrow v_5$ .  $w(P_1) = (-2) + (-6) = -8.$   
 $P_2 = v_1 \rightarrow v_3 \rightarrow v_4 \rightarrow v_5$ .  $w(P_2) = 3 + (-4) + (-1) = -2.$

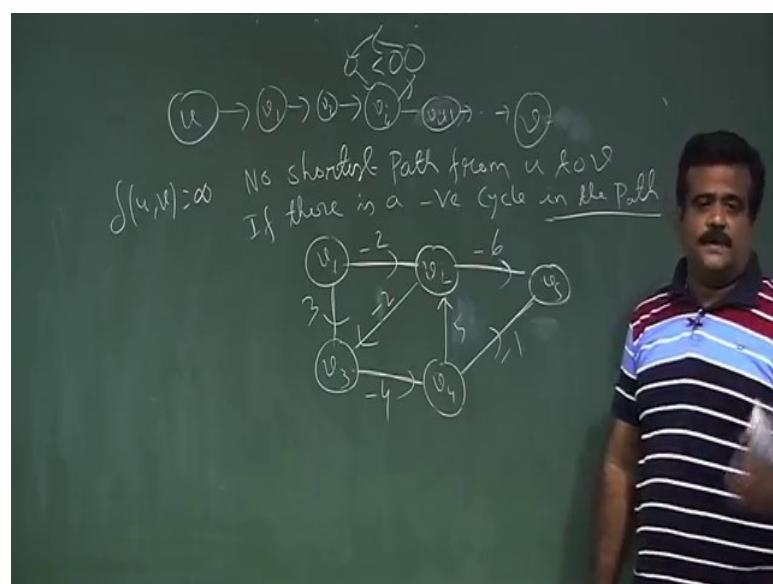
Similarly we can have multiple paths. After going through multiple paths, we see that  $P_1$  is the shortest path.

(Refer Slide Time: 13:29)



Our problem is given any two vertex we need to find the shortest path if it exists. Shortest path will not exist even if a physical path is present, in the case of negative cycle.

(Refer Slide Time: 14:37)

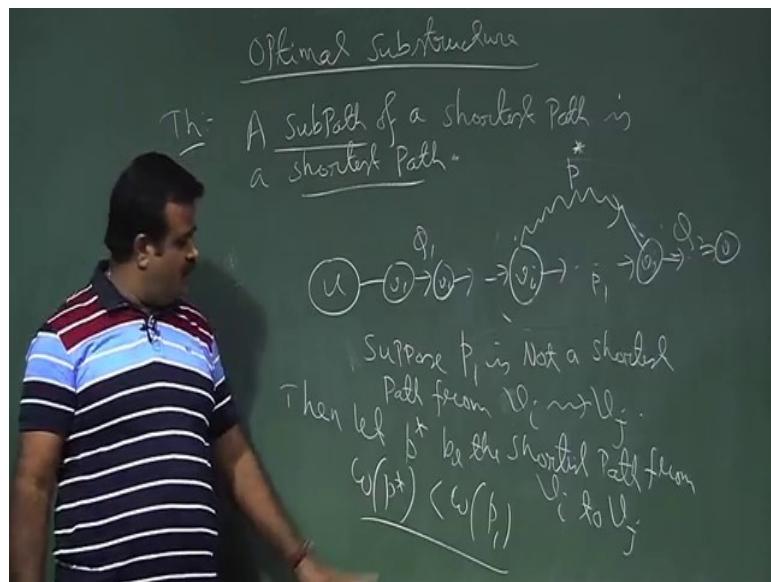


In our example let us make  $w(v_2, v_3) = -2$ . Then we have a negative cycle  $v_2 \rightarrow v_3 \rightarrow v_4 \rightarrow v_2$ .

If we say  $v_2 \rightarrow v_3 \rightarrow v_4 \rightarrow v_2$  will be my shortest path then one can take another loop of the path and claim the same. For, this reason a shortest path can not exist.

We want to see whether we can apply dynamic programming technique to find shortest path. So, for that we want to see whether there is a optimal substructure design in the problem.

(Refer Slide Time: 17:54)



First hallmark of dynamic programming technique is optimal substructure. That means, if we have a solution of a problem then it contains solution of the sub-problems of the whole problem.

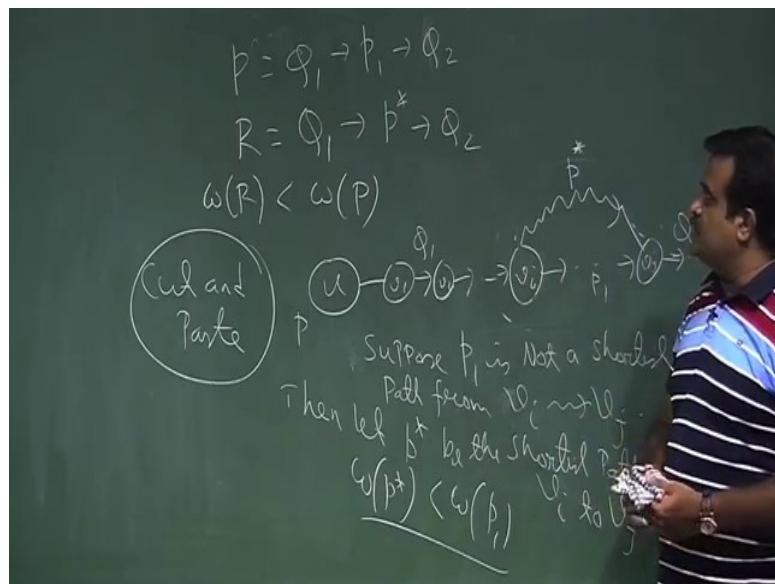
How do we establish this hallmark?

We need to prove that a subpath of a shortest path is the shortest path possible. How to prove this?

Let us assume that shortest path 'u' and 'v' consists of  $v_1, v_2, v_3, \dots, v_i, \dots, v_j$ . If we take a subpath,  $v_i, \dots, v_j$ . We need to prove that this will be the shortest path from  $v_i$  to  $v_j$ .

Let  $P_1$  be the current path between  $v_i$  to  $v_j$ . Suppose  $P_1$  is not the shortest path. So, there is a path which is shortest path. Let us denote that by  $P^*$ . Let  $Q_1$  be the current path between 'u' to ' $v_{i-1}$ ' and  $Q_2$  be the current path between ' $v_{j+1}$ ' to 'v'.

(Refer Slide Time: 23:37)



That means,

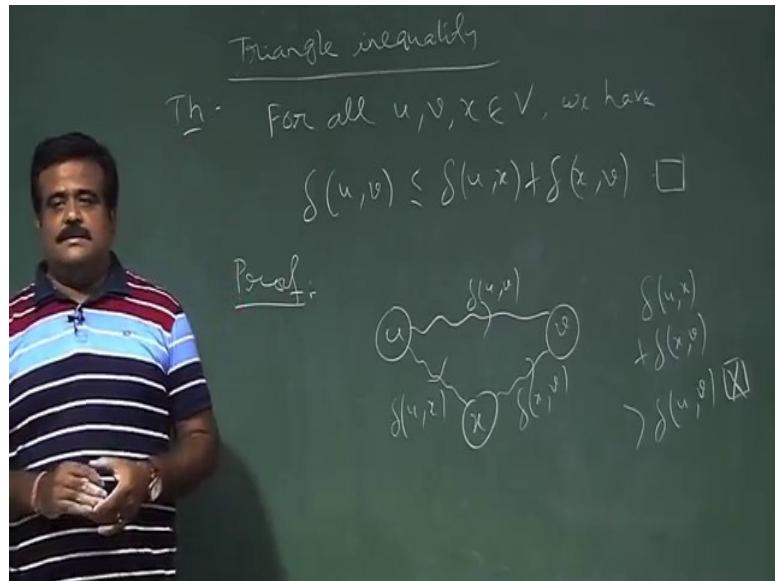
$$Q_1 + P^* + Q_2 < Q_1 + P_1 + Q_2 \text{ as } P^* < P_1$$

But this is in contradiction to the fact that  $Q_1 + P_1 + Q_2$  is the shortest path. Hence, there is a optimal substructure

Second hallmark is overlapping subproblem.

We will see whether greedy algorithm can be applied or not.

(Refer Slide Time: 25:31)



For that we will see triangle inequality. Let there be 3 vertices as shown in above diagram.  
So, the theorem tells us,  
 $\delta(u, v) \leq \delta(u, x) + \delta(x, v)$  where  $u, v, x \in V$  and  $\delta(u, v)$  stands for shortest path between 'u' and 'v'.

Let us prove this. This theorem is straight forward from the above image because if we take delta to be the weight of the shortest path. So, weight of the shortest path must be less than that of any other path.

If we take path  $u \rightarrow x \rightarrow v$  to reach from 'u' to 'v' then weight of the path will be  $\delta(u, x) + \delta(x, v)$ . So, this path has to be greater than the shortest path from 'u' to 'v'. We will use this inequality to have an algorithm in finding the single source shortest path.

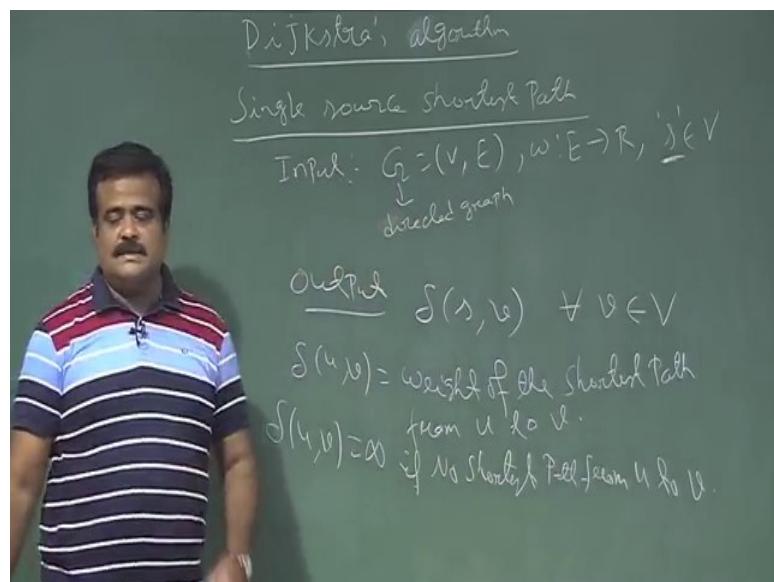
Thank you.

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

**Lecture – 43**  
**Dijktra's**

So we will talk about one algorithm to find the shortest paths, specially the single source shortest path.

(Refer Slide Time: 00:25)



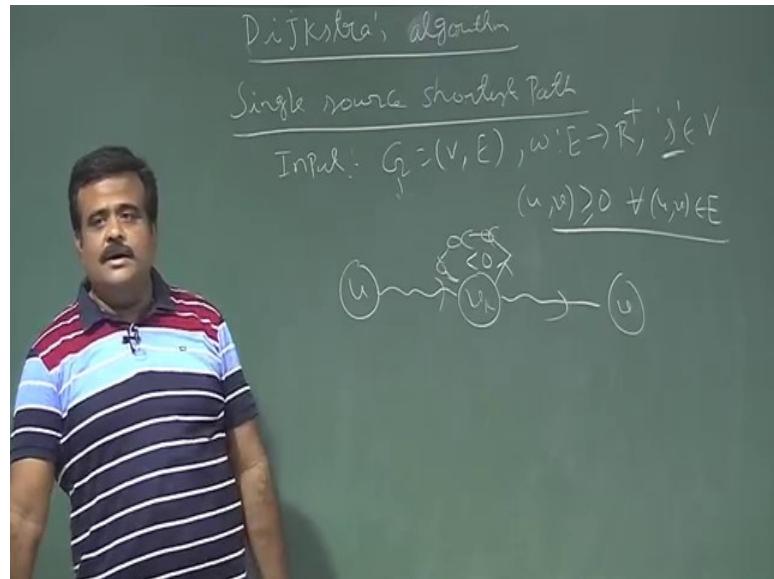
Single source means we have been given the source; i.e one vertex is input.

Input is a directed graph;  
 $G = (V, E)$  where  $V$ : Set of vertices,  $E$ : Set of edges  
 $w: E \rightarrow R$  where  $w$  is the weight function from  $E$  to  $R$   
 $s \in V$  where  $s$  is the source vertex

The problem is to find the shortest path from that source to all other vertices. So, output will be the weight of the shortest path. Output is  $\delta(s, v) \forall v \in V$  where  $\delta(s, v)$  is the weight of shortest path from vertex 's' to vertex 'v'.

$\delta(u,v) = \infty$  if there is no shortest path from u to v. There will be no shortest path from u to v if there is no physical path from u to v or there is a physical path with a negative cycle.

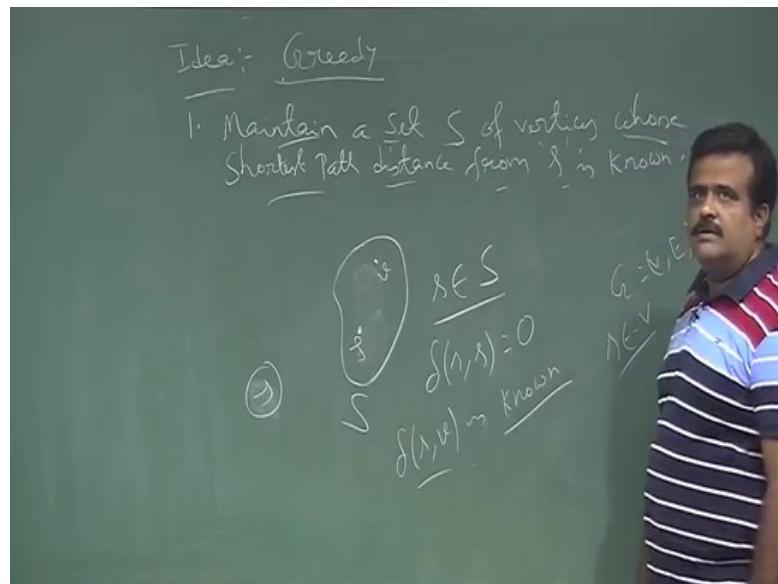
(Refer Slide Time: 02:50)



So, if there is a negative cycle between the path from u to v then there will be no shortest path. So, u to v if there is a path, but there is a vertex  $v_k$  such that there is a negative cycle in the path. If there is a negative cycle then there is no shortest path because we can loop in this cycle. If we claim this is my shortest path then we can make another loop and minimize the weight.

So, this algorithm cannot handle negative cycle. So, that is why in this algorithm we assume weight to be non-negative. So, we assume this weight function to be non-negative. So, this is one of the assumptions for Dijkstra's algorithm.

(Refer Slide Time: 04:35)



This is a greedy approach.

So, what do we do?

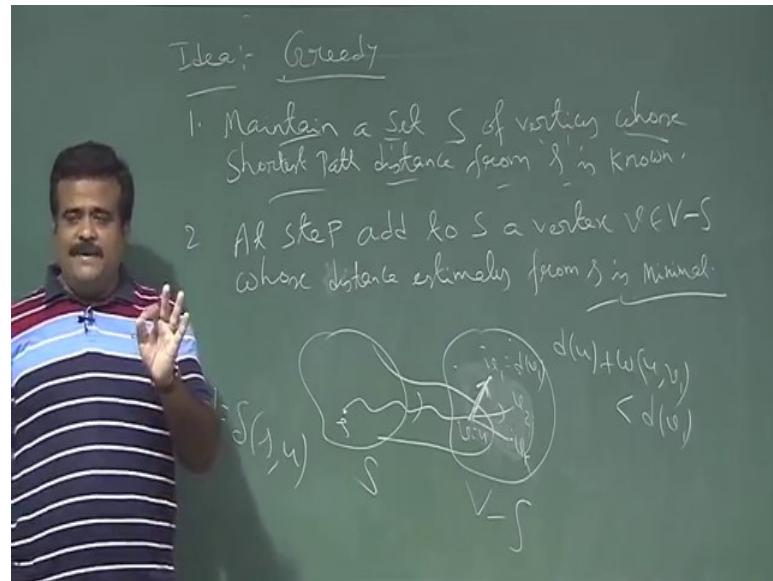
We maintain a priority queue. We maintain a set  $S$  of the vertices whose shortest path, whose shortest path distance from source is known.

Obviously 's' will be in set  $S$ , why?

Because  $\delta(s,s)=0$  as we do not have any negative cycle.

's' will be the first vertex which will be in  $S$ . Then slowly we will capture all the vertices with  $S$ . So, that is the greedy way. So, we maintain the set  $S$  of all vertices whose shortest path distance from 's' is known.

(Refer Slide Time: 07:49)



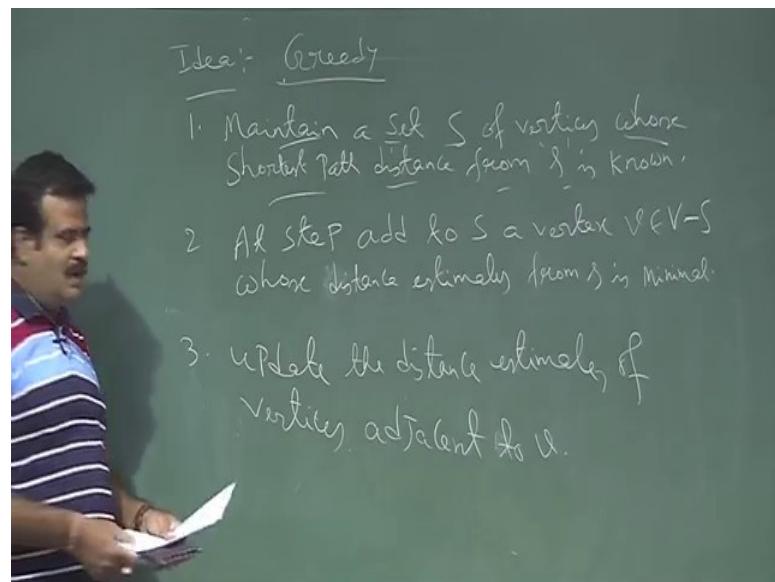
How to grow this S?

This is the second step. At each step add a vertex to S from V-S, whose distance estimate from S is minimal. This is the greedy choice. So, we have the set S and V-S. Now, we consider all the vertices in V-S whose  $d(u)$  (where  $d(u)$  is the distance estimate from 's' to this node 'u') is minimal and that is the greedy choice. And that vertex('u') we take into S from Q(a priority queue in which all other vertices other than S are kept and then we are extracting the minimum from this Q based on distance estimate). Then we know all the vertices which are connected to 'u'. So, we have all the vertices which are connected to this 'u'.

So, this was having a distance estimate  $d(u)$ . So, once it is captured in S that means,  $\delta(s,u) = d(u)$ . Now the vertices which are directly connected with this vertex('u'), now these vertices were having some distance estimate. What is the weight of this new path to  $v_1$ ? It is  $d(u) + w(u,v_1)$ . If this is less than the distance estimate of ' $v_1$ '. Then we have to update this so we get a better path.

So, this means we have a path from 's' to that vertex  $v_1$  with the cost  $d(v_1)$ . Now we have a new path once we capture this('u') in S then we have a new path we can go from 's' to 'u' then ' $v_1$ '. And if this path is having less weight than the earlier degree then we must update the new path. So, that is called relaxation. So, relaxation means if we are updating the degree of this by a new path.

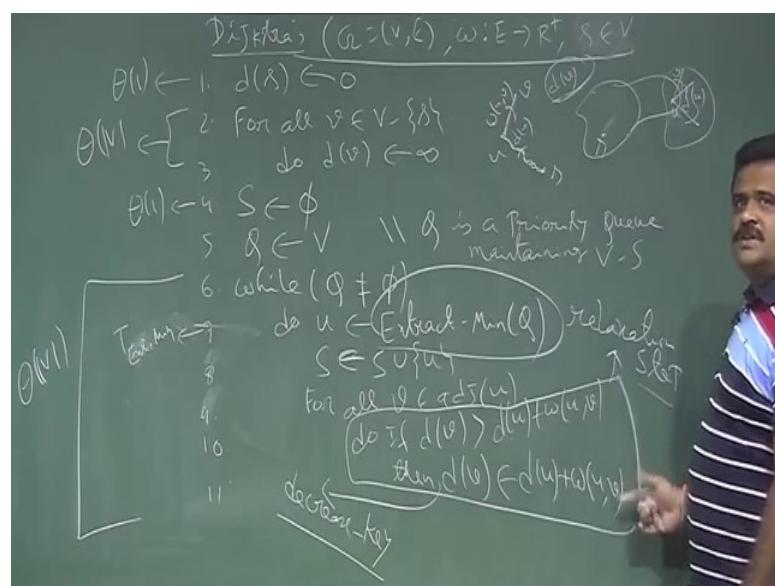
(Refer Slide Time: 12:35)



So, this is the idea of behind Dijkstra's algorithm.

So, let us write the pseudo code for dijkstra's. So, the idea is greedy.

(Refer Slide Time: 13:46)



The input is a directed graph and we have a edge weight which is non-negative. And then we have a source vertex. So, this is a single source shortest path.

So, we are defining degree of 's' to be 0. For all other vertex 'v' we do  $d(v) = \infty$ . This is same as the prim's algorithm. So, we are putting the degree to be infinity for all other

vertices because we haven't explored them yet. Queue is like we did in prim's algorithm, key is the degree of that vertex. And every time we extract a minimum from this Queue until the Queue is empty. So, this is the operation while Queue is not null we do extract minimum from this Queue.

So, extracting means we are deleting from the Queue, and we are adding in S. This is a 'u' which is having the minimum degree. So, we capture this in S. Then we have to update the distance estimate in all above the vertices which are adjacent to 'u' as discussed above.

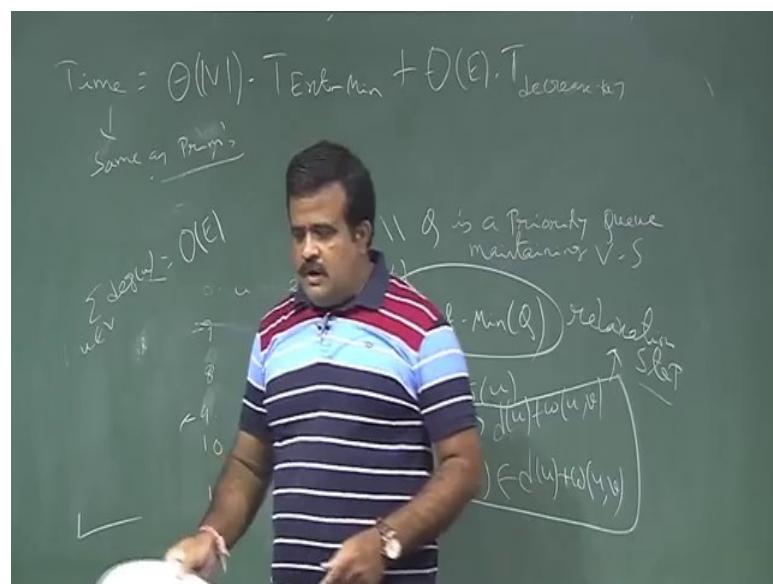
So, this is the pseudo code for Dijkstra's algorithm(refer above image).

We have a extract min step. So, depending on how we implement this in a data structure it will take the time. Time complexity :

Takes  $\Theta(V)$  for initialization. Then we run a loop V times. Inside the loop we are doing the extract minimum. If we use the min heap as discussed in prim's algorithm analysis it will take logarithm time. Then we decrease the key. We need to heapify that. Time complexity of this is same as prim's algorithm time complexity.

First part is order of V times extract minimum cost.

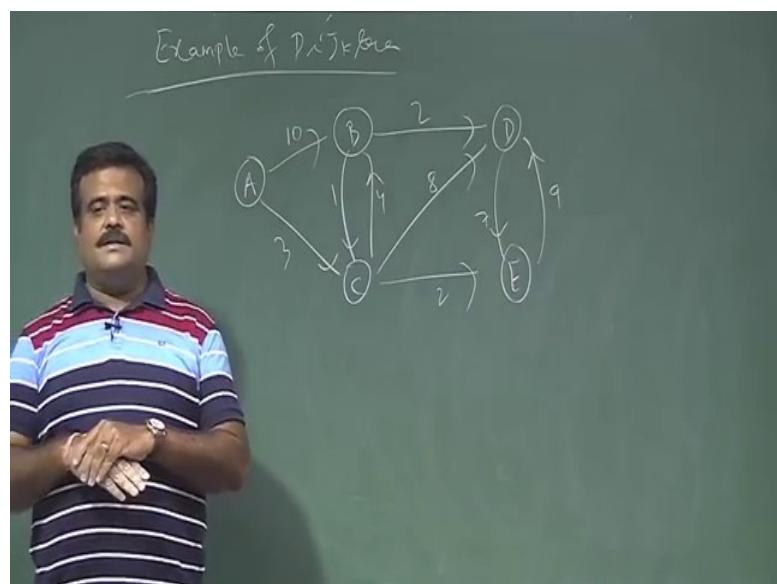
(Refer Slide Time: 23:23)



Now summation of degrees is the handshaking lemma  $O(E)$ . So, this is basically  $O(E)*\text{time}$  to decrease key. So, this time complexity is same as the prim's algorithm. So, if we use heap then this will be  $O(\log(V))$ . So, second part will be  $O(E\log(V))$ .

Now we will talk about example for this dijkstra's algorithm. So, let us take an example of a graph and on which we can have the dijkstra's algorithm.

(Refer Slide Time: 25:25)



So, example of Dijkstra's ok. So, we take a graph. So, let us take this graph. So, these are the weight 3 2 1 4 8 2 7 9. So, this is given graph. And this is a directed graph. So, there is no negative edge. Then there will be a if there will be a path from 's' to vertex. Otherwise that path will be infinity if there is no physical path from S to that vertices. So, we will continue this in the next class.

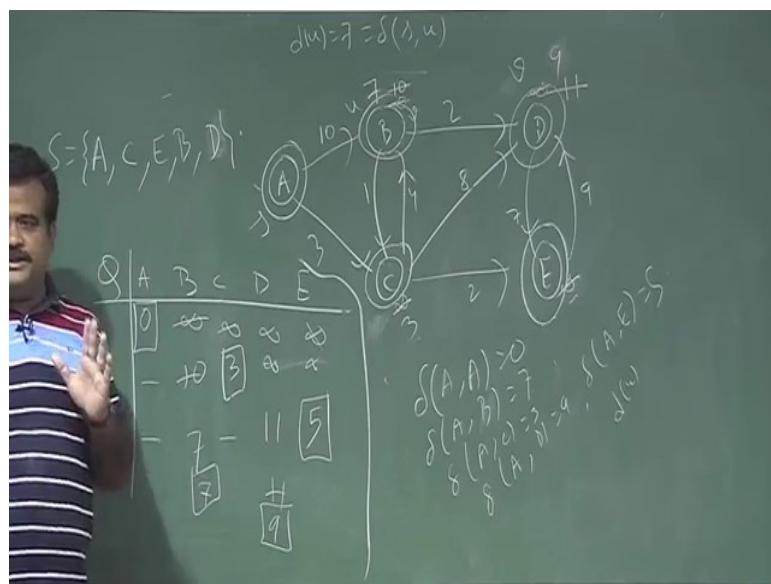
Thank you.

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

**Lecture - 44**  
**Example Of Dijktra**

We are talking about Dijktra's algorithm. So, we want to work out the Dijktra's algorithm on this given input.

(Refer Slide Time: 00:17)



Just to recap Dijkstra's algorithm, we are starting with the vertex 's' that is also a part of the input. So, given a vertex s we have to find out the shortest path from all the vertices, ok.

So, basically we are just maintaining this QUEUE. So, our QUEUE is basically 'A B C D E'. So, we are putting everything into the QUEUE. We are just putting the degree of 's' as 0 and for all other vertices V-S we are putting degree to infinity, because this is a initialization step.

The idea is we capture one; each time we just capture one vertex in S, we update the distance estimate of that vertex which are adjacent to the vertex which we have recently captured. Now, we do the extract minimum. So, we do the extract minimum. So, extract minimum from the QUEUE. Now, everything is infinity except souce. So, 's' will be our u. So, S will be S union u. So, S will be 'A' because it was empty. So, this is our u. So, we look at all the vertices which are adjacent to u. Now, B's degree was infinity, now we have  $d(A)=0$ ; so

$0+w(A,B) = 10$ . So, 10 is lesser than infinity. So, we have to decrease the key. Similarly now 'C' is v which is another adjacency vertex of 'A'. Now it was having a degree infinity, now we have a path we can go from A to C with the cost 3.

Again we have to check whether QUEUE is empty. Now we have to extract minimum again. So, now, if we extract minimum, so then this 3 is the minimum, now 'C' is our u.

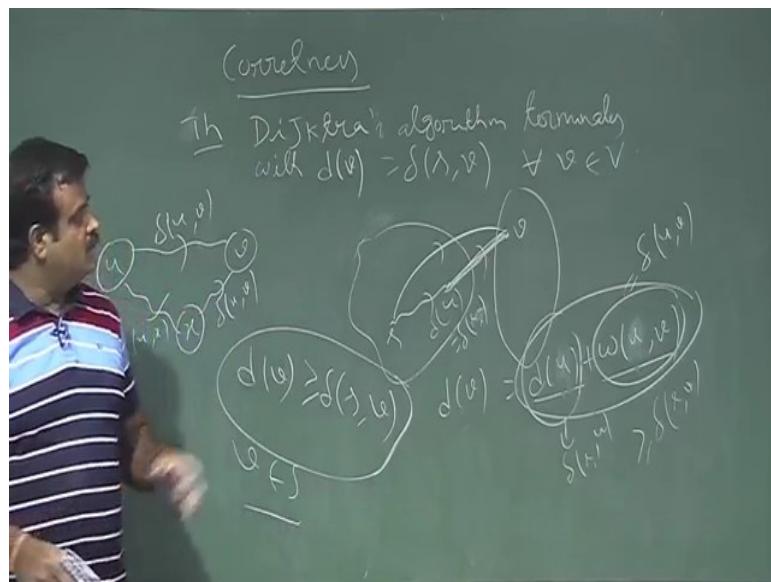
So 'C' is added in S. Now, this is our u, now we check all the adjacency vertex of this. So,  $d(E)$  now becomes 5.  $d(D)$  becomes 11. Also  $d(B)$  becomes 7 as 7 is less than 10.

Now we extract the next minimum, next minimum is 5('E'). We capture this in S. Now, we will look at the adjacent vertices. So,  $d(D)$  will remain 11 as the new path cost is 14 which is greater than current degree.

Next minimum is 7('B'). So, this is our u. Now let us consider all the adjacency vertices of this node in V-S. Now,  $d(D)$  will become 9.

We again call extract min, we are going to be add D in S. This QUEUE is empty now because all the vertices are now deleted from this QUEUE. So, this is the Dijkstra's algorithm execution.

Here we are assuming no negative cycle. So, let us go for a quick correctness of this theorem. So, the idea is we just grow this S, we capture one vertex in S whose distance estimate is minimum. So, distance estimate from 's' to that vertex is minimum and that is the greedy choice.



So, Dijkstra's algorithm terminates when we have  $d(v) = \delta(s, v) \forall v \in V$ . So, this is what we are expecting, because when we capture a vertex in  $S$ . So, when we finish the execution of this algorithm then the degree is becoming  $\delta(s, v)$ . We can prove this. So, if you recall the Dijkstra's algorithm, we are maintaining a set  $S$ . So, first of all we are initializing degree of  $s$  is 0 and degree of all other vertex to be infinity.

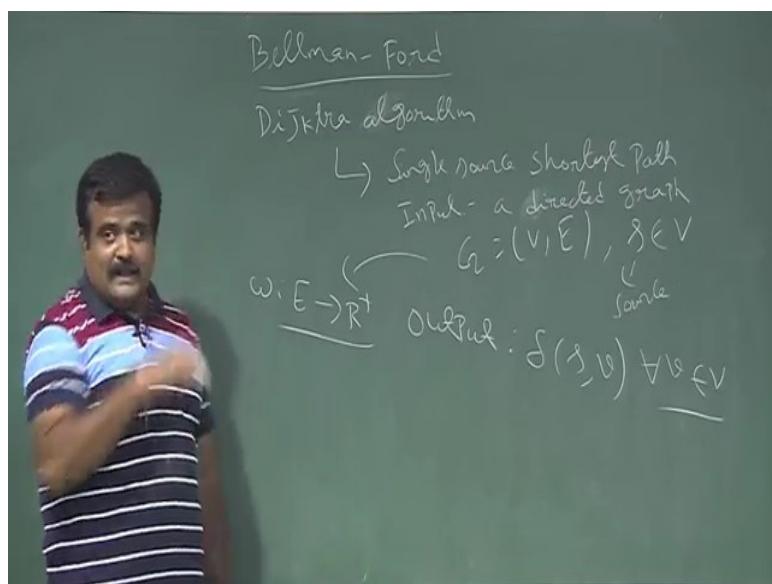
Now, in the first step  $S$  is becoming ' $s$ ' because this degree is 0. Now  $\delta(s, s) = 0$  because there is no negative cycle.

Now, we have ' $s$ ' in  $S$ , now we have a set  $V - S$ . Once we capture a vertex as  $u$  then we take a direct path to  $v$ . So, what is the degree of  $v$ ? So, degree of  $v$  is basically degree of  $u$  plus  $w(u, v)$ . So, initially this  $u$  is ' $s$ '.

So, my claim is  $d(v)$  must be greater than equal to  $\delta(s, v)$ . So, how to prove this? This can be proved by triangle inequality introduced in lecture 42.

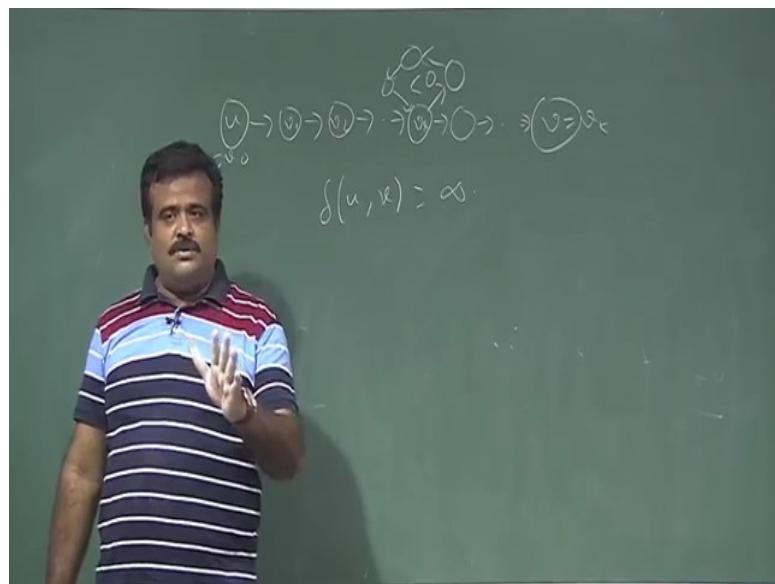
Now we will talk about another algorithm which can handle negative weight cycle, it is called Bellman-Ford algorithm.

(Refer Slide Time: 23:18)



We have already discussed problems due to negative weight cycle in Lecture 42.

(Refer Slide Time: 25:22)



In case of negative weight edge, we have  $\delta(s, v) = \infty$  as no shortest path is defined.

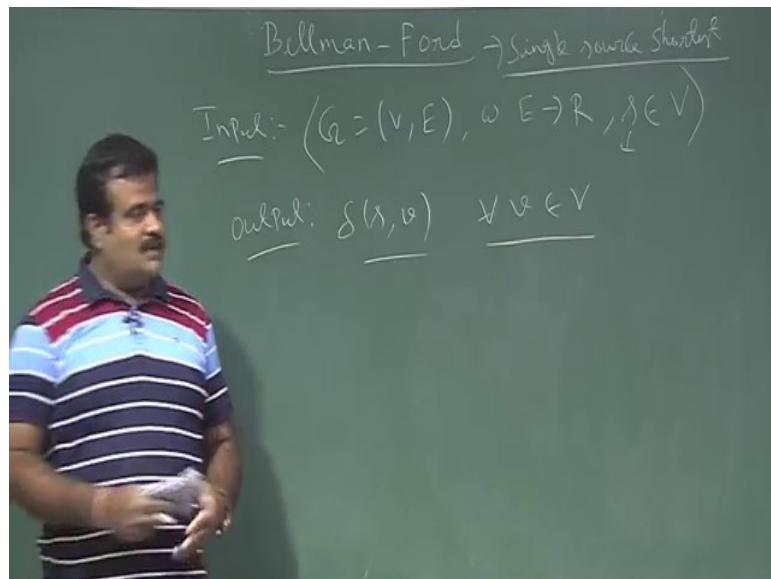
The algorithm which we will discuss in the next class is called Bellman-Ford algorithm. And Bellman-Ford algorithm can handle negative weight cycle. So, for Bellman-Ford algorithm or graph is any graph with the any edge weight.

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

**Lecture - 45**  
**Bellman Ford**

We will talk about Bellman-Ford algorithm.

(Refer Slide Time: 00:44)

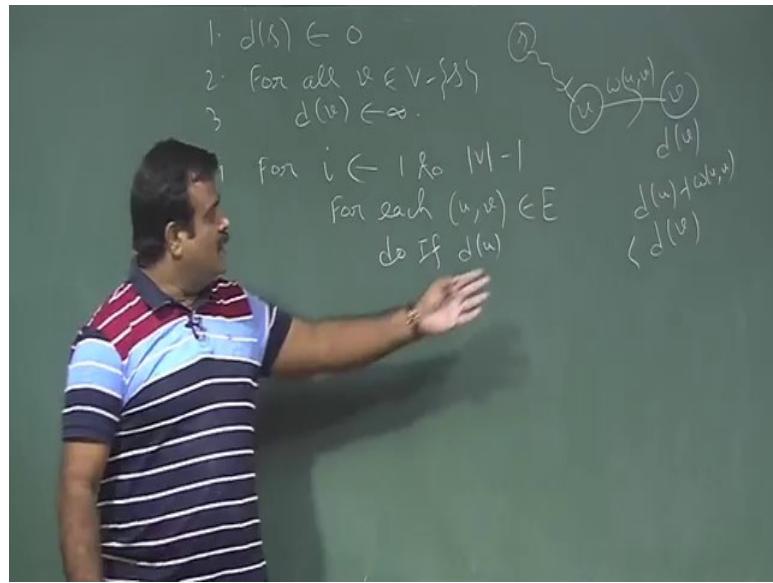


Input is a directed graph;  
 $G = (V, E)$  where  $V$ : Set of vertices,  $E$ : Set of edges  
 $w: E \rightarrow R$  where  $w$  is the weight function from  $E$  to  $R$   
 $s \in V$  where  $s$  is the source vertex

So, given a source we have to find the shortest path from that source to other vertices. Output is  $\delta(s, v) \forall v \in V$  where  $\delta(s, v)$  is the weight of shortest path from vertex 's' to vertex 'v'. If there is a negative cycle we have to report the negative cycle.

So, let us write the pseudo code for Bellman-Ford algorithm.

(Refer Slide Time: 02:23)

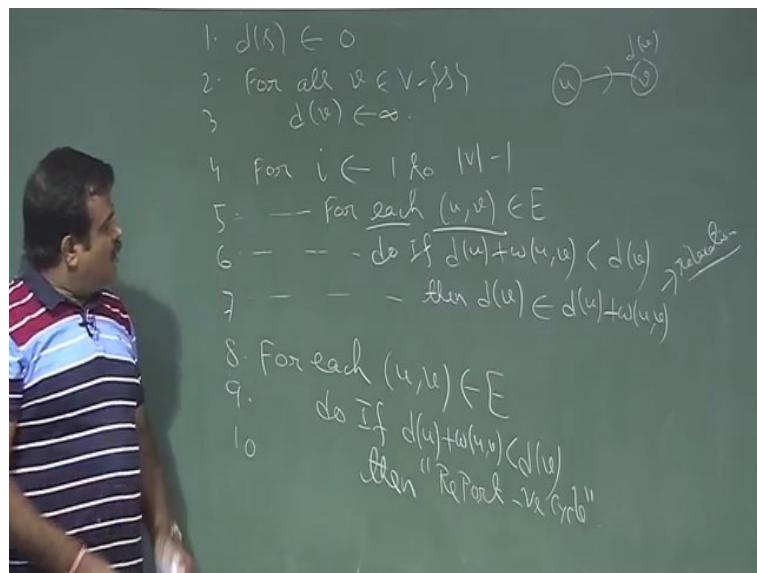


The initialization part is same as Dijkstra algorithm.

1.  $d(s') = 0$
2.  $d(v') = \infty \forall v \in V - \{s'\}$ .

We do the relaxation in the same manner as well. If  $d(u) + w(u, v) < d(v)$  then we relax that edge and make  $d(v) = d(u) + w(u, v)$ . We will do this step for every edge  $|V|-1$  times in Bellman-Ford (as seen in diagram below).

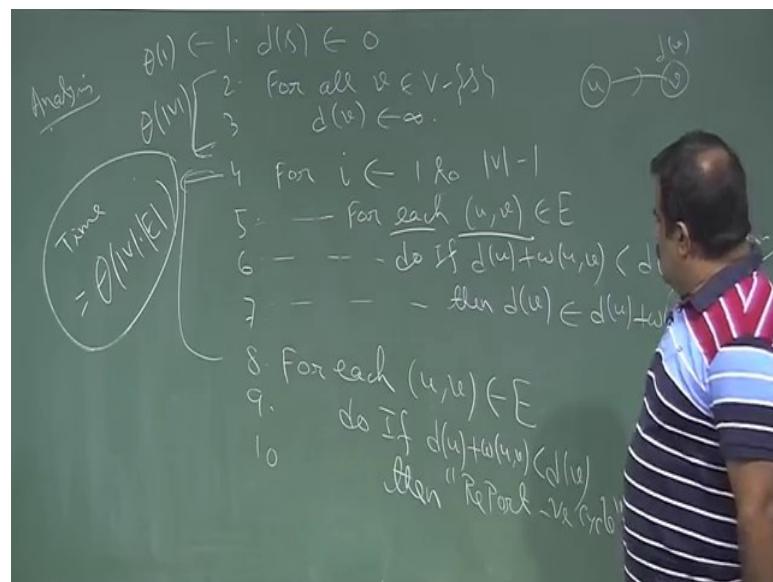
(Refer Slide Time: 04:34)



The last loop in the algorithm is to detect negative cycle. If there is no negative cycle, then the shortest paths must have converged by the last iteration of loop(4-7) in the above pseudo code. If there is anymore convergence, that would mean a negative cycle exists. Hence, we can check presence of negative edge via this algorithm

What is the time complexity?

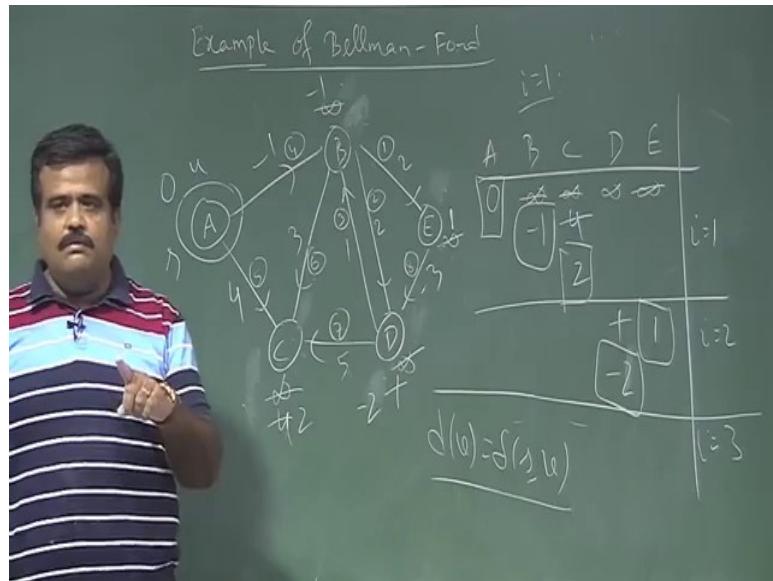
(Refer Slide Time: 09:41)



Time complexity boils down to the loop(4-7) which does  $O(E)$  iterations  $O(V)$  times. Hence, time complexity is  $O(VE)$ .

Now, let us take an example of how this algorithm works.

(Refer Slide Time: 10:42)



Let us take a directed graph 'A B C D E' as shown above.

So, this is the input of the Bellman-Ford algorithm, now we take a source vertex. Suppose 'A' is our source vertex. So, source vertex has degree 0 and others have infinity. Now we have to run this for we have a outer loop from  $i = 1$  to  $(|v|-1)$  times and in inner loop we have to run it for each edge. So, somehow if we have an edge numbering that will help us.

There are total 8 edges(see numbering in the above image). Let us start with edge (B,E). No relaxation is possible as both  $d(E)$  and  $d(B)$  are infinity. Similar for edge 2 and 3. Edge 4 is E(A,B), here  $d(A) + w(A,B) < d(B)$ . So, we will do relaxation here. Now,  $d(B)$  becomes -1.

Similarly at edge 5, we will update  $d(C) = 4$ . Then, at edge 6, we will again update  $d(C) = 2$ . But there will be no updates at edge 7 and edge 8 as  $d(u)$  in that case is infinite. So, all edges have exhausted. This was one iteration of the outer loop( $i=1$ ).

We have to do the above iteration over each edge a total of 4 times as  $|v|-1 = 4$  in our example. We can do the iteration over edges in the same numbering order or use other order. This won't change the outcome of the algorithm.

In our example there are no negative cycle. So after 4 loops we will get a converged shortest path value for every node. I will suggest you to do an example at home with negative cycle. You will see that it is not converging in the final check of our algorithm.

We will discuss correctness in the next class.

Thank you.

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

**Lecture - 46**  
**Correctness of Bellman Ford**

So we are talking about bellman ford algorithm. We will discuss the correctness of the algorithm.

(Refer Slide Time: 00:27)

The input is a directed graph and a weight function and we have a source vertex. And the output will be  $\delta(s,v)$ .  $\delta(s,v)$  is the weight of the shortest path.

Let us recap the bellman ford algorithm.

(Refer Slide Time: 03:19)

The initialization part is same as Dijkstra algorithm.

1.  $d('s') = 0.$
2.  $d('v') = \infty \forall v \in V - \{'s'\}.$

We do the relaxation in the same manner as well. If  $d(u) + w(u,v) < d(v)$  then we relax that edge and make  $d(v) = d(u) + w(u,v)$ . We will do this step for every edge,  $|V|-1$  times in Bellman-Ford.

The last loop in the algorithm is to detect negative cycle. If there is no negative cycle, then the shortest paths must have converged. If there is anymore convergence, that would mean a negative cycle exists. Hence, we can check presence of negative edge via this algorithm (see the pseudo code in last lecture).

So, let us just talk about the correctness of this algorithm. Why the outer loop is  $|V|-1$  times? So, let us talk about correctness of Bellman-Ford.

(Refer Slide Time: 07:45)

Bellman-Ford tells us that if there is no negative cycle then we will get the shortest path in  $|V|-1$  iterations over the edges.

How to prove this? Let 'v' be a vertex in V. So,  $d(v) = \delta(s,v)$  which is the weight of shortest path from 's' to 'v'. This value can exist only if there is no negative cycle in the path and a physical path exists. Hence, we can say that each vertex can appear only once in the path. Because if a vertex appears more than once that would mean a cycle in the path, but we just now said that there can't be negative cycle in the shortest path. The cycle must be positive or 0. So, we can totally skip the cycle and reduce the shortest path.

So, let us consider a path p.

(Refer Slide Time: 11:57)

Let  $v_0$  be our source vertex and 'v' be our destination. Suppose, there are  $k$  vertices in the shortest path from  $v_0$  to  $v$ . We can write  $\delta(v_0, v_i) = \delta(v_0, v_{i-1}) + w(v_{i-1}, v_i)$  for any vertex  $v_i$  in the path because any suppath of the shortest path must also be the shortest path.

Initially,  $\delta(v_0, v_0) = 0$  and rest are infinity. In the first pass of our outer loop( $i=1$ ), we will see the edge  $E(v_0, v_1)$  at some point. When we see this edge then we will relax  $v_1$ . So,  $d(v_1) = w(v_0, v_1)$ .

(Refer Slide Time: 16:11)

After one pass, we must have relaxed  $v_1$  by seeing the edge  $E(v_0, v_1)$ . Now, let's see what happens in the second pass. It may have happened that  $v_2$  was not relaxed in the first pass as we could have seen  $E(v_1, v_2)$  before  $E(v_0, v_1)$ . But, in the second pass we will relax  $v_2$  on seeing  $E(v_1, v_2)$  as  $v_1$  has been relaxed once. By, 3rd iteration we would have relaxed  $v_3$ .

(Refer Slide Time: 22:18)

So, by  $k$ th round we will relax  $v_k$  atleast once,i.e we would have found a shortest path to  $v_k$  from ' $v_0$ '.

We have seen that if there is no negative cycle, then the shortest path is a simple path(no repetition of vertices). That means, there can be maximum of  $|v|-1$  vertices in between any two nodes. For, this reason the outer loop of Bellman-Ford runs  $|v|-1$  times. Now we will discuss some application of bellman ford algorithm, like how to solve the linear programming problem. So, next topic will be some application of bellman ford algorithm.

(Refer Slide Time: 25:04)

So, well we will talk about the how to solve the linear programming problem. Not the general linear programming problem, but a particular version of a linear programming problem which is called linear constraint.

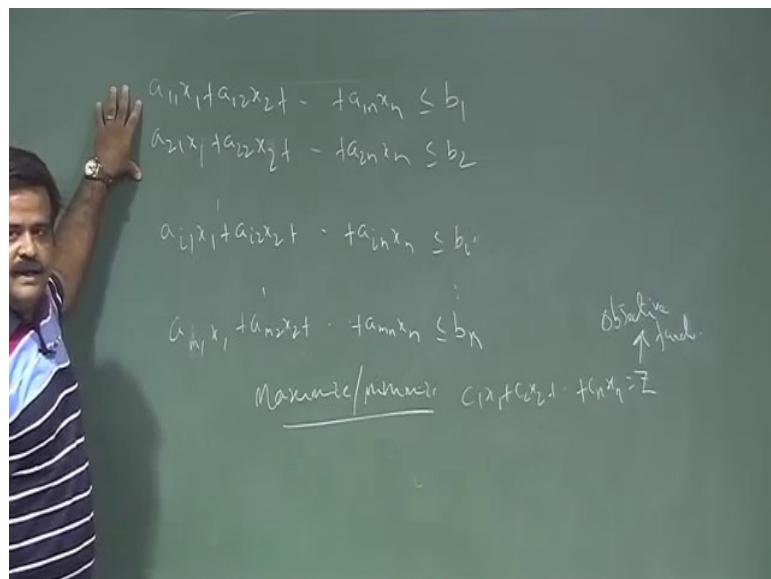
Thank you.

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

**Lecture - 47**  
**Application Of Bellman Ford**

We will talk about some Application of Bellman Ford algorithm. So Bellman Ford algorithm can be applied to the area like linear programming problem. We will talk about that and also we may discuss some of the other applications like how to detect a negative topological ordering of a graph. So, those we may discuss. So, let us start with the linear programming problem. So, what is a linear programming problem? In short it is called LPP.

(Refer Slide Time: 01:03)



So, what is a LPP problem? Basically we have some linear constant like say:

$$a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n \leq b_1$$

$$a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n \leq b_2$$

$$a_{n1}x_1 + a_{n2}x_2 + \dots + a_{nn}x_n \leq b_n$$

So, these are the variable. So, this is a given system of linear constant. Now our  $x_1, x_2, x_n$  must satisfy this and it must maximize or minimize some objective function like  $c_1x_1 + c_2x_2 + \dots + c_nx_n = Z$ . So, this function is called objective function. So, we are given a system of constants like this which is basically a system of inequalities. So, which we have to find  $x_i$  which satisfy this system of constant and then which will optimize this objective function.

So, this also can be written as the matrix form.

(Refer Slide Time: 03:54)

$$Ax \leq b, A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix}$$

$$Z = c^T x$$

$$c = \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{pmatrix}, x = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}, b = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}$$

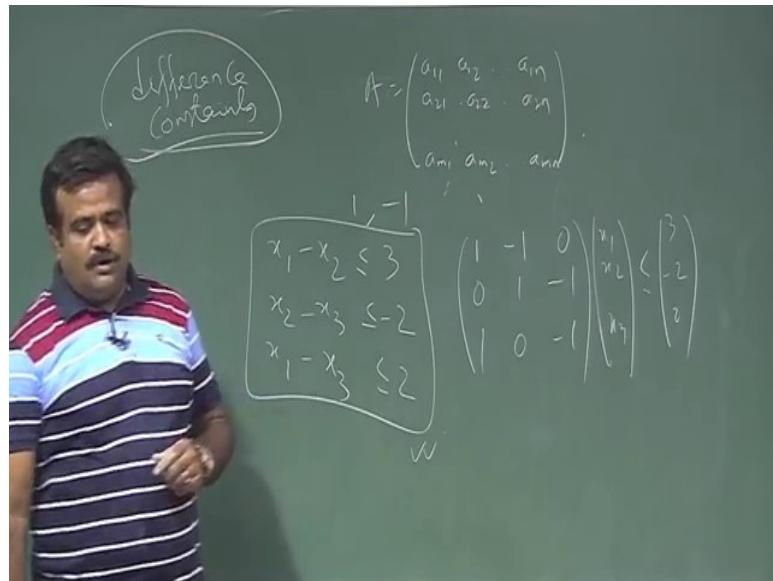
↑ objective function

Maximize/minimize  $c_1x_1 + c_2x_2 + \dots + c_nx_n = Z$

$Ax \leq b$  and  $Z = c^T x$ , where  $A$  is a  $n \times n$  matrix of all the variables in the system,  $c$  is a vector having  $c_1, c_2, \dots, c_n$  as its elements similarly we have  $x$  and  $b$  vectors.

So, there are some algorithms to solve this linear programming problem. One is simplex method, another one is some graphical method to solve the linear programming problem. Now, here we are restricting our LPP problem to a very particular case where each element of  $A$  matrix is either 1 or -1.

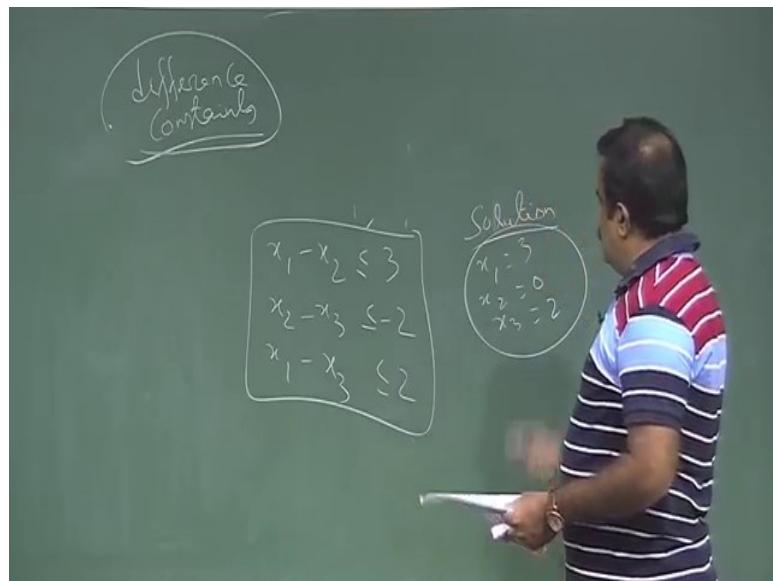
(Refer Slide Time: 05:45)



This is a very very special very particular case of LPP. So, these are called linear constant or difference constant. So, these are called difference constants. (Please see above example)

Now, we will just concentrate on this particular case or difference constant. We will just look at how to solve the difference constant. And then after getting the solution we have some optimization function maximizing or minimizing some objective function there.

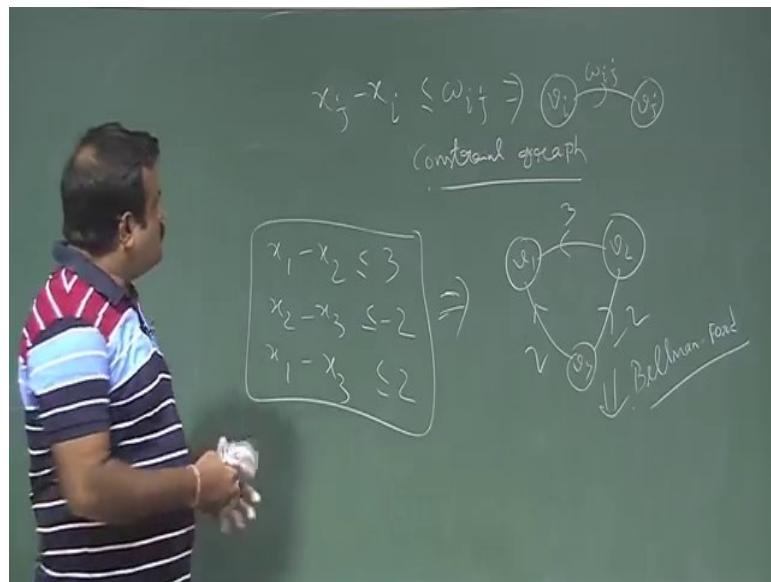
(Refer Slide Time: 07:50)



We can see the solution of previous example simply as  $x_1 = 3$ ,  $x_2 = 0$ ,  $x_3 = 2$ .

Now, we want to see how Bellman Ford algorithm can help us to get the solution of such difference constants and how we can make use of Bellman Ford to have the solution of this thing. Now for Bellman Ford we need a directed graph. How to construct a graph out of this?

(Refer Slide Time: 08:54)

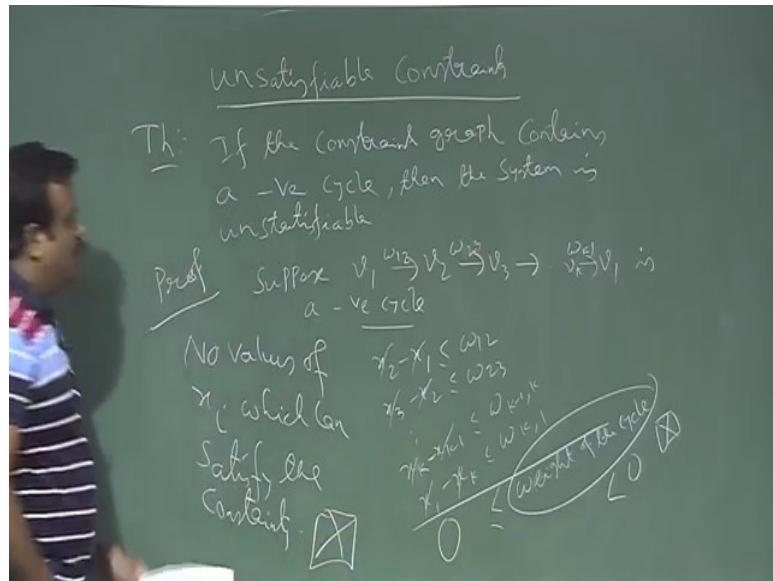


To construct a graph suppose we have a different equation  $x_j - x_i \leq w_{ij}$ . Let us create two vertices  $v_i$  and  $v_j$  corresponding to  $x_i$  and  $x_j$ .  $x_i$  is called graph constant. We have an edge with value  $w_{ij}$  going from  $v_i$  to  $v_j$ .

Once we have a directed graph with the edge weight. So, there is no restriction on the negative weight edge. Once we have the graph we can apply the Bellman Ford algorithm. Once we apply the Bellman Ford it will give us a negative cycle or it will give us the shortest path weight.

Now, suppose it is giving us a negative cycle; that means, there is a negative cycle in the corresponding graph then we will have to show that there will be no solution of the system of constant. That means, system of constant is infeasible.

(Refer Slide Time: 12:41)



So, this is called infeasible or unsatisfiable constraint. It is telling that if the constraint graph contains a negative cycle and that can be determined by Bellman Ford then the system is infeasible.

So, how to detect that whether there is a negative cycle? So, we convert this system to a constraint graph, then we run the Bellman Ford, once we run the Bellman Ford we can easily see that whether there is a negative cycle or not, because Bellman Ford will report the negative cycle if there is. If Bellman Ford is reporting negative cycle then there is no solution exist for this system. So, how to prove this?

Let the following constraints be there:

$$x_2 - x_1 \leq w_{12}$$

$$x_3 - x_2 \leq w_{23}$$

$$\vdots$$

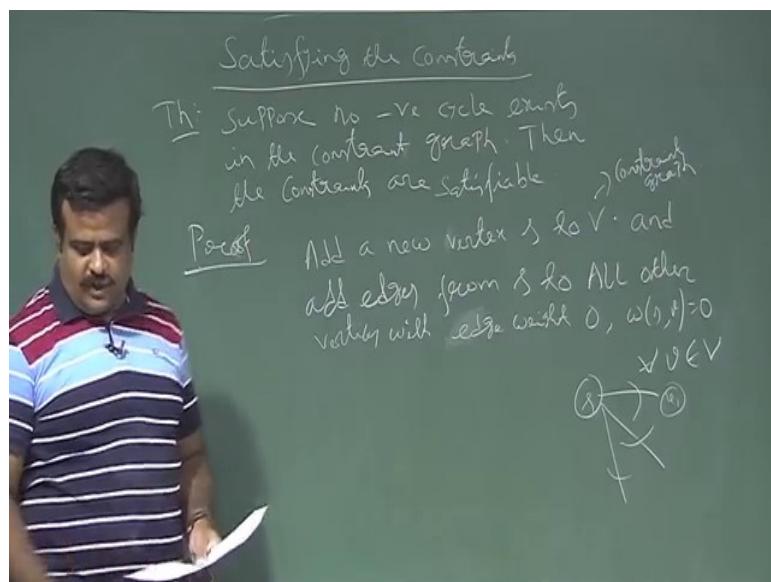
$$x_1 - x_k \leq w_{1k}$$

If we add all the variables on left side and right side. It evaluates to 0 on left side whereas right side is the sum of the weight of edges of the cycle in the graph.

Now this is a negative cycle. So, that weight is negative. Hence, this is the contradiction. That means, we are not having a solution for this system of constraint.

That means, the system of constraint is not feasible, it is unsatisfiable. Now we have to proof the other way around. Suppose there is no negative cycle. That means, Bellman Ford will give us the solution, Bellman Ford will give us the single source shortest path then from there how can we get the solution for this constant. So, that is the satisfying of the constant.

(Refer Slide Time: 19:12)



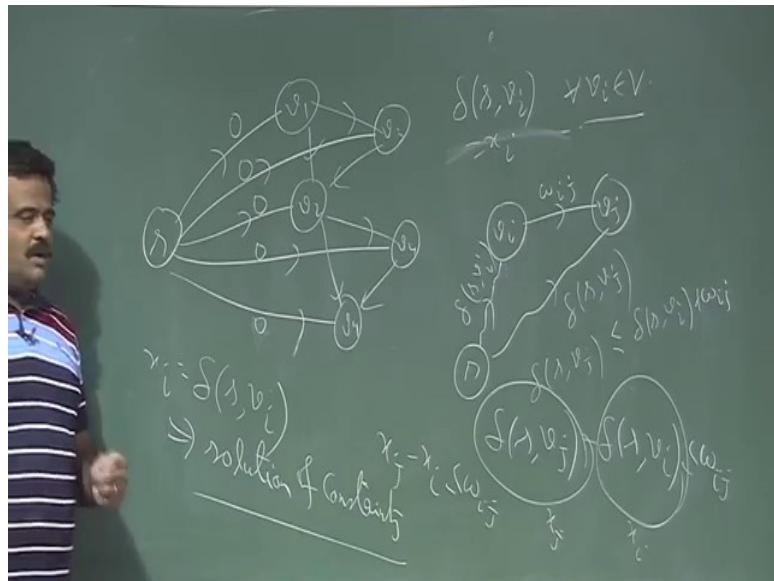
Satisfying the constraint. Suppose no negative cycle in the graph exists in the constraint graph then the system is feasible.

So, this is the case where we have a solution for this system. So, to prove this we just add some vertex in the graph. So, what we do? We just add a new vertex. So, what we have? We have a system of constraint from there we create constant graph and in that constant graph we are adding a new vertex. That will be a starting vertex because Bellman Ford needs a source.

Add a new vertex 's' and an edge from 's' to all other vertices with edge weight 0.

Basically we have the constraint graph (image below).

(Refer Slide Time: 22:55)



Now, if the original graph is having no negative cycle; that means this new graph will not have a negative cycle. No negative cycle means shortest path exists. That means, we will get  $\delta(s, v_i)$  for all  $v_i \in V$ .

So, now by triangular inequality we can say this  $\delta(s, v_j)$  is basically weight of the shortest path from  $s$  to  $v_j$ . So,  $\delta(s, v_j)$  must be less than equal to  $\delta(s, v_i) + w_{ij}$ . So,  $\delta(s, v_j) - \delta(s, v_i) \leq w_{ij}$ . So this is the system of constraint. This is the  $i$ -th constraint and this is the solution. So, if we just assign  $\delta(s, v_i)$  to be  $x_i$  and  $\delta(s, v_j)$  to be  $x_j$ .

So, what we are doing? We are having a constraint graph, we are just adding a vertex 's' and we are adding the edges with the 0 weight. Then we run the Bellman Ford and since there is a no negative cycle in the original graph. So, there will be no negative cycle in this new graph. After running the Bellman Ford we will get the deltas, because the shortest path will exist. And these deltas are nothing but the solution of this constraint that is coming from this triangular inequality. And so these deltas are basically the shortest path solution of the difference constraints.

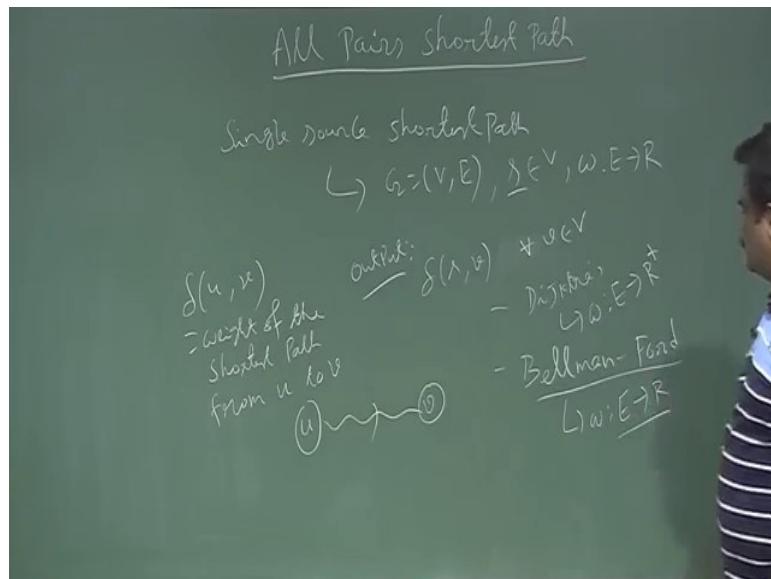
Thank you.

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

**Lecture - 48**  
**All Pairs of Shortest Path**

So, far we have seen the single source shortest path.

(Refer Slide Time: 00:29)



So, single source means we have a directed graph  $G = (V, E)$  and we have a source vertex from where we need to get the shortest path to other vertices.

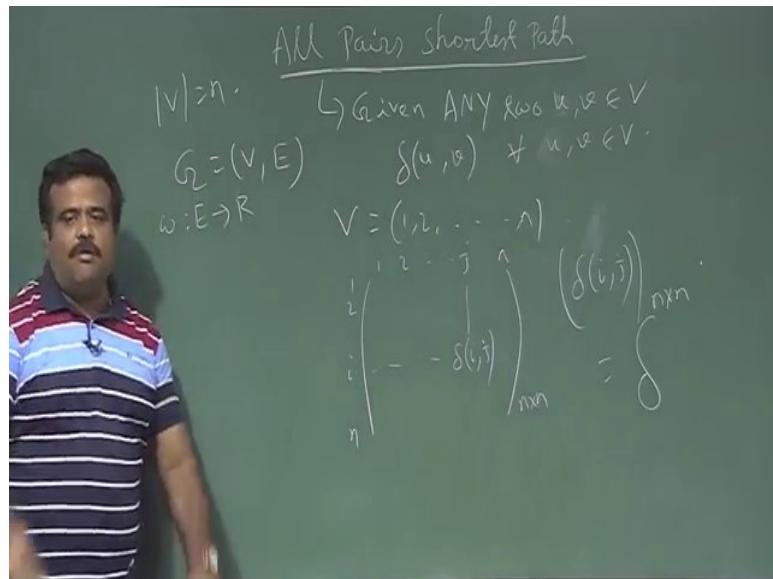
Basically we have a vertex 'v' and we have vertex 'u'. So, you consider all the paths from u to v. Shortest path will not exist if there is no physical path and also if there is a physical path it may not exist, that is in case of a negative cycle in the path.

We have seen two algorithms for this. One is Dijkstra's algorithm, but this algorithm has a limitation that it cannot handle negative cycle.

But we have generalized an algorithm which is Bellman Ford, we have no restriction on the edge weight because Bellman Ford can handle negative cycle. It can report there is a negative cycle in the graph.

So, these are the single source shortest path algorithms. Today we will discuss all pair shortest path problem. So that means, given any two vertex we should be able to find deltas.

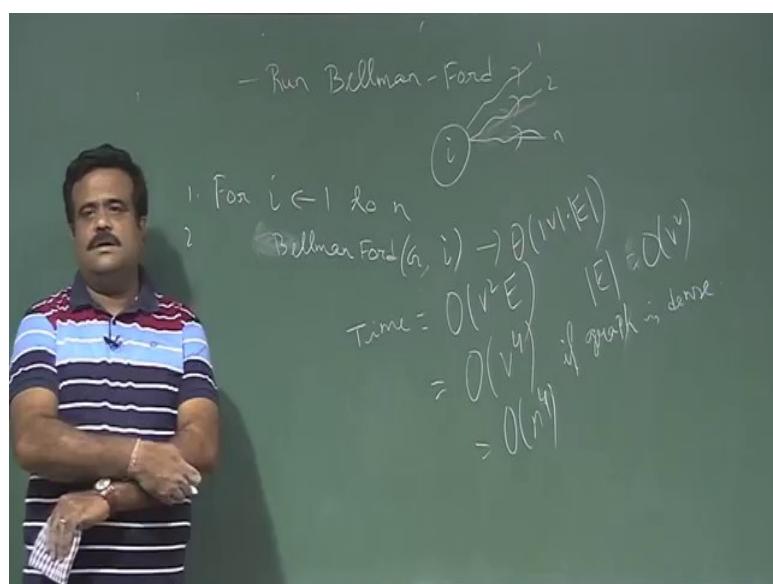
(Refer Slide Time: 04:22)



If we write this in the matrix form. Suppose there are  $n$  vertices. So, this is our graph  $(V, E)$ . We have a edge weight  $R$ .

So, basically what we need to find out? We need to find out a matrix of  $nxn$  dimensions. The  $(i, j)$ th element of the matrix is  $\delta(i, j)$ . This matrix we denote by  $\delta^{nxn}$ . Delta could be infinity if there is no shortest path from  $v_i$  to  $v_j$ .

(Refer Slide Time: 07:20)

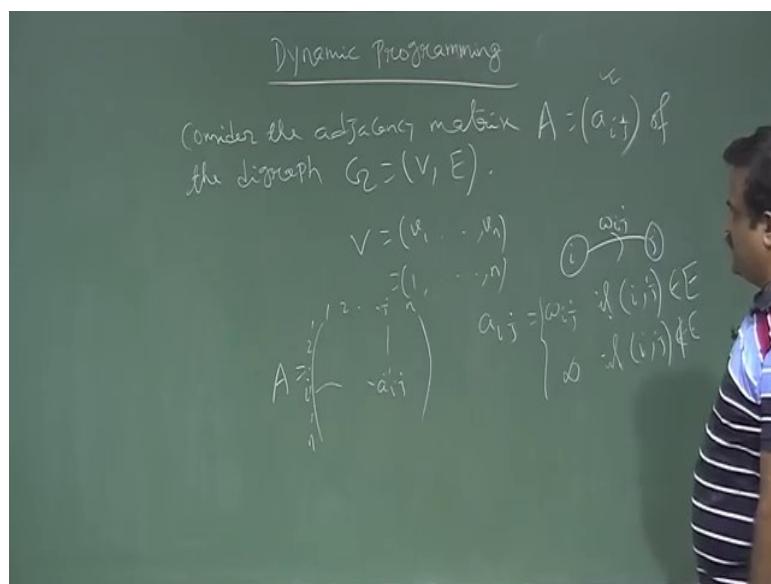


The first idea is we can make use of Bellman Ford for all pair of vertices. First we fix the  $i$  and then find the deltas to every other vertex. We run the Bellman Ford for all such  $i$ .

Now, what is the time complexity? Each time Bellman-Ford takes  $O(VE)$  time. Now this we are running for  $i = 1$  to  $n$ . So, the total time complexity for this is  $O(V^2E)$ .

If the graph is dense graph then order of  $E = O(V^2)$ . So, this is basically  $O(V^4)$  if the graph is dense. But we want to do something better than that. We want to do it in order of  $n^3$ . Now, we want to see whether we can use dynamic programming technique for this.

(Refer Slide Time: 10:16)

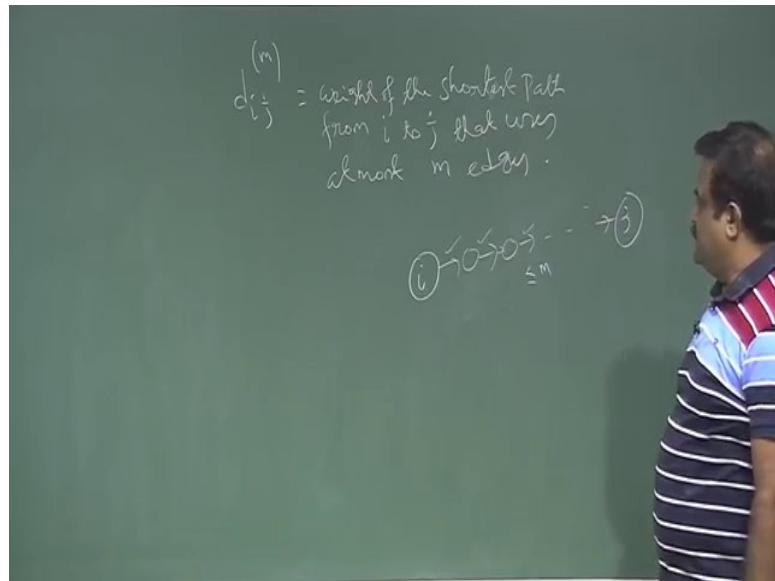


We have seen that shortest path has hallmark for Dynamic Programming. We have seen in the previous classes that it is having the optimal substructure problem and also the repetition of the sub problems.

So, let us just talk about the dynamic programming technique for this. So, consider the adjacency matrix  $A$ .

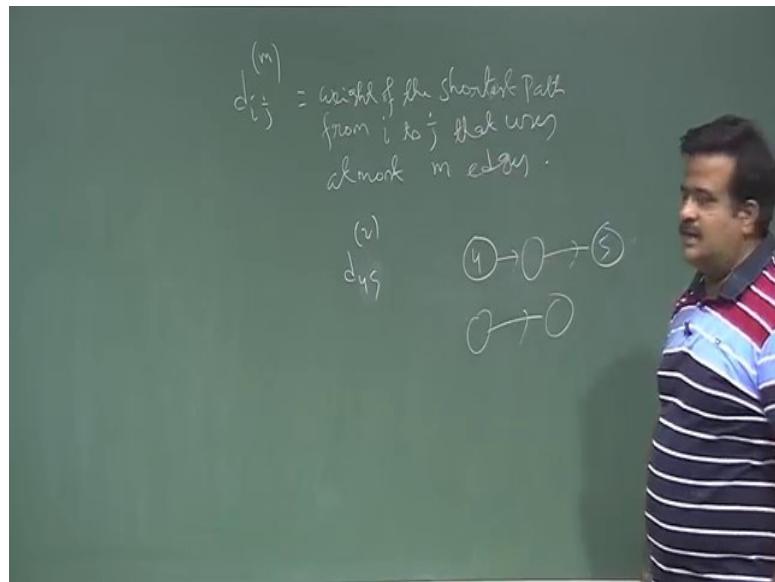
Now,  $a_{ij} = w_{ij}$  if an edge exists from  $v_i$  to  $v_j$ . Otherwise it must be infinity because we cannot put 0 because 0 could be the weight of an edge. So, this is the way we define the adjacency matrix on this directed graph.

(Refer Slide Time: 13:33)



Now, we define a  $\mathbf{d}$ . So,  $d_{ij}^{(m)}$  = weight of the shortest path from  $i$  to  $j$  that uses at most  $m$  number of edges. So, we consider all the path from  $i$  to  $j$  and we count the number of edges.

(Refer Slide Time: 15:26)



Suppose we are trying to find  $d_{45}^{(2)}$ , we consider all path from  $v_4$  to  $v_5$  in which we just take maximum two edges.

So, among these paths we take the shortest path with the least weight. In order to apply the dynamic programming we need to have some recurrence relationship.

(Refer Slide Time: 16:18)

$d_{ij}^{(m)}$  = weight of the shortest path from  $i$  to  $j$  that uses at most  $m$  edges.

claim:  $d_{ij}^{(0)} = \begin{cases} 0 & \text{if } i=j \\ \infty & \text{if } i \neq j \end{cases}$

and for  $m=1, 2, \dots, n-1$

$$d_{ij}^{(m)} = \min_k \{ d_{ik}^{(m-1)} + a_{kj} \}$$

The base case:  $d_{ij}^{(0)} = 0$  if  $i=j$

$\infty$  if not

This is due to the fact that there must be at most 0 edges between  $i$  and  $j$ . So, there is no possibilities of direct path.

General expression for  $m = 1, 2, \dots, n-1$

$$d_{ij}^{(m)} = \min_k \{ d_{ik}^{(m-1)} + a_{kj} \}$$

(Refer Slide Time: 18:32)

$$d_{ij}^{(m)} = \min_k \{ d_{ik}^{(m-1)} + a_{kj} \}, \text{ for } m=1, 2, \dots, n-1$$

Proof

$$d_{ij}^{(m)} = \min_k \{ d_{ik}^{(m-1)} + a_{kj} \} \quad \square$$

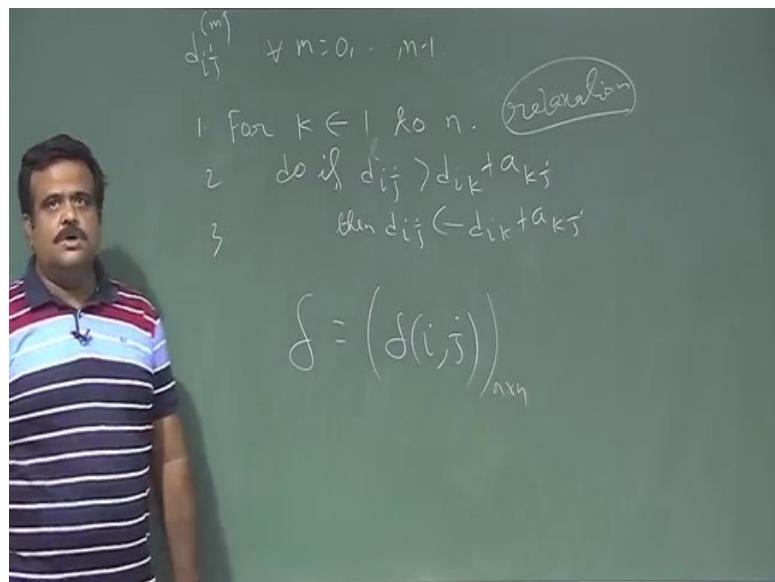
So, how to prove this?

So, to prove this, we have to draw the picture (refer above image). We are  $i$ 'th node and we want to go to the  $j$ 'th node. In the middle, we take all the incoming edges (direct edges) from any other node to the  $j$ 'th node.

Now, we have to go from ' $i$ ' to ' $j$ ' with total number of edges at most  $m$ . We already have one edge weight due to direct edges drawn earlier. Now, we need  $m-1$  edges. So, we have basically  $d_{ik}^{(m-1)}$ . So, the weight total weight is  $d_{ik}^{(m-1)} + a_{kj}$  and we choose the minimum among all such  $k$ 's. So, that is the proof.

This will give us an algorithm.

(Refer Slide Time: 22:16)



Algorithm is:

For k is equal 1 to n

do if  $d_{ij} > d_{ik} + a_{kj}$

then  $d_{ij} \leftarrow d_{ik} + a_{kj}$

What is the relationship between deltas and  $d_{ij}$ 's? If there is no negative cycle then how many edges will be there in the middle? There will be at most  $n-1$  edges in the middle.

(Refer Slide Time: 24:20)

Chalkboard notes:

- $d_{ij}^{(m)} \forall m = 0, \dots, M-1$
- 1. For  $k \in 1 \dots n$ . (relaxation)
- 2. do if  $d_{ij} > d_{ik} + a_{kj}$
- 3. then  $d_{ij} \leftarrow d_{ik} + a_{kj}$  (.)

Note: No negative cycle  $\Rightarrow \delta(i, j) = d_{ij}^{(n)}$

$$\delta \geq \begin{pmatrix} \delta(i, j) \end{pmatrix}_{n \times n} = D^{(n)} - D^{(n-1)}$$

No negative cycle implies  $\delta(i, j) = d_{ij}^{(n-1)}$ .

Define the D matrix:  $D^{(m)} = (d_{ij}^{(m)})_{n \times n}$ . Now, in the next class we will talk about how we can get this D matrix. So, we will see the matrix multiplication technique there. So, we will discuss this in the next class.

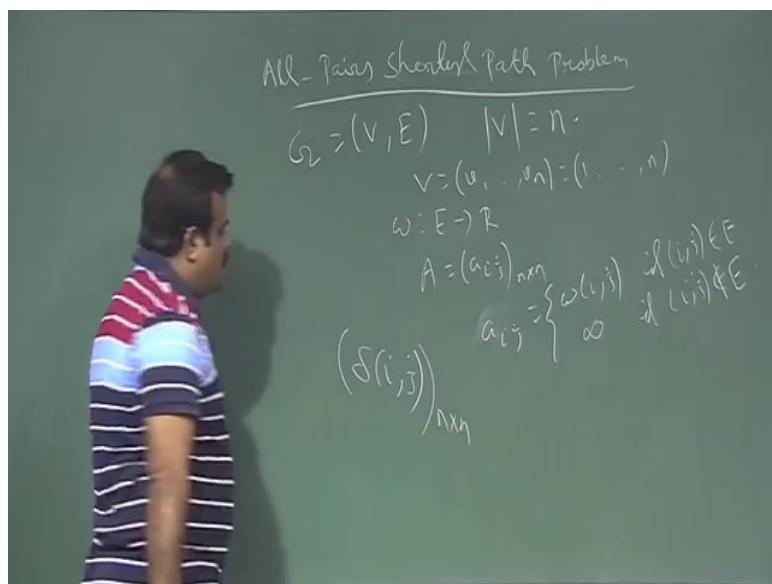
Thank you.

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

**Lecture – 49**  
**Floyd – Warshall**

So we are talking about all pair shortest path problem. We will discuss the Floyd-Warshall algorithm. So, let us just recap.

(Refer Slide Time: 00:30)



Given a graph  $(V, E)$  directed graph with  $|V| = n$ . We have adjacency matrix  $A$ . So,  $A$  is a  $n \times n$  matrix containing edge weights (refer above image).

We have to find out all pair shortest path =  $\delta(i, j)_{n \times n}$ . Delta is the weight of the shortest path from  $i$  to  $j$ . In the last class we defined some  $d_{ij}$ 's.

(Refer Slide Time: 01:53)



Dynamic Programming

$d_{ij}^{(m)}$  : weight of shortest path from  
i to j that uses at most  
m edges

$d_{ij}^{(0)} = \begin{cases} 0 & \text{if } i = j \\ \infty & \text{if } i \neq j \end{cases}$

$d_{ij}^{(m)} = \min_k \{ d_{ik}^{(m-1)} + a_{kj} \}$   
for m = 1, 2, ..., n

So, we have defined this  $d_{ij}^{(m)}$  is basically weight of the shortest path from i to j which uses at most m edges.

That means, if we go from i to j and we just count the number edges in this path, then that number must be less than equal to m. And we take the minimum weight. Base case  $d_{ij}^{(0)} = 0$  if  $i = j$ , otherwise it is infinity and we have seen  $d_{ij}^{(m)}$ .

Now, we will use this result to have the delta. So, what is the relationship between deltas and  $d_{ij}$ 's? That also we have discussed in the last class. So, if there is no negative cycle, then there will be shortest path. In that case  $\delta(i, j)_{nxn}$  is basically  $d_{ij}^{(n-1)}$ .

(Refer Slide Time: 04:21)

$$d_{i,j}^{(n-1)} \geq d_{i,j}^{(n)} = d_{i,j}^{(n+1)} = \dots$$

$$D^{(n)} = (d_{i,j}^{(n)})_{n \times n}$$

$$d_{i,j}^{(n)} = \begin{cases} 0 & \text{if } i=j \\ \infty & \text{if } i \neq j \end{cases}$$

$$d_{i,j}^{(m)} = \min_k \{ d_{i,k}^{(m-1)} + a_{kj} \} \quad \text{for } m=1,2,\dots$$

Because, if there is no negative cycle then the shortest path will be simple path so that means, we have at most  $n - 1$  edges in the path.

So, basically all pair shortest path problem is to find this delta matrix. So now, the question is how we can get this matrix, how you can we get  $D^{(n)}$  in general because it will converging there is no negative weight edges.

(Refer Slide Time: 06:02)

$$D^{(m)} = (d_{i,j}^{(m)})_{n \times n}$$

$$d_{i,j}^{(m)} = \min_k \{ d_{i,k}^{(m-1)} + a_{kj} \}$$

$$D^{(m)} = D^{(m-1)} \times A$$

↳ matrix multiplication

$$A = (a_{ij})_{n \times n}, \quad B = (b_{ij})_{n \times n}, \quad C = A \times B = (c_{ij})_{n \times n}$$

$$c_{ij} = \sum_{k=1}^n a_{ik}b_{kj} = \text{Sum}(a_{ik}b_{kj})$$

Now we will see how to get this  $D$  matrix. For that we need to take help of what is called matrix multiplication. So, we want to claim that  $D^{(m)}$  is basically  $D^{(m-1)}$  into a matrix. This is

sort of some sense matrix multiplication. Let us just go back to what do we mean by matrix multiplication of 2 matrices? Just concentrate on matrix multiplication. So, suppose we have 2 real matrix: A, B.

For the simplicity we have 2 matrices A and B both are of same size  $n \times n$ . Now then what is the multiplication of these 2 matrix? C is  $A \times B = (c_{ij})_{n \times n}$ . And  $c_{ij} = \sum a_{ik} b_{kj}$  and this k is from 1 to n.

Instead of sum operation in matrix multiplication if we think this is the minimum operation. And then addition in the earlier algorithm, if we replace by real multiplication. Then this is nothing but a matrix multiplication.

In general if the numbers are coming from a real field, and we have a operation on the general field. One operation is multiplication operation and another operation is summation. So, similarly if we have another operation say star and we can define this matrix multiplication over this field suppose we have a field with 2 symbols. Provided this satisfies some of the properties. So,  $D^{(m)} = D^{(m-1)} \times A$ .

(Refer Slide Time: 12:30)

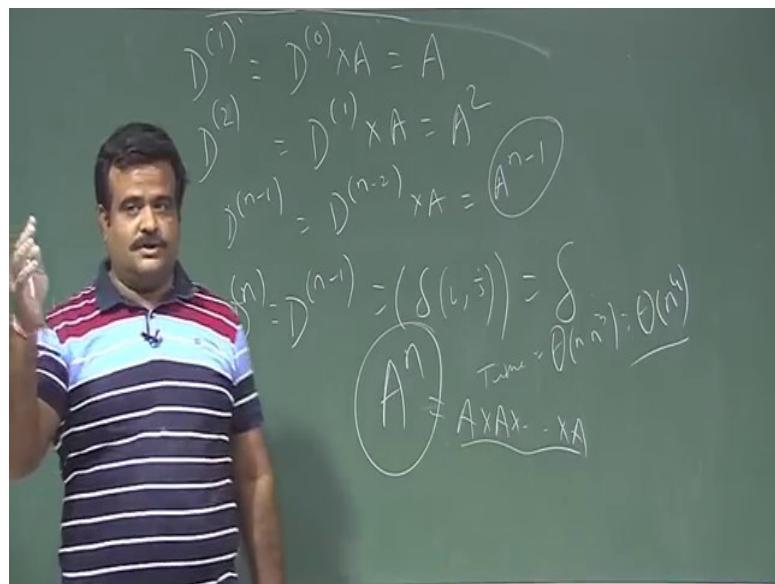
The image shows a teacher in a striped shirt writing on a chalkboard. The chalkboard contains the following text and equations:

- $D^{(m)} = \left( d_{ij}^{(m)} \right)_{mn}$
- $d_{ij}^{(m)} = \min_{k=1}^n \{ d_{ik}^{(m-1)} + a_{kj} \}$
- $D^{(m)} = D^{(m-1)} \times A$  (with a note "matrix multiplication")
- $D^{(0)} = \begin{pmatrix} 0 & \infty \\ 0 & 0 \end{pmatrix} = I$  (with a note "n = 1, 2, ..., N")

This m is varying from 1 to n.

In  $D^{(0)}$ , the diagonal elements are all 0 and off diagonal all are infinity. So, this serves as an identity matrix, this serves as an identity matrix. Our operation is minimum. Our operation is minimum and the plus.

(Refer Slide Time: 13:38)



So,

$$D^{(1)} = D^{(0)} \times A = A$$

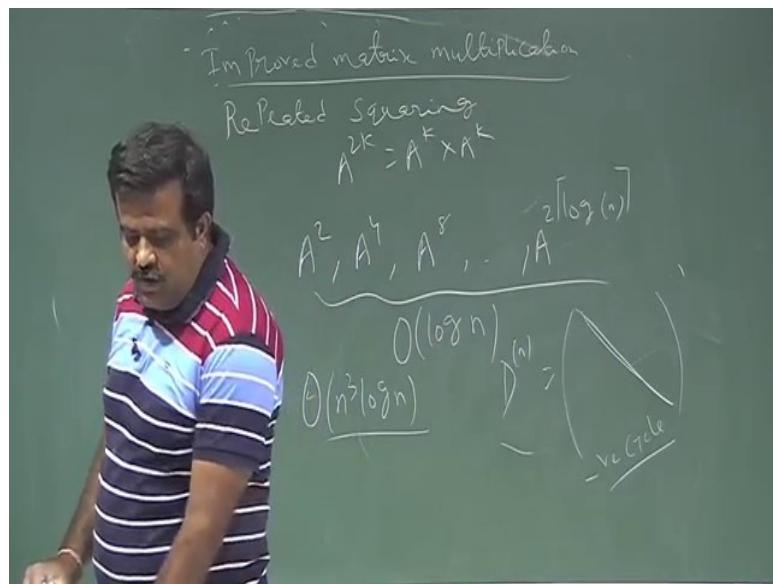
$$D^{(2)} = D^{(1)} \times A = A^2$$

$$D^{(n-1)} = D^{(n-2)} \times A = A^{n-1}$$

A is the adjacency matrix. If we can find  $D^{(n-1)}$  that will give us  $\delta(i, j)$  or  $\delta$  matrix. So, basically the problem is to find  $A^n$ . So now, the question is how to find this.

So, this is basically matrix multiplication. So, order of  $n^3$ . So, total time will be  $(n.n^3)$ . So, how to simplify this? We can simplify this matrix multiplication little bit in this naive approach. Each time we are doing a matrix multiplication with a cost order of  $n^3$ . Provided we can handle that 2 operation minimum or plus. Because we are not in real matrix. So, we have to be little careful about the field. So, our field is just minimum operation and the plus operation. So now, how to improve this matrix multiplication? So, we can improve by just repeated squaring.

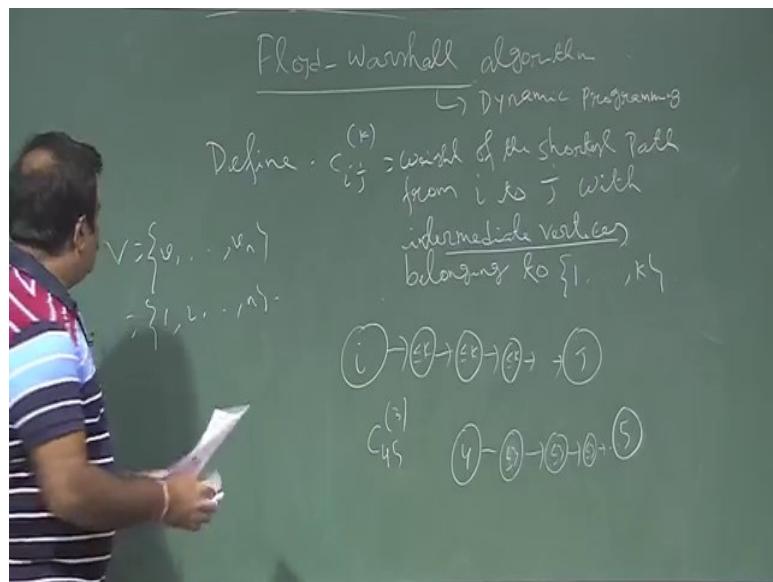
(Refer Slide Time: 16:21)



$A^{2k} = A^k \times A^k$ . We have to find  $A^n$ . To find that, we can find  $A^2, A^4$  then  $A^8$  like that to  $A^{2^{\lceil \log(n) \rceil}}$ . So, this is the way. So, this will give us  $\log(n)$  such squaring. It will take order of  $n^3 \log(n)$  which is not benefit. So, if we have to detect the negative cycle then it will not converge. And we have to take we have to check the diagonal element. And so, we find  $D^{(n)}$  say, and we check the diagonal element. And if the diagonal element has negative value then there is a negative cycle. So, this is how we detect the negative cycle.

So now we will discuss a better algorithm which is called Floyd-Warshall algorithm which will give us the order of  $n^3$  time for finding these deltas. So, for that we define new variable  $c_{ij}^k$ .

(Refer Slide Time: 18:36)



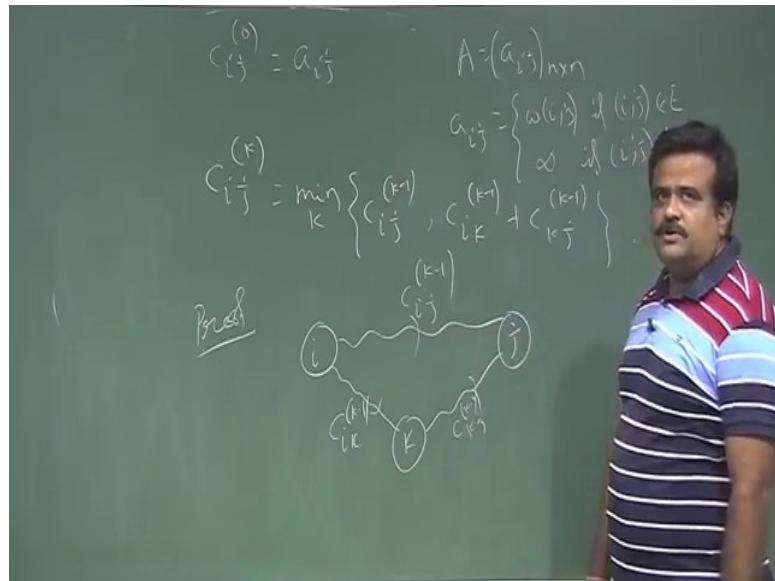
This is also a dynamic programming algorithm, but which is faster than the one we have seen.

Instead of  $d_{ij}$  here we are defining  $c_{ij}^{(k)}$  = weight of the shortest path from  $i$  to  $j$  with intermediate vertices belonging to  $\{1, 2, \dots, k\}$ . So, what is the meaning of this? Suppose we are at ' $i$ ' and we want to go to ' $j$ ' and in the middle we visit some vertices. These vertices must be less than equal to  $k$ . And we have no restriction on ' $i$ ' and ' $j$ ', they could be more than  $k$ , but we have a restriction on the intermediate vertices.

For example, if we just find out  $c_{45}^{(3)}$ . We have our  $v_4$  and  $v_5$ . Now we consider all the path from  $v_4$  to  $v_5$  such that all the intermediate vertices are either  $v_1$  or  $v_2$  or  $v_3$ . And we take minimum among that.

Now, let us have the recurrence relation.

(Refer Slide Time: 22:03)



What is  $c_{ij}^{(0)}$ ? Because there is no vertex in the middle. That means, there is direct edge. So, this is basically  $a_{ij}$ .  $A$  is the adjacency matrix  $(a_{ij})_{n \times n}$  where  $a_{ij}$  is  $w_{ij}$  if  $(i, j)$  is an edge otherwise it is infinity.

So,  $c_{ij}^{(k)}$  we have to write in terms of  $k-1$ . Our claim is,

$$c_{ij}^{(k)} = \min_k \{c_{ij}^{(k-1)}, c_{ik}^{(k-1)} + c_{kj}^{(k-1)}\}$$

Now, it may happen and we do not want to see a vertex whose number is more than  $k$ . So, that is one possibility of  $c_{ij}^{(k-1)}$ . And the other possibility is we must see the  $k$ th vertex in the middle. So, if you see the  $k$ th vertex then we go to the  $k$ th vertex then back to  $j$ . So, that is basically  $c_{ik}^{(k-1)} + c_{kj}^{(k-1)}$ .

These are the two possibilities we take minimum among these 2. Now, how this  $c_{ij}^{(k)}$  will help us to have a delta. So, basically we need to find out the delta. What is the relationship between delta and  $c_{ij}$ ?

(Refer Slide Time: 25:05)

Handwritten notes on the chalkboard:

$$c_{ij}^{(0)} = a_{ij}$$

$$A = (a_{ij})_{n \times n}$$

$$c_{ij}^{(k)} = \min_{k \in \{0, 1, \dots, n\}} \left\{ c_{ij}^{(k-1)}, c_{ik}^{(k-1)} + c_{kj}^{(k-1)} \right\}$$

$$c_{ij} = \begin{cases} a_{ij} & \text{if } (i, j) \in E \\ \infty & \text{if } (i, j) \notin E \end{cases}$$

Algorithm

```

1.  $C^{(0)} \leftarrow A$ 
2. For  $k \leftarrow 1$  to  $n$ 
3.   do for  $i \leftarrow 1$  to  $n$ 
4.     do for  $j \leftarrow 1$  to  $n$ 
5.       do if  $c_{ij} > c_{ik} + c_{kj}$  then  $c_{ij} \leftarrow c_{ik} + c_{kj}$ 
6.   then  $c_{ij} \leftarrow c_{ik} + c_{kj}$ 

```

Time =  $\Theta(n^3)$

So, basically  $\delta(i, j)$  is shortest path from  $i$  to  $j$  which is  $c_{ij}^{(n)}$ , why? Because if there is no negative cycle. So, every path will be simple path.

So, we are going from  $i$  to  $j$ . So, the path we visit here is the maximum up to  $n$ . So, that is basically  $\delta(i, j)$ . We are getting this recursive algorithm. So, what is the algorithm. So, basically we need to find  $c_{ij}$ . So, let us have the algo Floyd-Warshall.

We just initialize the  $C$  matrix as  $A$  this is the  $c_{ij}^{(0)}$ .

For  $k = 1$  to  $n$

do for  $i = 1$  to  $n$

do for  $j = 1$  to  $n$ .

do if  $c_{ij} > c_{ik} + c_{kj}$  then you must relax this.

then  $c_{ij} = c_{ik} + c_{kj}$ .

So, this is basically we are calculating this  $c_{ij}$  recursively. So, this is initialized by a matrix  $C$  matrix. So, this is the code, if this is greater than we are just relaxing, otherwise this will be the earlier one. So, what is the time complexity for this? How many loops. So, time is basically order  $n^3$  which is better than the earlier one. So, this is what is called Floyd-

Warshall algorithm. So, in the next class we will see some application of Floyd-Warshall algorithm.

Thank you.

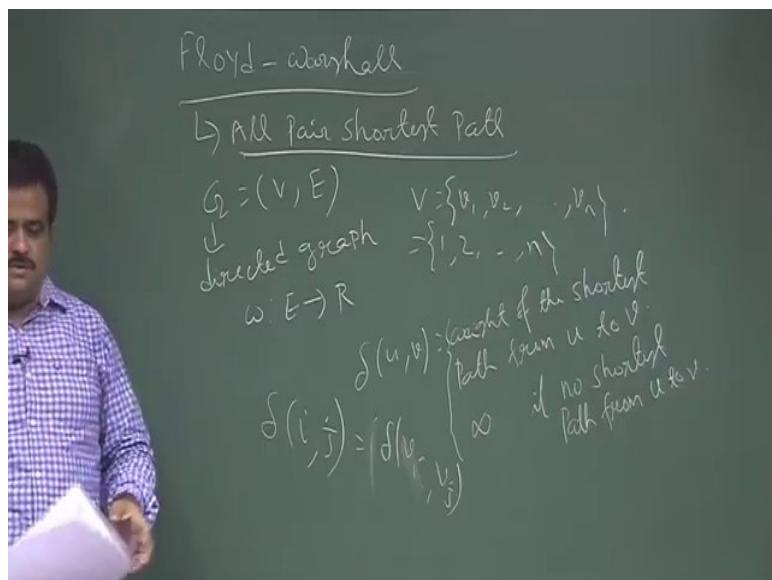
**An Introduction to Algorithms**  
**Prof. Sourav Mukhopadhyay**  
**Department of Mathematics**  
**Indian Institute of Technology, Kharagpur**

**Lecture – 50**  
**Johnson Algorithm**

So we are talking about all pair shortest path. We have seen two dynamic programming based approach algorithms. So, today we will talk about some application of the Floyd Warshall as algorithm, that is to find the transitive closure of a graph and then we will move to another algorithm for finding the all pair shortest path called Johnson Algorithm, but this will be using an old algorithm like Dijkstra's Algorithm.

So, let us first talk about the applications of Floyd Warshall. So, just to recap.

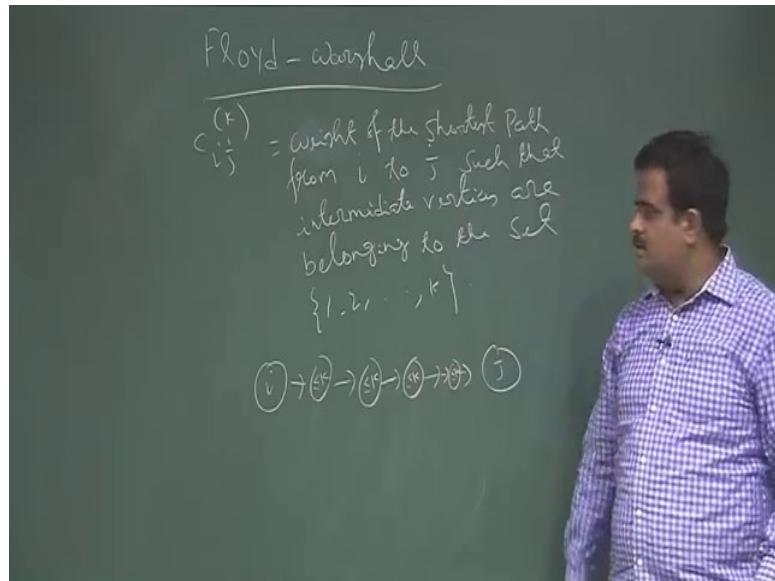
(Refer Slide Time: 01:02)



This is basically to find the all pair shortest path. Now, we have given a graph  $G = (V, E)$ . So, this is a directed graph or digraph and we have a edge weight  $E \rightarrow R$ . We denote  $\delta(u, v)$  is the weight of the shortest path from  $u$  to  $v$ .

Our aim is to find these deltas. For Floyd Warshall what we did to find out this delta. This we have discussed in the last class.

(Refer Slide Time: 03:57)

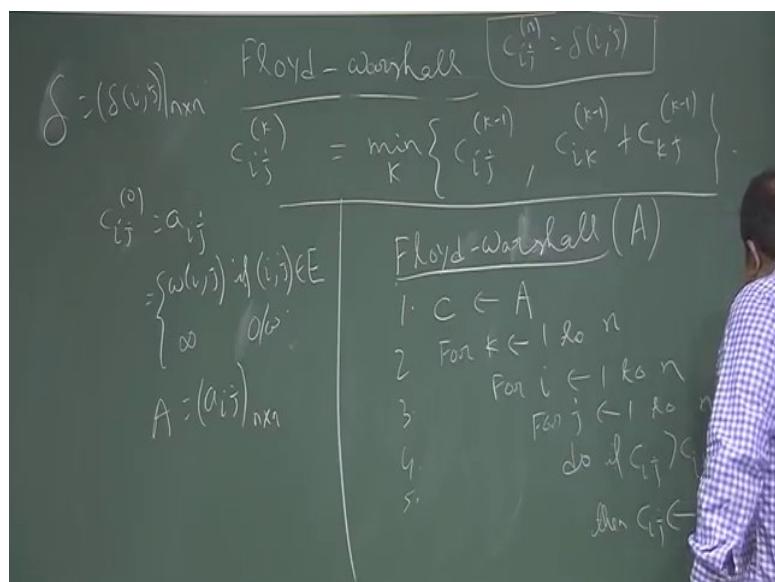


The recursive relation  $c_{ij}^{(k)}$  is the weight of the shortest path from  $i$  to  $j$  such that intermediate vertices are belonging to the set  $\{1, 2, \dots, k\}$ . So that means, we are at  $i$ 'th node, we want to go to  $j$ 'th node.

In the middle the vertices we are visiting they must be label from 1 to  $k$ . We should not see any vertex whose level is more than  $k$ .

What is the recurrence for  $c_{ij}$ ? So,  $c_{ij}^0$  is basically  $a_{ij}$ . Adjacency matrix means that  $a_{ij}$  is  $w_{ij}$  if  $(i, j)$  is on edge otherwise we have infinity.

(Refer Slide Time: 06:33)



What is the  $c_{ij}^{(k)}$ ?

In the last class we have proved that this is basically  $\min_k \{c_{ij}^{(k-1)}, c_{ik}^{(k-1)} + c_{kj}^{(k-1)}\}$ . So, minimum among these. So, from this we have the algorithm based on divide and conquer approach. So, that is Floyd Warshall Algorithm. So, this is basically taking a graph with the matrix A. A is the adjacency matrix. So, this A basically consist of  $(a_{ij})_{n \times n}$ .

What is the advantage of getting C? We are looking for delta. What is the relationship between delta and C? If we can find out all  $c_{ij}^{(n)}$ . So,  $c_{ij}^{(n)}$  is basically giving us  $\delta(i, j)$ .

So, we want to find the C.

We just initialize the C matrix as A this is the  $c_{ij}^{(0)}$ .

For  $k = 1$  to  $n$

do for  $i = 1$  to  $n$

do for  $j = 1$  to  $n$ .

do if  $c_{ij} > c_{ik} + c_{kj}$  then you must relax this.

then  $c_{ij} = c_{ik} + c_{kj}$ .

(Refer Slide Time: 11:37)

Floyd-Warshall  $\left[ c_{ij}^{(n)} = \delta(i, j) \right]$

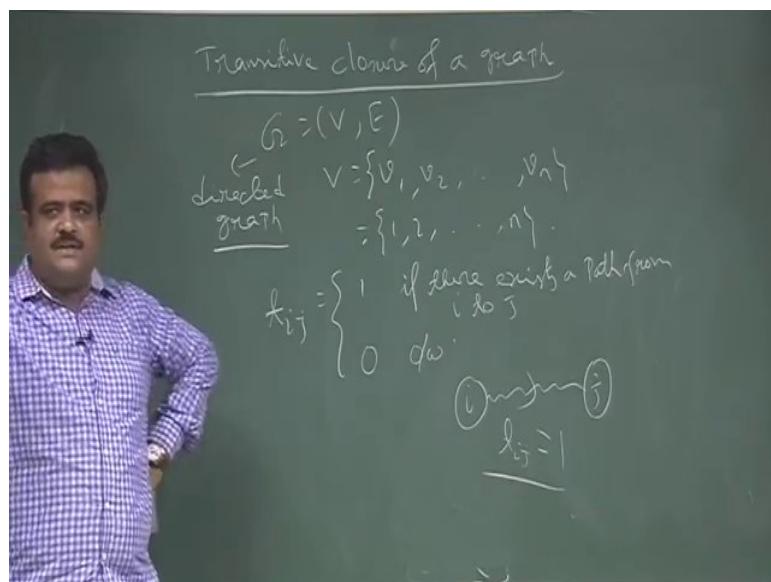
$$c_{ij}^{(k)} = \min_k \left\{ c_{ij}^{(k-1)}, c_{ik}^{(k-1)} + c_{kj}^{(k-1)} \right\}.$$

		Floyd-Warshall (A)
		1. $C \leftarrow A$
		2. For $k \leftarrow 1$ to $n$
		3. For $i \leftarrow 1$ to $n$
		4. For $j \leftarrow 1$ to $n$
		5. do if $c_{ij} > c_{ik} + c_{kj}$
		6. then $c_{ij} \leftarrow c_{ik} + c_{kj}$
		$\delta \in C$
$c_{ij}$	$i, j \in V$	
$A$	$A \in \mathbb{R}^{n \times n}$	

So, delta is basically the C matrix. This is what is called the Floyd Warshall Algorithm and the time complexity for this is basically order of  $n^3$ . So, now, we will talk about an application of this algorithm which is called transitive closure of a graph.

So, suppose you have a graph it is a directed graph and there is no weight on the edges.

(Refer Slide Time: 12:34)

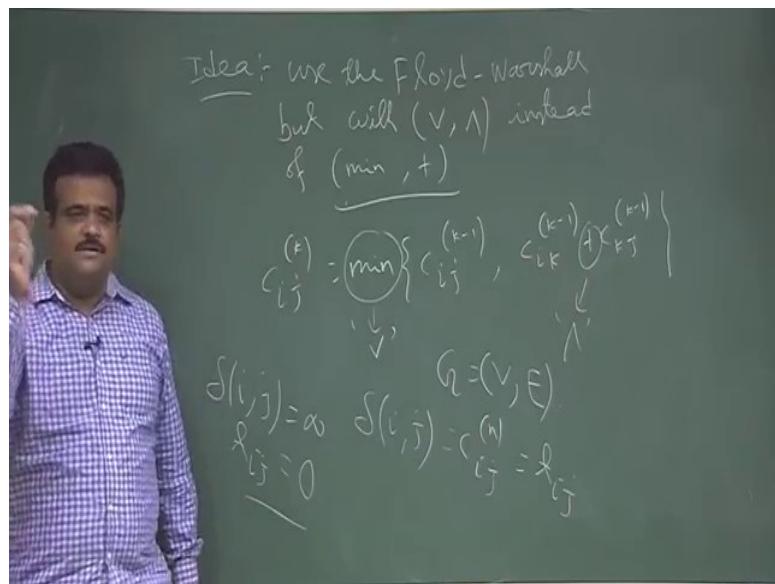


This is called transitive closure of a graph. Suppose we have a graph G. There were say n vertices and for simplicity we are writing vertex label as 1 to n. Now so we define  $t_{ij}$  to be 1 if there is path for i'th vertex j'th vertex, otherwise it is zero. If we have a direct edge; that means, there is a path. But we are not looking for only directives we are looking if at all we can reach to the j'th vertex.

So, that is the transitive closer. So, if there is a path from i to j then  $t_{ij}$  is 1. We have given a graph, this graph is a directed graph or digraph.

How we can get this  $t_{ij}$ ? So, we want to basically use the Floyd Warshall Algorithm to have this  $t_{ij}$ . So, what is the idea? So, idea is to use instead of minimum and plus we will just use the or ( $\vee$ ) and and ( $\wedge$ ) operation.

(Refer Slide Time: 15:30)



Idea is to use the Floyd Warshall Algorithm but with the operation logical or and instead of minimum and plus.

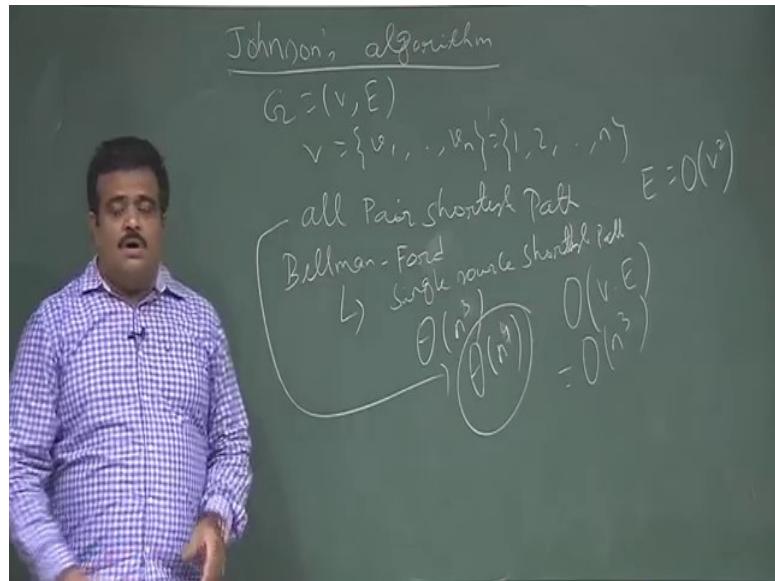
So, instead of this minimum we will use the ‘or’ operation and instead of plus, we will use the ‘and’ operation. We are not concerned about the weight of the graph. We have given a graph G which is (V, E).

Instead of find  $\delta(i, j)$ , what we are looking for? We are trying to find  $t_{ij}$ . So, basically if we run Floyd Warshall, the algorithm we have discussed with the logical ‘or’ and logical ‘and’ then this will give us  $t_{ij}$ . Why?

Because if we just add a weight 1 to each of the edges then if there is a path means there is a shortest path. So, we are not bothered about shortest path here we just bother about whether there is a physical path from i to j. So, if we run this for vertex number up to n, then eventually this will give us  $t_{ij}$ . If the delta is infinity; that means there is no shortest path because we are assigning 1 to each of the edges. So, if delta is basically infinity then the corresponding  $t_{ij}$  is basically 0; that means there is no path from i to j. So, that is the idea. So, this is the application of Floyd Warshall on the transitive closure.

Another algorithm which is called Johnson Algorithm to finding the all pair shortest path.

(Refer Slide Time: 19:00)



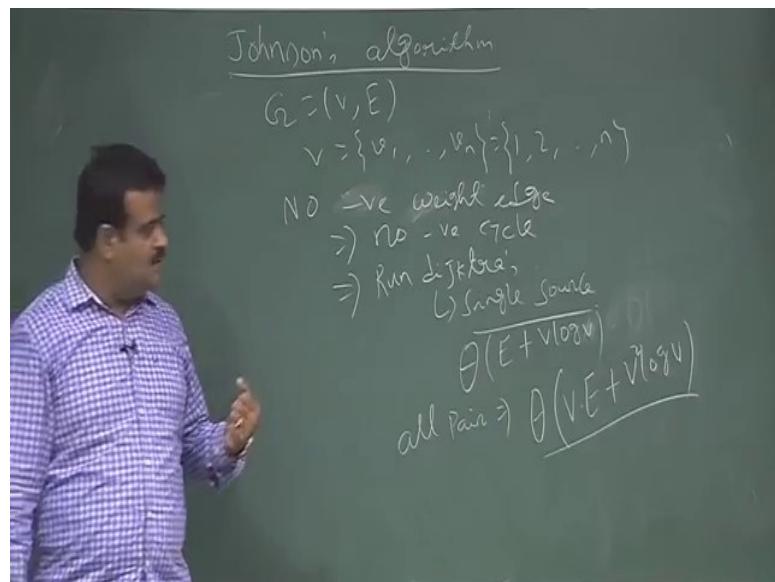
So, this is to find all pair shortest path. So, far we know some dynamic programming technique to find the shortest path. We have a graph  $G = (V, E)$  and vertex as numbering from 1 to  $n$ .

We already know like single source shortest path algorithm. We know two single source shortest path algorithm; one is Bellman Ford, another one is Dijkstra's. But in Bellman Ford we have seen that it will take order of  $n^2$  for a single source and we have to run for all the sources. So, this will give us for all pair in order of  $n^3$ .

So, this is not good as we already have order  $n^3$  algorithm. Bellman Ford is order of  $(VE)$ . So,  $V$  is  $n$  and  $E$  is basically  $O(V^2)$ , so this is  $n^3$ . So, total is  $n^4$ , this is not good. For Dijkstra's there is a restriction; Bellman Ford can handle the negative weight edges because it can handle the negative cycle, but dijkstra's cannot handle negative weight cycle.

So, somehow if we know that our graph is having no negative weightage then we one can try for Dijkstra's.

(Refer Slide Time: 21:51)

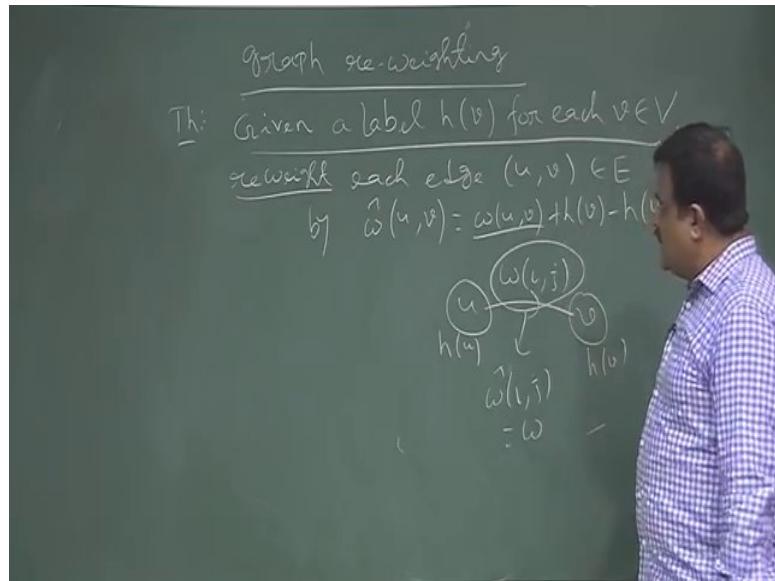


Now if we run Dijkstra's for all pair shortest path. So, Dijkstra's will take how much time? The Dijkstra's is also single source shortest path. So, now, one run of Dijkstra's will depend on which data structure we are using. So, if we use the fibonacci trees then the worst case amortize analysis is  $O(E + V \log(V))$ .

Now, if we run it for all pair shortest path then it will be for all pair, it will be  $O(EV + V^2 \log(V))$ . Now this is as good as the Floyd Warshall because this is at most  $n^3$ , but sometimes if  $E$  is not order  $V^2$  then it is less. So, depending on  $E$  it will be either order of  $n^3$  or order  $n^2 \log(n)$ .

So, this is good, now the problem here is we have this assumption of no negative weight edge.

(Refer Slide Time: 23:49)

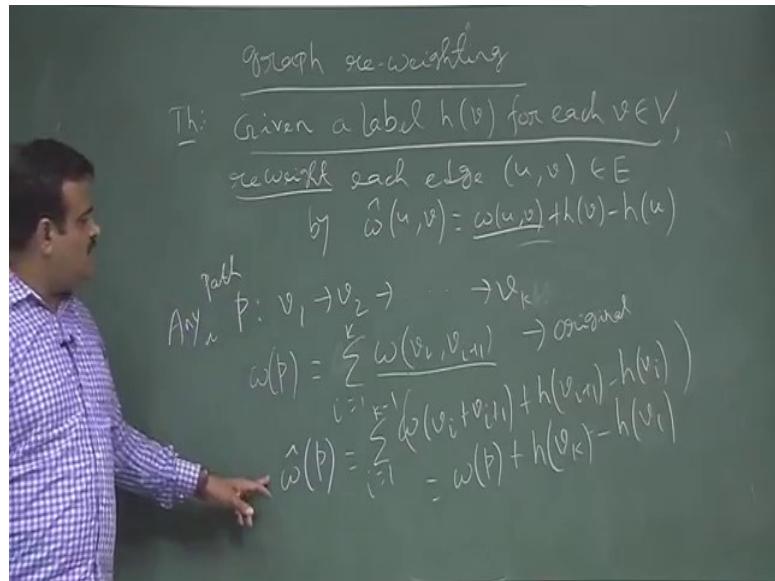


So, how can we do that?

To ensure that we have to do something called graph re-weighting. So, this is just to ensure there will be no negative weight edge. So, we are going to label the vertices. Given a label  $h(v)$  for each vertex  $v$ , we reweight each edge  $(u, v)$  belonging to  $E$  by  $\hat{w}(u, v) = w(u, v) + h(v) - h(u)$ .

So, basically what we are doing? We have two vertices. Now we are labeling the vertices by  $h$  which is a function we have to get this function. We will come to know how will get this function? Now the edge weight, we are rewriting as  $\hat{w}(i, j)$ . So we reweight all the edges like this.

(Refer Slide Time: 26:10)



If we do that then suppose we have a path  $p$  from  $v_1$  to  $v_k$ .

Suppose you have a path. Now what is the weight of this path? Weight of this path is

basically  $\sum_{i=1}^{k-1} w(v_i, v_{i+1}) = w(p)$ . Now what is the new weight of the path after the relabel of

the edges? This is basically  $\sum_{i=1}^{k-1} (\hat{w}(v_i, v_{i+1}) + h(v_{i+1}) - h(v_i)) = w(p) + h(v_k) - h(v_1)$

Now we want there should not be any negative cycle. So, we want weight of each path should be positive.

(Refer Slide Time: 28:50)

$$\hat{w}(u,v) \geq 0 \cdot \forall (u,v) \in E$$

$$\Rightarrow w(u,v) + h(v) - h(u) \geq 0$$

$$\Rightarrow h(u) - h(v) \leq w(u,v)$$

$x_1 - x_2 \leq w_{12}$

using  
Bellman  
Ford  
 $O(n^3)$

Any path  $p: v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_k$

$$w(p) = \sum_{i=1}^k w(v_i, v_{i+1}) \rightarrow \text{original}$$

$$\hat{w}(p) = \sum_{i=1}^k w(v_i + v_{i+1}) + h(v_{i+1}) - h(v_i)$$

So, we want to choose the labeling in such a way that for all the edges must be greater than 0 for all  $(u, v)$  belongs to  $E$ .

So, for that what we are doing? What this is  $w(u, v) + h(v) - h(u)$ ? So, this we want greater than 0. That means,  $h(u) - h(v) \leq w(u, v)$ . So, this we want for all of the vertices. We want to have a  $h$  function. We can get this  $h$  from solving the difference constant.

We have seen  $x_1 - x_2 \leq w_{12}$  before. So, for solving this again we can apply the Bellman Ford algorithm. Using the Bellman Ford algorithm we can get this  $h$ . This will take into  $O(n^3)$ . And then once we relabel it then we know that there will be no negative weight edge in the graph then we can run Dijkstra's. So, after relabeling once, we get the solution of this.

(Refer Slide Time: 30:55)

1.  $\hat{w}(u, v) \geq 0 \quad \forall (u, v) \in E$   
 $\Rightarrow w(u, v) + h(v) - h(u) \geq 0$   
 $\Rightarrow h(u) - h(v) \leq w(u, v)$

$x_1 - x_2 \leq w_{12}$

2. Apply Dijkstra's : Time =  $\Theta(VE + V^2 \log V)$

$\hat{w}(p) = w(p) + h(v_k) - h(v_1)$

$w(p) = \hat{w}(p) - h(v_k) + h(v_1)$

So, once we relabel the edges then the next step is to apply the Dijkstra's for all pair shortest path because there is no negative cycle. And this will take time  $O(VE + V^2 \log(V))$ , if we use the data structure fibonacci trees. This path is basically shortest path. Now from  $\hat{w}(p)$  we need to get back the original  $w(p)$ . So, for that we have the formula  $w(p) = \hat{w}(p) - h(v_k) + h(v_1)$ . We want to use the Dijkstra's Algorithm because Dijkstra's Algorithm's time complexity is good compared to all other algorithm like Bellman Ford's.

So, this idea is by Johnson's idea.

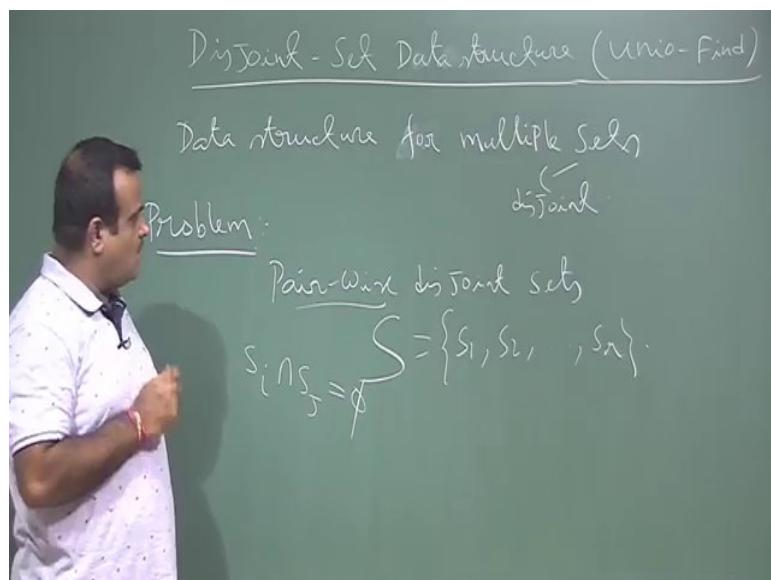
Thank you.

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

**Lecture – 51**  
**Disjoint Set Data Structure**

We will talk about multiple set data structure. So, we have seen the data structure for dynamic set, a set is where insertion and deletion is going on. So, we have seen heap, we have seen balanced tree, red black tree. So, these are the data structure we have seen for a dynamic set. Now we want to look at the data structure for multiple set.

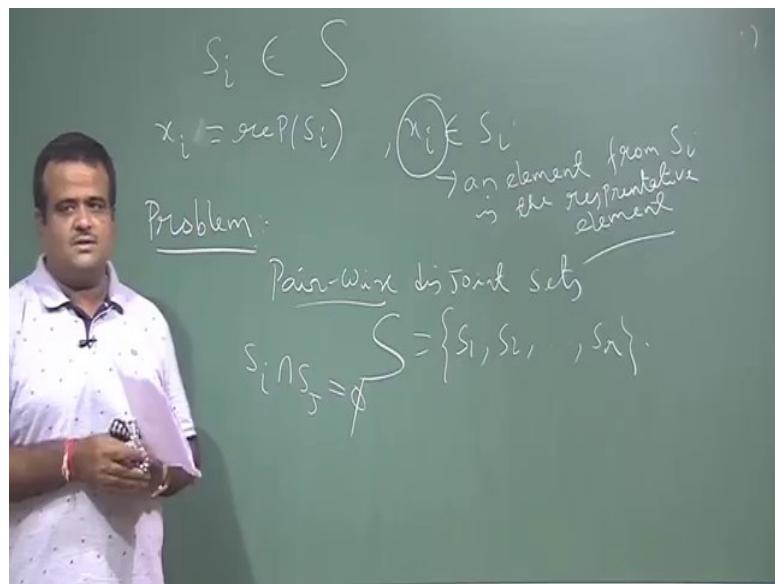
(Refer Slide Time: 00:48)



Data structure for multiple set, and these sets are basically disjoint. So, basically we have dynamic sets of disjoint sets. So, the problem is we have a dynamic collection of disjoint sets say- pairwise disjoint set collection. And this collection we denote by  $S = \{S_1, S_2, \dots, S_r\}$ , and this is a multiple set collection. If you take any intersection of any two they are empty.

So, this is a dynamic collection. We need to have a data structure for this collection of these disjoint sets where we want to support few operations.

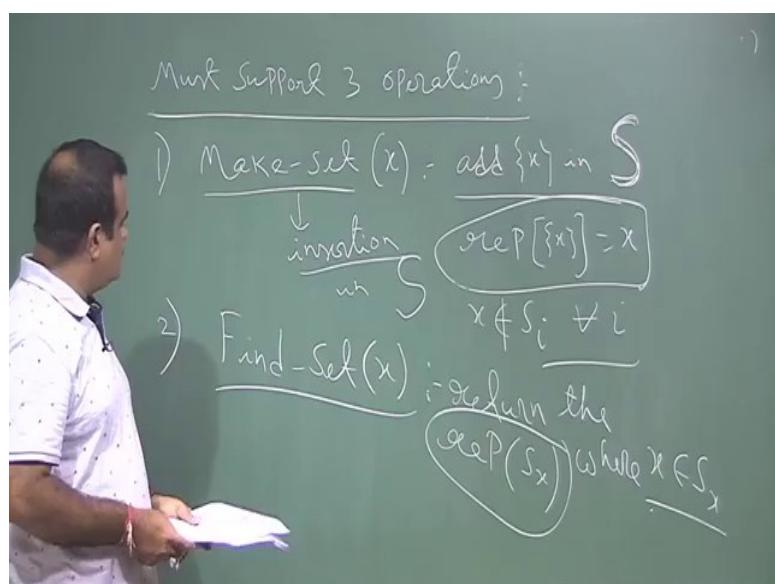
(Refer Slide Time: 02:53)



Suppose  $S_i$  is a set each from this  $S$ . Each set will have a representative element. So, an element from a set. So, this is referred as  $\text{rep}(S_i)$ . So,  $\text{rep}(S_i)$  is an element say  $x_i$ ;  $x_i$  belongs to  $S_i$ . We can choose any element to be a representative element. This is the distinguish element from this set. This is called representative element.

So, given dynamic sets of sets along with the representative element we would like to perform few operations.

(Refer Slide Time: 04:51)



So, this must support three operations. What are those?

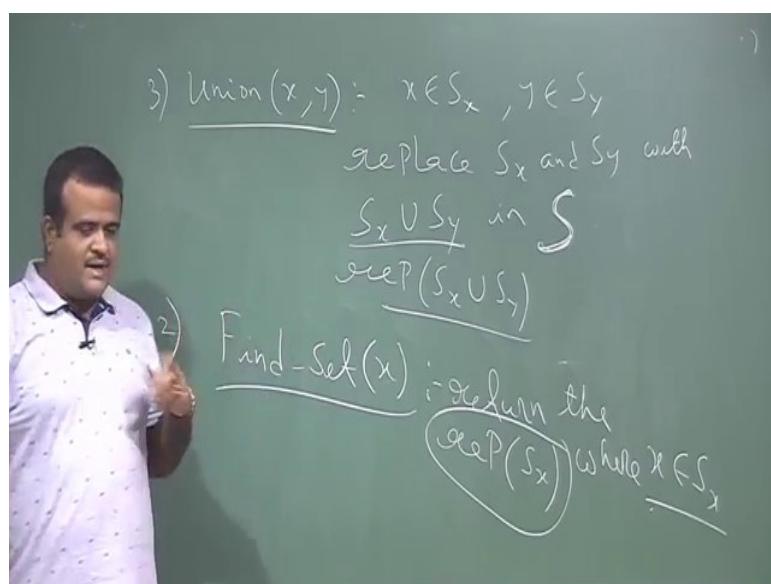
First operation is Make-set( $x$ ). Given an element  $x$ , we have to make it a set which is singleton set. We have to add this singleton set in  $S$ . This is sort of insertion. We are creating a set, so this is kind of insertion operation in  $S$ . Once we create set then have to insert this into that dynamic collection of sets. Then we have to choose the representative element, because each set is having a representative element. Now this set is a singleton set so it has only one element. So obviously, rep of this set is basically  $x$ . And  $x$  doesn't belong to any other set, because this is a disjoint collection. So, ' $x$ ' should not be a member of any other set, because this is a disjoint collection of sets.

Second operation is called Find-set( $x$ ) operation. So, find set operation means we have given an element  $x$ ,  $x$  must belong to some of the set. So, we want to return that set. So, in order to return the set we just want to return the representative element of the set.

So, for that this operation must give us the representative element of the set. So, this is basically returning the  $\text{rep}(S_x)$ .

Third operation is union( $x, y$ ). So, we want to merge two sets.

(Refer Slide Time: 09:03)

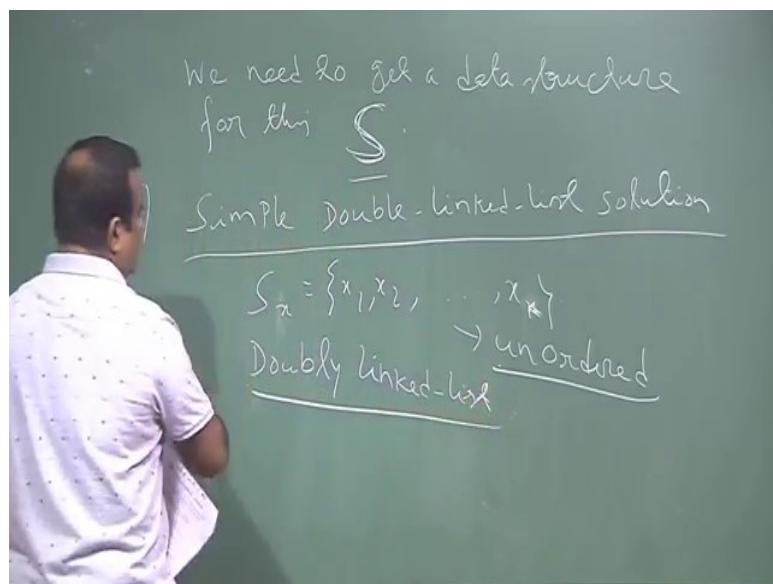


We are given two elements and these two elements will belong to two sets. So, say ' $x$ ' is belonging to  $S_x$  and ' $y$ ' is belonging to  $S_y$ . Now, we want to merge these two sets; we want to do the union of these two set and we want to replace these two sets by their union. So instead of  $S_x, S_y$  which we want to delete, we want to add  $S_x ∪ S_y$ .

Once we replace this then we have to find the representative element of  $S_x \cup S_y$ . We need to decide what will be the representative of this union.

So, this is the problem, this is a dynamic collection of disjoint sets where we must be able to perform these three operations. One is insertion operation we must create a set, then we must find a set. So, in order to return the whole set we can return the representative element of that set. And the third operation is union: we should be able to merge two sets. And so the problem is to get a data structure, we need to think of a data structure for this dynamic collection of sets.

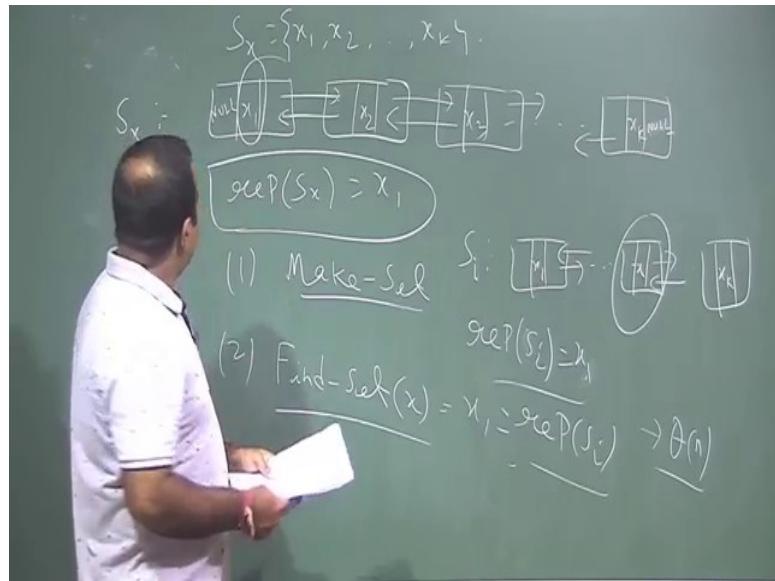
(Refer Slide Time: 11:58)



So, we need to get a data structure for this collection of disjoint sets. So, what data structure you can think of for this collection? So, let us start with simple doubly connected linked list. So, we want to store every set in a linked list. Suppose we have a set  $S_x = \{x_1, x_2, \dots, x_k\}$  there are k elements.

So, this set is an unordered collection, there is no ordering of the elements. So, it is an unordered collection. Given we store these elements in a doubly linked list. So, we store these elements in a doubly linked list.

(Refer Slide Time: 14:20)



So these elements are  $x_1, x_2$ . So there is a pointer from this node ( $x_1$ ) to this node ( $x_2$ ). One field contains address of the previous node and another field contains address of the next node. So, that is the doubly connected linked list.

Finally, we have  $x_k$  if there are  $k$  elements in the set. So, this is our proposed data structure for the sets. So, for  $S_y$  also we have another doubly connected linked list. For each set we have a doubly connected linked list. And we want to see how fast we can perform those operations.

So, now we want to see the operations and how we can perform them. Like how to perform this make-set operation. And now we need to choose a representative element. So, who is the representative element? So, we want  $\text{rep}(S_x)$  to be the first element.

Now we want to see how good data structure is in order to perform these operations. So, first operation is make-set. So, how to make a set from element  $x$  which is not there, because this set has to be disjoint. So, make set means we just create a node with  $x$  keeping both the fields null. And the  $\text{rep}$  of this is the first element ( $x$ ). So, how much time it will take? This will take  $\theta(1)$  time.

Now, how to do a find-set? So, we want to return a set where  $x$  is belonging. That means, we know that  $S_i$ . So, this (refer above image) is the set say  $x_1$  and we have  $x$  somewhere in the

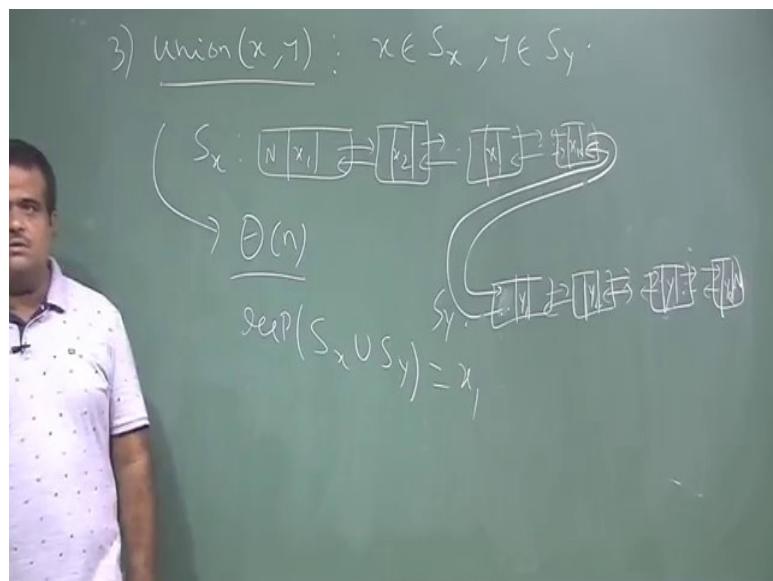
chain then we have this  $x_k$ . We know representative of this  $S_i$  is basically the first node which is  $x_1$ . Now to find the set we are given an element.

We know this  $S_i$  is containing this  $x$ . So now, we need to get the representative element of this set. Then representative element is the first element. We travel the list backward and we reach to the first element and we check whether this is first element or not, we return the first element. And that is the find set operation.

So, we return the  $x_1$  which is basically  $\text{rep}(S_i)$ . So, for that we need to traverse the list from backward, but this is doubly connected list there is no issue with that. So, how much time it will take to do that? It will take say linear time and we do not know where is  $x$ . May be  $x$  is the last element and if the total number of elements is  $n$  then this is a linear time operation.

Then how to do the union? Suppose we want to do the union of two sets. So, that is the third operation we want to perform.

(Refer Slide Time: 21:02)



So, this is the third operation. So,  $x$  is belonging to  $S_x$  and  $y$  is belonging to  $S_y$ . So, we have given two elements, we know where these two elements are belonging, they are belonging to 2 sets  $S_x$  and  $S_y$ . Now we want to replace this  $S_x$ ,  $S_y$  by their union:  $S_x \cup S_y$ . So, that is the union operation. So, how to do that? We have  $S_x = \{x_1, x_2, \dots, x_k\}$  and  $S_y = \{y_1, y_2, \dots, y_l\}$

Now we want to perform the union operation on these sets. So, in order to perform the union operation we just connect the last node of  $S_x$  and first node of  $S_y$ .

So, just we are joining two doubly connected linked list. So, these sets are now replaced by one set. So, we are doing this by just connecting one set with another set with the help of linked list. Just we are adding the pointer of one set to another set. So, this is basically the union. So, how much time it will take and now what is the representative element of this? So, representative element is the first element ( $x_1$ ). What is the reP of this? ReP of this is still remain  $x_1$ , if we merge alternatively it could be  $y_1$ . So, we will see what is the optimal way to merging this.

Now how much time it is taking to do this union? We need to reach to the end of one set. So, that will take linear time. So, this union will take linear time operation, so because we need to traverse the list for  $S_x$  to go to the end, so for  $S_y$  we need to go to the beginning in order to merge these two list.

Now, these we want to do in logarithmic time. So, we want to do it in  $\log(n)$  time instead of  $n$ . Make-set is ok but other like find-set and union are taking linear time. So, that we want to discuss how we can achieve by the logarithm time. We will think of balance-tree can help us for doing that. And then we will see how we can get the better data structure for this problem. That will do in the next class.

Thank you.

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

**Lecture – 52**  
**Union – Find**

So, we are talking about data structure for a dynamic sets. So, in the last class we have seen one solution. So, what was the problem we were given some sets of sets.

(Refer Slide Time: 00:34)

So there are say  $r$  set and this is dynamic set; that means, we can insert any set. So, we are looking for a data structure for this dynamic collection of sets and we should be able to perform the operations like make set. So, we are given element  $x$  we should be able to. So, so from this  $S_i$  there is an element which is called rep of  $S_i$  representative element, which is representing the whole set  $S_i$ .

Now, make set is basically we are  $x$  is not belongs to any of this given sets. So, we are going to add this collection and the rep of this is basically because there is only one element. So, this is one operation another operation is find set. So, you are given an element. So, you want to return the set where this element is belonging. So, in order to determine the whole set we just return the representative element. So, you return the rep of  $S_x$  where  $x$  belongs to that is that set. So, this is the operation and then the union.

So, we want to do the union of  $x$   $y$ . So, you are given 2 elements say  $x$  belongs to  $S_x$  and  $y$  belongs to  $S_y$ . Now we want to replace  $S_x$  and  $S_y$  by their union. We replace  $S_x$  and  $S_y$  by  $S_x \cup S_y$  and we want to have a representative element of this new  $S_x \cup S_y$  set. So, this is the 3 operations we need to perform. So, we have seen one data structure that is the doubly connected linked list.

(Refer Slide Time: 02:57)

So, we represent each  $S_i$  So, say  $S_i$  is  $x_1 x_2 \dots x_k$ . So, this we have seen we use a  $x_1$  sorry.  $x_2 \dots x_k$ . So, dot, dot, dot. So, this is basically null this is null.

So, this is the way we represent a set. So, using a doubly connected linked list. So, we have seen in the last class this data structure. And representative element is the first element of the first node. So, this is an unordered list because the sets itself is unordered. So, this is an unordered list. So, this representative element is the first element of that first node; the value of the first node. So, that is  $x_1$ . So, this is the way and we have seen how to make a set. Making a set means we just create a node. And this is a basically collection of doubly connected linked list. So, this is a data structure we are using. And then we have say in the find set we have  $x$ . So, we go there and we traverse the list and we back to the first element. And we return that and union; union means we have added the another doubly connected linked list.

So, for that also we need to travel. So, this find set and union took us order of  $n$  times if the size is  $n$ . So now, we want to think how we can do it in better way? Like at least  $\log$

$n$  time. So, for that we know any data structure which can give is log in performance? So, what about balance tree? So, we can think for a simple balance tree solution.

(Refer Slide Time: 04:57).

So, let us this is the second proposal simple balanced tree. So, one balanced tree we have seen in our class that is red black tree, this is a balanced tree. So, the idea is whether we can use the balanced tree for our set.

So, how we can do that? Suppose we have a set  $S$ ,  $S \in$  which is basically say  $x_1 x_2 \dots x_k$ . So, we want to have a tree for this and which is a balanced tree. So, we do not care about the key value because this set is unordered. So, this is the unordered set. So, there is no ordering of this element. So, there is no question of say binary sets tree property. Because there is no ordering of the element. So, this key we are ignoring the key value.

So, once we form this red black tree we have a key value and based of that key value we form the binary statistic property. I mean any node in the left of tree is less than the key value of the root any node in the right greater than the key value. So, that we are ignoring the key value because the sets is the stay on or collection of disjoint element. So we have a tree. So, we just make a balanced tree. So, we have a balanced tree while we are not bothering about the value of this nodes because this is say this is just unordered collection.

So, but the structure should be balanced. And then what is the representative element? So, representative element can be the root of the tree. So, that is the way. So, for example, suppose we are given this say 5 element say  $x_1 x_2 x_3 x_4 x_5$ . So now, how to form a tree? So, suppose this is  $x_1$  and we do not care about the value because this is unordered list. So,  $x_4$  can be come here  $x_3$  we just care about the structure this tree should be balanced the height is should belonging  $x_2 x_5$ .

So, suppose this is our  $S_i$ . Now what is the rep of  $S_i$ ? rep  $S_i$  is basically the root, rep of  $S_i$  is basically  $x_1$ ; root of the element. So, this is our data structure. So, we just make a balanced tree with this unordered list and unordered with this elements and we just ignore the key so, but the structure should be balanced height should belonging. So, this is our proper data structure, now we can perform those 3 operations like make set. So, how to make set? So, first operation is make set.

(Refer Slide Time: 08:54).

So, we are given a element  $x$  which is not there in the collections. So, we want to create a set we want to insert a set in; this is the insertion operation in that collections keep test. So, this is basically we just create a node that is it we just create node  $x$ . This is a single node tree just create a node and this is the rep of this  $x$  is basically the first element this is root is  $x$  itself. So, this is the way. So, what is the time complexity of this- constant So now, we want to see how we can do the find of  $x$ .

So, find of  $x$  take how long time. So, find of  $x$  mean suppose we want to find  $x$ . So, you suppose this is our  $x \in S$ . So, we know the position of this. So now, we have to return the representative element of this. So we have to get the parent of this. So, we return the  $x = 1$ . So, that is the operation for find set. So, we want to return this  $S[i]$ , because we know  $S$  is belongings to  $S$  if  $x$  is say  $x \in S$  now we have to return these  $x = 1$ .  $X = 1$  is the root of that. So, in order to reach to the root, what is the time? Time is  $\log n$  because this is the balanced tree.

So, this will take  $\log n$  time. So, find operation will take  $\log n$  time. Because we know the  $x$  say we want to find the set where  $x \in S$  is belonging. So, that is basically  $S[i]$  that is basically this tree now we have to return the representative element of this that is basically the root. So, to reach to the root we have to go to the height of this tree. Because this is the balanced tree. So, this is  $\log n$  now how to perform this union operation? So, that also you want to see whether you can do it in  $\log n$  time or not.

(Refer Slide Time: 11:34).

So, suppose we have 2 element  $x$  and  $y$  suppose say  $x$  belonging to the set and  $y$  belonging to the another set and we want to replace this 2 set  $S_x$  and  $S_y$  by their union and we have to.

So, how to do that? So, this is suppose this is our  $S_x$  this is our  $S_x$  is somewhere here and we have a another set say  $S_y$ . So, if this is say  $S[i]$ . So, this is say  $y_1$  say  $y_2$   $y_4$   $y_5$   $y_6$  something like that. So, we have 2 sets  $S_x$  and  $S_y$ , now the question is how we can

merge this 2 set? How we can merge this to tree, and in order to get a balanced tree again. So, we have to get the ultimately this union should be a set and union is a set and that is also be a tree. So, that has to be balanced. So, what is the proposal?

So, we can have a new root like we can have this S element over here then we can do like this kind of thing. So, this is already a balanced tree. So, we can just keep on insert these nodes. So, we have y 1 y 2 like this. So, we can insert this node into this tree. So, that again we will take in  $\log n$  time because if there are  $n$  elements in the set  $y$  order of  $n$  element then this itself will take  $1 \log n$  time, but we want to do it in  $\log n$  time. So, that is the idea of concatenation. So, if we have a 2 red black tree. So, that is the idea of concatenation like.

(Refer Slide Time: 13:58).

So, this we will do the concatenation of 2 trees concatenates this 2 red black tree.

So, like we have this tree we have a bigger tree. Let say this is say  $S_x$  we do not know  $S$  a  $S_y$  may be bigger. So, for exam this is another tree  $S_y$ . Now we can add this tree over here by concatenate operation we have seen this operation in a red black tree. So, this is basically these we will take and this is basic we are just concatenate this tree over here. And this will take  $\log n$  time the height of this tree. So, basically we take this element then we search the position of this element. So, this way we will do. So, this we will take  $\log n$  time. So, this is the concatenate operation. So, this is  $\log n$  time. So now, we want

to do something better on this like, we want to do  $\log \log n$  time, how we can achieve a  $\log \log n$  time?

So, what we are going to do now is called data structure augmentation. So, we want to argument our data structure So that we can do the operation in faster way. In a amortized sense. So, we want to do lots of amortized analysis. So, there we will see how we can get  $\log \log n$ .

(Refer Slide Time: 15:58).

So, this is the next things we will discuss this is the let us give a outline of this what we are going to do that is called plan of attack. So, this is sort of outline what we are going to do. So, this is basically bunch of augmentation you will do in our data structure. So, basically you have to basic data structure one is w connected linked list another idea is the tree.

Here it is a red black balanced tree. So, we will see a general tree. And then we will see how we can argument this 2 data structure and we will do some amortized analysis in order to get the performance is  $\log n$ ,  $\log \log n$  like this. So, the idea is we will build a simple disjoint union data structure data structure that in amortized sense we will do the amortized analysis. Amortized sense perform significantly better than  $\log n$ , perform significantly better than  $\log n$  per operation. So, we have basically 3 operations ok.

So, you want to make it faster on this 3 operation like make anywhere make set is always constant. So, that we are not bothering, but other 2 operation like find set for the w connected linked list find set we will take order of  $n$  time. So now, the question is how we can do some sort of augmentation or how we can do some. So, after augmentation how we can do the analysis amortized analysis to get it  $\log n$  not only  $\log n$  operations even better than even better than  $\log n$ . So, better than  $\log n$  is means  $\log \log n$  and so on, but not constant because  $\log \log n$ . So, it is not constant basically, but it tending to. So, this log this we have seen in one data structure what is that the van Emde Boas data structure ok.

So, there we have seen the fix in our successive problem. So, they are we have seen  $\log u$ , but there we have used 5 tricks to achieve that, but here we are going to use the 2 tricks only to achieve this. So, to reach this goal we will use 2 tricks. This goal, we will introduce 2 tricks. Each tricks convert this operation form  $\theta(n)$  to  $\log n$ . Each tricks convert a trivial solution a trivial  $\theta(n)$  solution into a simple  $\log n$  solution. So, that is the going simples  $\log n$  solution each of this string. So,  $\theta(n)$  solution means we have seen a doubly connected linked list. So, but this will be in amortized sense this analysis is amortized analysis. So, we will talk about that.

So, first trick is on this doubly connected linked list. So, we will do some augmentation on this doubly connected linked list and then we apply this trick. And the second trick is on the tree. So, we will apply the trick on the tree data structure. So, let us talk about first trick I mean, let us go back to the data structure which is we using for on the w connected list. So, we will do some augmentation.

(Refer Slide Time: 21:56).

So, augmentation on the linked list. So, we have a linked list solution. This is basically doubly linked list. So, doubly linked list linked list we will do some augmentation in the doubly linked list data structure to have the better performance on this operations.

So, just to recap. So, we have a said S i. So, it should be this is  $x_1 \ x_2 \ x_k$ . Now what is our data structure or data structure? Is just simply a w connected linked list. So,  $x_1$  So, this is our S i. So,  $x_2$  dot, dot, dot  $x_k$ . So, this is basically this is null this is null. So, this is the data structure we have use this is the doubly linked list we have used for our this problem. Now we have seen for this data structure suppose we have a node S of what is the find operation of x suppose x is here. So, we know the position of x.

So now to reach to the representative element what we did we have to come back to the beginning. So, that was taking  $\log n$  time. So now, how we can augment this in order to achieve this in a constant time. So, how to augment this data structure So there we can achieve this in a constant time. So, any idea? How we can make this operation find set faster? I mean theta one time in set of theta of n times. So, you have to do some augmentation in this a list. So, what we do? So, what if we store So each element in each element other than this we store the representative element; that means, each element we have a list to the each element we have a list to the beginning. So, we have a we store the pointer of the first element.

So, this is the augmentation we are going to do for our doubly connected linked list. So, we store each element  $x_j$ , also store pointers and that pointer is called rep of  $x_j$  pointer to Rep of S i.

So, that is basically the first element. So, you have storing the pointer of this  $S_j$ . So, each of this will point to the first element. So, that then the find set is easy. So, if we have a link from this to this. So, suppose this is  $x$ . So now, to get the representative we just take this pointer and we take this  $x_1$ . So, that will return. So, that will find set we will take constant time, find set we will take constant time after doing this augmentation. So, this is not a amortized analysis this is the exact analysis. So, this will take constant time. Now we want to see how we can manage the union.

We have already seen make say make said that is constant you are not bothering about that. So, we will see how we can manage the union operation. So, that is our next discussion how we can achieve the union operation, how we can perform the union operation of this 2 set. So, what type of augmentation we will do because for union if you have another set  $y$  So, if you have another set  $y$ .

(Refer Slide Time: 26:49).

So, say  $y_1$ . So, this is say we have say 2 element in this set. So, this is only 2 element say  $x$  and we have a  $y$  which is say 4 element.

So, this is also a doubly connected linked list. So, this is say x and this is somewhere y is there. So, this is S y this is y 1 this is the representative element of S y. Now we want to do the union, union of (x,y). So, for union what we are doing? We are just linking this to this and this to this. This is the union operation we are doing. So, for that what we have to do? We have to go back from here to here so, but we have the link for this. But we want to do further augmentation. So, now if you just join this like this, then again we have to for all this element we have to go to the beginning.

So, this we will take again linear time if the size of this is linear. So, anyway you will continue this in the next class.

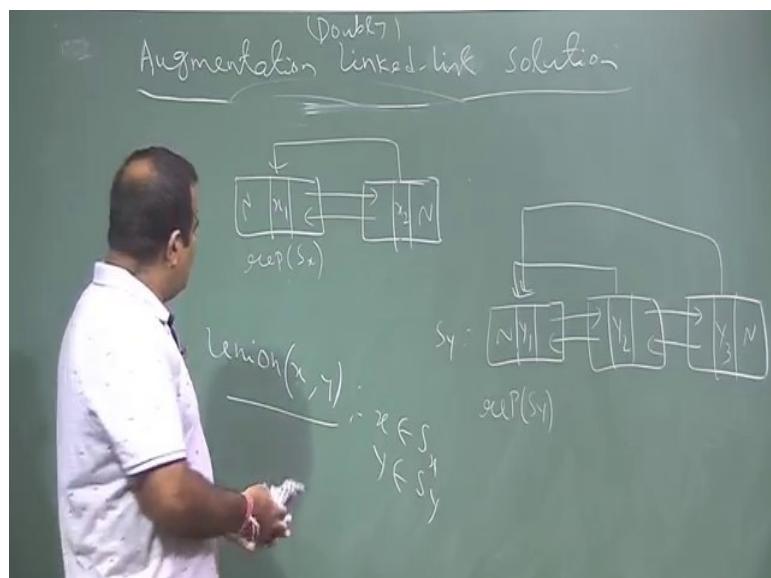
Thank you.

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

**Lecture - 53**  
**Augmented Disjoint Set Data Structure**

So we are talking about union operation, how we can do the augmentation in the w connected linked list. We are going to add a pointer from each element to the representative element that is the first element, because this w connected linked list and our first element is the representative element.

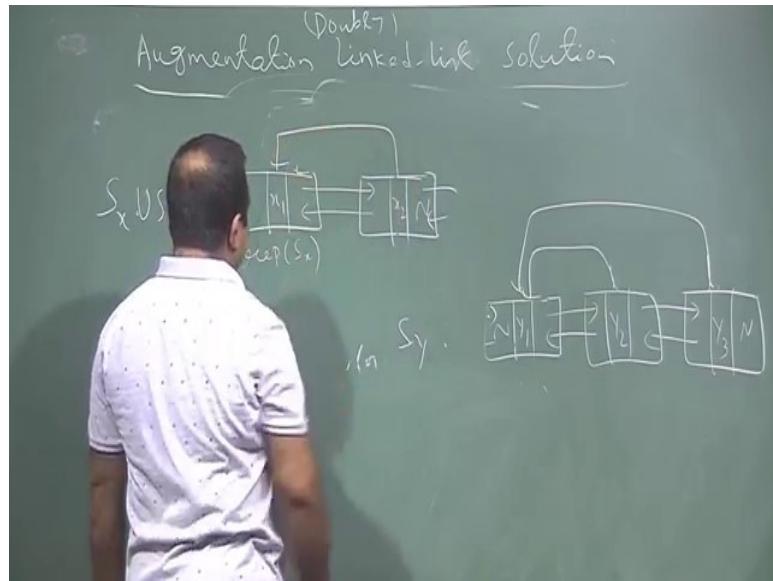
(Refer Slide Time: 00:34)



So, we are going to have. So, now, we are talking about the union operation. So, the union of 2 set union of  $x$   $y$ . So, basically a  $S_x$ . So, basically you have two sets  $S_x$  and  $y$  belongs to  $S_y$ , now we want to have the union of these two sets. So, that you want to do. So, this is set  $S_x$  and this is set  $S_y$ ,  $S$  is here  $y$  is somewhere here in any one of this.

So, for the union what we are doing? So, we are doing the union. So, union is basically  $S_x$  union  $S_y$ .

(Refer Slide Time: 01:23)

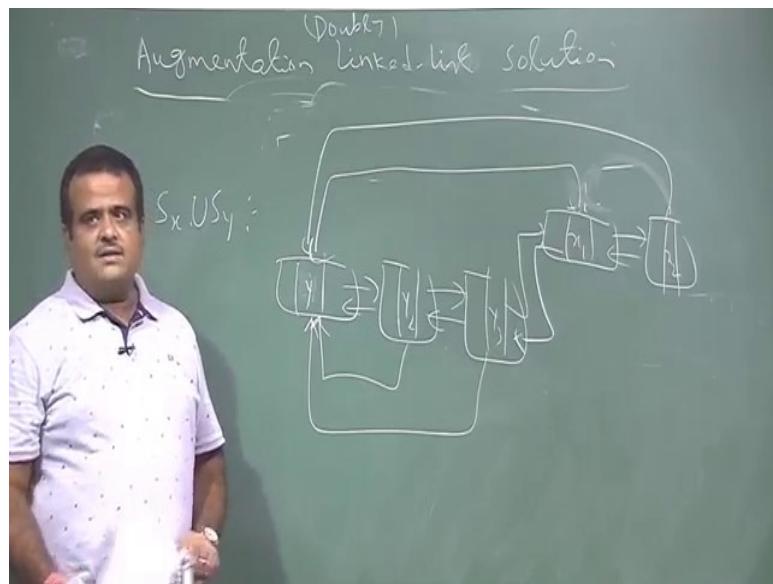


So, for that we are just adding this. So, this last pointer we are just adding this, and then this pointer also we have to change. So, this was the now this first element is the representative for this also, all this nodes. So, this we are going to change now so for this nodes also all this nodes having the representative element S<sub>1</sub>. So, this is the operation you are performing.

So, this is the union operation. So, how long time it is taking? So, now, the bigger set if it is order of n, then we are just taking theta of n times, because everybody has to change the pointer because this is the bigger set say now so what is the idea? Idea is to so, instead of merging this with this if we merge that does not matter if we merge this with this. So, that is the trick one. So, we merge the smaller set into the bigger set. So, that is the idea of trick one.

So, then I can save it sometime because this is if this size is n and I need this size is some constant then hardly we are changing the pointer of this side. So, instead of this what we do? So, this is our S<sub>y</sub>. So, everybody was pointing here in S<sub>y</sub> now this was pointing here now we are merging this set with this set. So, this is the at the end. So, this is let us try it again. So, instead of merging S<sub>x</sub> with S<sub>y</sub> we just going to merge.

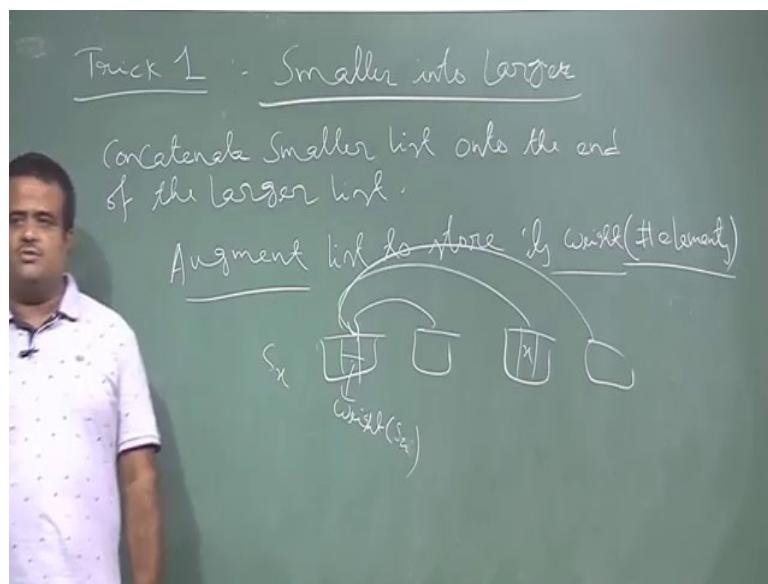
(Refer Slide Time: 03:26)



So, this is our union now. So, this is  $x$  1 sorry this is now  $y$  1. So,  $y$  2  $y$  3 and now we are merging this, this is the smaller set  $x$  1  $x$  2 now all this  $y$  here this was  $S_y$  and now this was pointing here and it pointing itself now we have a connection over here and all these are now pointing to here. So, this is the trick is we want to merge this smaller set with the larger set. So, that way if this size of this constant, but in the worst case it is order of  $n$ , but we will do the amortized analysis on this and we will see this operation we take in  $\log n$  time.

So, this is the idea of merging the smaller set with the bigger set. So, now, the question is how to get the smaller set. So, the idea is to. So, this is the trick one.

(Refer Slide Time: 05:13)



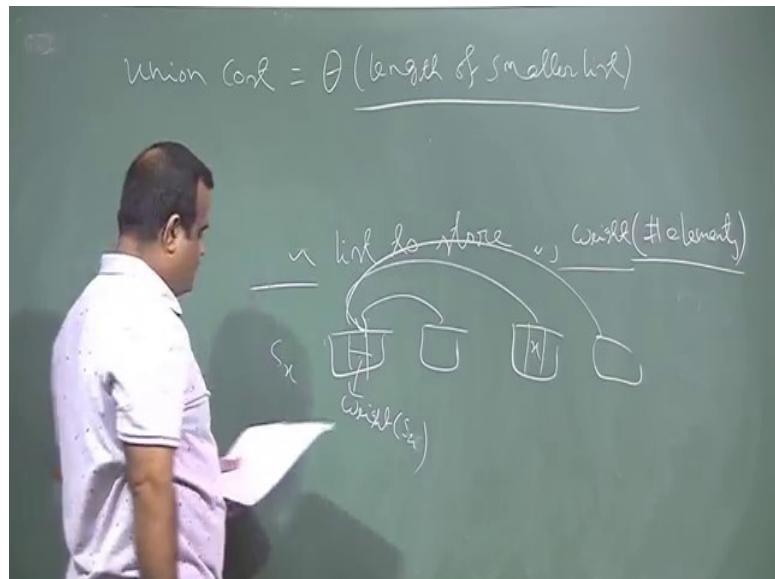
So, trick one. So, that is smaller set into larger. So, this type of trick we will use lot of other places also. So, the idea is to concatenate. So, we are concatenating two list. So, concatenate smaller least on to larger least, on to the end of the larger list. So, that is the idea. So, we are just because the earlier we are doing S is y we are concatenate with this, but is smaller. So, now, we after applying the trick we are going to use the S is smaller. So, we are going to concatenate S in at the end of y list. So, that is the idea.

Now, the question is how we can decide which list is smaller and which list is larger, then we need to try get the length of the list. So, again if you have to calculate the length then you have to traverse the list. So, that will take again linear time. So, to save that you have to do again the augmentation. We augment the least to store the size of the list to store that is called weight to store its weights weight means number of elements this is the number of elements. So, this is the new augmentation we have to give. So, we are going to store the e on each element because anyway we have access to the representative element. So, there we can store the size of the list. So, suppose we have given this is a least this is a list and every pointer pointing here. So, we have given a x, now we can reach here somehow if we can store the weight of weight of the list. So, this is S i or this x i. So, then we came to know which one is larger and which one is smaller then we can apply this trick ok.

So, if we have apply this trick, then what is the time for doing this union because. So, we are just not bothering about the find operation because find operation augmentation we did and

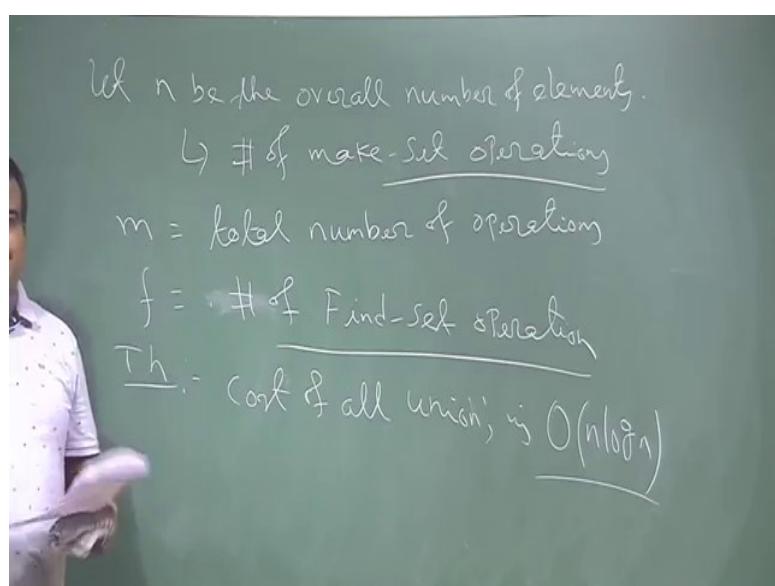
we can solve it constant time. So, we are all the concern here is now the union operation. So, how much time it will take for union.

(Refer Slide Time: 08:39)



So, union cost is basically theta of length of the smaller list or length of smaller list. So, now, we have to go for amortized analysis to show that average cost is log n. So, for that we have to do the amortized analysis. So, this is the without amortize analysis. So, now, we have to do the amortized analysis.

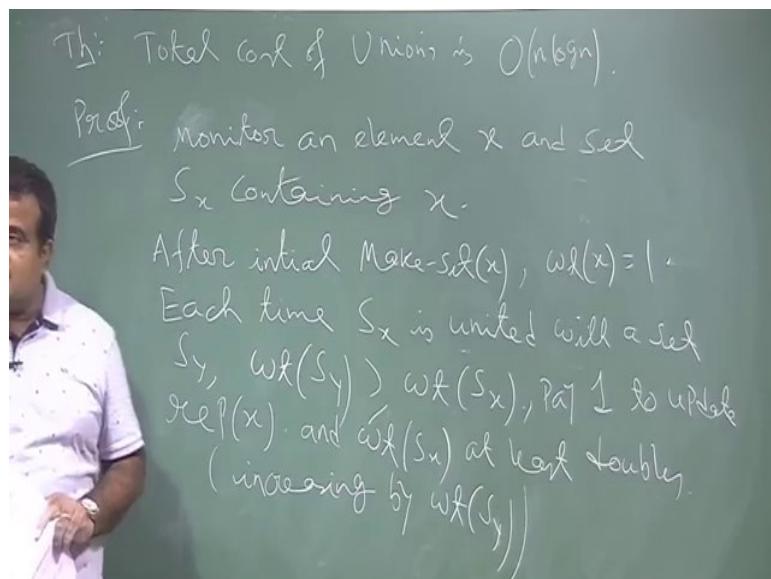
(Refer Slide Time: 09:38)



So, for that let us just denote some. So, let  $n$  be the overall number of elements. So, these are coming by say make sets. So, every time we are making set and doing the union and we are so, overall this is the order of  $n$  times you are doing the make set. So, this is basically equivalent number of make set operation. So, that is how we get the elements. Once we make set we get a single term set then we union. So, this is the way we got the elements. So, this is the number of make set operation, this is the way how we get the sets. So,  $m$  is the total number of operation and  $f$  is denoted by the number of find operation. So, number of find set operation. So, these are the notations will use in our time complexity in our amortization analysis. So, now we are going to prove theorem in a amortized sense, that the cost of all unions because worried we are worried about union for this augmentations is order of  $n \log n$ , there are  $n$  elements overall. So, if cost is order of  $n \log n$  in a amortize sense, then the average cost is there are  $n$  element average cost is  $\log n$ . So, that way we are going to prove this.

So, let us prove this theorem, the cost of total cost of union operation is  $n \log n$ .

(Refer Slide Time: 12:19)

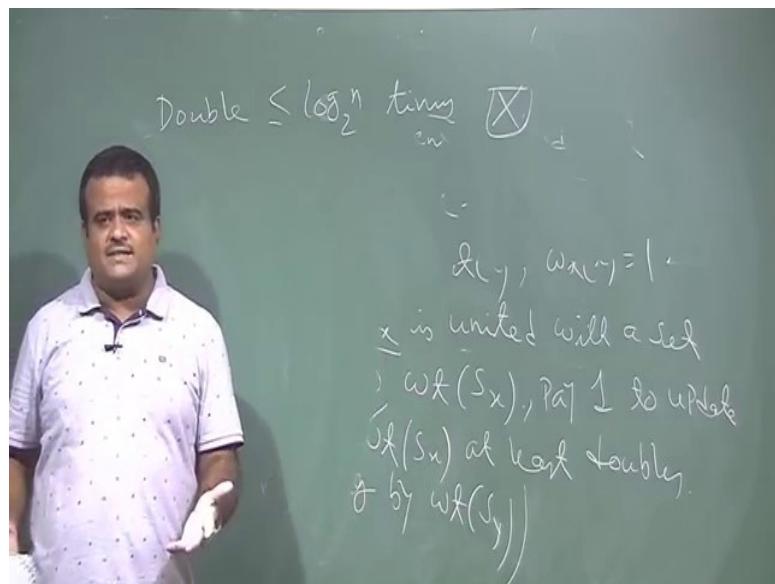


So, let us just state this theorem. So, the total cost of unions operation is order of  $n \log n$ . So, how to prove this? So, this is in the amortize sense. So, basically we choose  $x$  and we monitor  $x$ , and its set  $S_x$  containing  $x$ . So, we will see how many times it will get change and what is happening with this. So, that is our interest and there are  $n$  elements. So, you can just multiply with that ok.

So, we want to see the cost for this union overall cost for this union for this particular element  $x$ , and there are  $n$  elements. So, for this  $x$  we are just doing the how many times? We are just doing the  $\log n$  times then overall we are doing  $n \log n$  time. So, how to prove? So, how we are getting  $x$ ? So, after initial make set, we got  $x$  by doing the make set operation. So, that time weight of  $x$  is basically 1 because that is the single term element, and we in the union operation we unite  $x$  we merge  $x$  with some other  $y$ . So, each time  $S_x$  the set which containing  $x$  is united. So, well we are doing the union operation with  $S_y$  united with the set  $S_y$ . So, we are just doing union of  $x$  and  $y$ . So, we are doing this. So, now, our trick is we are going to merge the smaller set to larger set. So, if  $S_x$  is smaller set then only we are going to change all the element of  $S_x$  to this. So, that is the idea so; that means, if the weight of  $S_y$  is greater than weight of  $S_x$ . Then  $S_x$  is smaller than  $S_x$  is going to merge with  $S_y$  and then we are changing all the pointers of  $S_x$  to the representative pointer of  $S_x$  all the elements for that; and then we are we are paying some we pay one to update. So, there if this if  $S_x$  comma then we pay one say one dollar to update rep of  $S_x$ , because then we are going to because this is the smaller set then rep of  $S_x$  so that means, and then the weight of  $S_x$ , then  $S_x$  and  $S_y$  will be replace by the union then the weight of the  $S_x$ . So, then the weight of  $S_x$  will be how much? The double then the weight of  $S_x$  will increase by at least double because the weight of a  $S_x$  weight of  $S_y$  is greater than that. So, weight of the new set will be the weight of this plus weight of this.

So, it will be double. So, that is why this two is coming that  $\log n$  is coming. So, that is the reason this  $\log n$  is coming. So, this is basically increasing because the weight is increasing by the weight of  $S_y$ , which is basically greater than  $S_x$  so that means, weight is becoming double. So, that is the reason this  $\log n$  is coming  $\log_2 n$ . And if  $S_x$  is more if  $S_y$  is united with  $S_x$  then we are not paying anything. So, then the how many times we are doubling? Then we are doubling just  $\log n$  time.

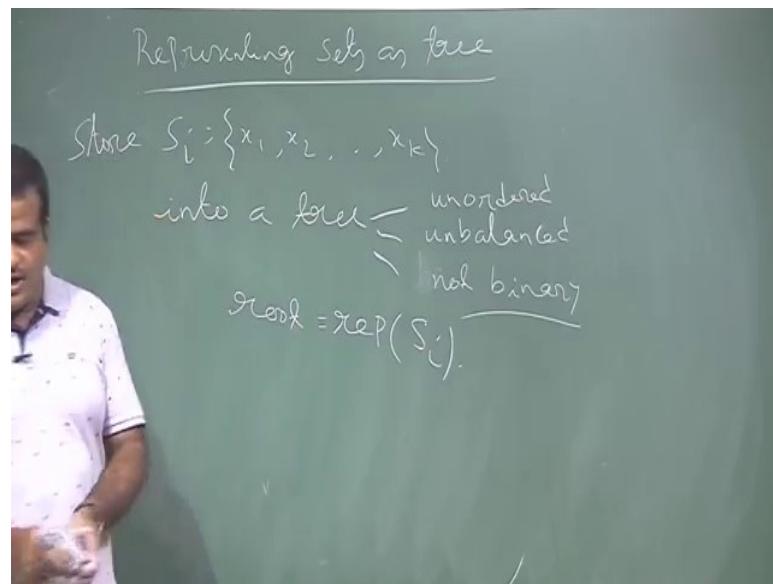
(Refer Slide Time: 18:13)



So, we are just making double is less than  $\log_2 n$  times. So, this will be the proof so; that means, that is why the  $\log n$  is coming because it is doubling. So, this is the proof. So, this is the one first trick.

Now, in order to bring the second trick, this analysis is amortize sense. So, amortize sense our total time complexity is the total union operation is  $L \log n$ , now if we have. So, we have taken the one  $x$  for that  $x$  it is  $\log n$ . So, there are  $n$  elements. So, it is  $n \log n$ . So, this is the total cost in a amortize sense. So, now, average cost is  $\log n$ , so for amortize sense it is average cost is  $\log n$ . So, now, we will discuss the second trick; trick 2 for that we need to bring the data structure which is tree.

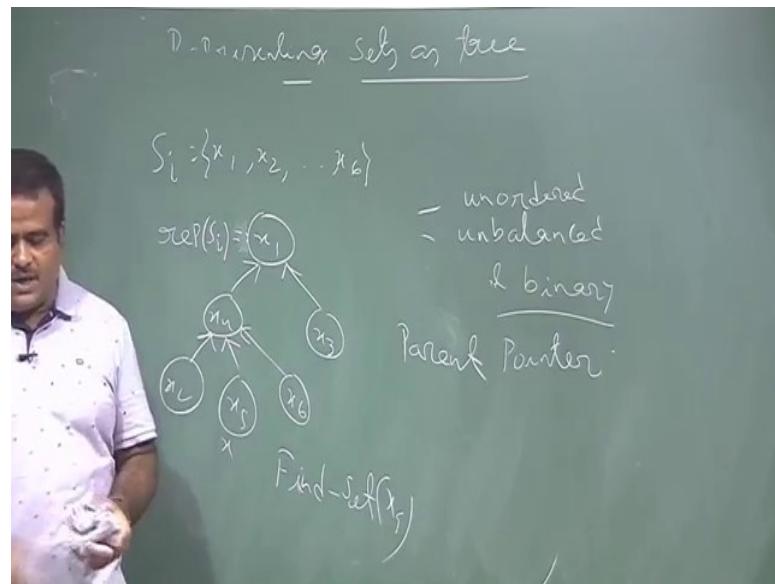
(Refer Slide Time: 19:35)



So, let us again bring the tree representation representing sets as trees. So, here we are not dealing with binary tree or it is not a balanced tree. So, suppose we have a set  $x, x_1 x_2 \dots x_k$ , now we store this into a tree and these trees unordered because our sets are unordered collections, unbalanced here we are not bothering about balance ok.

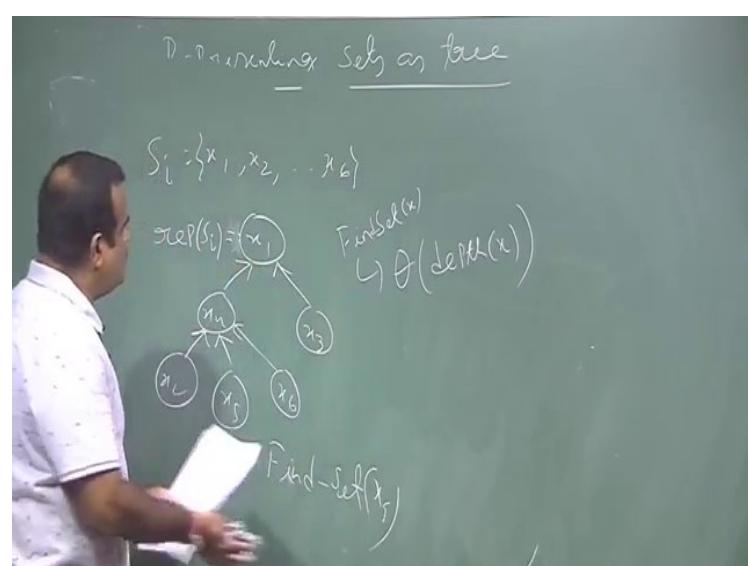
So, and not necessarily a binary not a binary I mean it could be any tree, I mean number of children could be more than 2. It could be 3 4 it could be 2 also it could be null. So, it is not necessarily binary tree it could be ternary it could be I mean it is not necessarily binary tree and the parent is basically the root, root of the tree is the representative element rep of  $S_i$ . So, basically you have a tree. So, suppose we have a set say  $x_1 x_2 \dots x_6$ .

(Refer Slide Time: 21:18)



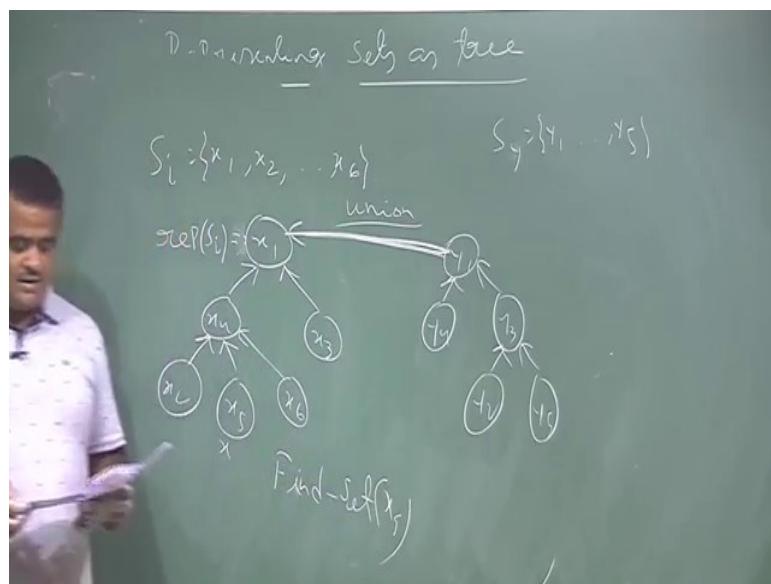
So, let us draw a tree  $x_1, x_4, x_3, x_2, x_5, x_6$ . So, this is not a binary tree and this is this is not a balanced tree also. So, now we do not have child pointer we only have parent pointer and this is the root rep of a  $S_i$  is this one the root of the tree. So, we do not care about the child because no need to care about the child pointer. So, why we need parent pointer because we need to find the set? So, we have to return this root element. So, suppose this is our  $x$  say, suppose we want to do the find set of  $x$ . So, what we do? We look at the parent pointer this way we reach to the root and we return  $x_1$ . So, that will take height of the depth of  $x$  basically.

(Refer Slide Time: 23:03)



Now, this is our data structure, now how to perform this operation like make set make set is easy just we make a these tree I mean a singleton tree that is it, and the find set is this one an union. So, find set will take how much time? Find set will take order of depth of x, this is the find set cost find set will take order of depth of x. Now what about the union? A union suppose we have a another set y.

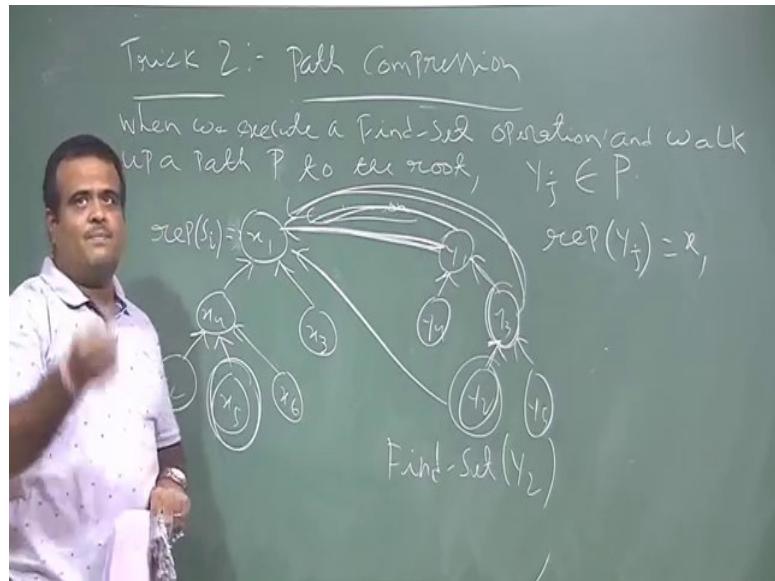
(Refer Slide Time: 23:41)



Y 1 say y. So, we have another set S y, y 1, y 2, y 5 say. So, y 1, y 4 y this is another tree we have y 2, y 5. So, now, this tree is also like this, we have all the parent pointers now how to find set.

So, we can add this root to any one of this node, that we'll do because if we add this root to any one of this node, then the we can just from this we go to that node and then we visit to the root or else we can just add this two here. So, this is the union operation we are doing. So, now, the question is if we adopt the trick one here. so; that means, if we just add the smaller set to the larger set then how it is helping us.

(Refer Slide Time: 25:11)



So, that is the next discussion we are going to do. This set is completely unordered. So, this is the trick one adopted to trees. So, that means the smaller is merging with larger. So, if we can find the smaller tree then we can add it to the larger set then the find will take the same time as find of  $x$  and height of the tree will increase because height of the tree will not increase because we are having smaller set we are adding merging with the larger set. So, height will remains same.

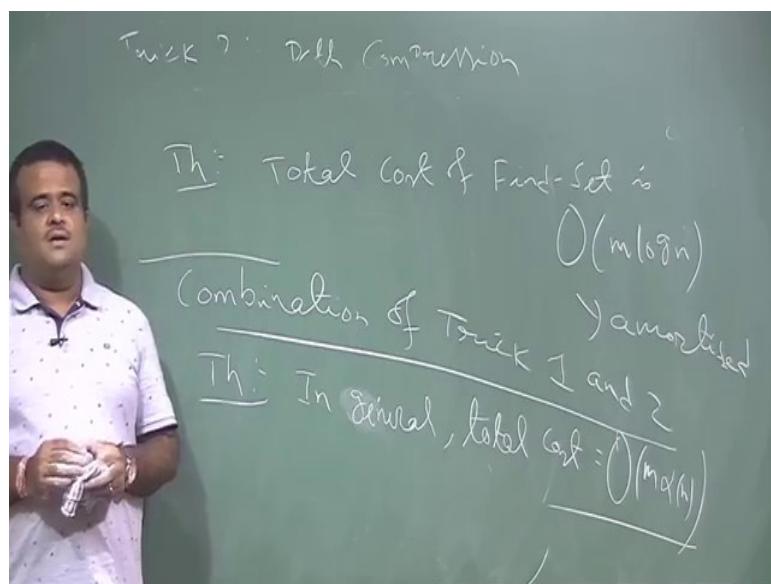
So, this is the trick one, now let us talk about trick 2. So, that is the trick 2 which is also referred as path compression. So, what it is telling? Now this is telling all about the find. So, when we perform a find operation say we are performing the find operation on this set say  $x_5$  or say yeah  $x_5$ . So, what say we are no we have say we are performing the find set operation on say  $y_2$ . See if you perform the find set operation on  $y_2$  then we go to the parent of this. So, once we got the parent of this, we have parent pointer from here to here this is the new augmentation we are doing. So, that if the next time we again search for the find  $y_2$  then we will get it directly. So, this is a path not only this. So, we are going to add the parent pointer for each of this node in this path.

So, that is the idea of trick two. So, that is called path compression. So, when we execute a find set of a node, find set operation and walk of a path  $P$  to the root. So, we know the representative. So, we are going to change the representative element of each node in that path. So, if a node  $Y J$  belongs to this path  $P$  then we are going to change the rep of  $Y J$  to be

x 1. So, we are going to have a. So, direct pointer to this. So, direct pointer to this all the elements in that path. So, this is called trick 2.

So, this is the operation we are doing. So, this is basically called compression path compression. So, instead of only one element we are just compressing the path I mean we are taking all the node, representative element of all the node, we are changing to this root. So, this we can have a theorem to. So, this is in amortize sense. So, let us just write the theorem we then have time to prove it just will state the theorem. So, this theorem is telling.

(Refer Slide Time: 29:22)



The total cost of find set is order of  $n \log n$  amortize sense.

So, this is the amortized analysis. So, there are  $m$  times we are doing this operations. So, now, the on an average it is  $\log n$ . So, this proof we are not doing. So, now, this is if we only applied trick 2. Now if we combined trick 1 and trick 2. So, that is the combination of trick 1 and 2. So this is more interesting if we apply both the tricks then this is the general case, this is the another theorem in general if we apply both the trick the total cost is order of  $m$ .

So, this is the again in amortize sense. So, this is the amortized analysis we did for our data structure augmentation and we apply the two trick one is the we are going to merge the smaller with the larger we are or we are going to combined the smaller with the larger and the second one is path compression. So, we are going to change each pointer of this representative element of each pointer of this set node on this path to the root.

Thank you.

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

**Lecture – 54**  
**Network Flow**

So we talk about network flow or flow network. So, what is the problem? Problem is we are given a graph.

(Refer Slide Time: 00:28)

So, this is the definition. So, if flow network is a directed graph or digraph; directed graph  $G(v,e)$  with 2 distinct vertex which is called one is source other one is called sink with 2 distinct vertex distinguish vertex vertices  $s$  and  $t$  this is called source and this is called sink and each edge there is a weight associated with each edge and that is called capacity.

So, each edge  $u v$  has a nonnegative capacity. So, some non negative capacity that is  $c(u,v)$ . And between if there is 2 vertex where there is no  $h$  then we different the capacity to be 0. So, if  $(u,v)$  is not to edge then we defined as  $c(u,v)$  this is the convention there is no edge. So, there is no question of capacity.

So, this is called flow network or flow graph. So, basically we will take some. So, basically each node. So, if you have  $u v$ . So, this is a directed graph. So, if there is a edge

from  $u$  to  $v$  then there is a capacity an integer non negative value. So, we'll take an example. So, we'll take such an example of such a flow network. So, it is basically a directed graph. So, let us draw a directed graph.

(Refer Slide Time: 03:24)

So, this is the source and we have a edges these are the direction this is the directed graph these are the vertices these are the edges.

So this is the source that is the sink. And this is the directed graph now we have to put the capacity. So, suppose the capacities are say 3 these are all non negative integer 2 1 3 1 2 3 3 2 3 2 suppose these are the 2 3 2 suppose these are the capacity. So now defined a network positive flow. So, this is another definition. So, a positive flow on  $G$  on this network  $G$  is a function  $p$  which is basically  $v \times v$  to  $r$ , satisfying the 3 conditions.

First condition is called capacity constant. So, we want to flow some current. So, say this is the current these are the capacity of each where now we want to flow some current. Now obviously, if the capacity is 3 then we cannot flow the current more than 3. So, that is the first condition. So, this is called capacity constant. So, for all  $u v$  belongs to  $v$  such that So, this positive flow was  $p$  positive and it must be less than  $c_{u v}$ .

So, if we have 2 vertex in  $v$ , if this is  $u v$  now this is  $c_{u v}$ , now you want to have a  $p_{u v}$ , we want to defined a  $p_{u v}$  positive flow. So, such that it should be less than the capacity because we cannot send more current then the capacity. So, that is the idea. So, this is the

capacity constant. So, you cannot say more than the capacity. So now, second condition is called flow conservation.

(Refer Slide Time: 07:07)

So, it is basically number 2 it is basically telling if you take any vertex other than sink and source.

So, flow conservation should be 0. So that means, we are not storing any current we just passing the current. So, in other words suppose this is  $u$ , this is a node which is not source and not sink. Now there are some edges coming over here and there are some edges going out to there. So, there are  $S_o$ , this is coming flow. This is the current coming to  $u$ . So, there are say more edge this is the current going from  $u$  total. So, this difference should be 0. So that means, whatever current it is taking it should distribute.

So, it is basically it is not holding any current. So, it is just a distributed sort or media. I mean it is it is just a middle man. So, this we have to write in a mathematical form. So, this is basically telling for all  $u$  which is not a source or sink summation of  $p(u,v)$ . So, this is basically the current which is coming from. So,  $u$  yeah, So this is basically current which are going from  $u$  to  $v$  this is  $v_1 v_2 v_3$  like this.

So, these are the all vertices which is just basically out going vertices this sum. Then minus the sum of a capacity  $v$  comma  $u$  this is. So that means, this telling this is the all the vertices  $v_1 v_2$  which is basically. So, this is basically sum of all this. So,

whatever current we are having you are passing that. So, these sum should be this difference would be 0. So, we are not any middle nodes other than source and sink are not conserving any current I mean it is just passing all the currents.

So, this is sort of kirchhoff's current law. So, whatever current is coming it is just going out. So, in that holding any current in the middle point. So, this is the definition of positive flow if the there are 2 condition, one is the flow the  $p(u,v)$  should not be more than the capacity and in the middle not we should not have any conservation of the current. Now let us take an example of the positive flow on this network. So, this is a given network flow where capacities are given and this is the source and that is the sink.

So now we want to have a positive flow on this network, which satisfy this 2 condition.

(Refer Slide Time: 11:01)

So, you can have something like 1, 1 flow on this then 2 is the capacity we can have at most 2 flow this is the 0 then this is 3. So, we can have 1. So, this is the basically positive flow, this is basically positive flow and this is the capacity. So, our flow must be capacity of that direction. So, our flow must be less than that capacity.

This is sort of current passing on this wire. Now let us give some. So, on the other edges. So, this is say 2 this is say 1, this is say also 2 this is say 1, and this is 2 and this is 1. So, this is satisfying first condition. Everywhere if you check every node suppose this is  $u$ ,

now if this is u and this is v. So, every node the capacity the flow positive flow is less than the capacity. So, that first condition is satisfying.

Then the second condition is if you have to take the current consumption. So, that flow consumption must be 0 for any intermediate node which is not s and t suppose for example, this node. So, this node must what is the flow consumption? So, this node this is the flow coming. So, 2 2 is coming flow and then 2 and then this is another flow is coming 3. So, basically. So, this is u So, what are the flow coming. So, 2 this is 1. So, total flow is consume so 3.

Now, any other node is incoming node a incoming vertex h node. And the going is basically this one and this one and this one. So, this is 0 this is 1 this is 2. So, this sum is 3 this sum is 3. So, no current consumption. So, kirchhoff's law satisfy that is second condition is satisfied. So, this we can verify from any other node suppose this node, for this node suppose this is our u for this node what you have how many are incoming? So, this is u incoming is this 2 this 2, and another 2 this 1 and no more. So, 4 total 4 and what is the outgoing? Outgoing is this one 1 and this one 1 and this one 2. So, 4 this is 4 this is 4.

So, incoming and outgoing flow positive I mean flow is said. So, this is said we can verify for other vertices also. So, this is a positive flow now we want to define the value of this flow, what is the value of a flow?

(Refer Slide Time: 14:51)

So, value of a positive flow is the net flow which is outgoing from  $s$ . So, this value basically is the net flow. So that means, this is basically all the outgoing edges from  $s$ . So, this is basically  $p(s,v)$ -(if there are any incoming edge from  $s$ )-(summation of  $p(v,s)$ ).

So now, for example. So, what is the net flow? What is the value of the flow? So, these are the flow going out this one and this one. And this is the flow coming in, but this flow value is 0 and this is 1 this is 2 3. So, value of this flow is 3. So, the value of this flow is basically  $1 + 2 - 0 = 3$ . So, this is the value of the flow. Now is this the maximum flow? And this is the value of the flow. So, what is the flow at  $t$ ?

So, here So, this is 2 this is 3. So, we have flow at  $t$  is 1. So, whatever current we have pass here that is we are consume here. So that means, and in between they are basically the intermediate node they are just passing the current not holding any current in. So, now we want to define what is the max flow. Is this the maximum current we can flow in this network? That is the question. Is the 3 is maximum positive flow we can pass into this network?

So, that is called max flow problem. So, that problem is referred as max flow problem.

(Refer Slide Time: 18:01)

So, we just defined that maximum flow problem. So, we have given a flow network  $G$ . So, this is the definition given a flow network this graph  $G$  v comma  $e$  and the capacity vector. Find the flow of maximum value. So, flow of maximum value. So, that is called

max flow, this problem is called max flow. So, we are given this for example, we have given this graph and we have a flow this is the flow  $p$ . So now, is this a max flow. Can you do more than this? Can you pass more current than this that is the question.

So, can you try to pass more current this, I mean more value on the flow. So, let us see. So, this is the flow value of this network is I mean positive flow is 3. So, can you pass it 4 like let us see. So, here we cannot pass more than 2, but here we can pass 2 or 3. So, let us try with 2. So, if you pass 2 over here, then this is this may create problem because this is consuming, how much? 2 3 2 3, now it is now it is passing. So, if you just make it 0. So, you make it 0 and then So, this is basically now we can pass it 3.

So, this consumption is basically 2 and here 3 yeah. So, 2 and here 3 we can pass this 3 because capacity is allowing us to pass 3. And then what we do? So, we have one we have one over here. So, you can instead of 1 we can pass 2 2 we can pass because capacity is 2 that is it. So, this is the way we just pass this current this is. So, this is the way we can pass 4. Now is this the maximum can we pass 5? Because this is 3 this is 2. So, this capacity is outgoing capacity is 5.

So now the question is can you pass 5? So, that we have to check. So, that may not be possible. So, we can check that is not possible, I think 4 is the maximum flow value. So, 4 is the maximum value. So we will see how we can get that.

(Refer Slide Time: 21:29)

So, the value of the maximum flow is 4. We cannot get 5 we I suggest you to verify that. So now, how we can get this maximum flow? So, for that we need to do some theorem and some more definitions.

So, first step we are going to do what is called flow cancellation. So, what is that? Without loss of generating the positive flow goes either from  $u$  to  $v$  or  $v$  to  $u$ , positive flow goes either from  $u$  to  $v$  or from  $v$  to  $u$ . So that means, if you have  $u$   $v$  like this. So suppose this is the example this is  $u$  and say this is  $v$ . So, here both the direction positive flow are going because from  $u$  to  $v$  we have a positive flow 2 and from  $v$  to  $u$  you have a positive flow 1.

So, this we want to do the some cancelation. So, what we do? We make it 0 we make it 1. So that means So, net flow this is sort of net flow. So, net flow is remain same it is basically 1. Because 2 was going one was coming. So, we can make it 0. So, positive flow without loss of generative we can assume the positive flow is going only to the one direction. Because if you have a flow in 2 direction we can cancel one I mean which is the lower one. So, we can make it 0 and we can reduce the flow in other one. So, that is what we did.

So, this what we did. So, this was 2 this was 1. So, we convert into this  $v$   $u$ . So, we make this 1 and we make it 0. So, this is referred as flow cancellation. So, you have canceling flow; that means, we are allowing the positive flow from only one direction not for the both direction. So, net flow from  $u$  to  $v$  is in case 1, because earlier also net flow is one now also net flow is 1. So now, we want to do some notation simplification of this. So, we want to defined the net flow. So, we want to defined the net flow.

(Refer Slide Time: 25:53)

So, this is one notational simplification. So, in set of positive flow we are just now interested in the net flow. So, the idea is because positive flow we have both the direction. Now without loss of generative we see that we can cancel the one direction flow. And then that direction on direction is 0 another. So, that is the net flow from u to v. And from v to u that minus of that. So, idea is work with the net flow between 2 vertices rather than the positive flow. Rather with the positive flow, because we can cancel the one direction flow because net flow ultimately we are looking at how many flow we are passing.

So, that way this net flow with this is some sort of simplifying we are our things. So, we have a positive flow we will check this 2 definition are same. So, we have an defined it in the next class we will define this what is the net flow. And then we will see how this 2 definitions are basically equivalent, if you have given the positive flow, we can have a net flow if you have a net flow we can defined the positive flow so that we will discuss in the next class.

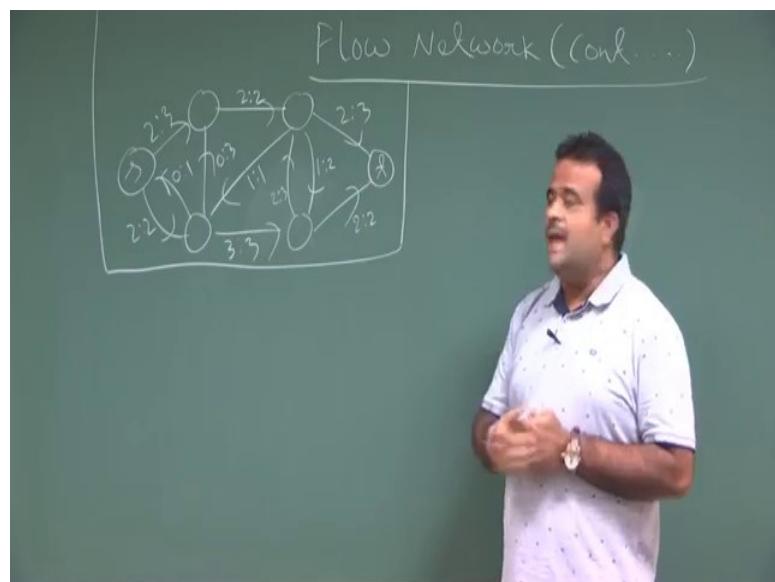
Thank you.

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

**Lecture – 55**  
**Network Flow (Contd.)**

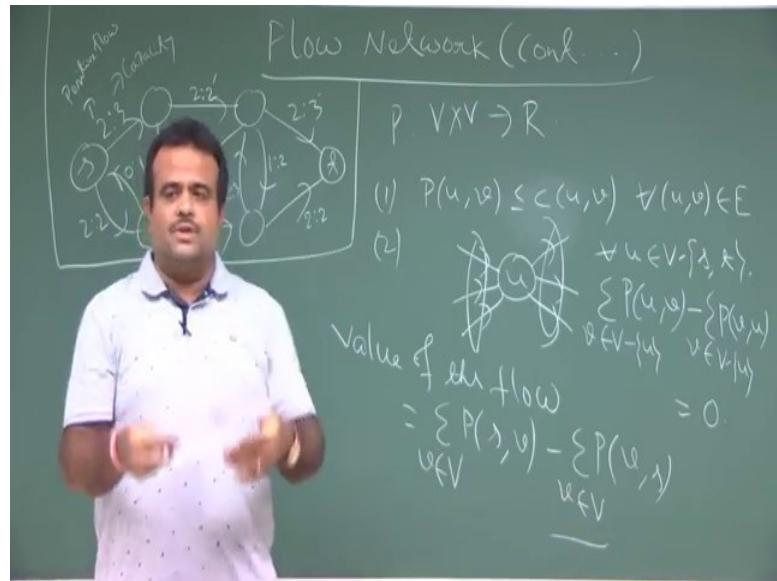
So we are talking about network flow. So, just to recap in the last class we have started this topic.

(Refer Slide Time: 00:27)



So, a network graph is a graph you say directed graph. So, where we have a source node  $s$  and we have a sink node  $t$ . And each node each edges are associated with some capacity this is the capacity. So, this is the capacity basically of each edge. So, if there is no edge between any 2 vertices then the capacity will be 0. So, then this is the flow now this is the positive flow. So, in the last class we defined the positive flow it is a we want to flow some current from source to sink so that in the intermediate vertex there will be no consumption of the current. So that means, it is a function form it is a positive flow.

(Refer Slide Time: 01:24)



It is a function from  $v \times v$  to  $r$  such that it satisfies few conditions like first condition is the flow positive flow should not be greater than the capacity.

So,  $p(u,v)$  must be less than or equal to  $c$  of  $u v$  for all  $u v$  belongs to  $E$ . And we have seen the flow conservation; that means, if we take a vertex  $u$ . So, the flow coming into this vertex and flow going out from this vertex this sum should be same. So that means, summation of  $p u v$ . So, this is flow going out  $v$  belongs to  $v$  minus  $u$  minus summation of  $p v u$  belongs to  $v$  this should be 0. So that means, no flow consumption should be there. So, this is called this is one of the properties this is for flow conservation. Then the value of the flow we defined.

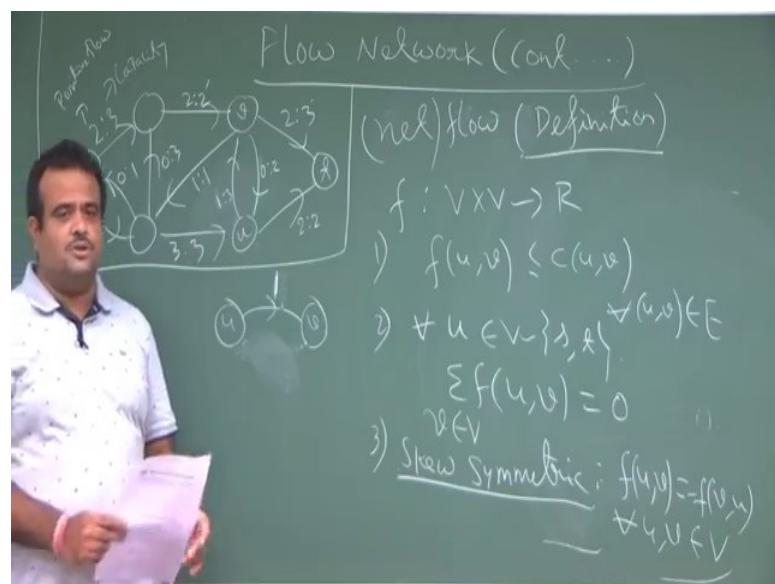
So, the value of the flow is basically, the flow which is going out from the source. And this flow conservation property is true for all  $u$  which is not source and sink, but for source and sink we have passing the current from the source and sink is. So, this is being the current sort of. So, this is the value of the flow basically the effective flow; effective current we are passing from  $s$ . So, that is basically summation of  $p(s,v)$  this is for all  $v$  belongs to  $v$  minus summation of  $p(v,s)$ . This is the value of the flow. So, same amount of flow is going to the sink also.

So now we have seen another concept which is called flow cancellation. So, that is that is like if you have a vertex say from this 2 vertex  $u$  to  $v$ . So, effective flow is this is called net flow. So, effective flow is we can make it 0 we can make it 1. So, this is the effective

flow from u to v. So, this is basically flow cancellation. So, this is basically without loss of generality we can assume that flow is going to only one direction. So, either from u to v. So, from v to u. So, if there is 2 direction we will just cancelling the flow and we will take the net flow. So, that is the concept.

So, let us define the net flow.

(Refer Slide Time: 05:00)



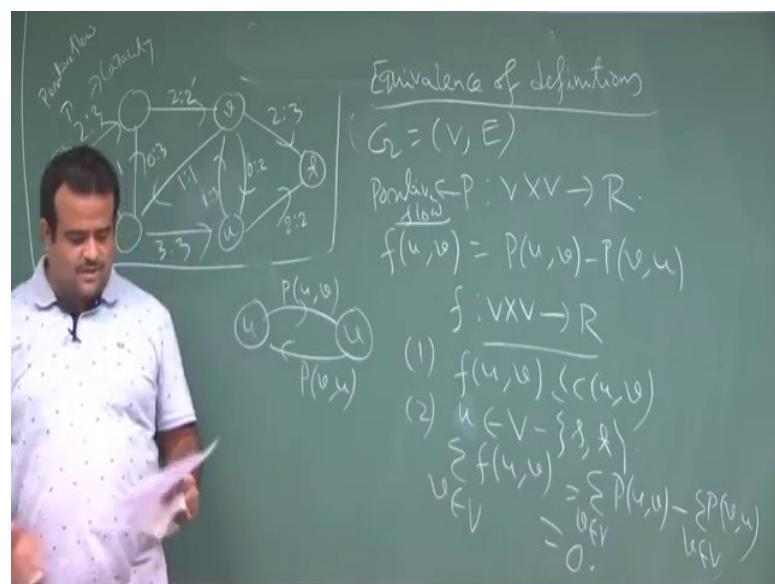
So, this is the definition of net flow. So, basically net flow is a function  $f$  from  $V \times V$  to  $\mathbb{R}$  such that it satisfies 3 conditions again the same capacity constraint say for  $u, v$ . So, it should be less than  $c(u, v)$  for all  $u, v$  belongs to  $E$ . So, net flow should be less than the capacity and flow consumption is 0 at every point. Because every point has effective flow is 0. So, that we have to write. So, for all  $u$  which is basically coming from this said we have summation of  $f(u, v)$  is basically 0. This  $v$  is belongs to this.

So, here we have one sum in the positive flow we are having 2 sum, but here we have one sum because we are just considering the net flow we are considering we are doing the flow cancellation if there is both side flow. So, this is the flow conservation property and another property is called skew symmetry, this is the new one skew symmetric property. This is telling so, suppose there are 2 vertices  $u$  to  $v$ . So, if there is a flow from  $u$  to  $v$  then there is a negative flow from  $v$  to  $u$ , this is the skew symmetric property.

So that means,  $f(u,v)$  is equal to minus of  $f(v,u)$  for all  $u, v$  belongs to  $V$ . So, if there is a flow then from reverse direction it is minus of that. Now this is the definition of the net flow and you already have a definition of the positive flow.

Now, we will see these two definitions are equivalent. So, how to get that? So, we want to see this 2 definition are equivalent positive flow and the negative flow. If you have given the positive flow we can basically have a net flow if you have a net flow we can have a positive flow.

(Refer Slide Time: 08:30)



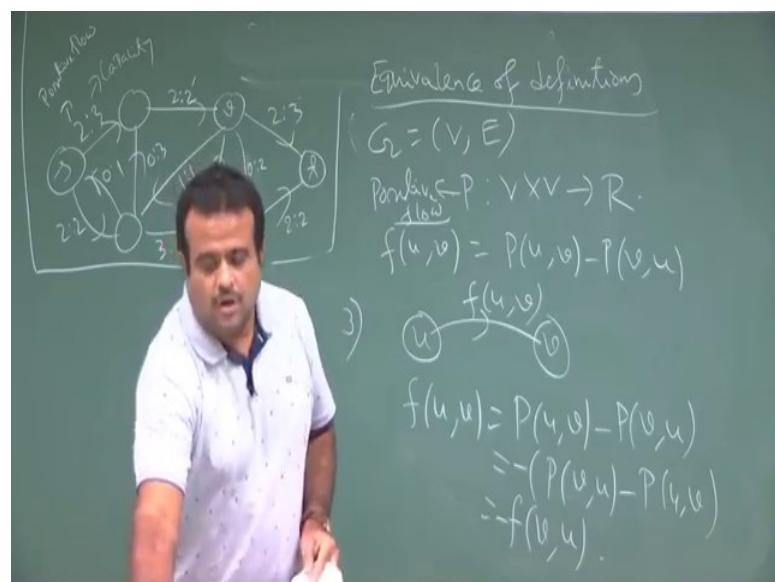
So, this is called equivalence of 2 definitions one is net flow another one is positive flow. So, first one suppose we have a flow graph. Now suppose we have a positive flow  $p$ . Let suppose you have a positive flow on this graph. Now you have to have a net flow. So, how to define the net flow? So, we just defined  $f(u,v) = p(u,v) - p(v,u)$ . So that means, if there is 2 vertex  $u$  to  $v$ . So, this is basically  $p(u,v)$ . Now the net flow is we defined this net flow by this way.

So, this is the function form  $v$  plus  $v$  to  $I$ . Now we can easily verify this is a net flow because we can see this is satisfying 3 conditions like capacity constant. So, this is basically  $f(u,v)$  is less than  $c(u,v)$ , because this is anyway less than the capacity and we have subtracting some quantity if which is this is greater than equal to 0. So, this is true and another property is the flow conservation property So that means, if you take a

vertex  $v$  from this which is not source are saying then we want to calculate summation of  $(u,v)$ ,  $v$  is basically summation of this is say  $(u,v)$ .

So, for it to be 0 this is nothing but summation of  $p(u,v) - \text{summation of } p(v,u)$ . So, this is all  $u$  and this is all  $v$ . Now we know this is 0, from the definition. This is the positive flow we have given a positive flow function. Now we want see the skew symmetric.

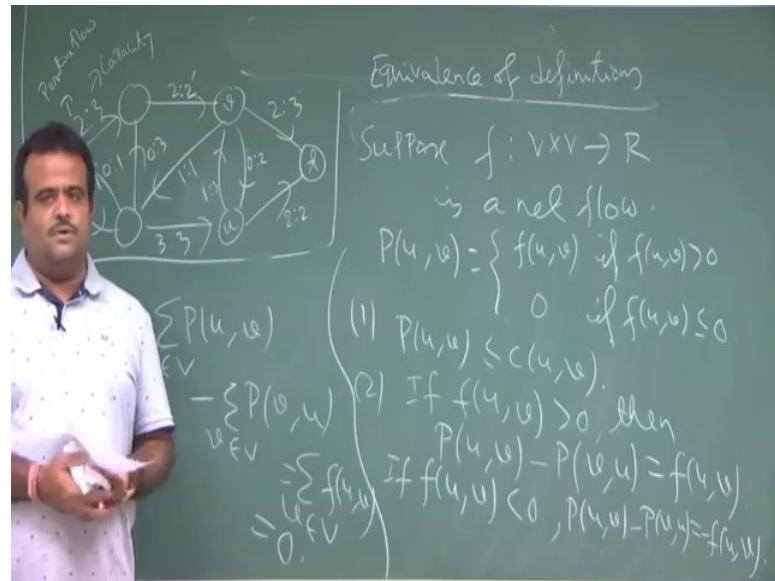
(Refer Slide Time: 11:46)



So, we have  $u$  and we have  $v$ . So, this is the third property.

So, what is the flow value from  $u$  to  $v$ ? So, this is basically  $f(u,v) = p(u,v) - p(v,u)$ . So, this we can take  $-(p(u,v) - p(v,u))$ . So, by this definition this is nothing but  $-f(v, u)$ . So, this is the skew symmetric property. So, these 3 properties are satisfying. So, this is a so, this is nothing but a net flow. So now, you want to so if we have given a net flow how we can get a positive flow. So, that is the other way around.

(Refer Slide Time: 12:58)



Suppose we have given the net flow say, then suppose if this is a net flow function. So, this is a net flow now if this is a net flow. Now the question is how we can have a positive flow function. So, we define like this  $p(u, v)$  is because positive flow is always positive. So, we defined this  $f(u, v)$  if it is 0 like sorry, if it is positive otherwise it is 0 if  $f(u, v)$  is less than 0. So that means so, if we have 2 vertex  $u$   $v$  the if there is a flow  $f$ , now if  $f$  is positive then that is our  $p(u, v)$ . But if  $f$  is negative then we do not have then  $p$  of  $v$  is 0 basically.

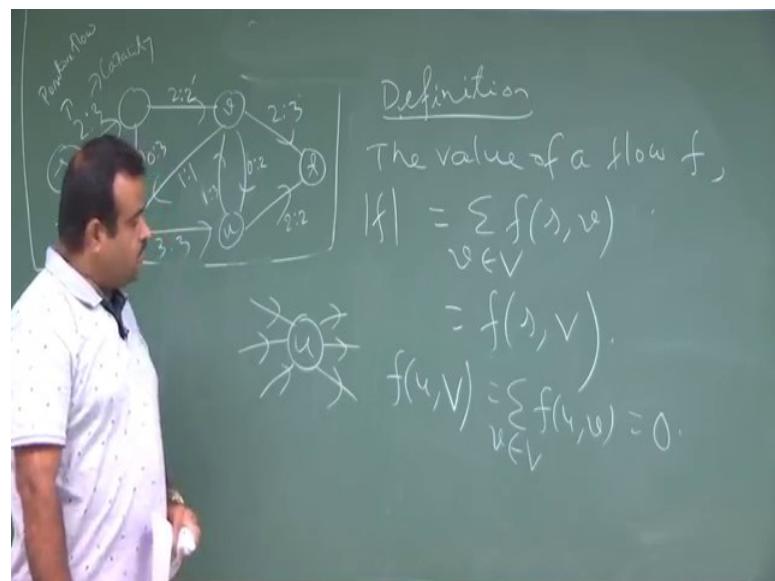
So, that is the way we defined this. Now the question is how to prove this is a positive flow? So, for positive flow we have taking so, the condition like capacity constant. So, this is trivial because  $p(u, v)$  is either 0 or this, and both are basically less than  $c$  of  $u$   $v$ . Because capacity if there is no ways then the capacity is 0 no edge between a 2 vertex  $u$   $v$ . Otherwise it is the positive. And the second one is basically we need to show flow conservation. So, for flow conservation what we need to do? So, we did so, if this is greater than 0, then we know  $p(u, v)$  minus  $p(v, u)$  this is basically  $f(u, v)$ . Because in that case this is 0 this is 0 because this is positive means, this is  $p(u, v)$  is this 1 and minus of that is negative so, that is 0. So, this is ok.

So now if this is less than if  $p(u, v)$  is less than 0, then we have that is minus, we have  $p(u, v)$  minus  $p(v, u)$  is basically minus of this. So, this is basically by the skew symmetric property. Now, therefore if you take the sum like summation of  $p(u, v)$  this  $v$  is wearing

from  $v$  minus summation of  $p(v,u)$ . So, this is nothing but summation of  $f(u,v)$ . And this we know will be 0, because net flow consumption at any point is 0. This we have already this coming from the definition of net flow.

So, these 2 definitions are basically equivalent. So, given a positive flow we can have a net flow given a net flow you can always think of a positive flow. So now we will deal with the net flow because it is the simplification in set of both the direction we have the net flow; effective flow. So, that is the basically simplification in the notation. So, we will deal with the net flow. So now, suppose we have a net flow, then how we will define the value of the flow? So, this is another definition.

(Refer Slide Time: 17:23)



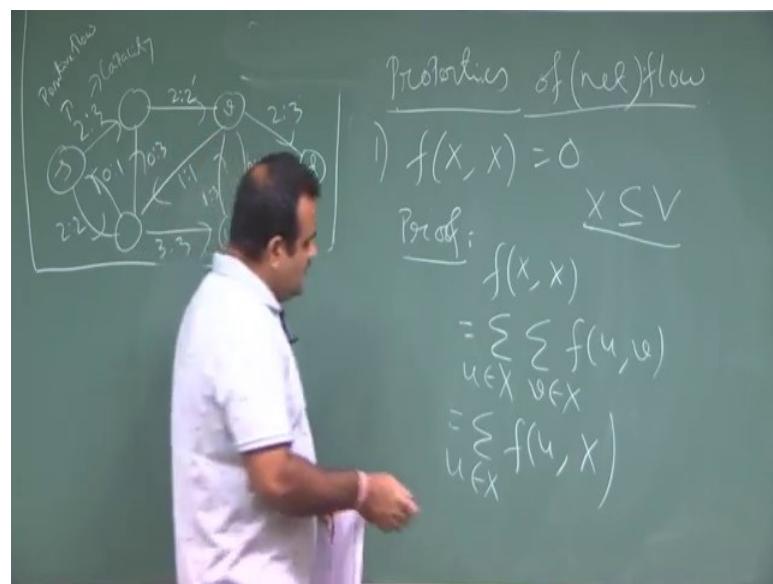
The value of a flow  $f$  is denoted by and it is basically given by summation of  $f(s,v)$ . And this  $v$  is coming from  $V$ .

So, this is basically the net flow. So, we have a source vertex  $s$ . So, you have other vertices over here. So, net flow we are just talking about net flow. We are not talking about positive flow, in that case there will be a flow from this side. So, we are just considering the one side flow. That is the effective flow or the net flow. So, that way. So, we just sum. So, these are the net flow. So, we sum all of this net flow. And that will give us basically summation of  $f(s,v)$  this. So, this is the flow we pass from the source to the sink. So, that is the thing. So now, this is the set theory notation  $f(s,v)$ .

So, if you take a vertex  $v$ . So, the flow consumption is basically on this vertex is 0. So, how we different that? So, that is basically in terms of net flow  $f(u,v)$  is 0. So, this is basically in said notation this is basically proper. So, this is nothing but summation of  $s(u,v)$ . So, this is in said notation. So, this is 0. So, this is basically the flow consumption at any node is 0.

Now, we will have some properties on this net flow.

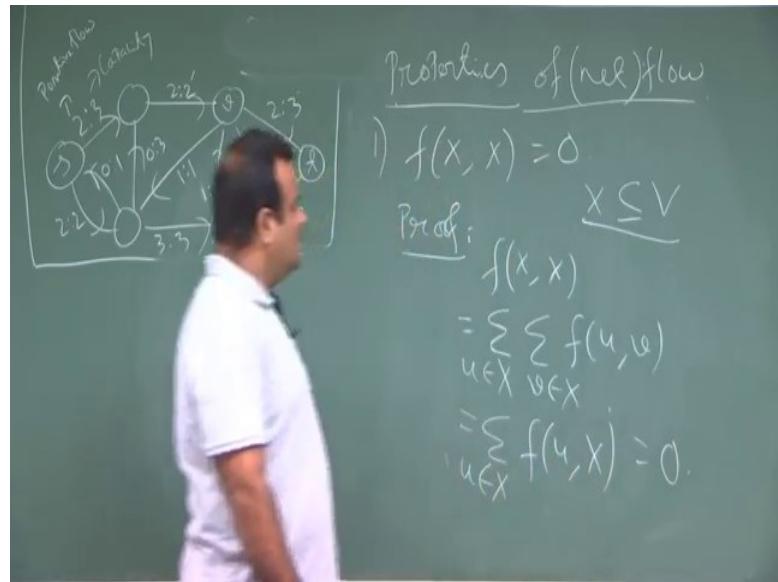
(Refer Slide Time: 20:04)



Properties of flow specially in net flow we have taking about now our flow is net flow. So, given  $f$  we want to talk about some properties of  $f$ . The first lemma first property of  $f$  of  $x$ ,  $x$  is 0, where  $x$  is a said. So, we are we have dealing with the said. So, how to show this? How to prove this? So, how to prove this?  $S$  of  $s$ ,  $s$  is 0. So, to prove this suppose what is  $s$  of  $s$ ,  $s$ ? So, this is basically double sum. So, this is  $s$  is a subset of  $v$ . So,  $s$  is a subset of  $v$  it contain some of the vertices,  $s$  is subset of  $v$ .

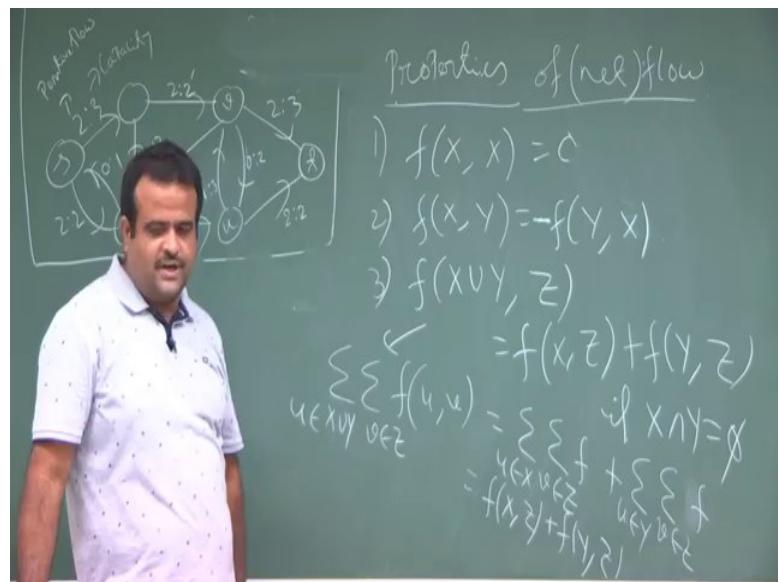
So, basically this is nothing but summation of double summation of  $f(u,v)$ . Now  $u$  is belongs to  $x$  and  $v$  is belongs to  $x$ , both are belongs to  $x$ . So now, this we can write as summation of  $f(u,x)$ . So, basically so this is basically the net flow which are going from  $s$  to  $s$ . So, this is nothing but 0 because you know the net flow of any vertices is 0. So, sorry this is basically coming from double summation.

(Refer Slide Time: 22:04)



So, this is basically summation of  $u \in X$   $f(u, x)$ . So now this is all the flow from  $x$  to that vertex. So, this is net flow will be same as the all the flow which are out going from this vertex. So, this is basically is 0. So, net flow is basically 0.

(Refer Slide Time: 22:46)



So, this is basically another property  $f(x, y)$ . So, this is this is basically coming from skew symmetric property of the net flow.

Then the another property is basically this is minus.  $F$  of  $(x, y, z)$  is nothing but so, this 2 are disjoint sets  $(x, z)$  plus  $f$  of  $(y, z)$  where  $x, y$  are disjoint set. So, this is the property we have. So, here  $x, y$  at disjoint set. So, how to prove this third property? Because see

this is  $x$  is  $x$  or  $y$  all subset of  $v$  basically. So, how to prove this? To prove is, if we have a disjoint subset. So, we just take the sum. So, this is nothing but double summation of  $f(u,v)$  where  $u$  belongs to and  $V$  belongs to  $Z$  ok.

Now,  $u$  belongs to this means it is basically disjoint set. So, it is basically we can write this sum as summation of  $u$  belongs to  $X$   $v$  belongs to  $Z$ , plus summation of  $u$  belongs to  $Y$   $v$  belongs to  $Z$ . So, this is nothing but  $f(x,z) + f(y,z)$ . And this is true because so, this is true because they are they are disjoint set. So now, we will use this result or this theorem or this lemma to prove another theorem which is basically telling us the flow; the net flow of a network. So, that we'll discuss. We'll start from here in the next class.

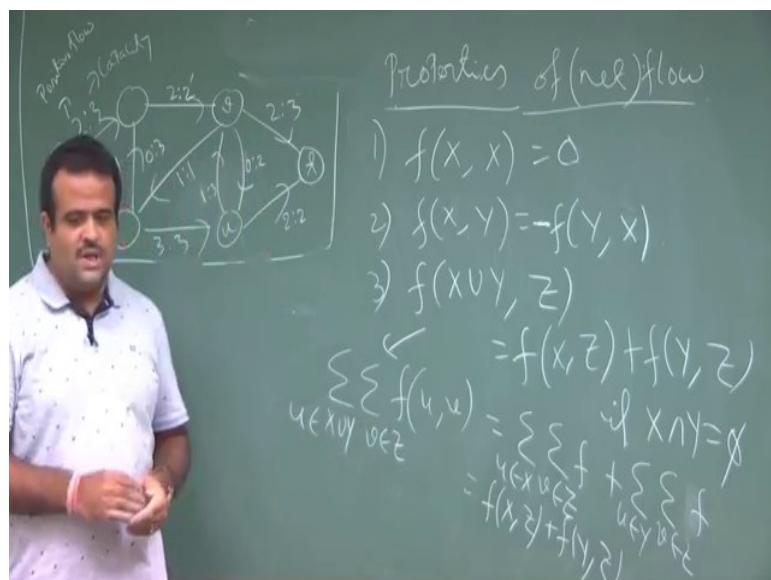
Thank you.

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

**Lecture - 56**  
**Network Flow (Contd.)**

So, in the last class we have seen this, properties of a net flow.

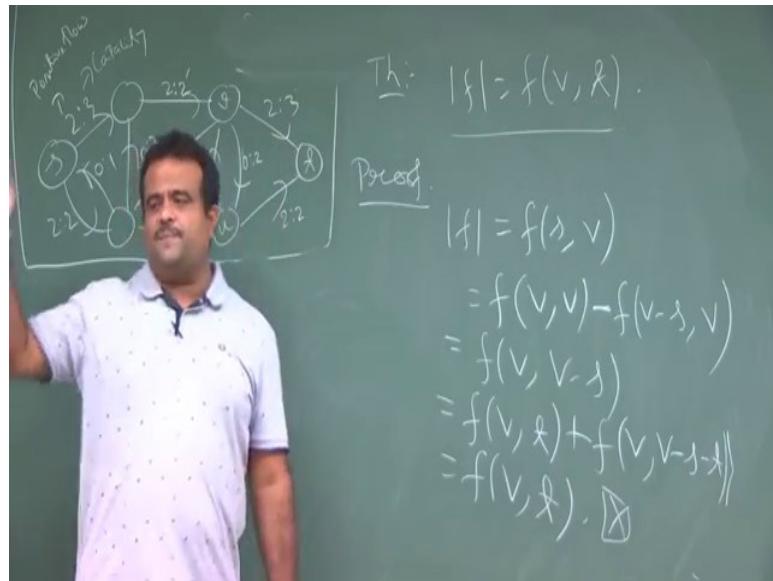
(Refer Slide Time: 00:23)



If we have a net flow then  $f(x, x)$  is 0, then  $f(x, y) = -f(y, x)$  and then if you have a 2 disjoint set, if of  $x$  union  $y$  then this is basically sum of this. So, this we have proved in the last class.

Now, we just want to use this to have the theorem which is related to the value of the flow. So, this is basically the value of the flow.

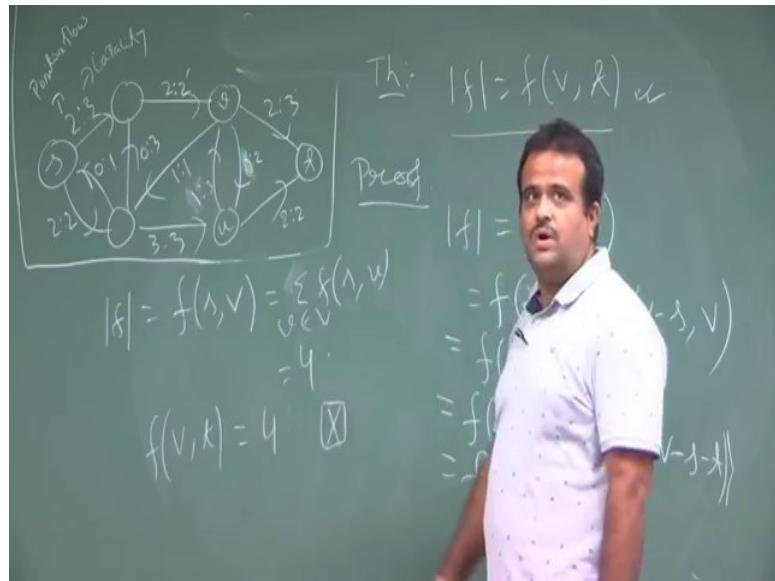
(Refer Slide Time: 01:05)



So, value of the flow is basically  $f(v,t)$ . So, it is the total flow which is receiving at the sink point. So, this we have to prove. Now we know what is the value. So, how to prove this. So, we know the value of the flow is basically the flow which is passing out from the source. So, we have the source here. So, this is basically the flow which is passing out from the source and this is the net flow. So, this is basically summation  $f(s,v)$ . So, this is the definition of the value of a flow because here  $f$  is net flow not the positive flow.

If you have taken positive flow then minus of that, but here you are dealing with the simplified notation which is net flow. So, then how we do that. So, this we can write  $f(v,v)$  because  $f(v,v)$  is 0, just now we have seen then. So, this is coming from these properties we have discussed. So, this is nothing, but  $f(v,v)-s$ . So, now, this we can write as  $f(v,t)+f(v,v)-s-t$ . So, this is the way we define this i mean we are using the last 3 lemma we have discussed. So, this is nothing, but  $f$  from  $v,t$ . So, this is the proof so; that means, whatever the flow we are passing at the sink source that will be absorbed in the sink. So, that is the property you want, and that is quite obvious because we want to say this is the network we want to pass the current.

(Refer Slide Time: 03:46)



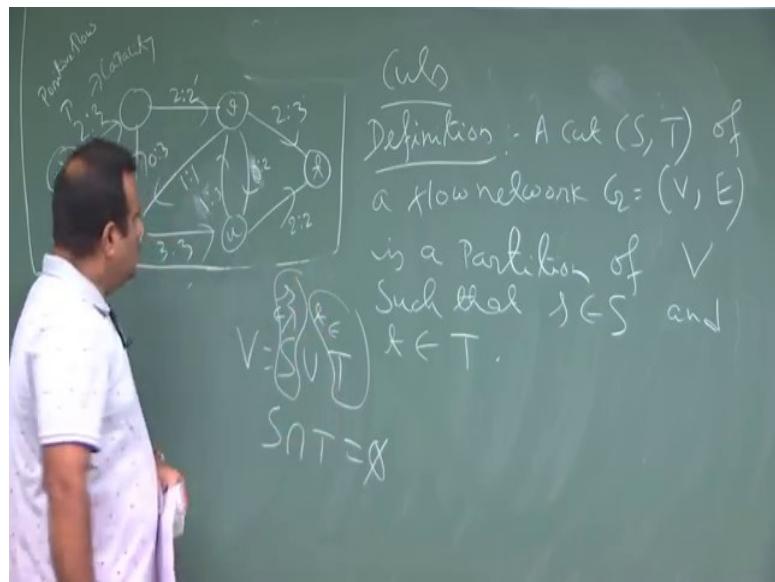
So, we want that whatever the current we want to pass from here net current that should be absorbed here and that is the value of the flow and this is the theorem is telling that will be absorbed there i mean that will be passing at that end c ok.

Now, if we take this example. So, what is the net flow what is the value of the flow. So, value of the flow for this network. So, this is a network we have these are the basically capacity and these are basically positive flow in the example we have discussed in the previous classes we discussed these are the positive flow now what is the net flow? Net flow is basically after we cancelling the flow cancellation if. So, we assume the without any loss of generality, the flow can the flow will be only one direction from u to v not from v to u. If there is a flow say earlier it was 2 it was earlier the positive flow was it was 2 it was 1. So, what we did once instead of flow in both direction we just want to flow in one direction and we are just talking about net flow. So, net flow is one because this was cancelling out. So, this is the flow cancellation we did. So, effectively we have this net flow.

So, then what is the value of the flow. So, value of the flow is net flow you are passing from this. So, this is basically  $2 + 2$  it is basically 4. So, this is basically summation of  $f(s, v)$ . So, this is basically 4. So, this is the value of the flow and now we can check what is the flow at the sink flow at the sink is basically  $2 + 2$ . So, 4 this is the example of this theorem. So, whatever flow we are passing from the source that will be going to the sink. So, this is that way now we defined another concept which is called cut flow cut I mean. So, cut. So, what is

the cut?

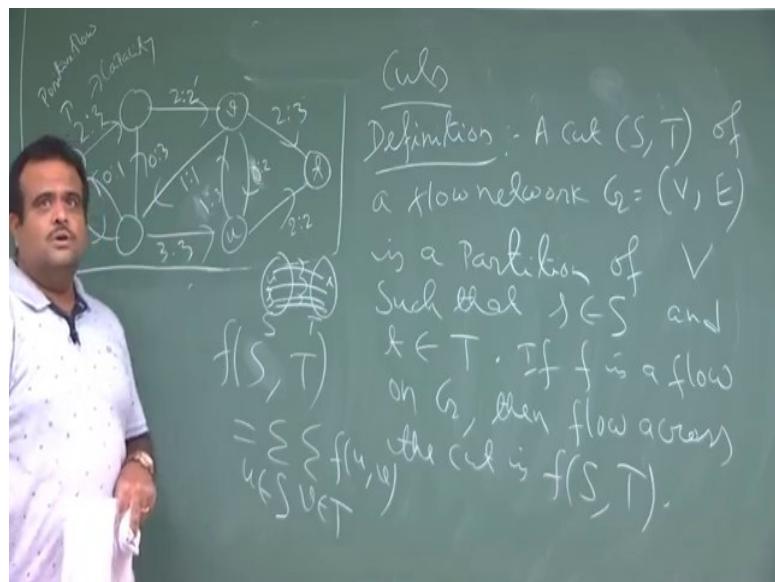
(Refer Slide Time: 06:27)



Let us just define the cuts. So, this is definition. So, a cut  $S$   $T$  this is to set  $S$  and  $T$  capital  $S$  and  $T$  of a flow network  $G$ , which is basically  $V, E$  is a partition of  $V$  such that the source belongs to  $S$  and the sink belongs to  $T$ .

So, any such partition basically we have we have a graph  $g$  and we are taking a partition; that means, we are dividing the vertices are there  $V$ . So, this is the  $V$  vertex. So,  $V$  is basically  $S$  union  $T$  and this is the disjoint set this is the disjoint partition and such that  $s$  will be here and  $t$  will belongs to here. So, this is the property. So, we take a set capital  $S$  from using this I mean which containing the source  $S$  and we take another set capital  $T$  which contain the sink  $T$ .

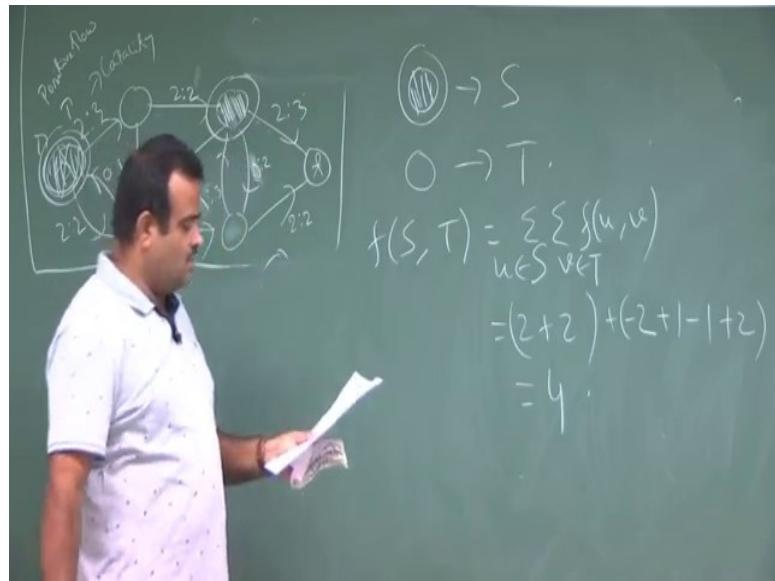
(Refer Slide Time: 08:37)



So, any such partition is called a cut. So, for example,. So, this is a cut now how we defined flow across the cut. So, suppose we have a cut here, now if  $f$  is a flow. Flow means we now means the net flow always because we simplify the notation flow on  $G$  then flow across the cut. So, flow across the cut is basically  $f(S, T)$  this is how we defined the flow across the cut. So, the total flow across the cut so; that means, basically. So,  $f(S, T)$  is basically nothing, but summation of double sum this of  $u, v$  where  $u$  is belongs to  $S$  set and  $v$  is belongs to  $T$  set.

So, basically we are partitioning  $v$  total vertex is  $v$  we will be partitioning into 2 parts  $S$  set  $T$  set. Now there are  $h$  form  $u$  to  $v$ ; one part is in  $s$ , another part is in  $t$  we know  $s$  small  $s$  is here we know cap small  $t$  is here. So, now, this is basically sum of the flow from one set to another set where the these are called bridges like one vertex is in  $s$  capital  $S$  another vertex is in capital  $T$ . So, we take the sum of all such flow and that will give us the total flow that will give us the flow across the cut that is how we define this. So, let us take some example over here for this network say. So, for this network let us take some example.

(Refer Slide Time: 10:57)



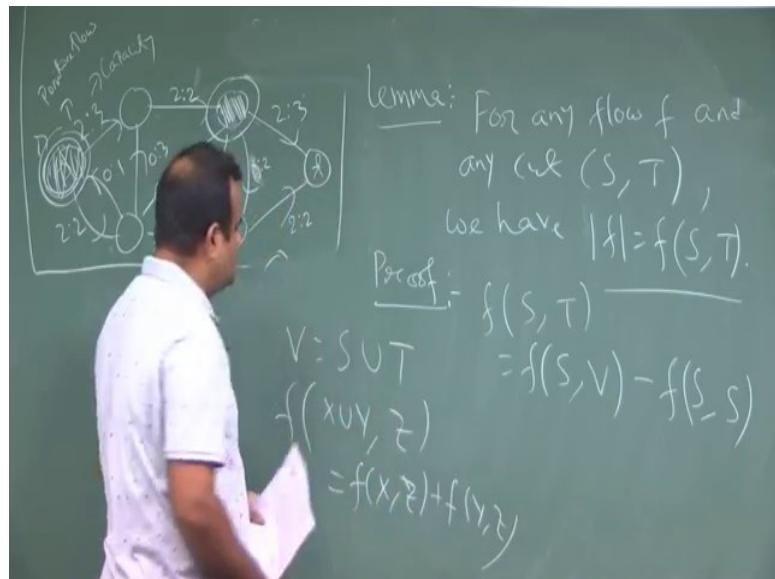
Suppose this is our source. So, basically these 2 vertices are in  $S$  say, these are in  $S$  and remaining all are in  $T$ . We can take any cut this is one cut, only thing we need to consider that this is a partition and the source must be in capital  $S$  and the sink must be in capital  $T$ . So, that is the only thing we need to take care.

Now, what is the  $f(S, T)$  then? So,  $f(S, T)$  is basically summation of all the vertices all the flow between the edges such that one edge in  $S$  and another  $S$  must be in  $T$ . So, what are the edges basically who are in. So, this is one edge this is one edge and this is one way. So, this is basically minus 2 will come. So, what is the flow from here to here? So, this is basically flow is 1. So, the flow from here to here is -1, because this is the skew symmetric property of the flow and there we have a flow from here to here. So, any other vertices are in capital  $T$ . So, at here to here to here to here it is it is basically one. So, now, we consider all the vertex form this to this.

So, this is nothing, but  $2 + 2$  for this 2 what I for this these and this and then from for we take this node from this node this is the  $2 + 2$  and this is another edge, but this minus of that is 0. So, that is no contribution now we take another vertex from  $S$  capital  $S$ . So, this vertex for these vertex. So, we have this is the flow from this to this is basically minus 2, then then we have flow from this to this is + 1, flow from this to this is minus 1 and then we have flow from this to this is + 2. So, this is basically coming out to be 4. So, that is that we know the value of the flow. So, this is the result. So, if you take any cut and it will basically give us

the value of the flow, if you take any cut. So, that is the theorem we are going to show. So, this is the observation. So, not only these 2, we can take any vertices and it will work.

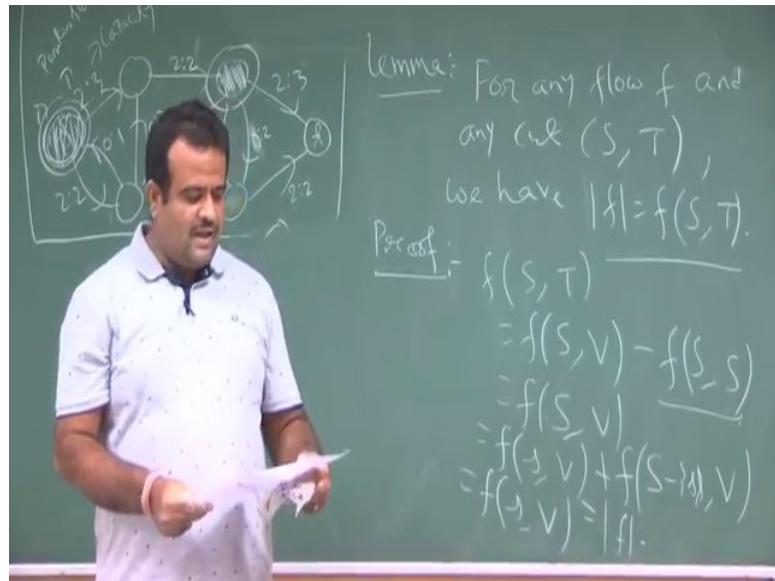
(Refer Slide Time: 14:26)



So, this is the lemma. So, for any flow  $f$  and any cut  $S, T$  we have the flow value, value of the flow is basically value the value of the flow across the cut. So, this is basically  $s$  of  $f(S, T)$ .

So, how to prove that, you have seen through this example. So, if you take these 2 vertices in  $S$ , then the value and other in  $T$  then the value of the flow across this cut  $t$  is 4. Not only this we can take any other partition and we can see the value of the cut is always 4 and that is the value of the flow. So, how to prove this? So, basically  $f(S, T)$  is nothing, but we will use that properties  $f(S, V)$  minus  $f(S, S)$  because  $V$  consist of  $S$  union  $T$  and  $S$  union  $T$  are disjoint. So, we know this property  $f(x, z) + f(y, z)$ . So, you are using this, where  $x, y$  are disjoint. So, by using this we can show this. So, this is 0 this we know  $f(x, x)$  is 0.

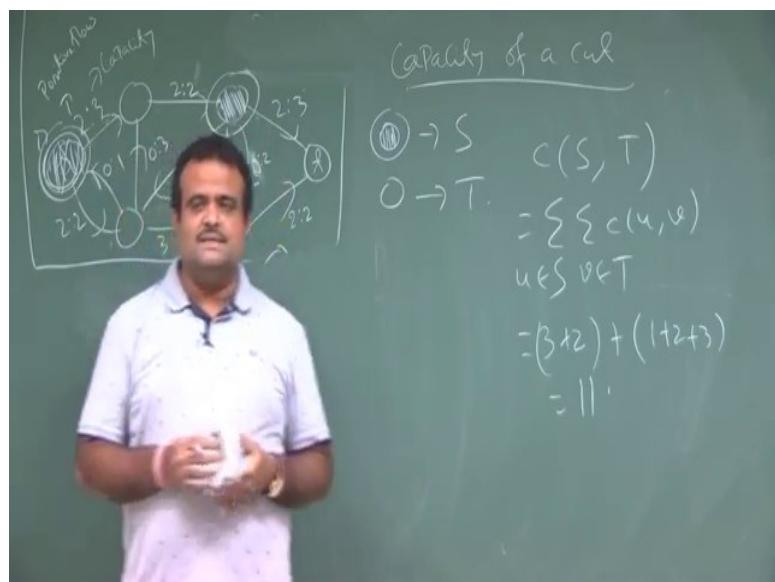
(Refer Slide Time: 16:39)



So, this is basically  $f(S, V)$ , now again we can write as  $f(s, v) + f(S - s, V)$ .

Again that property of disjoint property. So, this is nothing, but  $f(s, v)$ . So, this is basically the value of the flow. So, if you take any cut then that will give us basically the value of the flow. So, now, we define the capacity of a flow. So, capacity across a flow.

(Refer Slide Time: 17:41)

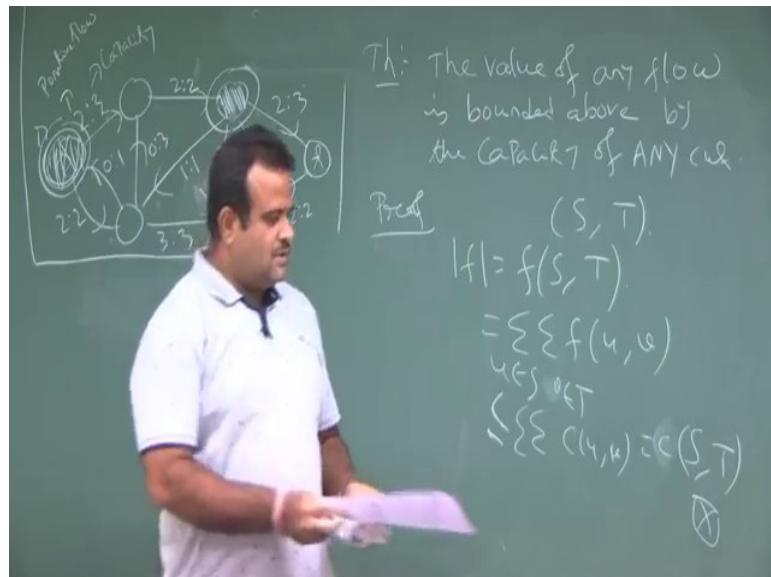


So, how we define that. So, for example, if we take this flow network where these are the vertices in  $S$  and other one  $T$ . So, the capacity of this basically the capacity. So, we add up the capacity of each of the vertices. So, this is basically we denote this by  $c(S, T)$  this is

nothing, but double summation of  $c(u,v)$ , where  $u$  is belongs to  $S$  and  $v$  is belongs to  $T$ . So, this is here. So, if you do this to  $3 + 2$  then + what are the values going  $1 + 2 + 3$ ,  $1 + 2 + 3$ . So, this is basically 11.

So, this is the capacity of this cut. So, now, we are having we want to use the cut for. So, basically what we are looking for we are looking for max flow we have a network we want to flow maximum current. So, max flows for that we are bringing this cut, and we will see that there is a relationship max flow with the capacity of this cut. So, that will give us a algorithm basically. So, mean cut max flow.

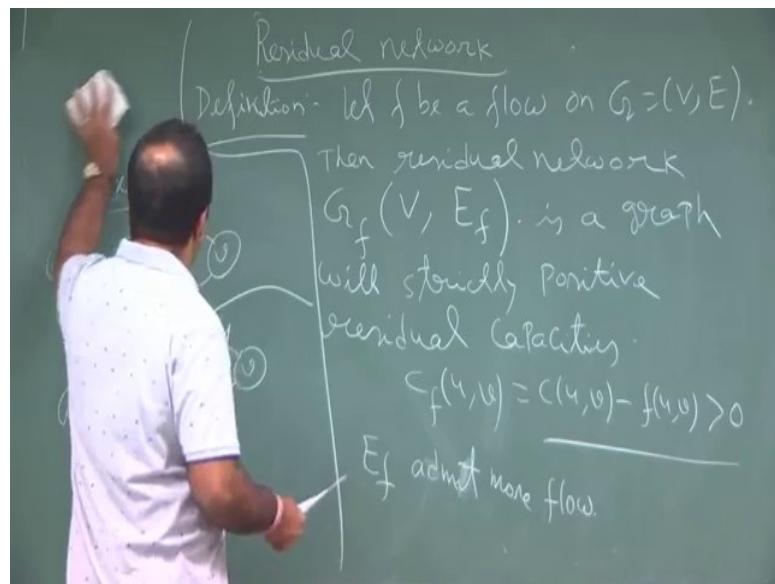
(Refer Slide Time: 19:26)



So, let us just write that theorem. So, this is sort of upper bound of the on the max flow. So, value of any flow is bounded above by the capacity of any cut; any cut. So, this is the upper bound. So, how to prove this? So, let us take a cut over here say  $S, T$  be any cut now we have seen the value of the flow is nothing, but value of the flow across the cut and this we know this is basically summation of  $f(u,v)$   $u$  belongs to  $S$  and this  $v$  belongs to  $T$ .

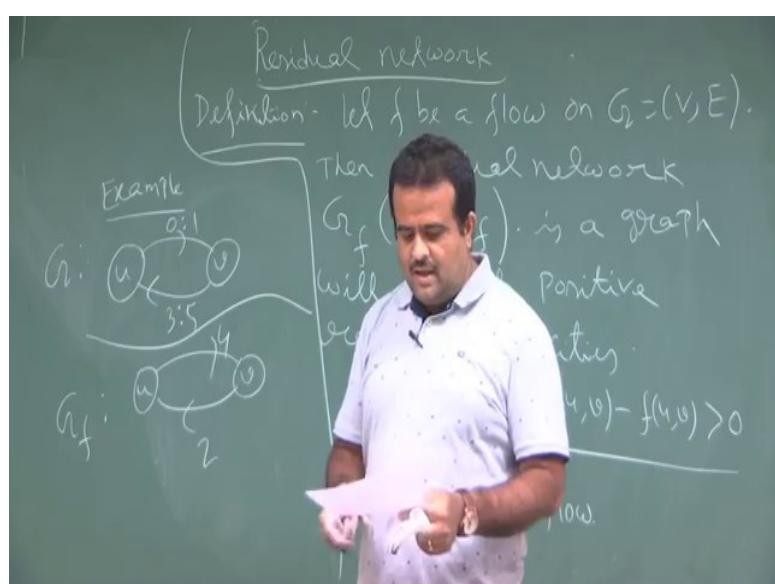
Now, this is the flow. So, this must be less than all the capacity on that  $h$ . So, this is basically double summation of  $C(u,v)$  and this is nothing, but our  $C(S, T)$ . So, this is the proof. So, this is the basically the value of this. So, now, we define another concept which is called residual network and that will give us the one algorithm to find the max flow.

(Refer Slide Time: 21:37)



So, this is the last concept and then we will talk about augmenting path. So, this is the definition the residual network. So, what is the definition of residual network? So, suppose we have given a flow let  $f$  be net flow, on a flow network  $G$  which is basically  $V, E$ . Now we defined a residual network we define the residual network which is another graph which is denoted by  $G_f$  and whose vertices are same, but only thing for the  $h$  we have  $E$  of  $f$ , because this is having different edge weight and we may have edges also different. So, this is the is a graph directed graph with strictly positive residual capacities so; that means,  $f(u v)$  is equal to  $c(u v) - f(u v)$  which is basically greater than 0.

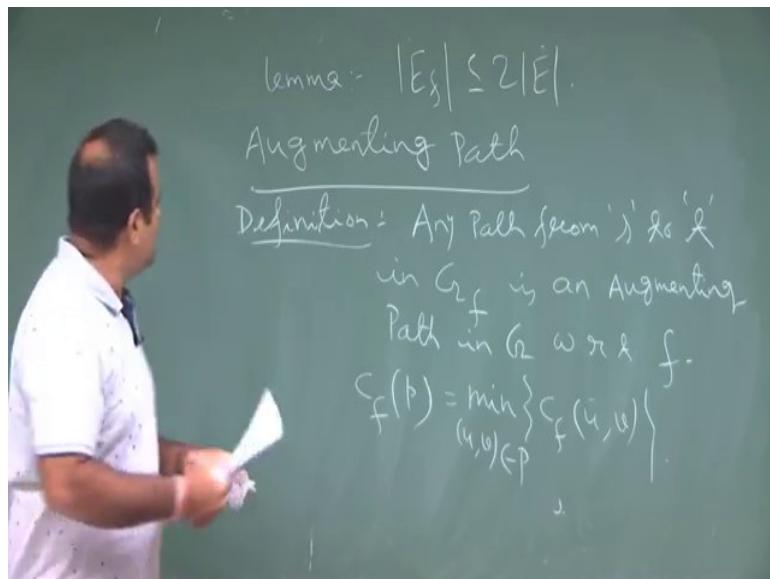
(Refer Slide Time: 23:54)



So, if we have a vertex  $u$ . So, this is our  $G$ . So, suppose there is a  $u v$ . So, suppose this is the situation 0 is to 1. So, this we are dealing with the say positive flow 3 is to 5. So, suppose this is the original graph part of the original graph  $u v$ . Now in the residual graph we have same vertex  $u v$ , but only thing we are changing the edge how it is changing the edge? We want to see how much more current we can send, how much more flow we can flow on that. So, that is the idea. So, how much more current we can pass. So, that is the thing. So, that will become in in  $G_f$  basically this will become like this  $u v$ . So, how much more current we can pass from this to this, this capacity is 5 we have 3. So, we can pass 2 and then from this to this we have my. So, from this to this, this is the net flow. So, this is minus 3 and the capacity is 1, 1 and then minus of minus 3. So, this is 4. So, this is the basically so; that means, 4 unit of current we can flow here and 2 unit of current we can flow.

So; that means, this  $E_f$  is basically admits more flow. So, how much more we can flow? So, how much more current we can pass in there. So, now, we will define. So, the lemma is basically.

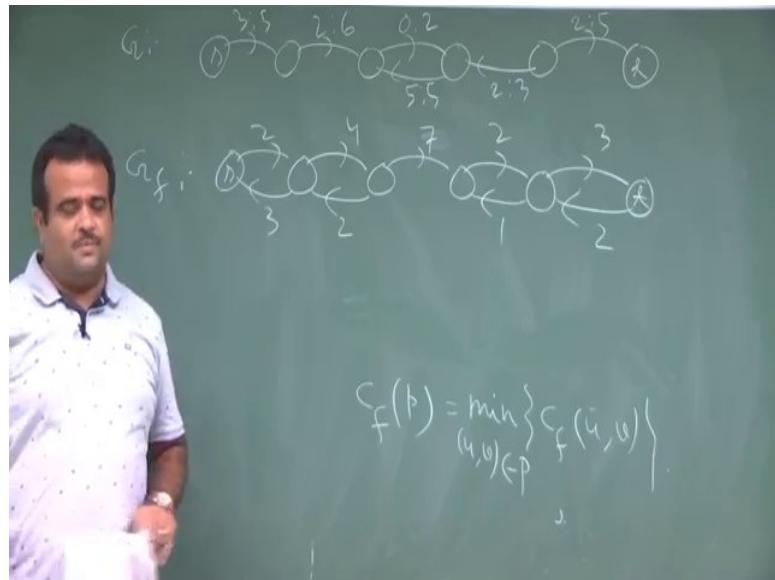
(Refer Slide Time: 26:02)



So, this is a lemma. So, this is basically number of edge must be twice off number of this. So, now, we will define augmenting path. So, this is basically path for the this residual graph. Now how we defined this definition. So, any path; path from  $s$  to  $t$  in  $G_f$  is an augmenting path in  $G$  with respect to the flow with respect to the flow  $f$ . So, then we can just write that  $c$  of  $f$  p the flow value can be increased among, because this much. So, we will take an

example. So,  $c_f$  of  $fp$  is basically minimum among this  $c_f(u, v)$ , minimum among the capacity in the in that path in that part and that minimum we can pass, this is the more value we can pass on that path.

(Refer Slide Time: 28:02)

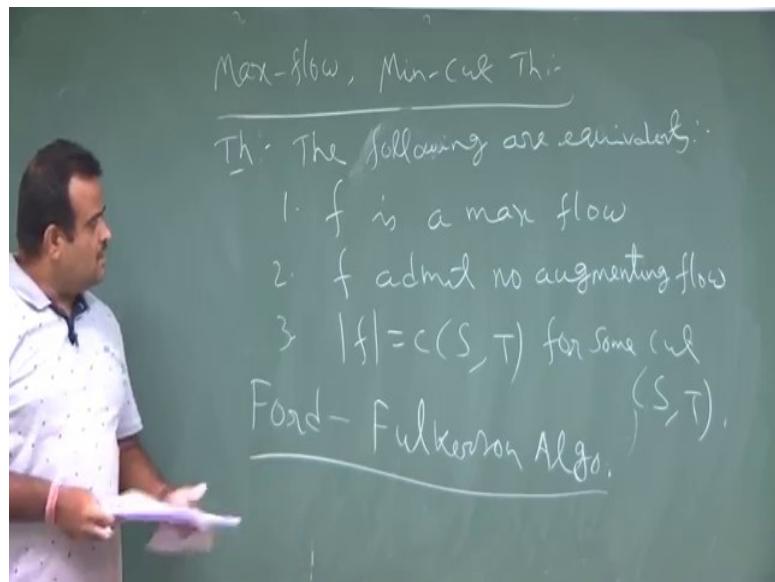


So, we will take an example let us take an example suppose we have a in the original graph we have. So, this is  $s$  and we have some vertices. So, this is these are the zeros, 2 these are the capacity and the flow, 55 i. So, this is  $t$  say 2,5. So, suppose this is a path from  $s$  to  $t$  in the original graph, now what will be the then the path for from the in the residual graph that is that is called augmenting path, in  $G_f$  how we draw  $G_f$ . So,  $G_f$  is basically this is  $s$ . So, now, how much more, we can put from here to here 2 now from here to here we can pass 3; now how much more we can pass here to here 4 and here to here 2 now here to here how much more we can pass 2 and this is basically overall all we have used. So, there is no way and then this is basically here to here no this is basically 7 because this is 5 current is coming. So, this way we can pass this 5. So, this is minus 2 minus or minus 5 this is 7 and here to here we can pass this 2 and here to here still here we can pass 1 and here to here we can pass 3 and here to here again pass 2.

So, this is the augmenting path from  $s$  to  $t$ . So, any path for me in the residual graph is augmenting path, but this is the augmenting path now what is the minimum capacity in this path? Minimum capacity is 2 here in this way so; that means, this 2 we can 2 value we can easily pass more on this network on this path. So, that will give us a theorem. So, this is

called min cut max flow theorem.

(Refer Slide Time: 30:48)



So, this theorem will give us the algorithm. So, max flow 2 minutes. So, we will just state the theorem we are not going to prove this in this course. So, the following are equivalent.

So, if  $f$  is a max flow then  $f$  admits no augmenting path; that means, because if there is a augmenting path, we can pass some more current there. So, that is; that means, it is not a max flow then this is basically  $c$  of  $S, T$  for some cut. So, there exist some cut. So, this is the theorem and there is a algorithm called ford Fulkerson algorithm based on this theorem, which will be which is the algorithm to finding the max flow. So, this is this algorithm is based on this theorem. So, we are not discussing this now, but this will be I mean this theorem I mean this algorithm is based on the theorem and we are not proving this theorem in this class.

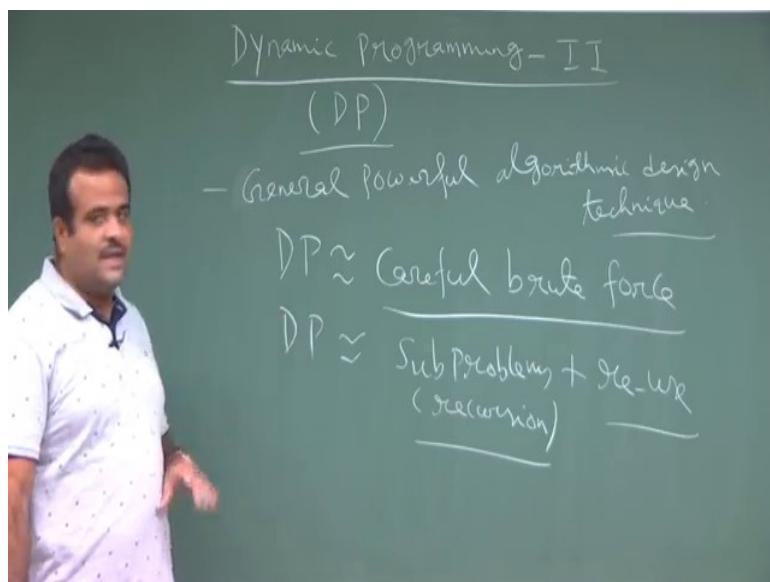
Thank you.

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

**Lecture - 57**  
**More On Dynamic Programming**

So, we talk about dynamic programming which we have seen earlier also, you have started.  
So, we will talk about more on dynamic programming.

(Refer Slide Time: 00:35)

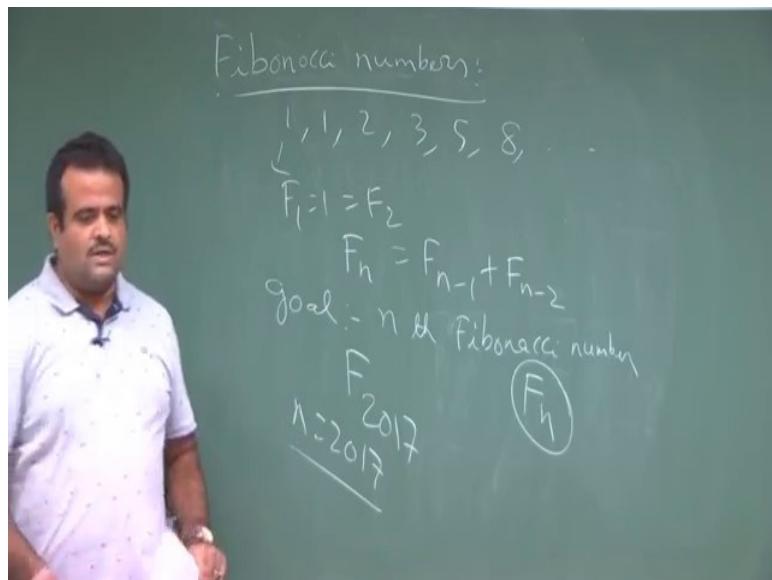


So, it is very exciting topic. So, dynamic programming in short it is called DP. So, it is basically a very powerful general design technique. So, it is basically a powerful general design techniques to solve the problems. So, basically it is a careful brute force. So, DP is basically careful brute force or exhaustive search. So, in exhaustive search the time is exponential. So, we look at the all possible solutions. So, that is called exhaustive search or brute force. So, that is the exponential algorithm, but we reduce that exponential space to the polynomial space carefully. So, that is the technique of dynamic programming. So, it is a basically; we reduce this exponential space to polynomial space. So, that is the idea.

So, it is specially good for kind of optimization algorithm like if there is a shortest path. So, dynamic programming is very much useful for optimization algorithm. So, it is basically. So, we have a problem we reduce this problem in a sub problems. So, this is basically sub problems we have to get the sub problems and then we reuse the sub problem. So, this is by

recursively. So, this is the recursion. So, recursion and then we reuse this sub problems to get the; I mean we reuse this value in the whole problem. So, this is the idea. So, basically we discuss this through some example like problem like finding the Fibonacci number and then the shortest path problem. So, let us just start with the Fibonacci number problem. So, finding the nth Fibonacci number; so, these we have discussed already in the beginning of this course like when you start the divide and conquer approach.

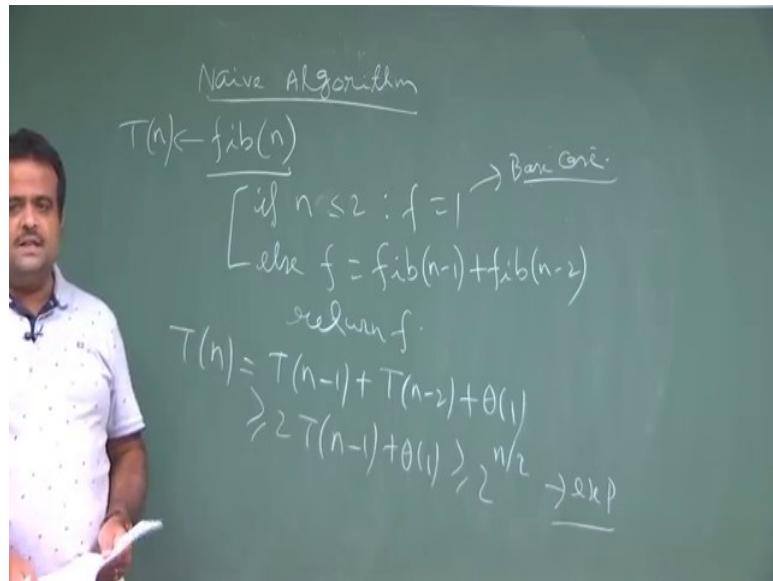
(Refer Slide Time: 03:47)



So, there we have discussed this. So, just to bring this problem and we will see how we can solve this using a dynamic programming technique.

So, here is the definition of the Fibonacci number problem, Fibonacci numbers. So, what are the Fibonacci numbers basically you start with one; 1, then 2, then 3, then 5, then 8. So, sum of previous 2. So, this is basically it start with say  $F_1=F_2=1$  and then  $F_n = F_{n-1} + F_{n-2}$ . So, this is the formula. So, our goal is to find the nth Fibonacci number that is  $F_n$ . So, this is the problem we have to find  $F_n$ . So,  $n$  is an input say we have to find  $F_{2017}$ . So,  $n$  is 2017. So, this is the problem. So, finding the nth Fibonacci number. So, what is the naive of algorithm to solve this problem?

(Refer Slide Time: 05:22)



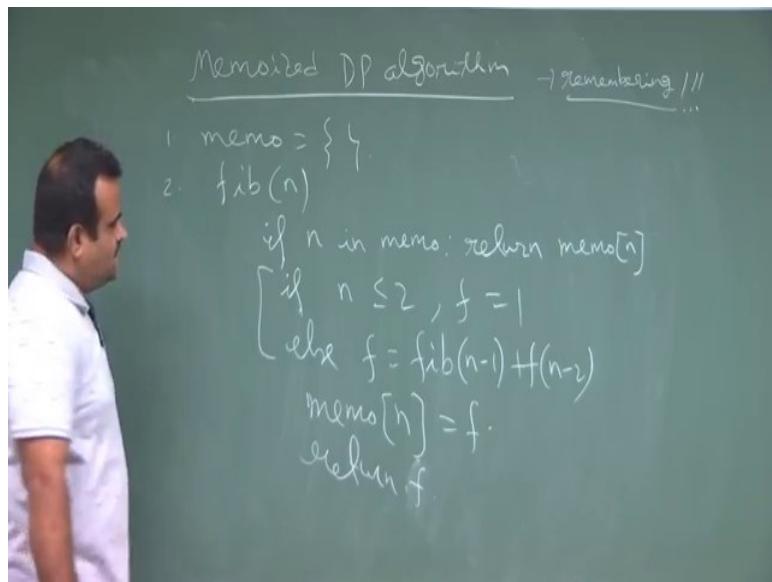
So, what is the correct algorithm to solve this problem? So, if you use this recurrence. So, you have to use this formula. So, this is the naive algorithm.

So, we have to follow the recursive formula like. So, suppose you want to find this is the call, now if  $n$  is less than 2 we return 1. So, this is the base case otherwise else. So, else what we do we compute  $f$  of  $\text{fib}$  of  $n - 1$  last  $\text{fib}$  of  $n - 2$  and we return  $f$ . So, this is the correct algorithm. So, to compute the  $f$   $n$   $\text{nth}$  Fibonacci number; so, this is the. So, this is the recurrence, this is the recurrence, right. So, this is coming from the definition of the  $f$   $n$ . So, this is the recursive definition and this is what is called base case. So, now, what is the time complexity of this algorithm or of this code? So, time complexity this is exponential algorithm why because if  $T$   $n$  is the time to time for this then  $T$   $n$  is basically we have 2 call  $T$   $n - 1 + T$   $n - 2 + \theta(1)$ .

Now, the why this is exponential this is basically we can write as greater than equal to  $2 T$   $n - 1 + \theta(1)$ . So, this will again give us greater than  $2^{n/2}$ . So, this is basically exponential algorithm. So, now, this is a exponential time algorithm. Now you want to use the technique which is called dynamic programming technique to reduce this exponential algorithm to the polynomial time algorithm like linear algorithm. So, we will use 2 techniques to do that one is memorization; memoize DP algorithm another one is bottom up DP algorithm. So, first let us start with memoized DP 1 because this is exponential exponential is bad. So, you want to reduce this we want to use the DP technique to reduce this time complexity from exponential

to polynomial.

(Refer Slide Time: 08:31)

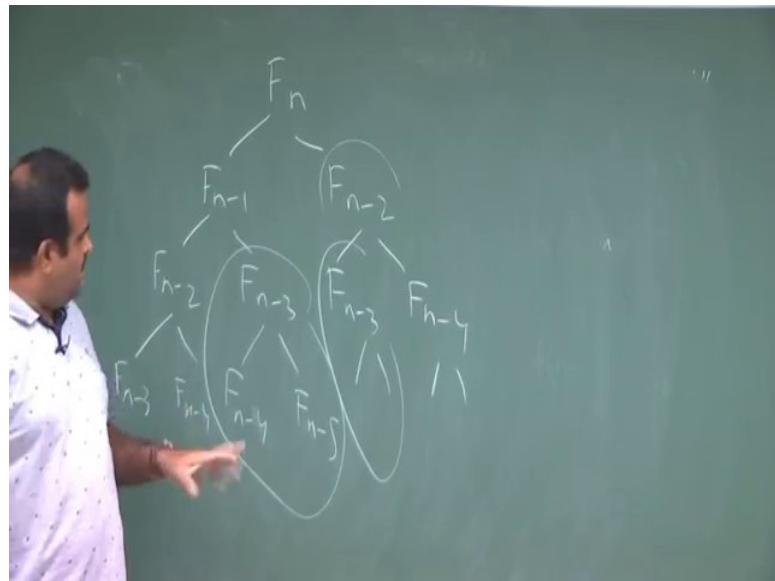


So, this is the first technique we will use memoized DP algorithm. So, memoize mean we will remember once we compute a value we remember. So, we will use a dictionary we will put the values over there, so that we do not need to compute it again. So, that is the memoization remembering. So, we remember the value. So, this is basically remembering. So, we remember the value. So, what is the how we remember we use a dictionary once we compute the value we put it into dictionary.

Now, when we look at the dictionary if the value is there then we will get that otherwise we have to compute it and once we compute it we put it into the dictionary. So, that is the memo memoize that is the remembering. So, suppose this is our dictionary which is empty initially and say this is the code and now we are computing fib of n now if n is not in the dictionary if n is in memo, then we return memo n. So, this is just a kind of table lookup we do what else if; then the code is similar to earlier is the recursive call else if n is less than 2 then F is equal to 1 else we compute we call fib of n - 1 F of - 2 and then we put this into the memo and the return F we return f. So, this is the recursive step same as earlier, but here we are just look at the table we had just look at the dictionary. So, if that is in the table then we are just we are just not computing that otherwise we are computing that.

So, this we can this type of technique we can use for any recursive algorithm this kind of. So, let us look at the recursive tree for this code. So, so suppose we want to compute F of n.

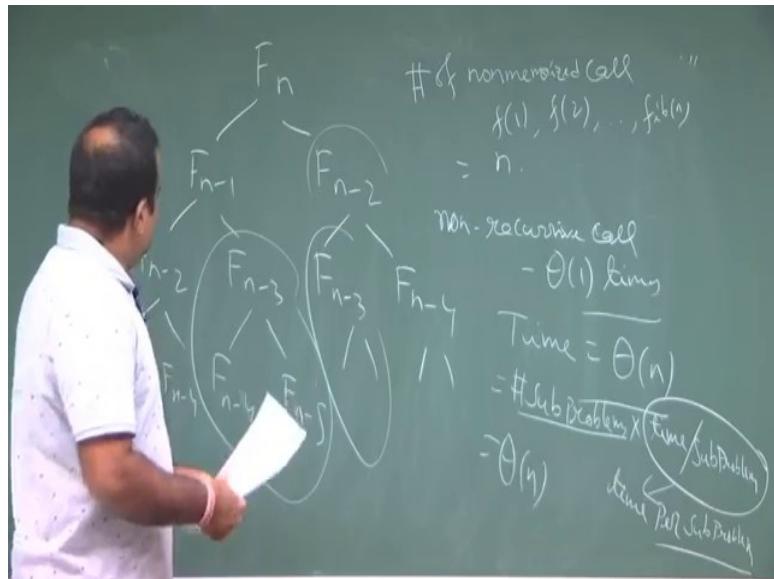
(Refer Slide Time: 11:41)



So, to compute  $F$  of  $n$  we need to compute  $F(n - 1)$  and  $F(n - 2)$ . So, in order to compute  $F(n - 1)$  we need to compute  $F(n - 2)$   $F(n - 3)$ . So, here also we need to compute  $F(n - 3)$  and  $F(n - 4)$ . So, this is the exponential time algorithm. Now here if we look at this tree and these trees are same and this tree and this tree are same. So, there are many sub problems are going on. So, this is one of the hallmark of dynamic programming technique. So, if you are many sub problems. So, by once you calculate this because here we are again calculating this once we calculate this why to calculate it again.

So, to avoid that; we are going to memorize that we are going to remembering that value. So, we are going to store into the table. So, this way we are saving the time otherwise this will be exponential. So, this is the technique. So, we are memoizing we are storing this once we calculate this we are storing into the dictionary now once we are going to calculate these.

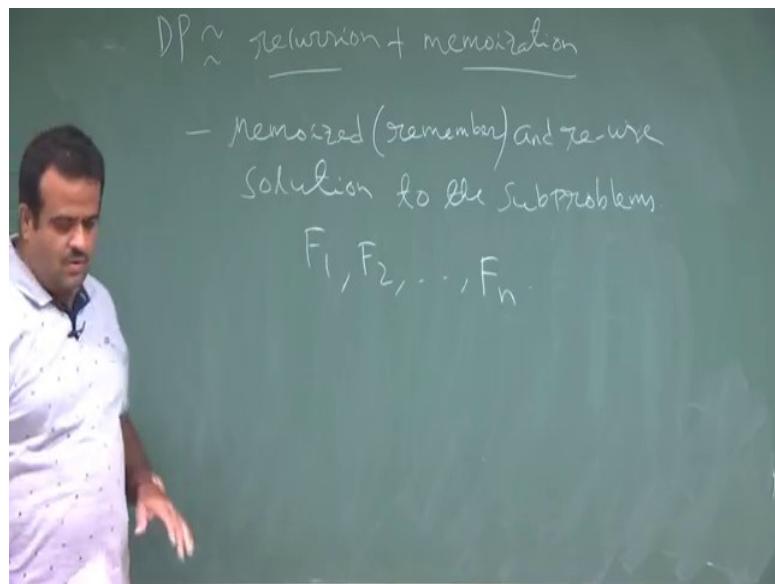
(Refer Slide Time: 13:34)



We look at the dictionary since we got it will not execute this calculation. So, that way we are saving the time. So, basically how many non memoized call you have to do. So, number of non-memoize. So, we have to calculate this  $F_1 F_2$  up to fib n. So, n times the number of non memoize calculation this is n times. Now for memoize calculation. So, that is the non recurrence. So, non recurrence; non recursive call; it basically takes theta(1) times because we have already calculated this. So, now, what is the total number of what is the time complexity? Time complexity is basically theta(n). So, time complexity is basically theta(n). So, basically number of sub problems into the time to solve each sub problems this is the amortized analysis because few sub problems we are calculating that by adding, but few sub problems we are getting from the look of. So, that is the way. So, basically we are just adding this 2.

So, this is basically time divided by sub problems. So, this is basically time per sub problems and time per sub problems is theta(1) because we are just looking at the table to get the value and this is n. So, that is why it is coming to be theta of n into theta (1). So, this is basically theta of n. So, this is the theta of n times algorithm to getting the Fibonacci number. So, now, we look at another approach which is instead of putting everything into a dictionary we look up what is called bottom up DP algorithm.

(Refer Slide Time: 16:46)



So, this is basically; this is basically bottom up DP algorithm. So, let us just before that let us. So, DP is nothing, but recursion + memoization. So, basically idea is we memoized or remember; we remember the value which you have already calculated and we reuse the and reuse the solution to the sub problems. So, that is the to the sub problem. So, basically, what are the sub problems in Fibonacci numbers? So, this is basically  $F_1$   $F_2$  up to  $F_n$ . So, these are basically our sub problems. So, there are  $n$  sub problems and to solve each sub problem it is taking constant time because we are just looking at the table and then getting the value we are adding. So, theta(1) time and there are  $n$  sub problems. So, that is why it is theta of  $n$ . So, this is the linear time algorithm.

So, now we look up the another approach another DP approach which is called bottom up DP algorithm where we just store the last 2 value instead of putting everything into dictionary because we just need the  $F$  of  $n - 1$  and  $F$  of  $n - 2$ , we do not need to store the whole dictionary.

(Refer Slide Time: 18:39)

Bottom-up DP algorithm

1.  $\text{fib} = \{\}$   $f_1, f_2, f_3, \dots, f_k = f_{k-1} + f_{k-2}$

2. for  $k$  in  $[1, 2, \dots, n]$

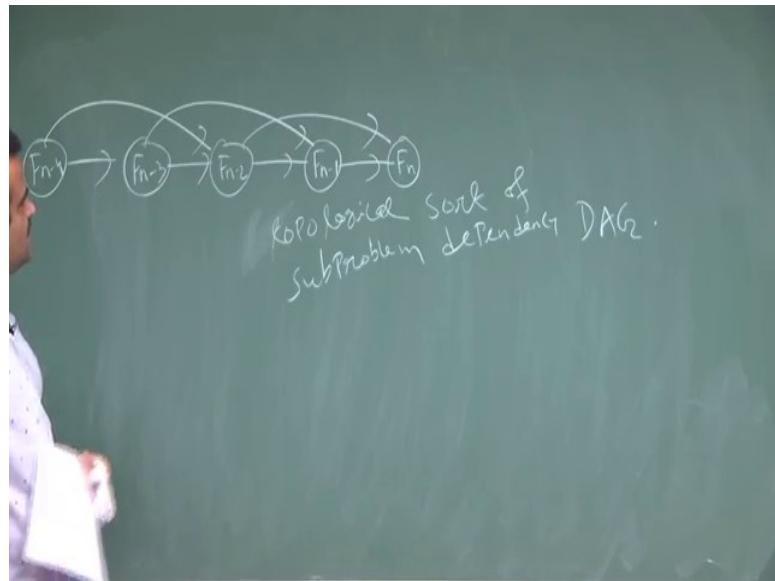
$\Theta(n) \left\{ \begin{array}{l} \Theta(1) \left\{ \begin{array}{l} \text{if } k \leq 2 : f = 1 \\ \text{else } f = \text{fib}(k-1) + \text{fib}(k-2). \\ \text{fib}(k) = f \end{array} \right. \\ \text{return fib}(n) \end{array} \right.$

So, that is the idea. So, this is called bottom up DP algorithm. So, what we are doing here this is the code. So, we have this empty set now for  $k$  in the; this is the range of the  $k$ . So, range of the  $k$  is basically 1 to  $n$ , we are calculating  $F_n$ . So, our range is 1 to  $n$ . So, now, we have the recursion that recursive formula less than equal to 2  $F$  equal to 1 else  $F$  is  $f$  of  $\text{fib}$  of  $k - 1$  +  $\text{fib}$  of  $k - 2$  and then  $\text{fib}$  of  $k$  is basically  $F$  and we return we return  $\text{fib}$  of  $n$ . So, this is basically. So, this part it is basically theta of one time. So, this is this is basically we are just storing last 2 value. So, this is the bottom up way. So, we start calculating  $F_1$ . So, we basically start calculating  $F_1$   $F_2$  then  $F_3$   $F_4$ . So, always we store the last 2 value like this.

So, we keep on calculating this way and ultimately we return this. So, this is the bottom up way. So, we start with  $F_1$ , then  $F_2$ , then  $F_3$ ; when you call  $F_3$ , we forgot  $F_1$ . So, this way; so,  $F_k$ ;  $F_k$  is basically  $F(k - 1) + F(k - 2)$  this way. So, every time we just store last 2 value of the Fibonacci number and then we add it up to get the next value; so, we stop at  $F_n$ . So, this is the linear time. This is basically theta( $n$ ) times algorithm this is the same time complexity as the memoize DP algorithm, but this has the advantage that the space complexity is less. So, for memoize we are putting everything into the dictionary. So, exactly same computation, but less space; so, this is practically faster because there is no recurrence.

Now, we want to see some topological ordering for this computation of  $F_n$ .

(Refer Slide Time: 22:02)



So, to calculate  $F_n$ , we need to calculate  $F(n-1)$  and  $F(n-2)$ . So, similarly it goes on. So, this is basically a topological sort of sub problems. So, these are the sub problems sub problems dependency graph dependency a I cyclic graph. So, this is basically this is basically the DP approach for this problem. So, this is giving us this. So, our naive approach was exponential, but we reduce this into polynomial time algorithm. So, this is basically linear term T turbine.

Now, we have seen this problem can be solved in  $\log n$  who can remember this problem of finding the Fibonacci nth Fibonacci number; so, how to how to get nth Fibonacci number. So, we have seen this in the divide and conquer technique in the earlier beginning of the course.

(Refer Slide Time: 24:07)

So, we have seen this  $F$  of  $n$  last one  $F$  of  $n$ ;  $F$  of  $n$ ;  $F$  of  $n - 1$ , this is this value  $[1 \ 1 \ 1 \ 0]^n$  this kind of formula we have seen. So, this is basically kind of powering a number powering a matrix, but this matrix is a constant matrix. So, once you have a constant matrix. So, this is some sort of  $A^n$ , but  $A$  is a 2 by 2 matrix; just 2 by 2 matrix. So, it is kind of same as powering a number. So, it is basically we have the divide and conquer formula for this; this is basically  $A^n$  by 2 into  $A^n$  by 2 if  $n$  is even otherwise  $A^n - 1$  by 2 into  $A^n$  by 2 \*  $A$ ; if  $n$  is odd.

So, this is the divide and conquer steps. So, we have a problem of size  $n$  we reduce this problem into sub problems of lesser size and then what is the recurrence. So, only sub problem  $T(n/2)$  last  $\Theta(1)$ . So, this will give us  $\Theta(\log n)$ . So, this we have seen in the divide and conquer technique class, but if we use the DP we can solve it using. So, DP is a general technique where we can reduce a exponential time algorithm to a polynomial time algorithm. So, that we will use for many optimization problem can be reduced in the DP.

So, in the next class, we will discuss another problem which is called shortest path problem and we will see how we can use the dynamic programming approach to finding the single source shortest path.

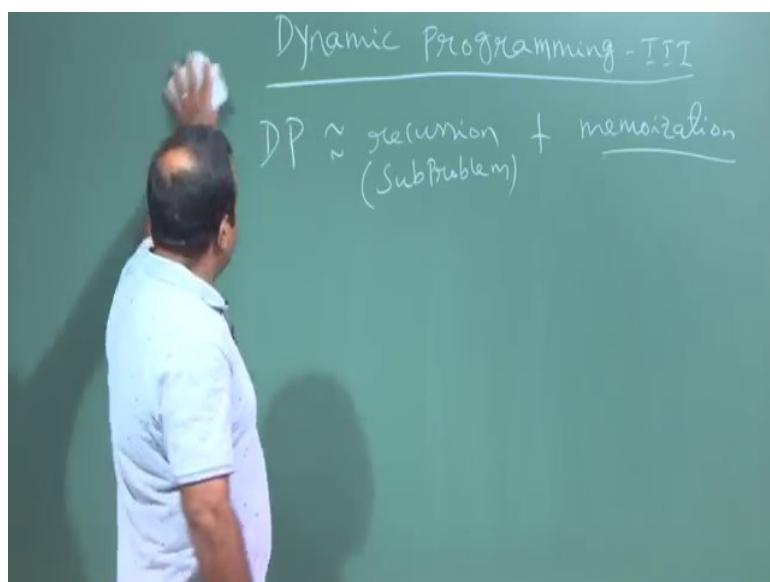
Thank you.

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

**Lecture – 58**  
**More on Dynamic Programming (Contd.)**

So, we are talking about dynamic programming technique it is a very powerful tool to solve most of the; I mean many optimization problem it is very much useful for optimization problem. So, in the last class we have seen the Fibonacci;; how we can use the dynamic programming technique to find the nth Fibonacci number. So, basically we have given a recurrence. So, we just try to write any formula in the recursive form if we can write that then we can apply the dynamic programming technique. So, basically dynamic programming is basically.

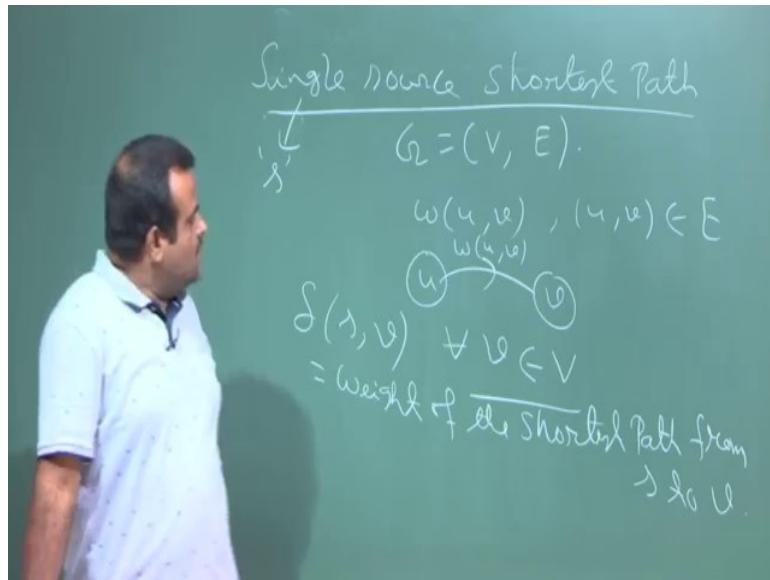
(Refer Slide Time: 01:02)



So, we call this as DP in short, it is basically the recursion. So, recursion means sub problem. So, we have a problem. So, we reduce into your sub problems and then and then we use this solution of the sub problem reuse; reuse means we have to memoize. So, there are 2 version of DP, we have seen in the last class one is memoization. So, we memorize the value once it is computed then we use that reuse that value. So, that is the memoization.

So, this is the basically. So, we memoize the; we stored this into some dictionary then we use it. So, in this lecture we will talk about another problem which is basically the shortest path problem single source shortest path problem.

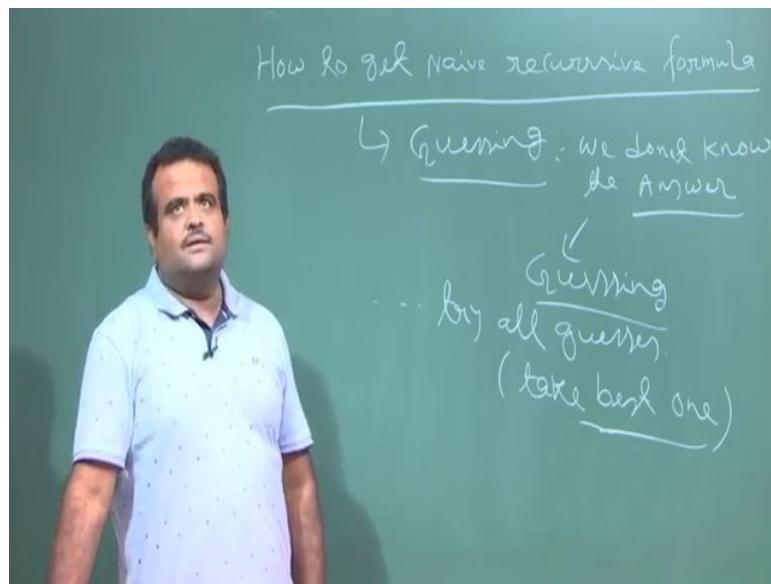
(Refer Slide Time: 02:05)



Single source shortest path; so, what is the problem? Problem is you have given a directed graph and we have some edge weight on the graph. So, that is basically  $w(u,v)$ . So, this is the weight function where  $uv$  is an edge. So,  $u$  and  $v$  are 2 vertices in the graph. So, each edge is associated with a weight function  $w(u,v)$ . So, then you have to find it. So, there is a single source there is a source  $s$  which is also another input. So, we have to find basically delta of  $s,v$  for all  $v$  belongs to  $V$ . So, this is basically the weight of the shortest path of the shortest path from  $s$  to  $v$ . So, this is basically we consider the path from  $S$  to  $V$  all path. So, among them which is the minimum weight. So, that is the weight of the shortest path. So, we have seen the algorithm like distress algorithm which will run for positive weights; positive weight edge and then we have seen the bellman ford algorithm.

So, these are all the greedy approach well one for now we will look at how we can solve this using the help of dynamic programming technique. So, for that; so, you want to write this in a naïve recursive way. So, the question is how we can get the recursive formula for this how we can get the recursive naïve recursive algorithm to solve this problem.

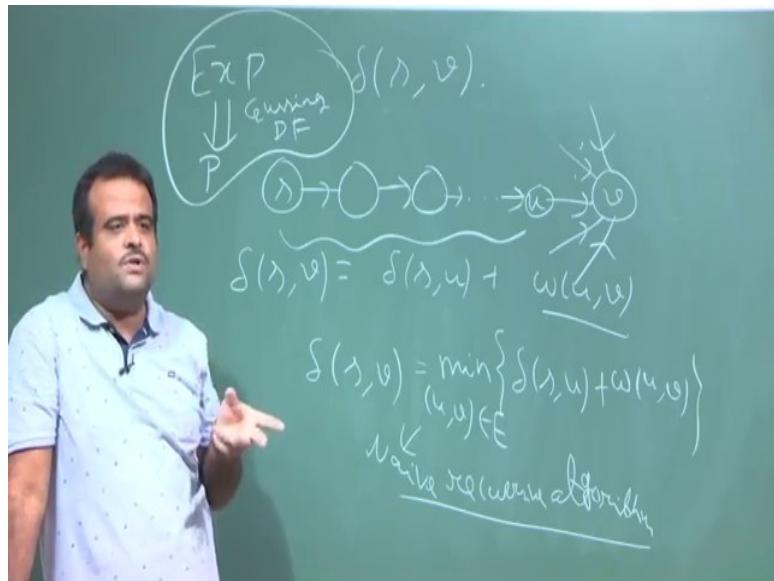
(Refer Slide Time: 04:29)



So, let us try to get that then we can think to apply the dynamic programming technique. So, question is how to get naïve recursive algorithm or recursive formula. So, the answer we do not know how to get. So, answer is we have guess. So, this is very powerful tool guessing we do not know the solution we do not know the answer. So, guessing means we do not know the answer. So, then what is the solution? Solution is to try all possible guesses. So, try all guesses this is sort of exhaustive search and this is the beginning of the dynamic programming technique. So, we have exhaustive search, then we will take the best one I mean we will try for all possible guessing then take the best one with the help of with the help of DP technique. So, that is the idea.

So, let us just try to, how we can get a naive recursive formula or naive algorithm for the single source shortest path. So, we want to find delta of s,v. So, basically we have source vertex s and we have another vertex V.

(Refer Slide Time: 06:30)



So, we have some intermediate nodes. So, there are some vertices which are incoming vertex of V. So, these are the edges which are direct edge form which are direct edge going to v there. So, these are the incoming edges which are going to V.

So, now we want to find the shortest path from s to v. Now we want to write this in a recursive form. So, suppose hypothetically suppose somehow we know some shortest path from s to u just one node before v. So, these are all direct edge now there are many direct edge, we do not know which will give us the minimum one. So, what we do we try for all and then we try for all and then we guess. So, we tried for all and then we have to choose the best guess. So, that the DP will give us; so, then this plus w uv, this is the one of the vertex direct edge form to v, but we do not know which one. So, we have to explore all positives is sort of exhaustive search. So, this is the exhaust, but we do not know which one. So, suppose we have a this is the sub problems. So, we reduce the problem into sub problems.

Suppose hypothetically we have the solution of this sub problem then we have this direct edge. Now we want to choose one of this will give this shortest path, but which one we do not know. So, we have to try for all possibilities these are the. So, we have to guess which one. So, this is the guessing technique. So, so if the; our guess is correct then this will give us delta of s,v but we have to do this exhaustive search. So, basically delta of s,v this is the recursive formula minimum among delta of s comma u + w (u v) where u v is the direct edge form to v. So, these are all possible edges from E.

So, this is the naive recursive algorithm. So, is this good? So, this is the exhaustive search. So, this is basically this is the exhaustive search we are doing. So, this is the algorithm is good. Now, this is not good, this is huge. I mean this is exponential. So, this algorithm is exponential algorithm.

So, now we want to reduce this to be a polynomial with the help of guessing and that will be done by DP method. So, that is the goal of the DP method. So, goal of the DP technique is we have a exponential we have exhaustive search it is a careful brute force. So, carefully we have to do the guess. So, this is the exponential time algorithm. So, we cannot go for all possibilities we have to guess which edge will give the correct one we will give the minimum one that you have to guess. So, that is the goal of the DP. So, you have to guess that. So, that is basically that we are going to do. So, that is the repeat techniques. So, that will do by memoization so; that means, we will once we; so, we will use the same algorithm what we have done earlier. So, let us just write this. So, basically the DP technique is now guessing is added there.

(Refer Slide Time: 11:22)

DP  $\approx$  glorification + memorization + Guessing.  
(SubProblems)

$$\delta(s, v) = \min_{(u, v) \in E} \{ \delta(s, u) + w(u, v) \}$$

Memoized DP Algorithm

1.  $\delta(s, v)$  in memo.

$\delta(s, u) = \delta(s, u) + w(u, v)$   
↳ memo( $\delta(s, u)$ )

So, DP is basically we have we have sub problems the recursion we have this is basically the sub problems and then what we have to do the memoization and one thing is added now is guessing we want to guess the correct path.

So, now, what is the delta of s,v? delta of s,v is the minimum of delta of this is the recursive formula  $y(s,u) + w(u,v)$ . So, this u v belongs to E. So, this is the basically now these formula.

So, this is the Naive approach this is the exponential time approach. So, we have to use we want to use the memoized DP algorithm for this. So, that will reduce this from exponential to polynomial. So, what is that algorithm?

So, we compute this deltas now we compute delta of s,v and we put it into a table. So, again if we want to compute it; so, we first look into the table. So, if it is in the table then we return it otherwise what we do? Otherwise we compute delta of s,v; it is equal to delta of s,u + w(u,v) and this is the sub problem. So, this is the sub problems now once we compute this then we put it into the memo. So, that next time if we need this value we can reuse that. So, that is the idea.

So, this is the memoization algorithm for this version now what is the time complexity. So, we know the memoization.

(Refer Slide Time: 14:03)

$$\text{Time} = \# \text{ of Subproblems} \times \left( \frac{\text{time}}{\# \text{ of Subproblems}} \right)$$

~~2~~  $\rightarrow v$   $\rightarrow u$

time per Subproblem.

$\# \text{ of Subproblem} = V$

Time for each Subproblem

$$\sum_{u \in V} \text{Time} = \Theta\left(\sum_{u \in V} (\text{undeg}(v) + 1)\right)$$

$$= 2|E|$$

$$= \Theta(E + V)$$

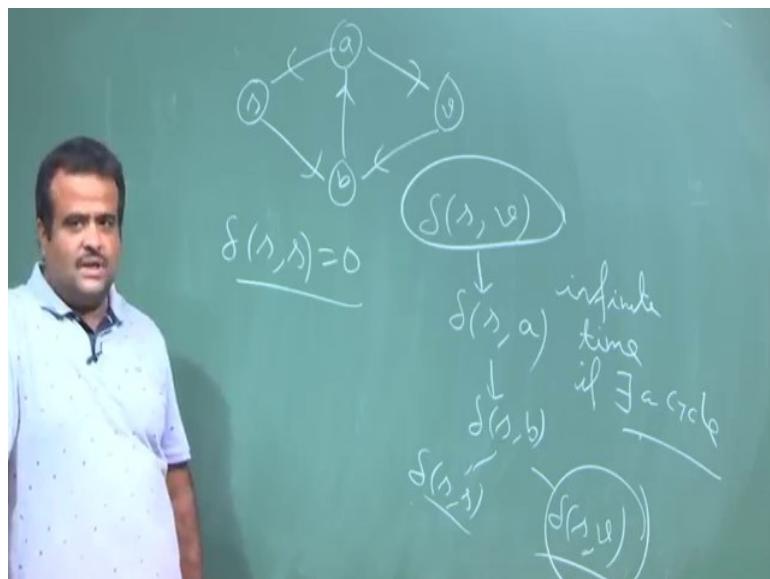
So, this we know the time for this type of DP is basically time is equal to number of sub problems into the time required parts of problems. So, this is basically total time divided by number of sub problems. So, this whole thing is specifically rate of executing the sub problem. So, this is basically time per sub problems.

So, now what is the time then? So, how many sub problems we have we have 3 sub problems. So, number of sub problems is equal to order of v now what is the time to solve each sub problems how many guesses are there the exhaustive search? So, the basically the

time for each sub problems is basically. So, this is basically indegree of  $v + 1$  now if you take the sum. So, we can put a theta over here summation in degree of  $v$  in degree of  $v$  plus  $v$  for this one. So, indegree of  $v$  is basically order of  $e$  + order of  $v$  this is coming from handshaking lemma. So, summation we know the summation of degree of  $v$  is basically I mean order of  $e$  this is the handshaking lemma this is coming from handshaking lemma.

So, now is this algorithm work for any graph no. So, the unfortunately no; so, otherwise this technique is quite because this is this will not work for this is work for only dag direct acyclic graph if there is no cycle, but problem is if there is a cycle then this method will fail.

(Refer Slide Time: 17:29)



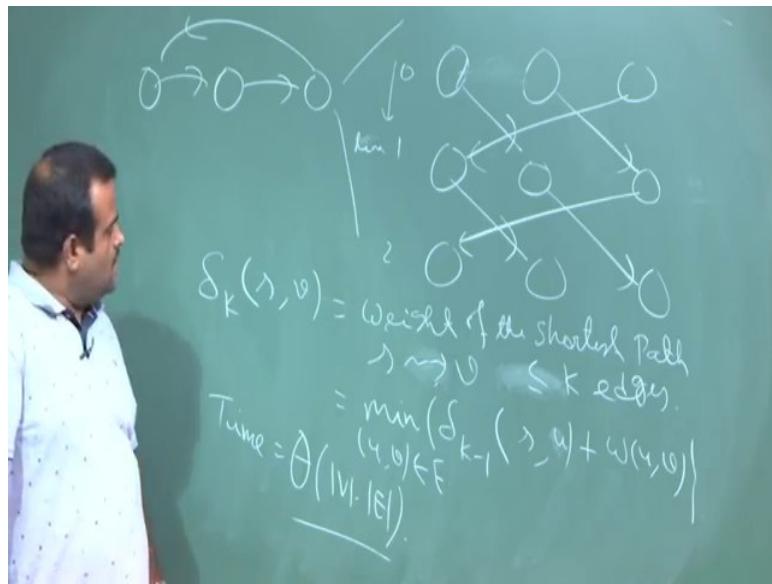
So, let us look at that. So, suppose we have a graph like this  $s$  if  $v$ . So,  $a$   $b$  and these are the vertices these are the edges.

So, now suppose, we want to apply this memoize algorithm the guessing algorithm and they want to memoize, suppose, we want to compute  $s$  this  $s$  to  $v$ . Now, to compute this what are the edges going to free in degree edges  $a$ . So, to compute this you have to compute  $\delta(s,a)$  because there is only one edge going for this now to compute this  $\delta(s,a)$  we need to look at number of edges which are in degree to this. So, for this we need to compute  $\delta(s,b)$ , now there are 2 edge incoming. So, to compute this we need to compute  $\delta(s)$  comma  $s$   $\delta(s,v)$ .

Now, delta of (s,s) this is the base case this we assume zero, but this again is same as this one now how to get this value we haven't computed see this is the we are storing in the dictionary after getting the value, but this value is yet to get we haven't completed. So, we stuck here we cannot proceed further because this is again a loop. So, this is the infinite time if there is a cycle. So, this will take infinite time this is a; this we stuck in finite time if there is a cycle in the graph that is the problem.

So, this method will not work directly on the on the graph which is having cycle because here we stop because how to get this value we haven't computed. So, we will look at the table, but it is not yet computed because we have to compute we are computing this and then we stuck in this way. So, how to overcome this problem? So, we know a technique. So, that is suppose you have a graph with cycle. Now how we can view this how we can remove the cycle from the graph.

(Refer Slide Time: 20:18)



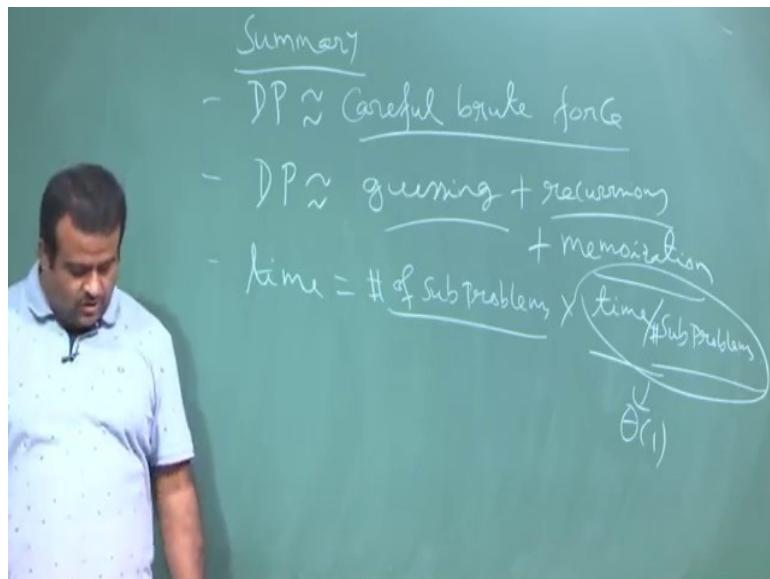
Suppose we have a graph like this and there is a cycle now we want to have a; many version of this graph. So, this is the way how this is the time. So, this is the; this is connection. So, this is these are the connection this is going here this is going here. So, this is the; this we are we are just writing the copy of the same graph many times like this. So, this is the zeroth copy one time.

Now, once we move from here to here we just move from here to here like this. So, this way we can just avoid the cycle so, but the thing is we are just increasing that. So, we will try to

this is a cyclic graph. Now we will try to find the; we will apply this DP memoize DP to get the shortest path on this graph, but there. So, what is the problem with this approach problem with this approach is it will in. So, it is increasing the number of vertices. So, if you write this delta k is s,v is equal to weight of the shortest path from s to v which use yeah which uses which use less than k edges. So, number of edges in that path is less than k. So, this is basically we know the formula there minimum of delta of k minus 1 is , v + w of uv. So, this is the recursive formula we used.

So, this is sort of its relaxation step if you remember the bellman ford algorithm or distress algorithm we have a relaxation step this is sort of the relaxation step. So, now, here what is the number of sub problem? it has increased by the order of v square. So, the time complexity here is for this graph is basically order of v into e. So, this is the same time complexity of bellman ford algorithm, but this is the way we just usually avoid the we just avoid the cycle in a graph. Now we will just quickly summarize the DP technique in general there are five steps in any DP approach.

(Refer Slide Time: 23:41)

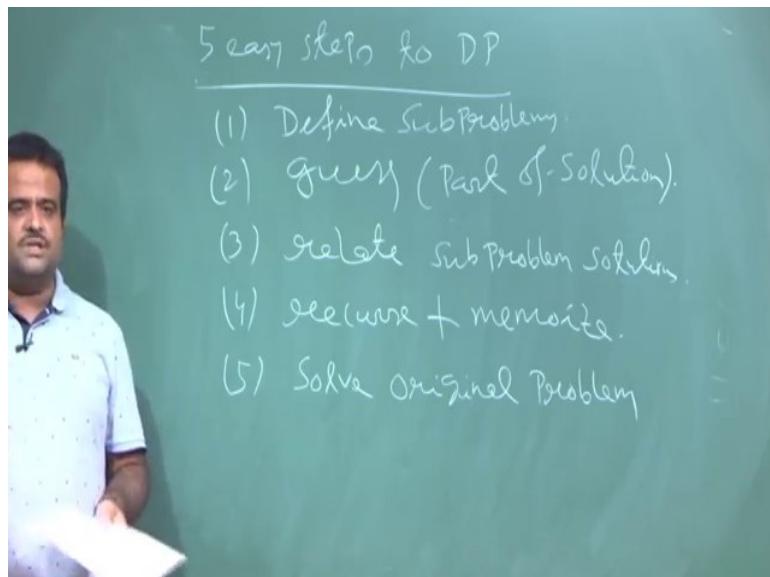


So, we have seen a powerful technique which is dynamic programming technique which is mostly used to solve any optimization problem because it is basically reduce the space form we have a exponential, it is basically careful good force. So, it is careful exhaustive search. So, we have exhaustive search we have exponential space now from there we are going to reduce this to a polynomial space carefully. So, basically it is we for this technique we need

to have a recursive formula or recursive algorithm which is naive approach which is recursive algorithm which for which we can use the memoization version. So, this is basically guessing plus recursion basically we need to have the sub problems plus the memoization. So, we will just store the value into the dictionary so that when the next time we need that value we can reuse that.

So, the idea is to divide the problem into reasonably number of sub problems and once we compute the value of the sub problems we store it, we memoize that and then we will reuse that and the time complexity is basically we know this is basically number of sub problems into time per sub problems this is the sub problems this is the time per sub problems and this is the total number of sub problems. So, this is basically our. So, usually this time is theta one because we just mostly we will do the table lookup. So, the total time is if this is order of n then the total time will be n. So, this again this analysis is amortized analysis. So, because we are doing the average case analysis I mean on an average what is the time. So, this is the amortized sense. So, then we write the five easy step for any DP method.

(Refer Slide Time: 26:27)



So, the five steps to dynamic programming technique one is first one is we have to define the recurrence we have to define the sub problem. So, we should have that naive recursive formula define the sub problems this is the first step second step is we need to guess guessing part of solution we need to guess. So, that way we reduce this exhaustive exponential space to polynomial space now we relate the solution of sub problems relate sub problem solutions

and then we just recurs plus reuse plus memoize or inset of memoize version we can have a bottom up version. So, that also can be done.

Then we solve the original problem which is a combination of solution of the sub problems. So, let us take quick example we have seen 2 problems in the today's lecture and last lecture one is Fibonacci number finding  $n$  th Fibonacci number and today we have talk about the shortest path problem.

(Refer Slide Time: 28:33)

(1) Subproblems  $F_k, 1 \leq k \leq n$       shortest path  $\delta(s, v)$   
 $\delta(s, v) = \min_{\text{edges } s \rightarrow v} w$

(2) guess      nothing, edge into  $v$ .

(3) recursion  $F_k = F_{k-1} + F_{k-2}$        $\delta_k(s, v) = \min \{ \delta_{k-1}(s, u) + w(u, v) \}$

(4) Topo. order

(5) Original Problem  $F_k$        $\delta(s, v)$

So, let us just what are the steps for these 2 problem. So, this is an example. So, this is basically we have seen the Fibonacci problem Fibonacci and then we have seen the shortest path problem.

So, step number one is basically sub problems. So, what are the sub problems basically we need to find  $F_k$  for this for one less than  $k$  less than  $n$ , this is the sub problems for this and what is the sub problems for shortest path we need to find delta of  $s, v$ . So, minimum from  $s$  to  $v$  which is used at most  $k$  number of edges. So, this is somehow hypothetically we know the answer up to the sub problem. So, that is the things. So, what is the number of sub problems here this is  $n$ , but this is here is  $v$  square.

Now, the guessing step is here we no need to do any guessing nothing. So, the choice is one and here we need to guess because there are how many vertices are coming from other. So, this is basically indegree + 1. So, this is basically  $h$  into  $v$ . So, this is basically indegree. So,

this is indegree +1 and then we have a recursion. So, for here we know the recursion  $F_k$  is equal to  $F_{k-1} + F_{k-2}$  and for here we know the recursion delta of  $k(s,v)$  is the minimum of delta of  $k-1(s,u) + w(u,v)$ , then we need to check the topological order; for that topological ordering and then we have the solution for the original problem which is going to use the; so, this is basically  $F_n$  and here it is delta of  $s,v$  this is basically delta of  $V_{k-1}(s,v)$ . So, this we will take order of one time and this will take order of  $v$  times.

So, this is the 5 basic step for any DP problem. So, we have jot down for 2 x 2 problem like Fibonacci number and the shortest path problem.

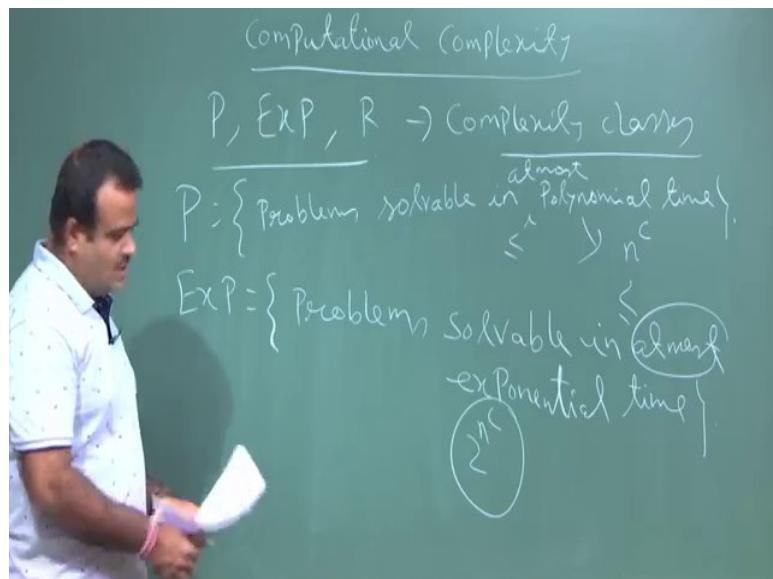
Thank you.

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

**Lecture – 59**  
**Computational Complexity**

So, we talk about computational complexity. So, what is feasible, what is not feasible in polynomial time. So, for most of the algorithm we have solved we can we have a polynomial time solution for that. So, now, this class we want to discuss when you cannot do that I mean when you cannot solve by polynomial times. So, we will talk about complexity classes.

(Refer Slide Time: 01:01)



So, basically we talk about three complexity classes P X and I. So, these are basically complexity classes.

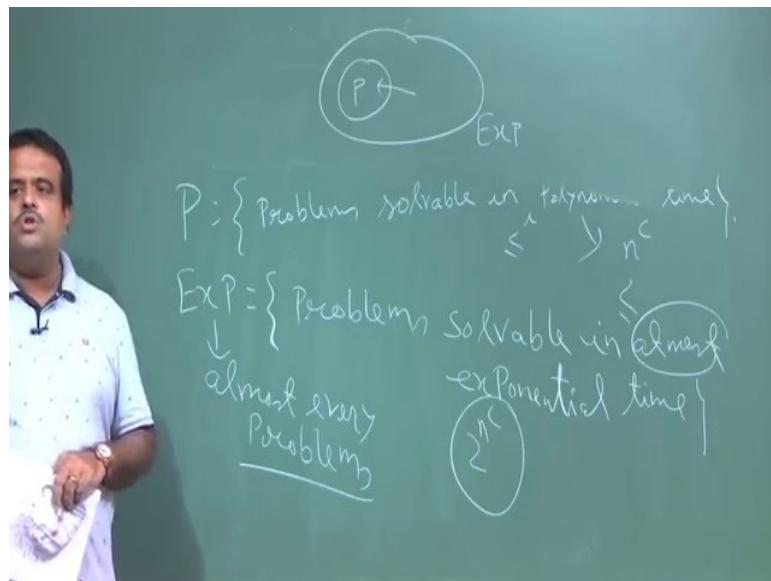
So, what is P? P is basically the set of all problems which are solvable in polynomial time. So, it will take at most polynomial time. So, P is basically problems which are solvable in at most polynomial time. So, polynomial time means the complexity is sum  $n^c$  where c is a constant. So, this is less than equal to at most polynomial time.

So, this is the class where we consider all the problems which can be solved in polynomial time. We have seen many problem so far in our lecture. So, most of them we have seen solved in polynomial time, but today we want to know there are a lot of problem which cannot be solved in polynomial time. So, the second class is X which is basically problems

solvable in at most that less than equal to exponential time that is why exp X. So, this is the class where we put all the problems, which will take exponential time.

So, exponential time in particular we look at  $2^{(n^c)}$  time. So, here n is a constant. So, this is the time complexity for such a problem. So, for example, so all the polynomial times here algorithm is subset of this. So, if we want to draw a picture, almost all problems are belongs to in this class.

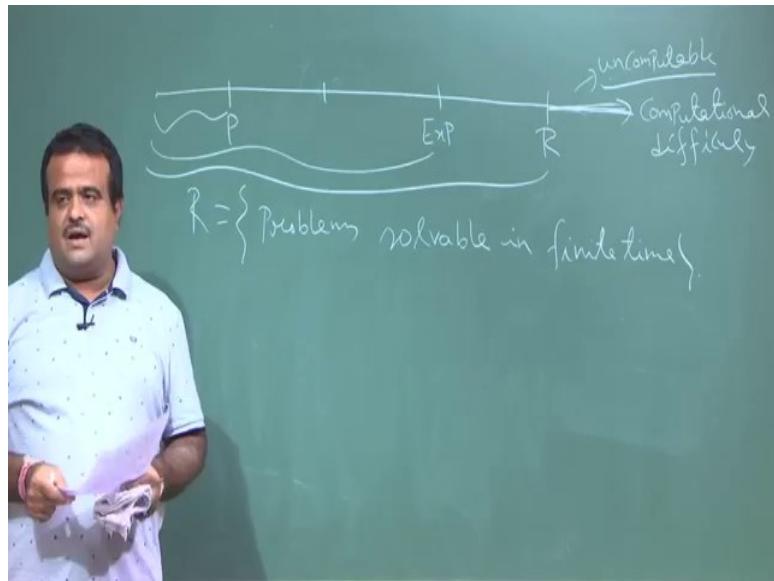
(Refer Slide Time: 03:53)



So, almost every problem are in this class. So, if you want to draw a picture here this is a P and this is X exponential time algorithm now what do you want? we want to bring one the problem from here to here. So, that we did for say dynamic programming problem, we know this is exponential time algorithm, but we try to solve it in polynomial time by the help of memorization. So, all this technique we use. So, we want to bring the problem, which we now it is an exponential time algorithm you want to bring it here.

So, P is a subset of X we have another picture to draw that this is not a very good picture to explain this. So, there are another class. So, we want to draw it in different way.

(Refer Slide Time: 05:12)



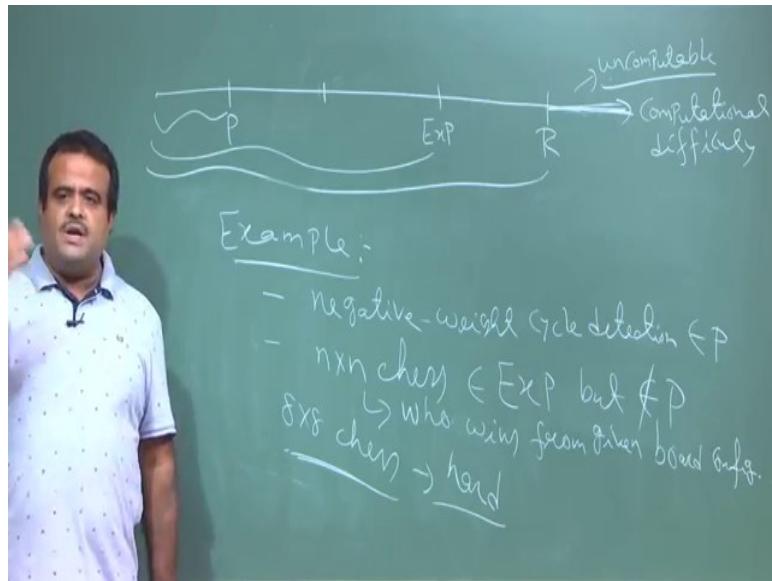
So, basically we denote this by computational difficulty. So, here up to this is P; that means, these are easy problem which can be solved by polynomial time algorithm and then we have another one which we'll explain and we have X exponential time, and then we have here what is called R, R is another complexity class. So, what is R? R is basically problem which we are solvable in finite time.

So, problems, this is the set of all problems which are solvable in finite time. So, this is basically R set. So, R is the collection of all problems which can be solved in finite time. So, this set is P which is easy problem, this set is X which is exponential time algorithm and this set is basically R which can be solved in finite time.

So, we should have a finite machine which can give as a solve state. So, anyway i am not going to that theoretical details and there are other problems which are here. So, these are basically problems which cannot be solved in finite time. So, this will take infinite time. So, these are called uncomputable problem or undecidable problem. So, these are basically uncomputable problem because this problem we cannot get a solution of this problem in a finite time.

So, this problem is undecidable. So, this problem has no solution in a finite time. In fact, most of the problem are in this range we will prove that most of the problem are in this range which are uncomputable. Now let us take some example of this classes example of problems which belongs to this classes.

(Refer Slide Time: 08:29)



So, let us take example of this classes. So, we want to take an example of polynomial time algorithm we have seen many one this is a negative weight cycle, to determine. So, we are given a graph, we want to determine whether there is a negative weight cycle or not. So, this is basically negative weight cycle detection.

So, we know this is in P because we know an algorithm which is bellman ford algorithm which time complexity is order of  $v * e$ .

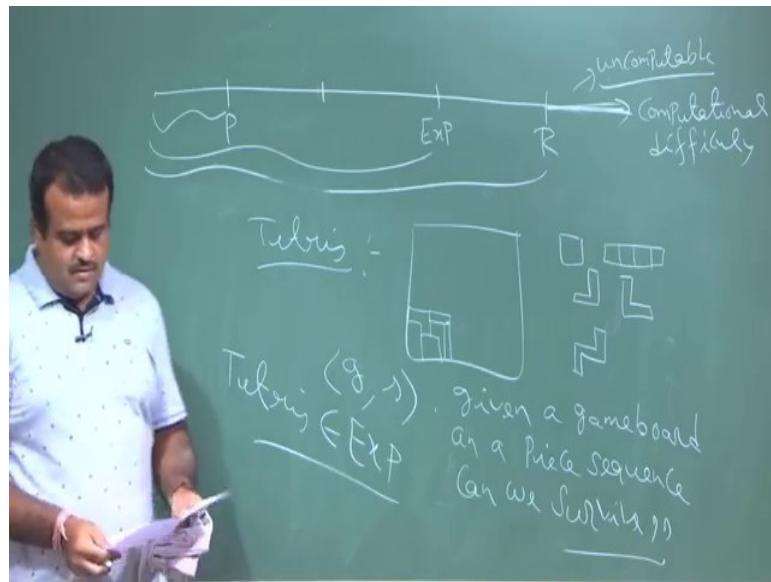
So, that is polynomial time algorithm. So, this is a example of P. Now we have to have an example of exponential time algorithm that is  $n * n$  chase problem this is belongs to x exponential time algorithm what is this? This is basically we have given a position of a chess. So, we are playing chess.

So, we have given a position of a chess position of this chess now we want to decide whether quit we not. So, this problem has no polynomial time algorithm to solve this problem. So, this problem is exponential time. So, we can try for all possible move and then. So, this is basically exponential time algorithm.

So, the problem is who wins from given chess board configuration. So, we have given a position of the board, now we want to take the decision whether white win or not. So, this is the problem. So, this problem is hard even this is hard for 8 cross 8 cross chess board this is even hard for 8 cross 8 chess box.

So; that means, there is no polynomial time algorithm to solve this problem. So, this problem is hard problem. So, this is basically belongs to exponent a exponential class and it is not in P, even this 8 bindings chess also this is not in p; that means, we are not having a polynomial time algorithm to solve this problem. So, this is a typical this is an example of exponential time algorithm. So, another exponential time algorithm is tetris.

(Refer Slide Time: 11:52)



Another exponential time problem which is basically a tetris what is tetris? We have a board like this is a game. So, we have a board this is a computer game and we have some blocks.

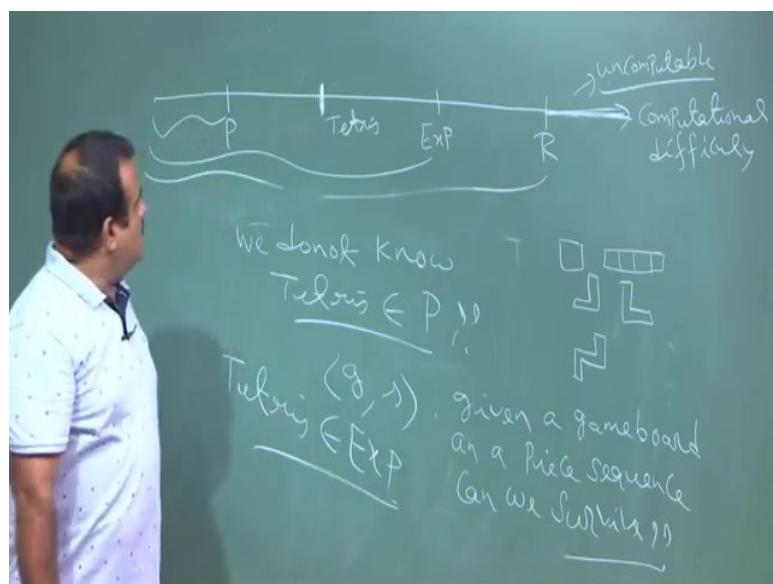
So, there are all possible blocks and then these blocks are falling and we have suppose we have given some situation like this. So, we have given some position of the board and. So, we have given the board and we have given a sequence of blocks.

Now we have to decide whether we will survive or not, whether we will stuck or not either we will survive or not. So, if there is a matching then this will have this something like that. So, when we will not survive, if it starts the top of the board. So, the problem tetris is → given this  $(g, s)$ ,  $s$  is basically a sequence of these blocks. So, given a board given a game board and a piece of sequence, I mean we have a this blocks, we have these shapes; can we survive? This is the question this is the problem.

We have seen this is a computer game this is called tetris game. So, this is hard. So, there is no polynomial time algorithm to solve this, to answer this question whether we survive or not. So, this is hard.

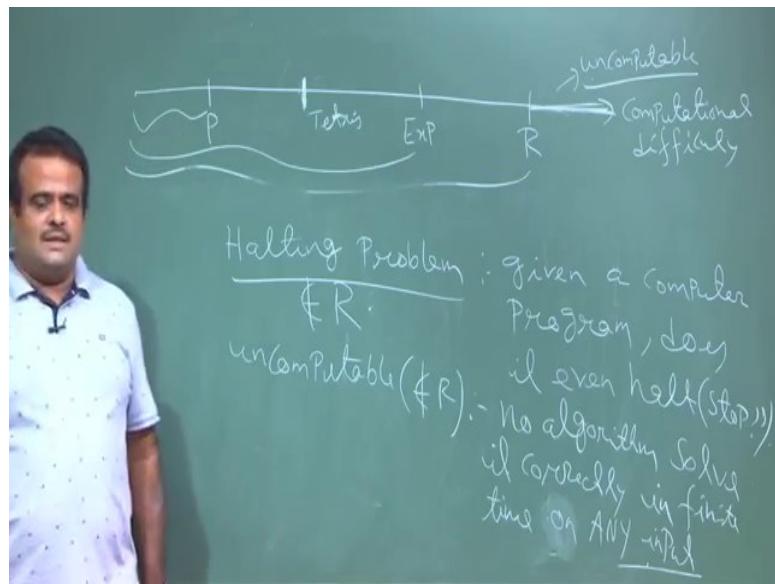
So, tetris says this is hard. So, this is another example, but we do not know whether this is belongs to P or not I mean we do not know whether this is belongs to P or not there is no algorithm. So, far whether this can check that given this whether we can survive, that is we do not know.

(Refer Slide Time: 15:11)



So, this we do not know. So, so this is another example of the exponential time algorithm. So, tetris is suppose here tetris. So, we do not know I mean this is exponential time algorithm, but we will see I mean it is also n we have not defined NP we will talk about NP so. So, now, let us talk about R some example of the problems which are say not in R say for example, halting problem.

(Refer Slide Time: 16:06)

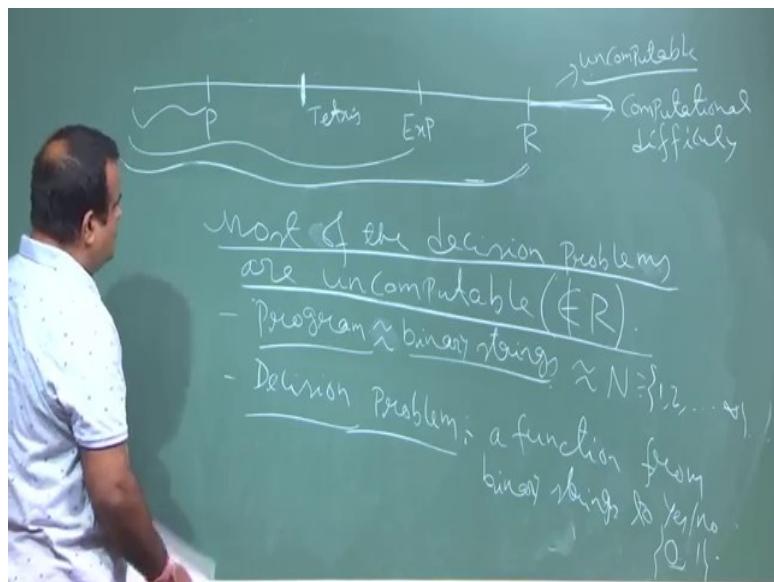


So, what is a halting problem? So, you have given a computer program and the question is does it ever halt. So, given a computer program the question is does it ever halt stop actually. So, this is the problem, problem is we have given a computer program the question is does it ever stop. So, there is no algorithm to solve this problem in finite time for all input there may be for certain input we can say the answer is yes or no, but in general for all input it is there is no such algorithm, we can say that the answer of this question. So, that is why this does not belongs to  $R$  so; that means, there is no algorithm which can tell given a arbitrary program whether it will halt or not in a finite time.

Obviously, in infinite time we can say because we can run the code and it may go up to infinite time. So, at infinite time we have the answer whether it will, but for finite because for infinity we can run the. So, we can run the code, we can run the program and then we can see so, but there is no program. So, this is basically uncomputable because this is not in  $R$  so; that means, no algorithm there is no algorithm solve it correctly infinite time for any input any. There may be certain input we can say yes, but in general for an arbitrary input there is no algorithm which can say that. So, this problem is not belongs to  $R$ . So, this problem is uncomputable problem. So, this this problem may take in general infinite time.

So, now our claim is most of the decision problem is uncomputable, most of the decision problem does not belongs to  $R$  so.

(Refer Slide Time: 19:46)



So, this is our next conclusion. So, what is decision problem? So, like we have discussed the negative weight cycle problem. So, this answer is yes or no. So, this program will give an program computer whether it will halt or not. So, this is called decision problem. So, you have two answer yes or no. So, every problem has a decision version of it, most of the problem has its decision version. So, most of the decision problems are uncomputable; that means, not in  $R$ .

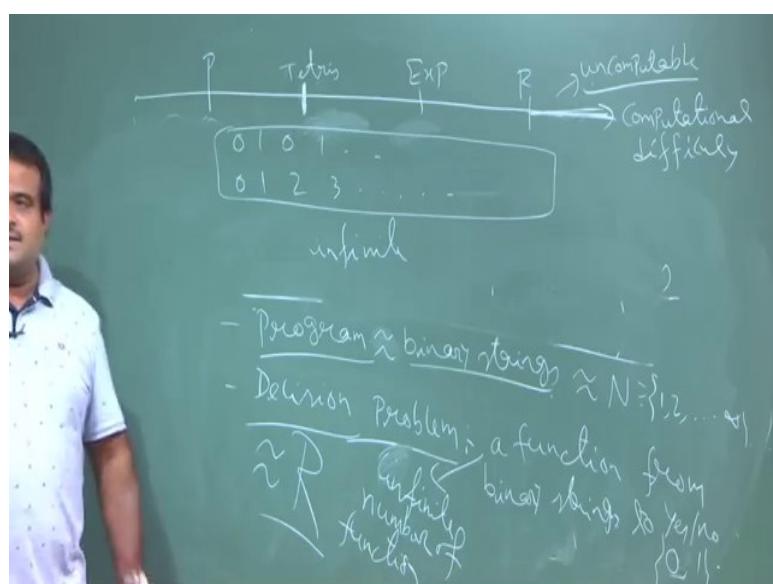
So, what is the decision problem? Decision problem means the answer is yes or no so; that means, does this program halt yes or no. So, given a board play board and a sequence of this sequence in tetris whether we can survive yes or no. So, given a position of the chess board whether white wins or not. So, this is all our decision problem. So, most of the problem has a decision version of it. So, we want to see why this is true I mean almost all of the decision problem is does not belongs to  $R$ .

So, why it is true? Because, if it is in say if we can be solve in finite machine. So, there is a program, computer program. So, program basically. So, program either in C C++ Python anything, but ultimately it will convert into ASCII and ultimately it will convert into machine code. So, machine code is basically zero-one bits because computer can understand only the 0 1 bits. So, it is ultimately converted into 0 1 bits. So, basically it is a binary string. So, this is basically a binary string. So, every program is ultimately converting into binary string.

Now, if you have a binary string 0 1 bits. So, that is basically a natural number. So, by every binary string we can think of set of natural number; natural number means 1 2 3 4 this set this is basically said 1 2 3 4 up to infinity. So, every binary string can be converted into natural number. So, every program is basically converting into a natural number. So, if we have a program it is basically a natural number there is a one to one correspondence. Now we would like to see the decision problem how we can see the decision problem. So, decision problem is basically it is a function from binary string to 0 1 because we have a this we have a problem we have to say yes or no. So, the it is a basically a function from binary string to yes or no; that means, yes or no; that means, 0 or 1 we can say yes is 0 and no is 1.

So, this is a function from a binary string to yes or no. Now how many such function is possible. So, that we have to see. So, how many such functions are possible.

(Refer Slide Time: 24:38)

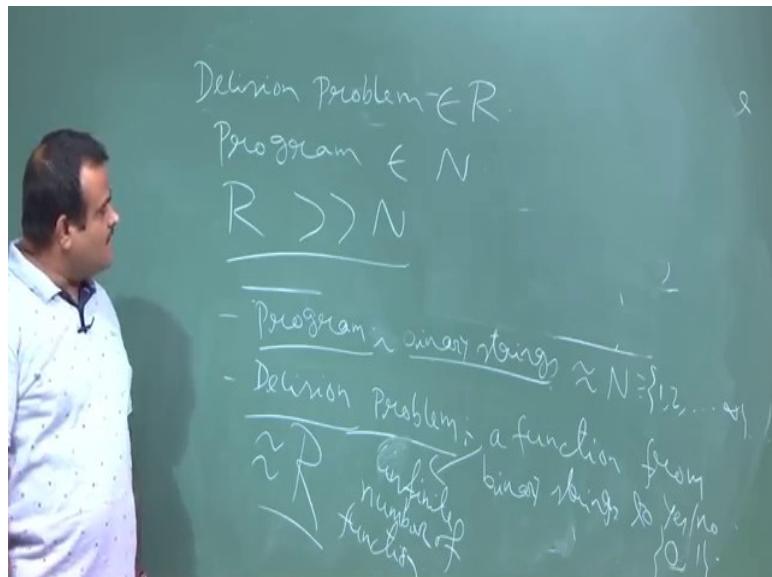


So, basically we will keep this, this is R. So, basically we have given a string and then we are converting into binary. So, this is a function form a binary string to 0 1. So, how many such function is possible? In another word we can say we can consider this as a stable. So, we have all possible binary string; binary string can be mapping from to the natural number, these are all possible binary string then we have a say 0 1 0 this is one functions.

So, how many such function are possible? If there are n bits and  $2^n$  because each of the field can be filled by n ways like two ways. So,  $2^n$ , but there are n possibilities I mean this is capital N. So, there are infinite number of bits. So, this is basically infinite. So, there are

infinite number of such functions are possible. So, this is this is basically R set, set of all possible real numbers. So, this is a uncountable remaining infinite set.

(Refer Slide Time: 26:29)



So, our decision problem is belongs to R set and our program computer is belongs to N set. So, now, R is far greater than N so; that means, we have very less number of program for our solution, because each program will give a solution of a particular problem and we have uncountably many problems and we have only countably many programs exist. So, there are many problems which are unsolvable, because this set is very much bigger than this set. So, there are many problems which are basically unsolvable.

So, this is the proof sort of where this is basically telling us the decision problem is uncountable. So, this is the bad news. So, there are many problems I mean, there are many problems which cannot be solved by the computer program and; that means there is no finite number of step we can solve this. So, these are uncountable these are beyond R because our computer problem is basically a given problem can solve one particular given program can solve one particular problem. So, this set is n this set is R this is very big set. So, that is the bad news we have many problems which is unsolvable.

So, we stop here, in next class we will discuss the more notion like np, np hard, np complete. We will continue in the next class.

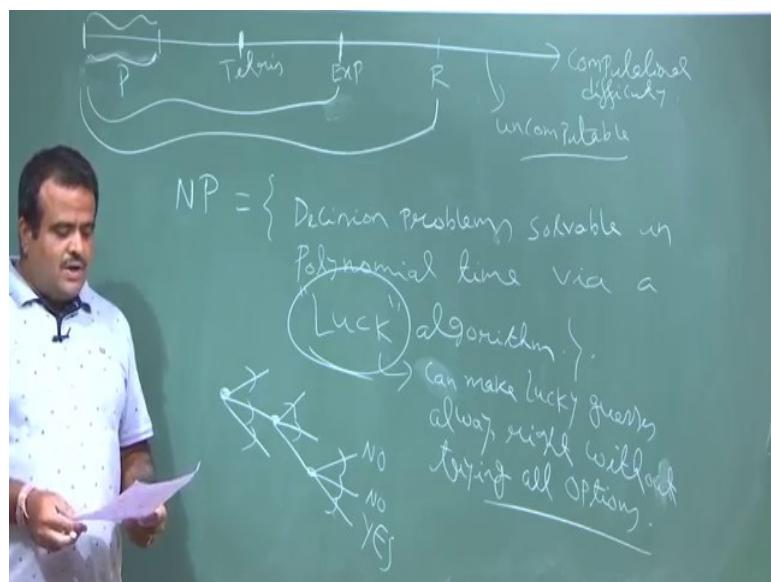
Thank you.

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

**Lecture – 60**  
**Computational Complexity (Contd.)**

So we are talking about computational complexity. So, in the last class, we have discussed 3 complexity class P, EXP and R.

(Refer Slide Time: 00:28)



So, P is the set of all problems which can be solvable by polynomial time and EXP is the set of all problem which can be solvable by exponential type. So, P is a subset of EXP and R is the set of all problem which can be solved in finite time I mean and after are all the problems is basically uncomputable problem. So, the here all the problems are uncomputable.

So, this is basically we have seen most of the decision problems are basically uncomputable. Decision problem means whether answer is yes or no I mean whether given a chess board we can tell white win or not. So, this is a decision problem given a directed graph whether there is a negative cycle. So, this is it; this is a decision problem answer is yes or no, but this is in p that detects design detection of in negative cycle.

Now, there are problem like Tetris; Tetris problem is basically this is also a decision problem. So, given a board and given a sequence of the block, now the question is whether we can

survive or not; so, these sequence are falling sequentially. So, whether we can survive or not either we can move this or this. So, these are the complexity class we have discussed now we talk about another class which is called N P that is non deterministic polynomial; NP. So, what is NP? This is basically set of all decision problem which are solvable in polynomial time via a lucky algorithm. So, what do you mean by lucky algorithm?

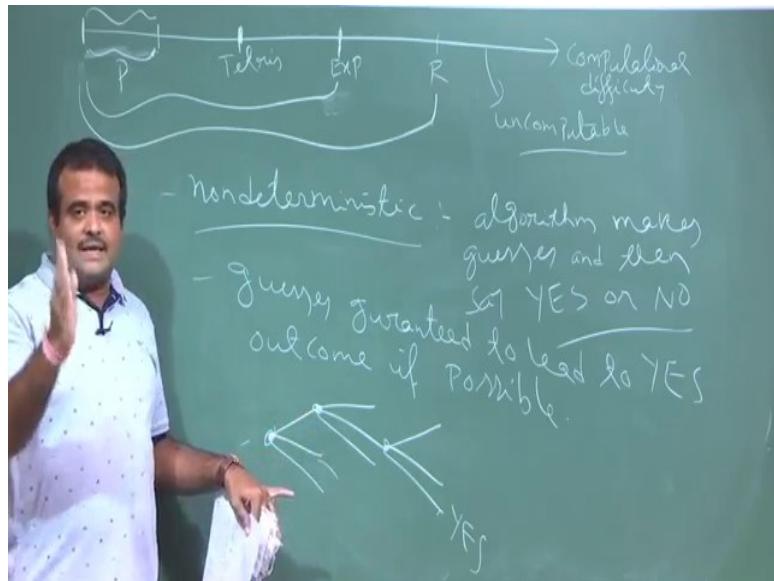
So, there are many possibilities from a starting point. So, say this is a decision problem. So, there are many possibilities over here. So, from here we can go like this. So, in the dynamic programming problem we explore all the possible ways now. So, from here say we are going like here, here, here, these are the possible moves. Now suppose this is a decision problem in some. So, this is no and if we have a yes in summary if we have a no everywhere then whatever move we'll take whatever path we choose will the decision will be no, but if there is a yes, then, every time we choose a path and that is magic, I mean we choose a path and that is called lucky path that will leads us to yes. So, we will not explore all the paths like in dynamic programming we will choose one path at each time and that will lead to that yes and that is the lucky choice and by magic we will always get the lucky choice I mean we always get the lucky part. So, this is a fair in magic you can say. So, so that is the lucky algorithm.

So, lucky algorithm means we can make lucky guesses. So, every time we guess this, then we guess this then we guess this and this is not random I mean we are guessing and that is happened to be a lucky guess. So, that is basically a magic. So, can make lucky guesses always right without trying all option like in the dynamic polling problem we are trying all option, but here we are just choosing all option and that is the good guess by guessing that my magic sale with without trying without trying all options. So, this is what is referred as lucky algorithm. So, we are not going for all path we are picking one path and that is happen to be a lucky choice; that means, that will leads us to the answer yes so; that means, it is the lucky choice of that.

So, this is called a non deterministic model. So, that is why it is called NP; so, non deterministic polynomial. So, this is a polynomial time algorithm, but non deterministic way it is not deterministic I mean by magic I mean we choose a lucky algorithm. So, lucky algorithm means we make the lucky guesses always without trying for all options. So, that is called non deterministic model.

So, let us write that. So, N P is non deterministic polynomial time.

(Refer Slide Time: 07:01).

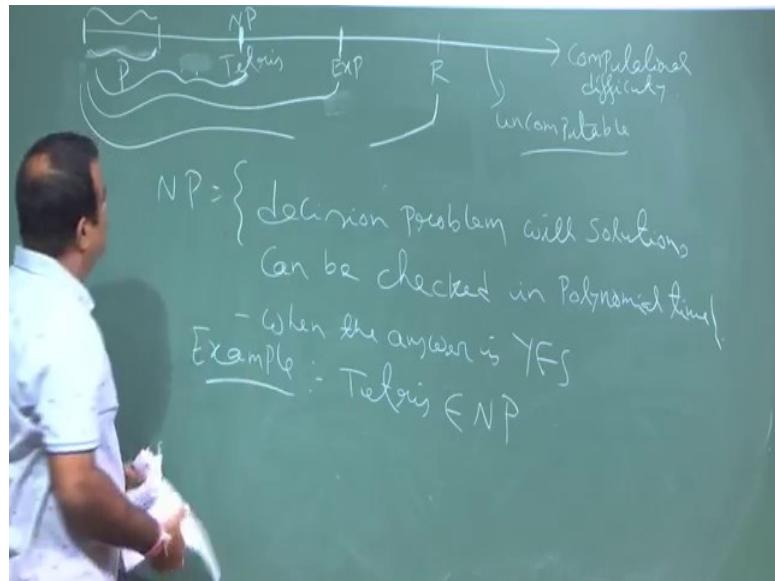


So, deterministic model is algorithm makes guesses and then say yes or no. So, this model we cannot be like a real computer, but theoretically you can study this model there is no real computer which for which we can build this model because we cannot I mean what is the guarantee that we will be lucky always. So, this model is not realistic model in the sense we cannot build this in a real computer, but theoretically it is we can have a theoretical study on this.

So, this means guesses are guaranteed it is not say random guess. Guesses guaranteed to lead to yes outcome if there is yes lead to yes outcome if possible so; that means, then suppose we start from here there are few possibilities then suppose we start here also few possibilities suppose from here also few possibilities suppose this leads to a yes I this is yes then. So, every step every path every step, we choose one path not all possibilities. So, here also we have some moves.

So, this is our guess and this guess is guaranteed in the sense we will always choose this path which will lead to us yes. If there is no yes then there is no I mean then the result will be no, but if there is a yes if possible then our guess is always correct. So, this is sort of magic. So, no realistic computer can guarantee that, but theoretically this is ok. So, this is basically the N P class. Now this is another definition of N P classes another way to view the N P classes.

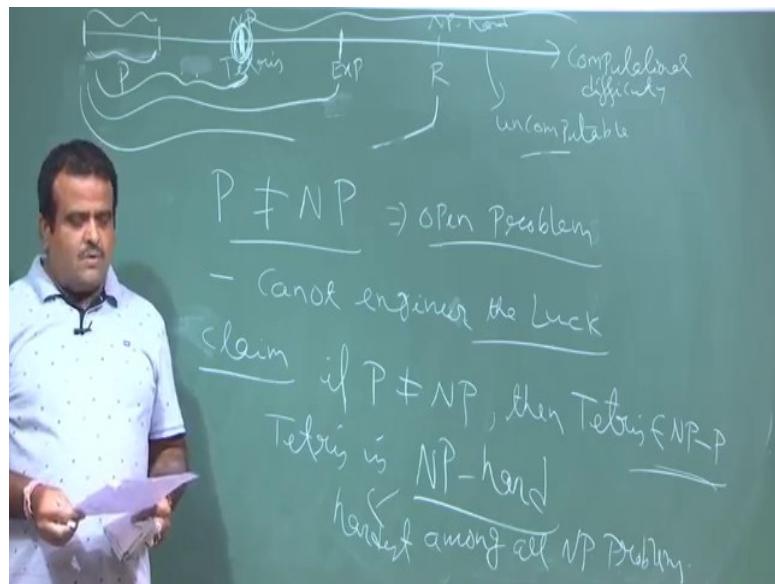
(Refer Slide Time: 09:51)



So, it is a set of all decision problem with solution that can check in polynomial time. So, when the answer is yes, we can prove it and we can check the proof in polynomial time. So, this is the another way to view the N P class. So, N P is non deterministic polynomial time. So, one example of N P is Tetris; Tetris belongs to N P. So, we can have a non deterministic algorithm so; that means, we can always guess and we can reach to a answer that whether we survive or not.

So, this is basically our N P class. So, we can just write this up to this is our N P and Tetris is a is belongs to because we can very well have a non deterministic algorithm non deterministic algorithm, I mean always we will choose a by magic whether we can fit this way these way other ways we can choose a option I mean always we can have a great guess algorithm which can lead to us whether we can survive or not. So, now we want to tell something about this N P . So, this is basically up to this it is NP. So, Tetris is belongs to NP.

(Refer Slide Time: 12:58)



So, now there are some conjecture like  $P \neq NP$ . So,  $NP$  is a harder problem right,  $NP$  is polynomial time it is a problems which can be solved by polynomial time, but with the help of a lucky algorithm. So, non deterministic algorithm. It must be a sort of a conjecture this is the open problem million dollar problem if you want to be famous you can solve this problem this is still a open problem I mean our intuition is this should not be equal set. So, people have tried whether equal whether people have tried whether they are not equal, but there is no such proof so far  $P \neq NP$ , but this is a conjecture that we have this.

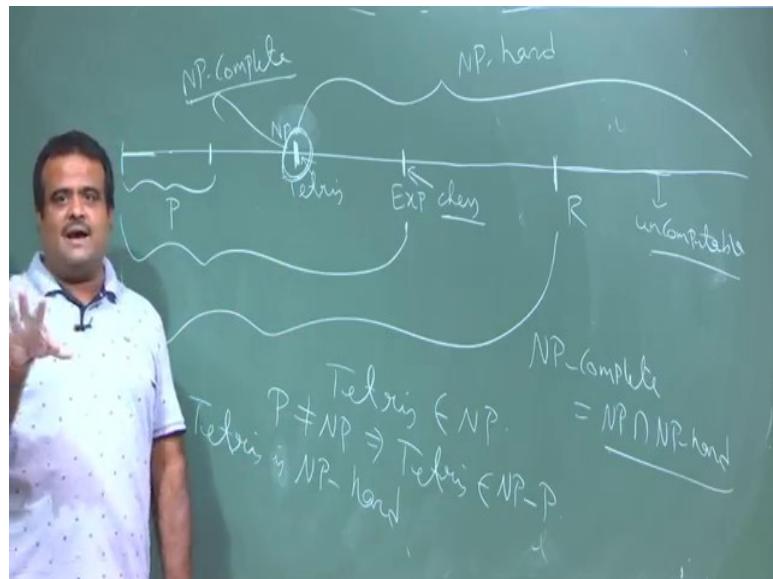
This matrix should be the bigger set of problems than  $P$  another way is to said is  $P \neq NP$  plusif we believe this that another way to say this we cannot engineer the guess so; that means, we can believe one case, but we cannot guarantee the guess. We cannot develop the case I mean.

So, now our claim is if  $P \neq NP$  then the Tetris belongs to  $NP - P$  So, if  $P \neq NP$  then the Tetris belongs to. So, this Tetris is the hardest problem our hardest problem in  $NP$  and. So, this is called this is why it is called Tetris is  $NP$  hard as hard as every  $NP$  problem. So, it is the hardest problem among all  $NP$  problems. So, this Tetris is  $NP$  hard. So,  $NP$  hard means it is the hardest problem among all  $NP$  problem.

So, this is called  $NP$  hard so; that means, from here all the problems are  $NP$  hard and Tetris is the is here. So, Tetris belongs to this and this is called  $NP$  complete problem. So,  $NP$  complete means which is basically intersection of  $NP$  hard and  $NP$ . So, which are the most

hardest problem in NP? So, these are called the NP complete, but NP hard is the all the problems which are basically harder than given any NP problem.

(Refer Slide Time: 17:43)



So, this is the line of computational difficulty; the line starts from here. So, here we have all pop easy problems and here we have Tetris and here we have exponential problem and this is basically one problem is chess; chess board n by n chess board and here we have r which is basically set of all problems which can be solvable in finite number of times and after all these problems are uncomputable like most of the decision problems are uncomputable we have proved that in the last class.

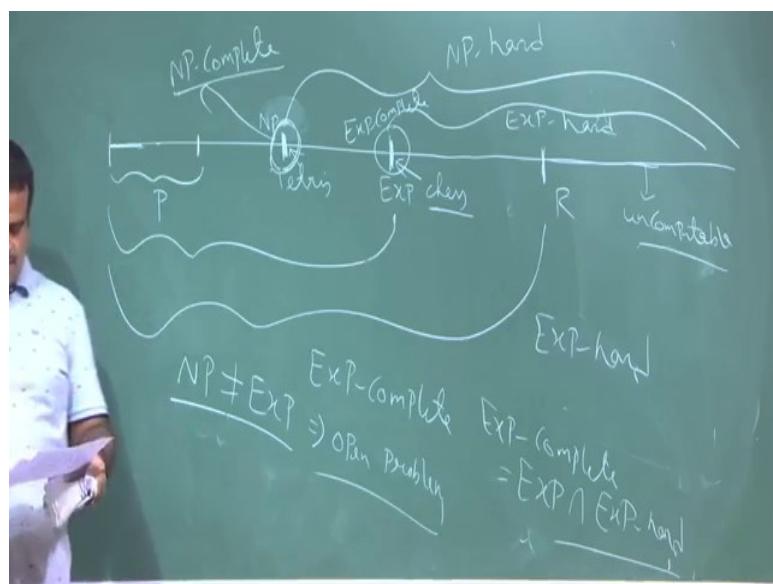
So, we have seen Tetris. So, this is basically NP this is NP and we have seen Tetris in NP and we assume if we use the conjecture  $P \neq NP$ . If  $P \neq NP$ , then Tetris must be the one of them which is in NP not in P because we are not having any polynomial time algorithm for this decision problem. So, then if this is true then Tetris must belong to NP - P otherwise if we have  $P = NP$  this is P this is NP if they are equal then Tetris must be sitting here it is the hardest problem because we have no polynomial time algorithm to solve this.

So, this is the hardest problem. In fact, Tetris belongs to NP hard; NP hard means set of all problems which are from here to here. So, they are basically NP hard so; that means, they are as harder than any NP problem. So, these are all problems which are basically harder than any NP problem. So, it is basically as hard as every problem belongs to NP and then we

have seen that if this is NP complete; NP complete means NP intersection NP hard this is NP complete. So, this is the collection where these are NP problem, but these are as hard as all given NP. So, these are as much hard as any given NP. So, these are all NP complete problem and Tetris is one of the example of NP complete problem. So, there are many NP complete ; Tetris is one of this example.

Now, similarly from here to here, these are exponentially X hard and we define exponential X complete.

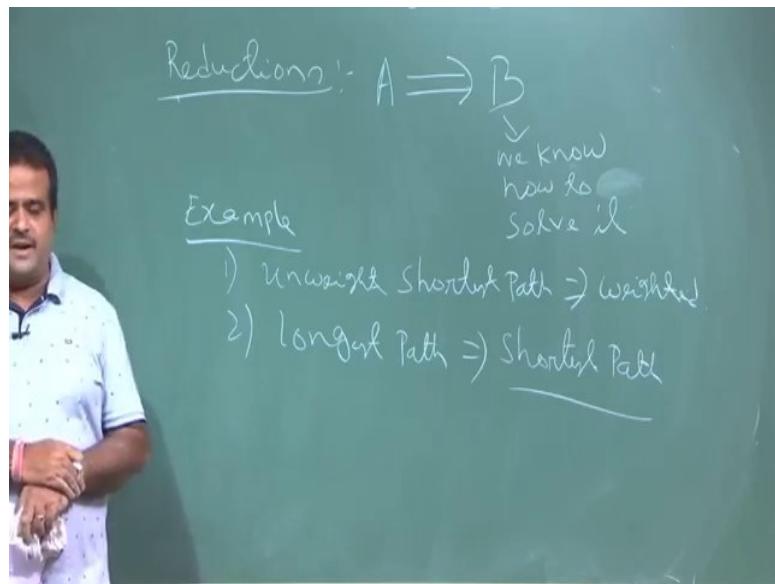
(Refer Slide Time: 22:04).



So, again this is a conjecture NP is not equal to X this is a conjecture this is another conjecture this is also open problem. So, NP is not equal to EXP. So, then we define this is hard and then we define if Xcomplete is basically EXP. So, then EXP hard means x hard means this is the set of problem which is as hard as any given x problem. So, then that is called X hard and this is the set all problems that are X hard from this set and if we take the union the intersection X hard then this is called X complete and chess is one of the example of x complete chess is one of the example of X complete.

Now, we will talk about reduction.

(Refer Slide Time: 24:19)



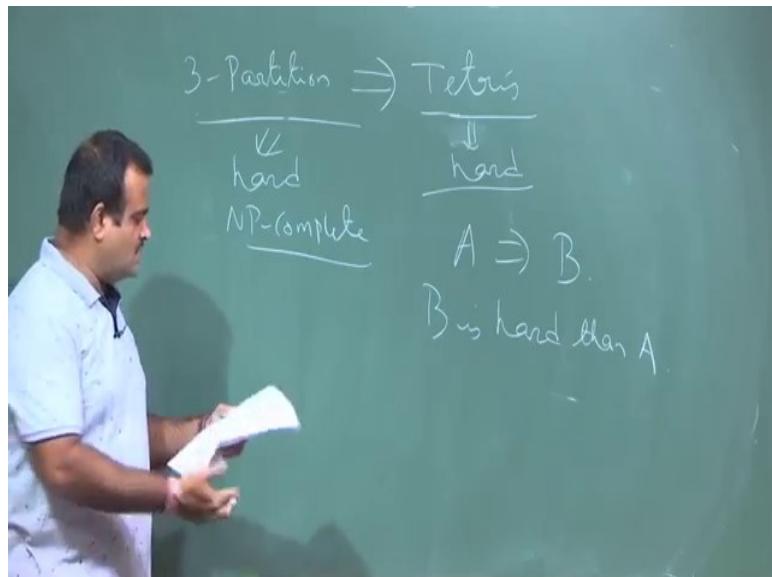
We reduce one problem to another problem mostly we reduce many problems to the graph problems and then we know the solution of this graph problem. So, that solution will give us the solution from this problem. So, this is basically the reduction this is a design technique. So, basically what is this? So, basically we have a problem A; we are reducing this problem to another problem B for which we know the solution; we know how to solve it.

Then in order to solve A, if we can reduce A to B another problem B which we know the solution which we know the program to solve it, then we know the program we know the solution of A because we have the solution of B. So, this is some example of reduction we have done. So, like if we have an un-weighted graph shortest path. So, this is the problem we have a graph on the weighted graph and you have to find the shortest path from one node to another node. So, we know the weighted graph shortest path.

So, what we do we reduce this we convert this problem to the weighted graph problem. So, this is basically we can put the weight of each vertex is weight of each edge is one. So, this is basically we can convert this into weighted graph that is it. So, this is one example; another example could be longest path . So, we have given a directed graph we need to find the longest path. So, we know the shortest path. So, what we do we just convert the weight into negative then this will be the basically is the shortest path problem and then we have the solution for the shortest path problem. So, this is a design technique. So, we basically convert this into a known problem for which we know the solution.

So, now we just talk about how we can use this reduction to prove something which is called N P completeness like.

(Refer Slide Time: 27:04)

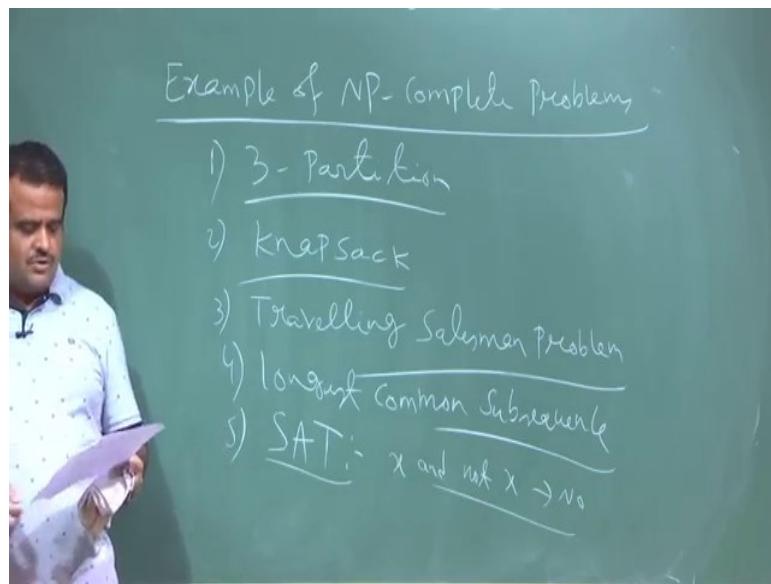


So, say for example 3 partition problem can be reduced to Tetris problem. So, what is the 3 partition 3 problem we have given a set we have given the numbers integers n numbers. So, the question is whether we can group this into 3 groups such that some of the each group is same. This is the 3 partition problem we can partition this group into 3 groups. So, that sum of each group is same and this we know this is a hard problem. So, this is N P complete problem.

So, now when we use the reduction; that means, B is harder than A because otherwise if you reduce this, if we can solve pay, then we can solve B if this reduction is in polynomial time. So, every N P complete problem. So, we can take any 2 problem and we can reduce one problem to another problem by polynomial time reduction. So, that this is one of the example. So, this must be harder than this because this we can transfer into polynomial time otherwise if we can solve these easily then we can solve this because we have a polynomial time reduction from this to this.

So, if we can solve this, then we have a solution for this because we have given this problem. So, if you can solve this problem then we got the solution of this problem, but we know this is a hard problem this is N P complete problem so, we can use the reduction to prove the N P hardness. So, since this is N P hard problem this will accompany.

(Refer Slide Time: 29:41).



So, there are some example of N P complete problems and every N P complete problems we can reduce form one to another. So, one we have seen is called 3 partition. So, you have given a set whether we can partition into 3 groups such that sum of each group is same. So, if we have given some numbers and sum of each group is same then the knapsack problem. So, knapsack problem is we have given some weight and we have given the target. So, we have to find a subset of this.

So, this is also called substance sum problem then the traveling salesman problem. So, we have given a graph and we have some edge weight on the graph. So, it a salesman's has to travel; find the shortest path that visit all the vertices from the given graph. So, it is a decision problem again whether we can have. So, yes or no; so, most of the problem have a decision version; so, this knapsack problem also. So, instead of finding this subset whether we can have a subset or not. So, that is the decision version of this problem this is also a subset sum problem then the longest common subsequence problem we have seen this problem at the starting of the graph algorithm.

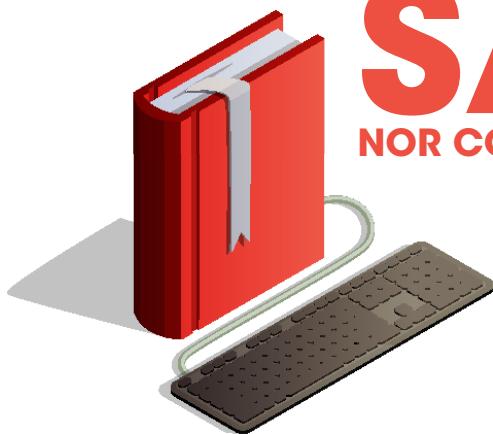
So, if we have given 2 string; 2 sequence, then this problem we have seen by dynamic programming, but if there are k string if there are k sequence then this problem is hard if k is greater than 2 then we have a problem called sat problem, this is the first problem of N P complete problem this is basically telling us given a Boolean formula is it ever true. So, suppose you have given say x and then not x. So, there is some sort of Boolean formula, we

have; now the question is; is it ever true. So, this type of problem this is the first problem came into the literature which is proved to be a N P complete problem.

So, all the problems are N P complete and other many empty N P complete problems, but we can use the reduction to one problem to another problem in polynomial time. So, if we know one problem is hard then another problem has to be hard. So, this is the N P completeness we can prove because if we can solve this problem in polynomial time. Because this problem we reduce in polynomial time to this problem. So, this is the sort of computational complexity we discuss.

Thank you.

**THIS BOOK  
IS NOT FOR  
SALE  
NOR COMMERCIAL USE**



(044) 2257 5905/08



nptel.ac.in



swayam.gov.in