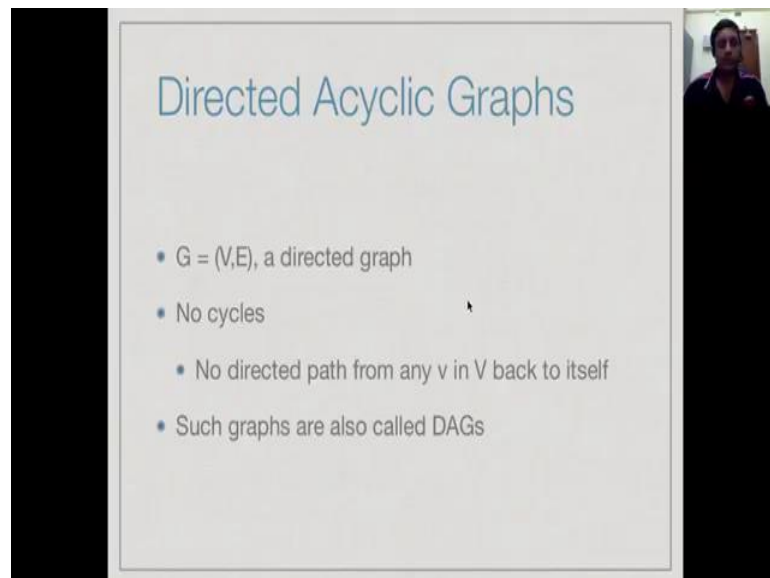


Design and Analysis of Algorithms, Chennai Mathematical Institute
Prof. Madhavan Mukund
Department of Computer Science and Engineering,

Module – 07
Lecture - 24
DAGs: Longest Paths

Let us continue to look at DAGs and in this section we will look at the different problem called identifying the Longest Path in a DAG.

(Refer Slide Time: 00:08)



Directed Acyclic Graphs

- $G = (V, E)$, a directed graph
- No cycles
 - No directed path from any v in V back to itself
- Such graphs are also called DAGs

So, recall that the directed acyclic graph is just a directed graph in which there is no directed path from any vertex back to itself, so it is direct and it is acyclic.

(Refer Slide Time: 00:20)

Topological ordering

- Given a DAG $G = (V, E)$, $V = \{1, 2, \dots, n\}$
- Enumerate the vertices as $\{i_1, i_2, \dots, i_n\}$ so that
 - For any edge (j, k) in E ,
j appears before k in the enumeration
- Also known as **topological sorting**

Any directed acyclic graph can be topologically ordered. If you think of the vertices has been 1 to n, you can write out 1 to n as a sequence in such a way that for every pair j, k which is an edge if j, k is an edge in my graph, then j will appear before k in the sequence. So, if you think of these as tasks it is dependencies means that I can do the task in such a way that before I do k, I would have finished it is dependence task j. So, this is called a topological sorting.

(Refer Slide Time: 00:51)

Questions about DAGs

- Imagine these are courses
- Edges are pre-requisites
- What is the minimum number of semesters to complete the programme?

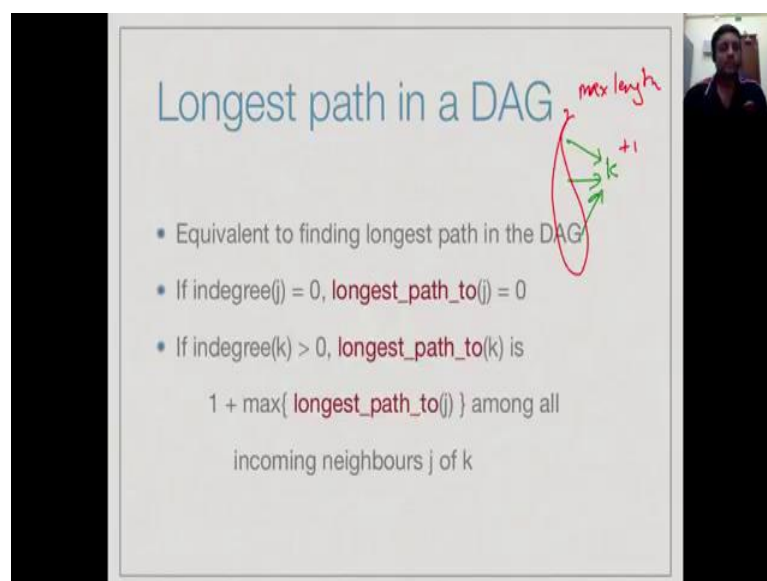
```
graph TD; 1((1)) --> 3((3)); 1((1)) --> 4((4)); 1((1)) --> 5((5)); 2((2)) --> 5((5)); 2((2)) --> 8((8)); 3((3)) --> 6((6)); 4((4)) --> 6((6)); 4((4)) --> 7((7)); 5((5)) --> 8((8)); 6((6)) --> 8((8)); 7((7)) --> 8((8));
```

So, let us look at different question about DAGs. So, supposing we have this DAG and we imagine that the vertices represent courses and the edges are prerequisites. Now, these are courses that we have to do to may be complete the degree, every course requires a semester, but we can do more than one course in a semester. So, the question is, what is the minimum number of semesters that I need to complete this program consisting of these 8 courses, with these prerequisites.

So, clearly I can start putting courses 1 and 2 in the first semester, because they have no prerequisites, so they can be done immediately. Now, having done 1 and 2 I can do 3, because 3 has only depends on 1 and 2. Similarly, I can do 4 and 5 because they depend only on 1, now 8 depends on 1 and 2 I need the material or at least depends on 2. I need 2 have done, 2 to do 8, but I still do cannot do 8, because it also requires 5, 4 and 7. So, at this point we only courses which are available at 3, 4 and 5. So, in my second semester I can do 3, 4 and 5.

Now, the only course for which all free prerequisites are satisfied is 6, for 7 requires 6 which is not been done and 8 requires 7 which is not been done. So, in the third semester I am struck with doing only one course namely 6. In the 4th semester, I can do 7 and finally, after 5 semesters I can complete this (Refer time: 02:12). So, in general we can ask this question, if I think of these as DAG is representing courses, what is the minimum number of semesters.

(Refer Slide Time: 02:24)



The slide is titled "Longest path in a DAG" in blue text. To the right of the title, there is a small diagram of a node 'k' with two incoming arrows from nodes 'j' and 'i'. A red circle is drawn around the node 'k', and a red arrow points to it with the handwritten text "max length". A green arrow points from node 'j' to node 'k' with the handwritten text "+1".

- Equivalent to finding longest path in the DAG
- If $\text{indegree}(j) = 0$, $\text{longest_path_to}(j) = 0$
- If $\text{indegree}(k) > 0$, $\text{longest_path_to}(k)$ is

$$1 + \max\{ \text{longest_path_to}(j) \} \text{ among all incoming neighbours } j \text{ of } k$$

Now, this problem corresponds to asking for the longest path in the DAG ((Refer Time: 02:28)) what we are saying is that it takes 5 semesters to complete 8, because there is a chain of dependencies, where 8 depends on 7, 7 depends on 6, 6 depends on say 3 or 4 it does not matter which one we choose and 3 depends on 1 and this chain forces us to spend 4 semesters, because it is a chain of length 4, 4 semesters before I can do 8. Notice that it is not the shortest chain, because there are shorter chains for example, 8 to 2 takes back to a vertex which has indegree 0, but this does not help us because after 2 I cannot do it. So, I must wait for everything that has to happen before 8 in order to get the job done. So, unlike other problems where we might be looking shortest paths, here we are actually interested in the longest path.

So, we can set up this problem as follows, so we can say that for any vertex which has indegree 0, the longest path to that vertex is of length 0, because I can do it immediately. And on the other hand, if I have a vertex whose indegree is not 0, then it has some incoming edges. So, I have a vertex k , so I must wait for all of these to finish and then do it. So, if I take all of these I have to take, among all of these I would take the maximum length, because that will be the last thing to finish and after that I have to do plus 1 account for k . So, if indegree is not 0 then the longest path to k has length 1 plus the maximum the longest path to all it is incoming neighbours.

(Refer Slide Time: 04:01)



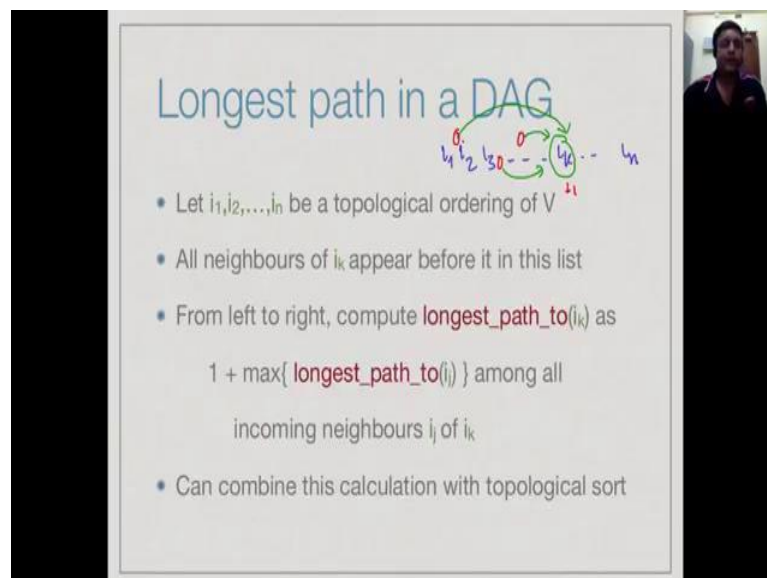
Longest path in a DAG

- To compute `longest_path_to(k)`
 - Need `longest_path_to(j)` for all incoming neighbours of k
 - If j is an incoming neighbour, (j,k) in E
 - j is enumerated before k in topological order
 - Hence, compute `longest_path_to(i)` in topological order

So, therefore in order to compute the longest path to k , I need to compute the longest path to all its incoming neighbours. Now, if we have arranged the vertices in topological order and we compute the longest path in that sequence, then when we get to k every incoming neighbour j will be on its left. Hence, we would already compute its longest path, so we would be able to take the maximum number of all of these and add them.

So, by sorting these vertices in topological order, I can then compute the longest path in the same order with the guarantee that when I want to compute the longest path to begin vertex, I have all the information available to do that namely all the longest path for its incoming neighbours.

(Refer Slide Time: 04:48)



The slide is titled "Longest path in a DAG". It features a diagram of a directed acyclic graph (DAG) with vertices labeled $i_1, i_2, i_3, \dots, i_k, \dots, i_n$. Edges are shown as arrows, with some labeled i_1, i_2, i_3 and others i_4, i_5 . The vertices are arranged in a sequence from left to right, representing a topological order.

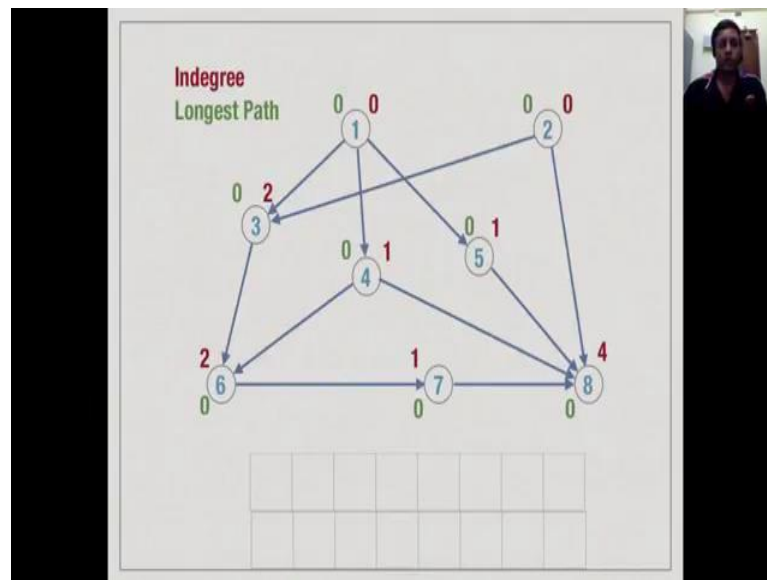
- Let i_1, i_2, \dots, i_n be a topological ordering of V
- All neighbours of i_k appear before it in this list
- From left to right, compute **longest_path_to(i_k)** as

$$1 + \max\{ \text{longest_path_to}(i_j) \} \text{ among all incoming neighbours } i_j \text{ of } i_k$$
- Can combine this calculation with topological sort

So, if I do it naively then I will write out my vertices in some topological order and now when I come to this vertex and I want to compute its longest path, I will look at in my graph all the incoming edges and they will all be from vertices which appear before. So, I can take the value here, the value here, the value here and then take its maximum and then add here. So, actually we will see that you do not need to do this backward, you can actually do it forward.

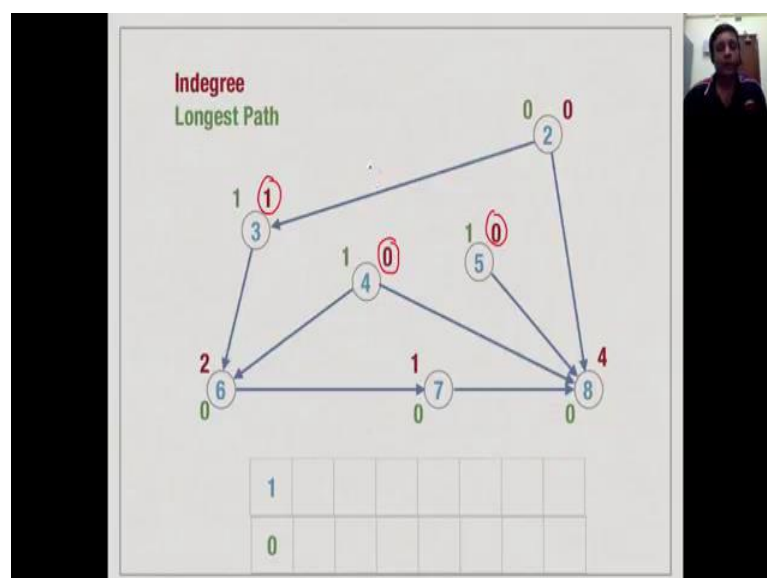
So, we can actually incrementally compute the value at i_k as you are going forwards. Because, going backwards requires you to scan this list and look for all the incoming neighbours. So, we will kind of implicitly do this longest path calculation along with topological sort side by side, as we will see in the next example.

(Refer Slide Time: 05:42)



So, here is an example in which we are going to compute the longest path as we are computing the topological order. So, as before the red numbers I given as the vertices are used for topological sort and they denote the indegrees. So, the initial indegrees are given where the initial edges in our graph and what we do is we compute the longest path incrementally by starting by assuming that the longest path to every vertex is actually 0.

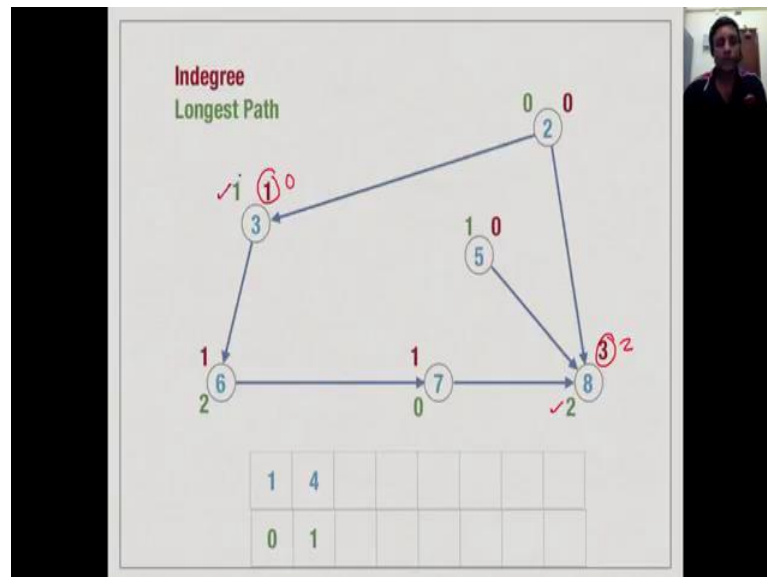
(Refer Slide Time: 06:15)



Now, when we enumerate a vertex in a topological sort what we did earlier was to update the indegrees, so this is something that we already did. But, now the additional thing that

we do is we say that well if the vertex 1 add to be enumerated before 3, 4 and 5, then among the values that I know for the incoming edges of 3, 4 and 5, 0 was the maximum in length. So, I must do 1 plus that, so these paths are at least of length 1. So, the longest path to 3, 4 and 5 is at least 1.

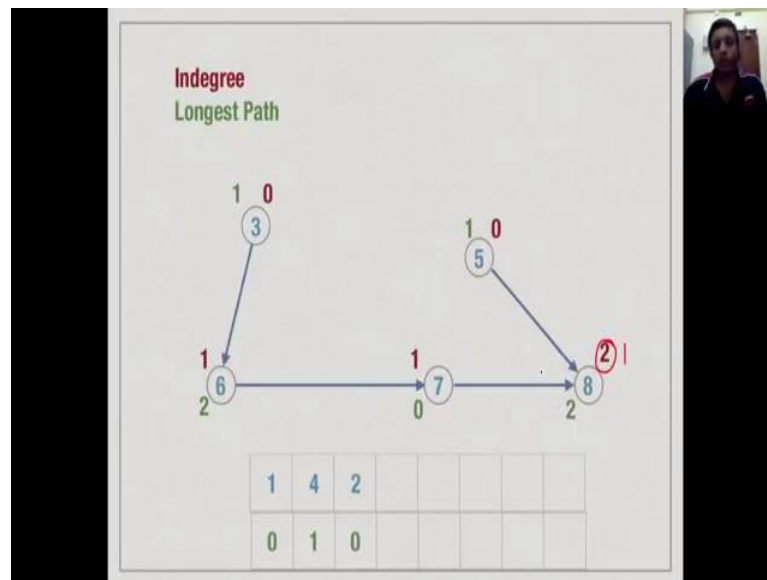
(Refer Slide Time: 06:47)



Now, if I enumerate vertex 4, then its longest path is at least 1, so therefore, the longest path to 6 must be at least 1 plus 1. Similarly, the longest path to 8 must be at least 1 plus 1 and of course, the indegrees will also reduce as before. So, the indegree of 6 goes on to 1, the indegree of 8 goes on to 3, but the longest path to 6 is now the longest path to 4 plus 1, so it is 2, it is same for 8. Now, supposing I enumerate the vertex 2, now 2 will again say that the longest path, because of 2 the longest path to 3 is 1, but it is already 1, so we do not change it.

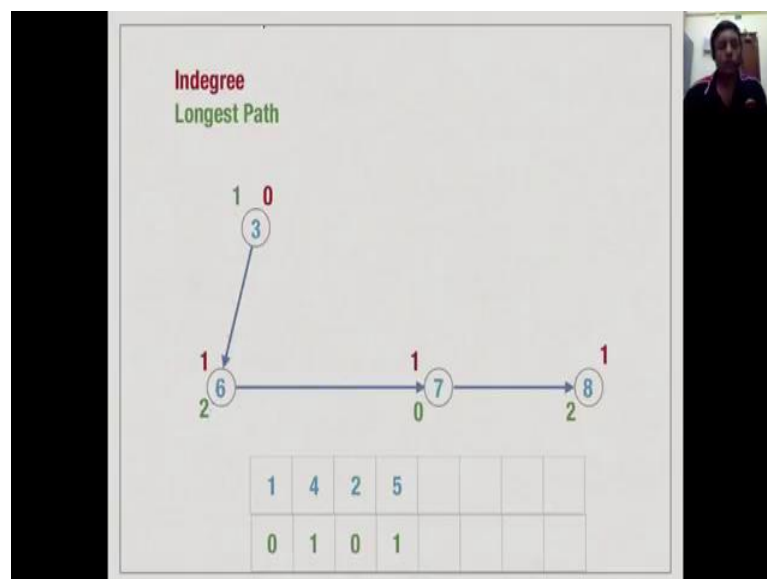
This will say because of 2 the longest path to 8 is at least 1, but we know it is at least 2, so again we do not change. So, when we delete a node, then we take basically the current value of the longest path for it is outgoing thing plus 1, the current value of the deleted node plus 1 and what is already known about at node and we keep the maximum. So, in this case this 1 will become 0, this 3 will become 2, but here there is no change and here there is no change.

(Refer Slide Time: 07:51)



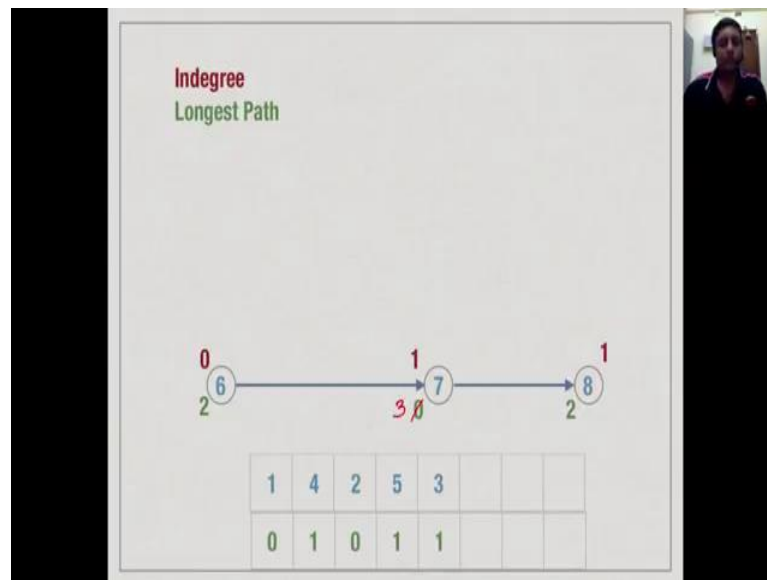
And likewise when I remove this 5, this 2 will become 1, but because 1 plus 1 is 2 and we already know that 8 has a longest path of at least 2, we do not make any change in the 2.

(Refer Slide Time: 08:05)



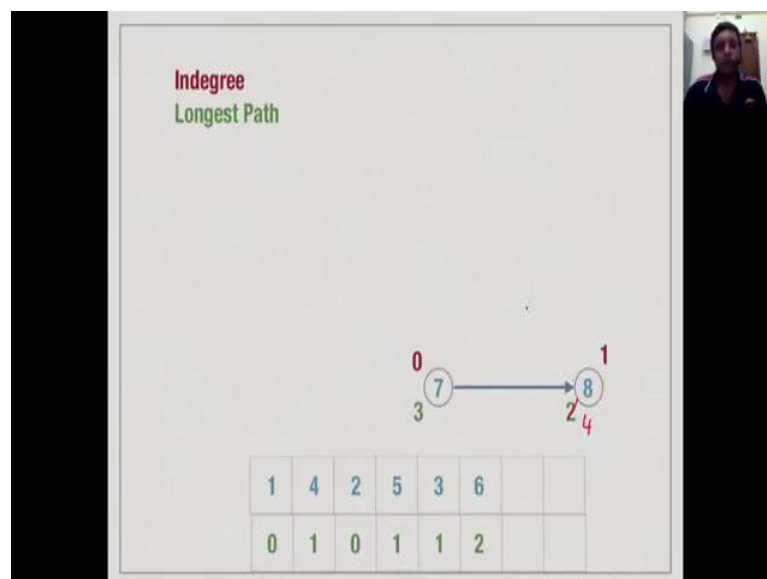
Now, when we go to 3, 3 says it has a longest path of 1, so therefore because of 3 the longest path is 6 at least 2, but we already know it is 2, so again there is no change.

(Refer Slide Time: 08:17)



Now, we shall do something interesting, so we say that 6 has a longest path of 2, 7 we so far I have believed it has longest path of 0, but through 6 it has a longer path. So, the path to 7 must now be updated from 0 to 3.

(Refer Slide Time: 08:34)



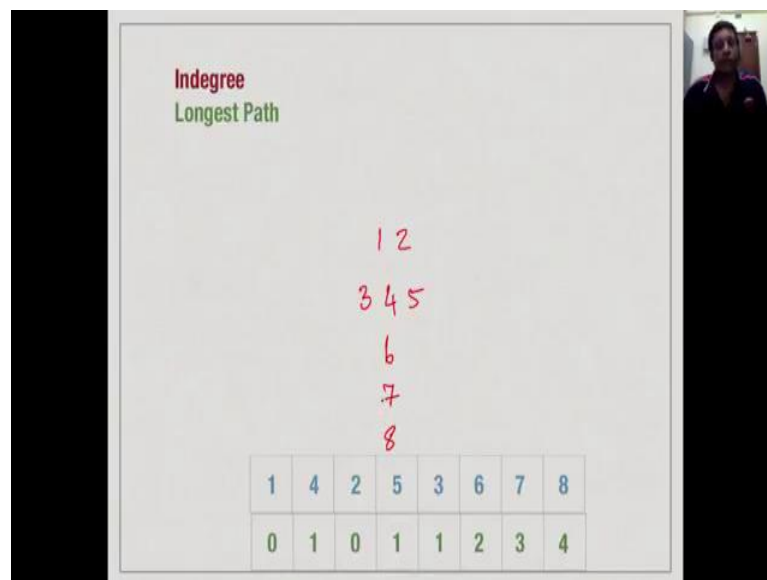
And now because of this, when we go from 7 to 8, the longest path for 8 must be updated from 2 to 4.

(Refer Slide Time: 08:46)



And now finally, this is my last vertex, so I just enumerated and I compute its longest path as 4.

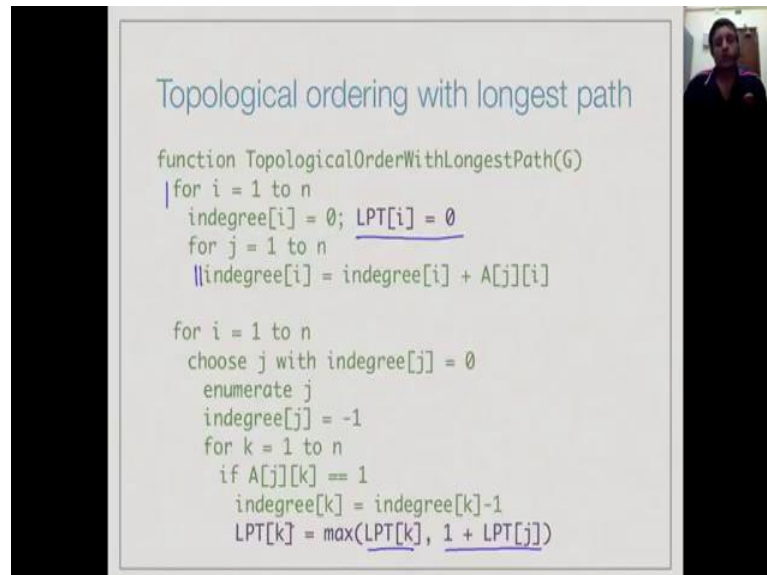
(Refer Slide Time: 08:52)



So, what this is said is the longest path is 4, now we said that we done in 5 semester, this same example with basically means that in the first semester all those whose longest path is 0, the second semester all those whose longest path is 1 and so we have the same sequence you had before. So, you initially do 1 and 2 in the first semester, then we do 3, 4 and 5 in the second semester, then we do 6 in the third semester, 7 in the fourth

semester and 8 in the fifth semester. So, this is the exactly the solution that we obtained informally in our initial example.

(Refer Slide Time: 09:22)



Topological ordering with longest path

```
function TopologicalOrderWithLongestPath(G)
  for i = 1 to n
    indegree[i] = 0; LPT[i] = 0
    for j = 1 to n
      ||indegree[i] = indegree[i] + A[j][i]

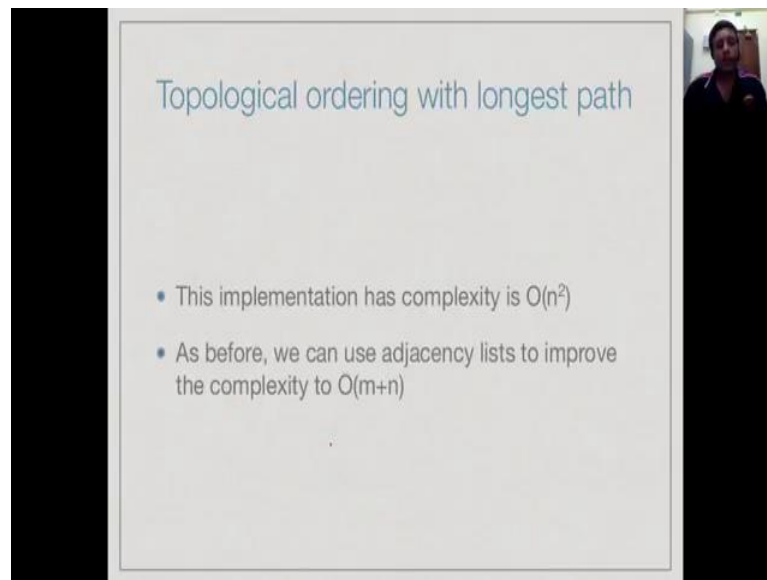
  for i = 1 to n
    choose j with indegree[j] = 0
    enumerate j
    indegree[j] = -1
    for k = 1 to n
      if A[j][k] == 1
        indegree[k] = indegree[k] - 1
        LPT[k] = max(LPT[k], 1 + LPT[j])
```

So, this pseudo code for longest path as we saw is very similar to what we did for the topological sort, we just have keep track of this extra longest path value. So, when we initialized the indegree we also initialize the longest path to i to be 0 for every vertex. So, this is we are doing again first the adjacency list version. So, we do n squared work for each vertex, we compute the indegree by looking at the column of the adjacency matrix with entry column entry i.

But, we also update, we initialize longest path to each i is 0, now when we are doing this enumeration as before we choose any vertex indegree 0, we enumerated and we update the indegree of this vertex to minus 1. So, it is no long were in contention for we chosen again. Now, for all it is out going neighbours, we update the indegree and we also update the longest path.

So, we take the longest path that we already know to that neighbour, the LPT of k and we take 1 plus the longest path for this node and whichever is larger we replace that pattern PT of k. So, it is a very simple variation of the basic topological ordering thing which allow also compute the longest path.

(Refer Slide Time: 10:34)

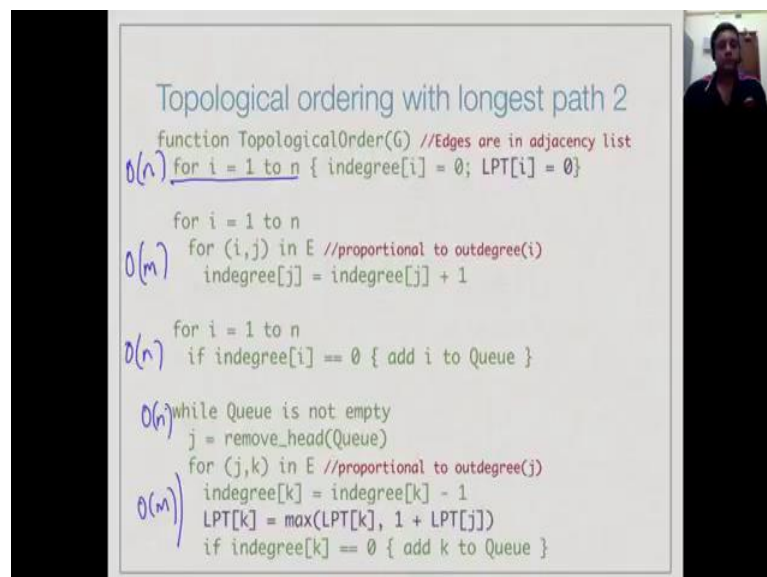


Topological ordering with longest path

- This implementation has complexity is $O(n^2)$
- As before, we can use adjacency lists to improve the complexity to $O(m+n)$

So, this has complexity order n squared for the same reason that we had found that topological sort with adjacency matrix or order n squared. Because, we have the nested loops in order to scan all the neighbours, so as before we can update this whole thing using adjacency list and a queue to make it a linear time algorithm.

(Refer Slide Time: 10:56)



Topological ordering with longest path 2

```
function TopologicalOrder(G) //Edges are in adjacency list
 $O(n)$  for i = 1 to n { indegree[i] = 0; LPT[i] = 0 }

    for i = 1 to n
     $O(m)$  for (i,j) in E //proportional to outdegree(i)
        indegree[j] = indegree[j] + 1

    for i = 1 to n
     $O(n)$  if indegree[i] == 0 { add i to Queue }

     $O(n)$  while Queue is not empty
        j = remove_head(Queue)
        for (j,k) in E //proportional to outdegree(j)
             $O(m)$  indegree[k] = indegree[k] - 1
            LPT[k] = max(LPT[k], 1 + LPT[j])
            if indegree[k] == 0 { add k to Queue }
```

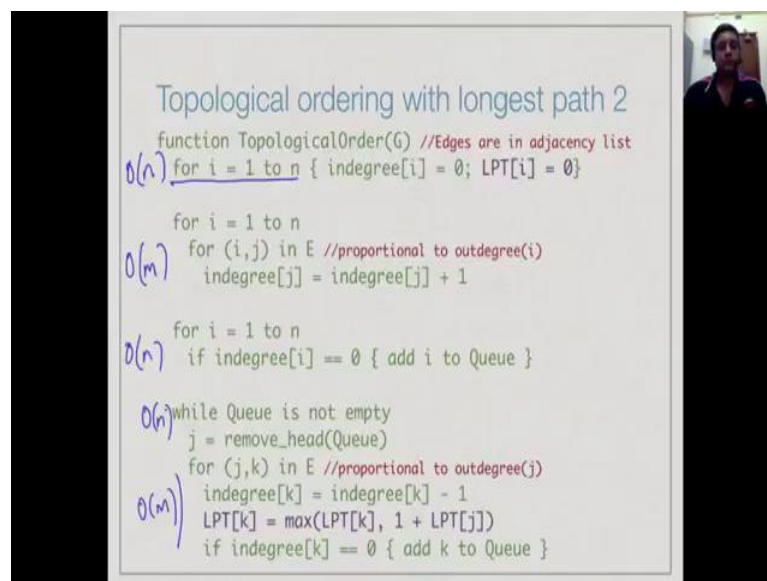
So, if you go back and look at the topological ordering algorithm, you will find that the same changes are required to make a additional compute the longest path. So, what we do is initializing is again the same we do it for the every vertex. So, this is an order n

step, we initialize both the indegree and the longest path. Then we go through all the adjacency list and together this order m step to compute the initial indegrees.

And now we have an order n step in order to setup this queue, where we will process the topological ordering. And then, we do an outer loop of order n , so this is an order n loop, because everything is going in and to the queue once and come out. So, we are going to remove from the queue n times and now because we are scanning the list across all the n iterations we are going to process each edge once. So, the total work done in this loop is going to be order n .

So, we do exactly as before we update the indegree, we update the longest path to k as maximum of the current value and 1 plus the value of the current node and if we do find that the k is now become a vertex to indegree 0 we added to the key.

(Refer Slide Time: 12:13)



Topological ordering with longest path 2

```

function TopologicalOrder(G) //Edges are in adjacency list
 $O(n)$  for i = 1 to n { indegree[i] = 0; LPT[i] = 0 }

    for i = 1 to n
    for (i,j) in E //proportional to outdegree(i)
 $O(m)$  indegree[j] = indegree[j] + 1

    for i = 1 to n
 $O(n)$  if indegree[i] == 0 { add i to Queue }

    while Queue is not empty
 $O(n)$  j = remove_head(Queue)
    for (j,k) in E //proportional to outdegree(j)
 $O(m)$  indegree[k] = indegree[k] - 1
    LPT[k] = max(LPT[k], 1 + LPT[j])
    if indegree[k] == 0 { add k to Queue }
  
```

So, DAGs are very useful because there many situations, where we want to model dependencies between various objects. And topological ordering is a canonical algorithm to list out vertices without violating dependencies, what we are seen in today is that, you can compute the longest path to the DAG and longest path to the DAG in some sense if we can list out vertices in groups, longest path in a DAG represents the minimum number of steps in order to enumerate all the vertices.

So, if we going to courses in groups, then if you want to bunch it and do things which are the same level at one time, then the minimum number of steps we need minimum of semesters to complete set of courses or the minimum number of days to complete the set of tasks is given by the longest path. Now, it is important that we have been to able to find efficiently linear time algorithm for longest path only because restricted ((Refer Time: 13:12)) DAGs.

If we look at arbitrary graphs which are not necessarily DAGs and we want to find longest path of course, if we have loops there are longest path will be undefined, because the can go around and around loop. So, if we define a longest path to be one in which we have a sequence of vertices with no duplicates. So, the length is at most 10, then arbitrary graph this is a very challenging problem and there is no known efficient algorithm. In fact, it is a part of a group of highly interactively problems which all are equivalent to each other and all believe to be very hard.

So, DAGs are a very significant subclass of graphs which have many practical applications and which admit efficient solutions for very important problems, which have not in general algorithm easily and the full class of graphs.