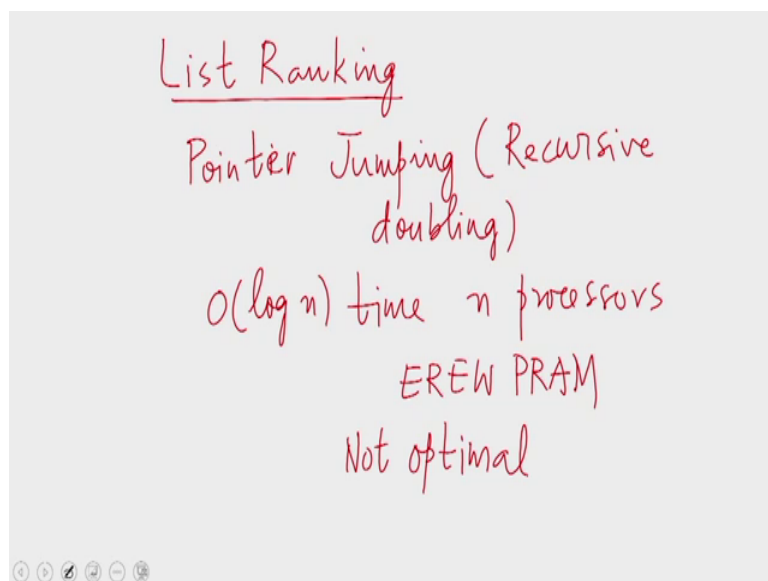


**Parallel Algorithms**  
**Prof. Sajith Gopalan**  
**Department of Computer Science & Engineering**  
**Indian Institute of Technology, Guwahati**

**Lecture - 13**  
**Description**

Welcome to the 13th lecture of the MOOC on Parallel Algorithms. Today we shall study an Optimal Parallel Algorithm for the Problem of List Ranking.

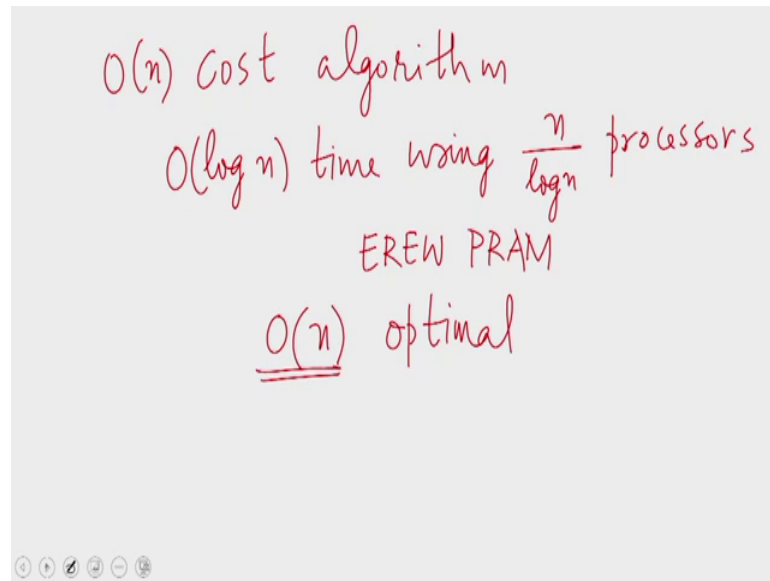
(Refer Slide Time: 00:41)



We have already seen the problem of list ranking before. In one of the earlier lectures we use the technique called pointer jumping, which is also called recursive doubling to devise an algorithm that runs in order of  $\log n$  time using  $n$  processors on an EREW PRAM.

As we discussed earlier this algorithm is not optimal. The cost of this algorithm is order of  $n \log n$ , whereas the sequential time complexity with the problem of list ranking is order of  $n$ .

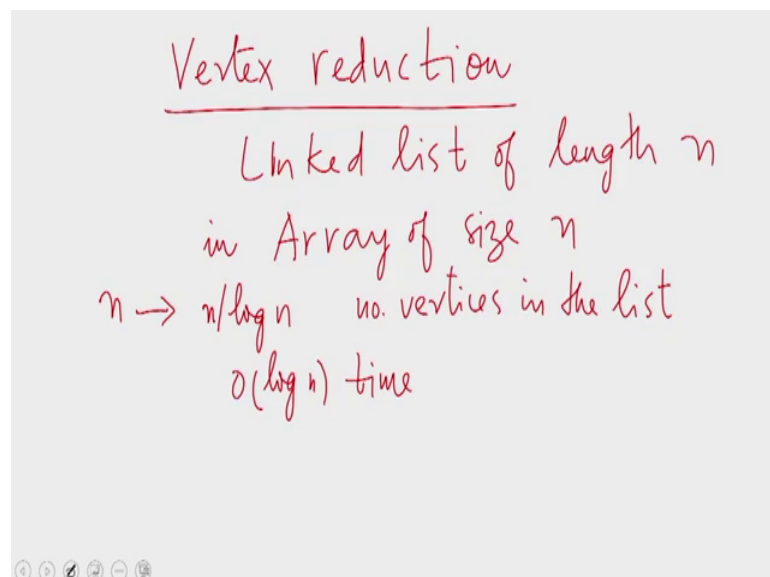
(Refer Slide Time: 01:55)



$O(n)$  cost algorithm  
 $O(\log n)$  time using  $\frac{n}{\log n}$  processors  
EREW PRAM  
 $O(n)$  optimal

So, for the problem to be solved optimally we want an order of  $n$  cost algorithm; for the problem. So, today we shall consider an algorithm that runs in order of  $\log n$  time using  $n$  by  $\log n$  processors. Again on an EREW PRAM and the cost of the algorithm would be the time processor product which is order of  $n$ , and therefore it will be; optimal the cost of the algorithm is order of  $n$  and therefore it would be optimal.

(Refer Slide Time: 03:07)



Vertex reduction  
Linked list of length  $n$   
in Array of size  $n$   
 $n \rightarrow n/\log n$  no. vertices in the list  
 $O(\log n)$  time

So, now let us see how the algorithm can be devised. Essentially the algorithm involves vertex reductions from the given list. So, we are given a linked list of length  $n$  that is

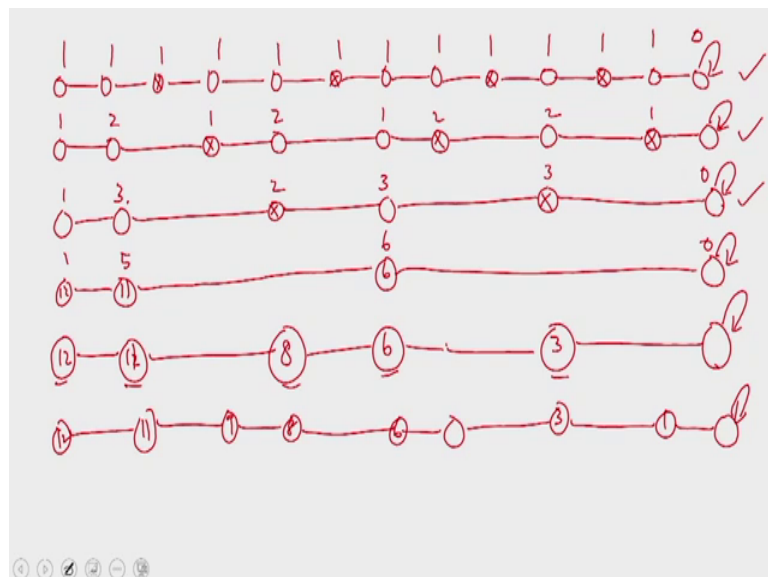
there are  $n$  nodes in the linked list. This is given in an array of size  $n$ , the classical presentation; what is required is to rank the list. But as opposed to the previous algorithm we have only  $n$  by  $\log n$  processors. So, we do not have enough processors to perform the recursive doubling.

If we manage to reduce the number of vertices in the list to  $n$  by  $\log n$  then we would have enough processors to run the previous algorithm. So, that is what we are going to do. We will reduce the number of vertices in the list from  $n$  to  $n$  by double  $\log n$ . We would like to accomplish this reduction in order of  $\log n$  time using the available processes.

So, we have  $n$  by  $\log n$  processes, with these  $n$  by  $\log n$  processors we would reduce the length of the list from  $n$  to  $n$  by  $\log n$  in order of  $\log n$  time. Once the list is reduced the number of remaining nodes is equal to the number of processors. Therefore, on the remaining list we will be able to perform the recursive doubling as we did in the previous algorithm. And then we have to reconstruct the original list along with the ranks for its vertices.

So, let us see how this vertex reduction can be accomplished.

(Refer Slide Time: 05:03)



Let us say we have a long list of people standing in front of a counter to buy tickets. So, the last node of the linked list let us say is the counter. In the pointer jumping algorithm

we initialized the last node to 0 and every other node to 1. So, we do the same thing here. So, we imagine that our linked list is a queue of people standing in front of a counter to buy tickets. The one at every single node signifies that each of these people want to buy one single ticket. The last node is the counter and therefore it has been labelled 0 signifying that it is not planning to buy any ticket.

Now, the queue is very long. Let us say in this queue we pick a number of people who form an independent set. What it means is that we do not choose two vertices that are adjacent to each other. So, we form an independent set. Let us say the people belonging to the independent set choose to stay out of the queue for the time being. Let us say they are going out for a tea. Then each of these persons would tell the person behind him to buy a ticket for him.

So, let us say this person in the queue asks the person behind him to buy a ticket for him. So, we find that the first element is the first person is still planning to buy 1 ticket, but the second person is planning to buy 2 tickets: one for himself and one for the person who went for the tea. The fourth vertex is still buying 1 ticket, but the fifth vertex is buying 2 tickets now: one for himself and one for the person ahead of him. So, likewise we update the labels of the vertices.

So, the length of the list has now reduced, but some of the labels have changed. The label signifies the number of tickets that a person is planning to buy. After a while again let us say another independent set chooses 2 go for tea, and each of these persons will tell the person behind her to buy tickets for them. So, the first vertex is still standing for herself, the second vertex now has to buy 3 tickets: two that he was planning to buy earlier and one extra ticket for the new person in front of her who is going out. So, the second vertex is now planning to buy 3 tickets. This vertex is still buying two vertex the 2 tickets, this vertex has to buy 3 tickets; its earlier one plus the new two. And this vertex is also buying 3 tickets.

So, the length of the list has further reduce. If another independent set now chooses to go for tea the length of the list will reduce again. Here it is 3, here it is 5, this node has to buy 5 tickets now; the three initial tickets plus the two for the person who went out. This node has to buy 6 tickets. Let us say at this point we issue the tickets. The tickets are nothing but the ranks in this case.

So, we run the notes from the tightened. So, this note gets a rank of 0, this note gets a rank of 6, and here the rank is 11: 6 plus 5 - 11 and here it is 12. So, this is how the notes are ranking now. So, all the tickets have been issued. Now the people who when for tea are coming back in, but then they have to come back in the opposite order in which they left; that is because every person who leaves tells the person behind her to buy tickets for her that scheme should not be confused. Therefore, we should splice the vertices in the same order. So, this vertex is now being spliced in.

So, this vertex is between two nodes that have been ranked 6 and 12 respectively this node was planning to buy 2 tickets. So, what it means is that between these two nodes, we have ranks 7, 8, 9, 10, 11 and 12 that is a total of pardon me this is 11. So, there are five ranks: 7, 8, 9, 10, 11; out of these five ranks the first two ranks of a this node which means this node will get a number of 8; that is tickets 7 and 8 are handed over to this vertex. And here we have this vortex coming in and it is handed over the tickets 1, 2 and 3.

So, this node now holds tickets 1, 2 and 3, this node now holds tickets 4, 5, and 6; tickets 7 and 8 are held by this node; 9, 10, 11 our held by the node. And the last node holds only ticket 12. So, the list has been recreated to this point, and all the nodes in this list have now been ranked. If you do a prefix song on this list from the right end you will get exactly the same values 0, 3, 3 plus 3 - 6, 6 plus 2 - 8, 8 plus 3 - 11, and 11 plus 1 - 12 which is exactly the values that we have on the list now.

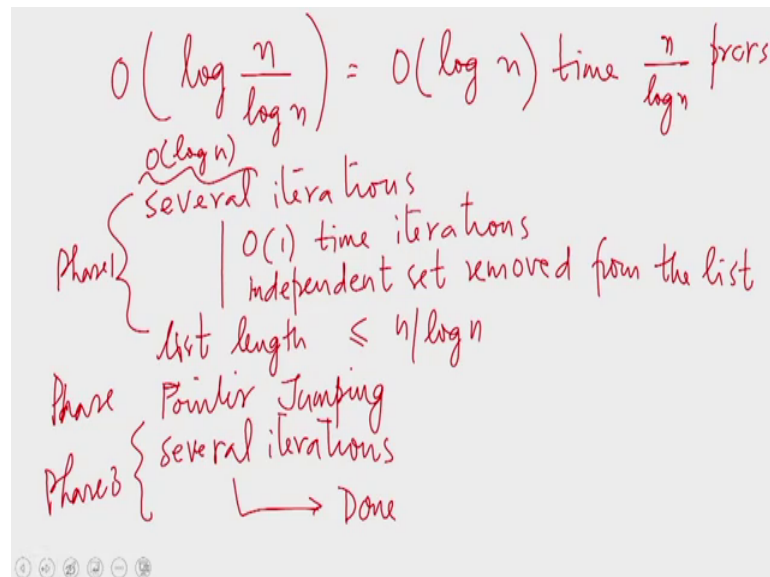
Now we can consider the second set that had gone for tea and asked them to stand back in the queue. And they can now take the tickets that they were supposed to buy. That is had they not left the queue they would have had the tickets with them by now. So, those tickets now we can distribute by taking from the people who are already in the queue. So, this node has a rank of 12 as before, here it is 11, here it is 8, and here it is 6, and here it is 3. Here the rank would be 1 and here it would be 5 and here it would be 9. So, that way we would be reconstructing the list back up to this level.

Then we can let the set of people who went for tea in the beginning and ask them to stand back in the queue recreating the original list. At this point the list will be completely ranked. In other words every person standing in the queue will now have the legitimate ticket that he should have had.

So, this is how we are going to reduce the length of a list. So, in our algorithm what we assume is that we are given a list of length  $n$ , and then from this list we will periodically remove independent sets. Once an independent set is removed the vertices that are behind the removed vertices will be rebated accordingly and then the vertex reduction will be applied further on the new list. So, we continue this process until the total number of vertices in the list reduces to below and by  $\log n$ . At this point the number of processes that we have is greater than or equal to the size of the list.

So, we can depute the processors to the nodes one per vertex therefore, we can now invoke our previous algorithm the point of jumping algorithm which when executed on.

(Refer Slide Time: 13:54)



A list of size  $n$  by  $\log n$  runs in order of  $\log$  of that many steps which is order of  $\log n$ ; using of course  $n$  by  $\log n$  processes. The only difference is that the initial values are now different. In the original point a jumping algorithm the initial values were like this; the node at the right end had a rank of 0 and the other nodes had a rank of one each.

Whereas now, the nodes will have different ranks each node will have a rank that is equal to one more than the number of elements ahead of it that have stepped out of the list. Then pointer jumping will in effect calculate the prefix sum of those values from the right end of the list. But these are these are precisely the ranks that these nodes would of had had the original list been ranked without any splicing out of vertices. Now, when the spliced out vertices are reinserted back in the list in the reverse order in which

they were removed, we would end up recreating the original list. And the time taken for that again would be order  $\log n$ .

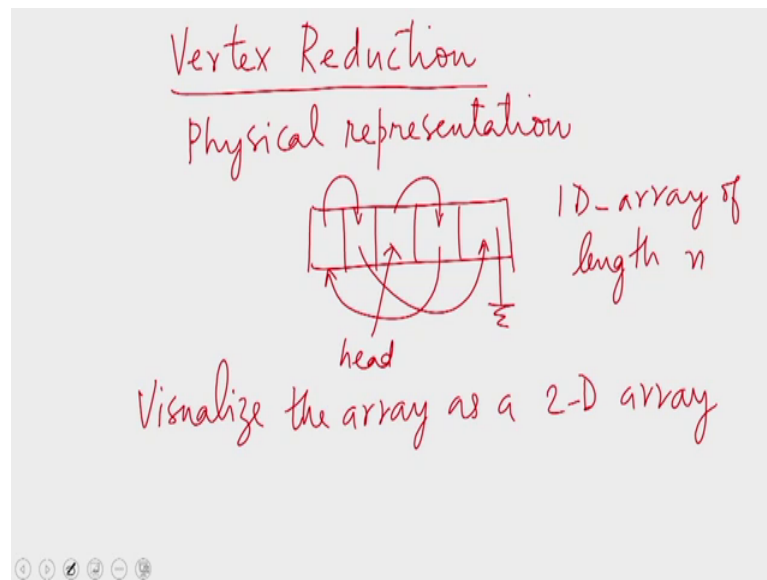
Therefore, to summarize what we have is this: we have several iterations each of these iterations are order 1 time iterations; in each iteration we remove an independent set of vertices from the list. We shall show that the number of iterations required would be order of  $\log n$ . At the end of all these iterations the length of the list would have reduced to at most  $n$  by  $\log n$ . Then the number of processes is enough to perform pointer jumping on the remaining list.

After that we again have several iterations. These are the original iterations done in reverse. If I call this phase 1 the pointer jumping, part phase 2, and the final part phase 3. We shall see that phase 3 is identical to phase 1 except in that the iterations are in the reverse order. That is the vertices that were spliced out last will be coming in first.

So, when these spliced out vertices are all reinserted back in exactly the same amount of time there is order of  $\log n$  time each iteration taking order 1 time; we would have recreated the original list and the list would be ranked. So, that is the overall structure of the algorithm. First we reduce the vertices from  $n$  to  $n$  by  $\log n$ , and then on the reduced list we run the pointer jumping algorithm to rank the reduced list. And then we splice in the vertices in the reverse order in which they were removed to reconstruct the original list. Now the original list would be ranked.

The cost of reconstruction is identical to the cost of the reduction; the schedule will be an exact reverse of the reduction. So, all that remains to be seen now is the reduction part.

(Refer Slide Time: 18:13)



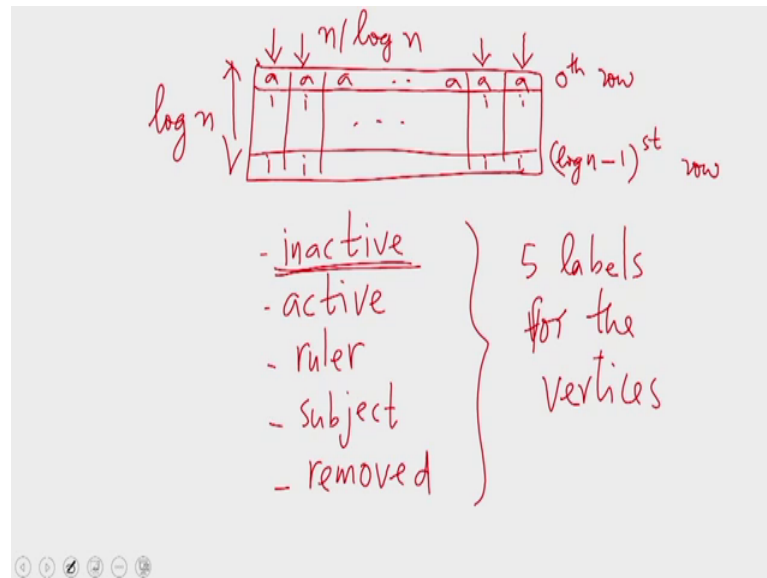
The second phase of the algorithm is the same as the old pointer jumping algorithm. And the third phase of the algorithm is an exact mirror image of the first phase. Therefore, all that needs to be specified as the first phase.

So, for the purpose of vertex reduction what we do is this: we take the physical representation of the list. The physical representation as we have seen before is an array in which the vertices are given in no particular order. That if the physical order has no relation to the logical order, the list could be represented in this fashion with the point crisscrossing through the array.

So, we are given an array of linked  $n$ , a 1 dimensional array of length  $n$ . This is the actual physical representation, but we can visualize this array as a 2 dimensional array. This is exactly as we did in the case of the optimal three colouring algorithm we saw in the last lecture.



(Refer Slide Time: 20:06)



Except that we assumed the 2 dimensional representation that we have in mind has  $n$  by  $\log n$  columns, and the size of each column is  $\log n$ . So, what we assume is that the given array is the row major representation of this 2 dimensional array. That has  $\log n$  rows and  $n$  by  $\log n$  columns

Now, the number of columns is exactly equal to the number of processors we have. So, we can depute one processor to each column. So, that is what we shall assume now; that every column has got one processor. Now during the course of this reduction algorithm will assume that the vertices hold various labels.

The various possible labels for the vertices are these: vertex could be in active, and inactive vertex is one that has not been taken up yet. A vertex could be active: an active vertex is one that has got a process of sitting on it now. And then we will have vertices that are called rulers: these are vertices that will be left in charge of some subjects. Therefore, then we will have vertices that are subjects to. And then finally, we have vertices that are removed: a vertex that has been processed and has been spliced out of the list is a removed vertex.

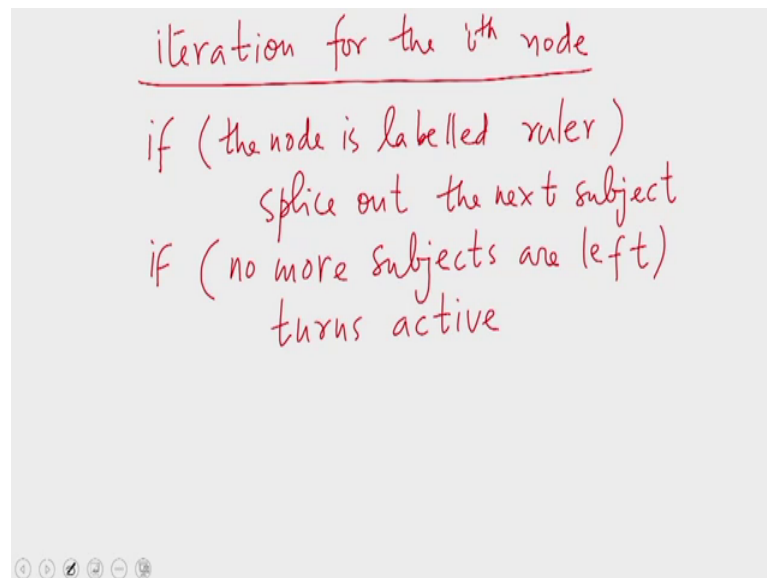
So, these are the five possible labels for the vertices. Every vertex in the list can hold one of these five labels during the course of the algorithm. In the beginning we assume that every vertex is inactive, that is because no vertex has been initially assigned a processor. So, every vertex is initially inactive, then what we do is: this we have exactly as many

processors as we have columns, we take the processors and place them in the first row or we can call this the 0-th row. Therefore, the final row will be number  $\log n$  minus 1. The rows are numbered from 0 to  $\log n$  minus 1.

So, let us take all the processors and place them in the 0-th row the  $i$ -th processor will occupy the  $i$ -th column. Therefore, all these nodes become active, every other node is inactive. So, at the beginning of the algorithm there are only two labels in the array: every node in the 0-th row is active and every node in the all the other nodes in the array are inactive.

Now the algorithm proceeds in this fashion.

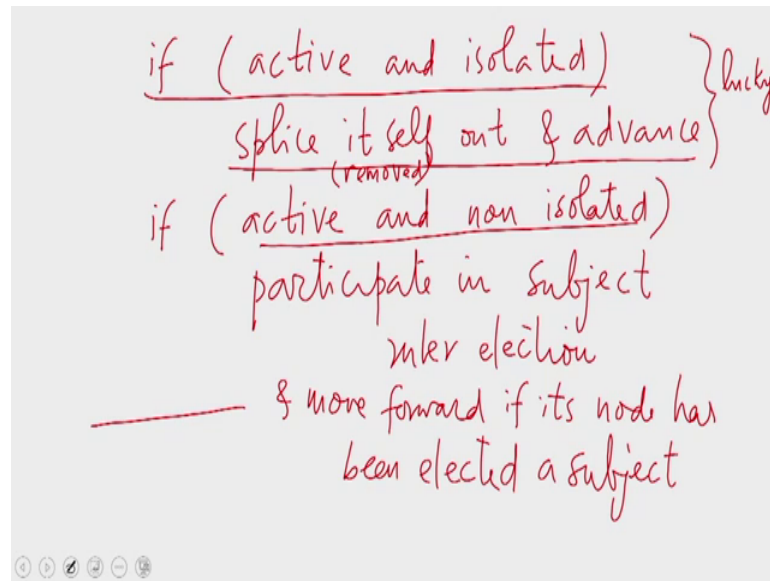
(Refer Slide Time: 23:34)



The algorithm proceeds in a number of iterations, in each iteration we do this. So, let us specify the iteration for the  $i$ -th node. So, in a iteration this is what is done: if the node is labelled ruler it splices out the next subject. This does not make sense now, because I have not told you have exactly subjects and rulers are elected, but I will come to that in a moment.

So, if the node is labelled ruler it will splice out the first subject, if no more subjects are left the node turns active.

(Refer Slide Time: 25:04)

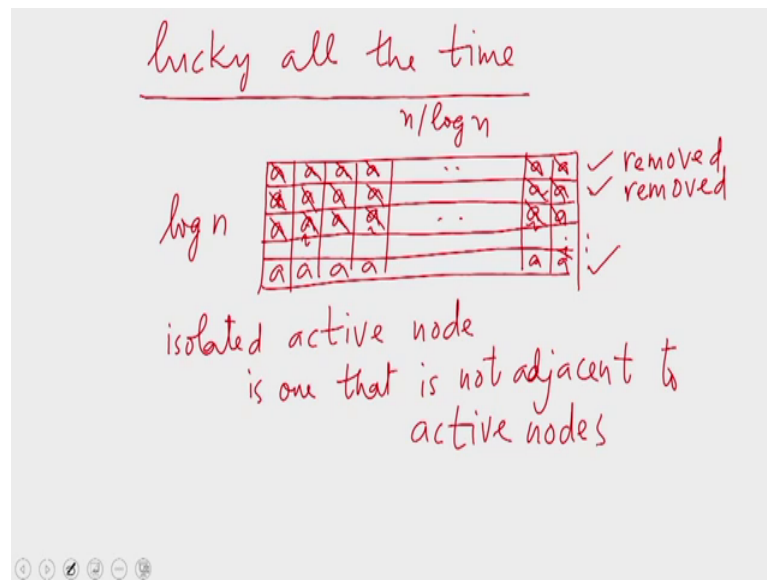


If a node is active and isolated, it will splice itself out and at once in its own column. On the hand if it is active and non isolated, the node will participate in subject ruler election.

So, that is an overall specification of an iteration. But then at the moment it will not make sense to you, but let me explain what it means. You should at the moment focus on only these two lines. So, forget the rest of the iteration and focus only on this part. So, the iteration involves justice let us say; if a node is actives and it is isolated it splices itself out and advances. So, this is we this is what will happen to the node if it is lucky.

So, this is what a lucky node this.

(Refer Slide Time: 26:48)



So, let us say we are lucky all the time. In that case what happens is this. We have this 2 dimensional array with  $\log n$  rows and  $n$  by  $\log n$  columns. And we have initially placed all the processes in the 0-th row. The 0-th row vertices are all active, every other vertex is inactive. This is how the initial array looks like. So, at the beginning of the algorithm this is what we have. So, the active vertices are all in the 0-th row.

So, let us say only the underline part happens. Every node that is active is isolated, that is every processor check for the node on which it sits the processors are all on active nodes to begin with and they find that they are all isolated. In the sense that its neighboring vertices are not active. We say that an active node is isolated if its neighbouring nodes are not active. An isolated active node is one that is not adjacent to active nodes.

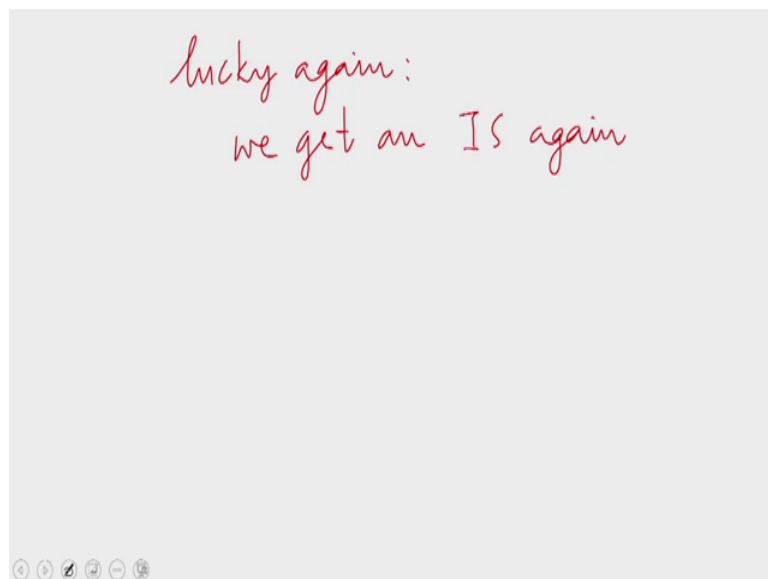
So, let us say initially every processor is lucky. In the sense that the nodes on which they sit are all isolated, which means all these are isolated vertices. In other words the vertices in the 0-th row forms an independent set. So, this is like the first set of vertices that when out for tea. So, these vertices have been randomly picked, because the physical representation of the list has nothing to do with the logical representation. And therefore, when the list is given we cannot predict which all vertices will be in the 0-th row: logically that is

Now, we have placed the processors on these vertices as such there is really no need that these form an independent set. But suppose we are lucky, because of our luck all these

nodes happen to be non-adjacent. In other words they form an independent set. Since they form an independent set they can all be spliced out simultaneously from the list. That is since no two of them are adjacent they can all go out for tea together. So, they are all spliced out of the list.

So, the length of the list reduces by  $n$  by  $\log n$ . These vertices all turn removed that is what the underlying part of the algorithm specified. If a node is active and is isolated then splice itself out and then advance in the corresponding columns. When a node is spliced itself out its label is changed to removed. So, all these vertices become removed. And then the processors will step down in their problems to the next element, which means the row below is activated all these nodes become active now. These all have been removed the elements of the 0-th row have all been removed the elements in the first row are now active.

(Refer Slide Time: 31:06)



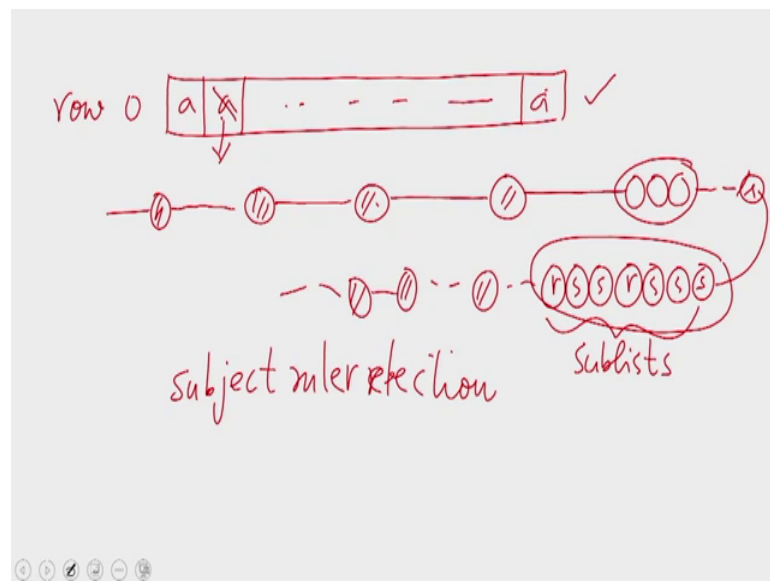
Let us say we are lucky again, which means we get an independent set again. Therefore, exactly as in the first step we can remove these vertices again. So, the second set of vertices are also removed, and the processors will advance further in the list; which means row number two is now activated and the algorithm will proceed in this manner. Every time executing only the underlined part, but this is if we are lucky every single time. So, as I was saying if you are lucky every single time; if you are lucky  $\log n$  times,

then we will be sliding down all the way until we reach the bottom at which point every vertex other than those in the last row have been removed.

So, now the scenario is like this the length of the list has been reduced from  $n$  to  $n$  by  $\log n$ , because we have only one element per column remaining in the list. And these are the active nodes which means all of them have processes citic on them. The remaining nodes have all been removed from the list one set at a time. This is precisely what we want. We want to reduce the length of the list from  $n$  to  $n$  by  $\log n$  and then on this list we want to run the pointer jumping algorithm. Once the pointer jumping algorithm is run we can start phase 3 of the algorithm in which we will put the vertices back in the reverse order. That is the rows of the array that were removed will be coming back into the list in the reverse order. When the last row that is the topmost row is put back into the list we would have reconstructed the list and every node in the list would be ranked; there is of the algorithm would run if we are lucky every single time.

So, you can see that if we are lucky every single time the reduction algorithm will run in order of  $\log n$  time, because order of 1 time is enough to remove one set of vertices. Now, let us consider the case where we are not lucky every single time. So, let us see what happens in the first step.

(Refer Slide Time: 33:47)



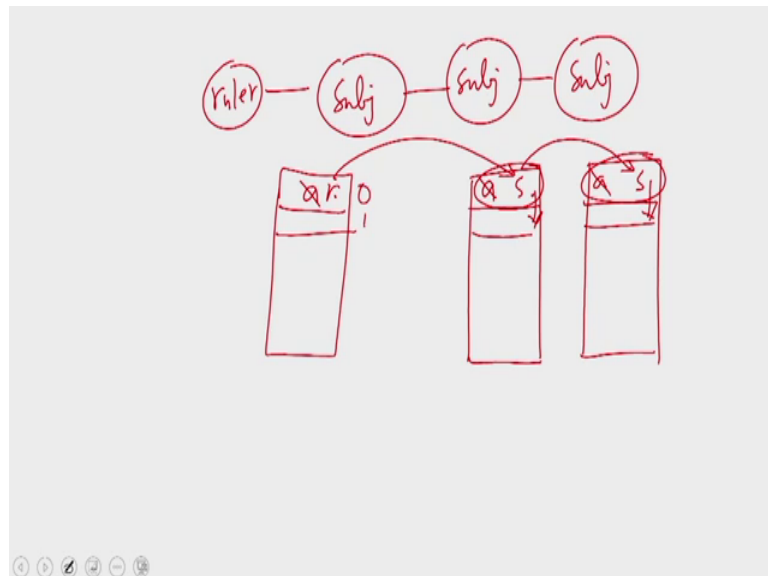
So, in the first step we are activating row 0, all these vertices are active. These are vertices that are physically consecutive, but then they need not be logically consecutive,

but some of these vertices could be logically adjacent too. Therefore, if we look into the logical representation of the list what you find is that these vertices are not necessarily consecutive, but some of them could be consecutive too. And when they are consecutive they could in fact be several of them that are consecutive and so on.

So, if you translate this into the logical representation what you find is that the set of vertices that are active, need not all be consecutive but some of them could be consecutive; and those that are consecutive will form sub lists of vertices that cannot be spliced out all simultaneously. The active vertices that are isolated: for example, all these vertices. In our example these are isolated vertices, they can be spliced out exactly as we did it before. That is what this part of the algorithm does: if a node is active and isolated we splice those vertices out and the processors in those columns will advance to the next element. But the remaining elements which are active, but are non-isolated they will participate in what we call subject ruler election.

So, these sub lists that form the set of non-isolated active vertices they will participate in subject ruler election. Out of each sub lists we will elect some as rulers and remaining as subjects.

(Refer Slide Time: 36:22)



So, every ruler will be left in charge of several subjects. If you look at the physical representation what we find is this: if the topmost vertex in a column which is at present active is chosen as a ruler of which the topmost element of another column is made a

subject; that could be several such, it could be that the topmost element of another column has also been made a subject. Then we will get a linked list of this form this is a sublist: a ruler followed by its subjects.

So, once again we have the scenario the processes are all placed in the 0-th row. Every vertex in the 0-th row is an active vertex, but then when we look at the logical sequence of the vertices what we now find is that: the processes are not on consecutive vertices necessarily because there is no relationship between the physical order and the logical order in general. These active vertices could be isolated or they could be consecutive. If the active vertices are isolated then we are those are the lucky vertices, because those vertex vertices can be removed from the list and the corresponding processors can advance in their respective columns. Which means if this vertex happens to be the same as this vertex this is an isolated vertex, so this vertex will be removed and the processor will advance in the corresponding column it will move on to the next row.

But then vertices that are not isolated cannot do the same thing what they do is to participate in a subject ruler election. So, for every such consecutive set of active vertices we will have some rulers and some subjects. So, every ruler gets some subjects. So, for now let me assume that the subjects are consecutive with the ruler. So, a ruler will get some vertices, some consecutive vertices as its subjects. So, here in this stretch we have 7 vertices of which the first and the fourth are elected as rulers the remaining are subjects.

So, the first vertex gets the second and the third as its subjects, whereas the fourth vertex gets the fifth, sixth, and seventh as its subjects. Then what we propose is that a node which gets subjects remain where it is, which means the processor that is sitting on this vertex will not be allowed to advance in its corresponding column. Whereas, the processor which is sitting here will be allowed to advance to the next element in its corresponding column. That is if a node becomes a ruler it is stuck with a number of subjects, it can proceed in its own column only after it finishes with its subjects. That is the job of splicing out these subject subjects from the list is now with this ruler or the processor which is sitting on the ruler. This processor is now responsible for splicing out these subject nodes; that is this node as well as this node.



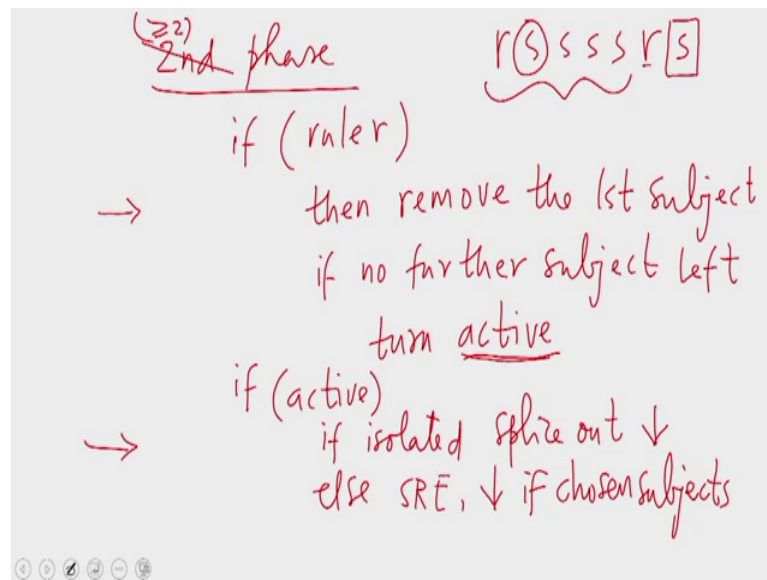
But the processes that are sitting on these subject nodes abandoned them and proceed in their respective columns to the next position. So, the processes that are sitting on the subject nodes get to advance in their respective columns. So, in this case the processor will advance to the second the second row, here also the processor will advance to the second row which means row number 1. Whereas, the process that is sitting on the ruler will be still stuck with row number 0.

So, the rules are stuck, whereas the subject processors get to move forward leaving the subjects behind in charge of the rulers. So, that is what we did in the last part of the algorithm specification. If a node is active and non isolated then it will participate in the subject ruler election and will advance forward only if its node has been chosen a subject.

So, after the subject ruler election has been taken place we can start the next phase. So, in the first phase we do not have to worry about what the subjects and rulers will do at the beginning of a phase, because when the algorithm begins there are no subjects and the and the rulers there are only active and inactive vertices; and the active vertices are all in row 0. So, in the first phase of the algorithm this is what happens. Every active node that is every row 0 node which is isolated will be spliced out, and the process of sitting on them will advance to the next row. But then for every set of consecutive row 0 vertices we will have subject ruler election, we will see how exactly the subject ruler election will take place. But once subjects and rulers are chosen the subjects will be left in charge of the rulers and the processes that was sitting on the subjects will abandon the subjects and move down their columns.

After this is done, we are ready for the next phase.

(Refer Slide Time: 42:45)



So, in the second phase of the algorithm; in the second iteration we have to consider every single ruler. So, every node checks whether if it is a ruler. So, if they node is a ruler then it must have a subject, then it will remove the first subject ; the nearest subject. If a node is a ruler it will remove the nearest subject. And after removing the subject, if it has no further subject left it will then turn into an active node. Then it will go back to the original game. So, this is the first part of a general phase, this will come into play only from the second phase onwards.

So, in general at the beginning of a phase we have vertices of all kind. There are the removed nodes which will not come up again except in phase 3 when they are put back in the list. Therefore, in the reduction phase when once vortex is removed it is out of play. Then we have vertices that are active and inactive. Inactive vertices have not been taken up yet they have not got a processor yet, so they will come in future. Then we have rulers and subjects. The subject nodes have lost their processors, because those processors have abandoned these subjects to be in charge of the corresponding rulers. So, the subject nodes are without processors. So, we have processors sitting on only the ruler nodes and the active nodes.

First at the beginning of a general phase we will look at the ruler nodes, for every ruler node we will remove the first subject since two ruler nodes are separated by several subjects. That is when you consider two consecutive rulers they will have the subjects of

the first ruler between them. So, these are all subjects of the first ruler. Therefore, when two rulers are removing therefore, subjects we are in essence removing an independent set. This ruler is removing its first subject namely this the circle vertex. The second ruler is removing its first subject namely this the squad vortex. So, we find that the circle vertex and the squad vertex are not adjacent, because they must have one ruler in between them.

Therefore, according to the scheme when every ruler is removing its first subject we would be removing an independent set. So, at this instance in the execution of the algorithm we would be removing an independent set from the list. So, several rulers are removing their respective first subjects all simultaneously. And then every ruler has to examine whether it has now removed its last subject. If the last subject has been removed then the ruler goes back to being active. And then it will take part in the second part of the algorithm which we saw early. If a vertex is active and if it is isolated it will splice itself out and go down the column, otherwise it will take part in subject ruler elections and then advance if chosen subjects. That is how the algorithm works in general.

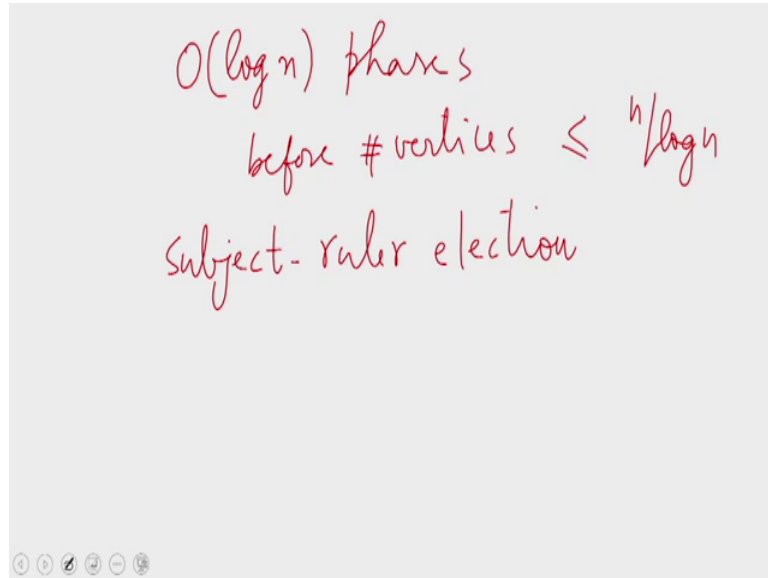
So, this is not just the second phase, this is every phase greater than or equal to 2; every phase numbered greater than or equal to 2 will function like this at the beginning of a general phase. We have processors sitting on rural ruler nodes as well as active nodes. First we activate the processors that are sitting on the ruler nodes. All these ruler nodes will check if they have they do have subjects that they have at least one subjects, because as soon as a ruler loses all its subjects it would have turned active. Therefore, if a node is still ruler it is with subjects. Therefore, every ruler will remove the first subject that it has and if there is no further subject left it will turn active.

So, this is one time instance when we remove an independent set from the list. So, after the turn off the ruler nodes are over we consider all the active nodes. If an active node is isolated we splice the active node out of the list. So, this is another instance during a phase when an independent set will be removed from the list. So, in every phase we are in fact removing two independent sets from the list. In the first instance every ruler removes its first subject, all this happens simultaneously. In the second instance every active node that is isolated will splice itself out, this will also happen simultaneously. And then if a node is active, but is not isolated then it will take part in subject ruler election, and every processor that has its node chosen a subject abandons this node to be

in charge of the corresponding ruler and advances in its column and that is the end of the phase. Then we continue to the beginning of the next phase

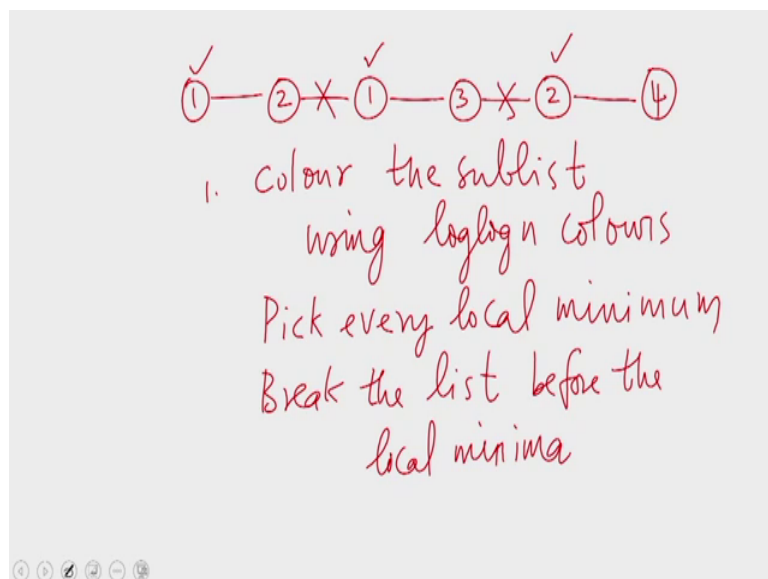
So, this is how the algorithm proceeds phase after phase until the number of vertices is reduced to less than  $n$  by  $\log n$ .

(Refer Slide Time: 49:14)



We wish to be able to claim that we require only order of  $\log n$  phases before the number of vertices falls to below  $n$  by  $\log n$ . But this requires specifying the details of the subject ruler election. The subject ruler election proceeds in this fashion.

(Refer Slide Time: 49:57)

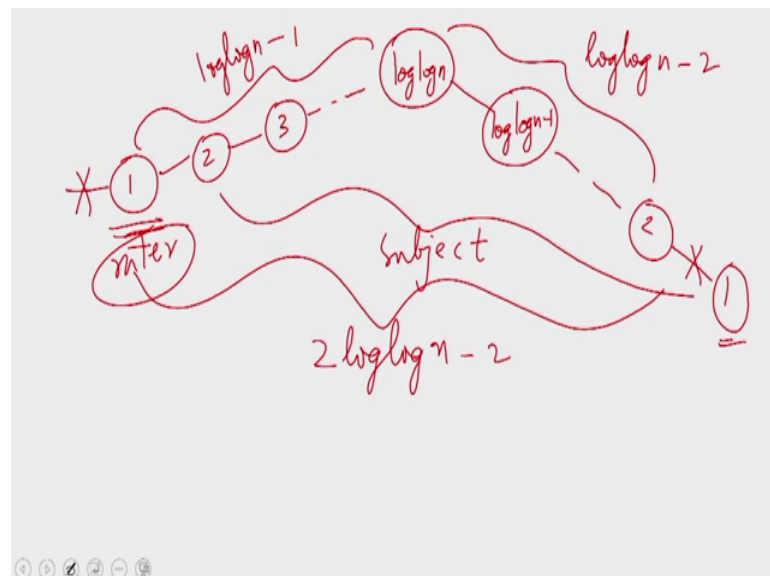


The subject ruler election is proceed is done on a set of consecutive active nodes.

So, all these nodes have processes sitting on them; what we do first is this. Colour these sub lists using double log n colours. At most three steps of the symmetry breaking algorithm are enough to colour the sub list with double log n colours. Once the list is coloured in this fashion a very local minimum is picked out. For example, if this list is coloured in this fashion the local minima of this list are these this node is a local minimum because it has a neighbour of colour two, but it does not have a neighbour of colour less than one. This node is also a local minimum, because it has two neighbours of colours two and three it does not have any neighbour of smaller colour. And this node is also a local minimum, because it is two colours that are larger than two and it does not have a neighbour of colour less than 2.

So, we pick out the local minima and then break the list immediately prior to the local minimum. So, let us say we break the list in this fashion. So, the list is broken into three pieces: the first stretches 1-2, the second stretch is 1-3 and the third stretches 2-4. Now, we can claim that the length of each stretch is at most 2 double log n. If you break the list before the local minima the length of the resultant list will be at most 2 double log n.

(Refer Slide Time: 52:36)



That is because we have used it most double log n colours. In the worst case scenario we could have a colour distribution like this. Let us say one is a local minimum then we have 2, then we have 3 let us say the colours going all the way up to log of log n, and

then let us say the colours start decreasing. This is the worst that can happen. The colours can go increasing from  $1-2 \log \log n$  and then decrease from  $\log \log n$  into 1. If this is the case then this one is the local minimum and this one is a local minimum, none of the other vertices in this stretch is a local minimum. And then the preceding link of every local minimum is broken, which means we end up getting this entire stretch as a sub list. In this sub list the colours are increasing from  $1-2 \log \log n$  and then are decreasing from  $\log \log n$  into 2.

So, the length of this stretch can be at most  $2 \log \log n - 1$ . This part is  $\log \log n - 1 - 2 \log \log n$  the number of edges, and then here it is  $\log \log n - 2$  for a total of  $2 \log \log n - 3$  edges or  $2 \log \log n - 2$  vertices.

So, you can say the maximum stretch of any resultant list is going to be  $2 \log \log n - 2$  vertices. And then we can choose the first vertex here as the ruler and the remaining vertices as the subjects. So, this would ensure that every ruler has given at most  $2 \log \log n$  subjects. If we had in fact, coloured with  $\log \log n$  by two colours then we would be able to ensure that every ruler gets at most  $\log \log n$  subjects, which is a condition that we need for the correct analysis.

Now the analysis of the algorithm that we shall do in the next lecture. That is it from this lecture hope to see you in the next lecture, where we shall do the analysis of this algorithm.

Thank you.