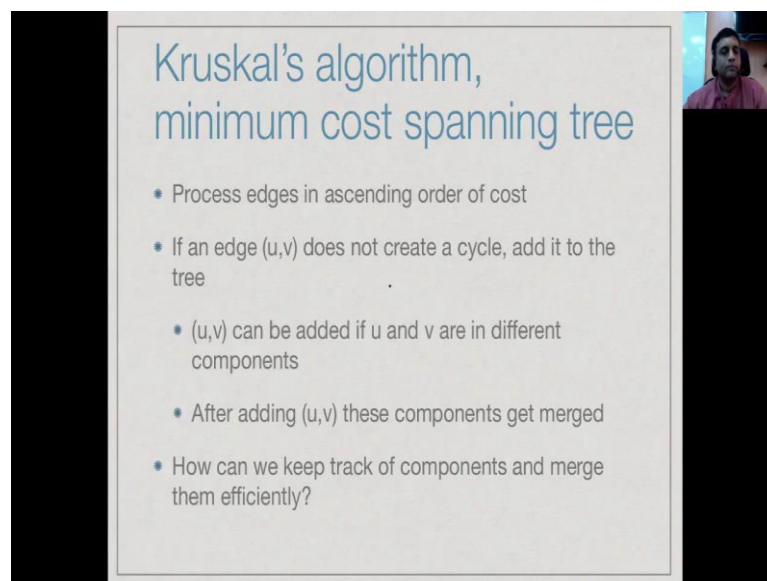


Design and Analysis of Algorithms
Prof. Madhavan Mukund
Chennai Mathematical Institute

Week 5
Module - 01
Lecture - 32
Union-Find data structure

So, when we look at algorithms and weighted graphs for shortest paths and for minimum cost spanning trees, we had to use some data structures in order to make the updates efficient. So, at that time we assume that these data structures were available and we went ahead to determine analysis of these algorithms. Now, let us go back and look at these data structures. So, we begin with the Union Find Data Structure which is used in Kruskal's algorithm for the minimum cost spanning tree.

(Refer Slide Time: 00:28)



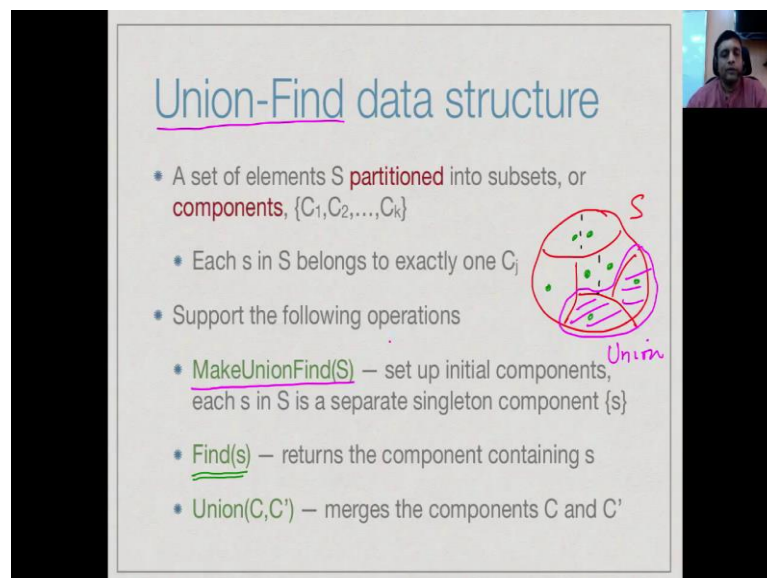
Kruskal's algorithm,
minimum cost spanning tree

- Process edges in ascending order of cost
- If an edge (u,v) does not create a cycle, add it to the tree
- (u,v) can be added if u and v are in different components
- After adding (u,v) these components get merged
- How can we keep track of components and merge them efficiently?

So, recall how Kruskal's algorithm works. We arrange the edges in ascending order of the cost and we process the edges in this order. So, each edge we pick up. If it does not create a cycle, we add it to the tree and we observe that not creating a cycle is as same as keeping track of the components that we have so far, and checking that the end points line different components. So, the edge $u v$ can be added if u and v currently are not connected. They are not in the same component. Now, as the result of adding the edge, the two components do get connected. So, we have to merge those two components. So,

the bottle neck in implementing Kruskal's algorithm efficiently is to keep track of this collection of components in order to check which component a vertex belongs to, and to merge two components whenever we add an edge connecting them.

(Refer Slide Time: 01:22)



Union-Find data structure

- A set of elements S **partitioned** into subsets, or **components**, $\{C_1, C_2, \dots, C_k\}$
- Each s in S belongs to exactly one C_j
- Support the following operations
 - MakeUnionFind(S) — set up initial components, each s in S is a separate singleton component $\{s\}$
 - Find(s) — returns the component containing s
 - Union(C, C') — merges the components C and C'

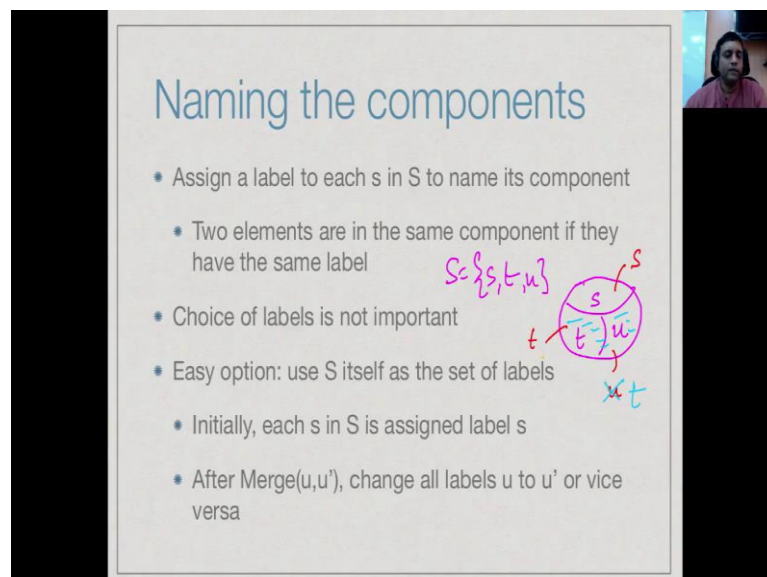
The diagram shows a large circle labeled S containing several smaller green dots representing elements. These elements are grouped into several overlapping regions, some of which are shaded in pink and purple, representing the components of the partition. The word 'Union' is written in pink cursive next to the diagram.

So, formally the problem that we have claimed to solve is to maintain a partition of a set and update this partition, right. So, by partition we mean that we have a set s and it is broken up into some disjoint subsets. So, these subsets do not overlap, right and every element belongs to none of these things. So, there may be some which have more than one element and each element is assigned to exactly one partition, right and we call these partitions also components. So, in Kruskal's algorithm and other applications, very often we start with a partition in which every element is on, it is over. So, we never have maybe two elements in a partition.

So, we will call this setting up as data structures. So, we will call it union find because the two operations we actually support on this data structure are called find. So, this is a query operation. It is as given an element s , right. Let me know which component it belongs to currently. So, this is an update which it does not update the data structure. It just queries the data structure and tells us in which of these partitions does s currently live, and then we have an update which allows us to take two partitions, right. So, may

we take these two partitions and say now combine them into single partition. So, we call this union, right. So, there is a union operation which merges partitions together and there is a find operation which has to keep track of which partition is said belongs to an element belongs to over time with partition. It originally would have got merged with other partitions because of these two operations union and find. We call this a union find data structure which supports these two operations efficiently, and the initialization of this union find is an operation which takes a set and breaks up to it. It has n elements up into n components each containing one element.

(Refer Slide Time: 03:09)



Naming the components

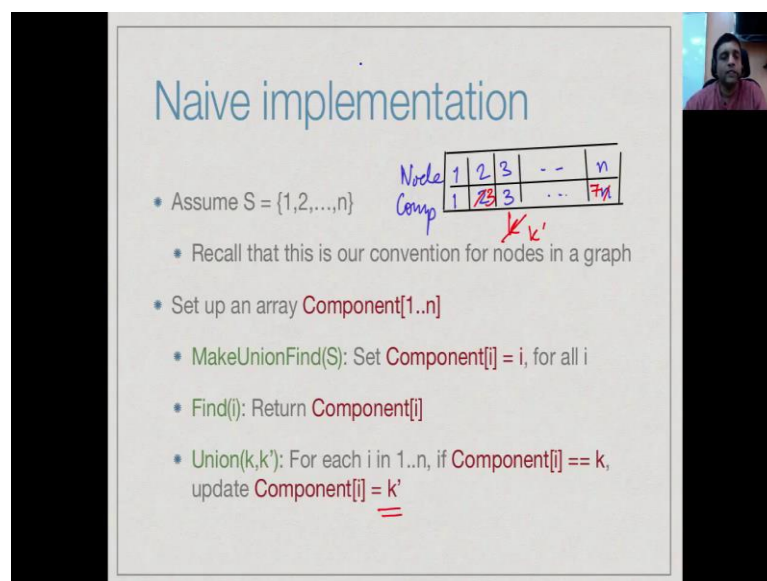
- Assign a label to each s in S to name its component
- Two elements are in the same component if they have the same label
- Choice of labels is not important
- Easy option: use S itself as the set of labels
- Initially, each s in S is assigned label s
- After $\text{Merge}(u, u')$, change all labels u to u' or vice versa

So, the first issue that we have to deal with is about the names of the components. What do we call these components? It is a simple solution. We will find to just use the elements of the set itself as names, right. So, it does not really matter of what names we give. We just need to be able to check periodically whether s and t or u and v belong to the same component. So, we just need to know whether find of u is equal to find of v , right. So, the exact choice of how we label find of u and find of v does not matter. So, as long as we can check whether two labels are the same or differ, but rather than manufacturer set of labels out of them have, we will actually choose the labels to set elements themselves. So, initially we said every element lies in a single partition. So, supposing I have a set consisting of s t u , then I would initially have three partitions. One

contains s, one containing t, and one containing u.

You have the question is what do we call this partition. Well, we just call them the same thing. We call this partition s, call this the partition t and call this the partition u. So, sometimes the names of the element will refer to names of partitions. Sometimes they will refer to names of the elements themselves. Now, what happens when we merge? For example, supposing we merge these two partitions, right. Then, the label has to be the same. So, the element does not change, but maybe we might take the set, the label u and make it t. So, now, both element u and element t belong to the partition label t, right. So, we just use as the set of labels, the names of the elements themselves.

(Refer Slide Time: 04:44)



Naive implementation

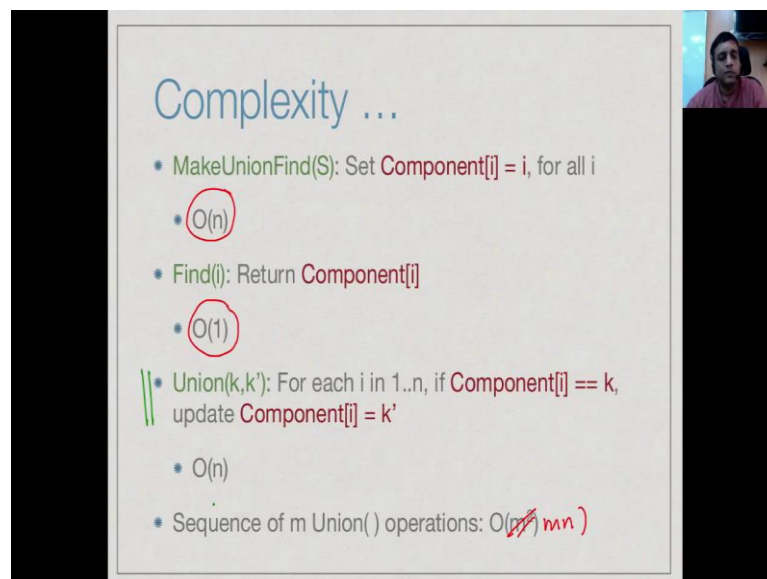
- Assume $S = \{1, 2, \dots, n\}$
- Recall that this is our convention for nodes in a graph
- Set up an array `Component[1..n]`
- `MakeUnionFind(S)`: Set `Component[i] = i`, for all i
- `Find(i)`: Return `Component[i]`
- `Union(k, k')`: For each i in $1..n$, if `Component[i] == k`, update `Component[i] = k'`

Node	1	2	3	...	n
Comp	1	2	3	...	n

So, in particular if you are dealing with graphs, the elements are the vertices to $(())$ and we already had convention that we have n vertices in our set and we call them 1 to n . So, set of elements is 1 to n and sources are set of components, right. So, what we will do now is the easiest way to keep track of this is to setup an array, right. So, we have an array which we will component and what will this array say. Well, this will say that for each of the vertices are nodes to end which component it belongs to, right. So, initially we said each component will contain exactly one vertex. So, we can just have a vertex n a vertex i and component i for every in general after some time, these might change.

So, this might have gone to component 3 is might have gone to component 7 and so on, right. So, over time the component that more belongs to changes because of the union operations, so then when we find you, just return the current value of component and for union all we have to do is, we have to check and make all the components, both components k and k prime have the same label. So, rather than invent a new label, we will choose either k or k prime. In this case we choose k prime. So, what we will do is, we will go through and wherever we see a k , we will replace by a k prime. So, we will systematically replace every entry of the form k in this array to k prime. So, after this all component values we choose to be k or now k prime, all virtual k prime and remain k prime. So, effectively the two components have been merged. So, this is a very simple implementation of union find. So, let us try and understand why this is not a very good implementation from a complexity point of view.

(Refer Slide Time: 06:35)



Complexity ...

- MakeUnionFind(S): Set $\text{Component}[i] = i$, for all i
 - $O(n)$
- Find(i): Return $\text{Component}[i]$
 - $O(1)$
- Union(k, k'): For each i in $1..n$, if $\text{Component}[i] == k$, update $\text{Component}[i] = k'$
 - $O(n)$
- Sequence of m Union() operations: ~~$O(mn)$~~ mn

So, clearly in order to make the initial things, we have to scan the array once, and then we just have to initialize component of i to be the value i . So, this takes order n time. So, this is find. Similarly, finding an element is efficient. We just have to look up the i th element in the array and remember that in an array accessing any element takes constant time. So, this is an efficient operation. It takes constant time. On the other hand, union is a problem because the way we have described union, we have to go through every node,

check if its component is k and if its component is k, if component of device k, we have to update it k prime, right. So, regardless of what the components k and k prime currently look like, we will have to scan all the elements and update those which are k to k prime. So, this will take order n time for just one union operation which is independent of the size of k and k prime as current partition sets.

So, if we do a sequence of m such operations, then this will be order m times n, right and if it is n such operations, it will be n square, right. So, a sequence m operation, each of them will take order n time and we would like to improve on this. So, basically we want to see we can improve on the speed of the union operation.

(Refer Slide Time: 07:54)

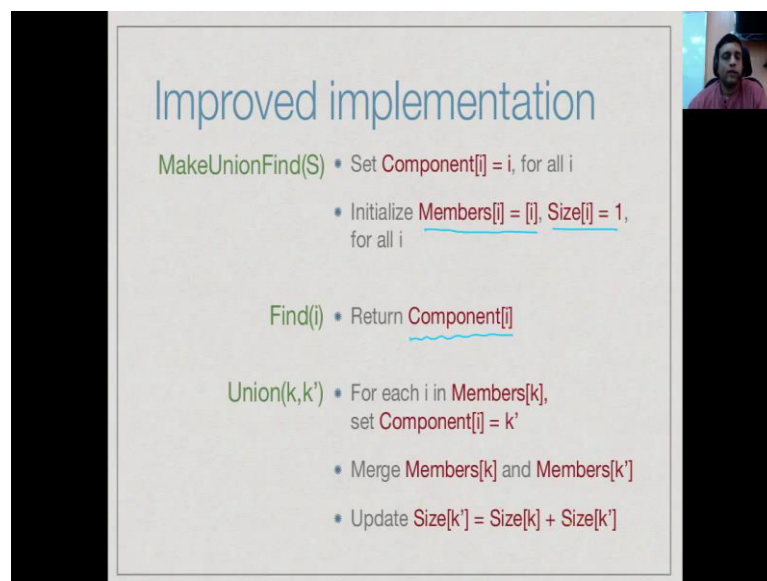
Improved implementation

- As before, array **Component**[1..n] 1 - [1]
2 - [2]
Size = 1
- Also, for each component k, a list **Members**[k] of its members.
- Array **Size**[k] records size of list **Members**[k] 4 -> [1, 2, 4, 7]
Size = 4

So, let us make a slightly more elaborate representation. So, we keep this array component as before which tells us for each vertex I which component it belongs to, and the components as before are initially labeled 1 to n. The names are drawn from the same set. So, basically 1 to n has the vertices 1 to n or also names are components, and initially the component of I is the vertex. I is an component time. Now, we have a separate array of list, right. So, for each component, there will be currently we keep a list. So, initially the list is that the component 1 consist of the vertex 1 component, 2 consist of the vertex 2 and so on, but over a period of time we could have situation whether the component 4

consist of vertex 1, 2, 4 and 7, right. So, for each component we have the list of vertices that it belongs to and we will also separately keep track of the size. So, we will say that the size of this component is also 4 and the size of this component is 1. So, we have two auxiliary things. We keep for each component k the list of its members explicitly and we also keep the size of this list. So, we know exactly how big each component is at any given time initially, of course the size is 1.

(Refer Slide Time: 09:10)



Improved implementation

MakeUnionFind(S)

- Set $\text{Component}[i] = i$, for all i
- Initialize $\text{Members}[i] = [i]$, $\text{Size}[i] = 1$, for all i

Find(i)

- Return $\text{Component}[i]$

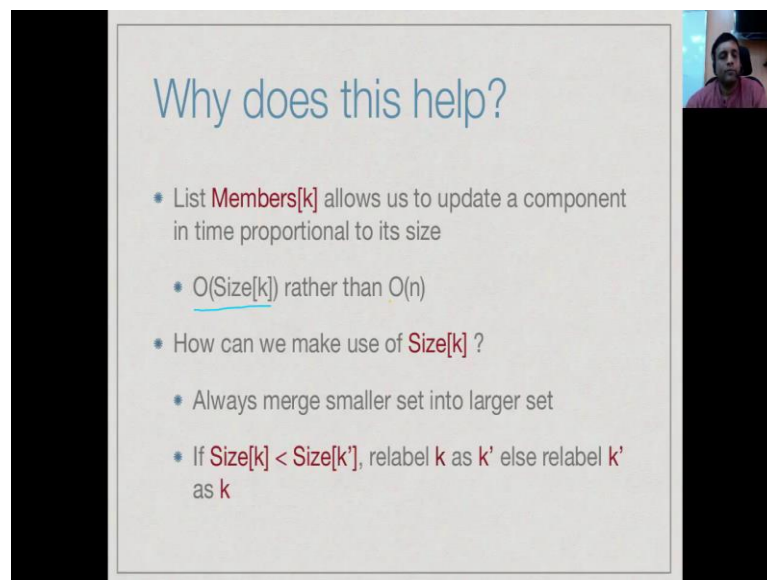
Union(k, k')

- For each i in $\text{Members}[k]$, set $\text{Component}[i] = k'$
- Merge $\text{Members}[k]$ and $\text{Members}[k']$
- Update $\text{Size}[k'] = \text{Size}[k] + \text{Size}[k']$

So, when we need to make union find, we said component i equal to i as before, then we initialize that the members of the list i are component i adjust the list containing i itself in the size of each component is 1. Now, find is exactly the same as before. We just have to look at component and return the value. The component i points two unions is also similar to before. So, what we need to do is, we need to set everything that is pointing to k to point to k' inside. So, every entry in component which k should be, k' prime, but now we can do this by just looking at members. So, we do not have to scan all the elements 1 to n . We can look up every element that appears in members of k and update its value to k' prime, right. So, this is one saving. We no longer have to go through 1 to n . We have only looked exactly at the member which belongs to the set k . Then, of course we need to update these new things.

So, members of k now become members of k primes. So, we will merge these two lists. Now, remember that these two lists are in sorted order. We can assume that they are always kept in ascending order of the names of the elements. So, merging two sorted lists as we did in merge sort, takes time proportional to the length of the final list. So, this will be a linear time thing in proportional to the size of the components k and k prime. Finally, now we have one new component which subsumes earlier to. So, its size is exactly the sum of the previous two of course. These are partitions. No element was repeated. So, every element that joins the partition is a new one. So, size of k prime is exactly size of k plus the old size of k prime.

(Refer Slide Time: 10:47)



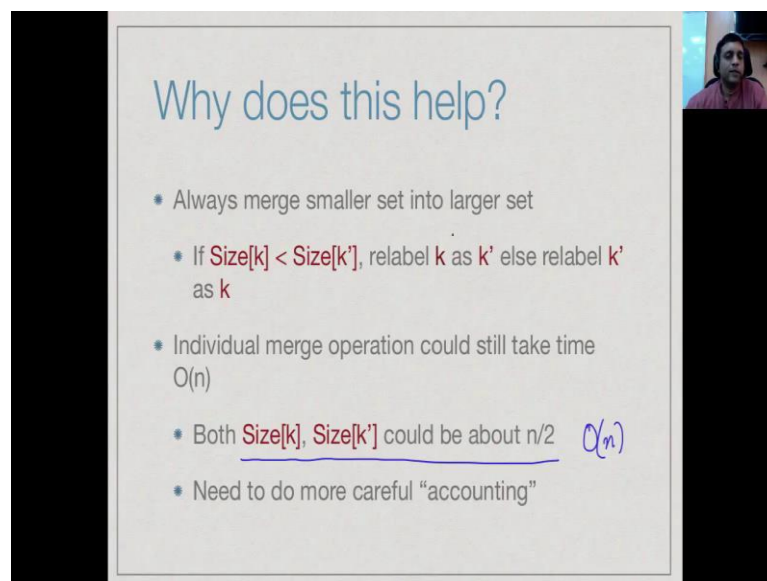
Why does this help?

- List **Members[k]** allows us to update a component in time proportional to its size
- $O(\text{Size}[k])$ rather than $O(n)$
- How can we make use of **Size[k]** ?
- Always merge smaller set into larger set
- If **Size[k] < Size[k']**, relabel k as k' else relabel k' as k

So, why do we get some benefit from this? So, the first thing as we said is updating a component is now proportional to its size, right. Up going updating component takes orders size of k is kept. It does not take order n steps. We do not have to go through every vertex 1 to n . We can explicitly look up those elements mentioned in members of k and only update those values, but size of k actually places in much more important role. What we can do now is we can determine whether to re-label k is k prime or k prime is k . Remember we have a choice when we merge k and k prime. All the elements are going to become part in the same component. So, the new component will be either call k or it will be call k prime.

So, which one we should choose? So, this strategy that we are going to do is to keep the name of the larger, right. So, the size of k is smaller than the size of k' . That means, k' currently has more members in k . Then, we will keep k' as a name of the final set. So, we will replace all cases k' and symmetrically size of k is bigger than size of k' . We will replace all the k primes as k , right. So, the smaller set changes its name and the bigger set keeps its name.

(Refer Slide Time: 11:58)



Why does this help?

- Always merge smaller set into larger set
- If $\text{Size}[k] < \text{Size}[k']$, relabel k as k' else relabel k' as k
- Individual merge operation could still take time $O(n)$
- Both $\text{Size}[k], \text{Size}[k']$ could be about $n/2$ $O(n)$
- Need to do more careful "accounting"

So, this does not give us any benefit in the worst case of an individual union operation. Suppose we have size of k and size of k' roughly the size of half. So, if we are built-up two components which are roughly half the size of the overall set, then whether we merge k or k' into the other set, we have to update about half the values. So, this is r order n operation, right. So, earlier we have said that without doing anything fancy, we will scan all the vertices and worst case in fact is every case of update will take as order n time. Now, it says that there is a worst case where we cannot avoid taking order n time. So, n by 2 is order n . So, therefore what we gain? So, what we have gained cannot be really accounted for terms of individual merge operations. We have to look at the humility effect of punch of merges. So, we need to do what is more careful accounting. Now, we have to account for the operations. So, remember we did some careful accounting when we did things like you know when we used adjacent list in breadth first

search. We said across all the loops, we will see each edge exactly twice. Therefore, across all the loops we take order n time. So, we need to do a similar kind of careful accounting across, not one merge, but all the merges that they take x .

(Refer Slide Time: 13:17)

Why does this help?

- For each element s , size of Component[s] at least doubles each time it is relabelled
- After a sequence of m Union() operations, at most $2m$ elements have been "touched"
- Size of Component[s] is at most $2m$
- Size of Component[s] grows as $1, 2, 4, \dots$, so s is relabelled at most $O(\log m)$ times

Handwritten annotations on the slide:

- Diagram showing two overlapping circles labeled K and K' . Above them is $K \leq K'$. Below the intersection is $K + K \leq K + K'$.
- Below the fourth bullet point, a sequence $1, 2, 4, \dots, m$ is written with an arrow pointing from 4 to $\log m$.

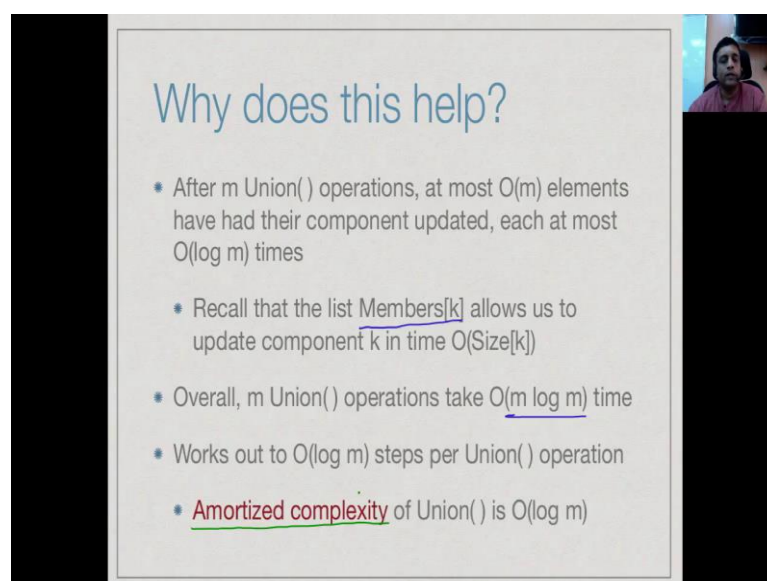
So, the effect of merging the smaller set into the bigger set is that if I look at individual element, the component if it changes, it is labeled, right. If the component is currently k and it becomes k prime, then new set is at least twice is because of the votes, right. So, supposing I had set k and an element is in it and I had another set k prime, then I decide to merge these into two single set. Now, by assumption if the new set continues to be called k prime, then that means that k worse less than k prime or you could even say less than equal to does not really matter. So, since case smaller than k prime, what it means is that if I look at k plus k which will be double the size of the old set, this will be less than or equal to k plus k prime thus by substituting. Therefore, this is the size of the new set, right. K plus k prime is the size of the new set I construct and it is going to be at least twice the size of the old size. Therefore, whenever the name of the set labeling component of an element changes, the new set belongs to at least double the size.

So, now let us look at some sequence of m union operations starting from the initial condition when I have all elements in separate partitions. So, what can happen in each

operation, perhaps I combine two elements, right. Now, if the next time if I combine these two, then totally in two operations I only affect three elements. So, in the worst case if I start doing this separately, each time in one operation I accept it affects and in other operations it affects more and so on. So, after a sequence of m operations at most $2m$ elements have had the status change from the initial condition, when they have a pointing two or component consisting only themselves. So, that means, only $2m$ elements have ever been affected by my union operations. That means as component cannot have got larger than $2m$, because in order to get into a component, something must change. Only $2m$ elements are allowed to have any changes apply to them at all, right.

So, the size of any component after m union operation at most $2m$, but how does a size grow? It goes 1, 2, 4 because it keeps doubling up to m , right. So, after $\log m$ steps if I double 1 $\log m$ times, I will get m . So, therefore, a component s can be relabeled a fixed, sorry a fixed element s can be relabeled at most $\log m$ times. So, because we have this doubling the number of times that is set can be an element can move it to a new set. It is restricted because each time it moves the size of its component doubles, and there is a limit on the largest component it will belong.

(Refer Slide Time: 16:22)



Why does this help?

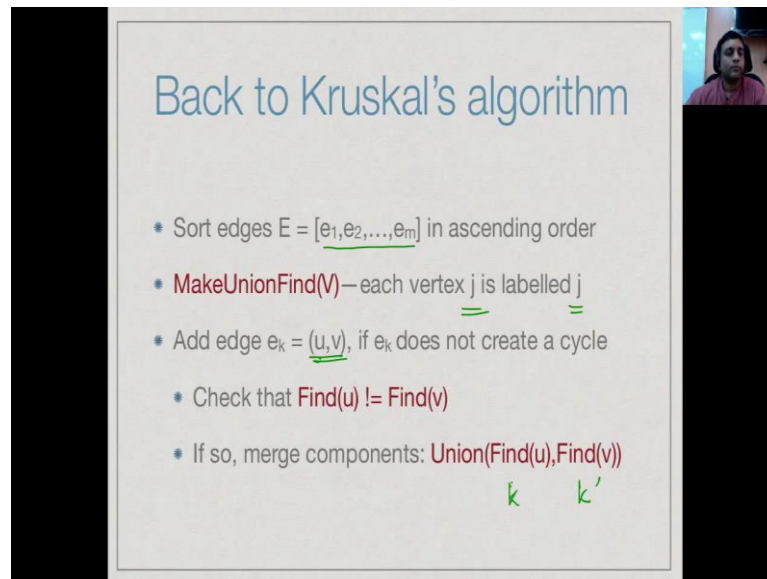
- After m Union() operations, at most $O(m)$ elements have had their component updated, each at most $O(\log m)$ times
- Recall that the list Members[k] allows us to update component k in time $O(\text{Size}[k])$
- Overall, m Union() operations take $O(m \log m)$ time
- Works out to $O(\log m)$ steps per Union() operation
- Amortized complexity of Union() is $O(\log m)$

So, therefore, if we look at some total of m union operations, we know that $2m$. So, order m elements have had the component updated and each has been updated at most $\log m$ times, right. Remember now that when we are updating elements, we do not touch any element which is not updated. It is not the old setup, where we have to scan all the nodes 1 to n in order to decide which one is not because we have the list members. When we want to update the component k , we exactly update only those components. So, we only touch the components we change. So, m elements changes $\log m$ times. So, totally we have $m \log m$ steps of change, right. So, if we do m union operations, we take $m \log m$. It is not that each one takes $m \log m$ time directly, but the cumulative total of m union operations is only $m \log m$, right.

So, this we can average out in a way you say that therefore, since where m operations in the total is $m \log m$, even though some small some more big on an average, they take $\log m$ operations. So, this is a different kind of analysis. It is not quite the analysis we did for. For example the adjuration seal is in b of s , where we just added of everything across whole thing. Here also we are adding, but we are also kind of pointing backwards and saying now there are totally so many operations and the total time taken across all these operations so much. We divide and we give each share of the total cost. So, though it is not the case that in every single operation takes $\log m$ time, we can kind of believe that this gives a. So, $\log m$ implementation of the union operation. So, this kind of analysis is called amortized complexity.

So, this is the term which actually comes from accounting from financial things, where you have certain cost which for example you might have in order to run a business. You might have to setup something, right. You may setup an office, and then all that now if you cannot say that my cost in give one was a lot, because I have to do a lot of work to setup the office. Therefore, day one with expensive what you have to do is look at the total life time of your operation and say this course is divided across the entire things. So, is it work, right that is called amortization, where you kind of take some fixed cost of f , fixed piece of equipment are said or worked and divided across the entire life span when it is going to be used. So, in the same way here we are kind of doing amortized analysis. We are counting across all the m union operations how much time we take, and then working out that each one takes roughly $\log n$ time.

(Refer Slide Time: 18:49)

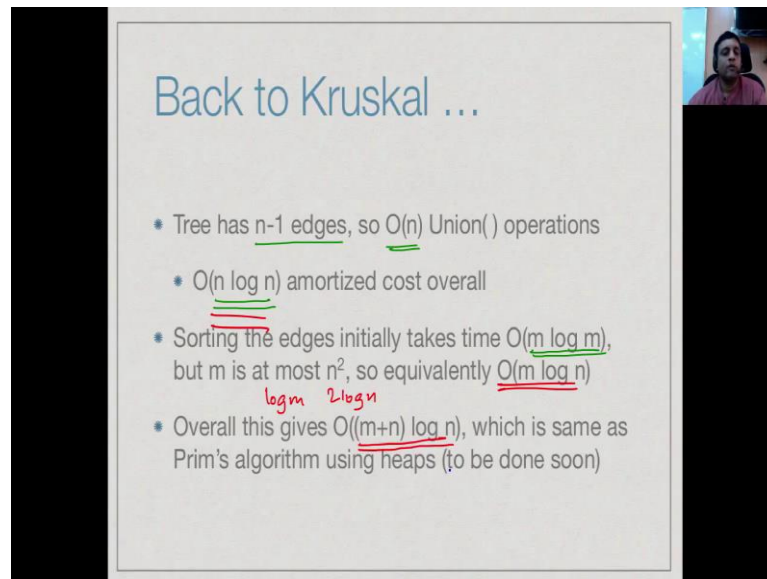


Back to Kruskal's algorithm

- Sort edges $E = [e_1, e_2, \dots, e_m]$ in ascending order
- **MakeUnionFind(V)**—each vertex j is labelled j
- Add edge $e_k = (u, v)$, if e_k does not create a cycle
- Check that $\text{Find}(u) \neq \text{Find}(v)$
- If so, merge components: $\text{Union}(\text{Find}(u), \text{Find}(v))$
 $k \quad k'$

So, how do we use this union find data structure and Kruskal's algorithm? Remember Kruskal's algorithm be initially sort the edges. So, we have u and to e_m and ascending order of cost. Now, we begin with trivial partitions. So, we do make union find of our set of vertices. So, each vertex j is label j . So, we have exactly n partition. One containing each vertex and now we need to add the current edge we are looking at, provided it does not create a cycle. This is the same as saying that send point was being different components. This is the same as saying that if I do find of u and find of v for an edge u, v , right, find of u is not equal to find of v and if find of u is not equal to find of v , then I need to do a merge. So, I need the union of these two components. I can get their component names by using find u and find v . So, this will be some k in general the u th component containing u . This will be k' th component containing v and I do a union of course, and I got in new component which contains both u and v .

(Refer Slide Time: 19:57)



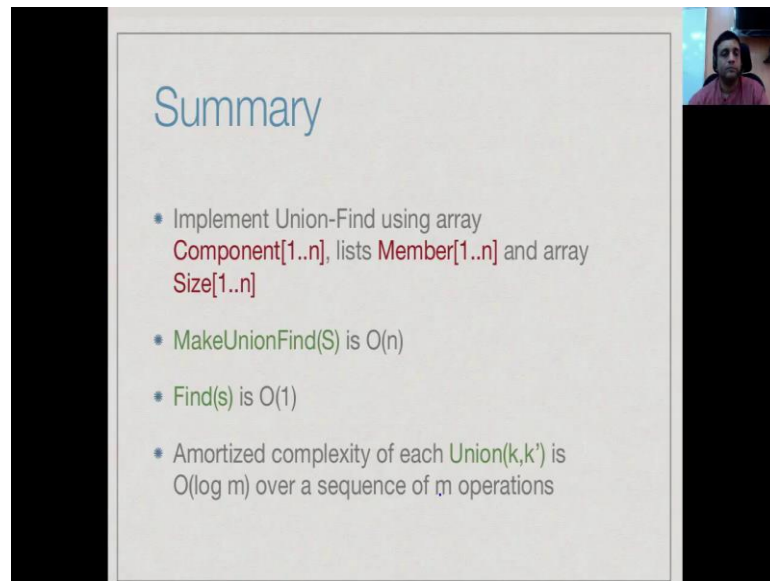
Back to Kruskal ...

- Tree has $n-1$ edges, so $O(n)$ Union() operations
- $O(n \log n)$ amortized cost overall
- Sorting the edges initially takes time $O(m \log m)$, but m is at most n^2 , so equivalently $O(m \log n)$
- Overall this gives $O((m+n) \log n)$, which is same as Prim's algorithm using heaps (to be done soon)

So, since that tree has only n minus 1 edge, we will only add n minus 1 edge out of the total set to find the spanning tree. So, therefore, we have order n union operations to come over all and v k . Now, therefore, for any m operations we said into will take $m \log m$. So, for m operations we have an overall amortize cos if $n \log n$, right, so maintaining the partitions and adding them in all that takes $m \log n$. Now, initially we have to sort the edges. So, that takes $m \log m$, but as we discussed when we actually look at Kruskal in detail, since m is at most n square, we know that $\log m$ is $2 \log n$ at most. So, at the level of orders of magnitude $\log m$ and $\log n$ at the same, so we can look at $m \log m$ as same as $m \log n$. So, the total cost now comes out to $m \log m$, sorry $m \log n$ plus $n \log n$. So, we get m plus $n \log n$.

If you remember the complexity that we claimed for Prim's algorithm and also, actually for these algorithms which are similar to prim's in structure using heaps, we claim that those who are both of that type m plus n , if we use the heap to do the minimum distance calculation. So, therefore, Kruskal's algorithm with the union find data structure essentially has the same complexity as prim's algorithm using heaps.

(Refer Slide Time: 21:28)



Summary

- Implement Union-Find using array
`Component[1..n]`, lists `Member[1..n]` and array
`Size[1..n]`
- `MakeUnionFind(S)` is $O(n)$
- `Find(s)` is $O(1)$
- Amortized complexity of each `Union(k,k')` is
 $O(\log m)$ over a sequence of m operations

So, to summarize what we have seen is that we can implement Union-Find using array to components and array of list to name the vertices in each components, and another array to keep track of the size of each component. With this the initialization step of making the union find data structure of disjoint individual element partitions is order n find takes constant time. Amortize complexity for a sequence of m operations is $m \log n$. So, we can think of each union operation is taking $\log m$ time for a sequence of search m operations starting from the initial.