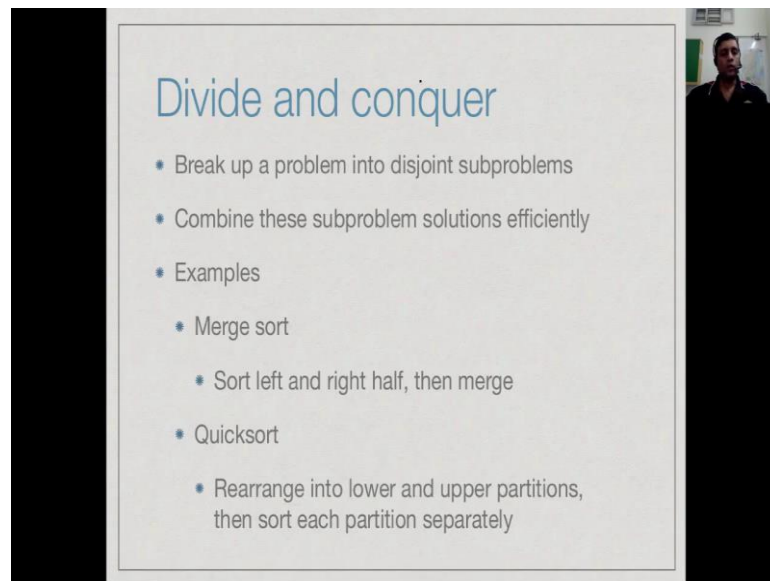


Design and Analysis of Algorithms
Prof. Madhavan Mukund
Chennai Mathematical Institute

Module – 6
Lecture - 37
Divide and Conquer: Counting Inversions

Let us go back and look at Divide and Conquer again.

(Refer Slide Time: 00:06)



The slide is titled "Divide and conquer" in a blue font. It contains a bulleted list of points:

- Break up a problem into disjoint subproblems
- Combine these subproblem solutions efficiently
- Examples
 - Merge sort
 - Sort left and right half, then merge
 - Quicksort
 - Rearrange into lower and upper partitions, then sort each partition separately

In the top right corner of the slide, there is a small video inset showing a man in a dark shirt speaking.

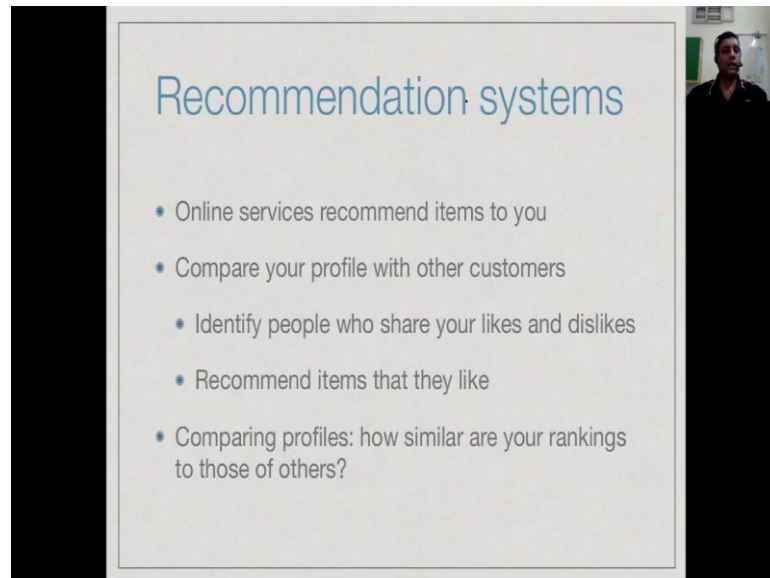
So, recall the divide and conquer paradigm consists of breaking of a problem into disjoint subproblems. Then, we solve each of these subproblems separately and then we combine them efficiently to form a solution to the original form. So, we have seen two examples of divide and conquer, merge sort is a classic example of divided conquer, where we divide the list to be sorted of the array to be sorted into equal parts.

We sort these two parts separately and then, we efficiently merge them, it was sorted list. Quick sort has the different strategy, what it tries to do is avoid the merging step. So, you rearrange the original list, so that you have a lower and upper partition with respect to a pivot. Having rearranged them, you can sort the lower half and the upper half independently and now, because they already rearranged, you do not have to merge them.

So, basically there is a cost involve with setting up this sub problems and a cost to involved with combining the subproblems. And if this set up cost and combination cost

is efficient, then the overall solution gives you something much better than a direct approach could.

(Refer Slide Time: 01:15)



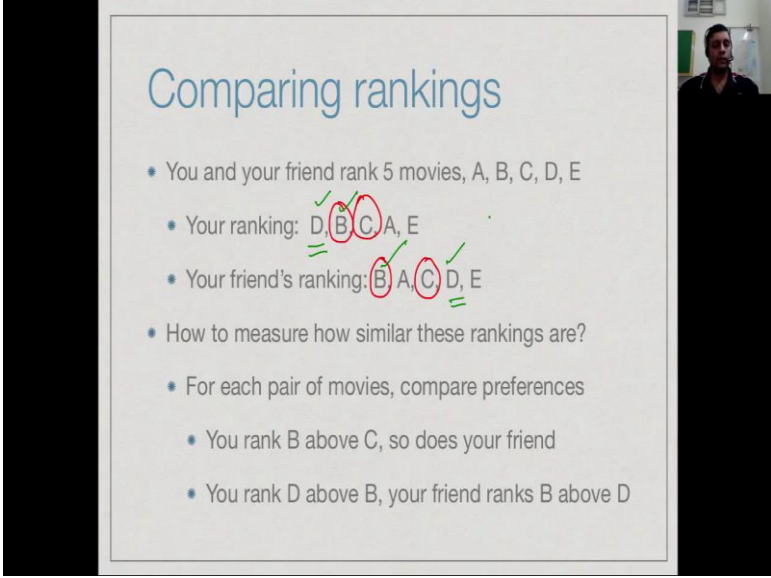
Recommendation systems

- * Online services recommend items to you
- * Compare your profile with other customers
 - * Identify people who share your likes and dislikes
 - * Recommend items that they like
- * Comparing profiles: how similar are your rankings to those of others?

So, let us look at the following situation, very often when you go to an online store, you find a recommendation. For example, it would say, the customers like you who were interested in books like these or customers who bought this phone also look for this pair of head phones. So, these services are recommended to you based on your profile, your online service maintains some profile information about what you like and what you do not like. It compares what you likes and do not like with others and identifies the similar category of people.

And then, it looks for products or services that category has opted for, which you have not and then recommends these three. So, fundamental step in such a recommendations system is that of comparing profiles, how does one persons likes, how do one persons likes and dislikes compared to those of others.

(Refer Slide Time: 02:13)



Comparing rankings

- You and your friend rank 5 movies, A, B, C, D, E
- Your ranking: D, B, C, A, E
- Your friend's ranking: B, A, C, D, E
- How to measure how similar these rankings are?
- For each pair of movies, compare preferences
 - You rank B above C, so does your friend
 - You rank D above B, your friend ranks B above D

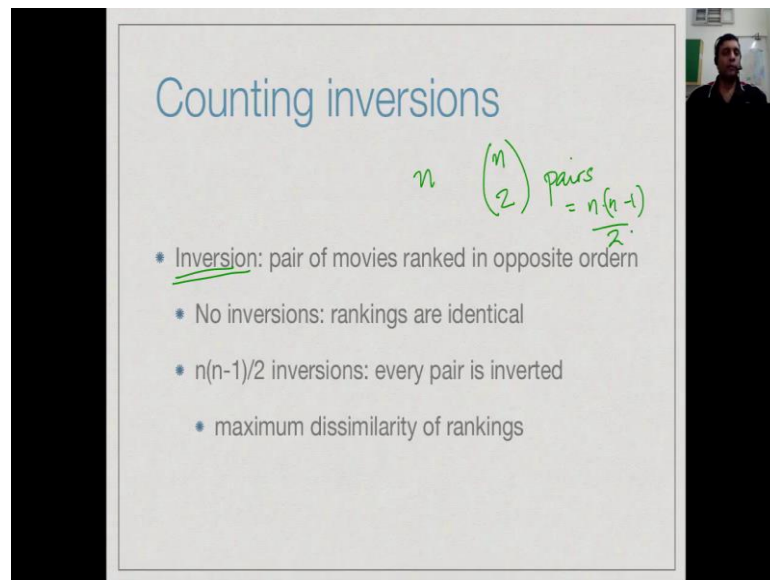
So, one instance of this is when you have preferences over things like movies or books, so suppose over the sequence of time, you have gone to some website and entered your preferences about movies that you watch. So, say there are five movies, let us just call them A, B, C, D and E which both you and somebody else have ranked on this website.

So, you in general two individual who come up with different rankings. So, perhaps you rank D first and E last and your friend ranked B first and E last. So, you both agreed that E was the worst, but you disagree on whether B or D was better. In fact, your friend thinks D was so bad, that is actually only next E and from the bottom. So, D which was your first has become your friends 4th.

So, now what we can do is we can take two such sets of rankings and ask how similar or dissimilar they are. So, one way of measuring this is to compare how you rank pairs of movies. So, for each pair of movies, you can compare whether you rank one better than the other and your friend also does or you done. So, here for instants, if you look at B and C, then this is ranked in a similar way by you and your friend.

On the other hand, if you looked at D and B, then in one case, you rank D above B and your friend rank B above D. So, we do not particularly care how far a part there are, how many other things there are, we are just saying given choice of two things, which do you would prefer, which does your friend prefer and combine the choices across all the choices available in the given list.

(Refer Slide Time: 03:54)



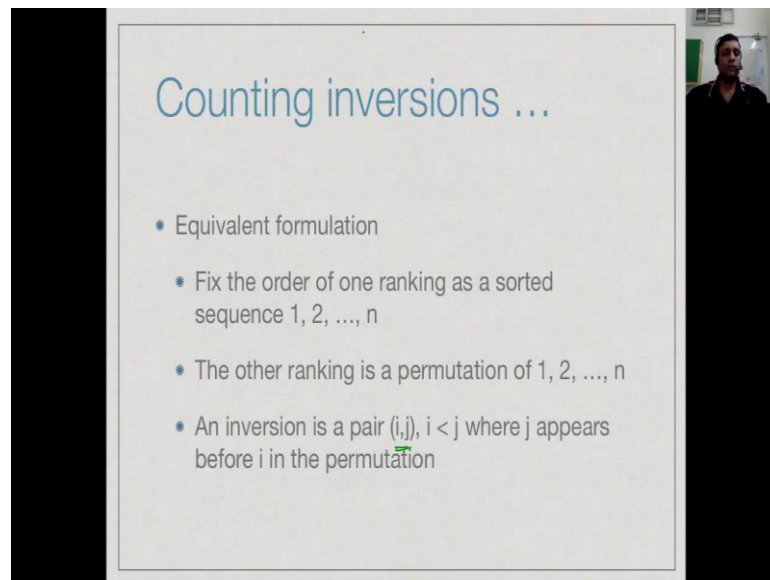
The slide is titled "Counting inversions" in blue text. To the right of the title, there is a handwritten note in green: $n \quad \binom{n}{2} \text{ pairs} = \frac{n(n-1)}{2}$. Below this, there is a bulleted list:

- Inversion: pair of movies ranked in opposite order
- No inversions: rankings are identical
- $n(n-1)/2$ inversions: every pair is inverted
- maximum dissimilarity of rankings

So, what we are trying to do is measure the dissimilarity in terms of what we call inversion. How many pairs of movies or rank in the opposite way between you and your friend? So, if you and your friend rank every pair of movies in the same order, then your total order of performances must be the same. So, if there are zero inversions, then you have exactly similar in your taste to your friend and the rankings are identical.

On the other hand, if you have n movies, then you can do n choice, n choose to pairs. So, the number of different pairs of movies are n choose 2 which is n into n minus 1 by 2. So, if every possible movie you disagree with your friend, then the number of inversions will be n into n minus n by 2, which is an order n squared. So, you can use now this as a measure, how many pairs are out of order, as a measure of how similar or dissimilar, two sets of rankings are and this could be used for instance in this recommendation since to decide which customers to compare in making a recommendation. So, you only want to pick customers who are close to the one, whose recommendation is being made. There is no point recommending something that the personal is not going to like because you comparing it with somebody who has very different taste.

(Refer Slide Time: 05:10)



Counting inversions ...

- Equivalent formulation
 - Fix the order of one ranking as a sorted sequence $1, 2, \dots, n$
 - The other ranking is a permutation of $1, 2, \dots, n$
 - An inversion is a pair (i, j) , $i < j$ where j appears before i in the permutation

So, we can formulate this in another way. So, now, we take our ranking and we assume that is the given order. So, we pick the certain order for the movies and we call that the basic ranking $1, 2, 3, 4$ up to n . Now, our friends ranking would rank what we called 1 as may be 5, what we call 2 is may be 3 and so on. So, everything that we rank with a rank i will be ranked where different rank j by our friend and of course, every rank will appear there somewhere are the other.

So, the friends ranking will be a permutation of 1 to n and what you are asking is if I rank i before j , that is before i is the smaller number than j , does the friend rank j before i , any such think would be an inversion. So, inversion would be a pair i comma j in my list, where i smaller than j . So, my list i is update of j , but in my friends list, j appears before i in the add permutation.

(Refer Slide Time: 06:07)

Counting inversions ...

- Your ranking: D, B, C, A, E
- $D = 1, B = 2, C = 3, A = 4, E = 5$
- Your friend's ranking: B, A, C, D, E
- Corresponding permutation — 2, 4, 3, 1, 5
- Inversions are (1,2), (1,3), (1,4), (3,4)

The diagram shows the permutation 2, 4, 3, 1, 5. Red arrows indicate inversions: from 2 to 1, from 4 to 1, from 3 to 1, and from 3 to 4. Green arrows show the original order 1, 2, 3, 4, 5.

Let us look at this little more concretely. So, supposing this was our original example. So, there was 5 movies A, B, C, D, E and I rank them and you rank them is D, B, C, A, E. Then, I would say D is 1, B is 2, C is 3 A is 4 and E is 5. So, this is my original list 1 to 5. Now, because I have this correspondence between their movies and that things and the rankings, then I know that the D for a since is 2. So, B is 2, A is 4, C is 3, D is 1 and E is 5.

So, from this list of preferences, I can read write it as a reordering of my ranking and now, we are asking when whether there are pairs like this 2 and 1. So, 2 appear before 1 and my friends list, it appears after 1 obviously in the original list, so this is an inversion. So, 2, 1 is inversion, likewise 4, 1 is inversion, so it is 3, 1, so we have these three inversion, whether we have write them as 1 of 2, 1; 3 of 3, 1, because these are pairs, so it was in order is not important, these pairs of movies a rank oppositely by you and your friend.

And the final inversion in this particular example is 3 and 4, so 3 and 4 appear in opposite order, you can check therefore, every other pair for example, 4, 5 or 2, 3 or 2, 4 the order is present. So, there are four inversions in this particular list between your ranking and your friends ranking and our goal is to count this number of inversions given to permutations.

(Refer Slide Time: 07:43)

Graphically ...

- Your ranking: 1, 2, 3, 4, 5
- Your friend's ranking: 2, 4, 3, 1, 5

- Every crossing is an inversion
- Brute force: check every (i,j) — $O(n^2)$

So, another way of thinking about this, though it will not materially affect how we compute it, is to draw this kind of a graph. So, you take the rankings to start with as the correct order on top. So, you have one set of vertices 1 to 5 and then, you have to permutation of the vertices 1 to 5 listed in the order of your friends ranking. And then, you combine 1 to 1, 2 to 2 and so on, so that you build up in the graph which as if you have 5 on top and 5 in the bottom, you have 5 edges, if you have n and n you have n edges.

Now, in this graph every time a line crosses this indicates a mismatch, so 2 has a gone ahead of 1, likewise 4 is gone ahead of 3 and so on. So, there are 4 crossing is between these lines and that really corresponds to four inversions in this example. So, now, there is a very simple brute force way to check, because we know that every inversion is a pair i, j , such that j appears before i in my friends list.

So, we can just check that, we can just check for every i and every j which is different from i , whether i and j is an inversion and this will give us a Brute force order n squared algorithm. So, this actually enumerates all the inversions, it checks every possible pairs and if it is an inversion it says yes, if it is not an inversion it says no and then, you count how many inversions you solve. And we saw and that we can actually in the worst case have complete set n into n minus 1 by 2 inversions. So, this will exhaustively enumerate every inversion in check it with yes or no.

(Refer Slide Time: 09:15)

Divide and conquer $i < j$

- Consider your friend's permutation $[i_1, i_2, \dots, i_N]$
- Divide into two lists
- $L = [i_1, i_2, \dots, i_{N/2}]$, $R = [i_{N/2+1}, i_{N/2+2}, \dots, i_N]$
- Recursively count inversions in L and R
- Add inversions across L and R
- How many elements in R are bigger than elements in L?

Handwritten notes: $i < j$, $[i_1, i_2, \dots, i_N]$, $Solve$, $Combine$

So, our whole is to give a more efficient algorithm, so we will move to this divide and conquer paradigm. So, suppose your friend's permutation, our permutation always 1 to n, so we can just assume it is given. So, what is early interesting is our friends permutation, so the friends permutation is some order of 1 to n jumbled up, let us call it i_1 to i_n .

So, now, we will do something similar to merge sort, so you will take this list i_1 to i_n and divide in two parts. So, we have i_1 to $i_{n/2}$ which is the left and $i_{n/2+1}$ to i_n which is the right. So, divide and conquer is a very simple minded strategy, you can only do one thing, you can solve this and then, you can combine. So, this is the basic paradigm, so you have to divide, solve the divided parts and combine.

So, we will recursively assume that we can count the inversions in left and right. Now, what is left to count are those inversions which cross the boundaries. So, is there at j, i pair that is looks like this. This would not be counted when I count only the left, because i is not in the left, it will not be count only in the right, because j is not in the right. So, there would be an inversion give i is less than j as numbers, but j appears in the left, i appears in the right.

So, this has to be done after we are solve the recursively, so this is basically the combinations step, how many elements in the right are bigger than elements in the left. Anything on the left, if it is smaller than something on the right, then it is not an inversion, because it is already in the correct order in the overall list. But, it something

on the right is bigger than something on the left, then that is an inversion, we have to count all such pairs.

(Refer Slide Time: 10:55)

Adapt merge sort

- Divide $[i_1, i_2, \dots, i_N]$ into two lists
- $L = [i_1, i_2, \dots, i_{N/2}]$, $R = [i_{N/2+1}, i_{N/2+2}, \dots, i_N]$
- Recursively sort and count inversions in L and R
- Count inversions across L and R while merging
- merge and count

Diagram: Two boxes labeled L and R. Arrows point down to sorted L and sorted R. Below sorted L is count L, and below sorted R is count R. Red plus signs and a red arrow point from count L and count R to merge count.

So, in order to solve this, we will make or recursive procedure a little stronger than just counting. So, we will assume that not only does count in the two halves; we sort them well we were counting. So, what happens is now we have divide our problem in two parts and then, we come back, so this is my L and this is my R, I have now sorted L sorted R and I have a counters, so I have a count L and a count R.

Now, the factors these are sorted, means that I can do some kind of merging. So, I can use a version of merge. So, we will describe a version of merging which allows us to count. So, this gives as other count and then, we have three counts the left count the right count and the count return by merge and so what you want to a merging is to actually check how many elements on the right or bigger than how many elements on the left.

(Refer Slide Time: 11:57)

Merge and count.

The diagram illustrates the merge process of two sorted arrays, L and R, into a single array. Red arrows indicate the pointers moving through the arrays. A small diagram shows two elements, i and j, with i < j, and a box containing i, j, and a cross, representing an inversion.

- $L = [i_1, i_2, \dots, i_{N/2}]$, $R = [i_{N/2+1}, i_{N/2+2}, \dots, i_N]$, sorted
- Count inversions across L and R while merging
- Any element from R added to output is inverted with respect to all elements currently in L
- Add current size of L to number of inversions

So, how do we do this? So, what is the principle of merge and count? So, remember that an inversion across L and R consist of an element in R. So, we have an element in R and an element in L, such that this number is smaller than this number. So, we have some i here and some j here, such that i is smaller than j. So, what will happen in a merge procedure is at some point, so we are merging, so we pick the smaller of these two and pulled out.

So, any time now if I had pulled out an element from here; that means, at this current pointer I have merge up to this an up to this. So, there are two pointers in my list left and right, sorted list up to which the merged is proceeded so far. Now, at this point I choose the right hand side element, because it is smaller. So, if it is smaller than, it is smaller than everything which I not yet looked at, that is why, because it is smaller than the first element I am looking at in L. Therefore, smaller everything else in L, because everything else in L sorted ((Refer Time: 13:04)).

So, therefore, this entire segment which is left in L, corresponds to elements which are smaller than the current element am pulling out of it. In other words, this element in R contributes as many inversion as there are elements in L at the point when it is extracted in the merge process. So, whenever I add an element from R to the output, it is inverted this pick to the all the elements currently in L. So, I should add the current size of L to the number of inversions.

So, this gives as are merge and count, while we are merging, every time we pull from the left, there is no inversion, every time pull in the right, we see how many elements as to remaining in the left and that many items need to be added to our inversion part.

(Refer Slide Time: 13:50)

Merge and count

```

function MergeCount(A,m,B,n)
// Merge A[0..m-1], B[0..n-1] into C[0..m+n-1]
✓ i = 0; j = 0; k = 0; count = 0;
// Current positions in A,B,C and inversion count

while (k < m+n) ✓
// Case 1: Move head of A into C, no inversions
if (j==n or A[i] <= B[j])
C[k] = A[i]; i++; k++;
// Case 2: Move head of B into C, update count
if (i==m or A[i] > B[j])
C[k] = B[j]; j++; k++;
count = count + (m-i)
return(count,C)
  
```

So, here is a merge procedure for merge and count which is very similar to the merge procedure in the basic merge set. So, we had two list A and B to be merge, both are sorted and A has m elements and B has n elements and we want to produce an output list C, which has m plus n element. So, we begin with signing of pointers to tell us how call we are born on each list, we just set as 0. And now, we have to keep track of the number of inversions, so we keep a variable called count which is initialized 0, the total number of inversions which have been seen so far.

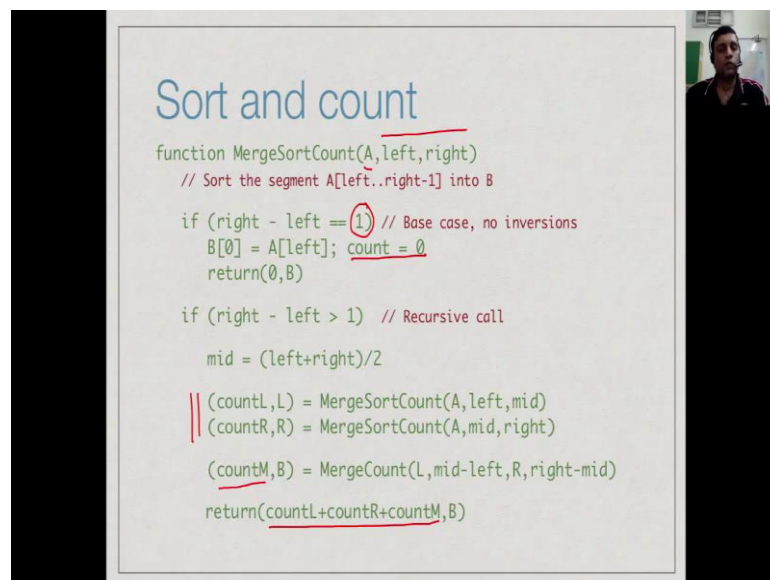
So, along the something is there to move into C, we move something, so there are two cases, the first cases to move from A. So, I have B is empty, j is equal to n are the element of the head of A, A i is smaller than equal to B j in which case we do the usual think, we copy the ith element of A into C k and then, we increment both again n k. The other cases when either A is an empty, i is equal to m or B j as a smaller value, in this case we have possibly an inversion.

So, how many inversions do we have, we have exactly m minus i inversions, we have as many inversions as so this is i, we have as many inversions as are elements currently in A, which is n minus 1. Now, notices that in the specific case where we are copying from

B, because A is empty, we have i equal to m. So, this would actually be 0. So, they should not be an inversion, if they are just duplicating B and C, because A is exhausted.

So, that is also taken care of the m minus i being 0, so although we are updating the count, we are assuming not adding in it. So, only we are doing a nontrivial merge that is when we are moving something from B, when A is still not empty to increment the count, we are moving because we are run out the elements in a count to remain the same. So, the end of this merge procedure, we return these number of inversions we saw plus the merge list.

(Refer Slide Time: 15:48)



```
Sort and count

function MergeSortCount(A, left, right)
  // Sort the segment A[left..right-1] into B

  if (right - left == 1) // Base case, no inversions
    B[0] = A[left]; count = 0
    return(0, B)

  if (right - left > 1) // Recursive call
    mid = (left + right) / 2

    (countL, L) = MergeSortCount(A, left, mid)
    (countR, R) = MergeSortCount(A, mid, right)

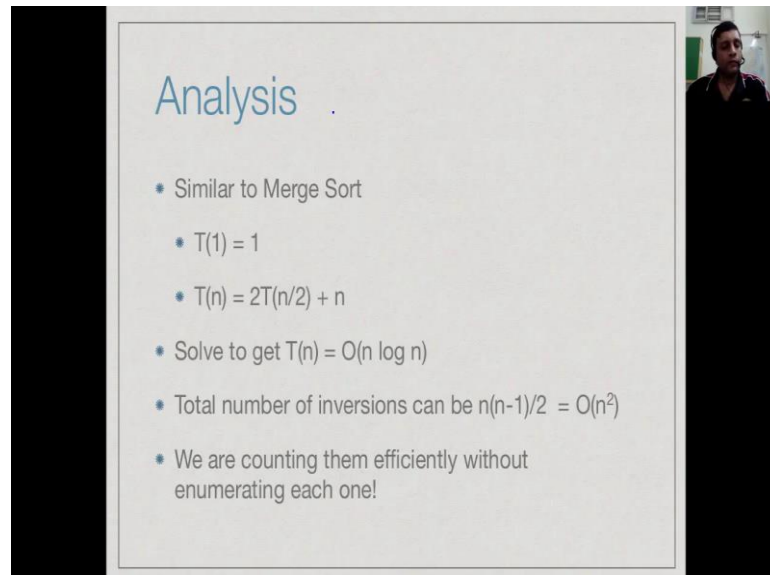
    (countM, B) = MergeCount(L, mid - left, R, right - mid)
    return(countL + countR + countM, B)
```

So, now, we incorporate our merge procedure into the merge sort with counting as follows. As usual, we will sort in general an array from some left index to some right index, because we would sort different segments and different times. So, if the current segment is to be sorted as length 1, then there is nothing to be done, we just set up a new, the sorted segment is just the value that we see and there is no inversion, because it is only one value.

On the other hand, if it is bigger than 1, then we compute the midpoint and we do these two recursive calls, each of them will return the count on the left and right respectively, then we call our merge procedure and get a count from the merge section. So, then our total number of inversions is the count from the left and the right count from the merge together and the new array is the one returned by the merge. So, this is the very simple

extension of merge sort which allows us to count inversions which is useful for a recommendations system.

(Refer Slide Time: 16:47)



Analysis

- Similar to Merge Sort
 - $T(1) = 1$
 - $T(n) = 2T(n/2) + n$
- Solve to get $T(n) = O(n \log n)$
- Total number of inversions can be $n(n-1)/2 = O(n^2)$
- We are counting them efficiently without enumerating each one!

So, the analysis of course, given that the structure resources is similar to merge sort, the analysis is similar, we have this recursion for the time taken on n steps. So, T of 1 is 1 and T of n is 2 times T n by 2 plus n exactly. So, merge sort because it only take as a linear time to merge sort with count. So, you merge with count it takes a same amount of time as merging without counted, so we solve this and we get $n \log n$.

Now, an important think to note is when we did are Brute force calculation, we looked at every possible pair and decided whether are not to is an inversion, so they actually explicitly counted the inversions to get the answer. Now, here we are doing it an $n \log n$ which is potentially much smaller than the number of inversion you want to get. So, we are actually counting the number of inversions or estimating are actually calculating how many inversions are there without actually counting them manually one by one. Because, they could be a n squared inversions, but use there are n squared inversions, we could find that out in $n \log n$ time. So, we not manually counting every step, rather we are getting this through a recursive calculation.