**Week - 02**
**Module - 05**
**Lecture – 13**
**Merge Sort**

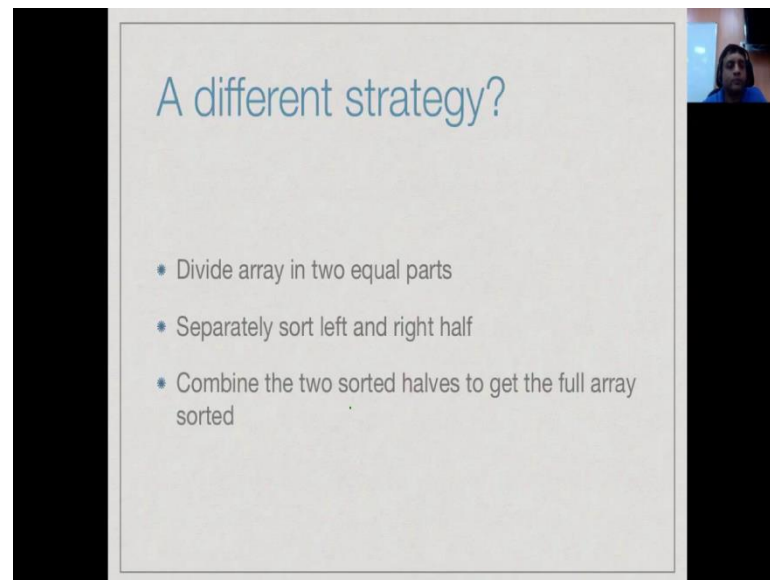(Refer Slide Time: 00:06)



So, we have seen two intuitive sorting algorithms; selection sort, and insertion sort, but unfortunately for us, both of then turn out to be order n square. And we know that order n square is not really good enough for sorting large arrays. So, what can we do instead?
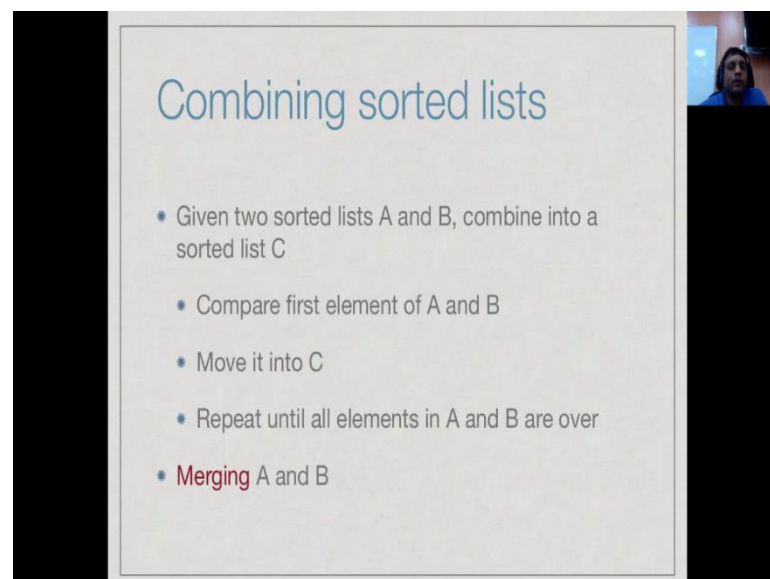
(Refer Slide Time: 00:18)



So, here is one way to sort an array more effectively. So, suppose we divide the array into two equal parts. So, we just bracket in the middle, we look at the left separately and the right separately. Now assume that we can sort to left and right into independently sorted halves. So, we have the left half sorted, the right half sorted. Now if there is a way to combine the two halves efficiently to get a fully sorted array, then we can say that we have achieved the sorting by breaking it up into two smaller sub problems and combining the result.

(Refer Slide Time: 00:54)

So, let us first look at this combining step. So, we are given two sorted list a and b or two sorted array is a and B, and we want to combine them into a single sorted list. So, this is something that we can easily do. Supposing we have two stacks of cards in front of us. Each of them is range in the same way in ascending order. Then what we will do is we look at the top most cards needs stack, and take the smaller of the two, and move it to new stack. And we could keep doing this, until eventually we are move all the elements into a new sorted stack. So, this is an intuitively merging. So, I have two stacks. So, I have two stack is of cards, right. So, I will look at the top most card in each, and then one of them goes to the new stack. And then this gone and so now I will look at the next value, and compare it with this, and then one of these goes here and so on. So, eventually I build up a new stack of sorted list.

(Refer Slide Time: 01:53)
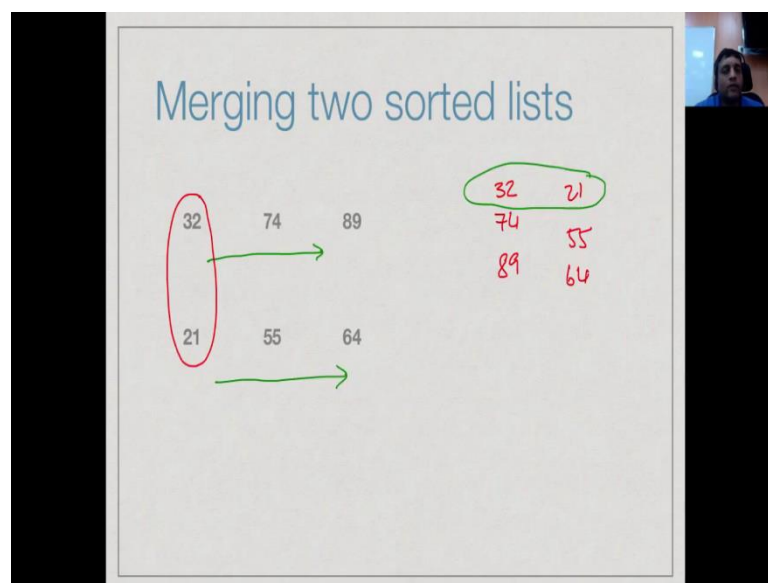


So, let us look at this in a very simple example. So, supposing you want to merge the two sorted list. So, this is sorted in ascending order and so is this sorted in ascending order. So, the first step, is to look at the top most. So, let us assuming that in terms of top most these are written like this. So, I have this stack, and I have this stack. So, I look at the top most elements, and then I say that the smaller of the two must be the top most element of my new stack.

(Refer Slide Time: 02:23)
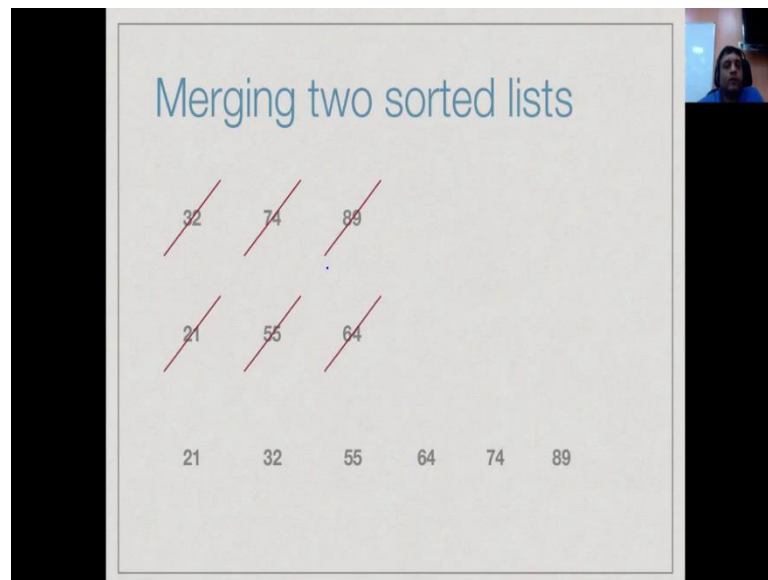


So, in this case I take 21 out, and I move it here. Now I have to compare, what is the top most elements in the two stacks; can be 32 and 55. So, now, I compare this smaller the 2 as 32 and move it out. Now I compare 55 and 74, and so 55 moves out, and I compare 74 and 64. So, now 64 move out. Now I have nothing left in the second stack, so the first stack is in sorted order, so I just copy it out in order. So, first I moves 74 and then I move 89. So, this is in very intuitively merging thing that we do again quite naturally when we are a dealing with physically two sorted list, and we can do it with a normal array as well.

(Refer Slide Time: 03:05)
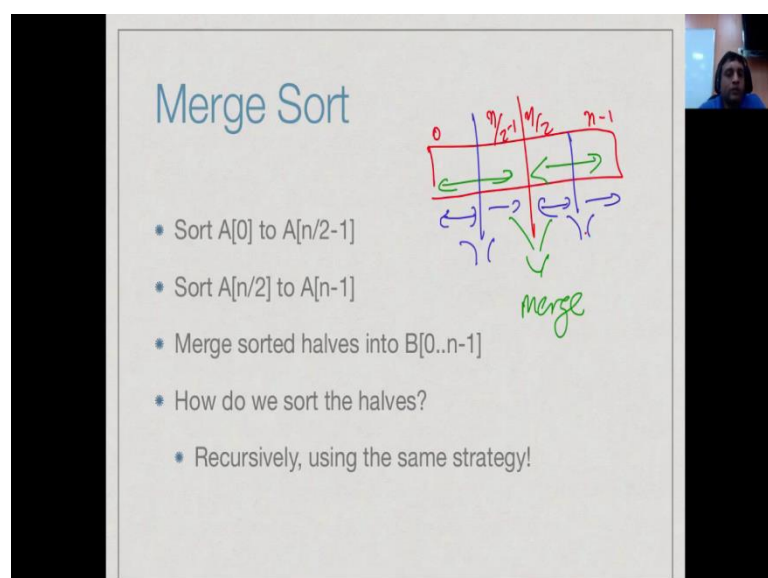
So, now, how do we use this to sort. As we said our aim is to break up problem into two equal sub problems. Solve the sub problems, and there merge the two solution into a final solution. So, we will sort a 0 to a n by 2 a n by 2 n minus 1 to make it distinct. So, we have a with indices 0 to n minus 1. So, we take n by 2 minus 1 and n by 2 as a midpoint. So, we sort this separately, we sort this separately, and then we merge them. So, this is the strategy that we have, and this is since the final step involves merging two solutions. This is quite naturally call merge sort. Now I have said that we will break up the thing in two sub problems. So, how do I solve this. Well I will recursively do same thing. I will break this two submit to two sub problems, and I will merge this. I will break this happen to two sub problems and merge this. So, I keep breaking up to the thing into sub problem, until you reach at trivial the sub problem, which is as we have seen, which is an array of size of 1.
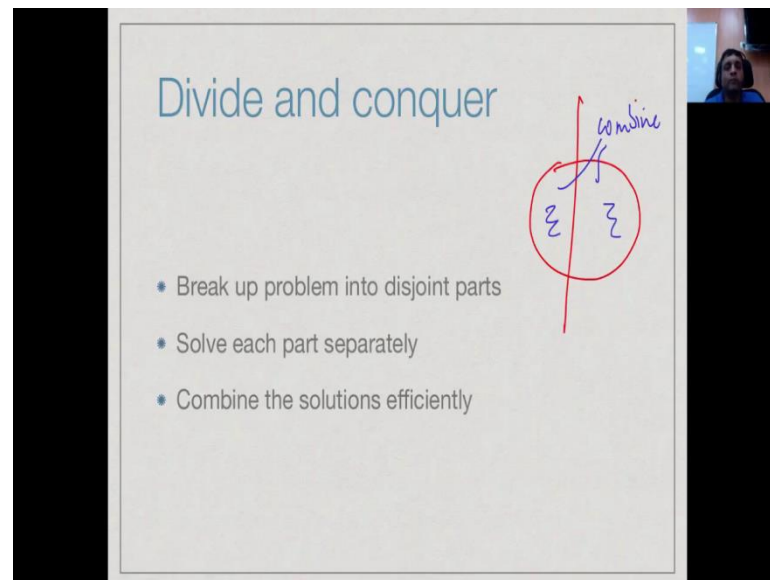
(Refer Slide Time: 04:10)
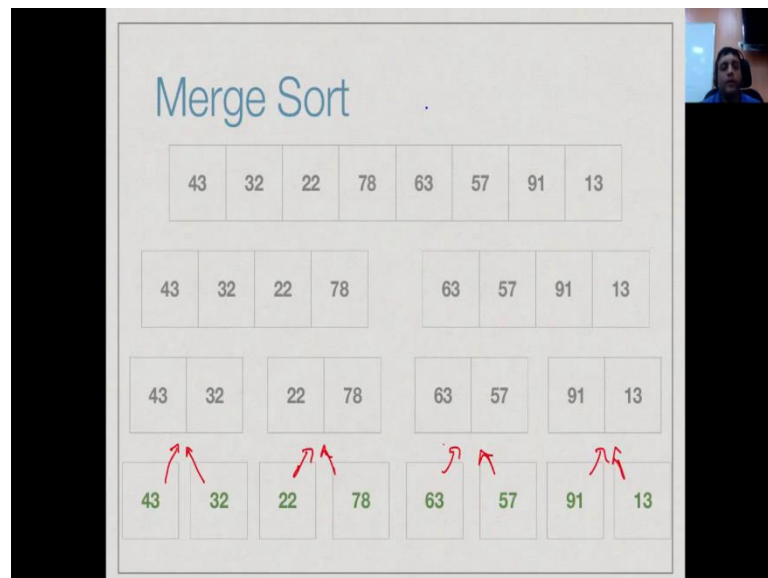


So, let us look at an example before we proceed. So, here is an example of an array that we would like to sort. So, the first step is to break it up into two parts. So, we take the left part. So, this is a midpoint. So, we take the left part, which is a first four elements, and the right part. And now we will apply merge sort separately to these part; finally, we will merge the answer. So, having applying merge sort to the left part, you must again break it up to two parts. I would take this point in dividing into two parts. And similarly on the right, I will have to take this point, and divide into two parts. Now we could say that we can easily do array of size two, but let us just keep going till we reach the base

case. The base cases when we have only one element and no sorting is required. We will against split each of these into two parts. So, notice that for convenience we have taken something where I can keep dividing by 2, but there is no limitation in merge sorted, it will work for any size.

At some point when you do an unequal's split, the two halves will not be the same size, but that has really matter. Now we break the last step into two parts. So, 43 is now. So, this is in green to indicate that this is now a single element and hence sorted, so is this. And in the same way we can take each of these last steps and get now 8 single element, which are sorted. Now we start merging. So, we want to merge these two, in order to (( )), and similarly we want to merge these two. We want to merge these two, and we want to merge these two. So, we merge first two, and then we merge this, the smaller then it goes first. So, this is now sorted. Similarly we merge the second pair, it does in change in order, because 22 smaller than 78. We merge the third pair and they get exchanged, and we merge the fourth pair again they get exchanged. There is an important note that exchange did not come by looking at array of size to directly, but rather by looking at these two value, taking the smaller one first then the below one.

Now I want to merge these two arrays into a sorted segment of length 4, and these two arrays it was sorted segment of length 4. So, again apply merging. So, note that 22 will come first then 32 will come here, then 43 will come here, and then 78 to come here. If I apply this I will get 22 32 43 and 78. The same way here 30 should come here, then 57, then 63, and then 91. So, this is the effect of merging. And finally, I have to merge these two. So, this smallest 1 and 13 will go here, then I will 22, then I will 32, then I will get 43, then I will get 57, then I will get 63, and I will get 78, then I will get 91. So, merging these two sub arrays of size four will give me the final answer. So, this is how merge sort works. You break it up in two parts, recursively solved two parts using the same strategy and merge them.

So, this is generally a principle that can be applied to many problems. So, if you can take a problem, and divide it into two or more parts; such that this part can be solved independent of that part, and there is no overlap. You solve this separately, you solve this separately, and now you want to somehow combine. In this particular algorithms combination is merging. We will look later on it other divide and compare algorithms, whether combination might require a different strategy, but the whole ideas is, that you can break up the problem into smaller problems, and then combine them. Then you can sometime get a lot of benefit in terms of efficiency. The crucial thing, is to identify how to break up the problem into disjoin sub problems, and how to combine the solution to these sub problem efficiently to solve the problem at hand.
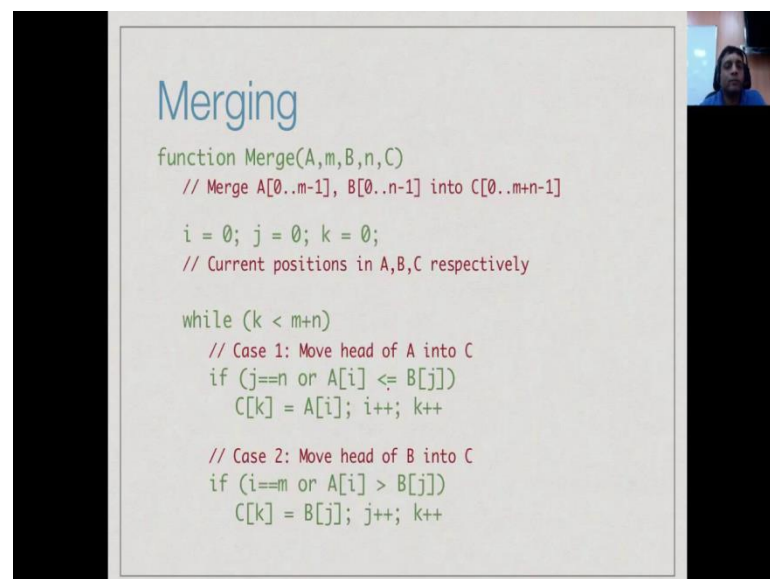
So, let us come back to our merge sort and try to formalize the algorithms in terms of actual code. So, how do I combine two shorted list or two sorted arrays A and B into A third sorted list C. Well as we saw if one of the two is empty, then I do know do anything I just have to copy the other one. So if there is a no element left in A, then I can just copy there is to B into C. If B is a empty I can copy A into C; otherwise we compare the first element of A and B and move the smaller then go into C, and we repeat this until everything has been moved.
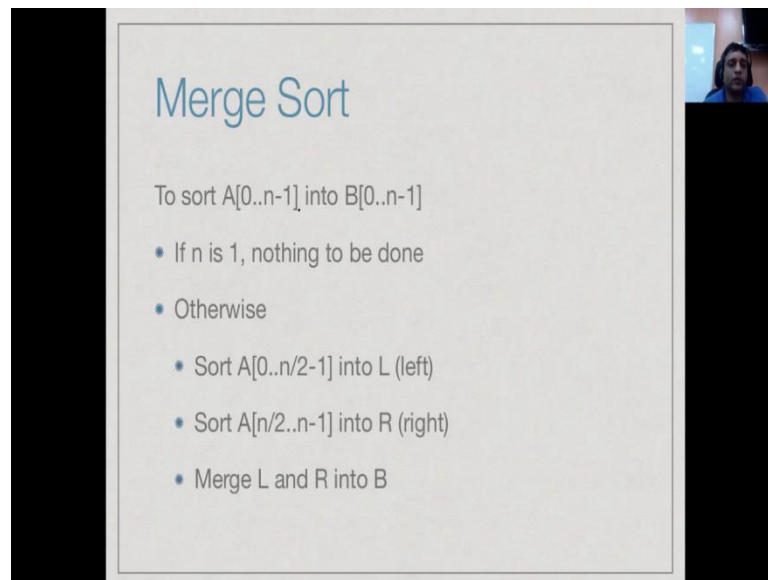
So, here is the simple iterative merge function. So, it takes two arrays as input. Take an array a of size m. So, 0 to n minus 1. Let takes an array B, which may be of different size 0 to n minus 1, and the aim is to construct and new array C. Now we know the size of C, because everything there must come here. So, it will be equal to 0 to m plus n minus 1. So, what we do is, we maintain some position. So, we say that okay let i B the current position I am looking at in this, j B the current position I am looking at B, and k B the current position I am trying to fill in C. So, now, we know that there are n plus 1 steps. So we put a loop which says this think must run n plus 1 m plus n times. So, if we have already reached the end, if j has already is the end, or if A i is smaller than B j. So, this is the real merge step. If A i is smaller or equal to B j what we will do is, we will copy is value here, and then we will increment i and will increment j. So, we will move C k will be A i and i will be incremented and k will be incremented. But this also happens in case j is equal to n. If j is equal to n what it means is that, this pointer actually reached all way there.

So, there nothing left to scan. So, j is equal to n means actually beyond the right point. So, it is beyond the point. So, there is nothing to scan. So, we will just copy everything from A to C. So, either we copy element, current element from a to C if A i is smaller than bj or if there is nothing in B, we copy the current element, because we are just copy everything from a to C. So, the symmetric case is, when we have A i bigger then B j. So, if A i is bigger than B j, then what we want to do is, copy this value here. So, we want to move j up and k up. So, we copy C k is equal to B j, copy the value at B j to C k, and increment both j and k. And like we did earlier, the other reason we might want to do is, as if we have already exceeded the length of. So, if we are currently here. So, A has been exhausted, then we would also move the next element blindly from B to C. So, this is the simple while loop right. It takes exactly as many steps as there are elements to move, and in each step I move one more element to C; either from a or B depending on the criterion of the current element I am looking at.
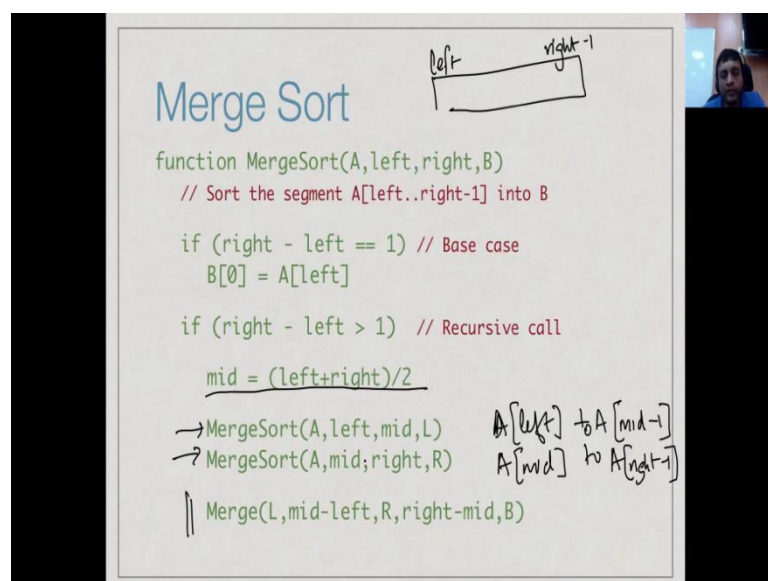
(Refer Slide Time: 10:56)



So, having got merged out of the way. Now, we can look at merge sorted itself. So, if we want to sort a of size n indices C 0 to n minus 1. We have to create a new array as we said, because merging has to copy those two things into a new thing. So, eventually going to take a and sorted into new array B of the same size. So, if n is 1. If we have an element of size one, we have the base case, nothing should be done. Otherwise we will sort the left part into a sub array l. We will sort right half into sub array r for left and right, and then we will merge that two using function we just got.

(Refer Slide Time: 11:38)

So, this is the very clear recursive algorithms. So, we want to merge sort a from left to right. So, we will assume that, and we say from left to right, we mean that the left most position is called left, and the right most indexes is actually right minus 1. So, this is actually one more then the index of the position that we want to sort out. So, we want to sort this segment, starting at a left and going up to A including a minus 1. So, first of all if this segment is of length one, then we just copy the value into B. We copy the value at a left. Otherwise what we do is, we make, we find the midpoint, and then we copy up to, but not including mid. So, this is why we use is the convention that sub to mid minus 1. So, this will be from left a left to a mid minus 1, and this will merge sort from a mid to a right minus 1.

So, what you want to makes sure is, that we are not accidentally over lapping. So, we want to make sure that the mid value which appears in both things. The value at a mid is only in none of them; namely in the right hand side ((Refer Time: 13.02)), so having done this. Then I will use my earlier merge function, to merge l, which is also size mid minus left, and r which is the size right minus mid into B. So, this is the function that we already had. So it is a very simple recursive thing. Find the midpoint, sort the left half, sort the right half and merge them. As we said, it does in really matter that A is of even length or it is a multiple of two. So, it might be that left half and the right half are not of same length; one it will be longer, one will be shorter. It does not really matter. This works in all cases. To analysis this, is not so straight forward, so we will postpone that to the next module.