

Week – 02

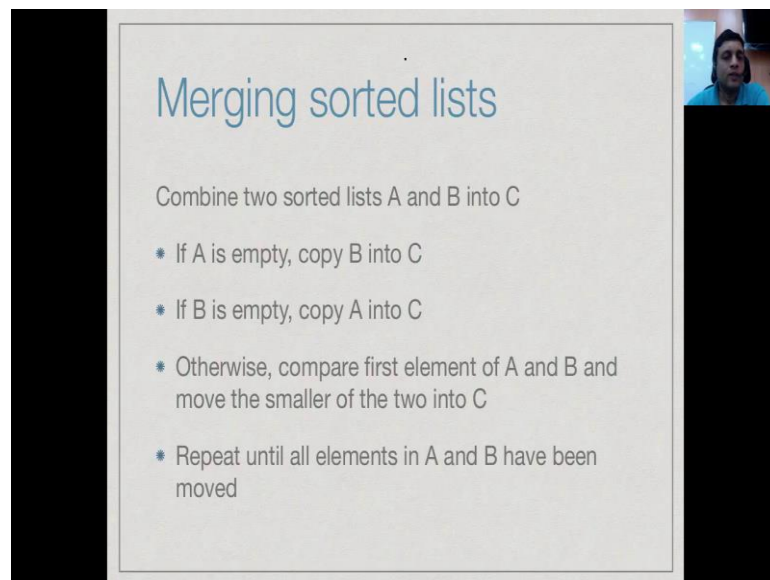
Module – 06

Lecture - 14

Merge Sort: Analysis

So, we have seen how to use a divide and conquer strategy, we implemented sorting algorithm called merge sort. So, now we want to analyze whether the merge sort actually behaves, better than an order  $n^2$  intuitive sorts like insertion sort and selection sort.

(Refer Slide Time: 00:18)



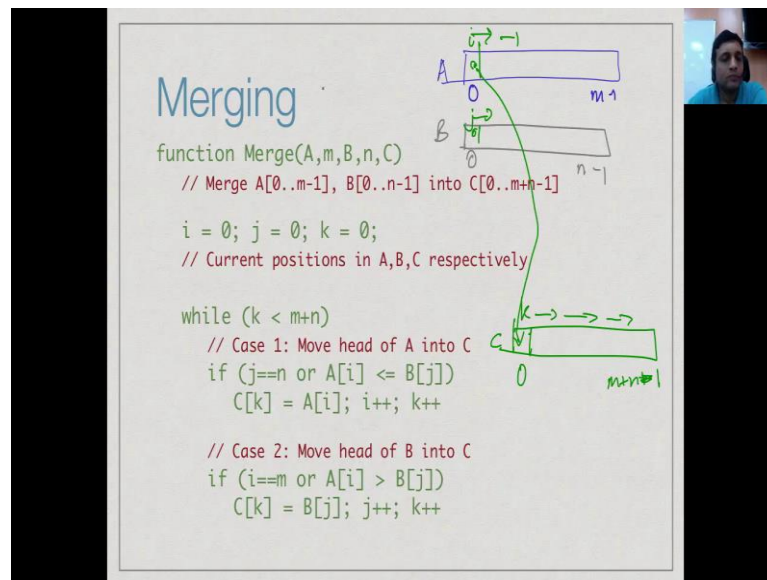
Merging sorted lists

Combine two sorted lists A and B into C

- \* If A is empty, copy B into C
- \* If B is empty, copy A into C
- \* Otherwise, compare first element of A and B and move the smaller of the two into C
- \* Repeat until all elements in A and B have been moved

So, in order to analyze merge sort, we first need to look at the merge operation itself. So, remember that when we merge two sorted list, what we do is, we take both list side by side and we keep copying the smaller of the two elements at the header each list into the final list C.

(Refer Slide Time: 00:37)



**Merging**

```
function Merge(A,m,B,n,C)
// Merge A[0..m-1], B[0..n-1] into C[0..m+n-1]

i = 0; j = 0; k = 0;
// Current positions in A,B,C respectively

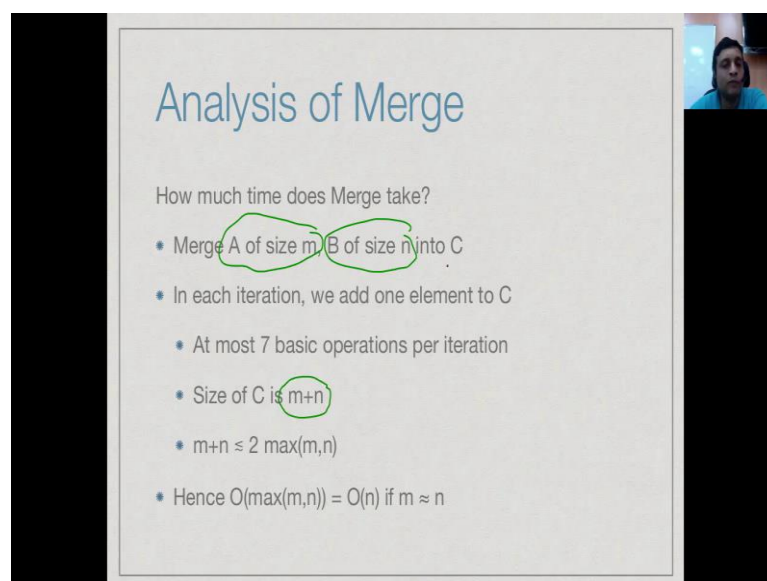
while (k < m+n)
// Case 1: Move head of A into C
if (j==n or A[i] <= B[j])
C[k] = A[i]; i++; k++;

// Case 2: Move head of B into C
if (i==m or A[i] > B[j])
C[k] = B[j]; j++; k++
```

The diagram illustrates the merging process. It shows three horizontal arrays: A, B, and C. Array A has indices 0 to m-1, and array B has indices 0 to n-1. Array C has indices 0 to m+n-1. A green line connects the current indices i, j, and k. i and j are at index 0, and k is at index 0. Arrows indicate the movement of i and j as they traverse their respective arrays. The final state shows i at m-1, j at n-1, and k at m+n-1.

So, this was the code that we wrote last time. So, basically we have a list A, which runs from 0 to m minus 1 and we have another list B, which runs from 0 to n minus 1 and we produce the output in a third list C which has indices from 0 to m plus n minus 1. So, what we do is, we start by putting an index i, j into these two list and index k into the list C and we compare these two values and we take the smaller of the two values and move it. And then we move this and then we keep moving these two indices and all the while we keep moving k. So, the question is how long does this take?

(Refer Slide Time: 01:22)



**Analysis of Merge**

How much time does Merge take?

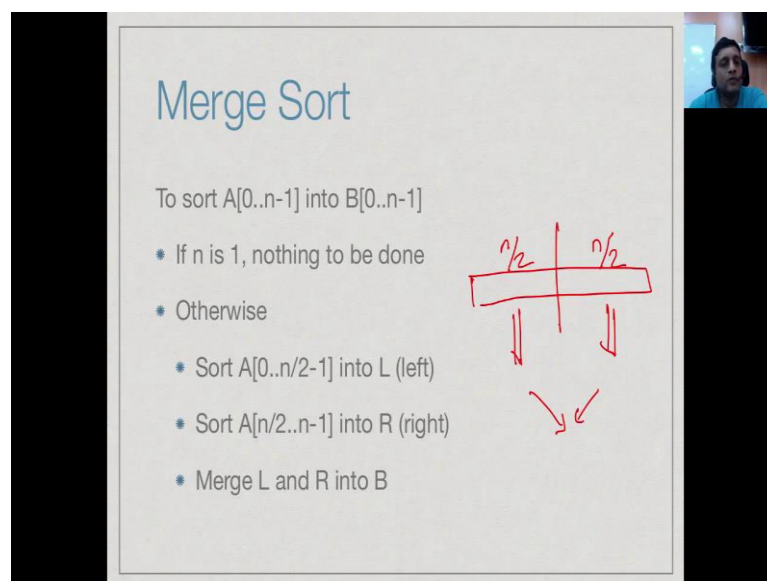
- Merge A of size m, B of size n into C
- In each iteration, we add one element to C
  - At most 7 basic operations per iteration
- Size of C is m+n
- $m+n \leq 2 \max(m,n)$
- Hence  $O(\max(m,n)) = O(n)$  if  $m \approx n$

So, in each iteration notice that we add one element to C, but the size of C is exactly  $m$  plus  $n$ , because there are  $m$  elements in A and there are  $n$  elements in B and each of them will eventually make it to C. So, there are  $m$  plus  $n$  elements and in each iteration of this loop, the loop that we had before, in each iteration of this loop, one element gets added to C, either in this if or in this if,  $k$  is incremented.

So, this list as you can see will make,  $k$  will make progress in every iteration. So, there is a bound of  $m$  plus  $n$  steps for this loop and in the loop, we will if you count very carefully, we have a comparison, we may have one more comparison, we have an assignment and so on. But, you can check that there are no more than seven steps involved, so some constant number of steps regardless, so which branch we taken this, so we have  $m$  plus  $n$  iterations, each of which has some constant number of steps.

So, we have over all order  $m$  plus  $n$  steps, but  $m$  plus  $n$  is of course, bounded by 2 times the maximum of  $m$  plus  $n$ . So, we can just say that this thing takes order of maximum  $m$  plus  $n$ . If  $m$  is roughly the same as  $n$  which will typically be the case in merge sort, because we split it in half, the two will differ by at most 1 in terms of length. Then, we find that merging is a linear time operation, so we want to merge two lists, it takes time proportional to the length of the two lists together.

(Refer Slide Time: 02:46)



## Merge Sort

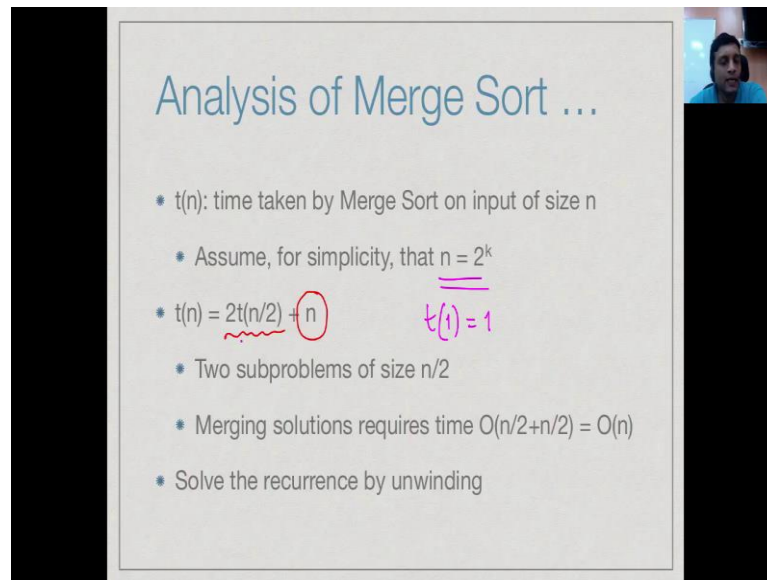
To sort  $A[0..n-1]$  into  $B[0..n-1]$

- If  $n$  is 1, nothing to be done
- Otherwise
  - Sort  $A[0..n/2-1]$  into L (left)
  - Sort  $A[n/2..n-1]$  into R (right)
  - Merge L and R into B

The diagram illustrates the recursive splitting of an array. A horizontal bar is divided into two equal halves, each labeled  $n/2$ . Two arrows point downwards from each half to two separate, smaller horizontal bars, representing the recursive sorting of the left and right halves. A third arrow points upwards from the space between these two bars to a single horizontal bar below them, representing the merging of the two sorted halves into the final array B.

So, now coming to merge sort itself, what we want to do is, we want to take a list of size  $n$  and you want to split it into two lists of size  $n/2$ . And then, we sort these separately and then we merge.

(Refer Slide Time: 03:05)



Analysis of Merge Sort ...

- \*  $t(n)$ : time taken by Merge Sort on input of size  $n$
- \* Assume, for simplicity, that  $n = 2^k$
- \*  $t(n) = 2t(n/2) + n$  (with  $t(1) = 1$  written in purple)
- \* Two subproblems of size  $n/2$
- \* Merging solutions requires time  $O(n/2 + n/2) = O(n)$
- \* Solve the recurrence by unwinding

So, in order to do this, it is very clear that if we look at  $t(n)$  as a time taken by merge sort on an input of size  $n$ , then this requires us to sort two lists of size  $n/2$  and then, as we have seen it takes order  $n$  time in order to merge that. Now, in order to compute this  $t(n)$  explicitly, we will assume that  $n$  is a power of 2. Now, it turns out that merge sort does not in any way require  $n$  to be a power of 2, not even an even number. If it is not even, then when we split it into two parts, they will not be equal, because we cannot split an odd number into two equal parts.

But, it does not matter, the algorithm will still work correctly, this assumption that  $n$  is  $2^k$  is only a signification for us to be able to come up with a nice calculation. So, we have two problems of size  $n/2$ , 2 times of  $t(n/2)$  and then we have a merge of  $n$ . So, how do we solve something that ((Refer Time: 04:00)). So, of course, we also have a base case which says that when we have only one element, it takes no time except to just check that you have only one element, so  $t(1)$  is point.

So, if you have a solution I mean an expression like this two recurrence like this, then how to be solve them. So, the basic idea is to keep the simplest way to do it. Of course, you can come up with more sophisticated way which we will not go into right now. But,



one simple way is you just keep expanding using the same analysis that the same expression, until you come down to the base case.

(Refer Slide Time: 04:31)

**Analysis of Merge Sort ...**

- $t(1) = 1$
- $t(n) = 2t(n/2) + n$ 

$$= 2 [ 2t(n/4) + n/2 ] + n = 2^2 t(n/2^2) + 2n$$

$$= 2^2 [ 2t(n/2^3) + n/2^2 ] + 2n = 2^3 t(n/2^3) + 3n$$

$$\dots$$

$$= 2^j t(n/2^j) + jn$$
- When  $j = \log n$ ,  $n/2^j = 1$ , so  $t(n/2^j) = 1$
- $\log n$  means  $\log_2 n$  unless otherwise specified!
- $t(n) = 2^j t(n/2^j) + jn = 2^{\log n} + (\log n) n = \textcircled{n} + n \log n = O(n \log n)$

*Handwritten red annotations:  $O(n^2)$  and a circled  $n$ .*

So, we start with the base case  $t(1)$  equal to 1 and the general case  $t(n)$  is 2 times  $t(n/2)$  plus  $n$ . So, now what we can do is, we can use the same equation to expand  $t(n/2)$ . So, we expand  $t(n/2)$ , then we find that  $t(n/2)$  becomes 2 times  $t(n/4)$  plus  $n/2$ , because I take whatever here is divide 2,  $n/2$  by 2 is  $n/4$  and I take whatever is here and plug it in here, so  $n/2$  becomes.

Now, if I expand this out carefully, then this 2 and this 2, other than write it as 4 and I will write it as 2 squared, I will also observe and this will be useful that this 4 is again 2 squared and then this  $n/2$  is multiplied, so this  $n/2$  is multiplied by 2. So, I get 2 times  $n/2$ , so that is  $n$  plus  $n$ , so I get 2  $n$ .

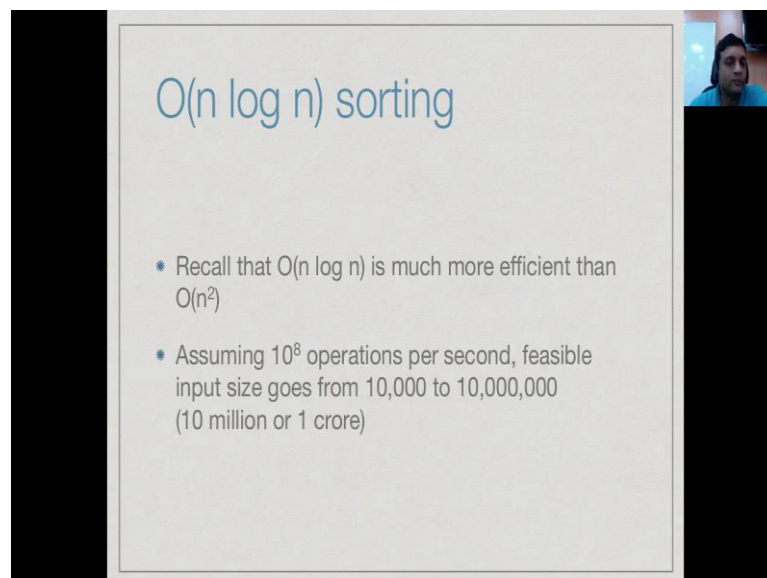
So, it is not an accident that I have 2 squared, 2 squared in 2  $n$  as we will see, if you do one more step, now if I take this expression  $t(n/2)$  square and I apply the same rule to this, then this  $t(n/2)$  square expands out like this. It is  $n/2$  square divide by 2  $n/2$  by 2  $q$ , so I get 2 times  $n/2$  to plus  $n/2$  square. Now, this and this will cancel, so I get 1  $n$  here, I already have 2  $n$ , so I get 3  $n$  and this 2 into 2 square, I will write as 2's cube, so I am going from 2 square  $t(n/2)$  square plus 2  $n$  to cube  $t(n/2)$  cube plus 3  $n$ . So, it is easy to verify that if you do this  $j$  times we will have 2 to the power  $j$  times  $t(n/2)$  by 2 to the power  $j$  plus  $j n$ .

So, now when do we reach the base case, we reach the base case when  $n$  by 2 to the power  $j$  goes to 1. So, when does this happen? Well,  $n$  by 2 to the power  $j$  is equal to the 1, it means that 2 to the power  $j$  is equal to  $n$ , so  $j$  is nothing but,  $\log$  of  $n$  to the base 2. So, we will always use  $\log n$  in general to mean  $\log$  of  $n$  to the base 2, unless otherwise specified.

So, after  $\log$  of  $n$  steps, we end up with something which looks like this, it has 2 to the  $j$  times  $n$  by 2 to the  $j$ , after  $\log n$  steps,  $j$  is  $\log n$ , so 2 to the  $\log n$ , then  $n$  by this, so this is  $n$  times 1 now, so this is just 1, so this multiplied by 1. And then, I have  $j$  times  $n$ , so  $j$  is  $\log n$ , so  $j$  is  $\log n$  and  $n$  is side. So, I have after doing this expansion  $\log n$  times I end up with 2 to the  $\log n$  plus  $\log n$  times  $n$ , 2 to the  $\log n$  by definition is just  $n$ , that is the definition of  $\log$  that is the exponent of 2 which gives you  $n$ .

So, this is  $n$  and I have this gives  $n \log n$  and we know from general things that  $n$  is we go of  $n \log n$ . So, this is bounded by 2 times  $n \log n$ . So, we can say that over all merge sort  $x$  times  $O n \log n$ . So, we have achieved a significant improvement, because remember that, so far both insertion sort and selection sort your  $O n$  square and we have come down from  $O n$  square to  $O n \log n$  by using this divide and conquer strategy.

(Refer Slide Time: 08:06)



$O(n \log n)$  sorting

- \* Recall that  $O(n \log n)$  is much more efficient than  $O(n^2)$
- \* Assuming  $10^8$  operations per second, feasible input size goes from 10,000 to 10,000,000 (10 million or 1 crore)

So, why is this big deal, we saw in the beginning that  $O n \log n$  is much more efficient than  $O n$  square. If we assume as we have said that a reasonably standard desktop machine can do 10 to the 8 operations per second, then the size of the feasible input of

what we can sort with a sorting algorithm, of course from 10,000 for an  $n^2$  algorithm to 10 million or 1 crore for an  $n \log n$  algorithm.

So, therefore if you really have to sort large volumes of data,  $n \log n$  makes it feasible for us to do so in a reasonable amount of time, whereas  $n^2$  would take years as we solved it before.

(Refer Slide Time: 08:43)

**Variations on merge**

- Union of two sorted lists (discard duplicates)
  - If  $A[i] == B[j]$ , copy  $A[i]$  to  $C[k]$  and increment  $i, j, k$
- Intersection of two sorted lists
  - If  $A[i] < B[j]$ , increment  $i$
  - If  $B[j] < A[i]$ , increment  $j$
  - If  $A[i] == B[j]$ , copy  $A[i]$  to  $C[k]$  and increment  $i, j, k$
- **Exercise:** List difference: elements in  $A$  but not in  $B$

Handwritten notes on the slide:

- Top right:  $[1, 2, 4] \rightarrow \{1, 2, 4\}$  and  $[2, 3, 6] \rightarrow \{2, 3, 6\}$
- Below that:  $[1, 2, 3, 4, 6]$  (result of union)
- Intersection conditions:  $A[i] \leq B[j]$  and  $A[i] > B[j]$

So, before we conclude there are some nice things to notice. So, this merge operation which we are used in merge sort, it is actually a very useful operation on any sorted list. So, one thing that we do in merge is that we actually combine the list without losing anything. So, if we merge two lists such, let us say 1, 2, 4 and 2, 3, 6 then in our final thing, we will have 2 copies of 2, because there are 2 copies and then 3, 4, 6.

So, sometimes we may want to not have this, you may want to keep only one copy, we may want to treat this as a set, so if I will take this as set 1, 2, 4 and a set 2, 3, 6, then the resulting union of the two set is 1, 2, 3, 4, 6 you have only one copy of it. Then, we can easily do this in our merge procedure by just saying that when I find, earlier we had two cases, we had  $A[i] \leq B[j]$  and then we had  $A[i] > B[j]$ .

So, in the first case, we copied from  $A[i]$ , the second case we copied from  $B[j]$  and either case we only incremented either  $i$  or  $j$  along with  $k$ . In this case, we say that if the two sides are equal, we keep one copy of it in the final list, but we skip over both these

copies, we increment all three pointers. So, if we add this position and this position, then our output you will copy at 2 and then we will move this pointer to the right and this pointer to the right, so I do not copy 2 twice, so this is one thing that we can do.

The other thing that we can do is to intersect two lists, so if we again have 1, 2, 3 may be 2, 3, 6 then may be you want to make sure that we have in our output only 2 and 3. So, now what we do is, when we start looking at these two, if they are not equal, then obviously they are not in the intersection. But, which one should be leave, well 2 could still be there, because 1 is smaller, so we move this pointer right.

So, if  $A_i$  smaller than  $B_j$  we increment  $i$ , now if say when  $A_i, B_j$  are the same, we would as we did for union, we will move that 2 out and we will merge, we will remove both pointers, so now both pointers come to this position. Now, again we will say that we have 3 and we will continue and then when we see the other way around, if  $B_j$  is smaller than  $A_i$ , we will increment  $j$  and not  $i$ .

So, the important thing to notice that merge is a very generic operation, we can do what we did earlier which is to actually shuffle the two sorted lists into one big sorted list, when no elements are lost or we can remove duplicates, what I will doing is, treated as a set union operation or we can only keep the common pairs which is set intersection operation.

So, to check that you have understood this, maybe you should try out the following exercises. So, supposing we want to do what is called list difference, so set difference is sometime written like this. So, if I take the set 1, 2, 3 and the set say 3, 4, 6 and I ask the set difference, it says whatever is in A, but not in B, so the answer should be in this case 1 and then 2, but not 3, because 3 is there at the end.

So, we can do the same thing for list. So, for example, if I have list 1, 2, 3 and list 3, 4, 6 then I may want to keep the list only 1, 2. So, just check whether you can adopt merge in the way we have done now for union and intersection in order to list difference.

(Refer Slide Time: 12:20)



The slide is titled "Merge Sort: Shortcomings" in blue text. Above the title, there is a diagram showing two boxes labeled 'A' and 'B' with arrows pointing from them to a larger box labeled 'C', illustrating the merging process. Below the diagram, there is a bulleted list of shortcomings:

- Merging A and B creates a new array C
- No obvious way to efficiently merge in place
- Extra storage can be costly
- Inherently recursive
- Recursive call and return are expensive

A small video feed of a person is visible in the top right corner of the slide.

So, merge sort though it is an order  $n \log n$  sorting algorithm and therefore it is significantly faster than insertion sort or selection sort, it does have some short comes. The main problem with merge sort is that it requires extra space. See, when I take A and B and I merge it into C, it is almost impossible to do this without storing the merge array in a separate place. Because, if I stack merging in place, then the sorted order of A and B gets merge sort or I have to keep shifting things and so the merge is no larger in linear time.

So, if I want the linear time merge a way of A and B, this can only be achieved by using an external list of the size which is the sum of two lists in order to store it. This means that if you are sorting large arrays and also extra space is being used, so that to keep duplicating the space in order to the merge operation. And of course, the other thing about merge sort which is very difficult to overcome is that it is inherently recursive and I mean there is no way to actually easily make it iterative, because we need to construct the sorted things and then merge them.

So, we need to store this recursive solution at each point in order to combine. So, the recursive call and return can be expensive also. So, although merge sort is a very attractive sort in terms of it is theoretical complexity, it is not sometimes used in practices, because of these limitation and we will see how to overcome this in some other way.

(Refer Slide Time: 13:40)

The slide is titled "Alternative approach". It features two diagrams. The top diagram shows two boxes: a red box containing "2 4 6" and a green box containing "1 3 5". Arrows indicate elements from both boxes being merged into a single sequence. The bottom diagram shows a red box with "2 1 3" and a green box with "6 4 5". Below the red box, an arrow points down to "1 2 3". Below the green box, an arrow points down to "4 5 6".

Alternative approach

- Extra space is required to merge
- Merging happens because elements in left half must move right and vice versa
- Can we divide so that everything to the left is smaller than everything to the right?
- No need to merge!

So, the main reason we need extra space and merge sort, because of the merge operation as we saw in order to combine A and B into a list C in linear time you need extra space. Now, suppose and this the reason this happen is because we could have something like the left side could be say 2, 4, 6 and the right side could be say 1, 3, 5 and now we need to take things some both side. So, sometimes I need think something from here, then I need take something from here, then I need take something from here and so on.

So, though each of the two half is sorted, the elements in left which a must go to right and the elements from right must go to the left. So, now, if we could come up with the way to avoid doing this, so such that everything in the red part is smaller than everything within the green part, then once I have sorted the red part and I have sorted then green part then automatically everything in the red part stays to the left of everything in the green part.

So, if I had add instant of this, if I had add on the left I started with say 2, 1, 3 and the hand side started with 6, 4, 5 and then, I locally is sorted these things. So, I got 1, 2, 3 and then here I got 4, 5, 6 then after this if I guarantied the this the split was search in everything on the left is smaller than everything on a right, after doing is divide and then solving the sub problem there is no combination is no need to merge. So, next sorting algorithm will look at the strategy to implement this idea.