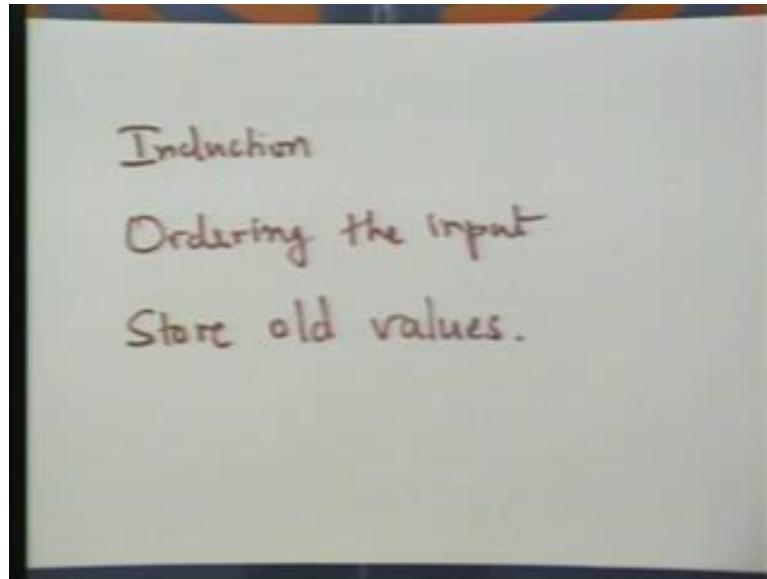


Design and Analysis of Algorithms
Prof. Sunder Viswanathan
Department of Computer Science Engineering
Indian Institute of Technology, Bombay

Lecture - 6
Divide and Conquer – I

(Refer Slide Time: 01:11)



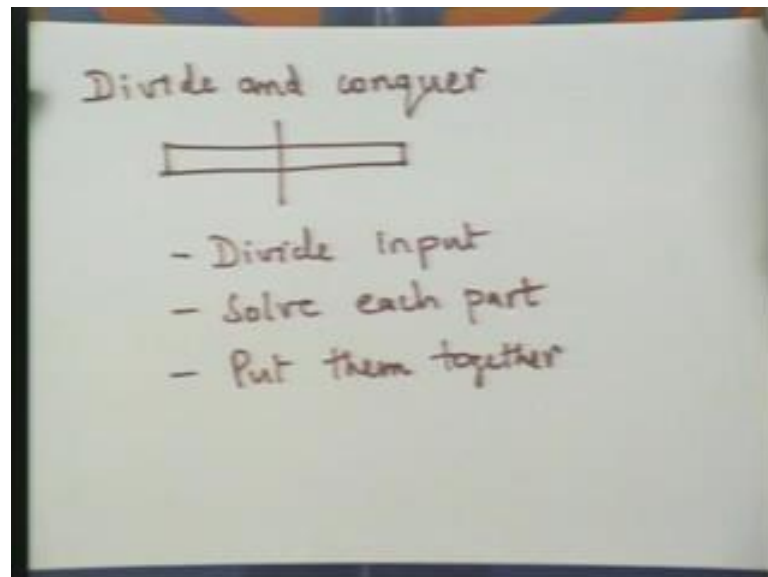
The three of them are first 1 is induction. Then, ordering the input and the third is storing old values. So, let me sort of let us talk about all these three once more. Induction essentially says, supposing you can solve the problem for smaller inputs. How do you solve it for larger inputs? So, this is the first term design principle. So, supposing you know, you can put together, these solutions for these smaller inputs. And somehow generate a solution for the larger input.

Then, your algorithm is clear. You sort of recurse on the smaller inputs solve them. And when you get the solutions back you put them together. And get the solution to the larger input. That is induction, you could use either recursion or it could even be once you know what the smaller inputs it could even be ((Refer Time: 02:42)). The next point is ordering the input, which is you look at the input in some order. For instance, if it is an array you look at it. Let us say, in increasing order of indices.

For other structures it is could be different. If it is a list again, you will have to look at it element by element. For other data input and other data structures, you could this could

vary. The other thing is store values, which you may use later on. If you are not going to use a value, which you have computed later on then you can of course, discard it. Otherwise, this very simple principle will help us design algorithms in the future good. So, we have seen, some algorithms simple algorithms for simple problems using these.

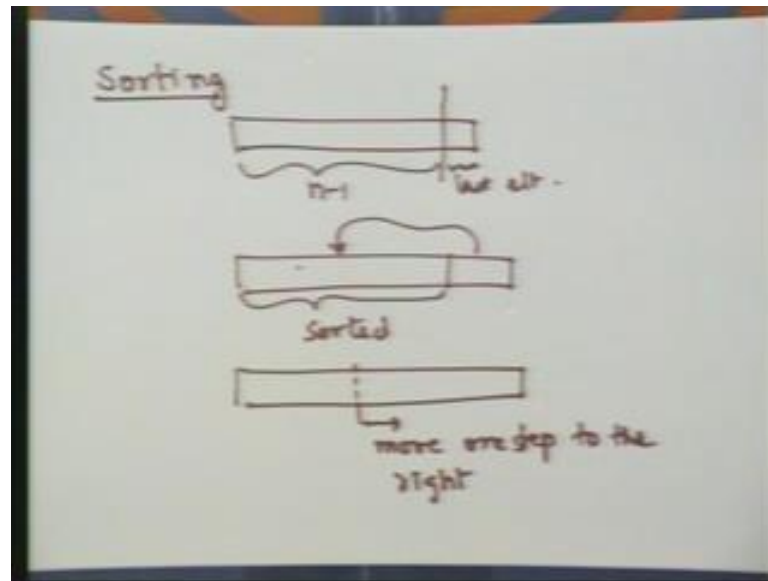
(Refer Slide Time: 03:44)



The paradigm, that we considered last time was divide and conquer. So, the basic idea was this, if I have an input let us say an array. But, it could absolutely anything. Divide it into two parts. Solve for the left, solve for the right say. And then, put the solution back together. So, you divide input. Solve each part and put them together to get a solution for the big problem. And well often it pays to divide it into equal parts. We will in fact, see an example why this is so...

But, even in the previous cases for instance max min or finding the second minimum. The two examples that we saw, you could try breaking at breaking the input up into unequal pieces. And then, try and solve the recurrence. And see, what answer you get. So, you should I sort of encourage you to try this out. So, the next problem, we are going to consider is the very familiar, should be very familiar to all of us it is sorting.

(Refer Slide Time: 05:22)



Only we are going to look at it now from the new sort of design principles, that we have learnt. And we are going to try and apply those to sorting. So, let us take the simplest sort of case. You have an array. An array of n elements and you would like to sort the elements in increasing order. Let us, assume that elements are distinct. This is not going to change any of our design principle or that or the algorithm we come up with, but it will just, it is just to keep your mind less cluttered.

So, you have an array of let us say, n elements and you would like to sort them in increasing order. Let us put the first design principle into practice. Supposing you can solve it for a smaller array, how do you extend it to a bigger array? And the most natural smaller arrays let us say, the first n minus 1 elements. Supposing, I could solve this problem of sorting the array for the first n minus 1 elements. How do I, now extend it to the bigger array.

So, here the problem, here is what the input looks like. This will be first n minus 1 elements and this is the last element. So, you first sort this is by recursion let us, say and now I want to place this in the right position. Now, I guess most of you know binary search by now. So, the position is very easily identified. So, let us say, this is now sorted. So, this portion is sorted and this position now is identified. Let us say, they are you do a binary search. And identify exactly where this goes in.

Well now, what you have to do is to insert this in the right position. You have to move, you have to make essentially make space. So, from this point onwards, you need to move everything one step to the right. Every element here, you have to move one step to the right. Move one step and then insert the element in the right order. So, here is an algorithm. So, let us go over this again. You recursively, sort the first n minus 1 elements. Figure out where the last element sits and then put it in place. This is called insertion sort.

So, in your previous courses, you have dealt with this algorithm. You have learnt insertion sort there people I guess told you what insertion sort is all about. Here, we are trying to understand, how it is that people come up with these algorithms. So that, when you are faced with a new problem, you can come up with an algorithm of your own. That is the idea. So, this is how you come up with an insertion sort. It is just putting you know the principle, we had into practice.

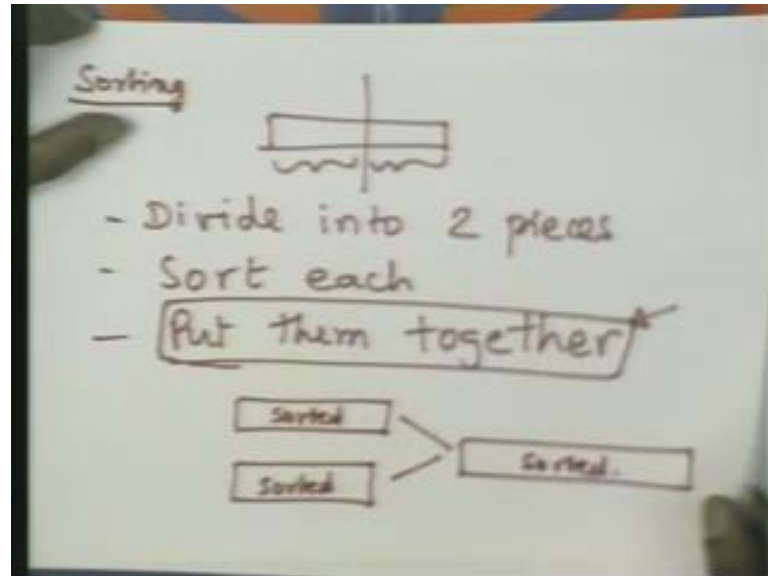
((Refer Time: 09:03)) how much time does this take. Well, if we just look at number of comparisons. If just want to find out the number of comparisons. The last element, you use the number of comparisons is $\log n$, because you are just doing a binary search. So, and every other element use less time. It was for the i th element it was $\log i$. The time taken was $\log i$. So, the total time is in fact, roughly order $\log n$, order $n \log n$. I mean it is order $\log n$ per element, there are n elements and it is $n \log n$.

But, the main step here, which takes time is not comparison, but this movement. Let us, look at the last step. The last step says, once I find this place I have to move everything to the right. Now, this place could actually be the beginning of the array, in which case you are going to shift the entire array by 1. So, while you just took $\log n$ time to find the place to move elements, you may take you know n units of time. This is the expensive step in insertion sort.

And if, implemented in this in this manner, each time you take order n steps. For the i th step, you take i units of time to move it to the right. You may take I units in the worst case, in which case the time will be order n square. So, I will leave it for you to see if you can use better data structures. So, that you can sort of avoid this movement. But, implemented this way insertion sort takes order n square time though the number of comparisons here is still $n \log n$.

Let us, put our other design paradigms into play into practice. One was to divide the array equally. Work on each piece and then put them together. Let us look at this.

(Refer Slide Time: 11:15)



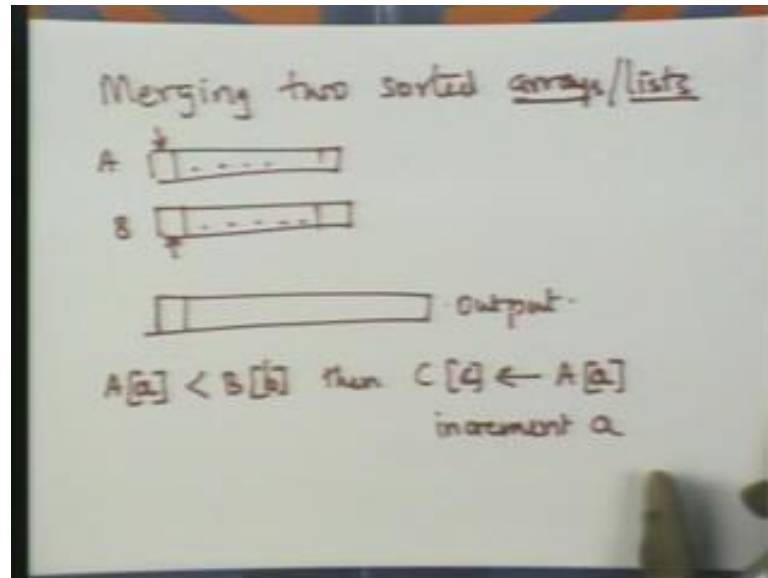
So, here is my so sorting. So, this is the second algorithm we are going to try. So, here is my array. You divide it into two equal halves, two halves. Sort each piece, sort each piece and then you have to put these things together. So, let us, write this down. So, divide into two pieces. Sort each piece and then put them together. So, the algorithm is clear, except for this step. What does it mean to put them together? So, far it is okay. Dividing into two pieces no sweat, sort each recursion.

You recurse on each piece you sort each. Well, here is the crucial step that we need to implement, which is putting two sorted arrays together. Let me, also say that this you could work with other data structures. Instead of arrays you could have list. And you can apply the same design principles to list too. In fact, we will do this with arrays. But, you it is really does not matter. So, when I mention array, you could you could put a list there equally well. Especially, for this method, so we divide into two pieces.

We sort each of them by recursion. And then we put the solutions together. What does this step entail? Well, this let us focus on this step. Because, once we implement this the algorithm is ready. And you can program it at your leisure. So, this step entails the following. I have two sorted arrays, we sort it. I need to combine this to get one big sorted array. This is what I want. I have these two pieces, which are sorted. This piece is

sorted, so is that piece. We need to put these two sorted pieces together to get a big sorted piece. So, this is the problem that we would like to solve. So, this is called merging two sorted arrays or lists. It does not matter.

(Refer Slide Time: 13:54)

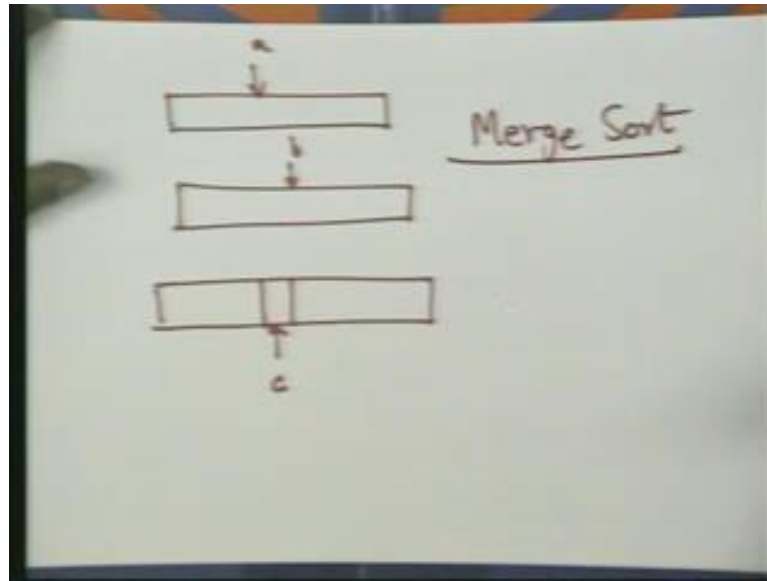


So, we are given elements. These are sorted let us say, sorted in ascending order and I would like to build a bigger structure, which is which contains elements in both of them and which is sorted. So, I guess the procedure should be fairly natural. And most of you should have gotten this by now. Which is well, what is the smallest, so here is my new list or array. This is my output. What is which element will occur will occupy the first position. Well it has to be the smaller of these two elements.

So, what I do, I first compare the first elements of these two lists or arrays or sub arrays. And put the smaller of them here. So, I can have two pointers or two temporary variables, which point to some places in the array. I compare these two put the smaller one here and move that corresponding pointer. Let us, call this A and B, supposing A 1 is smaller than B 1. Then C of 1 I should assign the smaller of 2, which is A 1. And I should move this pointer.

So, let us call these two pointers something. So, small a and small b. So, c small c is small a and then I need to increment. So, increment so a, which is pointing here will point to the next element and so on. So, I just keep doing this at any stage. So, what is the generic step at some stage I have these arrays A and B.

(Refer Slide Time: 16:24)



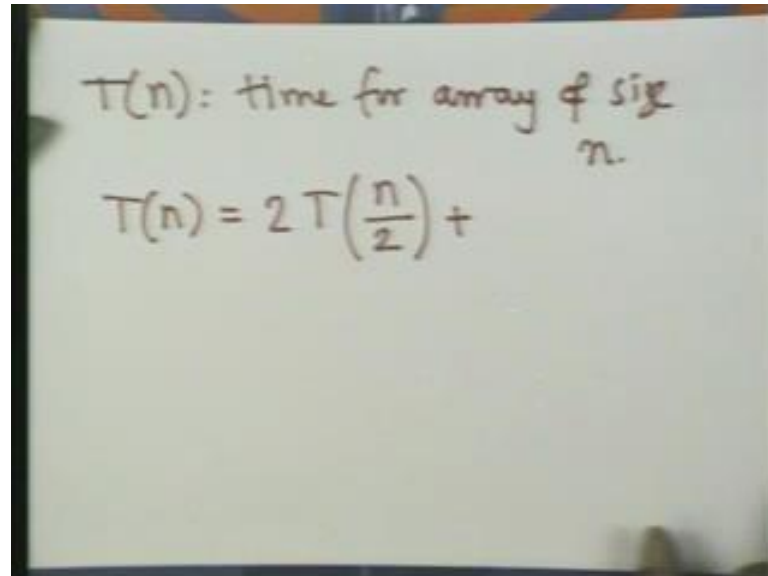
I have pointers small a and small b. And then I have pointer small c. I compare these two elements, put the smaller of them here and increment the corresponding pointer. This is the algorithm. So, I just essentially, somehow scan these two and keep filling in these elements in this big array. So, this is merging. I hope you can write code, if need be. If the idea is very simple and if they are less of course, it is it is trivial, you just you have these two pointers you sort of compare these two values.

Put the new value in a new list and increment this pointer. That is the way it is done. And arrays if you want to you may actually, have sub arrays of the original array, where you are doing this work. You could have a temporary array, where you create this merge list and then write it back into the original array. So, this is the algorithm. So, the merge step is also now done. This is the algorithm. How much time does this take? This algorithm is called merge sort by the way. So, it is called merge sort. And how much time does this take.

Let us go back to this ((Refer Time: 17:52)) scribble piece of paper. So, here is merge sort. It says, divide into two pieces sort each of them and put them together. This is essentially, essentially merging two sorted sequences into one big sequence. That is the third step. The divide step well does not take any time. Sorting each is a recursive step. We sort on roughly half the array. And we need to figure out how much time does it take

to put these things together. Good. So, what is the time taken? Supposing T_n is the time for an array of size n .

(Refer Slide Time: 18:43)



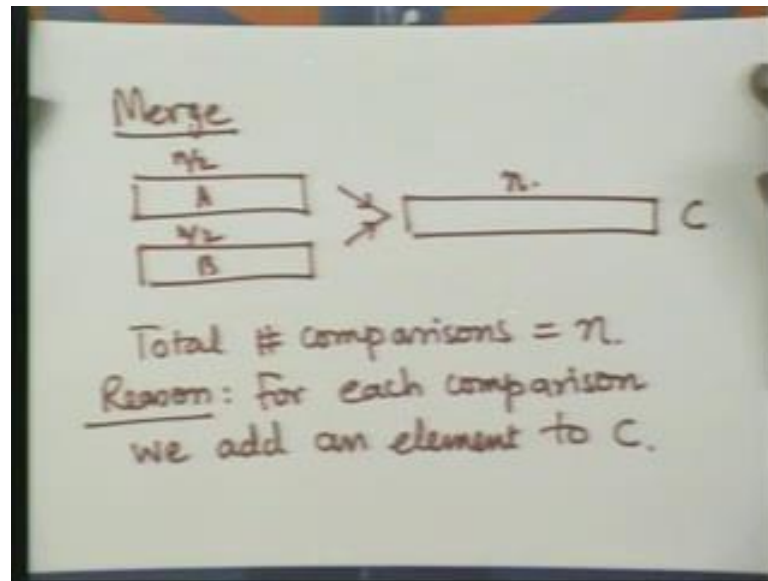
Handwritten notes on a whiteboard:

$$T(n): \text{time for array of size } n.$$
$$T(n) = 2T\left(\frac{n}{2}\right) +$$

By time again, we will focus on the number of comparisons, because even here. Every other operation is big o of number of comparisons. So, the number of, if I can bound the number of comparisons, the total time is a constant times the number of comparisons. So, please check this. For instance moving the pointer etcetera, etcetera, etcetera. And if, you have list copying one list to the other etcetera. They are all bounded by the number of comparisons.

So, I can just focus on this one quantity, which is number of comparisons. So, let T_n be the number of comparisons that we make, we would now like to bound the total number of comparisons needed for an array of size n . Well the divide step does not take any time. So, but there are two sub problems of roughly equal size, so that that gives you $2 T_n$ by 2. Again, we will not worry about floors and ceilings for ease of calculation. And then, there is a merge step. How much time does merge step take on an array of size n . So, this is something we need to do. So, what is the merge step on an array of size n .

(Refer Slide Time: 20:20)



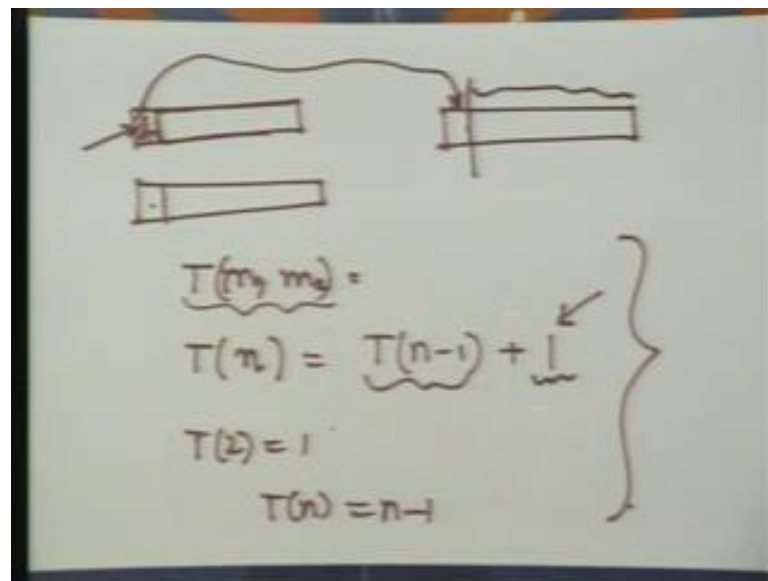
The merge, I have two arrays of size n by 2 . Two sub arrays of size n by 2 or two lists of size n by 2 . And I merge this to get something of size n . How much time does this take? Well, there are many ways of figuring this out. So, let me tell you one way of doing it. This is a smart way of doing it. And then, I will tell you a way, which is just using the, we will again design this algorithm using design principles, that we have already followed and then analyze it that way also. So, we will do it two ways.

So, what is each step? Each step you compare two elements and an element gets filled in the array C. The crucial sort of observation here is that. Each time you make an, each time you make a comparison. One element in C in this new array, the final sorted array gets filled. Once again, one for each comparison an extra element a new element gets filled in the total array. How many elements get filled? The total number of elements that gets filled into a new array is n .

So, the total number of comparisons you make is utmost n . Let me say, state this analysis again, because this is this is quite important. Each time you make a comparison an element of C gets filled. So, if I make k comparisons, then I fill k elements in C. The total number of elements in C is n . So, the number of comparisons I make must also be bounded by n . So, the total number of comparisons is n . The reason is for each comparison, we add an element to call this array C.

From these two smaller arrays A and B, I pick an element and add it to C. The total number of elements in C is n. So, the total number of comparisons I make is also n. Let us, do this merge differently. So, that total time now is n., we now this. And we can now, write the recurrence and we will solve it. That is fine. But, let us again do this merge a bit differently. So, we are now going to apply our design principles, which is induction like a merge to smaller arrays. How do I merge larger arrays?

(Refer Slide Time: 23:40)



So, let us look at this again. So, here are my two arrays ((Refer Time: 23:46)). Now, I know that the first element in C is the smaller of these two elements. The first element here is the smaller of these two elements. So, let us say this is smaller. Now, I look at the input without this. The input now consists of one less element from here and some elements from here. Now, the sizes need not be same. So, I can supposing I solve my problem for this smaller input.

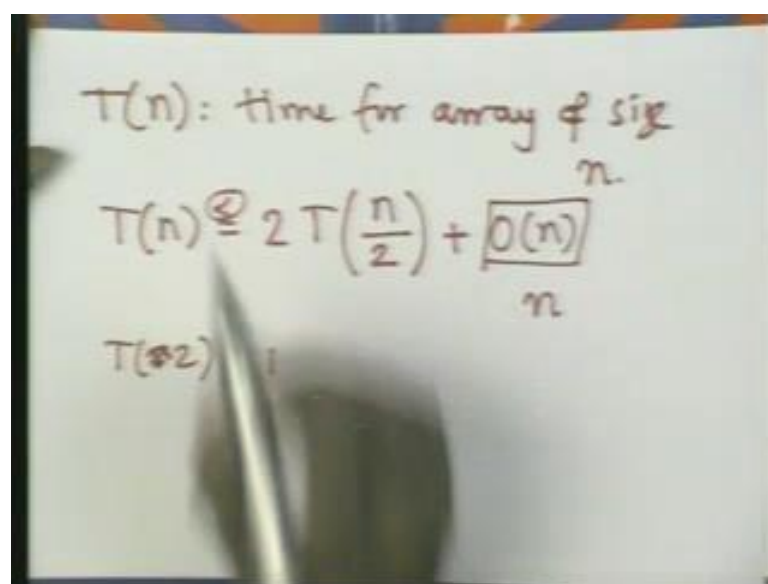
Can I solve the problem for the larger input? Well, the answer is should be you know vociferous yes. It can be solved. This is the smaller of this shaded is the smaller of these two. And now, I have two arrays here. So, the total size is smaller. One element I have removed. So, these two I merge to get this portion. And the smaller element I put here, this goes in here. And recursively, I can sort of merge these two to get this portion. So, this is the other way of doing this.

Here, you would naturally use recursion. And if not iteration at least initially, may be later on you will see that this can be implemented iteratively. And you could put them together iteratively. So, now how much time does this take? If there are two elements. So, let us write the recurrence here. So, T , so this could be n and m , because they could have different sizes. So, if it is, let me not use n here. Let us say m_1 and m_2 . T of m_1 m_2 is what, well one of them decreases by 1. We do not know which one.

So may be, I can just write a recurrence on the sum of these two sizes. So, I can actually, write it on n . This n stands for the sum of these two sizes, which is the total number of elements in the input. This is nothing but n minus 1 plus 1. Where does this 1 come from? This is two, when I have two lists. For the first comparison I make. I make I compare these two elements remove the smaller 1 and apply recursion. This 1 comes for the first comparison I make.

And you can check that T_n the solution to this. So, T of 2 is 1. The solution to this is T_n equals n minus 1. So, to merge two arrays the sum of two sizes is n . The number of comparisons is n minus 1. So, these are two ways to do this merge. When you write the code actually, the comparisons they, the both the algorithms do will be, if you unroll the recursion you will actually get the iterative process back. So now, we can go back to merge sort and look for the what is the time that we need.

(Refer Slide Time: 27:19)



Handwritten text on a whiteboard:

$$T(n): \text{time for array of size } n$$

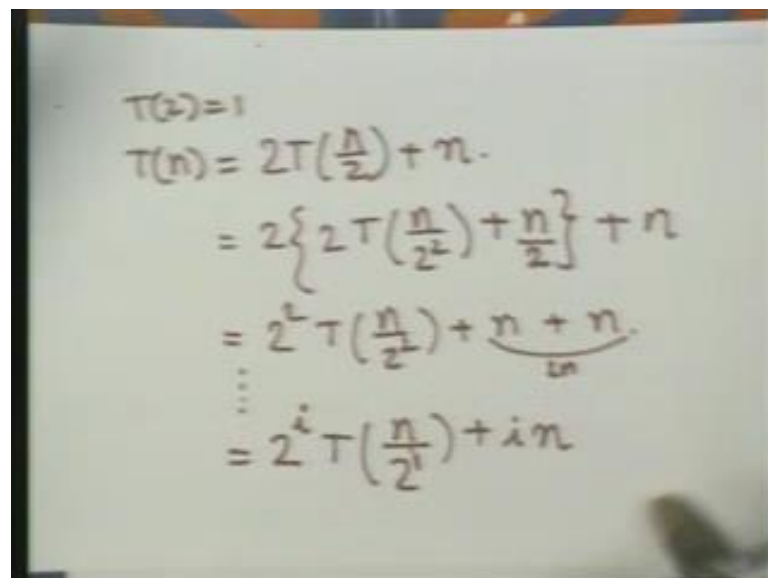
$$T(n) = 2T\left(\frac{n}{2}\right) + \boxed{O(n)}$$

$$T(2) = 1$$

So, it is 2 times $T_{n/2}$ plus order n . So, in fact, well I just put n here for simplicity. You can put a constant times n and calculate this, it will make no difference. These constant will just come out of the calculation. So now, we need to solve this recurrence. I guess most of you would have hopefully do know how to solve this. Let us, anyway do this. So, T_2 is 1. Well, I should strictly write T_n is less than or equal to $2 T_{n/2}$ plus order n .

If I am writing comparisons for the simple reason that, if I have two arrays whose let us say, they are not of equal size. Then I could use less than n . In this case, it is n and it is equal. But, often you should sort of check whether, this should be equal to or less than or equal to. In this case, it is in fact, equal.

(Refer Slide Time: 28:39)



The image shows a whiteboard with handwritten mathematical steps for solving a recurrence relation. The steps are as follows:

$$\begin{aligned}
 T(2) &= 1 \\
 T(n) &= 2T\left(\frac{n}{2}\right) + n. \\
 &= 2\left\{2T\left(\frac{n}{2^2}\right) + \frac{n}{2}\right\} + n \\
 &= 2^2 T\left(\frac{n}{2^2}\right) + \underbrace{n + n}_n \\
 &\vdots \\
 &= 2^i T\left(\frac{n}{2^i}\right) + i n
 \end{aligned}$$

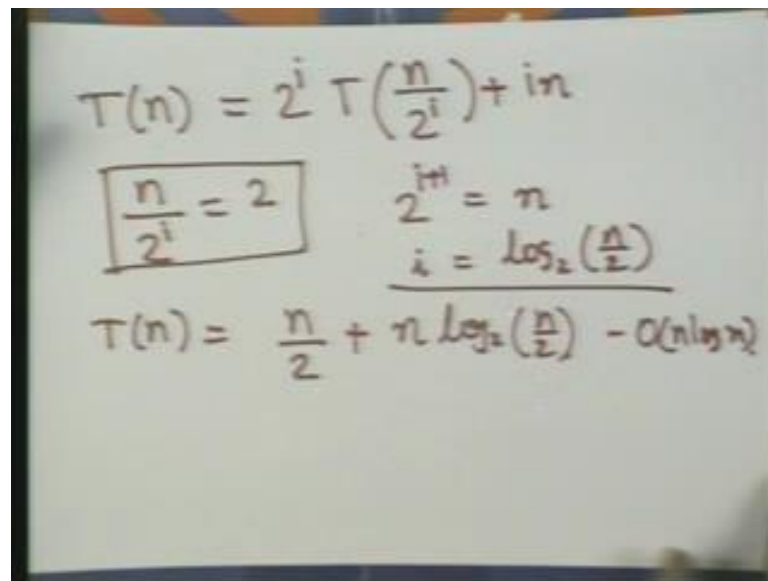
And T_n is twice $T_{n/2}$ plus n . So, this then is so let us open this out, twice $T_{n/2}$ plus n plus n . So, this $T_{n/2}$, I have replaced I have again used this recurrence. I am sorry. So, this should be $n/2$. I have just used recurrence. I use the same recurrence on $T_{n/2}$. So, that will be twice $T_{n/4}$ plus $n/2$, but just using this with $n/2$ in place of n . So, that is what I get. So, what is this, this is $2^2 T_{n/4}$ plus n plus n .

So, if I do this i times, well you could do it once more to see what the pattern looks like. If I do it i times, I get $2^i T_{n/2^i}$ plus there will be a number of n . So, how many times will I have n 's here, it will be i times n , this you can

check. You should do it once more, you will get 3 times n. So, here I have 2 times n. If I do it once more I get 3 times n. The next step you get 4 times n. So, you check, you sort of guess that this is what is going to after I steps.

And you can actually check this by induction on i. Once, you have guessed this you can check that T_n is in fact, equal to this by induction on i. That I will let you know. Now what, well we continue till n by 2 to the i. So, let me let us start here.

(Refer Slide Time: 30:58)

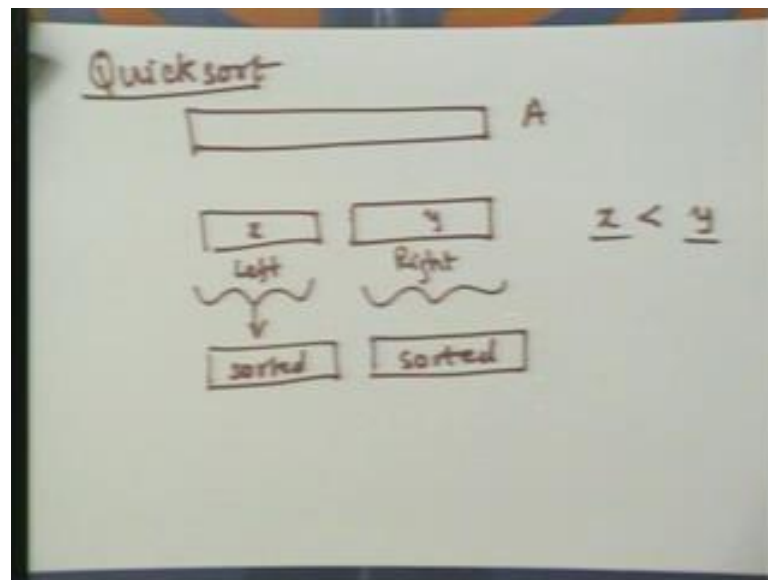


The image shows a handwritten derivation on a whiteboard. At the top, the recurrence relation is written as $T(n) = 2^i T\left(\frac{n}{2^i}\right) + in$. Below this, a box contains the equation $\frac{n}{2^i} = 2$. To the right of the box, the equations $2^{i+1} = n$ and $i = \log_2\left(\frac{n}{2}\right)$ are written. At the bottom, the final result is written as $T(n) = \frac{n}{2} + n \log_2\left(\frac{n}{2}\right) - O(n \log n)$.

Now, we know T of n is 2 to the i times T n by 2 to the i plus i times n . Now, I know t of 2 is 1 . So, I will let n by 2 to the i to be 2 . So, then what do I get. If I choose an I , so that this is true. Then, I get T_n is well 2 to the i is n by 2 . This is t by t of 2 , which is 1 plus i times n . what is i . Well, I know 2 to the i plus 1 is n . So, i is \log base 2 of n by 2 . This is what i is, so plus n times \log base 2 n by 2 . This is order $n \log n$. So, the time taken by merge sort is $n \log n$.

There is one more way to do this divide and conquer business on arrays to sort. And it in fact, gives rise to another well known algorithm, which you may have studied. Let us see, what this idea is. So, here is the array.

(Refer Slide Time: 32:37)



I still want to do divide and conquer. So, I still want to divide the array into two parts. Somehow work with each part. And then put the solutions back together. But, the way I do, it will be slightly different in this case. So, what I do is this. So, what I do is in the divide part, I ((Refer Time: 33:02)) start rearranging the elements of the array. So, I now have two parts to the array. But, these are not divided as in the previous case.

Remember in the previous case, we just picked up the array and just divided arbitrarily into two parts. This time we are going to be more careful. What we would like is every element landing up in the left is smaller than every element landing up in the right. So, there is a x here and y here. I know that x is less than y . Again, every element in the array must occur in one of these. So, somehow, I have divided this array into two parts left and right. So, that elements in the left are smaller than the elements in the right.

This somehow we have done this. Now, what we do is just recurse on these two parts. So, you sort the left separately. You sort the right separately. Let us see, what happens once you do this. Once you do this, I claim that the entire array is now sorted. Why is that so? Well this portion is sorted right. This left portion is sorted. This right portion is sorted. I also know that every element on the left is smaller than every element on the right. So, the largest element here is smaller than the smallest element here.

So, when I look at the entire array, you can check that the entire array is now sorted. So, this is also an example of divide and conquer. The divided step is where we did work. In

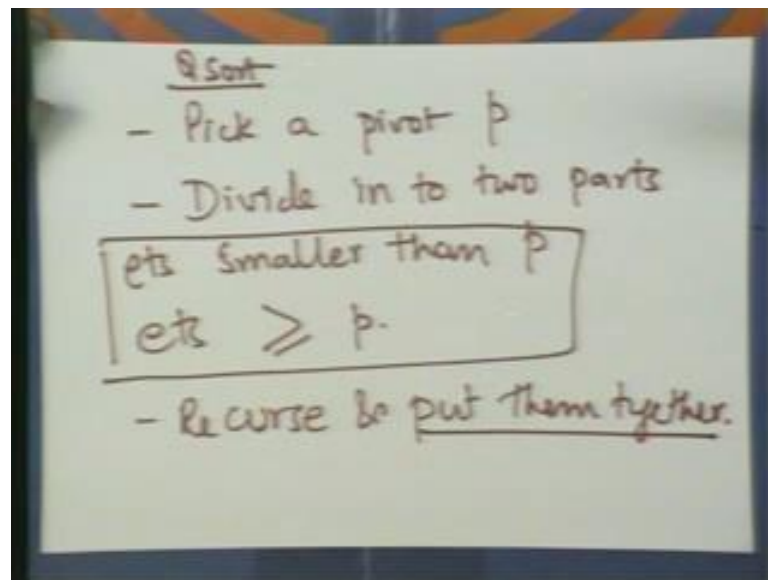
the divide step, which I have still to specify you somehow divided this array into two parts left and right. So, that elements in the left are smaller than the elements in the right. This was the divide step. Now, it is recursion ((Refer Time: 35:06)) sweat.

You just recurse on these two parts and recursion does it for you and you sort these two. Putting things together is trivial. You just have to do no work. Both of them are sorted, when you put them together in fact, the entire array is sorted. The only thing we need to figure out, it how to divide the input. So, that everything on the left is smaller than everything on the right. Well, some of you may have noticed that, this is an algorithm that you have seen before and it is called Quick sort.

Again, let me sort of emphasize that. Earlier on you were told what these algorithms are. You were given code for ((Refer Time: 35:54)) while this is quick sort, that this sorts an array. Our emphasis here is to see, how do people come up with these algorithms. How does somebody come up with an algorithm called quick sort. Well, this is how they come up with. So, you start with divide and conquer at the back of your mind. And you figure out, how do you put this paradigm into place.

So, now how do people, how do we split an array into left and right. Well, you pick a pivot, some element in the array, which you call the pivot. Left consists of every element, which is smaller than the pivot. And right consists of every element, which is greater than or equal to the pivot. So, the smallest element on the right is the pivot. And you know that the left of the array every element is smaller than the pivot. So, that does it. Let me just write this down. So, that we have quick sort in front of us.

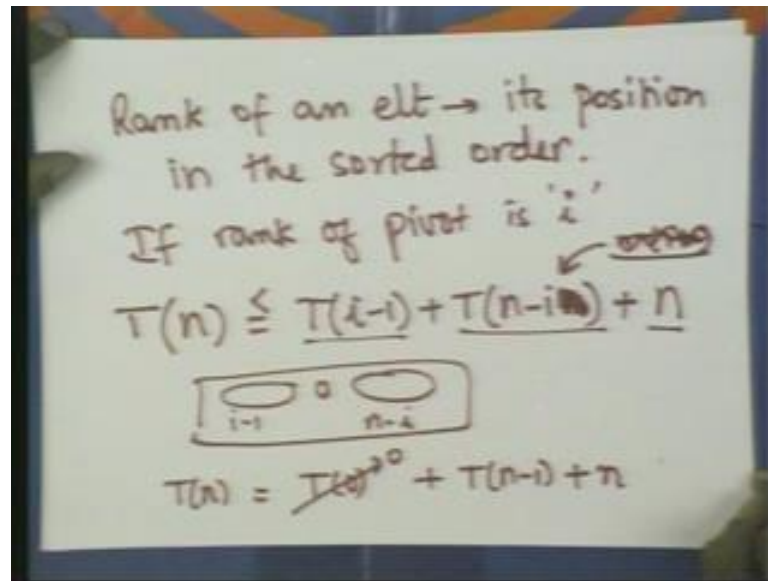
(Refer Slide Time: 37:03)



So, pick a pivot. This is quick sort. Pick a pivot divide into two parts smaller than the pivot elements. Let us, call this p pivot p. And elements greater than or equal to p. These are elements less than p and elements greater than or equal to p. These are the two parts. Recurse and put them together. That is the last step. Well, unlike the difference between merge sort and quick sort is essentially this. In quick sort this is where you spend most of your time. Putting them together is easy.

In merge sort dividing into two parts is easy. Putting them together is where you take time. So, this is the this is this is quick sort. How much time does quick sort take? Well, it depends on which element you pick as a pivot. So, let us see, what really happens. So, supposing you pick the i th element. What do i mean by i th element, means in the sorted order it is the i th element. So, let me make a definition here, which we will use later on.

(Refer Slide Time: 38:53)



The rank of an element is its position in the sorted order. Remember recall that, we are assuming that each element in the array is distinct. So, each element has a distinct rank. If there are i elements smaller than x , say x is an element i elements are smaller than x the rank is i plus 1. The minimum element has a rank 1. And the maximum element has a rank n , which is the size of the array. So, the rank of an element is its position in the array. So, if everything depends on the rank of the pivot.

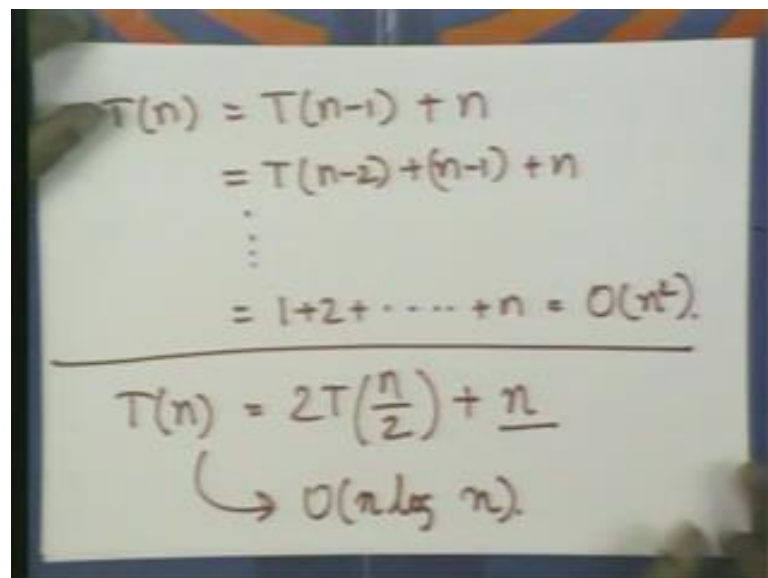
So, if the rank of the pivot is i . Then, the time taken there will be i minus 1 on 1 side and n minus i plus 1 on the other side right. So, is T of i minus 1 plus T n minus i plus 1 plus the time taken to partition the array. What was this time? How did we partition the array? Well, we compared each element to the pivot. Put the ones on the left on the, put the lesser ones on the left and the larger ones on the right. So, we compared every element to the pivot. So, number of comparisons we made is n minus 1.

For ease of calculation, we will just say this is less than or equal to n . It will not bother us much. What is the constant between ((Refer Time: 40:54)). So, if the rank is i then it is utmost T of i minus 1 plus T n minus i plus 1 plus n . This is the time taken to partition the array, because every element was compared to the pivot ones exactly once. So, what is the solution to this recurrence. Clearly sort of it depends on i . It depends on what i is and various values of i it varies, for instance if I picked the minimum element.

If i picked the minimum element then, then what happens. Then i was 1, oh sorry this should be minus 1 ((Refer Time: 41:47)) n minus i minus 1 or it should be n minus i plus 1. That is what fits in here, N minus i minus 1. So, if i were 1 then there is nothing on the left. And there are n minus 1 elements on your right hand side. There is something wrong here. This should just be n minus i. There is no 1 here. This is exactly n minus i. I have the pivot. I have i minus 1 here. And i have n minus i elements on this side.

The pivot sits at the center. So, the right hand side had size n minus i. So, if I has chosen the minimum element, then I get T of n to be T of 0, which is 0 plus T of n minus 1 plus n. And if I kept picking the minimum element as the pivot what is this time. So, let us write this again.

(Refer Slide Time: 43:14)



$$\begin{aligned}
 T(n) &= T(n-1) + n \\
 &= T(n-2) + (n-1) + n \\
 &\vdots \\
 &= 1 + 2 + \dots + n = O(n^2)
 \end{aligned}$$

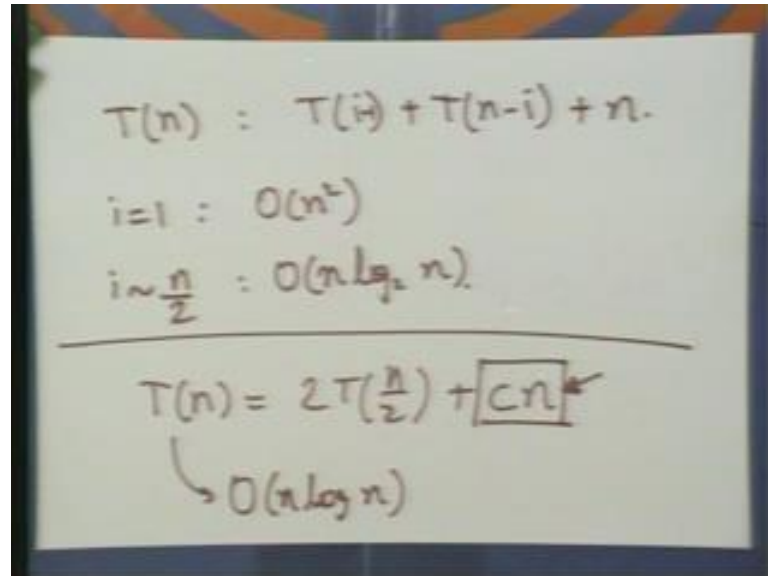
$$\begin{aligned}
 T(n) &= 2T\left(\frac{n}{2}\right) + n \\
 &\rightarrow O(n \log n)
 \end{aligned}$$

So, T n looks to be T of n minus 1 plus n. Let us expand this further. It is n minus 2 plus n minus 1 plus n so on and so on. Well, it comes down to 1 plus 2 plus so on up to n. This is n times n plus 1 by 2. This is order n square. So, if the pivot turns out to be the minimum element in each case, then the total time taken is order n square for quick sort. And you can check that if I start increasing now. Let us say, the pivot was a second element. There is a time decreases. It will decrease till you reach the middle element.

Let us see what happens, if I pick the roughly the middle element. Then, T of n roughly, I will have two problems of equal size. That is 2 T n by 2 plus n. This does not change. Because, every element I have to compare with the pivot anyway. This recurrence I

know what then solution is. This looks like order $n \log n$. So, let us sort of write all this on one slide and look at it.

(Refer Slide Time: 44:47)



Handwritten notes on a whiteboard:

$$T(n) : T(i) + T(n-i) + n.$$
$$i=1 : O(n^2)$$
$$i \sim \frac{n}{2} : O(n \log n).$$

$$T(n) = 2T\left(\frac{n}{2}\right) + \boxed{cn}$$

↳ $O(n \log n)$

I started out with saying let us say, T_n with T_i plus T_{n-i} plus n . Now, if i equals 1, there is $i-1$ here. If i is 1, then i get order n square. This i is roughly n by 2. Then i get order $n \log n$. Well, one can check that, if I plot T_n versus i then it initially falls. If we start with n square it initially starts falling, till I roughly n by 2. And then, again it starts rising. So, if I pick the maximum it is very similar to picking the minimum. The array sizes split much the same way.

And it again goes back to n square. So, the worst case time is n square. And if you manage to pick the pivots correctly, I mean if you pick the middle element at each stage. Then the time taken by quick sort is $n \log n$. So, we are only worried about the worst case time. So, if somebody asks you, what is the time taken by quick sort. Well, it is n square. This implementation of quick sort, takes order n square time, because that is the worst case time.

If somehow, we could pick an element of rank n by 2 efficiently. If you can pick an element of rank n by 2 efficiently, what do you mean, what do I mean by efficiently here. It means the recursion should not change much. So, if I know I get $n \log n$ with the following recurrence T_n is $2T_{n/2}$ plus n . In fact, I could add a constant here. And it

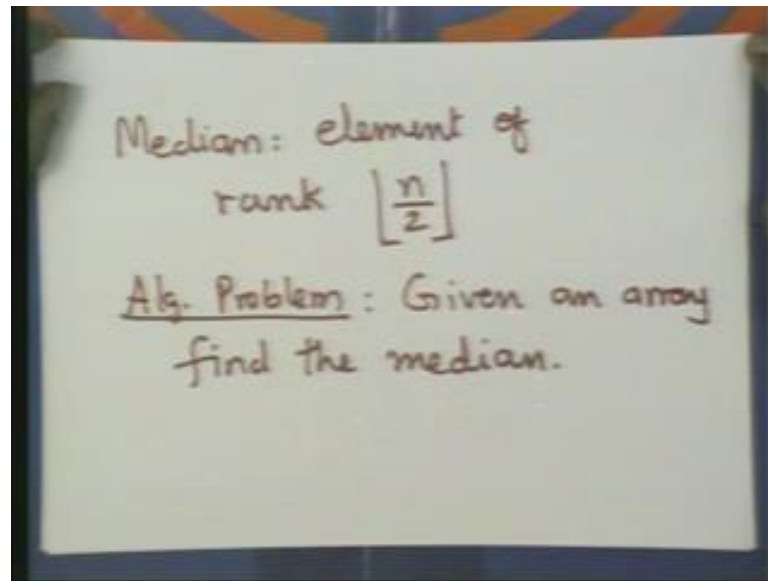
would still be $n \log n$. This will still give me order $n \log n$. So, you can check this. This constant will just come out of the calculation.

If I have $c n$ then I get $n \log n$. Now, if I can somehow find the middle element of an array, which means an element of rank roughly $n/2$. So, if I can find an element of rank $n/2$ in constant times n time. Then, I can implement quick sort to take time order $n \log n$. The way I do it is I first pick I spend $c n$ time pick this middle element. The element of rank roughly $n/2$. Now, I do the usual. Use this pivot divide the array into two parts.

Now, the array is divided into two roughly equal parts. Once, I divide them into equal parts. Now, it is the usual recursion and I am done if I take $n \log n$. So, I can implement quick sort. There is an implementation of quick sort. There will be an implementation of quick sort, which takes $n \log n$ time, provided I can find this element of rank $n/2$ in linear time. Which is what we do next? Before we do that let me so point this fact out, that in the recurrence, it was desirable that the two parts were of roughly equal size.

If they were not of equal size, then the time was proportionally it was higher. So, often in these divide and conquer kind of situations. We try and see, if we can somehow split the input into exactly two parts in two halves. So, once we have halves, some because of the way this recurrence these recurrences work the time taken is usually smaller. So, our job next is to figure out, how to pick this element of rank $n/2$ in linear time. So, let me state this problem. So, this element of rank $n/2$ is called the median.

(Refer Slide Time: 49:28)



So, the median is an element of rank n by 2 . Let us say, floor of n by 2 . You could take the ceiling of n by 2 . It does not matter. So, n is odd. You get the floor of n by 2 and our job. So, what we want to do is this. The algorithmic problem given an array find the median. How do you find this element of rank roughly n by 2 ? Well, the easy way to do it. Is sort the array and pick the middle element. That takes time $n \log n$. The time for that is $n \log n$.

So, if you for instance use merge sort. That is not good enough. Can we do it faster? The answer is yes. You can in fact, do it find median in linear time. And this will be the first non trivial algorithm that you will see. It is a really smart algorithm and as we present, I hope you appreciate the beauty of the solution.