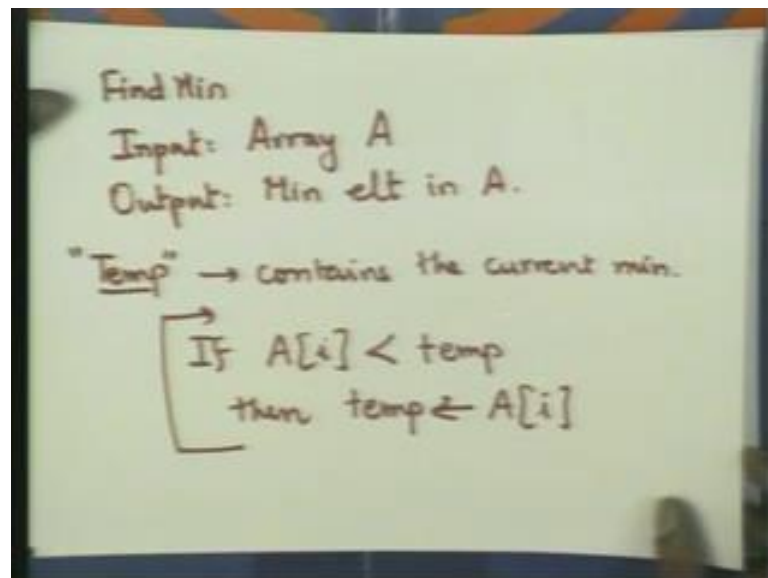


Design and Analysis of Algorithms
Prof. Sunder Vishwanathan
Department of Computer Science Engineering
Indian Institute of Technology, Bombay

Lecture - 5
Algorithm Design Techniques: Basics

We begin with discussion of some Basic Design Techniques for Algorithms. We will start with fairly simple problems that many of which you may have seen. But, we will hopefully see solutions to non trivial problems, which you may not have seen. The first example, I would like to start with is finding the minimum element in an array.

(Refer Slide Time: 01:21)



So, the input problem is Find Min. The input is an array A. And the output is a minimum element in A. A element with whose value is the minimum. This is a problem, which most of you would have see earlier, in your. Perhaps the first programming course may be later even in your course on data structures. Let us anyway go over the solution. So, the standard solution is, you have you take a temporary variable.

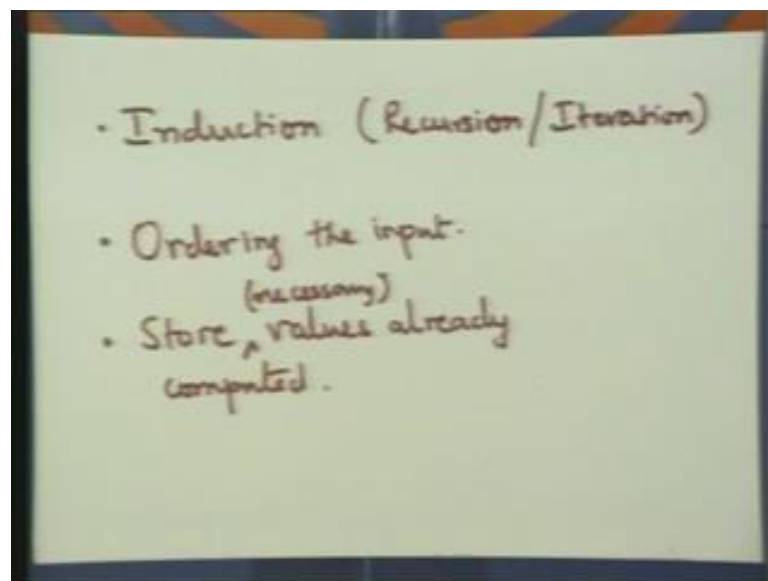
There is a temporary variable, which contains the current minimum. So, this contains the current minimum and it is updated at each step. So, you start with the first element in the array. And then, you scan the array element by element. The First element then the second element third element and so on up to the last element in the array. Each time, you compare the array element with temp and update temp if necessary.

This simple technique actually is used is very powerful. And used often, which is if your input is in the form of an array or a list, then if you can solve the problem for the first n minus 1 elements of the array. Can you solve it for the n th element? This is the step, which is put in a loop. So, let me just write this down. So, the important step is this. If A_i is less than temp, then temp is set to A_i .

This is the crucial step. And this is usually put in a loop i varies from 1 to n , where let us say n is a size of the array. And at the end of this temp will have, the minimum element in the array. So, I will be when I discuss algorithms, often I may not write the full code. In fact, I will not write the full code. The idea is to give you the main ideas, behind the design of this algorithm. I may not take care of some of the stray cases. I may not initialize variables properly etcetera, etcetera.

So, all these programming details I will not get into. The main idea is to, get the main the techniques, and the ideas behind the algorithm which solves the problem. And you should be able to sort of take these ideas together. And write the program, which actually works. Yes. So, if you look at this, this is the level at which we will describe algorithms may be even less. I could just say, scan the array element by element. And update the current minimum as required.

(Refer Slide Time: 05:15)



There are, so there is two things couple of things that I would like to point out. So, this thing this design technique is often called well induction. Both recursion and iteration

actually go with this. I will explain this a bit more. The second technique, the second sort of thing we will keep in mind is ordering. And the third thing is to store value, already computed. I must add store necessary this value. So, let us go each of these one by one.

So, induction what you mean by induction is this. You want to solve a problem. Your input has some size. The array has say size n . Now, the one way to solve it is, supposing you can solve this problem for smaller values of input. If this problem can be solved for smaller values of input, now can I extend these solutions to a solution for the bigger input. If you can solve the problem for the smaller values of input, can you extend the solution to a bigger value of the input.

For instance, if I can solve a problem for all arrays of size n minus 1, can I solve this problem for an array of size n ? Now, this step which takes you from n minus 1 to n is often put is the crucial step. And once you come up with this step, we just put this step in a loop or you use recursion. You first recurse on an array of size n minus 1, and then extend it to an array of size n . Or you scan the array one by one, element by element. And you update the solution as you go along.

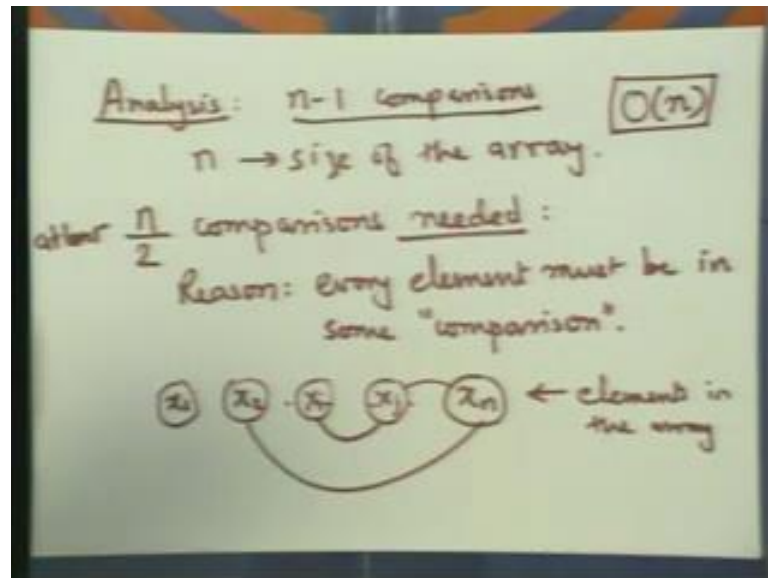
So, this is what I mean by induction. And it is at the base of every algorithm design technique. You can call it the mother of all algorithm design techniques. You will learn some more fancier things later, but this is the very crux of design of every algorithms. Second point that, I would like to make is ordering the input. Looking at the input in the right order often helps. In this case, it is simple. You know, you look at the array elements in the increasing order of the array index.

In fact, you could look at the array in any order. It really does not matter. But, there are cases, we will see cases where ordering plays a very crucial role in solving the problem. The third thing, which is also fairly simple here is to store some of the values that you already computed. That is the third point, we want to make. Now, even in this case it is simple. You just store the previous minimum, the variable temp the temporary variable which we had.

So, these are values that you would like to use in future. Almost every algorithm design technique that, we will study are a combination of these three. Some of them will use just induction and storing whole values. Some of them will use all three. Some of them will

just order the input and use induction. So, these three are things that you must keep at the back of your mind, when you design any algorithm. Let us analyze this algorithm.

(Refer Slide Time: 09:23)



So, here is the here is an analysis. Every algorithm that we design, we will analyze. Analysis is as important a part of this subject as design. And we would like algorithms to be as fast as possible, in the worst case. So, all our analysis pertain to worst case. And we would like to design algorithms, which are as fast as possible. So, in this case we look at every array element one. And we make one comparison for array element. This leads to, we make n minus one comparison.

We do not really compare anything with the first element. But, with each of these subsequent we make one comparison. n is the size of the array. So, we make n minus 1 comparison. Along with comparison, we need to we also store this new value in temp. But, the number of times we do this exactly equals the number of mean. It is let us say one more than the number of comparisons, we make. So, if you are not really worried about constants, then you can just focus on the number of comparisons.

As long as I bound the number of comparisons, the other small operations that I do, like incrementing the index variable of the array or storing the value in temp. They all are of the same order as the number of comparisons. So, the time taken here we would say is order n . The time taken is order n , but our focus is just on the number of comparisons. I think the question that one can ask and one should ask is this the best?

Can I find the minimum element in an array in using less than n minus one comparison? Well, we just think about it. Or even, if you do not think about it. I guess, most of you will jump up and say, of course you need n minus 1 comparison. That is the sort of first reaction that, most people will have. Without making n minus 1 comparison, how can you find the minimum? Why is this so?

Can you logically argue that, you actually need n minus 1 and you know you cannot do with less than n minus 1. This argument is easy, but not absolutely trivial. Let us see. So, why should we make n minus 1. Can we do it with less? Well, every element you have to look at least. Every element in the array must be compared to something or the other. If it is not compared then, you may actually make it the minimum. It could be the minimum.

If you had output something else as a minimum then, this could have been made the minimum. Or you can easily make some other element of the array the minimum. So, without comparing if you do not compare an element then, you cannot tell the minimum element in an array. Now, this gives you this does not give you n minus 1. It gives you n by 2, assuming n is even. So, at least n by 2 comparisons are needed.

The reason every element must be compared, must be in some comparison. What I mean is, every element must be compared with some other element. This can be done with n by 2, by the way. So, the first element is compared with the second element. The third element is compared with the fourth element and so on. The fifth element is compared with the sixth element and so on. So, with $n/2$ two comparisons, if n is even or $n/2$ plus 1, if n is odd.

Well, ceiling of $n/2$ comparisons are all that is necessary. To sort of satisfy this condition that, every element must be in some comparison. Now, clearly $n/2$ is not the right answer. n minus 1 is the right answer. And why is it that 1 is n minus 1? So, here is an argument. See initially, when you have not made any comparison there are n candidates for the minimum. Each element in the array can actually be a minimum.

You do not know, which of these n elements are minimum. Now, when you make a comparison, you can get rid of only one of these candidates. When you compare, let us say at some stage you have candidates x_1, x_2 up to x_k , some k elements are candidates for the minimum. Each of them based on the comparisons that you have made, previously each of them is equally likely, I mean each of them can be a minimum.

Now, if you compare two of these candidates you can get rid of one. Whichever one is smaller, that still remains a candidate. Whichever one was larger, that no longer remains the candidate. But, you can only get rid of one candidate. So, with each comparison I can only get rid of one candidate. I have n candidates to start with. So, I need n minus 1 comparison. This is actually a complete proof, though a bit hang wavy. You can make it more rigorous also.

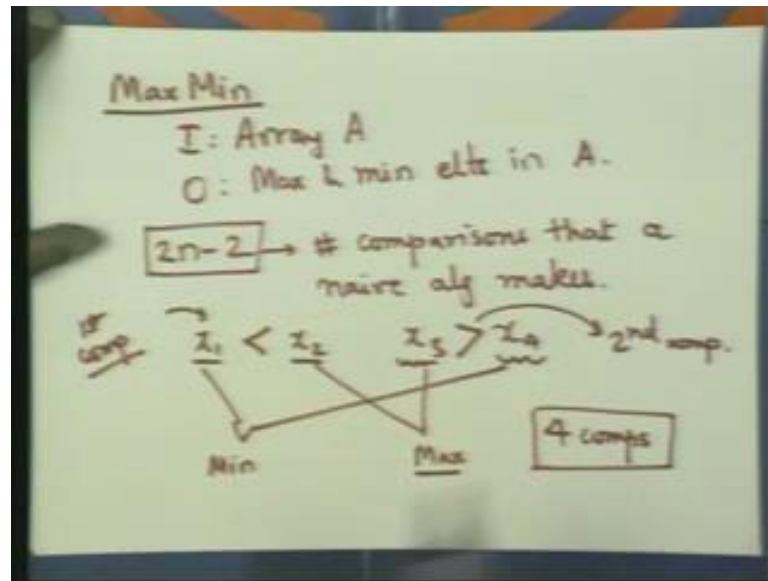
Here is one more way of looking at it. Supposing, you draw the following graph. So, initially we have I have all these nodes. Let us say the elements x_1, x_2 up to x_n . These are elements and the array and also vertices in our graph. Let me just put a circle around these two, indicate that these are also vertices. Now, when you compare x_i and x_j . Here is x_i and here is x_j , I draw an edge between these two.

Now, when you compare let us say x_1 and x_2 and x_n , I draw this edge. Now, x_j and x_n are compared I draw this edge and so on. As you make comparisons, I keep drawing these edges. So, once your program ends terminates, you have done all these comparisons. Now, I look at this graph. And I look at each connected component in this graph. If there are more than one connected components in this graph, then I will not be able to tell the minimum.

Now, in each connected component I know which is the minimum? There will be one which is a minimum, but I can surely give values to these. So, that I can pick the global minimum from, any one of these connected components. Here, this is arguments that after the comparisons are over, once you finished all comparisons I better have one connected component. This means from a discrete structure's class, you know that you need n minus 1 edges.

So, this is just the same argument. Both of them have the same idea behind both. For instance, here you start with n connected components. And each time you add an edge, you can decrease the number of connected components by 1 at most 1. So, it is the same argument. So, you need n minus 1. And this sort of simple scan of the array does it with n minus 1 comparison. Let us look at a slight variation of this problem. Now, I want to find not just the minimum in the array, but also the maximum.

(Refer Slide Time: 18:29)



So, this problem is called Max Min. So, the input is in array A. And the output is the maximum and the minimum elements in A. So, I want both the maximum and the minimum element. Now, if I just wanted to find the maximum, the procedure is clearly the same as the one for the minimum element. What if I want to find both the maximum and the minimum? Well I could first find the minimum and then, I can find the maximum. How many comparisons does this take?

Well n minus 1 for the maximum. n minus 1 for the minimum. And that makes it $2n$ minus 2. So, the number of comparisons that a naïve algorithm. So, this is the number of comparisons that, the naïve algorithm makes. We can ask the same question. Is this the best? You can try the previous argument that, we had of connected components. And so, you can show that you need n minus 1.

That is fine, but when you see more than, that is very difficult to prove. It is not absolutely impossible, but it is difficult. And if you try to certainly increase it to $2n$ minus 2, it is impossible. You will not be able to prove it. Let us look at some small values. Supposing I have four elements, i have let us say x_1, x_2, x_3 and x_4 . The naïve algorithm took x_1 compared with all of them. Found the minimum and then we were done.

Then, we took again took x_1 we compared it with all of them. And while maintaining the maximum, the temporary maximum. Now, many of these comparisons are repeated.

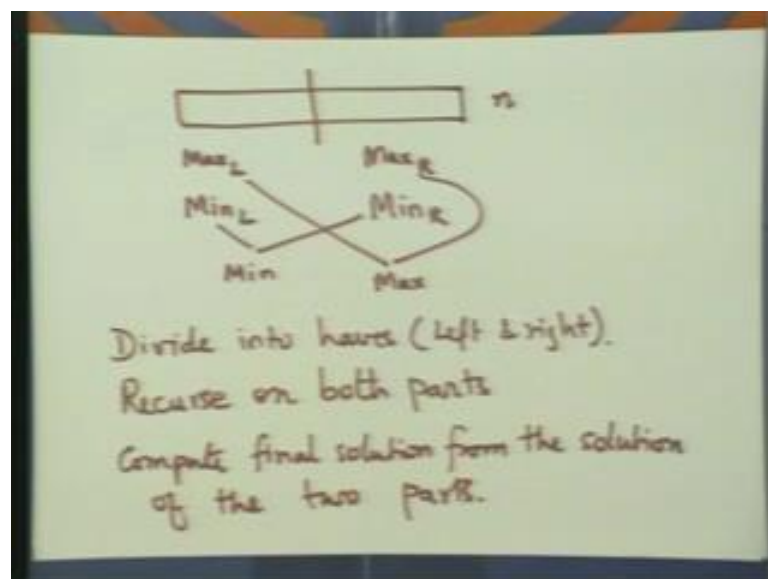
So, you see that many of some of these comparisons that you make, when you on roll the whole thing out, are repeated. So, our aim is to sort of get rid of these unnecessary comparisons. Now, in four elements here is what you can do. I first compare x_1 and x_2 .

So, supposing x_1 is less than x_2 . So, this is the first comparison. x_1 is less than x_2 . Now, I compare x_3 and x_4 . Supposing x_3 is greater than x_4 . This is my second comparison. Now, where do I find the minimum? I mean, how do I find the minimum? The minimum clearly is either x_1 or x_4 . It is the smaller of x_1 and x_4 . So, I compare these two and find the minimum. Similarly, I compare these two and find the maximum.

So, how many comparisons I have made? I have made 1 2 3 and 4, so four comparisons. What does our old algorithm say? It says $2n - 2$. n is 4. So, this is 6 when n is 4. So, we seem to have done certainly, better than $2n - 2$. When there are four elements, we have certainly done better than $2n - 2$. And well the trick was, once we found the minimum and maximum between x_1 x_2 and x_3 x_4 then, for the minimums I only need to look at x_1 and x_4 .

The smaller on the left hand side and the smaller on the right hand side. I do not have to bother about the bigger one and similarly for the maximum. So, this trick can be applied recursively. Well, it is certainly worth trying and let us do it. So, what we do is this.

(Refer Slide Time: 22:57)



Here is the let us say I have you have an array of size n . So, let us divide this into two parts. Recursively find the maximum in this part. Let us say Max L. This is the left part and that is the right part and Min L. Recursively, find the maximum minimum on the right hand side. So, that is Max R and Min R. Now, how do I find the maximum and minimum? Well, I need to compare these two to find the minimum. I need to compare those two to find the maximum.

So, here is an algorithm that seems natural. I divide this into two parts. Divide this into two parts. Let us say two equal parts. Two equal halves. Halves are always equal. I find the maximum and the minimum on the left hand side. The left half, I find the maximum and minimum in the right hand side. Now, I compare the two minimums to output the minimum of the array. I compare the two maximums, to find the maximum of the array.

How many comparisons does this algorithm take? Let me write down the algorithm. But, in future with this explanation you should be able to write the algorithm. So, divide into halves. Let us say left and right. Then, recurse on both parts on the left and on the right. And the answers are Max L and Min L and Max R and Min R. Then, put these things together, to get the minimum and maximum. And then, compute final solution from the solution of the two parts.

Well, I have written the essence of the algorithm without really writing details. I hope, you can fill in the details. Define procedures and write down recursive calls. And you know, do these two comparisons and output the minimum maximum. Do how many comparisons does this take? That is the question, we need to answer.

(Refer Slide Time: 25:53)

Handwritten notes on a piece of paper:

$T(n)$: time taken by Maxmin on arrays of size n .

$$T(n) = 2T\left(\frac{n}{2}\right) + 2$$
$$T(2) = 1$$
$$T(n) = 2\left\{2T\left(\frac{n}{4}\right) + 2\right\} + 2$$
$$= 2^2 T\left(\frac{n}{2^2}\right) + 2^2 + 2$$

...

So, let us say T_n is the time taken by max min on arrays of size n . It is the time taken by max min on arrays of size n . Then, T_n is there are two problems of half the size. You solve two problems of half the size. There is $2n$ by 2 plus; two more comparisons, one between the 2 maxs to get the new max, one between the 2 minus to get the new minimum. These are this is for the recursive call. So, you call the left hand side, that is n by 2 right hand side n by 2 and then 2 .

Well, if n is odd I would have a ceiling and floor somewhere. But, let us not worry about it for the time being. Let us assume that, n is even. You can assume n is the power of 2 . We also know that, T_2 is 1 . For two elements, I can find it in one comparison. So, what is the solution to this recurrence? Let us see. So, the easiest way to solve all this is to check, how this recurrence behaves. So, T_n is nothing but 2 . And now, I open this out.

This is $2T_n$ by 4 plus 2 plus 2 , which is $2^2 T_n$ by 2^2 plus 2^2 plus 2 . You can right this down, once more. And essentially we want to see how, what pattern this follows? Well, it is not too difficult to guess what the pattern is? The pattern is this.

(Refer Slide Time: 27:55)

$$T(n) = 2^i T\left(\frac{n}{2^i}\right) + 2^i + 2^{i-1} + \dots + 2$$

Set $\frac{n}{2^i} = 2$ $2^{i+1} = n$

$$T(n) = 2^i + 2^i + 2^{i-1} + \dots + 2$$

$$= \underbrace{2^{i+1}}_n + \underbrace{2^{i-1} + \dots + 2}_{\frac{n}{2} - 2} = \frac{3n}{2} - 2$$

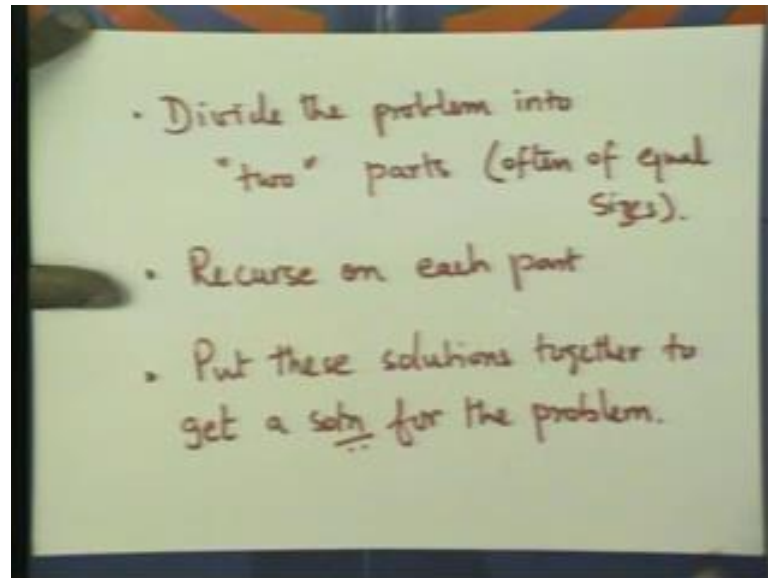
So, T_n is 2 to the i $T_{n/2}$ plus 2 to the i plus 2 to the $i-1$ and so on. All the way up to 2 . So, now we set $n/2^i$ to be 2 , because we know that t of 2 is 1 . Then, we have t of n is 2 to the i . This t , this becomes 2 . So, t of 2 is 1 plus 2 to the i plus 2 to the $i-1$ and so on up to 2 . So, this is nothing but 2 to the i plus 1 plus well 2 to the $i-1$ and so on up to 2 . You can check that this is nothing but we also know that 2 to the i plus 1 is n . And I hope you can solve this.

I will leave it for you, to solve this. This is nothing but n . And this sum, you will get as n by 2 minus 2 . If you sum this up, using the usual geometric series and use this fact, that n is 2 to the i plus 1 . You will get that, this sum is nothing but n by 2 minus 2 . Well, put this together. You get t of n is $3n$ by 2 minus 2 . You can check that this, when n is 2 this is 1 , which is what we want. And this also satisfies the recurrence that we had.

The recurrence was let me refresh your memory. The recurrence was T_n is twice $T_{n/2}$ plus 2 . So, if I put T_n equals $3n$ by 2 minus 2 , this satisfies the recurrence. You can prove that T_n is this ((Refer Time: 30:12)). Well, so the number of comparisons that we seem to make using this method is $3n$ by 2 minus 2 , which is certainly better than $2n$ minus 2 . We seem to have done something fairly mechanically and we seem to have improved the number of comparisons made, quite drastically.

So, this technique is called divide and conquer, is used by the British in the last century, I mean last century and even before that. We will put it to good use, in designing algorithms. So, let me write down the main steps of this technique.

(Refer Slide Time: 31:02)

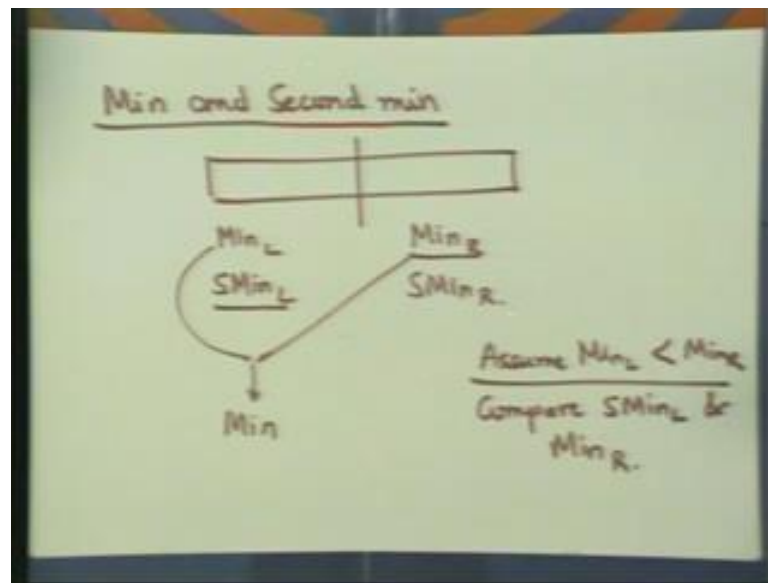


The first step is to divide the problem into, I will say two parts. Often we will want these parts to have equal sizes, often of equal sizes. The next step is recurse on each part. So, you recurse on each part and solve each of them. Both the parts, if there are two. And the final step is put these solutions together. Put these solutions together, to get a solution for the original problem. Often, you can just do this blindly.

In fact, for the max min we could have done it blindly. Take this array of size n , divide that divide this array into two arrays to size n by 2. Find the maximum and minimum on the left array, the maximum and minimum on the right array. And once you have these two solutions together, now you find the maximum of the whole array comparing the two maximums. Find the minimum of the array by comparing two minimums.

Again the essence is this induction. In the sense that, if you could solve problems of smaller size which is what you are doing in this recursion. You are somehow putting these together to get a solution, for the big problem. How you will find these small problems varies from problem to problem. Let us look at another problem, where we will apply this method blindly and we will see what we get?

(Refer Slide Time: 33:14)



This problem is to find both the minimum and second minimum. The minimum is the smallest element in the array. The second minimum is the next one, the second smallest element in the array. So, your input is an array. And you are going to find both the minimum and the second minimum. The usual way you would do it is, you first scan the array and find the minimum. Now, you again scan the array and find the second minimum.

The other way to do it is, to have two temporary variables Temp 1 and temp 2. In temporary 1, I store the current minimum. In temporary 2, I store the current second minimum. Now, when I get to an array element i let us say the i th element. I compare this first with the minimum then, with the second minimum that I have. And based on the result of these two comparisons, I update minimum and second minimum.

Now, this will take we have seen how many comparisons this will take. It is $2n - 2$ as it was in the max min case. And let us just apply our divide and conquer paradigm, blindly to this problem and see what you get. So, how would we do this? So, here is the array. The array is of size n . I divide this equally into two paths. I find a minimum and second minimum here. So, let us say Min left and the second Min left, Min right and S Min right.

So, I have found these four values. And now, I want to find the minimum and second minimum for the entire array. The minimum is not a problem. So, I just compare Min L

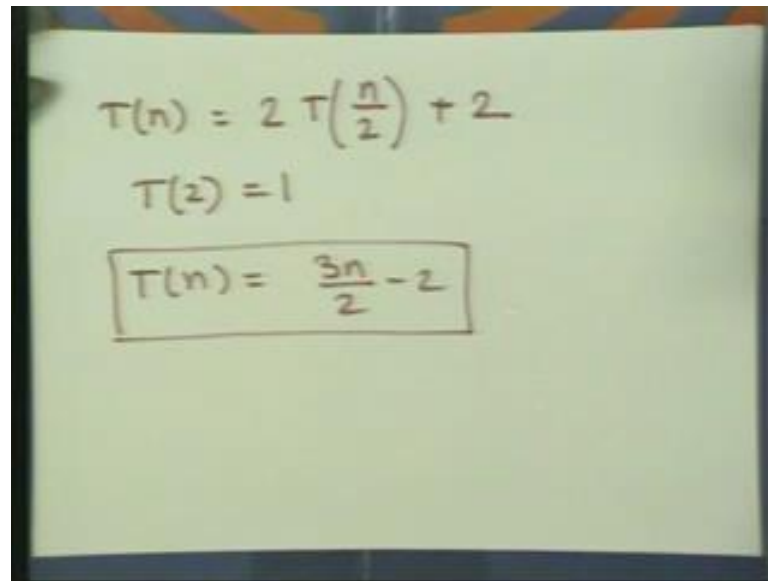
and these two values and I can output the minimum. The smaller of these two is the minimum. Now, what do I do about the second minimum. Now, supposing Min L was smaller than Min R, so without loss of generality this Min L. So, assume Min L was less than Min R, which means at this point Min L has been output. Now, what are the candidates for the second minimum? Clearly, Min R is still a candidate for the second minimum.

Second minimum of the left hand side $S \text{ Min L}$ is also a candidate for the second minimum. But, one of these elements we have sort of thrown out, which is $S \text{ Min R}$. This does not figure in the picture at all. So, we need exactly one more comparison to get the second minimum, which is supposing this is true then compare the second minimum on the left and the minimum from the right. So, you need to compare just these two elements.

And you can see that, the minimum of these two will give me the second minimum. The minimum of the entire array I get by comparing these two minimums, the minimum the left hand side minimum the right hand side minimum. And the second minimum I can get by comparing the minimum element, which lost the first comparison which was larger and the second minimum of the element that one.

So, that will give me the second minimum. So, how many comparisons have we does this take. Well, if you write down the recurrence this seems to be very similar to the previous one. So, what is?

(Refer Slide Time: 37:23)



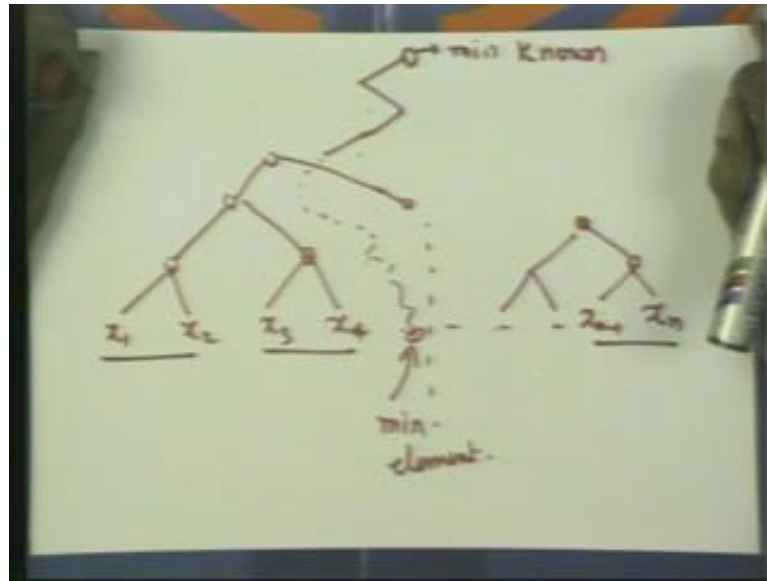
The image shows a piece of paper with handwritten mathematical equations. The first equation is $T(n) = 2T\left(\frac{n}{2}\right) + 2$. Below it is $T(2) = 1$. The final result, $T(n) = \frac{3n}{2} - 2$, is enclosed in a rectangular box.

So, if T_n is the time taken by the algorithm, we have two sub problems each of size n by 2 . That takes time t of n by 2 . And then, we have two more comparisons, one with the two minimums and one to find the second minimum. We also know that, t of 2 is 1 . If I have two elements, one comparison suffices to find both the minimum and second minimum. And these set of equations are exactly the same as the set of equations, we had before.

So, the solution is T_n is $3n$ by 2 minus 2 . So, the number of comparisons we make is 3 half n minus 2 . And it is not $2n$ minus 2 . It is much less. One can ask, is this the best? Can we do better than 3 half n minus 2 ? This question can be asked both for max min and also for min and second min. Well, it turns out that these two problems behave differently. For max min 3 half n minus 2 , is the best we can do.

So, 3 half n comparisons is the best we need 3 half n comparisons. While in this case, when minimum and second minimum you can do with actually less. The divide and conquer sort of paradigm gave us 3 half n , but that is not the best. So, why? So, let me give you a reason why you can do better here. To better this, you need to understand a bit more as to how this algorithm works? Let us unfold the recurrent, the recursion out and see what this looks like.

(Refer Slide Time: 39:18)



Initially, I have array elements x_1, x_2, x_3, x_4 and so on, all the way up to x_n . What we do is, we just divide it into two parts. We divide it down the middle. And then, you recurse on these two. On the left half, we divide it again into two halves, recurse divide recurse divide recurse. Now, we come down all the way down to, when they avail as size 2. This is when the comparisons start happening. The recurrence sort of bottoms down, till you reach arrays of size 2.

Now, I will find the minimum of x_1 and x_2 that goes up. The minimum of x_3 and x_4 is pushed up. Also the second minimum is pushed up. But, let us not worry about that for the minute. The maximum case also it works very similar. You put the max min, max mins are pushed up at each level. We just focus on the minimum element. So, the minimum element is pushed up. From here, the minimum element is pushed up from there.

At the next level, I will compare the minimum of these two. This contains a minimum of x_1, x_2 . This contains a minimum of x_3, x_4 . This contains a minimum of these two, which is actually the minimum of these four and so on. This would be a bigger tree and so on. All the way up to the root and this root, you can the minimum is known. So, for instance here x_{n-1} and x_n are compared. This is the minimum. At the next instance, you compare it with the other two.

This would be $x \cdot n - 2$ and $x \cdot n - 3$ and so on. All the way up to the root, where the minimum is known. Now, this looks like if n was say power of 2. This looks like very familiar complete binary tree. So, there are $\log n$ levels. There are n leaves. There are $\log n$ levels. And in each level, we sort of have some sort of minimum and some portions of the array and these are pushed up. Now, where was the minimum element?

The minimum element sits somewhere in this array. And at each stage, it is pushed up. It sort of wins, its comparison each time at each level of this tree. And it finds its way to the top. Somewhere with perhaps, came from the left. May be it came from the right, came from the left came from the left and so on. So, it does traverse some root all the way from the root node to a leaf. And this is where the minimum element resided.

Now, what can you say of the second minimum element. Now, well the crucial sort of observation that you need to make to speed up the algorithm is that, at some stage the second minimum must have been compared with the minimum element. This is absolutely crucial. If the second minimum element were never compared with the minimum element then, you really do not know which of these two is the minimum, because in each comparison the second minimum one, it was smaller than every other element.

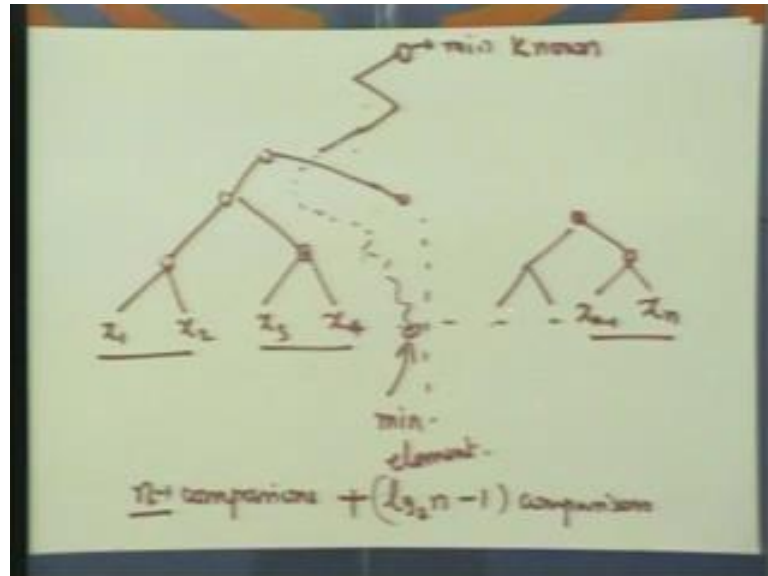
So, was the minimum element which of these two is minimum, to know that you must have compared the second minimum element with the minimum element. Now let us look at this picture. Here is the picture. How many elements did the minimum element, you know win against. How many elements were compared with the minimum element? If you look at this picture and you follow this, at each stage in this path down from the root to the leaf, the minimum element was compared with exactly one element.

The length of this path is $\log n$. So, the minimum element was compared with at most $\log n$ elements in this tree, which means $\log n$ elements in this array were compared with a minimum element. And one of these $\log n$ elements, remember must be the second minimum. So, to find the second minimum all we do is this. Find the minimum using this tree. You can do it recursively, if you want.

Once you find the minimum element, collect all elements that the minimum element one against was compared against. If you have this tree in front of you, you can certainly go

down the tree and figure out, which were these elements. Among these elements, find out which is the minimum? And that will give you a second one. There are $\log n$ elements.

(Refer Slide Time: 44:24)



So, initially you made n comparisons. May be, it is n minus 1, n minus one comparisons to find a minimum. And then, you need about $\log n$ minus 1 comparison more to find the second minimum. This then is actually optimum, though we will not do it in this course. There is an argument, which shows that you need n plus $\log n$. But, it is surprising that you can actually do this in n plus $\log n$. And this problem in this way, it differs from the previous problem.

A straight forward application of divide and conquer does not work. You need to use some more intuition. You need to understand a problem a bit more come up with new ideas. And that is what algorithm design is all about. Often, there are problems which are hard. You really do not know what to do? And when you come up with a smart answer to an algorithm, you feel really you feel nice.