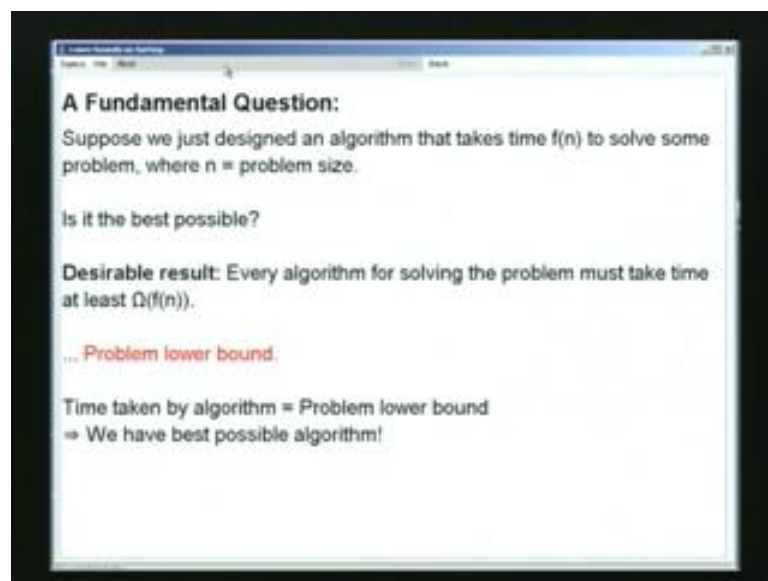


Design and Analysis of Algorithms
Prof. Abhiram Ranade
Department of Computer Science Engineering
Indian Institute of Technology, Bombay

Lecture - 24
Lower Bounds for Sorting

Welcome to another lecture on Design and Analysis of Algorithms. The topic for today is Lower Bounds for Sorting. Let me begin with a very fundamental, very basic question.

(Refer Slide Time: 01:03)



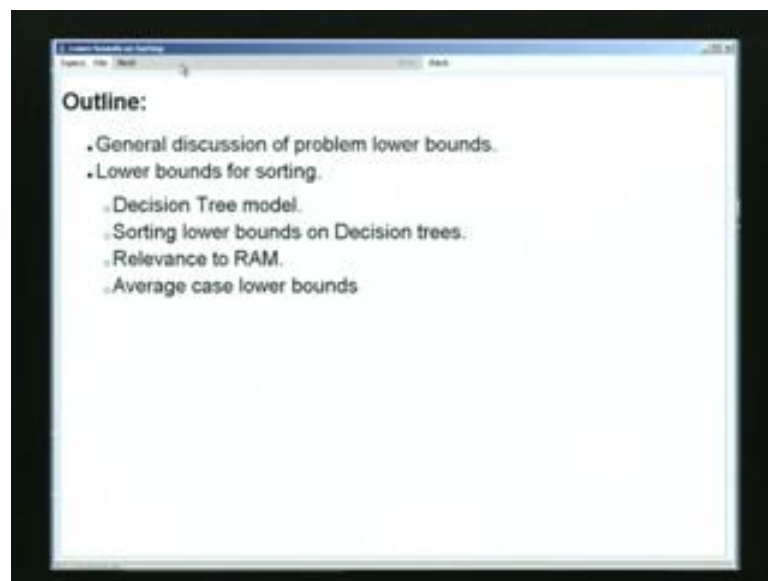
Suppose, we just designed a algorithm, that takes time f of n to solve some problem, where n is the size of the problem. What would we like to know then. Naturally the question, that we would like to ask is, is this the best possible algorithm or can we do better than this. Here is the possible result, we will desire. We would like it, if we can prove that every algorithm for solving the problem, must take time at least omega of f of n .

Remember, f of n is the time taken by the algorithm that we just designed. Supposed, we proved a result like this, what could it mean? Would it be a valuable result, well first of all this result if we can prove, it is called a problem lower bound. It is called the problem lower bound. Because, it says something about the problem, it is not really saying anything about a specific algorithm.

It says every algorithm for solving the problem, must take time at least $\Omega(f(n))$. So, it is a lower bound, then the time taken for any algorithm, which solves the problem. Now, if we could prove something like this. That is, if we can prove that the time taken by the algorithm, is equal to the problem lower bound or is equal to within this Ω , or within some proportionality constant.

What do we know, well if we could prove something like this, we know that we have the best possible algorithm. If this equality is exact and we know, we have the absolute best possible algorithm. If this equality is approximate, well if it is of the form Ω . So, we know that this bound, that we have the best possible algorithm to within a constant factor. So, this is the main motivation for studying, what are called problem lower bounds.

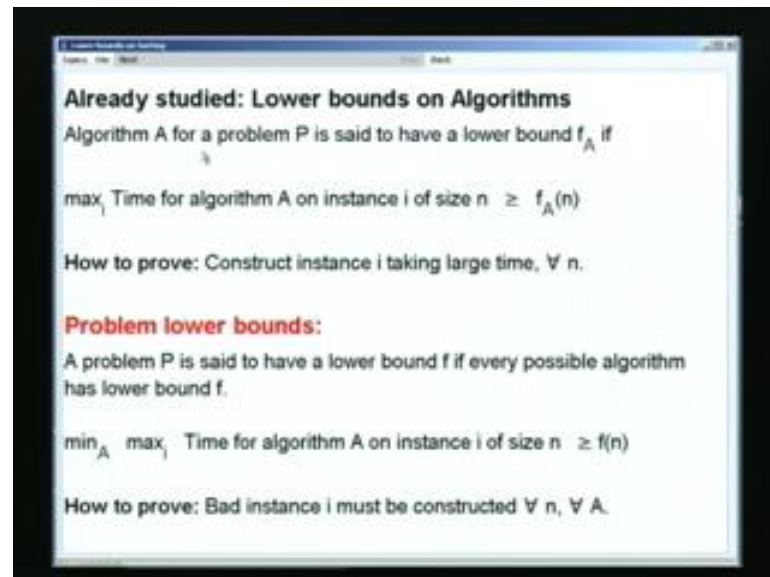
(Refer Slide Time: 03:05)



So, here is what we are going to do today. So, we will be having a general discussion, of problem lower bounds. Then, we will consider the question of lower bounds for sorting. So, problem lower bounds, but the problem being sorting. Regarding sorting I will introduce a model of computation, called the decision tree. And then, we will prove problem lower bounds on decision trees, for the problem we are sorting. Now, you may wonder why do we care about decision trees, is it not the random access machine or the RAM model, which we have been defined and which we have been using so far in the course; is it not, it a good enough model.

Well, it turns out that whatever we do on decision trees, is actually quite relevant to the RAM model. So, we will see this relevance. And then, we will come to average case lower bounds. Again we will prove average case lower bounds, on decision trees. But, again all that will be relevant to the RAM as well.

(Refer Slide Time: 04:12)



So, let us start with something, which we have already studied, at least a little bit. So, we have already defined the notion of putting the lower bound on the time taken by an algorithm. So, here is what we said, we said that if we have an algorithm A for a problem P. Then, we will say it has a lower bound $f_{\text{sub } A}$, where $f_{\text{sub } A}$ is a function. If the time for algorithm A on instances of size n , is greater than or equal to f_A of n .

And of course, we are not worried about every instance, but we are worried about the worst case instance, the worst instance, the instance for which the time is a largest. So, always in this course, we have been stressing the worst case bounds. Or the worst case times, when we measure the performance of algorithms. And even here we are doing that. So, we are asking, what is lower bound on this time, the time of the worst instance i of size n of course.

So, it will be f_A of n , this is what this means. This is what it means to have f_A as a lower bound on a problem P, on a algorithm A for a problem P. Now, of course, it is customary to say that, this inequality holds, only for large enough n . So, there is actually

a clause over here, which says for all n greater than n_0 , where n_0 is some number. But, let us not worry about too many technicalities ((Refer Time: 05:47)).

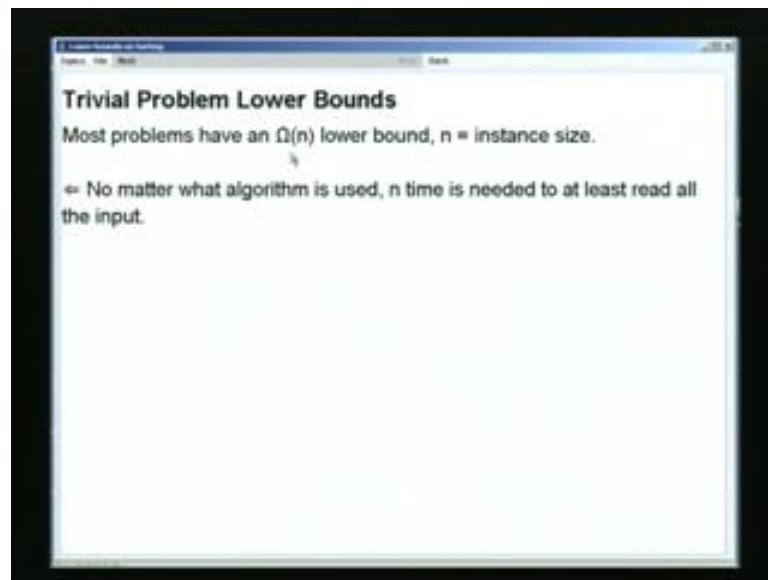
Now, the typical way we prove lower bounds, is by constructing instances i which take a large time. So, if we have an algorithm, we look at where the algorithm is weak so to say. And we construct instances, which show that the algorithm must take a long time on that particular instance. So, that gives us good value of $f \leq A$, $f \leq A$ of n . And we have to do this for every instance, every size instance size. If we can do this, then we can construct this lower bound function.

Next, we will look at problem lower bounds. A problem P is said to have a lower bound f , if every possible algorithm has lower bound f . So, this statement has to apply, but it has to apply for all possible algorithms, for this problem P . Here is the more algebraic statement. So, notice that this part, is just similar to this part; and now we have an additional quantification. We are saying that this must hold, even for the best algorithm. So, the algorithm which takes the minimum worst case time, must have time bigger than f of n .

Of course, with the best algorithm has time bigger than f of n . Then, clearly every algorithm will have time bigger than f of n . So, this is what it means for f to be a problem lower bound, for this problem P . How do we prove such bounds, now we have to construct bad instances. Instances which show, that the time taken is large. But, we have to construct them, not only for all problem sizes, but for all algorithms as well.

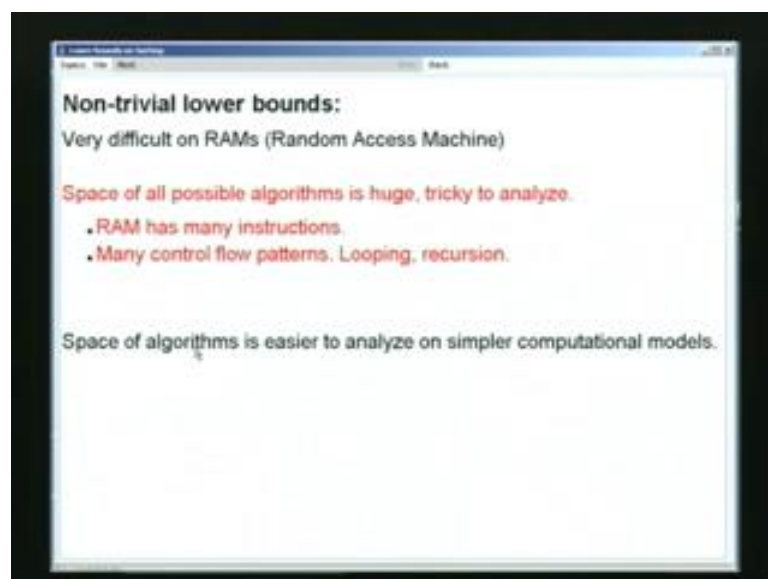
And of course, the instance it need not be the same instance for all algorithms. It could be that for one algorithm, it is one instance which is bad. For another algorithm, it is another instance which is bad, that is ok. But, somehow we must give a construction, which shows that no matter what algorithm you give. Here is an instance on which it will take a long time. We may not do this directly, but at least indirectly something like this has to be proved, in order to prove a lower bound over here.

(Refer Slide Time: 08:21)



Here are some trivial lower bounds, that we can prove. Most problems have omega of n lower bound, where n is the instance size I say this is trivial. Because, no matter what algorithm is used, the algorithm at least has to read all the input. The input has length n and therefore, omega of n time has to be taken. So, in that sense, there is nothing clever about this. And it by and large applies to every problem. There could be some problems where it may not apply, but those are not probably very interesting problems.

(Refer Slide Time: 09:06)

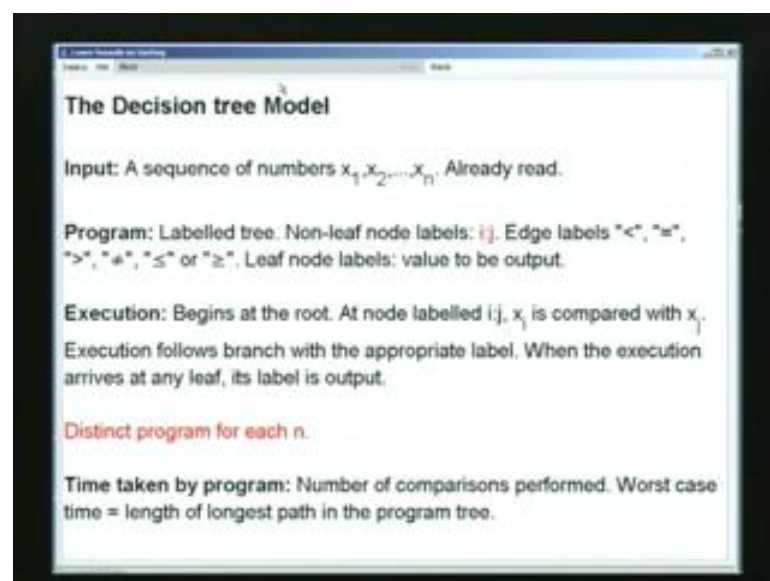


What about non-trivial bounds? It turns out that on random access machines RAM's, proving non trivial bounds is actually very difficult. And here are some of the reasons. Basically when we assert a problem lower bound, we are saying something about every possible algorithm. Now, on RAM's, the space of all possible algorithms is really huge; and it is tricky to analyze. So, a single problem may have lots and lots of algorithms.

Enumerating those algorithms or analyzing them in some structured fashion, is often a very tricky business. It is tricky, because of some of these reasons, a RAM has many instructions. RAM has many instructions to do all arithmetic, they can be composed to do more complicated things. You can take logs, you can exponentiate, you can take trigonometric functions. You can do all such things with Ram instructions. There are also many control flow pattern, there is looping, there is recursion and so.

If I give you a RAM program, analyzing it is pretty difficult. And saying something about all possible RAM programs is even more difficult. So, here is what is typically done. So, we define a simpler computational model, which say does not have that many instructions. And which does not have that many control flow patterns. So, we define such computational models and then, we analyze that. It will turn now, that on such models the space of programs or space of algorithms is actually fairly small. Well, it is never really small, but it is much easier to visualize. So, that is what we are going to do next.

(Refer Slide Time: 11:13)



The Decision tree Model

Input: A sequence of numbers x_1, x_2, \dots, x_n . Already read.

Program: Labelled tree. Non-leaf node labels: i, j . Edge labels " $<$ ", " $=$ ", " $>$ ", " \neq ", " \leq " or " \geq ". Leaf node labels: value to be output.

Execution: Begins at the root. At node labelled i, j , x_i is compared with x_j . Execution follows branch with the appropriate label. When the execution arrives at any leaf, its label is output.

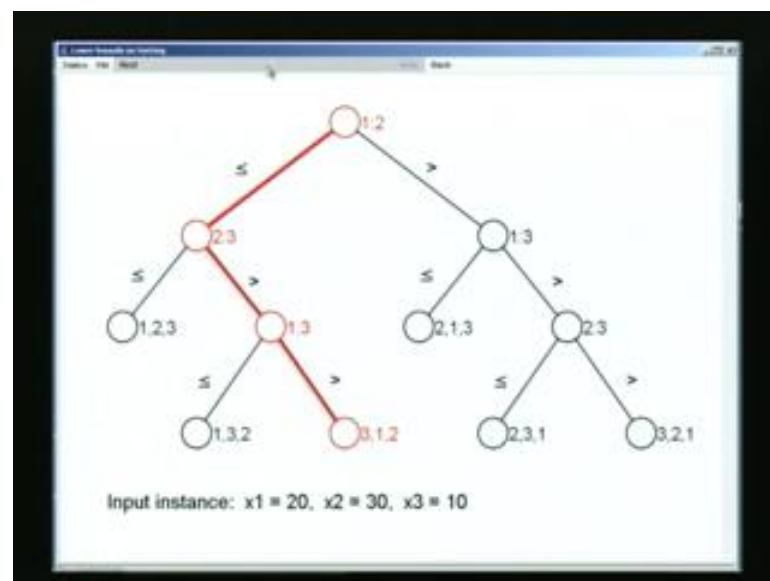
Distinct program for each n .

Time taken by program: Number of comparisons performed. Worst case time = length of longest path in the program tree.

So, here is one such model, this model has been introduced in the context of algorithms such as sorting. So, it will instructions which are relevant for sorting, but which are perhaps not very useful for other problems, but since typically we will consider this model in the context of sorting. This model will actually be a reasonably good model to look at. So, here is what the model looks like. So, the input to every algorithm is always going to be a sequence of numbers.

And we are not going to be worried worry about inputting those numbers. We will assume that those numbers have already been read. And they are already stored somewhere in the model. A program for this model is a labeled tree. So, every non-leaf node has labels of the form i colon j , where i and j are integers. I will tell you exactly what a label means in a minute, but let me just describe the model first. Each edge also has a label and the label could be any of these relational operators. Leaf node labels are the values, that are going to be printed. So, that is what the structure of a program is.

(Refer Slide Time: 12:40)



So, here for example, is a program which will be used for sorting 3 numbers, which can sort 3 numbers. I have not told exactly how it sorts 3 numbers, I will tell you that. But, I just wanted to give you a picture for the program tree model, that I just mentioned. So, as you can see, the label over here is 1 colon 2. This edge is labeled with greater than and so on. And the leaf is labeled with 3, 1, 2 which is what it is going to output, as per the execution model, which I will describe next.

((Refer Time: 13:17)) The execution in this model begins at the root, at any node labeled i colon j input x_i , which came over here is compared with input x_j . The result of this comparison is some relational label, so it is either say actually its either less than equal to or greater than. And based on that result, the execution follows the appropriate branch. So, say for example, the result is less than. Then, we will look for branches having either the label less than. Or the label less than or equal to branch on that.

We will require that only one out of less than or equal to and less than be a label present, on the outgoing edge. This is because, we want our algorithms to be deterministic. So, once we make the comparison, we want a unique path to follow outwards. When the execution arise at any leaf, the label of the leaf is output. So, let us try this out for this program, that we have drawn over here. So, as I said this program is going to be used for sorting 3 numbers.

So, let us say the input instance is x_1 equal to 20, x_2 equal to 30, x_3 equal to 10. As we said the execution starts by looking at the root by starting at the root. At the root we compare x_1 with x_2 . So, in general if the label is i and j , we compare x_i with x_j . So, when we compare x_1 which is 20, with x_2 which is 30, we discover that x_1 is smaller. So, if it is smaller, then we follow this branch.

So, we follow this branch and we arrive at this node. And when we arrive at this node, we have to act according to the instruction, represented by this node, which is to compare x_2 and x_3 , x_2 is 30, x_3 is 10. So, therefore, we find that x_2 is bigger than x_3 . And therefore, we follow this branch. Then, we perform the instruction represented by this node, which requires to compare x_1 and x_3 . So, x_1 is 20, x_3 is 10, x_1 is bigger.

So, we arrive at this node and at this node, we just output. So, we output 3, 1, 2 which is representing our conclusion, that x_3 is the smallest, x_1 is the next smallest. x_2 is the largest and indeed you will see, that x_3 is 10 which is the smallest of 10, 20, 30. x_1 is the next largest and x_2 is the largest. So, at least for this instance, we have checked that this decision tree, which represents a sorting program has in fact, correctly sorted this input instance.

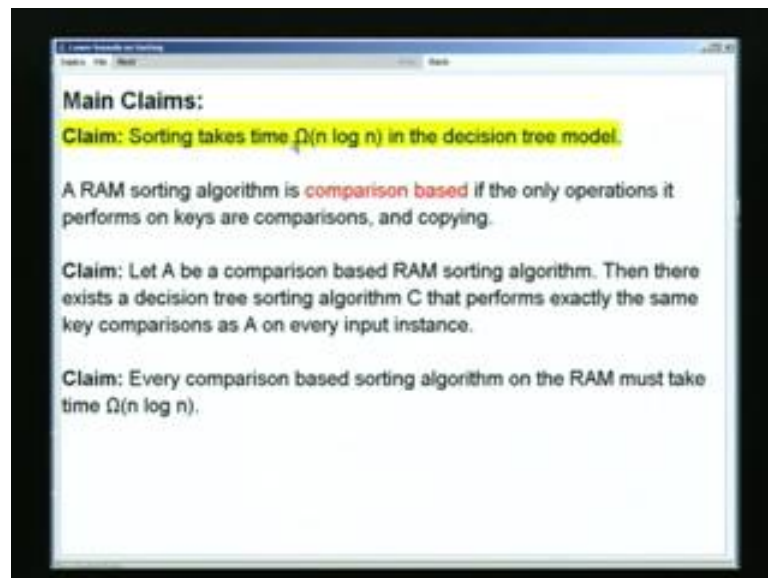
Notice that if I increase the size of my input, ((Refer Time: 16:44)) I will have a different program. I will have to have a different program. This is unlike what we usually do in a RAM model. But, that is we are going to allow that in this model, to complete the

discussion of the model, let me just mention that the time taken by the program is simply the number of comparisons performed for each instance.

The worst case time is equal to the length of the longest path, in the program tree. Because, if in any execution you follow that path, that is the time you will end up setting. Likewise, we can also define average case time. So, here the input instances will be chosen randomly with equal probability, from all possible input instances. Or we are just asking for an average time over all possible input instances.

And therefore, we should consider all possible root to leaf paths. And it is simply the time for each path is, it is length. And so the average case time, is just the average root to the leaf path length. So, notice that, the time taken has a very nice graphical interpretation in this model. So, now here are the main claims, that I am going to make.

(Refer Slide Time: 18:03)



The first claim is that sorting will take time $n \log n$ in the decision tree model. This claim has to be understood properly. This claim says, that no matter what algorithm you use. And by algorithm we mean no matter what tree you use. Because, that is the space of all possible algorithms, no matter what we use that time taken will be $n \log n$. The length of the longest path in the tree, will be at least $n \log n$. So notice that, this is already a non trivial bound, because a trivial bound will be omega of n .

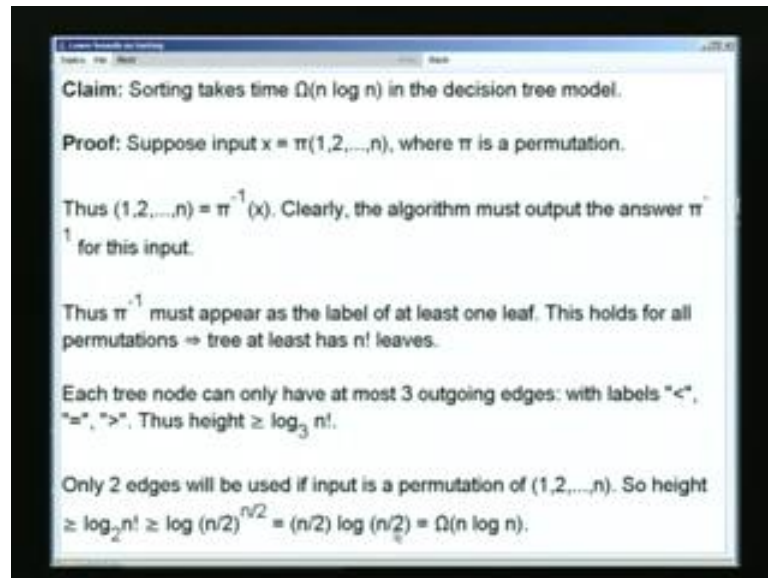
We will prove this in a minute, but I first want to relate all this to a RAM model. Because, RAM is after all what we use every day. So, a definition first, RAM sorting algorithm is said to be comparison based, if the only operations it performs on keys are comparisons and copying. So, if you go back and think about the sorting algorithms, that you have seen. There is a good chance that most of them are in fact, comparison based.

Take heap sort for example, the only operations you do on keys, are compare to keys. And maybe you copy that, similarly for merge sort, similarly for insertion sort. Similarly, for shell sort, if you have studied that. So, in fact, most of the algorithms, that you have studied probably are comparison based algorithms. And it turns out, that they are actually quite nicely connected to decision tree algorithms.

So, here is the claim, let A be a comparison based RAM sorting algorithm. Then, there exists a decision tree sorting algorithm C . That performs the same key comparisons, as A on every input instance. In fact, in the same sequence, this means that the time on this decision tree model, is intimately related to the time on the RAM model also. For this class of algorithms. The final claim says, every comparison based sorting algorithm on the RAM, must take time $n \log n$. So, this is where we have finally, come back to the RAM.

So, here is a non trivial result. Not about all sorting algorithms, but for comparison based sorting algorithms. For comparison based sorting algorithms, we have proved that the time taken must be at least $n \log n$. Well, we have claimed and we will have proved this in a minute. So, let us try proving this first this. This in fact, turns out to be the main, the crux of the matter. So, we want to prove that sorting takes time, $\Omega(n \log n)$ in the decision tree model.

(Refer Slide Time: 21:04)



So, we are going to start with a certain input x . Remember x is that sequence x_1, x_2, x_3 . And let us suppose, we have a sequence x which is the permutation of the sequence 1 through n . Permutation is just the rearrangement of that sequence. So, x is π of this. So, what can we conclude from that, well then 1 to n must be π inverse of x . So, if I apply the inverse permutation on both sides, I will get π inverse over here, and 1 to n over here.

So, notice that when we do the sorting, we actually are supposed to compute this inverse permutation. So, every leaf the answer, essentially must be π inverse 1, must be π inverse. So, essentially π inverse must appear as a label of at least one leaf, essentially, because that leaf must have somehow identified this permutation. And only then, can it name how this permutation can be unscrambled to get 1 to n . So, π inverse must appear as a label of at least one leaf.

Now, this holds good for all possible permutations. Not just a particular π , how many permutations are there, well there are n factorial permutations. And therefore, there are n factorial leaves. So, this is the key insight. We have proved that this tree, the algorithm tree must have at least n factorial leaves. Each node of the tree can only have at most 3 outgoing edges. Well there are so many relational operators which we mentioned. But, we cannot use them simultaneously, as I mentioned earlier.

((Refer Time: 23:15)) We can only have, so here is a node. If we have less than here, then we cannot have less than or equal to over here. We must have something which is disjoint from this. So, the only possibilities are we have greater than or equal to, in which we can have only 2 labels. Only 2 outgoing edges or we could have 3 outgoing edges less than, equal to and greater than.

So, at most 3 outgoing edges are possible. Now, each tree node can only have 3 outgoing, if it can only have 3 outgoing edges. Then, the height must be log of the number of the leaves to the base 3. The height of a ternary tree, is log of the number of leaves to the base 3 at least. And therefore, height must be log of n factorial to the base 3, because n factorial is the lower bound on the number of keys number of leaves, turns out however that if you are using only permutations of this, then we will never pass through this equality edge. So, which means the n factorial leaves must be accessible only passing along the strict inequality branches. So, which is as good as saying that, if our input is permutation of 1 to n . Then, we are only considering a sub tree whose degree is at most 2. And therefore, the height is at most log of n factorial to the base 2.

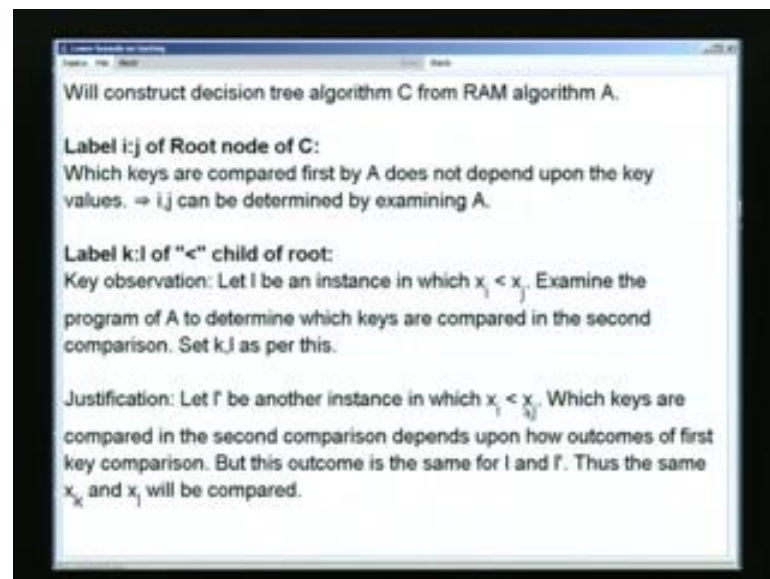
But, log of n factorial to the base 2, can be now thought as to expand out the product. You can see that it is this and this is actually easy to see, so let us do that. So, n factorial is n times n minus 1 times n minus 2 and so on. Somewhere in this, we are going to have n upon 2 and then, there are going to be some more terms over here. But, all these numbers you can see, are going to be at least n upon 2. And how many such numbers are there.

Well, there are n over 2 numbers each of which is at least n over 2. So, therefore, n factorial is bigger than n over 2 to the power n over 2. So, what we have now is log of n factorial to the base 2, is at least log n over 2 to the power n over 2. Well, we can simplify this. And we get n over 2, this is equal to n over 2 log of n over 2 and therefore, this is $\Omega(n \log n)$ ((Refer Time: 26:04)). So, what you have done, so far is that we have proved this. Now, we want to look at the next claim.

The next claim is the one which relates comparison based sorting algorithms in the RAM, to decision tree sorting algorithms. So, the claim says that let A be a comparison based sorting algorithm. Then, there exists a decision tree sorting algorithm C , that performs exactly the same key comparisons as A on every input instance. So, we are

going to prove this and the proof is actually going to be constructive. So, what we will do is, we will assume that we have been given this algorithm A which is a comparison based RAM sorting algorithm. And then using that we will construct a decision tree sorting algorithm C. And we have to make sure, that C performs exactly the same key comparisons as A does on every input instance, and also in the same sequence.

(Refer Slide Time: 27:22)



So, this is what we are going to do, we are going to construct C from RAM algorithm A. So, let me remind you what C looks like. So, C is going to be some root node ((Refer Time: 27:37)), may be some branches may be nodes over here and so on. So, if I tell you how these things are going to be labeled, then I am done. So, let me tell you the edge labels right away. The edge labels are going to be less than over here, equal to over here greater than over here. So, let us keep it really simple and similarly subsequently, but I have to tell you what the label here is going to be.

So, the question is how do I do that, so it is going to some i colon j. But, I have to figure out what i colon j, it is going to be. And here is the key insight in figuring this out. The main idea is that which keys are compared first by A, is our RAM algorithm. Does not depend upon the key values, why is that? Well, A is going to read things and then, it is going to compare. It is not going to look at the keys and base its decision on which keys to compare on the value of any key.

Because in fact, value of any key, does not really come into the control structure of A, other than in the comparisons that A does. So, if A does a comparison, that is the only time when A can actually peep inside a key value. So, clearly the first time that A peeps inside a key value, is the first time A does a comparison. And in fact, that which keys it compares is going to be the same no matter what the values of keys are.

So, i, j can be determined by examining the algorithm A. We just look at the code of A. And will know which key is going to examine first. So, whatever they are we can write that now over here, i colon j . We find actual numbers i and j and we put them down over here. So, I claim now that whatever numbers i and j be put down over here. They will be the same, no matter what the values of key are and that is what I just explained. So, next we want to find the labels of say this node over here, to do that here is what we consider.

So, suppose i is an instance, in which x_i is less than x_j . Now, we are going to examine the program of A to determine, which keys are compared in the second comparison for this instance i . Suppose, x_k and x_l are compared. So, those values, those k and l , we will put down over here. So, now what is the idea? So, for instance I what will happen, the first comparison will be this. Then of course, in the decision tree, this will be the branch taken and the second comparison will be this.

So, instance I , in fact in instance I , the first two comparisons have been the same in the decision tree. As they have been in our algorithm A, so far so good. So far we have built this part of our tree. And indeed in this tree the decision, the comparisons made on this instance, are the same as comparisons made on the instance, in our RAM model also. But, let us consider another instance I' , what will happen in that.

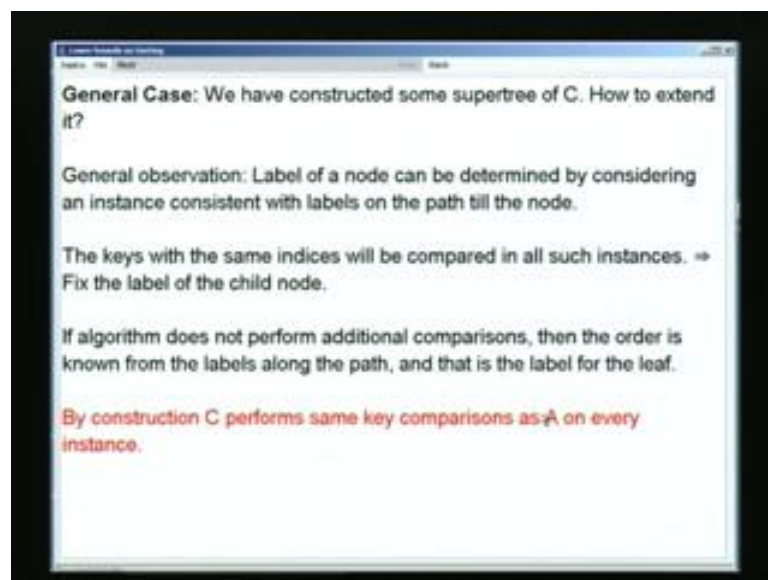
So, if we consider another instance, I' in which x_i is less than x_j what will happen, which keys will be compared next. I claim that even in that case, the keys to be compared will be just be these x_k and x_l , why is that? So, let us examine this, ((Refer Time: 32:06)) so here is an execution of I and here is an execution of I' . So, somewhere we start execution. So, we compare x_i versus x_j . And we know that the first comparison is the same in both and so we compare x_i versus x_j .

Then, we do some instruction over here and we compare x_k versus x_l . So, when we do this comparisons, the question is what will happen over here. I claim that if in these instructions, no key is even touched. Then, exactly the same instructions will be executed

over here. Why is that, well the control flow can change, only if there is a comparison, and if the comparison has a different result. So, in this entire part, there has been a comparison with keys, but the result over here is the same as the result over here.

Because, in both cases we said that x_i is less than x_j . So, during this entire portion and this entire portion, wherever there have been comparisons. The comparison outcomes have matched. And therefore, the instruction which is going to be executed over here, is going to be exactly the same. And therefore, we will also compare x_k and x_l over here. So, which means even for this instance I , I' k and l , x_k and x_l will be compared. And therefore, we can put down this label of the node as k compared to l . Now, we can generalize this.

(Refer Slide Time: 33:46)



So, the general pattern is, we have constructed some super tree of C , how do extend it. So, we have constructed something like this, and so in general we might have had constructed some path. And then, we want to extend this path outwards, how do we do that. So, here is the observation, the label of a node can be determined by considering an instance, consistent with the labels on the path till the node.

So, I want to determine the label of this node, how do I do it. Well, I look at instances on the path from the root, I look at the labels on the path from the root to this node. And I construct a instance, which is consistent with these labels. So, which means that x_i must

be equal to x_j . And then, whatever the label is over here the corresponding keys here must satisfy this label and so on.

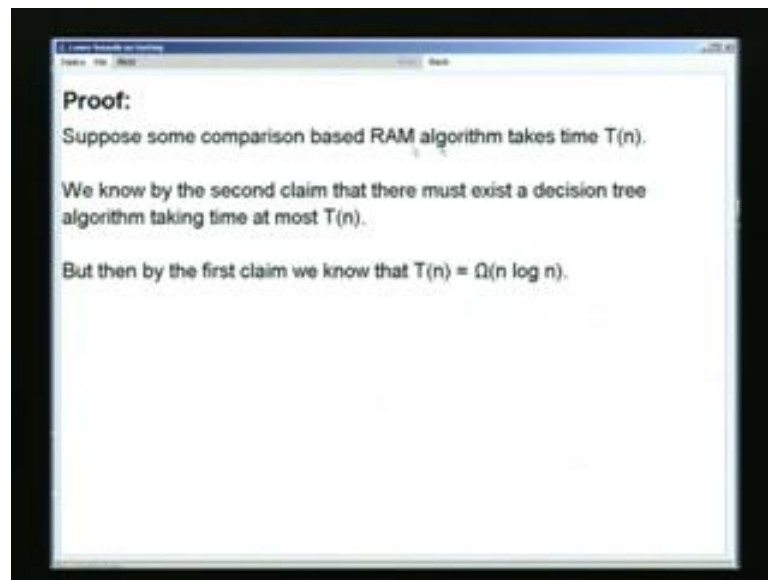
And then, we simply look at the algorithm for A. And ask what is the next comparison, that is going to be made. And we put down those labels over here. So, we have constructed an instance, which is consistent with the labels along this path. And we look at the algorithm and decide, what is the comparison that is going to be done by the algorithm in that instance.

So, if the comparison is between some x_p and x_q . We put down p colon q over here ((Refer Time: 35:21)). So, exactly like what we did in this case, but just more general. So in fact, in all such instances, which are consistent with the labels along this. You can argue exactly in the same way, that x_p will get compared with x_q . And that will fix the label of this. If it turns, that the information which has been gleaned along this path, is enough to determine the final sorted order.

And because of which the algorithm does not do any comparisons at all. Then, well we simply take that sorted order. And put back the label and name and make that leaf. Now, the important point is, that we have constructed our decision tree algorithm in this manner. And by construction, because of the care that we took in the construction. You can see that C performs, exactly the same key comparisons as A does. And exactly the same sequence on every instance.

((Refer Time: 36:31)) So, this finishes this claim. So, now I want to argue that every comparison based sorting algorithm on the RAM, must take time $n \log n$. But, this is actually fairly straight forward.

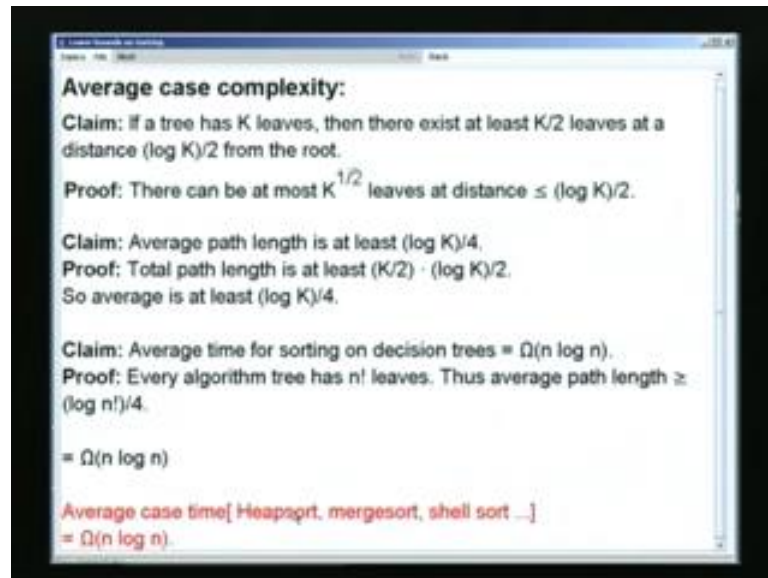
(Refer Slide Time: 36:41)



So, suppose some comparison based RAM algorithm, takes time T of n , what do we know about comparison based RAM algorithms, by the claim that we just proved. We know that there must exist a decision tree algorithm, taking time at most T of n . But, the first claim tells us something about decision tree algorithm. The first claim says that the time taken by the decision tree algorithms, must be at least $n \log n$.

So, T of n had better be at least $n \log n$, but then this T is the same as this T . And therefore, comparison based RAM algorithms must also take time $n \log n$. So, we have proved this claim ((Refer Time: 37:29)). So, notice that in a single claim, we have proved a lower bound on a variety of sorting algorithms, heap sort, merge sort, shell sort, insertion sort, and many other algorithms, which we have not even thought about. But, which only if they are comparison based, then this lower bound will apply to them.

(Refer Slide Time: 37:57)



Next we turn to average case complexity. So, basically we are going to ask the question, what is the average case, can we put a lower bound on the average case complexity. So, on the average time to sort, not just the worst case time. So, this actually first follows fairly easily, just based on structures, on some properties of trees. So, here is the first claim, if a tree has K keys, then there exist at least $K/2$ keys at a distance $\log K/2$.

Why, well there can be at most square root K leaves at a distance, just like worst case. We can use similar arguments, to prove bounds on the average case complexity of sorting as well, on the decision tree model of course. And of course, that will imply some lower bounds, on comparison based sorting algorithms. As far as the average case complexity is concerned. And these bounds follow very simply from some properties of trees, with respect to the relation between the height of the tree and the number of leaves.

So, here is the first claim, if a tree has K leaves, then there exist at least $K/2$ leaves at a distance. At least $\log K/2$ from the root. Well, here is the proof, there can be at most 2^l leaves at a distance l , which means there can be at most $2^{\log K/2}$ leaves at a distance $\log K/2$, which is nothing but $K^{1/2}$, square root of K . So, that is what is claimed over here. The rest of the leaves have to be at larger distance.

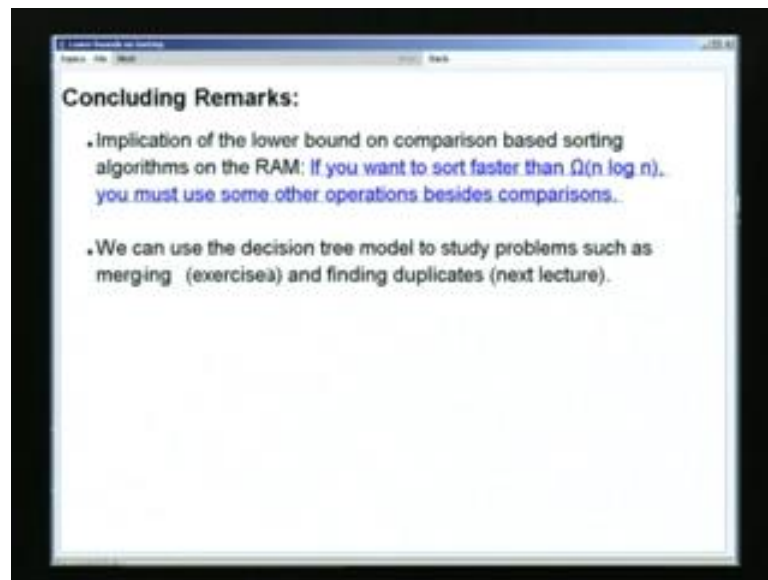
And in fact, this says that as many as K minus square root K , have to be at a larger distance, whereas what is claimed over here is just K by 2. So, the claim is very easily proved. In fact, we can prove something stronger. But, once we have that, we can prove the following, we can prove that the average path length. Average is taken over all leaves, is at least $\log k$ upon 4, why is that? Let us count what the total path length is.

The total path length, we are going to calculate first for these leaves which we talked about. So, there are K over 2 of them. And each has a path of length $\log K$ upon 2. So, the total path length just for these K over 2 leaves is at least this much. So, now the average is going to be a K th fraction of this. And therefore, it is simply going to be this K factorial drop out, this 2 and this 2 will give us 4. So, it is going to be $\log K$ over 4.

From this it immediately follows that, the average time for sorting on decision trees, is $n \log n$. And here is why, every algorithm tree has n factorial leaves. Thus average path length is bigger than this claim, \log of the number of leaves or the number of or \log of n factorial upon 4. But, \log of n factorial upon 4 is $n \log n$. And therefore, average time for sorting on decision tree is this much. And this means that the average case time, for all comparison based sorting algorithms.

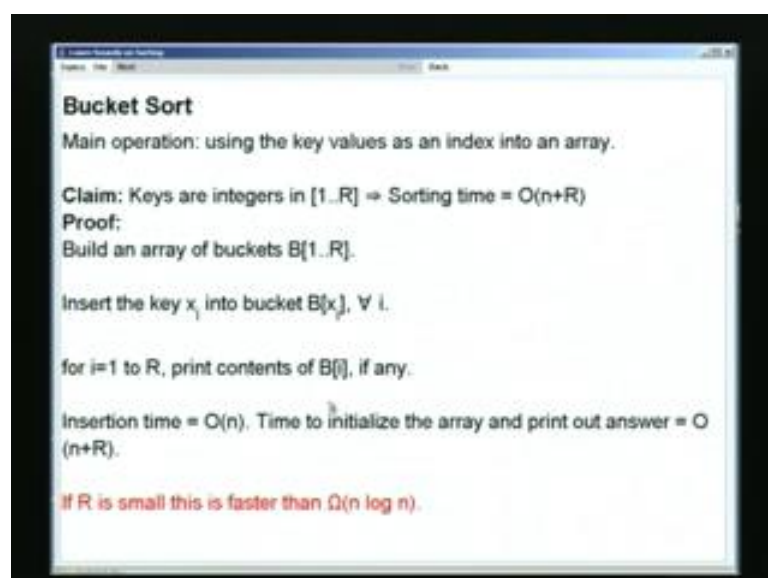
Say heap sort, merge sort, shell sort all of these is $n \log n$. So, when we did, when we studied average case complexity of quick sort. We proved that although the worse case complexity for quick sort was n square. The average case was $n \log n$. So, in a similar manner, it may be conceivable, that heap sort may have worse case complexity of $n \log n$. But, its average case complexity is just n , it might have been conceivable. But this result shows that the average case time for heap sort, is also going to be at least $n \log n$. And similarly for merge sort, shell sort and any other comparison based sorting algorithm.

(Refer Slide Time: 42:51)



So, now I want to conclude and here are some remarks. First some implications of the lower bound, on comparison based sorting algorithms in the RAM. So, here is an important application. So, if I want to sort faster than $n \log n$, what do I know, well I know, that I must use some other operations besides just comparisons. If I limit myself to just comparisons what happens. Well, there is a decision tree algorithm, which mimics the RAM algorithm. And therefore, I am stuck to a lower bound of at least $n \log n$. So, I must use some other operations besides comparisons, are there such operations there actually are...

(Refer Slide Time: 43:46)



So, in fact a popular sorting algorithm called bucket sort, does exactly this. The main operation it performs, is that uses the key values, as an index into an array, which was not allowed in comparison based sorting algorithms. So, here is the main claim, if the keys are integers are 1 to R , in the range 1 through R . Then, sorting time is at most n plus R . So, I will just give the algorithm as the proof.

So, first we are going to build an array of buckets B_1 through R . So, these buckets will be initialized to null. So, that will take about R time. Then, we take key x_i and we insert it into this bucket B of x_i . So, B of x_i can be compute in constant time. So, in constant time, we keep a list over there and we insert this key into that list. And we do this for all the keys, for all i . So, now all keys are now into their buckets.

Now, for i equal to 1 to R , we go visit each bucket in turn. And we print out the contents, clearly we will print out the contents in increasing order. What is the time taken, well the insertion time is proportional to the number of keys. So, because of this statement, the time taken to initialize the array of this B array, is going to be R . And printing out the answer, well we have to visit every element of the array. And we also have to visit every key, which has inserted in some bucket.

So, the time taken is going to be O of n plus R . So, the total time is going to be O of n plus R . And this is going to be faster than $n \log n$, if R is small. In fact, this is not the only range, if you want some bigger ranges say 1 through R^2 . Is slight modification of this will in fact, do the job. ((Refer Time 45:56)) The second concluding remark that I want to make, is that not only sorting. But, you can use the decision tree model to study several other problems.

So, the other problems are say merging. And this will figure in the exercises, which I will show later. And also things like finding duplicates. And this is going to be the topic in the next lecture. In fact, let me point out, let me say in conclusion. That in fact, we can extend this decision tree model. So, I want to go to my second concluding remark, which is to say that the decision tree model, can be used to study other problems as well, not just sorting.

So, in particular it can be used to study merging problems, which I guess you might think as very similar to sorting. It can also be used to study problems, like finding duplicates. Or it can also be used to study problems, such as putting lower bounds on the time to

calculate the intersections, between two given sets. Now, this problem, might seem like rather different from sorting. And it is perhaps somewhat surprising, the decision trees can be used for this.

Nevertheless, they can be used and in fact, we will see the finding duplicates example in the next lecture. I will also state that the idea of decision tree, is the general model of decision tree is can be extended somewhat. So that, rather than just allowing comparison, you can also allow some arithmetic incident. And so the lower bound model can be made somewhat stronger. In any case, we have seen some lower bound results. And we will continue this with the next lecture.

Thank you.