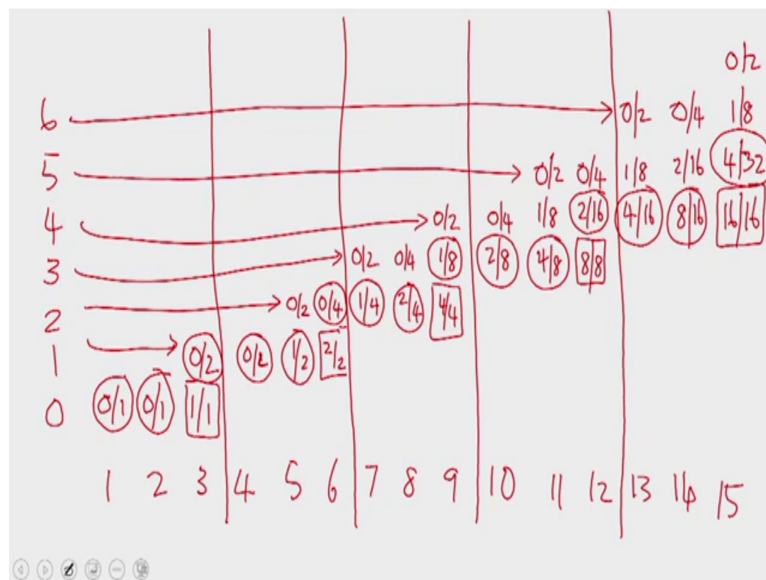
Parallel Algorithms
Prof. Sajith Gopalan
Department of Computer Science and Engineering
Indian Institute of Technology, Guwahati

Lecture – 20
Analysis of Cole's Merge Sort; Lower bound for sorting

Welcome to the twentieth lecture of the MOOC on Parallel Algorithms. In the last two lectures, we discussed Cole's merge sort. What remains to discuss in Cole's merge sort is the fact that the total number of cache and sample elements in all the live would nodes put together, this order of n. Once we show this, we would have established that Cole's merge sort sorts n elements in order of log n time using n processors on a C R E W PRAM. So, let us try to estimate the total number of cache and sample elements.

(Refer Slide Time: 01:03)



Let us consider the various stages horizontally we are representing the various stages and vertically, the various levels. In stage 1, only the leaf nodes are active. They are full, but they have a emitted all the sample elements. Therefore, they have caches of size 1, but samples of size 0.

So, let me denote that by 0 1; so the first step as we know is a dummy step; nothing happens in the first step. In the second step similarly, the sample size is 0 and the cache size is 1 at all the leaf nodes. All the other arrays are empty. Now, in the third stage, the cache will emit all it is elements into the sample. So, the sample and cache sizes will be

identical. Once the sample arrays are filled, then their parents will get the merge of the sample arrays in their cache array. So, the cache array of a level 1 node will now be of size 2. So, at this point, the cache here is full. Let me denote a full cache by a circle and the node that has finished emitting all it is cache elements upwards is represented by a square let us say.

So, our level 1 nodes have just become full. In stage 2, in stage 4, again this is a dummy dummy stage because nothing happens. In stage 5, level 1 nodes will emit every second element into the sample. So, the sample will have a size of 2. Now, once the sample array has the size of has something in it, this will merge to form the cache array at the parent. So, at the parent now we have something in the cache.

So, the cache size at the parent is 2 but mind you the parent has not become full yet. So, let me mark every three steps because that is when one level becomes full. In the sixth stage, the level 1 nodes will emit all their cache elements into their caches in to into their samples and therefore, become dead from here onwards. The size of the cache at the parents will now be 4, still there is no sample because the samples were to be drawn from two elements.

Therefore, nothing could be drawn in step 6. This is how the size of the sample array and the cache array would be. In stage 7, first let us see how the sampling will be done. A level 2 node will be picking every fourth element as a sample. Therefore, we will have a sample of size one here this is already become full; therefore, here we have 1 4. Since, the sample array is filled, now at the third level we will have something in the cache. There will be two elements in the cache. In the eighth stage, level 2 nodes will pick every second element into the sample.

So, now there are four elements; so you will get two elements in the sample array. The level 3 node cannot pick any sample because it is still picking every fourth element but it is cache array now becomes bigger. It will have four elements here. Now, coming to the ninth stage, at this point the level 2 nodes will emit all it is cache elements into the sample array. The level 3 nodes will pick a sample; therefore, they will have a sample array of size 1 and of course, above that since there is nothing in the cache, there is nothing in the sample as well. So, we decide on the samples in this fashion, then we can decide on what the size of the cache will be the cache will be the two samples of the

children merged. Therefore, here we will have 8 which is 4 plus 4 and here we will have 2 which is 1 plus 1.

So, this node has become dead now because it has emitted all it is cache elements into the sample and this has just become full. So, at the third stage, level 3 nodes have become full. Now, we come to the tenth stage. In the tenth stage, first we draw the samples. From 2, we cannot draw any samples. So, there is no sample here. From 8, we will be picking every fourth element as a sample. So, we get two elements. So, the cache sizes will be like this 2, 8 and 0, 4 sample and cache sizes. Level 3 nodes are already full; they became full by the ninth step.

Then, when we come to the eleventh stage, there will be samples drawn. At the fifth level, there will be no sample to be drawn. At the fourth level, we will have a sample size of 1 because we are picking every fourth elements; out of four, one will come. And at the third level, we will be picking every second element. So, we will be picking four elements into the sample. So, the cache sizes are like this. At level 3, the cache sizes 8; it is already full. So, it is not going to change any more. At level 4, we have 8 again because the sample arrays of the children are of size 4 each and at the fifth level, we have a sample size of 2. So, that is how the sizes look in stage 11.

Now, we come to the twelfth stage. In the twelfth stage, when we draw the samples here, we find that the level 3 node gets all the cache elements into the sample array and therefore, it becomes dead here after. Then, the other samples would be like this. At the level 4, nodes we pick every fourth element.

So, we get a sample size of two and here of course, we are still continuing to get a sample size of 0. So, the cache sizes would be 2 plus 2, 4 here and 8 plus 8, 16 here. This has just become full. The level 4 node has become just has become full just now. Now, when we come to the thirteenth stage, in the thirteenth stage we have to pick every fourth element at this node.

So, we get four elements and every fourth element here again. So, we get one element here and above that way, of course, do not have anything to pick at; level 6, we have zero elements. So, the cache sizes would be 2 here, 8 here and 16 here of course, so that is how the sizes would look like in stage 13. When we come to stage 14, here we would be

picking 8 elements into the sample. Every second element from the right hand side is picked.

So, the sample size would be 8. Here, we are picking every fourth element. So, out of eight, we are getting two elements. And here, we will not be able to pick anything. So, here the sample size continues to be 2, but the cache size will be 2 plus 2, 4 here and 8 plus 8, 16 here. Now finally, when we come to the fifteenth stage, here we will be picking every element into the sample and here we are picking every fourth element into the sample and here we will be picking one element into the sample every fourth again and here of course, the 7th level has not opened. Therefore, it is cache size is 0.

But now, it is going to get a the sample size is 0, but now it is going to get two elements in the cache. And here we will get 4 plus 4, 8 and here we will get 32 which is 16 plus 16 and this of course, had already become full. So, this node becomes dead now. So, in the fifteenth stage, the fifth level node becomes full. So, this is how the cache and sample sizes vary through the stages. So, now let us make some observations.

(Refer Slide Time: 10:37)

In stages 3r
→ n/64 :
→ n/8
→ n
15) 32 → 8 → 2
Total cache size at the parent : $\frac{1}{8}$th cache size at children

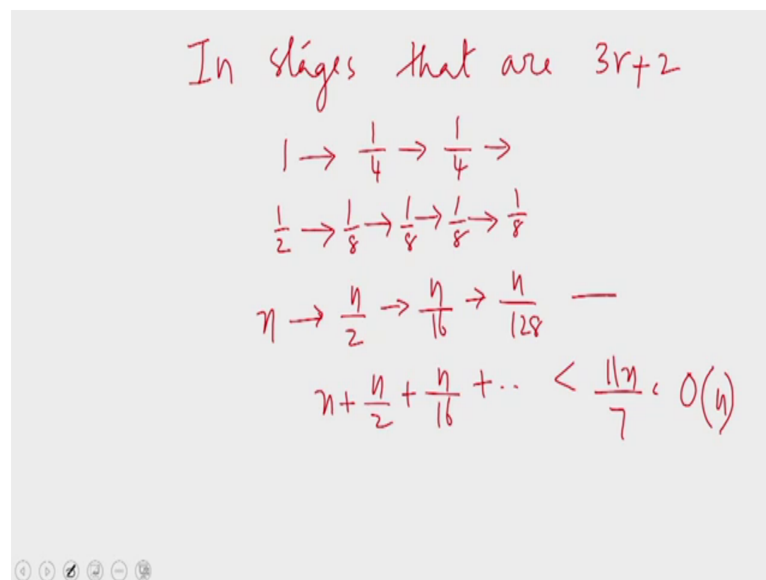
$$n + \frac{n}{8} + \frac{n}{64} + \dots < \frac{8n}{7} = O(n)$$

In stage numbers of the form 3 r, that is stage numbers which are multiples of 3; for example, when we consider 15, we find that the cache sizes are decreasing 32, 8 and 2. For example, in the 15 stage, the cache sizes are going 32, 8 and 2 which means the cache sizes are falling by a factor of 4 as you go from children to the parents. Therefore,

the total cache size at the parent is one eighth of what did is at the children; that is when you put both the children together.

So, what we find is that the total cache size at the parents level is one eighth of the total cache size at the childrens level. So, if you start from the bottom most live level, there are n elements in the cache there, they are all the full nodes. Therefore, above this we will have n by 8 elements, above this we will have n by 64 elements and so on. Therefore, if you sum the total number of cache the elements in the cache arrays would be, there is a total size of all the cache arrays put together will be n plus n by 8 plus n by 64 plus etcetera; all the way to the top of the tree which we know is less than 8 n by 7. This is order of n. So, in stages that are multiples of 3, the total cache size would be order of n.

(Refer Slide Time: 12:32)



Now, let us consider stages stage numbers, that are 1 modulo 3, that are of the form 3 r plus 1 for some integer r.

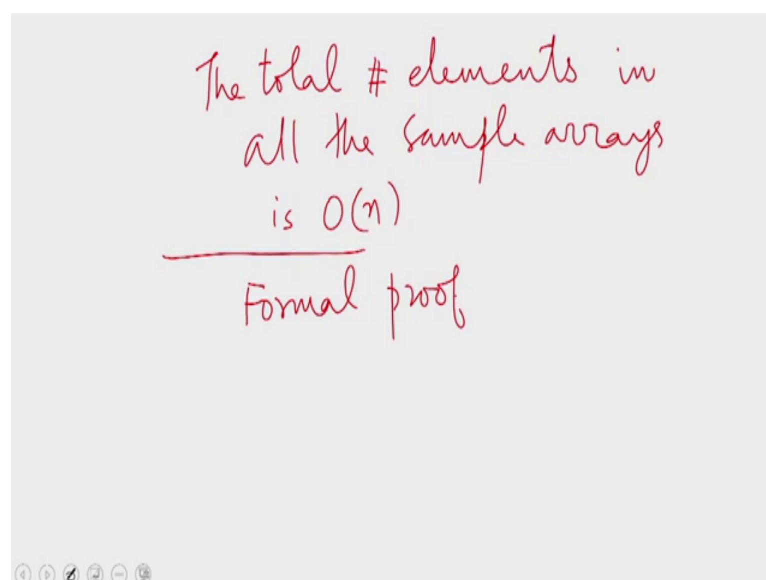
So, in particular if I consider stage 13, we find that the cache sizes are decreasing from 16 to 8 to 2, 16 to sorry here 16 to 8 to 2. The cache sizes are decreasing from 16 to 8 to 2 which means, we can generalize this to note that from the bottom most level to the next level, there is a reduction in size by half. After that, we have reduction by a factor of one fourth. Therefore, the total number of cache elements in the live band would be n plus n

by 2 at the next higher level, you will have a total number of cache elements which is n by 2. After that, it is division by 8 because the factor one fourth is for a child to parent.

But when the when two children are put together, we have a factor of 8 and so on; sorry here it would be n by 4 and here it would be n by 32 here, it would be n by 256 and so on which is less than 9 n by 7, just again orders order n. So, the total cache size of all the live band vertices put together is order of n in stages that are numbered 1 modulo 3. So, when we come to the remaining stages, in stages that are 2 modulo 3, again looking at the pictures we find that, let us take the fourteenth stage, here we find that the cache at level 4 has a size of 16, at level 6 5 again has a size of 16 and after that it is falling by a factor of 4.

So, we will have a factor of 1 followed by factors of one fourth. Therefore, if you consider the parent to when both the children are put together first, we will have half then one eighth, then one eighth and so on which means at the lowest level we have a total of n cache elements and the next level we have n by 2. Then, we have n by 16, then n by 128 etcetera. So, when you add up all the number of cache elements, you find that the sum this again order of n, this is 11 n by 7 which is again order of n. So, what we have established is that the total number of cache elements in all the vertices of the live band would be order of n.

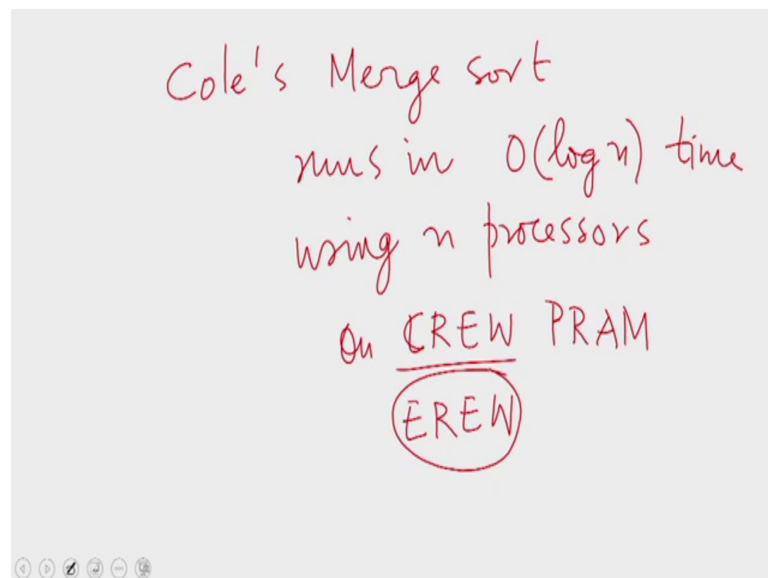
(Refer Slide Time: 16:27)



A similar analysis for the sample elements will also show that the total number of elements in all the sample arrays, this order of n. So, this part I am leaving to you as an exercise and I would also like you to do this formally.

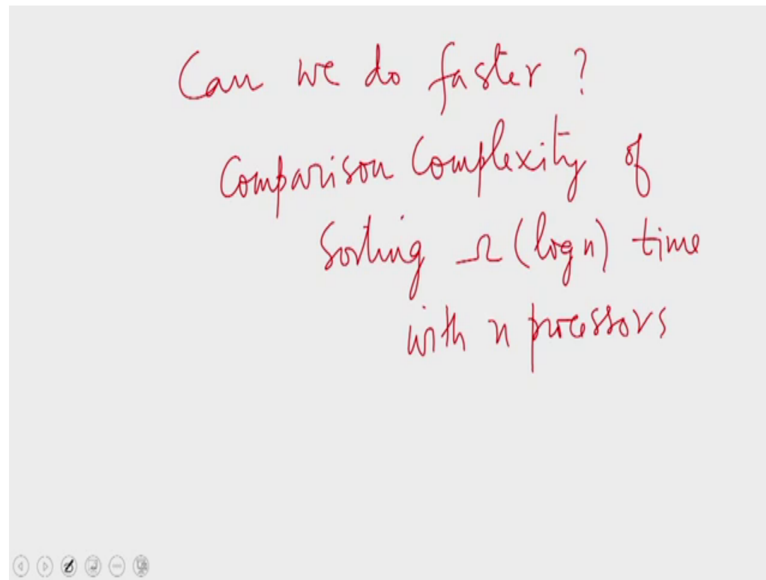
Write a formal proof. Here of course, we have looked at this example and have generalized from this but formally prove that in all stages that are numbered with multiples of 3, the cache sizes would decrease by a factor of 4 as we go up and similarly for the other stages.

(Refer Slide Time: 17:30)



So, this establishes that Cole's merge sort on inputs of size n runs in order log n time using n log n process, using n processors so that the cost of the algorithm is order of n log n. The implementation that we have discussed is for the C R E W PRAM. There exists another implementation a slightly more involved one which involve which runs on E R E W PRAM for the same complexity that is n numbers can be sorted on an E R E W PRAM in order of log n time using n processors. The E R E W version is outside the scope of this course; so we will not be discussing this.

(Refer Slide Time: 18:34)

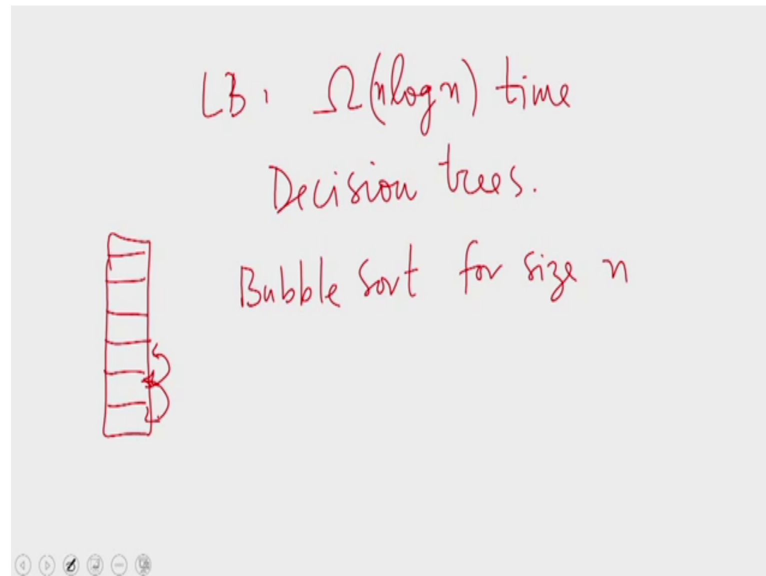


But can we do faster? As it happens, we cannot do faster because the comparison complexity of the problem stipulates that you require omega log n time with n processors or with even a polynomial number of processors. You will not be able to beat the bound of log n with even with the polynomial number of processors.

(Refer Slide Time: 19:30)

So, let us prove this lower bound. Now, a lower bound for comparison based sorting.

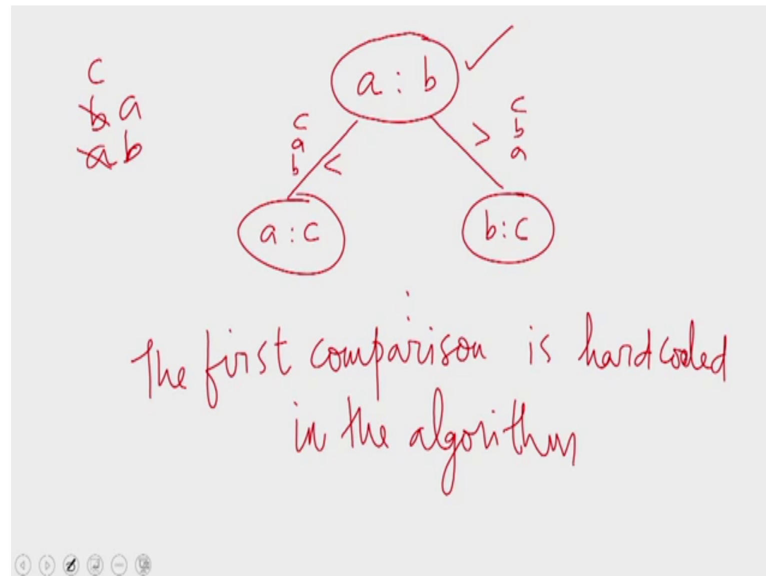
(Refer Slide Time: 20:00)



You are aware of the sequential lower bound which is establishes that to sort n elements using comparisons, you require omega log n time omega n log n time. We prove this lower bound using decision trees. The proof uses the fact that any comparison based sorting algorithm can be translated into a decision tree.

For example, when you have a bubble sort, implementation for size n let us say, we are considering the algorithm for an input of size n. So, in bubble sort what we do is this. We consider an array of size n. We assume the elements are given vertically. So, in bubble sort, we compare the two lowest elements first and then we perform a comparison and exchange if necessary; that is if the smaller lemon happens to be at the bottom, we swap; then we compare the next two positions and if and perform a swap of necessary. By doing this, when we have completed one pass, the smallest element would have reached the top; therefore we can draw a decision tree.

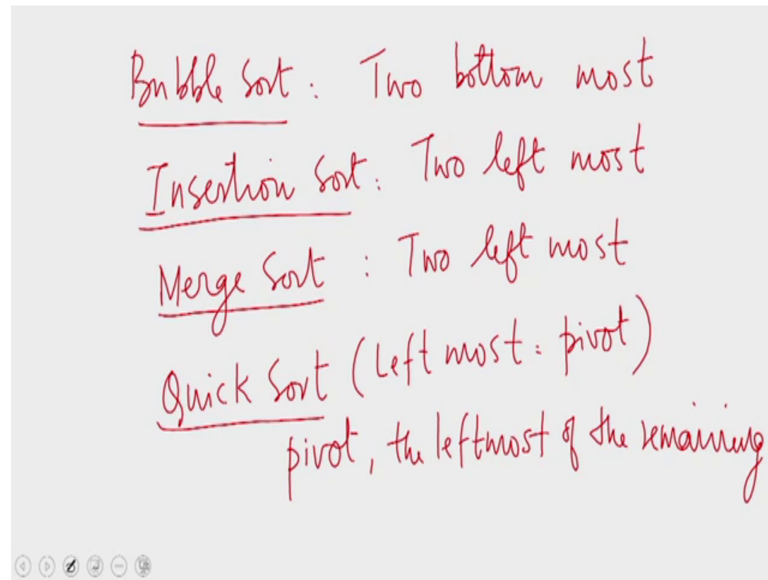
(Refer Slide Time: 21:25)



If we have for example, let us consider an input of size 3. On bubble sort, we go like this. When we have three elements a, b, c, the first two to be compared are a and b. If a is less than b, we do something. If a is greater than b, we do something else. If a happens to be less than b, we swap them. So, we will have the order c, a, b. If it is not we do not perform a swap and we will have to continue dealing with the original order c, b, a. And then, we compare the second element with the third element. So, in this case, we will be comparing a with c. In this case, we will be comparing b with c and so on; we continue to generate the tree.

So, given a comparison based algorithm, you can draw a decision tree in this manner. In particular, what you have to observe here is that the first comparison to be performed is hard coded in the algorithm.

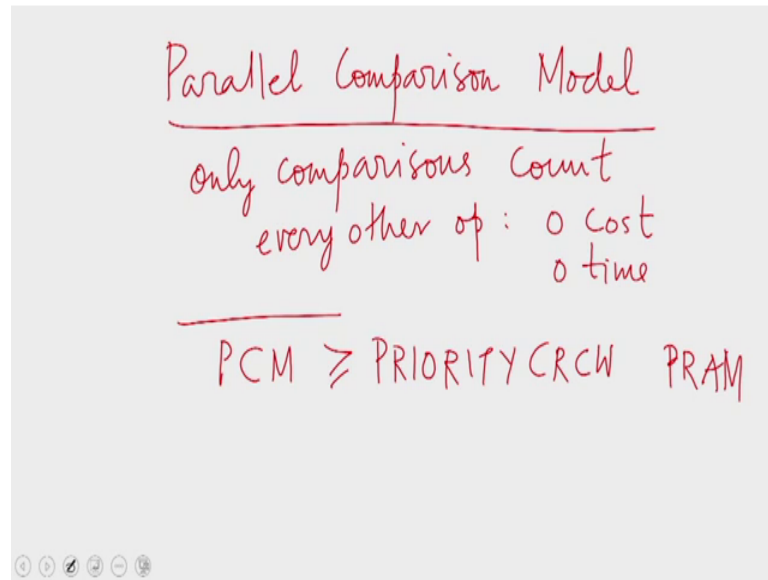
(Refer Slide Time: 23:08)



For example, in the case of bubble sort, we compare the two bottom most elements. These are the two elements to be compared first, irrespective of the input. So, this is a fact that is hard coded in the algorithm. In the case of insertion sort, we would be comparing the two leftmost elements. In the case of merge sort, merge sort is the divide and conquer algorithm. So, you divide the array into two halves and then recursively invoke sorting on each half and then again, we recursively invoke sorting on further halving's and so on.

So, if the recursion always prefers left over right, that is if the left half was executed first, then we would be moving to the left. So, we would end up comparing the two leftmost elements again. If we execute the standard version of quick sort were always the leftmost element is picked as the pivot, the first comparison to be performed is between the pivot and the left most of the remaining. So, for each comparison based sorting algorithm, the first pair of elements to be compared as fixed and based on the outcome of this first comparison, the remaining comparisons will be done. Now, this essentially gives us an idea about how the proof or the parallel sorting should be devised.

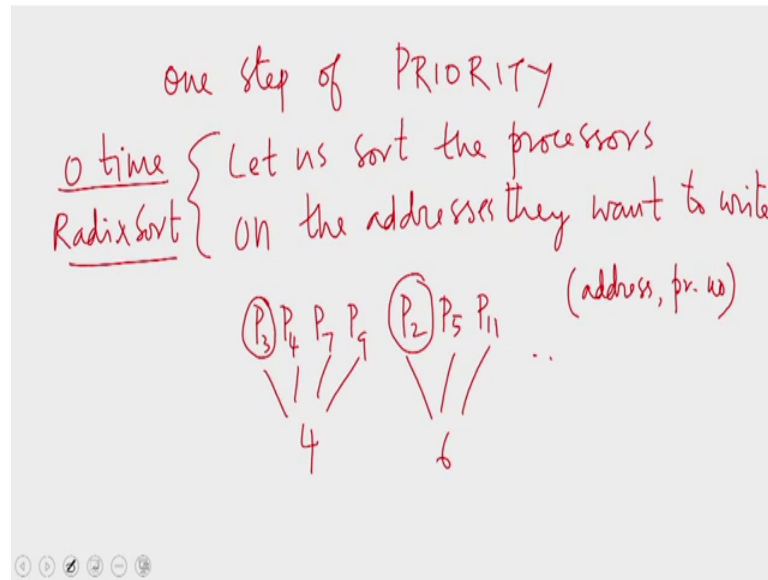
(Refer Slide Time: 25:36)



To devise a lower bound for the parallel sorting algorithms, we introduce a model called the parallel comparison model.

In the parallel comparison model, only comparisons count. What we assume is that, apart from comparisons of the key values involved, every other operation has 0 cost and takes 0 time. This is a simplification that we make for devising lower bounds. Once, we devise a lower bound on a parallel comparison model it should apply to the other PRAM models that we are talking about. This is because a parallel comparison model is at least as strong as a priority C R C W PRAM. How is this so? It can be seen like this.

(Refer Slide Time: 26:57)



Consider one step of the priorities C R C R C W PRAM. In this one step, various processes want to write in various locations. So, let us sort the processors on the address in which they want to write. We sort the processors on the addresses we want to write. Once we do this, all the processors that want to write in the same memory location will come together.

For example, let us say an the number of processors want towrite in location 4. So, we will have several processors, all wanting to write a location for coming together. Then, we will have several other processors wanting to write in a higher larger memory location. For example, let us say these want to write an location 6 and so on. So, the processors will be grouped by the location in which they want to write.

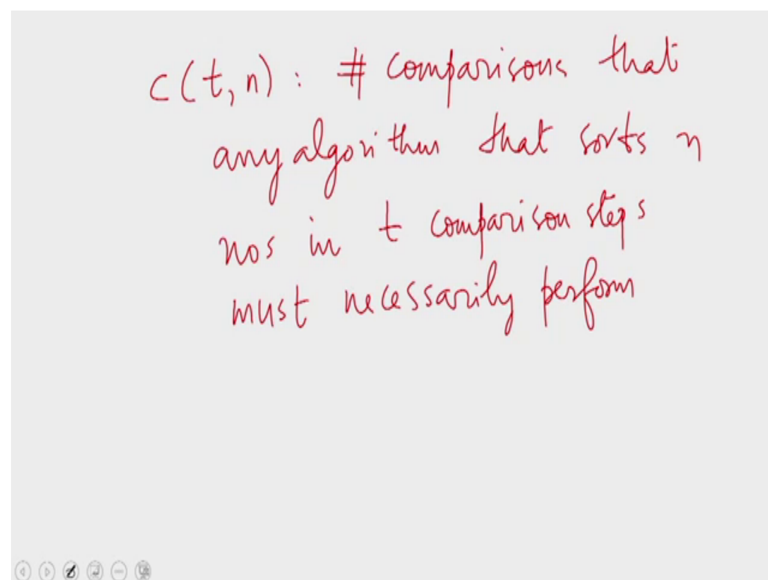
That is if we sort the processors on the addresses in which they want to write, we will get the processors that want to write in the same memory location together, but how much time will this sorting take on a parallel comparison model; the sorting will take 0 time because what we do here is radix sort. Radix sort does not compare the key values. What we have to sort are a set of integers. These are the address values into which the processes want to write.

So, we sort the processes on these addresses that are indulges, we can affect the sort using radix sort and since radix sort does not use comparisons, this can be achieved in 0 time at 0 cost. So, what we assume is that the processors are all sorted on the address into

which they want to write in this fashion. Now, all the processors that want to write in location 4 have come together. All we have to do is to pick the smallest among them. In fact, if we had sorted on the ordered pair which is the address and the processor number, then we would have the processors in the increasing order of their indices. For example, we would have P 3, P 4, P 7, P 9 in this order because we are effectively sorting the ordered pairs 4, 3, 4, 4, 4, 7, 4, 9. Then, we only have to pick the smallest processor, the smallest processor index out of all the processors that want to write in the same memory location.

So, P 3 will be selected from the set, P 2 will be selected from the set and so on. So, all this selecting of the processors come for free on the parallel comparison model and therefore, the writing itself can be done in order one time now after the processors are selected which means one step of the priority C R C W PRAM can be simulated on a parallel comparison model in order one time. What this establishes is that, any lower bound that we prove on the parallel comparison model will also apply to all the PRAM models that we have been discussing

(Refer Slide Time: 30:41)



Let c of t n denote the number of comparisons that any algorithm that sorts n numbers in t comparison steps must necessarily perform; c t n is defined like this. Imagine an algorithm that sorts n numbers in t comparison steps, then the number of comparisons that any that this algorithm must necessarily perform this what c t of n. There is a

minimum number of comparisons that any such algorithm must perform this what c t of n is.

(Refer Slide Time: 31:56)

Claim $c(t,n) > \frac{t\,n^{1+1/t}}{e} - tn$

Proof Basis

✓ $t=1 \quad c(1,n) = \frac{n(n-1)}{2} > \frac{n^2}{e} - n$ ✓

✓ $t \geq 1 \quad c(t,1) = 0 > \frac{t}{e} - t$

Now, we claim that c t of n is greater than t times n power 1 plus 1 by t divided by e where e is the base of natural logarithm minus t n. This is what our claim. The proof of the claim will be using induction. So, for the basis case, let us consider t equal to 1 which means we have to sort n items in one single step. We can sort n items in one single step only if we perform all the comparisons at one go. Once all the comparisons of performed that is every pair of elements are compared with each with each other, then the rest is only manipulating the comparison data.

We do not have to perform any further comparisons. All possible comparisons have been executed. Therefore, if you want to sort n items in one single step, you will have to necessarily perform n into n minus 1 divided by 2 comparisons. If you omit to compare at least one pair, we can prove that the sorting has not worked correctly by manipulating the values that you place on these two elements. This you can see is n squared by e if you put a t equal to 1 in this expression, you would get n n squared by e minus n. So, you can show that n into n minus 1 by 2 is greater than n squared by e minus 1. Therefore, if t equal to 1, our claim is correct.

For t greater than or equal to 1, let us consider c of t 1 which means you want to sort one element in t steps. If you have only one element to sort, you do not have to perform any

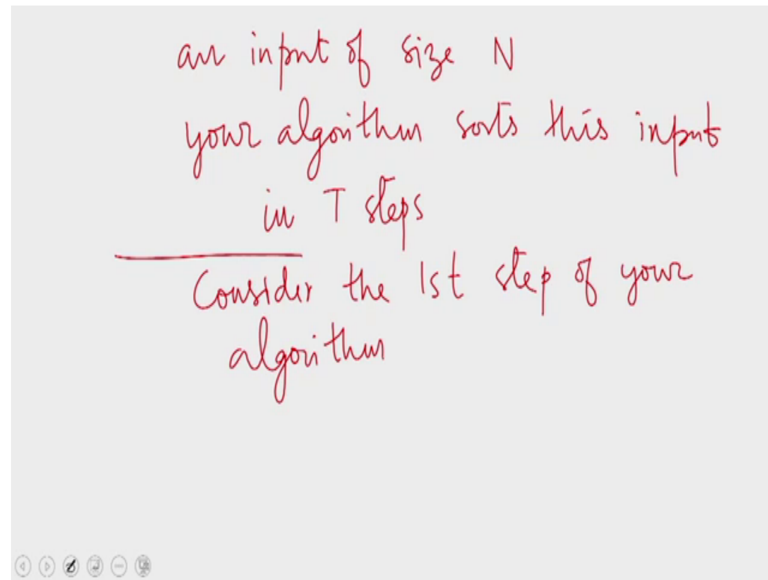
comparison; therefore, c of t 1 equal to 0 for all t greater than or equal to 1. So, if you substitute n equal to 1 in our claim, we find that this is t by e minus t. So, these two will form the basis.

(Refer Slide Time: 34:41)

So, now let us hypothesize that for all small t less than capital T and n small n less than or equal to capital N or small t less than or equal to capital T and small n strictly less than N, it is the case that c of t n is greater than t n power 1 plus 1 by t divided by e minus t n. So, this is what are hypothesis is.

That is we are varying n and t both, so we are considering a particular value of capital N and capital T and what we assume is that for every small n and every small t, that is at least one of them must be less than or equal to n of t which means for all these values for every value within the square the within this rectangle, the hypothesis holds except for n and t. That is what we want to establish. Let us we consider the rectangle of size n by t, then here all nodes except the bottom right node satisfies the hypothesis is what we have assumed.

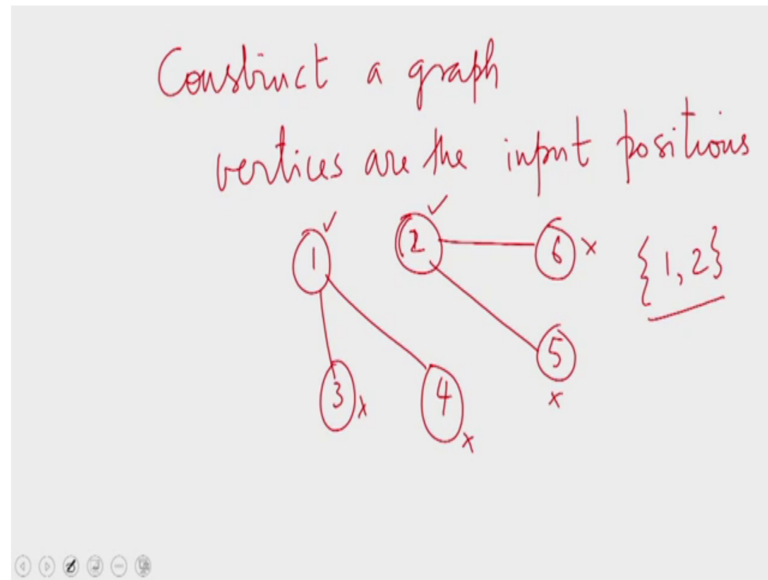
(Refer Slide Time: 36:28)



So, now, let us consider an input of size N and let us assume that an algorithm sorts, let us say your algorithm sorts this input in T steps.

So, this is an adversarial argument. You suggest an algorithm that sorts an input of size N in T steps, capital N and capital T. Then, I claim that this N and T should satisfy the claim. For this purpose, I as an adversary of your algorithm will try to make it work badly. To this end, what I do is this consider the first step of your algorithm. In the first step of your algorithm, the comparisons that are performed would be hard coded into the algorithm. Exactly as we have seen for the sequential algorithms, here again whatever be the input the comparisons that are to be performed in the first step must be hardwired into the algorithm.

(Refer Slide Time: 38:07)



We construct a graph; in this graph, the vertices are the inputs, are the input positions. For example, if n is 6, we take 6 vertices and if your algorithm specifies that in the first step we will have comparisons between 1 and 4, 1 and 3, 2 and 5 and 2 and 6. Then, we would form a graph in this manner. In general, we construct a graph in this manner, we take vertices for all input positions and we place an edge between two vertices if elements at those positions are compared in the first step of your algorithm. So, in this example, elements at positions 1 and 3 are compared in your algorithm and elements at positions 1 and 4 and compared in your algorithm and so on.

(Refer Slide Time: 39:21)

Find a maximal independent set of this graph

IS: a set of vertices no two of which are adjacent

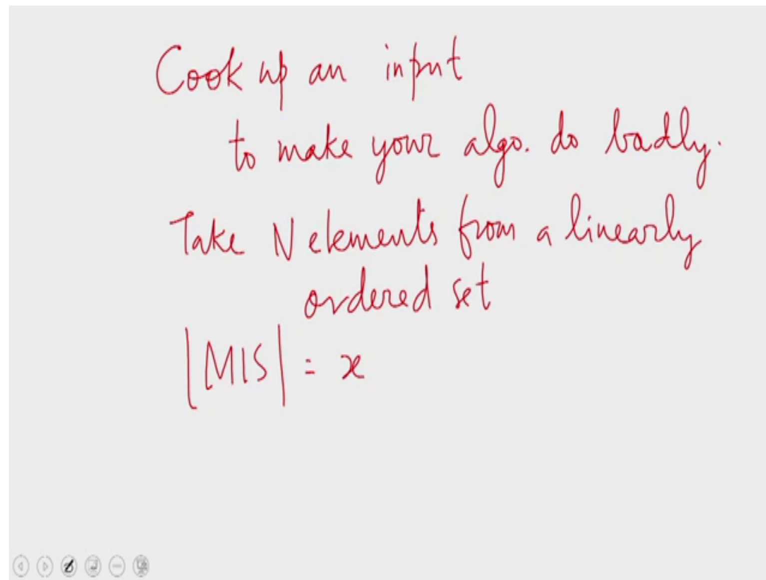
MIS: an IS that is maximal

So, after forming this graph, we find a maximal independent set of your graph. Remember, this graph is a function of your algorithm. It is not dependent on the given input.

Because the pair of elements that are to be compared in the first step, this already hardwired in your algorithm. Therefore, this graph is a function of your algorithm alone. Then, we find a maximal independent set of this graph. You know what a maximal independent set is; an independent set is a set of vertices no two of which are adjacent. A maximal independent set is an independent set of an independent set that is maximal. It is maximal in the sense that, you cannot add one more vertex to it without violating it is independence requirement. That is for every vertex that is not in the independent set, that vertex is adjacent to some vertex in the independent set.

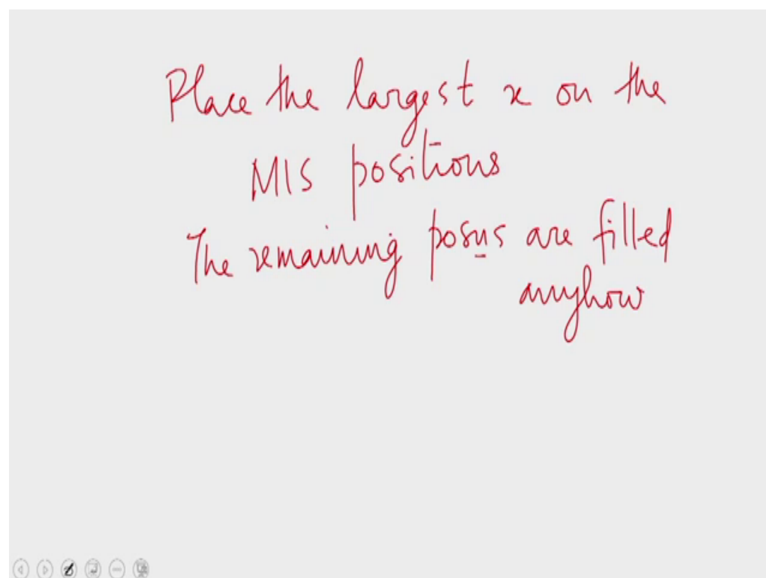
So, in this example, 1 and 2 will form a maximal independent set because the other vertices are all disqualified now from adding to the maximal independence set, maximal independent set. For example, if we were to add 3 to the maximal independent set, then there would be a conflict between 1 and 3. So, 3 cannot be added. Similarly, 4 also cannot be added. Therefore, this graph has a maximal independent set; 1, 2 is a maximal independent set of this graph. Of course, there could be multiple maximal independent sets for a given graph. So, what we have done is to construct this graph based on the algorithm that you have given and then, we have formed a maximal independent set of this graph.

(Refer Slide Time: 41:45)



After this, let me cook up an input to make your algorithm perform badly. What we do is this. Take N elements from a linearly ordered set. Let us say the size of the maximal independent set of the graph that we have found this x, that is we have taken we have drawn a graph and we have found a maximal independent set in it, let us say the size of the maximal independent set is x.

(Refer Slide Time: 42:50

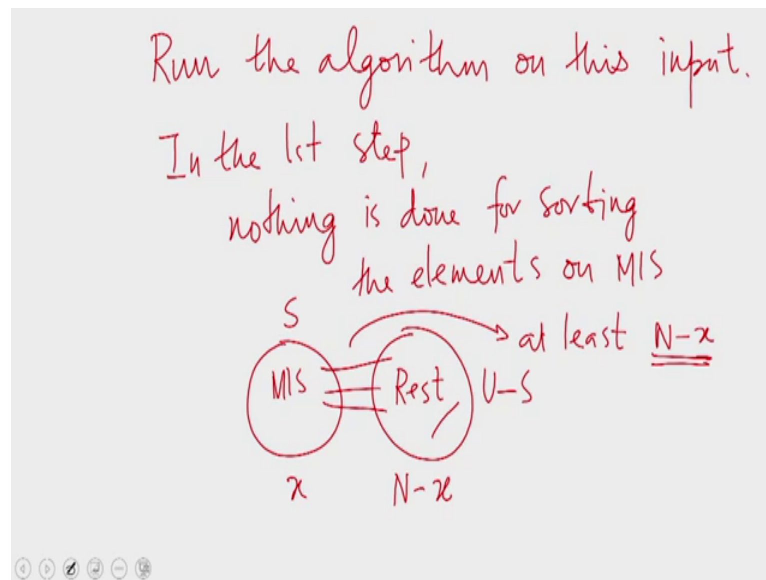


Then, of the n elements that we have taken, place the largest x on the M I S positions. The remaining positions are filled anyhow. What this means is that in our example, the

maximal independent set consists of vertices 1 and 2 and we take an input of size 6, some 6 elements that are linearly ordered and then we place the largest two elements at positions 1 and 2. The remaining four elements are placed at positions 3, 4, 5, 6.

So, in general, what would this ensure? On this input, now we run your algorithm.

(Refer Slide Time: 43:55)



Then, what we find is that in the first step, your algorithm does not do for sorting the elements that are placed on the maximal independence set. Why is that because the elements that are placed on the maximal independence set are not being compared in the first step. The maximal independent set in particular is an independent set. Therefore, no two vertices belonging to it are adjacent to each other. What it means is that, these two are not being compared in the first step. Therefore, these x elements will not be compared with each other.

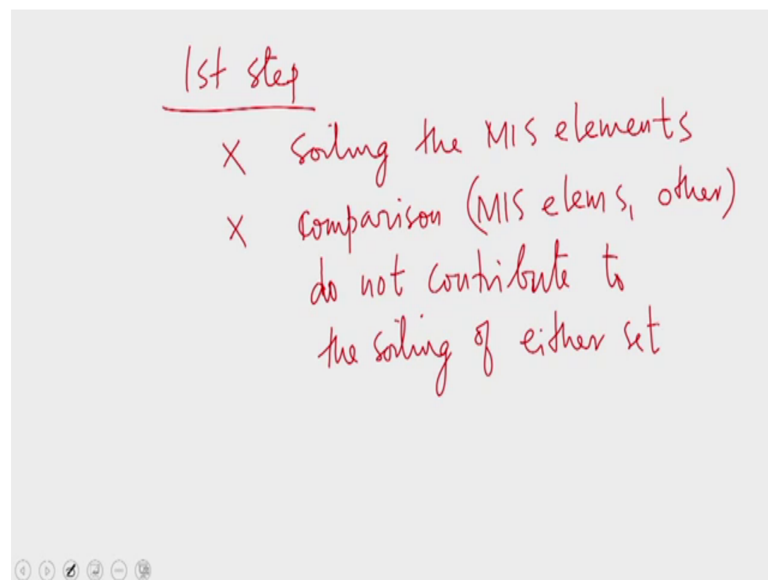
Since, we are not comparing these x elements with each other, what it essentially means is that your algorithm is not doing anything towards sorting these x elements. The other comparisons that are being performed will be between these x elements. So, here we have a maximal independent set of size x and here we have the rest of the vertices which are n minus x in number and then all the edges in the graph are either within the rest or between the maximal independent set and the rest. So, every edges either of this form or of this form, there is no edge that is within the maximal independent set. That is two

vertices of the maximal independent set cannot be adjacent to each other, so every edges of this form.

Now, how many such edges could there be? The number of edges between the maximal independent set vertices in the rest should be at least n minus x. Why is that? A vertex that is not in the maximal independent set should be adjacent to at least one vertex in the maximal independent set. If this is not the case, then we would be able to add that to the maximal independent set and the maximal independent set would no longer be maximal. So, the number of edges between the maximal independent set and the rest of the graph is at least n minus x. So, in the first step, the algorithm performs these many comparisons, these n minus x comparisons but these comparisons are all between the largest elements in the set and the rest.

But, we have purposefully kept the largest elements at these x positions. Therefore, all these comparisons are actually not doing anything towards sorting the elements that are in the maximal independent set or sorting the rest of the elements. So, you can say that all these elements are in that sense wasted.

(Refer Slide Time: 47:14)

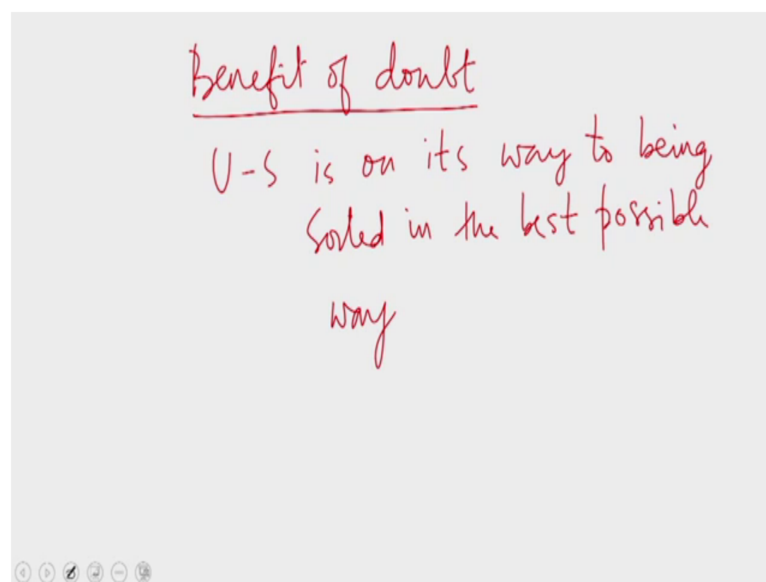


Now, we have ensured two things. The first step of your algorithm has done nothing towards sorting the M I S elements and the comparisons between the M I S elements and the other elements do not contribute to the sorting of either set. But then, we can give a

benefit of doubt to the algorithm. We can assume that all the comparisons that the algorithm has performed within this set.

If I call this S and this as U minus S where U is the set of all the vertices, then any comparison that you perform within U minus S, you can assume is contributing towards the sorting of U minus S. Therefore, now you can assume that U minus S is already on it is way to being sorted. So, you can assume that the algorithm has made the best possible progress towards sorting of U minus S.

(Refer Slide Time: 49:01)



So, this is the benefit of doubt I am giving to your algorithm in the best possible way. So, with these assumptions, we are now able to write a recurrence relation.

(Refer Slide Time: 49:38)

$$c(N,T) \geq c(T, N-x) + (N-x) + c(T-1, x)$$

Sort U, $|U| = N$

S, U – S

We can say that the cost of sorting N elements in T steps, this greater than or equal to the cost of sorting N minus x elements in T steps. This is the cost of sorting the set U minus S. In particular, when you want to sort U, you have to sort S as well as U minus S.

But then, the algorithm has been given a benefit of doubt and what we have assumed is that the algorithm is on course to sorting the elements of U minus S and T steps in the best possible way. Therefore, the total number of comparisons to the algorithm will perform in sorting these N minus x elements would be c T N minus x. The algorithm has wasted at least N minus x comparisons by comparing elements of the maximal independent set with the rest. Now, when we consider the elements of the set S, we find that no comparison has been performed between these two elements and we have already lost one step. So, if the algorithm has to finish sorting the entire set in T steps, then this one particular set will have to be sorted in T minus 1 steps and there are x elements in it.

So, this would be the recurrence relation that governs c N T. Once again, the recurrence relation is argued in this fashion. We have to sort the set of size N, let this set be called U. So, we have to sort the set U. We have separated the set U into two sets; S and U minus S. S is the maximal independent set and U minus is the set of all the remaining vertices. Now, the algorithm has been given the benefit of doubt. What we have assumed is that it has done the best it could towards sorting U minus S. So, this algorithm is on close to sorting U minus S in T steps. Therefore, the total number of comparisons that the

algorithm will be performing within U minus S would be c T N minus x. The size of U minus S is N minus x and this is going to be sorted in T steps in the best possible way.

So, the total number of comparisons that the algorithm will perform within U minus S is c T N minus x. Then, the algorithm spends N minus x comparisons between S and U minus S. So, that is what the second term is. So, you can say that this is a set of wasted comparisons because they do not contribute towards sorting either U or U minus S. And then finally, when we look at the set S, we find that the first step has not done anything towards sorting S.

It has already wasted one step. Now, when the algorithm is over, S will have to be in sorted order. So, S will have to be now sorted in T minus 1 steps. So, the number of comparisons, we will have to perform will be c T minus 1 x. So, this is the recurrence relation that governs the number of comparisons that the algorithm must necessarily perform to sort n elements in T steps.

Now, using the hypothesis, we have assumed that the claim is correct for all ordered pairs where at least one of the components is smaller than the components here. That is either the first component is less than N or the second component is less than T. That is the case with both of these. So, we can substitute the hypothesis for both these terms.

(Refer Slide Time: 53:38)

$$c(T,N) > T\left[\frac{(N-x)^{1+1/T}}{e} - (N-x)\right] + (N-x) + (T-1)\left[\frac{x^{1+1/(T-1)}}{e} - x\right]$$

What we find is that c T N is greater than T times N minus x power 1 plus 1 by T by e minus N minus x plus N minus x and then applying the hypothesis for the third step. Simplifying this expression, we will be able to prove our claim. The rest of the proof, we shall see in the next lecture; so that is it from this lecture. Hope to see you in the next lecture.

Thank you.