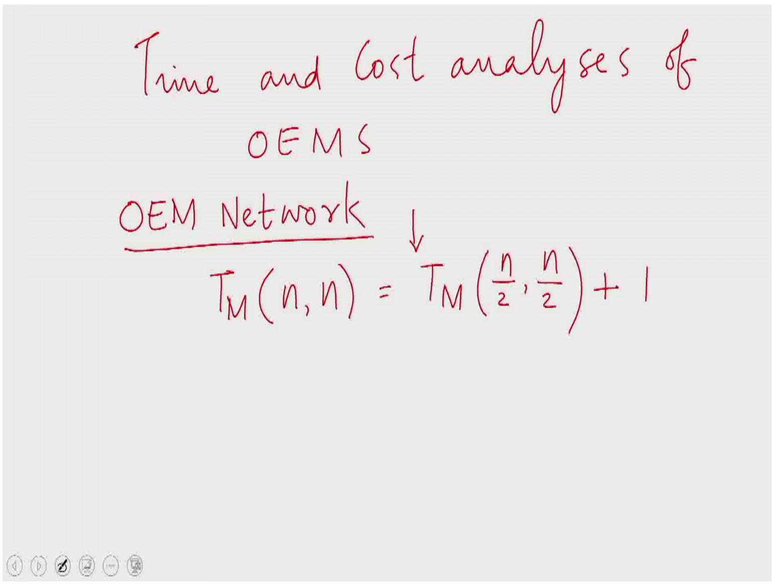
Parallel Algorithms

Prof. Sajith Gopalan
Department of Computer Science & Engineering
Indian Institute of Technology, Guwahati

Lecture - 11
OEMS, Bitonic-Sort-Merge Sort (BSMS)

Welcome to the 11th lecture of the MOOC on Parallel Algorithms. In the 10th lecture of the MOOC we studied odd even merge sort using comparator network. We had proved the correctness of the network. Now it remains to do the analysis for the time complexity as well as the cost of the network.

(Refer Slide Time: 00:52)



Time and Cost analyses of OEMS

OEM Network

$$\downarrow$$

$$T_M(n,n) = T_M\left(\frac{n}{2},\frac{n}{2}\right) + 1$$

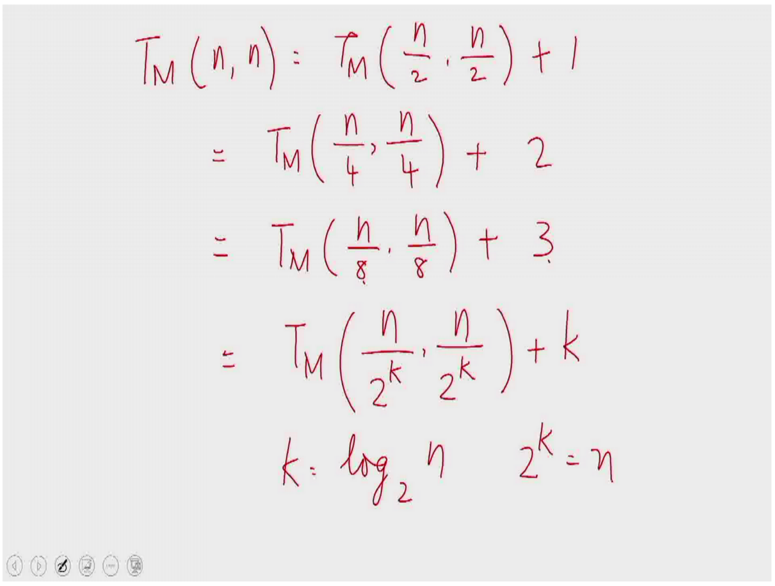
So, today we do the time and cost analysis of the odd even merge sort algorithm. First let us begin with the basic building block which is the odd even merge network. In the odd even merge algorithm, when we are given two arrays to merge the two sorted arrays to be merged; what we do is to take the odd elements to one side and the even elements to the other side.

And then merge the odd instance and the even instance recursively interleave the outputs. And then perform a comparison between pairs obtained by leaving out the first element in the last element. So, the time complexity of a merging two arrays of size n each. Let me denote that using this expression TM of n n. This is the time complexity of the odd even merge network that merges two sorted arrays of size n each.

This would be the time taken by the recursive curve plus 1 single time unit. So, as I said when we a given two sorted arrays of size and each to merge them what we do is to take. The odd elements to one side and the even elements to the other side the odd side and the even side are solved recursively and simultaneously. Since they are solved simultaneously when we analyze the time we have to account only for one that is why there is no multiplicative factor here.

So, there is only one instance of a problem of size half after that is done, that is after the recursive calls are over we interleave the outputs. And then leave out the first one and the last one and compare and exchange the remaining pairs. All these comparison and exchanges are taking place simultaneously therefore, they together will take only unit time. Therefore, the recursive the recurrence relation is as shown here.

(Refer Slide Time: 03:20)

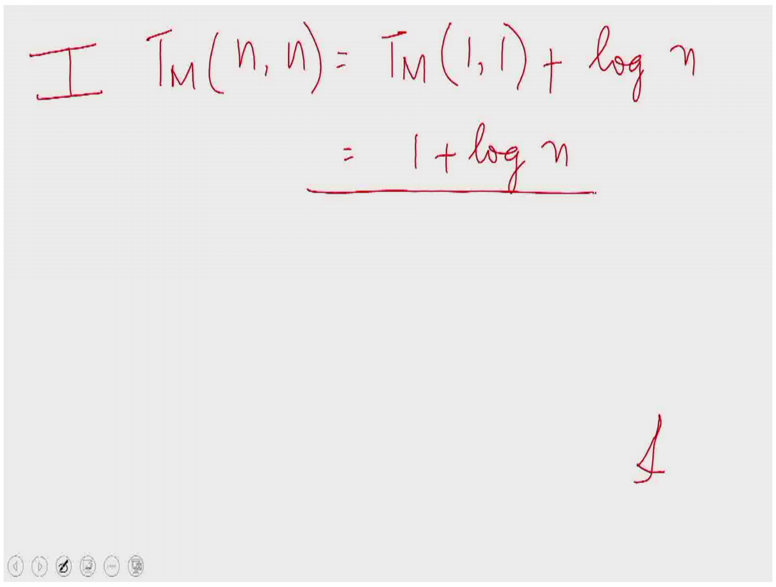


$$T_M(n,n) = T_M\left(\frac{n}{2}, \frac{n}{2}\right) + 1$$
$$= T_M\left(\frac{n}{4}, \frac{n}{4}\right) + 2$$
$$= T_M\left(\frac{n}{8}, \frac{n}{8}\right) + 3$$
$$= T_M\left(\frac{n}{2^k}, \frac{n}{2^k}\right) + k$$
$$k = \log_2 n \qquad 2^k = n$$

Now, let us try to solve this recurrence relation. If we unroll the recurrence relation once, we find that this is nothing, but which means the additive term here is 2. If we unroll this once again we get.

So, now you can see the pattern we have 8 here and we have 3 here and 8 is 2 power 3. Therefore, if we unroll this k times we would get. So, after unrolling k times this is what we get the recurrence relation becomes like this. If I substitute k equal to log into the base 2 then 2 power k equal to n n by n becomes 1.

(Refer Slide Time: 04:58)

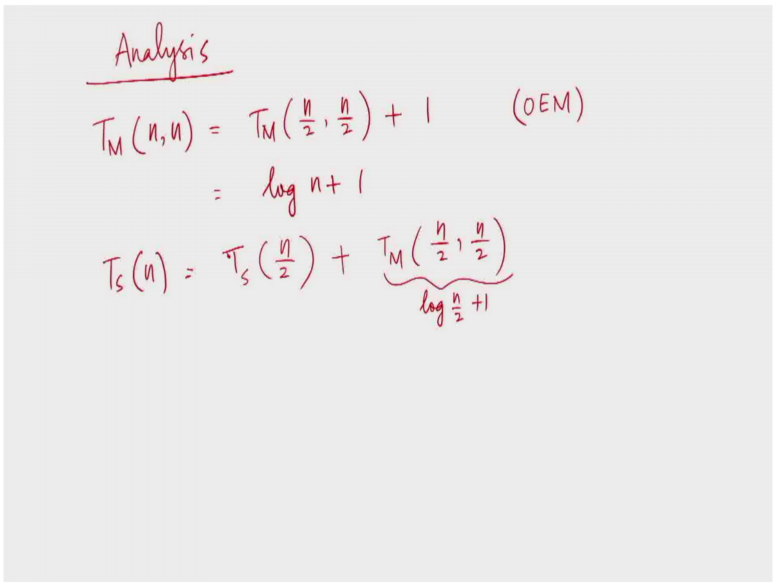


$$\text{I} \quad T_M(n,n) = T_M(1,1) + \log n$$
$$= 1 + \log n$$

Therefore the recurrence relation becomes here we have TM of 1 1 plus the additive term is k which is log into the base 2. But then the cost of merging two arrays of size 1 each it is the time complexity of the basis case which is this odd even merge network.

The time complexity of that is order 1 so, this is 1 plus log n. This is the time complexity therefore, of the odd even merge network, an odd even merge network that merges two arrays of size n each will run in log n plus 1 steps.

(Refer Slide Time: 06:02)



Analysis

$$T_M(n,n) = T_M\left(\frac{n}{2},\frac{n}{2}\right) + 1 \qquad (\text{OEM})$$
$$= \log n + 1$$
$$T_S(n) = T_S\left(\frac{n}{2}\right) + \underbrace{T_M\left(\frac{n}{2},\frac{n}{2}\right)}_{\log\frac{n}{2}+1}$$

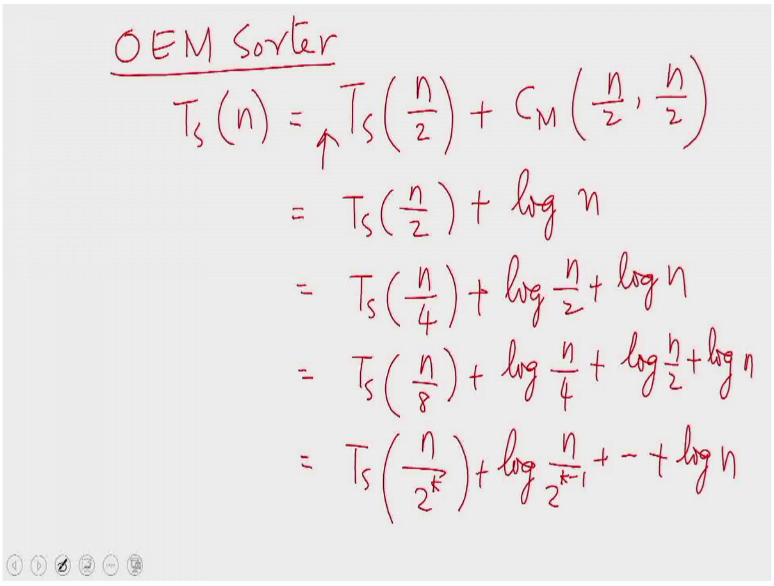
Now, let us move on to the time complexity of the odd even merge sort. To sort n elements let us say the time complexity is TS of n what the algorithm does is to split the array into two halves two equal halves assuming that n is a power of 2. We divide the array into two equal halves we sought each half recursively and then merge the two sorted arrays.

Since the two halves are sorted simultaneously in the recurrence relation we have to count only ones. Therefore, there is no multiplicative factor here TS of n by 2 plus we have now to merge two sorted arrays of size n by two each. So, this is the recurrence relation that governs the time complexity of an odd even merge sort for an input size of n.

But this can be written as this we already know is log of n by 2 plus 1, but log of n by 2 plus 1 is the same as log n. Therefore, I can write this as log n. So, the recurrence relation now becomes TS of n is equal to TS of n by 2 plus log n.

So, if you unroll the recurrence relation. We find that this is TS of n by 4 plus log of n by 2 plus log n. If we unrolled once again we have TS of n by 8 plus log of n by 4 plus log of n by 2 plus log n.

(Refer Slide Time: 07:36)

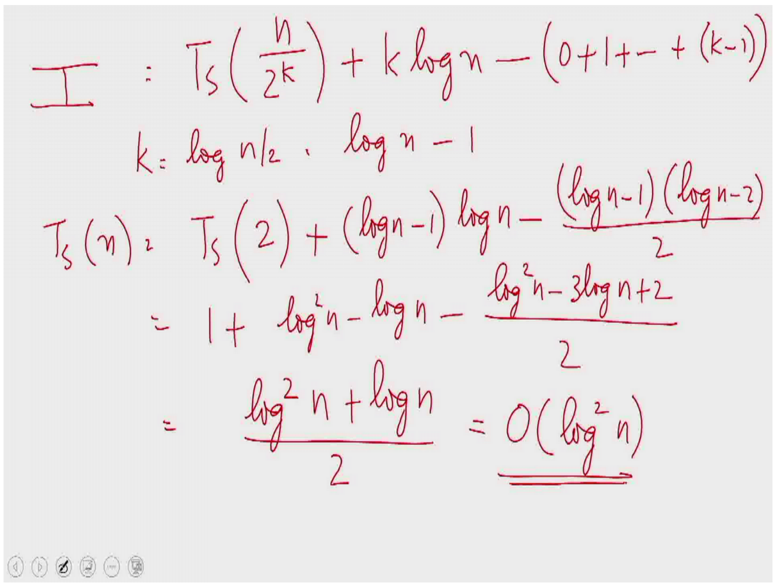


OEM Sorter

$$
\begin{aligned}
T_s(n) &= T_s\left(\frac{n}{2}\right) + C_M\left(\frac{n}{2}, \frac{n}{2}\right) \\
&= T_s\left(\frac{n}{2}\right) + \log n \\
&= T_s\left(\frac{n}{4}\right) + \log \frac{n}{2} + \log n \\
&= T_s\left(\frac{n}{8}\right) + \log \frac{n}{4} + \log \frac{n}{2} + \log n \\
&= T_s\left(\frac{n}{2^k}\right) + \log \frac{n}{2^{k-1}} + \cdots + \log n
\end{aligned}
$$

So, if we continue like this and unroll the recurrence relation k times, we will have the expression would look like this. Now, let us see how to simplify this.

(Refer Slide Time: 08:21)



$$I \;:\; T_S\left(\frac{n}{2^k}\right) + k\log n - \left(0+1+\ldots+(k-1)\right)$$

$$k = \log n/2 = \log n - 1$$

$$T_S(n) = T_S(2) + (\log n - 1)\log n - \frac{(\log n-1)(\log n-2)}{2}$$

$$= 1 + \log^2 n - \log n - \frac{\log^2 n - 3\log n + 2}{2}$$

$$= \frac{\log^2 n + \log n}{2} = \underline{\underline{O(\log^2 n)}}$$

This is the same as TS of n by 2 power k and here we have logarithms of log n through log n power log n log of n by 2 power k minus 1. So, in this case the denominator of n is 2 power 0 and in this case the denominator of n is 2 power k minus 1 on the leftmost term. Therefore, the number of terms here is k. So, if we split this we will be able to write this as k log n taking all the numerators.

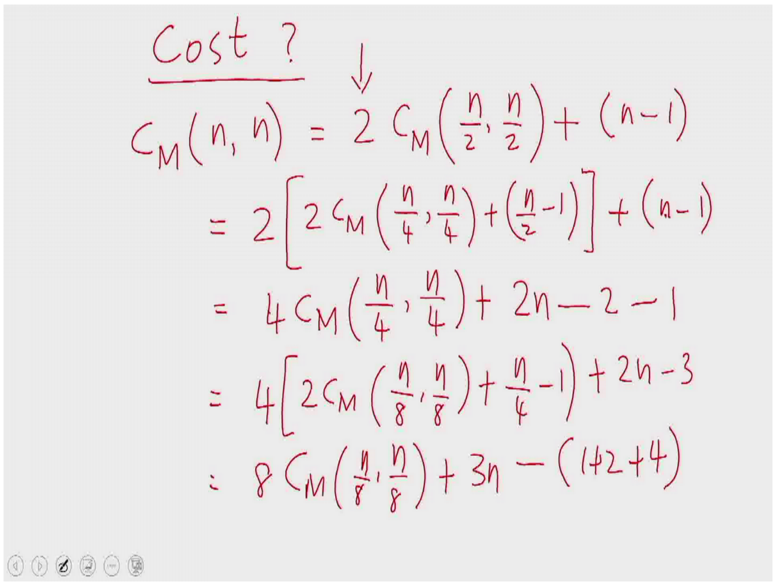
And then we have the logarithms of the denominators summing to all the way from 0 to k minus 1. Now, if we substitute k equal to log of n by 2 which is the same as log n minus 1. We find that TS of n is equal to TS of here we have substituted the log of n by 2 for k. Therefore, 2 power k is n by 2 n by n by 2 is 2 therefore, we have TS of 2 plus k is log n minus 1.

So, we have log n minus 1 into log n minus here k minus 1 into k by 2 and k minus 1 is log n minus 1 k minus 1 is log n minus 2 and k is log n minus 1. So, that is what the expression would be. And since we know that TS f 2 the time required to sort two elements is again the cost of the basic two sorter this is the two sorter.

So, this is 1 plus log squared n minus log n minus the log term and simplifying that we would get TS of n equal to log quite n plus log n divided by 2. So, this is the exact time complexity of the odd even merge sort algorithm the algorithm runs in log squared n plus log n by 2 which we can say is order of log square n.

So, the odd even merge sort algorithm sorts n elements in order of log squared n time the time complexity of the algorithm is order of log squared n. But then what is the cost complexity the number of instructions the instructions complexity.

(Refer Slide Time: 11:44)



Cost ?

$$C_M(n,n) = 2\,C_M\left(\frac{n}{2},\frac{n}{2}\right) + (n-1)$$
$$= 2\left[2\,C_M\left(\frac{n}{4},\frac{n}{4}\right) + \left(\frac{n}{2}-1\right)\right] + (n-1)$$
$$= 4\,C_M\left(\frac{n}{4},\frac{n}{4}\right) + 2n - 2 - 1$$
$$= 4\left[2\,C_M\left(\frac{n}{8},\frac{n}{8}\right) + \frac{n}{4} - 1\right) + 2n - 3$$
$$= 8\,C_M\left(\frac{n}{8},\frac{n}{8}\right) + 3n - (1+2+4)$$

Now, let us analyze the circuit for it is cost. So, we should begin with the merge network. So, the cost of merging two sorted arrays of size n each is let us say CM of n n, here we have the odd side and even side to be merged each of size n by 2 each. But then when we estimate the cost we have to count all the comparators used in the network.

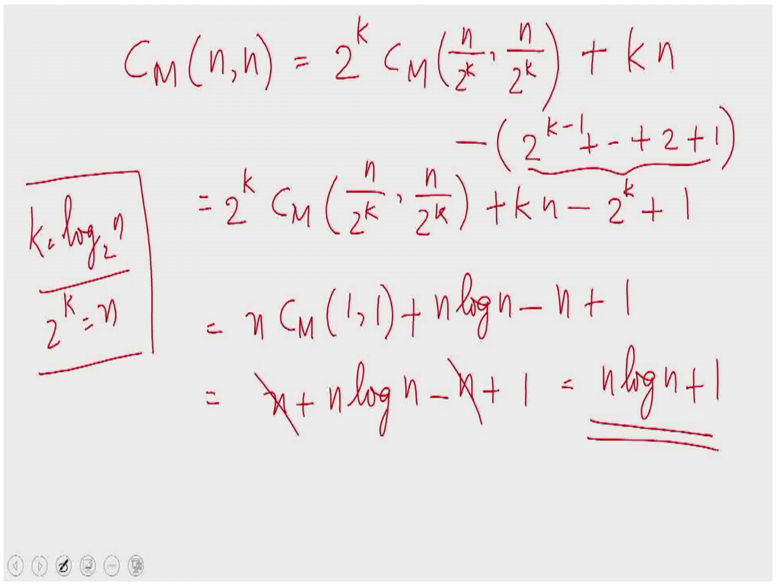
Therefore, in the case of a cost in the case of the cost analysis we do have this factor of 2 which was not present in the time analysis. So, what the algorithm does is to take the odd elements to one side and the even elements to the other side merge the two sides separately. And therefore, the cost of these recursive invocations will be twice of CM of n by 2 n by 2. After that the outputs are interleaved we leave out the first one in the last one and then the remaining pairs there are n minus 1 such pairs.

These pairs are compared and exchanged. Since there are n minus 1 such pairs the cost of performing those compare exchanges is n minus 1. So, this is the recurrence relation that governs the cost of the odd even merge network. So, let us try to estimate what this would be let us try to unroll this. So, applying the recurrence relation on CM of n by 2 n by 2 we find that that is nothing, but twice CM of n by 4 n by 4 plus n by 2 minus 1.

And then of course, the original additive term and minus 1, taking this to inside we have 4 CM n by 4 n by 4 plus these two multiplies to n by 2 to give n. And that plus the original n will give us 2 n and then we have minus 2 and minus 1. So, this is what we get after 1 unrolling. If we unroll again we have 2 CM n by 8 n by 8 plus n by 4 minus 1 plus the previous terms 2 n minus 3.

This would be 8 times CM of n by 8 n by 8 and then 4 times n by 4 is n that plus 2 n is 3 n. And then we have another minus 1 so, this in fact, is 1 plus 2 plus 4, 4 added to the previous 3. So, that tells us the pattern of the recurrence relation.

(Refer Slide Time: 15:17)



$$C_M(n,n) = 2^k C_M\left(\frac{n}{2^k}, \frac{n}{2^k}\right) + kn - (\underbrace{2^{k-1} + \cdots + 2 + 1})$$

$$= 2^k C_M\left(\frac{n}{2^k}, \frac{n}{2^k}\right) + kn - 2^k + 1$$

$$= n C_M(1,1) + n\log n - n + 1$$

$$= \cancel{n} + n\log n - \cancel{n} + 1 = \underline{\underline{n\log n + 1}}$$

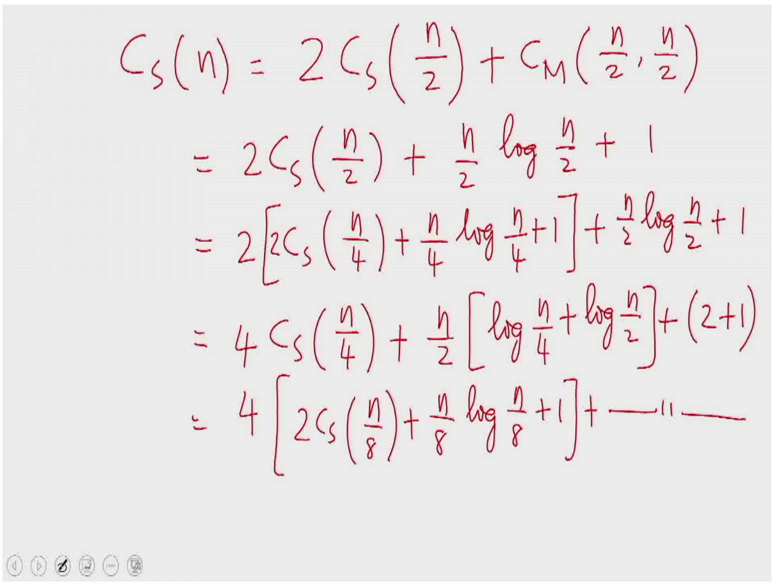
$$\boxed{\begin{array}{c} k = \log_2 n \\ \hline 2^k = n \end{array}}$$

So, if we unroll like this k times what we would get is; this recurrence relation. CM of n n would be 2 power k times CM of n by 2 power k n by 2 power k plus k n. And then there is a subtractive term so, this can be simplified in this fashion this is 2 power k times CM of n by 2 power k n by 2 power k k n minus the sum of all these terms would be 2 power k minus 1.

Now, if you substitute k equal to log n as before log n to the base 2. Then 2 power k equal to n with these substitutions we find that this is n times CM of 1 1 plus n log n minus n plus 1. But then CM of 1 1 is 1 the cost of merging two arrays of size 1 each is 1. Therefore, we have n plus n log n minus n plus 1 with these n s cancelling we have n log n plus 1.

So, this is the cost of the odd even merge algorithm. So, as you can see when you look at the merge algorithm non isolation we find that it runs in n log n plus 1 comparisons. But the time taken is log n plus 1. So, it is an order log n time algorithm with a cost of order of n log n. So, it is not an optimal algorithm when we simulate this on an EREW PRAM.

(Refer Slide Time: 17:47)



$$
\begin{aligned}
C_s(n) &= 2C_s\left(\frac{n}{2}\right) + C_M\left(\frac{n}{2}, \frac{n}{2}\right) \\
&= 2C_s\left(\frac{n}{2}\right) + \frac{n}{2}\log\frac{n}{2} + 1 \\
&= 2\left[2C_s\left(\frac{n}{4}\right) + \frac{n}{4}\log\frac{n}{4} + 1\right] + \frac{n}{2}\log\frac{n}{2} + 1 \\
&= 4C_s\left(\frac{n}{4}\right) + \frac{n}{2}\left[\log\frac{n}{4} + \log\frac{n}{2}\right] + (2+1) \\
&= 4\left[2C_s\left(\frac{n}{8}\right) + \frac{n}{8}\log\frac{n}{8} + 1\right] + \text{---}\,''\,\text{---}
\end{aligned}
$$

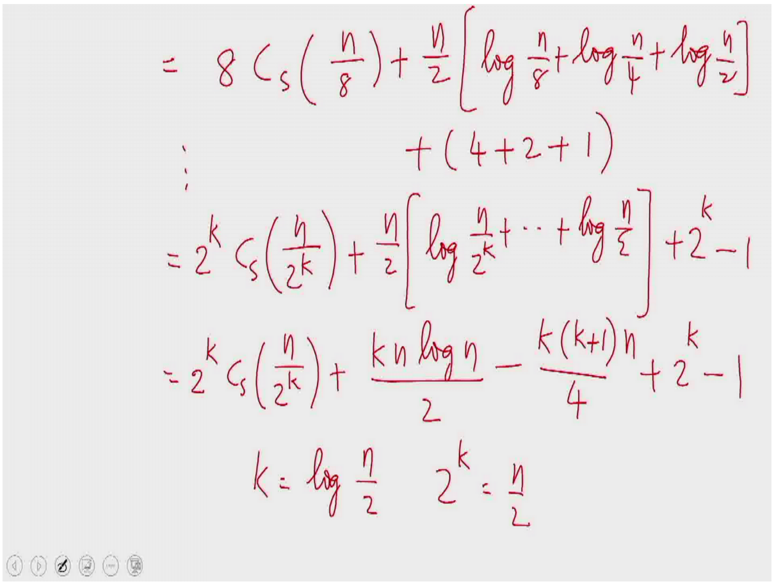
Now, let us see what will be the cost of the sorting network which we denote as CS of n to sort n items what we do is to divide the array into two equal parts sort each half recursively. But when we consider cost the cost of each half will count so, the recursive calls will contribute a total of twice CS n by 2.

Then we have to merge the two halves which will incur a cost of CM of n by 2 n by 2. We already know what CM of n by 2 n by 2 is. So, substituting that in our expression we get 2 CS of n by 2 plus n by 2 log n by 2 plus 1. Now, let us try unrolling again.

This would be twice CS of n by 4 plus n by 4 log n by 4 plus 1 all that is within the brackets. Then outside we have the terms n by 2 log n by 2 plus 1. So, simplifying we find that this is the same as 4 times CS of n by 4 plus n by 2 times log of n by 4 plus log of n by 2.

Clubbing the n by 2 terms together the two outside multiplied with n by 4 gives us n by 2. So, that is n by 2 log n by 4 and we already had n by 2 log n by 2 before. So, these two together will come to this and then plus we have 2 and 1. So, if you unroll once again we have 4 times 2 CS n by 8 plus n by 8 log n by 8 plus 1 plus these terms outside.

(Refer Slide Time: 20:31)



$$= 8\,C_s\left(\frac{n}{8}\right) + \frac{n}{2}\left[\log\frac{n}{8} + \log\frac{n}{4} + \log\frac{n}{2}\right] + (4+2+1)$$

$$\vdots$$

$$= 2^k\,C_s\left(\frac{n}{2^k}\right) + \frac{n}{2}\left[\log\frac{n}{2^k} + \cdots + \log\frac{n}{2}\right] + 2^k - 1$$

$$= 2^k\,C_s\left(\frac{n}{2^k}\right) + \frac{k\,n\log n}{2} - \frac{k(k+1)\,n}{4} + 2^k - 1$$

$$k = \log\frac{n}{2} \qquad 2^k = \frac{n}{2}$$

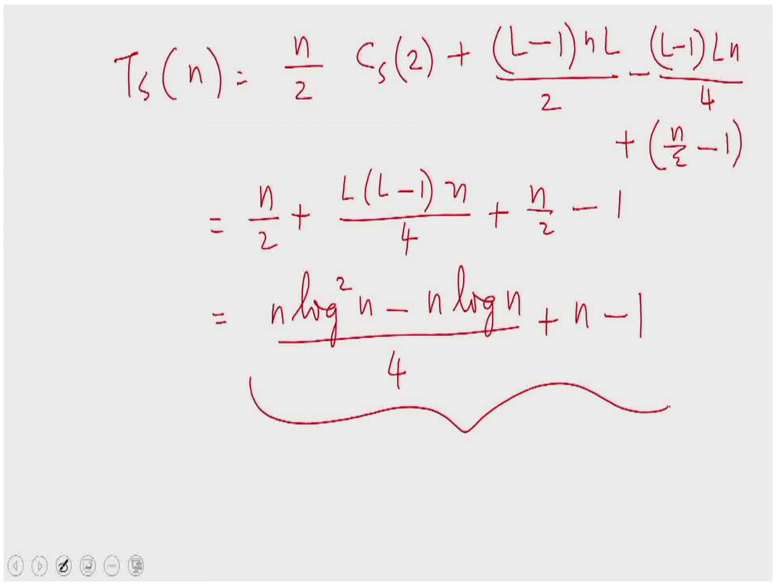
So, that will simplify to 8 times CS of n by 8 plus n by 2 times log of n by 8 plus log of n by 4 plus log of n by 2 plus the additive term 4 plus 2 plus 1. So, if you unroll like this for k steps you will have 2 power k CS of n by 2 power k plus n by 2 times log of n by 2 power k plus log of n by 2 power k minus 1 etcetera.

All the way up to log of n by 2 plus the additive term 2 power k minus 1 etcetera up to 1. So, 1 plus 2 plus 4 plus etcetera 2 power k minus 1 which will add up to 2 power k whole minus 1. So, I can replace that with 2 power k minus 1. This simplifies to 2 power k times CS of n by 2 power k plus k times n log in by 2 out of all these logarithmic terms.

Let us consider only the numerator. So, log of n by 2 power k decomposes into log n minus log of 2 power k which is log n minus k. So, out of each such term you get a log n so there are k such log ns. So, that is k log n that multiplied by n by 2 give us k n log n by 2. Then the remaining would the denominators in this case would give us log 2 is 1, so, 1 from the last term k from the first term.

So, we have 1 plus 2 plus 3 plus 4 plus etcetera up to k. So, that is k into k plus 1 by 2 that 2 in and 2 outside will give us 4 that multiplied by n plus 2 power k minus 1. So, like we did in the case of the time complexity analysis here again I can substitute k equal to log of n by 2 or 2 power k equal to n by 2.

(Refer Slide Time: 23:56)



$$T_S(n) = \frac{n}{2} C_S(2) + \frac{(L-1)nL}{2} - \frac{(L-1)Ln}{4} + \left(\frac{n}{2}-1\right)$$

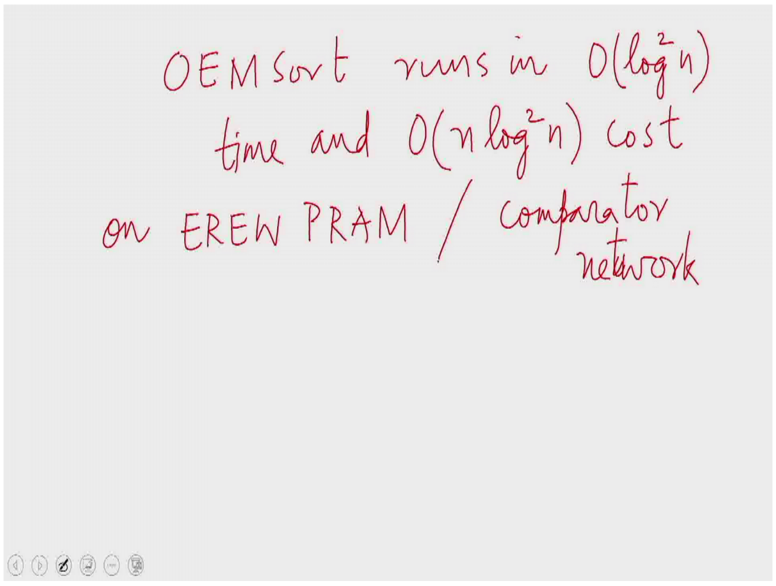
$$= \frac{n}{2} + \frac{L(L-1)n}{4} + \frac{n}{2} - 1$$

$$= \underbrace{\frac{n\log^2 n - n\log n}{4} + n - 1}$$

So, when we do that we get TS of n equals n by 2 times CS of 2 plus. Let me write L for log of n to simplify the expressions which simplifies to n by 2 plus L into L minus 1.

Where L is log n into n divided by 4 plus n by 2 minus 1. So, if you rearrange the terms you find that this is nothing, but n times log squared n minus n log n divided by 4 plus n minus 1. So, this is the cost of the odd even merge sort algorithm.

(Refer Slide Time: 25:19)



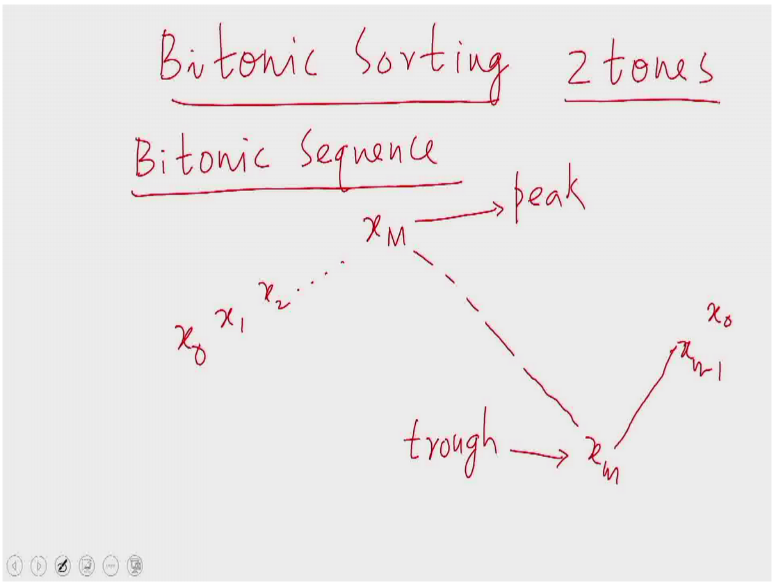
OEM Sort runs in $O(\log^2 n)$ time and $O(n \log^2 n)$ cost on EREW PRAM / comparator network

So, to summarize the odd even merge sort algorithm runs in order of log squared n time and order of n log squared n cost. The dominant term here is n log square n therefore, this is order of n log squared n. Then this can be implemented on and EREW PRAM as well.

That is because every competitor network can be simulated on in EREW PRAM at the same asymptotic constant time so, this will work on EREW PRAM as well. But the algorithm that we presented was on the comparator networks which is of course, a weaker model than an EREW PRAM.

So, we have now an algorithm for sorting which runs in order of log squared n time using n log squared n cost. And this algorithm has very small constant factors as we have seen the constant factor involved in both the cases is about this half and one fourth. Therefore, this algorithm is in practice quite efficient. So, that was the first comparator network sorting algorithm that we have seen.

(Refer Slide Time: 27:01)



Now, we shall see another sorting algorithm on the same comparator model. This is what is called the bitonic sorting network. First we shall define what is called a bitonic sequence? And then we shall see how to sort a bitonic sequence? Then we will convert a bitonic sequence sorter into a network for merging two sorted arrays. And then once we have a network for merging we can use that network for finding an algorithm for merge sort.

And then we get an algorithm we get a network which is very similar to the odd even merge sort in it is complexity. So, a bitonic sequence is defined like this a bitonic sequence of elements drawn from a linearly ordered set. It is like this you given in a sequence x 0 x 1 etcetera in increasing order to indicate their relative values I have written them slanted.

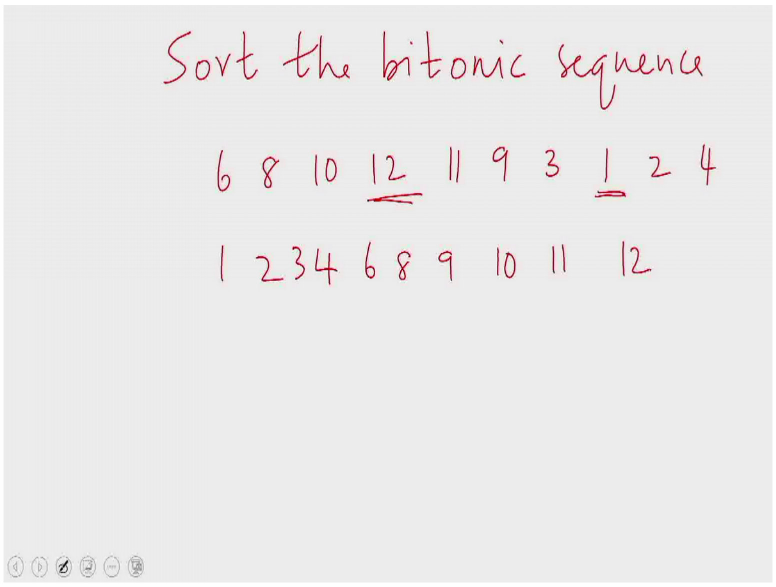
So, here we have the elements increasing from x 0 to x m. And then the values decrease all the way to x small m and then the values increase again to x n minus 1 which happens to be smaller than x naught. So, a bitonic sequence is a sequence of elements drawn from a linearly ordered set. So, that this relationship holds that is x naught is smaller than x 1, x 1 is smaller than x 2 and so on and then x m minus 1 x capital M minus 1 is smaller than x capital M.

So, until then from x naught to x m the values are in non decreasing order. Then from x m the values start decreasing and the values will decrease all the way down to x small m. So, x capital M can be called the peak of the sequence this is the largest element in the sequence and x small m can be called the trough of the sequence which happens to be the smallest element in the sequence.

So, in the bitonic sequence we have elements increasing all the way from x naught to x capital M and then the elements decreasing all the way from x capital M to x small m after that the elements increase once again to x n minus 1 which happens to be smaller than x naught. Therefore what we find is that if you take the two ends and paste them together. That is if you place x n minus 1 just before x naught the sequence will loop around to form a cycle.

So, if you look at the cycle you find that the cycle has 2 tones. In the cycle when you go from capital M to small m you are sliding down. That if you are going from the peak to the trough the values are steadily decreasing. And then once you hit the trough the other tone starts there is now you start climbing up in the values from x small m to x capital M the values are increasing. So, there are two tones in the sequence there is a decreasing tone and an increasing tone. Accordingly there is one peak in one tough such a sequence is called a bitonic sequence.

(Refer Slide Time: 30:56)



Sort the bitonic sequence

6 8 10 12 11 9 3 1 2 4

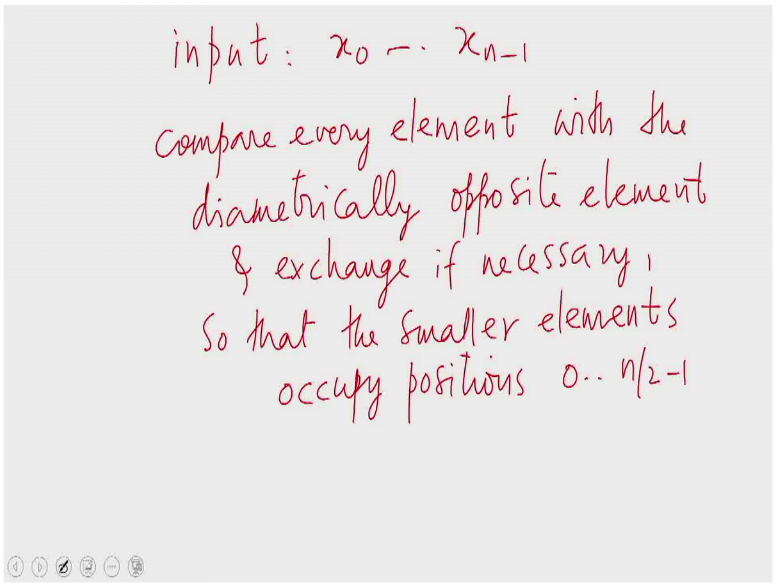
1 2 3 4 6 8 9 10 11 12

And let us say our requirement is this given a bitonic sequence sort the bitonic sequence. For example, we might be given this is a bitonic sequence in this the elements are increasing all the way from 6 to 12, 12 is the peak and then the values decrease all the way from 12 to 1 and 1 is the trough and then the values increase again. But the last value happens to be smaller than the first value.

So, if you paste the sequence end to end you will have 4 coming right before 6. So, we will have a decreasing tone from 12 to 1; 12, 11, 9, 3, 1 is a decreasing tone. And then we have an increasing tone from 1 to 12 1, 2, 4, 6, 8, 10, 12 is an increasing tone, that is why this is called a bitonic sequence.

What we require is to sort the sequence sorting? The sequence would involve producing this output that is 3 here 1 2 3 4 6 8 9 10 11 12 so, this is the sorted order of the bitonic sequence. So, the bitonic sequence sorter will perform this it will take a bitonic sequence and produce the sorted order of the bitonic sequence. Now, let us see an algorithm which will sort the bitonic sequence.

(Refer Slide Time: 32:46)



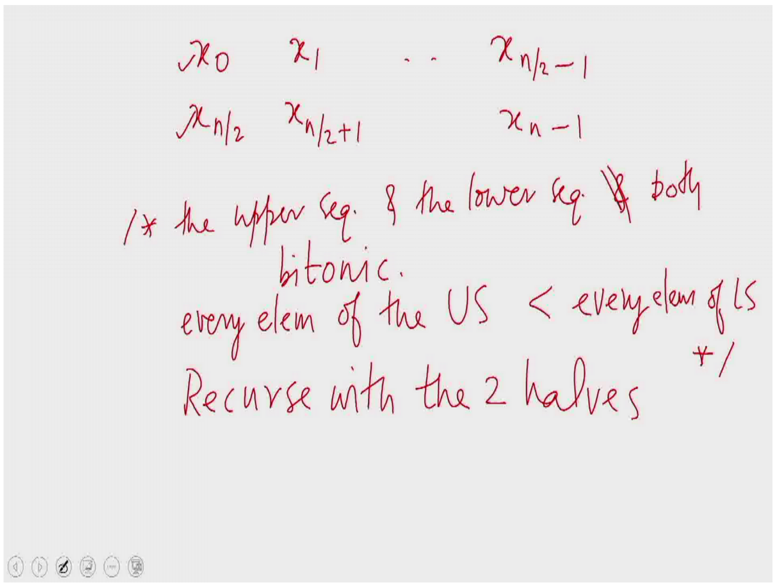
input: $x_0 \ldots x_{n-1}$

Compare every element with the diametrically opposite element & exchange if necessary, So that the smaller elements occupy positions $0 \ldots n/2-1$

So, let us say the input is a bitonic sequence x naught through x n minus 1 any one of them could be the trough and peak. What the algorithm does is this we compare every element with the diametrically opposite element and exchange if necessary. So, that the smaller elements occupy positions 0 to n by 2 minus 1. And this is done in parallel for every single element so, when I say compare an element with the diametrically opposite element. What I assume is that the sequence has been converted into a cyclic one we paste the two ends together so that x n minus 1 becomes a predecessor to x naught.

Then here again we assume that n is a power of 2, you can think of the situation where n is not a power of 2 and imagine how you would solve the problem then n is a power of 2. Then the diametrically opposite element for x naught would be x n by 2 minus 1.

(Refer Slide Time: 34:59)



$x_0 \quad x_1 \quad \ldots \quad x_{n/2-1}$

$x_{n/2} \quad x_{n/2+1} \quad \quad x_{n-1}$

/* the upper seq. & the lower seq. ~~is~~ both bitonic.

every elem of the US < every elem of LS */

Recurse with the 2 halves

So, the diametrically opposite pairings would be like this x naught is paired with x n by 2 minus x by 2 x 1 will be paired with x n by 2 plus 1 and so on. So, x n by 2 minus 1 will be paired with x n minus 1. So, this is how we will pair off the diametrically opposite elements.

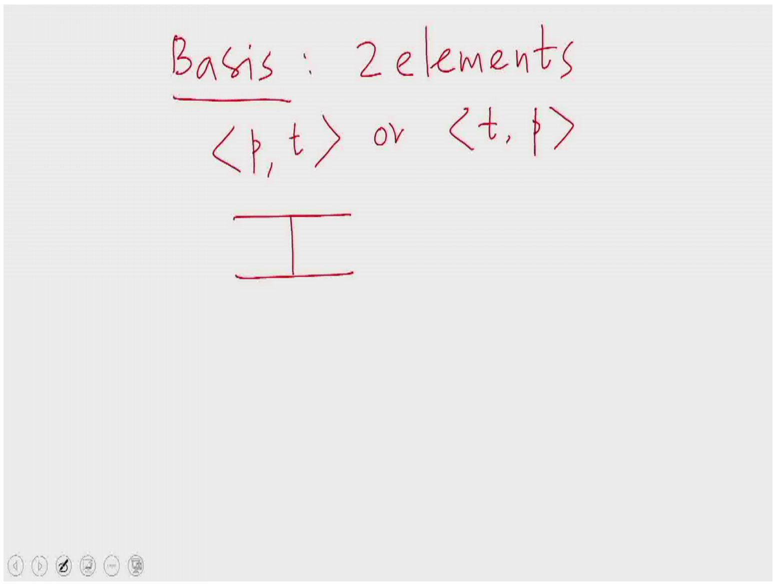
So, what we do is this we compare x naught with x n minus n n by 2. And the smaller of them will occupy this position and the larger of them will be occupying this position that is we perform a compare and exchange between diametrically opposite elements. Then we can claim that the upper sequence and the lower sequence or both bitonic.

And also every element of the upper sequence; every element of the upper sequence is smaller than every element of the lower sequence; which means we have divided the problem of bitonic sorting into two halves quite nicely. So, if the upper half and the lower half are sorted in place then the entire sequence would be sorted.

So, once again what we have done is this when we are given a bitonic sequence of length n we compare every element with the diametrically opposite element and perform an exchange so that the smaller element will occupy the range 0 to n by 2 minus 1. Once this is done we can show that the 2 halves are bitonic sequences of length n by 2 each.

Moreover, every element of the upper sequence that is the elements in the range 0 to n by 2 minus 1 now is smaller than every element in the lower sequence; which means if we sort these bitonic sequences in place then we would have sorted the entire sequence. So, the problem nice nicely divides into halves therefore, we can recurse with the 2 halves.

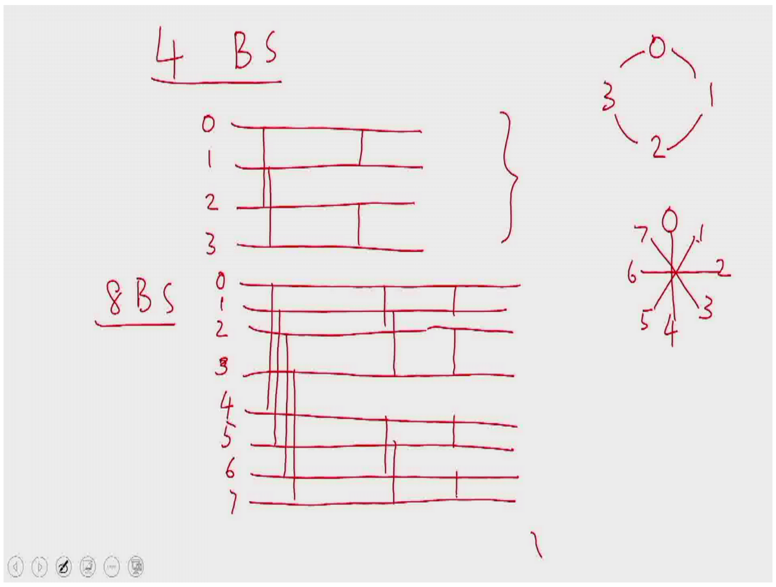
(Refer Slide Time: 37:46)



So, this is what the algorithm is the basis of the algorithm is where we have only two elements. A bitonic sequence has necessarily a peak and a trough and if there are only two elements they are the peak and the trough.

So, the two sequences could be either the peak followed by the trough or the trough followed by the peak. Such a sequence of course, can be sorted using a single comparator so, this is the basis of the network.

(Refer Slide Time: 38:29)



Now, using this basis let us form a 4 sorter a 4 bitonic sorter. So, in the case of a 4 bitonic sorter let us say we are given these 4 input lines. What the algorithm prescribes is that every element should be compared with the diametrically opposite element.

So, if you place them in cyclic order 0 will be opposite to 2 and 1 will be opposite to 3 so, this is the cyclic order. So, we have to compare and exchange 0 and 2 and we also have to compare and exchange 1 and 3. Since these two are exclusive operations they can take place simultaneously. So, what are what the result that we are going to show says is that once this is done the 2 halves will be bitonic sequences themselves.

And the elements in the upper half will all be smaller than the elements in the lower half. Therefore, it is sufficient to sort them in place therefore, we only have to sort the upper half and the lower half now. But when we look at the upper half what we find is a bitonic sequence of length 2. And similarly in the lower half also we have a bitonic sequence of length 2. Therefore, all that we have to do is to sort this half and this half.

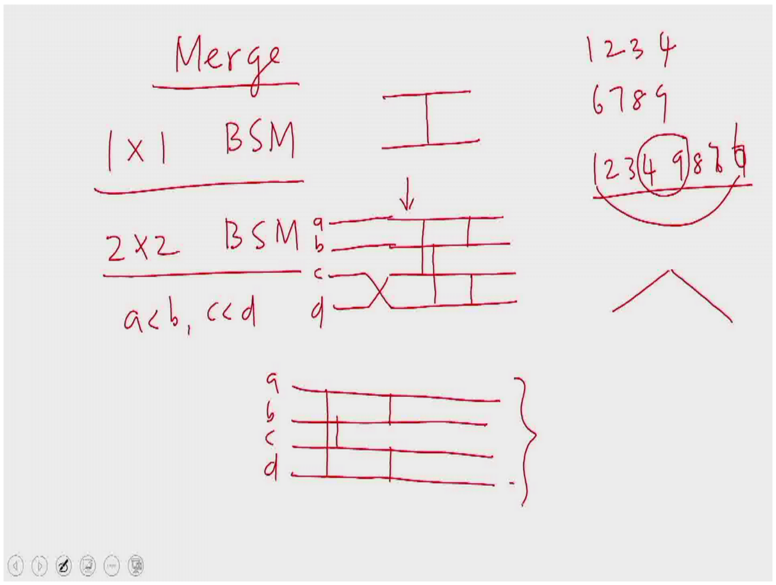
So, this is the network for sorting a bitonic sequence of size 4. Now, we can easily extend this to a bitonic sequence sorter for size 8. So, this is a bitonic sequence sort of a size 8. So, we have the input lines are numbered from 0 to 7. Here we have 0 opposite to 4 2 opposite to 6.

So, this is how the opposite elements go 1 to 5 2 to 6 and 3 to 7. So, we should compare 0 with 4, 1 with 5, 2 with 6, 3 with 7. Then the two halves are bitonic sequences themselves and it is enough to sort them in place. That is because every element of the upper bitonic sequence is smaller than every element of the lower bitonic sequence.

Then it is enough to sort the upper bitonic sequence and the lower bitonic sequence in place. But we already know how to sort a bitonic sequence of size 4 which is using this network. So, this network has to be replicated on the upper side as well as the lower side. So, this is now a bitonic sequence sorter for size 8 so, this is how we construct a bitonic sequence sort. But of course, this is subject to the correctness of our statement.

We made a statement that in a bitonic sequence if every element is compared with a diametrically opposite element and this exchange so that all the smaller elements will be in the upper half and the larger elements will be in the lower half. Then the upper half will form a bitonic sequence and the lower half will also form a bitonic sequence. And that every element in the upper half is smaller than every element in the lower half. Subject to the correctness of this statement we can see that our sorter works correctly. We shall prove the correctness of the statement shortly.

(Refer Slide Time: 43:07)



But first let us see; how this can be used to form a merger. First let us consider a 1 by 1 bitonic sort merger this will form the basis. So, we are given two sorted arrays of size one each these have to be merged. As in all our cases the basis is formed by a single comparator so, this is a 1 by 1 bitonic sort major merger.

Now, let us consider a 2 by 2 bitonic sort merger. So, we are given 4 input lines which is a 2 by 2 which is input word 2 by 2 merger so, we have two sorted arrays of size 2 each. Then what we observe here is this when we are given two sorted arrays. If we paste them back to back what we get is a bitonic sequence.

We do not know which of them will be the peak. The peak could be either the peak of this array or the other array or the trough could be either the trough of this one or the trough of the other one; here it is 7 and 6 so, the trough could be either 1 and 6. So, in general when we a given one sorted array and another sorted array you can turn the second sorted array around and paste them back to back to form a single bitonic sequence.

And once we have a bitonic sequence we can use a bitonic sequence sorter to merge the two arrays. So, that is a technique we are going to use here. So, we given two sorted arrays of size 2 each in the lower half let us say we twist the lines whereas, in the upper half we do not twist the lines. So, if we have the sequence a b c d here; a b and c d are sorted sequences, that is a less than b and c less than d.

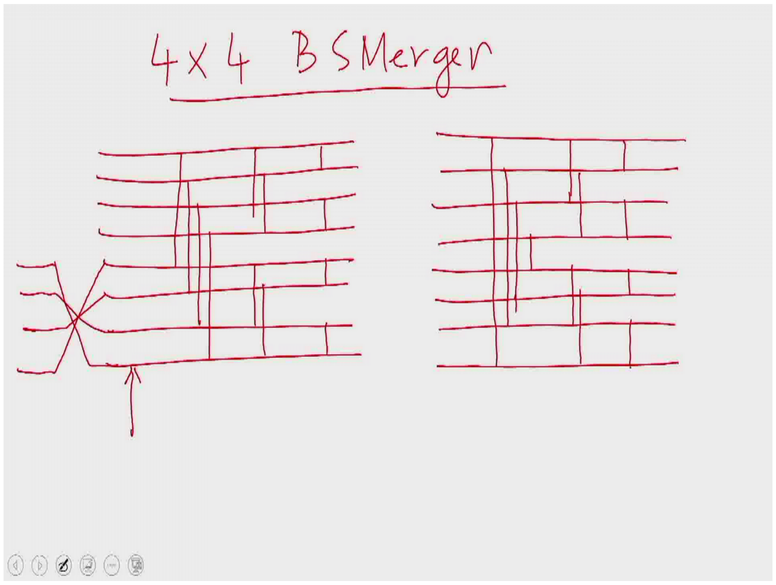
What we do is to invert the sequence c d to form d c and then paste d c to the end of a b so we have a b d c which means at this point what we have a bitonic sequence. So, the bitonic sequence can be sorted using a 4 bitonic sorter which will produce exactly the output that we want the output that we want is the merge of the two sequences a b and c d.

Therefore, we place the 4 bitonic sorter here this is what we want so, this is what a 2 by 2 bitonic sort mergers. But then we would want to draw this network without twisting the input lines which has been our convention. Therefore, if we avoid twisting the lines then the comparators will have to be adjusted here we find that we have comparisons between a d and b c.

Therefore, the first step of comparison will be in those fashion between a d and b c. Whereas, the remaining comparisons will be exactly as they were that would not make a difference because those comparisons were between a b and d c. Now d and c have been turned around, but that does not make a difference still that comparison is between c and d.

Therefore, this will be the 2 by 2 bitonic sort merger network when we avoid twisting the input lines. So, this is the circuit that we will be going we will be using henceforth.

(Refer Slide Time: 47:27)



So, going forward to a 4 by 4 bitonic sort merger we have 8 inputs line 8 input lines now, two sorted sequences of size 4 each. If you twist the input lines here if the lower half is twisted then what we get is a bitonic sequence.

Once again what we have are two sorted arrays of size for each these two have to be merged together. What we do is to invert the lower sequence and then paste the lower sequence to the upper sequence then what we get is a bitonic sequence. Therefore, once the input lines are twisted in this fashion at this point we have a bitonic sequence of size 8.

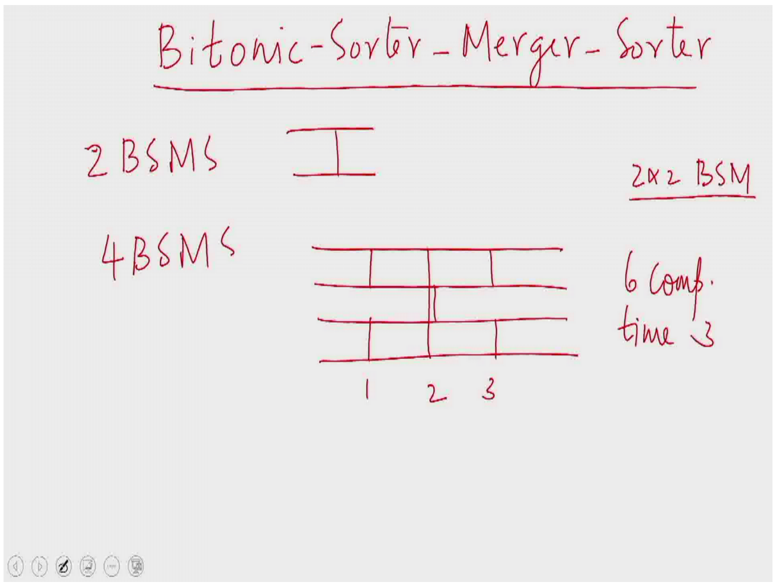
Therefore, to get the merged output we can now use a bitonic sorter of size 8 which we already know is like this. So, this is the 4 by 4 merger using a bitonic sorter. But in this case we have twisted the lower half, but if we want to avoid twisting of the lower half we will have to draw the network differently.

So, the network now looks like this to account for the twisting of the lines in the previous figure goes up to this point ok, then the rest of the network is identical. So, this is what a

4 by 4 bitonic sort mergers. So, we can construct 2 power k by 2 power k bitonic sort merges like this.

And as we said before once we have a merger we can find a sort of. So, to do a recap what we have done is this first we designed a network for sorting bitonic sequences. Then we use these bitonic sequence sorters to form mergers, once we have merges we can design general sorters.

(Refer Slide Time: 51:02)



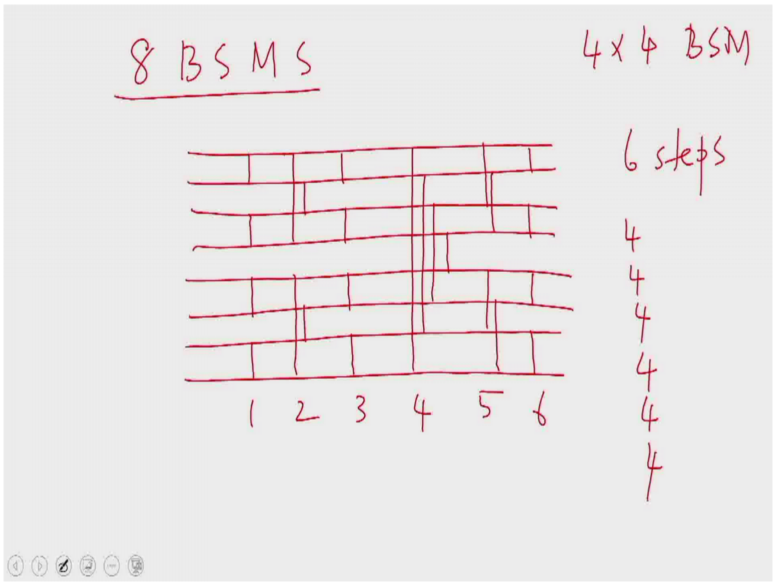
Bitonic-Sorter-Merger-Sorter

So, a general sorting network I would call this a bitonic sorter based merger based sorter. We use a bitonic sort of the form of merger and we use that merger to form a general sorter. So, bitonic sort merge sorter for two elements once again is our same old basis a single comparator.

When we have 4 elements to sort using the classic divide and conquer technique. We will divide it into 2 halves and then sort each half. So, we sort the upper half and the lower half in the upper half we have two elements they can be sorted using single comparators in this fashion. So, the upper half is sorted and the lower half is sorted then what we do is to merge the two lower halves.

But in this case for merging the two lower halves we will use 2 by 2 bitonic sort merger and we know how a 2 by 2 bitonic sort merger looks like. So, the result is that we managed to sort four elements in three steps. And the number of comparators used is 1, 2, 3, 4, 5, 6 six comparators and 3 time units. So, that is what a four bitonic sort merge sort arrays.

(Refer Slide Time: 53:00)



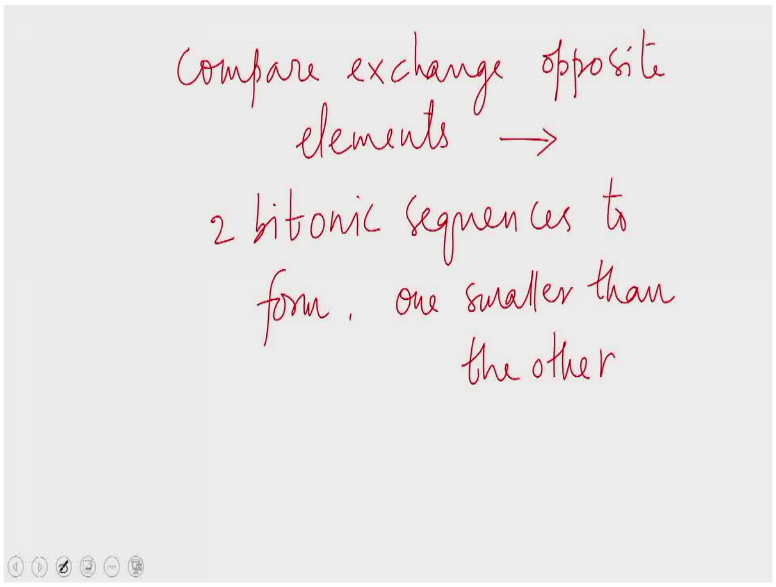
Then let us consider an 8 bitonic sort merge sorter. Here we are given 8 input lines, we divide them into two halves and then sort each half independently. And for sorting each half we will be using a 4 bitonic sort merge sorter. Therefore, on each half we will be fitting one 4 bitonic sort merge sorter.

That will take three steps as we have just seen a 4 bitonic sort merge sorter will take three steps complete. Now, the upper half is sorted and the lower half is sorted now all that is to be done is to merge the 2 halves. To merge the 2 halves we will use a 4 by 4 bitonic sort merger and we know how 4 by 4 bitonic sort merger looks like.

So, that completes an 8 bitonic sort merge sorter. So, the number of steps required is 1 2 3 4 5 and 6 so, it requires six steps. And the number of comparator comparators required is 4 plus 4 plus 4. In the first 3 steps then again 4 then again 4 and then again 4. So, it requires 4 comparators in every single step for a total of 24 comparators.

So, as you can see this is slightly more expensive than then odd even merge sort of. It takes seems to take the same amount of time for the cases that we have considered. When we do the analysis we will find that that is indeed the case and the cost seems to be slightly more which again is correct.

(Refer Slide Time: 55:47)



Compare exchange opposite elements $\longrightarrow$

2 bitonic sequences to form. one smaller than the other

So, what now remains to show is that in a bitonic sequence when we compare the opposite elements. When we compare and exchange the opposite elements for every element causes 2 bitonic sequences to form where one is entirely smaller than the other, in the sense that every element of one is smaller than every element of the other.

So, once we show this we will have established the correctness of the bitonic sort of. Therefore, the merger as well as the merge sorter will be working correctly. So, this is the crucial result that we have to prove and then we have to analyze the network for the time and cost complexities which we shall do in the next lecture. So, that is it from this lecture hope to see you in the next lecture.

Thank you.