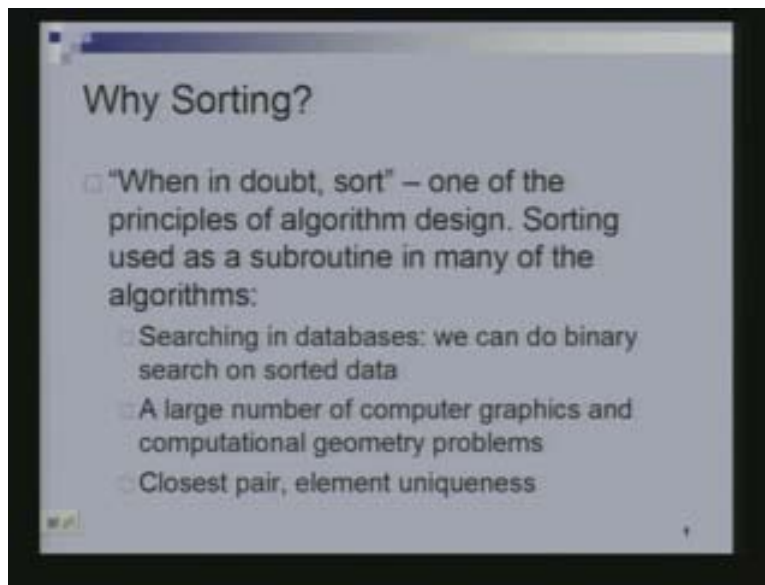


Data Structures and Algorithms
Dr. Naveen Garg
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi
Lecture # 22
Why Sorting?

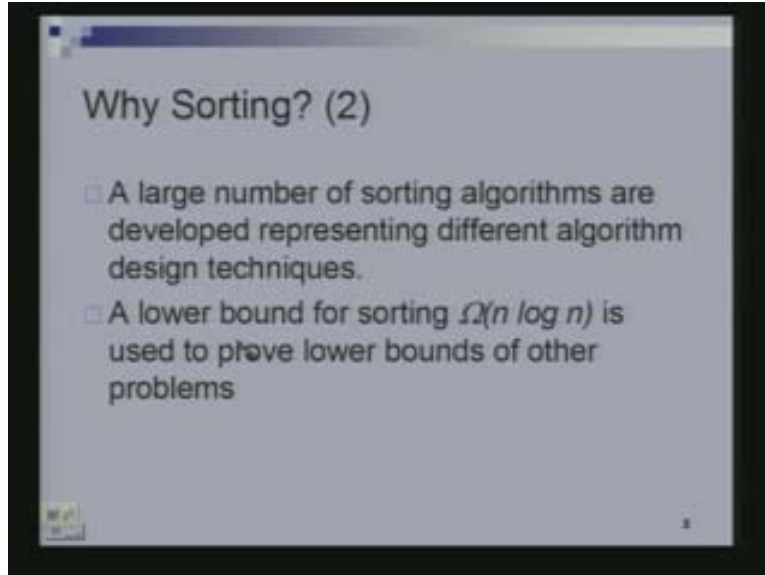
Today we are going to be looking at sorting. Sorting is actually one operation that computers love to spend their time on. You know it is estimated that 90 % of the time spent by all computers is on sorting numbers.

(Refer Slide Time: 01.26)



So it is really a problem which requires attention, requires consideration and so we need to develop good algorithms for it. So in the last class before you have looked at heap sort. so today we are going to look at another sorting algorithm. So where does sorting occur? Most oftenly in databases. When you are searching in the databases you want to keep data sorted in a certain manner so that the search becomes efficient. Recall the search is more efficient when you have the data sorted. There are lot of settings in computation geometry, computer graphics where sorting is essential. So I am not going to more of these applications. But we are going to look at certain sorting algorithms.

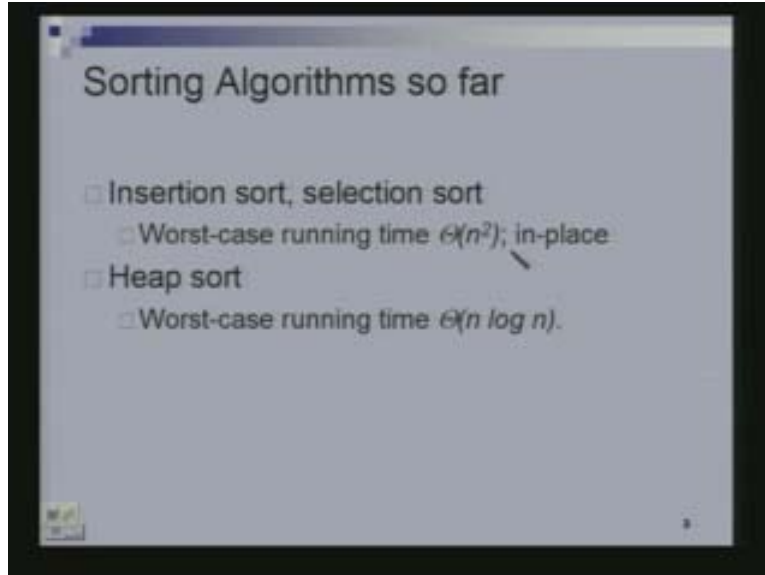
(Refer Slide Time: 02.14)



Sorting algorithms also give us an idea of different algorithm design techniques. So actually this is a point we have not spent any time on yet. What is an algorithm? It is a way of solving a problem and there are certain techniques that people adopt to design an algorithm. Given a particular problem, how do you design an algorithm for it? So there are certain standard of techniques in sorting. It shows you some of those techniques. So today we are going to look at one such algorithm design technique through a sorting algorithm.

There is also what a lower bound on sorting is. It means there is no sorting algorithm which requires comparisons. No sorting algorithm can sort n numbers in less than $n \log n$ comparisons. There cannot exist any sorting algorithm. So we are also not going to be spending time on this. But this fact is useful to prove lower bounds for other problems as well. This was as far as motivation for why are we spending so much time on sorting is concerned.

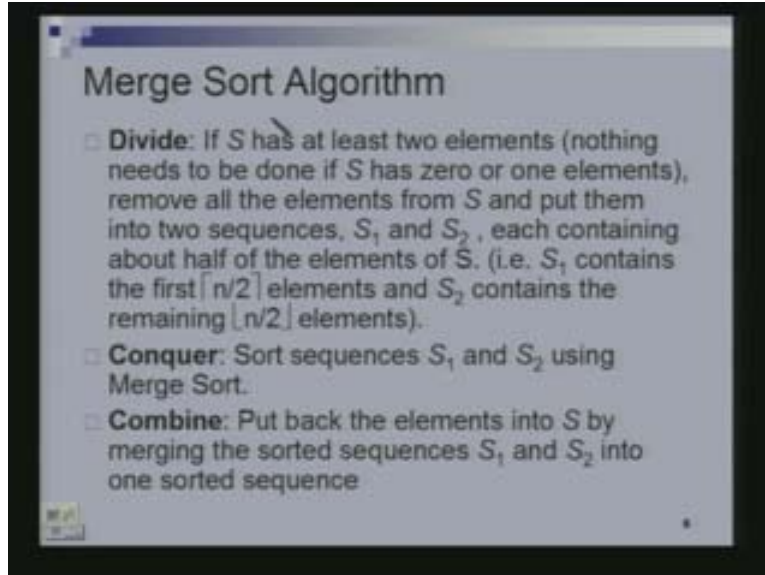
(Refer Slide Time: 3.32)



So far we have seen insertion sort and selection sort. These are two sorting algorithms with the worst case running time of n^2 . There are two algorithms. I should also have written quick sort. Actually heap sort has a worst case running time of $n \log n$ and one algorithm that is missing from the list is quick sort which has the worst case running time of n^2 but an average case running time of $n \log n$. So today we are going to see another algorithm and the algorithm design paradigm that is used here, what is called divide and conquer.

So some of you might have seen divide and conquer in your earlier CS Course. Basically what the idea is that, given a certain problem, to solve the problem you first divide the problem up into two or more parts. Then you recursively solve those sub problems. So that's what is called the conquer step. So recursively solve the problems on those parts. When I say recursively solve, how do you solve the problems on those parts? By once again dividing them into smaller parts and solving the problems on those smaller parts and so on and on. So use the same algorithm to solve the problem on the smaller pieces and once you have the solutions for the smaller pieces, you need to somehow combine the result to get the solution for the original problem. So that is what is called combine step. We will see through an algorithm for the sorting problem.

(Refer Slide Time: 05.02)

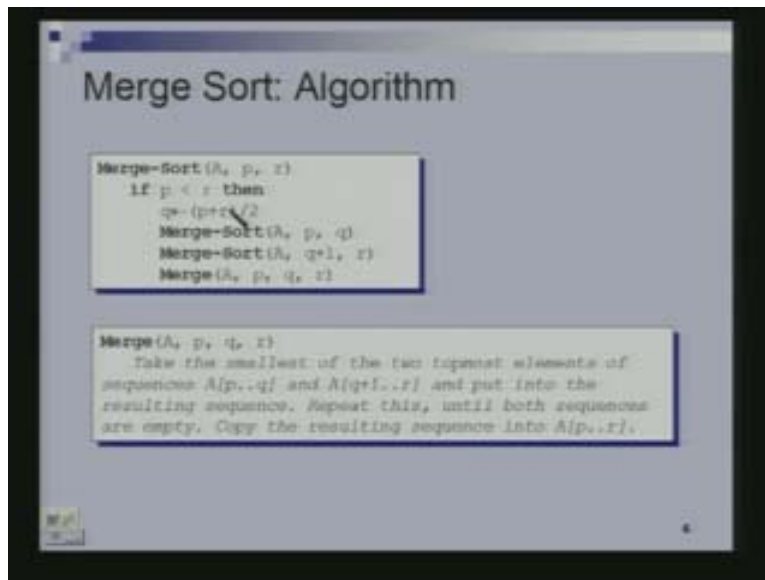


We are going to look at what is called the merge sort algorithm. Some of you again must have seen this before. If you have seen it, it is ok because we will see at least a different way of analyzing the algorithm. So what is sorting? Basically you are given n numbers which you need to put in increasing order, let's say. Let's say S is the set of n numbers. What we do is we would split ' n ' into two sets; S_1 and S_2 . So this is the divide step. The split would be such that S_1 and S_2 would contain roughly half of the elements of S . Now recall this should look very similar to another algorithm; the quick sort algorithm. There also we did a division at the very first step. But how was the division done there? It was not equal. So in the divide step there, we were dividing the problem into two sub problems but those two sub problems were not equal in size.

One could be smaller than the other. But here we would make sure that we divide it into two equal sub problems. Equal I mean, the numbers of elements in the two sub problems are equal. So if S_1 contains the first one, let's say $n/2$ elements, then S_2 contains $n/2$ floor elements. What do you mean by first $n/2$ elements? Not the first $n/2$ in a sorted order because I don't know what the sorted order is. S is just some arbitrary collection. I just take the first $n/2$, call that S_1 the next $n/2$, call that S_2 . Now conquer step is that I would sort these two sequences. I would sort S_1 . I would sort S_2 . So recursively I am solving the same problem and then once S_1 and S_2 are already sorted, I have to combine. I wanted a sorted sequence for S . I have S_1 as one sorted sequence, S_2 as another sorted sequence. But what is S ? It is not just S_1 followed by S_2 . No guarantee that is sorted. So I need to what is called 'merge' S_1 and S_2 into one sorted sequence. That is called the merge step or the combine step here. Now who can tell me, in the case of quick sort, the divide step was, I take an element and I compare. That's the pivot element and I compare every element against this element.

We had this procedure called partition which was helping us do that. that was a more involved procedure because we were trying to do it in place. So we were kind of just swapping elements till we got two sequences. One containing all elements less than the pivot and another containing all elements greater than or equal to the pivot. That was the division step. How much time did we spend in division there? Order n because we have to compare every element against the pivot. How much time do we spent in the division step here? Constant time because we are just saying if it is an array we just say this is S_1 & this is S_2 . We are not copying anything from anywhere to anywhere. That is one difference. We will come back to the conquer thing. In the case of quick sort, we were doing the same thing. We partitioned. Then we said we will quick sort one part, quick sort the other part and then we will combine. Did we have to do combine there? No because we could just take the first sorted sequence, put the pivot element and then take the second sorted sequence and that would be the sorted sequence for the entire thing. So how much time was combining there taking? Constant time. Here combining will not take constant time. Here we have two sequences and we have to combine them. We will see that it will take order n time. This is also called the merge step and the name merge sort is deriving from this step. So every one understands what the algorithm is.

(Refer Slide 09.24)

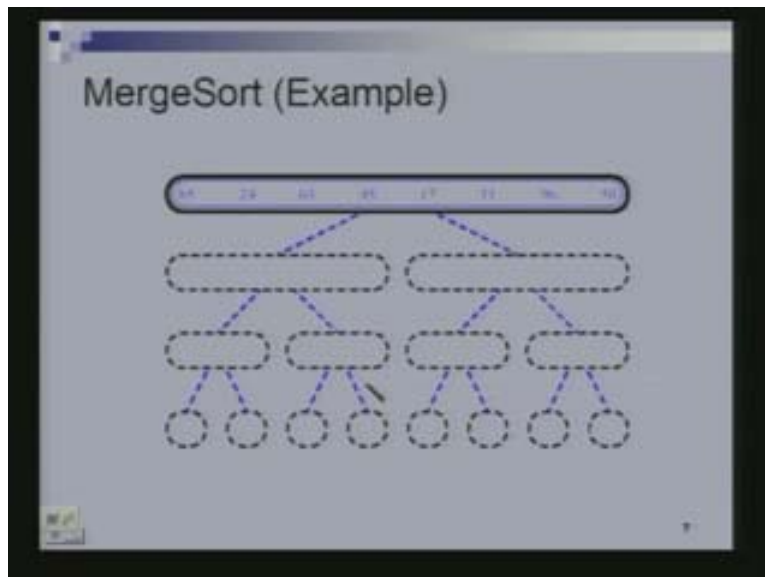


So for instance, this is the pseudo code for the algorithm. I give you an array A , p and r . identify the part of the array, the lower and the upper bound which needs to be merge sorted. if p is less than r , only then we need to do something. p is the lower and r is the upper index. So q is the mid point p plus r by two. So this is the division step. These two are the recursively solving the conquer steps. I am saying from p to q sort the first part of the array, from q plus one to r sort the second part of the array .sort using same merger sort procedure and once you have this part and this part sorted you have to combine. For that I have another procedure called merge to which is specifying p q and r .

So this procedure the definition is this following. This is what it is doing but what is the definition of merge? It is going to take the two sequences. One sequence is from p to q the other is from $q + 1$ to r and merge them and make one sorted sequence which will sit in p through r . That is merge and how does merge proceed? That's essentially what the merge algorithm is doing. We are going to see examples of it. So don't worry but next is read through this and you will understand a little bit of what merge is going to do. it is going to take the smallest of the two top most elements of these two sequences. What do I mean by the smallest of two top most elements? What is the top most element of this sequence? The first element. The top most of the first element which is p or the element at the location $p : A (P)$ and the top most element of this sequence is A of $q + 1$. I am going to take the smaller of these two and put it into the resulting sequence whichever is the smaller when I have to merge.

So if I have to merge these two sequences which will be the smallest element of these two sequences. It will be the smallest of this and these two the smallest of these two numbers. I don't have to worry about any other numbers because this is already sorted and this is already sorted in increasing order. Take the smaller of these two and that gives me the first element of my resulting sequence. In this manner I can continue building up the resulting sequence and we will see how that's done.

(Refer Slide Time: 12.07)



That's merge. Now let's look at an example for merge sort. This is my sequence. As you can see, it's not sorted. It's an 8 element sequence. So the first step is a divide step. You will divide this into two parts. So one is this and the other part is this. This is the division. I have shown it lower down because I am kind of doing a recursive call of merge sort. so this corresponds to one level of recursive call. So first do a merge sort here. This is another merge sort call. Then I will do another merge sort call here and so on. This is a recursive call on this part of the sequence on these 4 elements (Refer Slide Time: 12:55) and how is this done?

This is done by once again dividing this into two parts: 58, 24 is one part and the other is the other part and I do a recursive call on this part now first. First I do the division and then I do a recursive call on the first half. Then I do a recursive call on the second half. [HINDI] we just seem to be doing divisions and doing recursive calls as you can see. How do I sort this part?

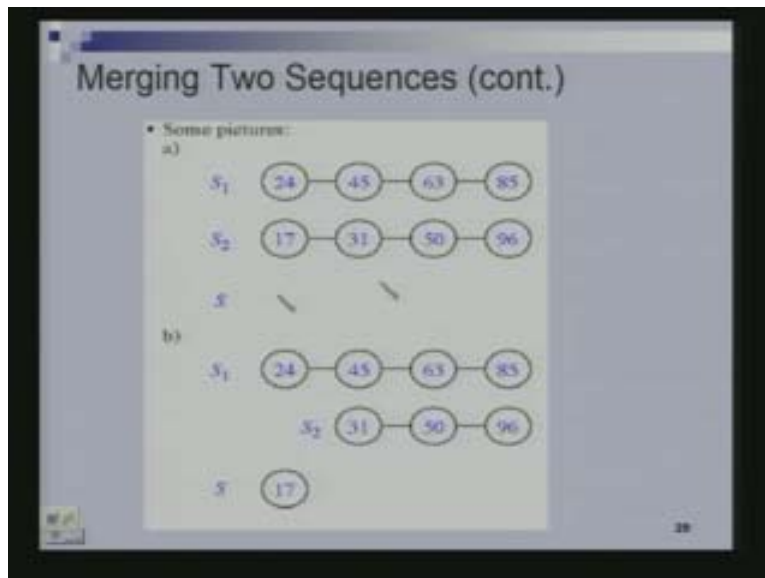
Once again I divide it into two parts - 85 and 24 and I do a recursive call on 85. Just a single element sequence and what is the solution for this when there is only single element? That itself is a sorted sequence. So 85 goes back up. So to say we go back to the parent calling procedure. So that is 85. So this is the sorted first half and this goes down. This is also sorted. So this is the sorted list and now we have to do the merge at this call. so we have to merge this sorted sequence which is just 85. Sitting in this blue oval are two sort sequences. We have to merge them. So right now we will just do the merge. So the result as you can imagine is 24, 85 and this is one sorted sequence. So now we have finished the merge sort for this call that we had done – 85, 24. So now this can go back up. We have finished that merge sort. Now we are getting ready to do the merge sort on this second half. Let me show you the procedure once again. First we divide. Then we merge sort the left half and then we merge sort the right half and when I merge sort the left half, what happens?

I once again divide and then I merge sort the left half and then I merge sort the right half. Then I merge sort the left half. What do I do? I once again divide and so on and on. So in some in some sense, first do a merge sort on this and nothing is to be done here so far and then how do I do this? Once again I divide and then I do a merge sort on these two but first I do it here. Only when I have finished this merge sort, will I come back to this one and how am I doing the merge sort here? Once again I am dividing. This is the trivial case. So this is easy to do. This is the trivial case. I get this sorted sequence. So I have finished the merge sort for this part now. So this goes back. Now I have to do the merge sort for this part. Once again I am doing a recursive call on this. I do a call, divide trivial merge; 45, 63 and now this is my sorted sequence. So I am now ready to go up and this is now the two sorted sequences that I get. Now I can merge this. So I will show you the merge step again more clearly but you can see which should be the first element. It should be 24. So this is the sorted sequence or the merge sequence.

Now I am done with this part. This merge sort call has finished. The merge sort call that I have made on these four elements that has finished. So it goes back. Now I do a merge sort call on this. So I do one more merge sort call here. I am not showing you the entire thing. I am just showing you that this entire thing would happen in exactly the same way. 17, 30, 1, 96, 30 would sort and you get 17, 30, 1, 50, 96. This goes back and then we will merge these two and this is the resulting sequence. So these are two sorted sequences we can merge them. I will show you how to merge them very quickly and that will give us the resulting sequence. So who can tell me how many merge sort calls have I made in all? How can you count the number of merge sort calls you have made? In this example how many merge sort calls have I made? Order n . Why because each of these blocks correspond to one merge sort call. [hindi] so the number of merge sort calls is just the number of nodes in this tree and how many nodes are there in this tree?

Well you can see that this is the tree? How many leaves are there in this tree? n leaves. There are n leaves. At this level there will be $n/2$ elements. at the next level there will be $n/4$, $n/8$ and so on and on. Some of this sequence n by two n by four n by eight n by sixteen so on is n minus one you can check that. There are many other ways of thinking of it. You can think of it has a complete binary tree on n leaves and you can apply those formulas that you learnt. [hindi] so this will do exactly $(n-1)$ recursive calls to merge sort.

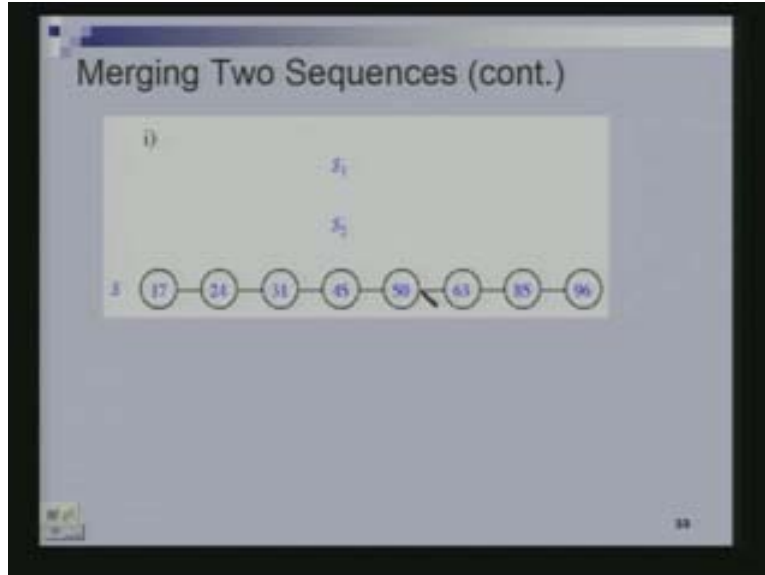
(Refer Slide Time: 18.49)



So let's see the part that we missed out on how do we merge two sequences. I should really say two sorted sequences. So let's say these are two sorted sequences. I have S_1 S_2 . 24, 45, 63, 85, 73, 150, 96. These are I believe coming from 24, 45, 64, 85. Are they the same? No they are not the same. So these are the two sorted sequences. We have to merge them. So what should be clear is that the first element of the resulting sorted sequence is the smaller of these two. So it is clear that the resulting sorted sequence which is S has its first element 17. Then I can strike away 17 from here because I have kind of removed it and I have put it here. I have to merge these two sequences and append them to this sequence that I am constructing. So how do I merge these two sequences now?

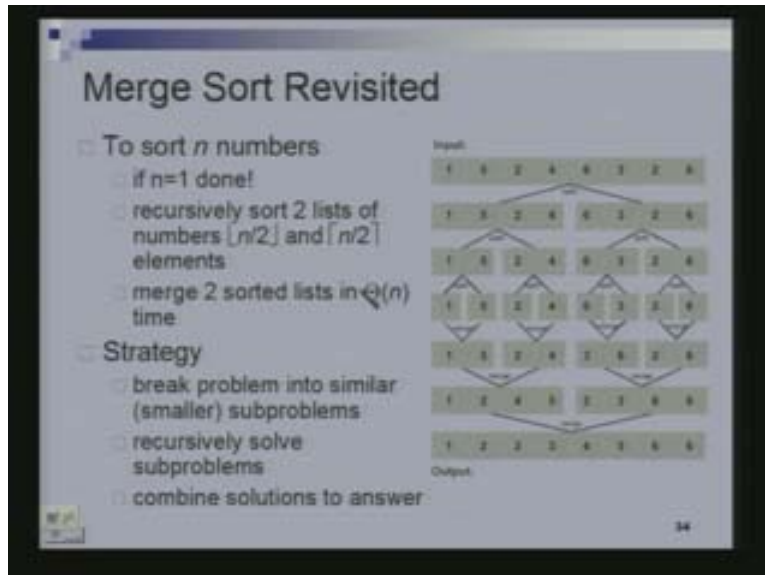
Once again the same idea which is the smallest element which is the next element will be the smallest element in these two sequences put together and that will be one of these two. The next element should be 24 and so I remove 24 out of there and I get two smaller sequences like this. Once again I have to find out the smaller of these 6 elements that has to be the smaller of these two. So I will just compare these two and I will write that down here. So I compared. I found 31 is smaller. I write that down here and I remove 31 from this and then in this manner I can proceed. So 31, 45 would be the next one to go. These are things left. Then next one would be 50. It would be left with 63, 85, 96.

(Refer Slide Time: 20.40)



Then the next one would be 63. We would be left with 85 & 96. Eventually it would be taking that and we would have nothing left. This is what you would get as a result.

(Refer Slide Time: 20.56)



So how much time do we take in doing? Order n . Why? Because what are we doing? We have these two sorted sequences. We are comparing two elements; the two right & the two front elements. We are comparing them and taking the smaller of those two and writing them out into the resulting sorted sequence. With every comparison, we are writing out one element.

How many elements do we write out in all? As many elements as there are. so if there are n elements, then I am going to be writing out n elements in all which means how many comparisons am I making? n comparisons. This was assuming whether two sequences were of length $n/2$ and $n/2$. Suppose one sequence was of length 'l' and the other sequence was of length 'm'. One sorted sequence of length 'l' elements and the other sorted sequence with 'm' elements in it. S 1 & S 2. What is the total time taken by merge? $(l + m)$. These are the four choices. Let's see minimum of (l, m) max of (l, m) [hindi]. $(l + m)$ [hindi] none of the above. [hindi]none of the above [noise]. Which do you think is the right? Let's look at worst case. How can you tell me what the worst case is? Let's look at worst case. In the worst case what do we have to do? We have to in the worst case we will have to compare. The worst case should be $(l + m)$. We might have to do every possible comparison.

What is the best case in your opinion? Best case it's not really clear. What I am trying to say by best case because what might happen is that if one of the sequences finishes then I can just copy the next one. So even the copying requires time. So if you include the time for the copy, then its always going to be $l + m$. you cannot do anything more about it. But if you don't want to include the time for the copying, if you are just counting the number of comparisons that you do. Then if you are just counting the number of comparisons, then you can do less than it will be. So you all understand this. In the worst case, why do you require $l + m$ comparisons? So you might require as many as $l + m$ comparisons. So in fact that was happening in this example. So you know if you count the number of comparisons that we did in this example. For every element that we had to write out, we have to do a comparison except for the last one. So you really require $l + m - 1$ comparisons. That is the worst case. The number of comparisons if you are counting the number of comparisons. You really require that many and you can create sequences of arbitrary values of l and m so that you require $l + m - 1$ comparisons.

(Refer Slide Time: 26.19)

Merge Sort Revisited

- To sort n numbers
 - if $n=1$ done!
 - recursively sort 2 lists of numbers $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$ elements
 - merge 2 sorted lists in $\Theta(n)$ time
- Strategy
 - break problem into similar (smaller) subproblems
 - recursively solve subproblems
 - combine solutions to answer

The diagram illustrates the merge sort process. It starts with an 'Input' array: 1 5 2 4 6 3 2 8. This array is recursively split into smaller sub-arrays until individual elements are reached. The process then reverses, merging the sorted sub-arrays back together. The final 'Output' array is: 1 2 2 3 4 5 6 8. The slide number 34 is visible in the bottom right corner.

So we have understood merge. We have understood the merge set we understood what merge sort is. So let's try and analyze it. How much time? How many comparisons does merge sort take in all? This is a quick recap of what we said so far. We have this sequence. We split it into two. Then we split this into two, this into two and so on and on. When we had these single ones, then we merge this and we got 15. In the previous example I had shown you them going up. I have just kind of created a mirror image. so I have merged them here. Merge these 2 get 2 4 merge this to get 3 6 merge these 6 get 2 6. Then I decided to merge these and I get 1, 2, 4, 5 here. I merge these two get 2, 3, 6, 6 and then I merge this to get 1, 2, 2, 3, 4, 5, 6, 6.

So you could also think of pictorially. We could also think of this as what is happening that you did your divisions of algorithmically. It's the same thing you are doing. Your recursive call this is just a way of representing it. Why am I doing it this way because it will now be easy to compute what the running time is. Can you by looking at this picture say what the running time should be? No time spent in all of this part. No time spent in this split part but when we are doing the merges we spend time. How much time do we spend in doing this? Merge two units essentially. How much in this two? Why am I counting two length of the two sequences? I am merging. I am always going to count that one $1 + m$. so [hindi] each of these merges require two unit of time and how many such merges would I have to do at this level n by two such merges? If these were n elements, then I would do n by two merges of two elements. So total time would be $n/2$ into $2n$. so that's what is going to happen. We are going to see another way of doing this in a short while. But you should understand what's happening. So each of these merges is taking two units and there are $n/2$ such merges and after that we get sequences of length two.

After this step we get basically sequences of length two. As you can see now we are merging these. So each of these merges is now going to take 4 units of time and there would be $n/4$ search merges. This should correspond to this and the next one would take 8 units of time and that corresponds to $n/8$ and there are $n/8$ such merges. n here is 8. So there is only one such merge but in general there would be $n/8$ and so you would get such a sequence. This is what we have to sum. This is the total time spent by your algorithm. We will see this in a slightly different way in a short while.

(Refer Slide Time: 29.45)

Recurrences

- Running times of algorithms with **Recursive calls** can be described using recurrences
- A **recurrence** is an equation or inequality that describes a function in terms of its value on smaller inputs

$$T(n) = \begin{cases} \text{solving trivial problem} & \text{if } n=1 \\ \text{max_pieces } T(n/\text{subproblem_size_factor}) + \text{dividing} + \text{combining} & \text{if } n>1 \end{cases}$$

- Example: Merge Sort

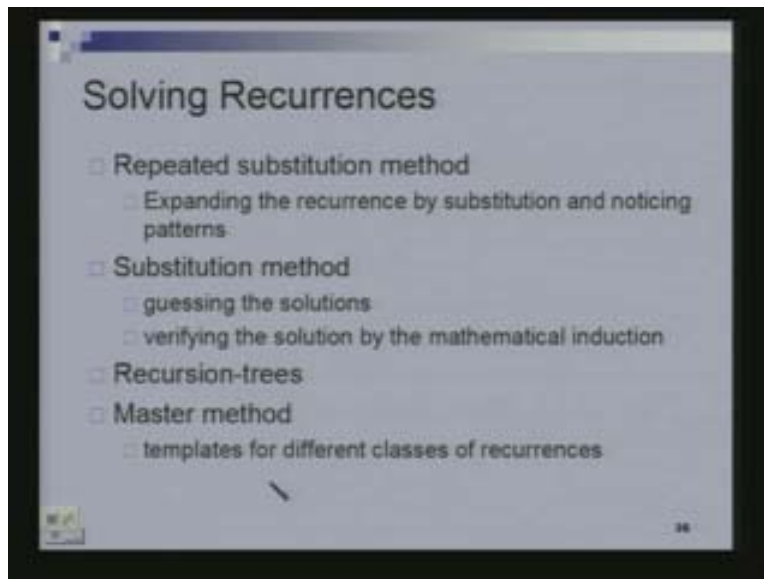
$$T(n) = \begin{cases} \Theta(1) & \text{if } n=1 \\ 2T(n/2) + \Theta(n) & \text{if } n>1 \end{cases}$$

So we are going to analyze it once again using recurrence relations. So we had used recurrence relation before I am going to say it once again so that it is refreshed in your mind. So typically for all recursive or algorithms of procedure which are recursive in nature, you can analyze their running time using recurrence relations and is basically an equation or inequality that describes a certain function in terms of its values at previous points at smaller inputs. So for instance, for a divide and conquer kind of a problem, the time taken to solve T if T of n is denotes the time taken to solve a problem of size n , Then it is equal to the time taken. Look at this part. Suppose I divide it into some number of pieces, how many problems did we divide it into two and each of the sub-problems was half of the original thing.

So the sub problem by sub problem size factor, I mean by what factor have I reduced the size of the problem? So that is the time required to solve each of the sub problems. This part for each sub problem, there is a time required that times the number of sub problems is the total time required to solve the various sub problems plus the time required to do the division plus the time required to do the combining. That would in general denote the recurrence that you would get. Of course this is when n is more; if $n=1$, then it is basically a trivial problem and whatever the time required to solve the trivial problem would be the value. It would be kind of the base case for this recurrence. The initial condition for instance for the merge sort problem, we divided the problem into two pieces and each of the pieces were of size n by two. So the time required to solve the problem of size $n/2$ to sort to merge sort $n/2$ elements will be $T(n/2)$. If $T(n)$ denotes the time required to merge sort n elements, we are using T of n to denote the time required or the number of comparisons required to merge sort n elements. Then this is the kind of a recurrence we get. I have written $\Theta(n)$ here but you can just write n here because that's the upper bound on the number of comparisons that we would do. Let's say that's the number of comparisons we are doing. So this is the kind of a recurrence relation. This is clear to every one. We have a problem of size one if $n=1$.

This is just a constant because we don't have to do anything here.

(Refer Slide Time: 32.48)



There are bunches of ways of solving recurrences. We have only looked at what is called repeated substitution method. What does that say? That basically says we expand the recurrence by substituting repeatedly and noticing any patterns that you get. There is another method which is called the substitution method where you guess a solution to the recurrence and then you verify that your recurrence really satisfies that guess. In the course of this course, we will perhaps see examples of this way of solving recurrence relations. There is a third method called the master method which basically says that you know if a recurrence is of this kind, then this is the solution for this recurrence and you just plug in the values of a and b and you get your solution. I personally don't like you to remember those formulas. It should always be possible to solve those recurrences by just looking at them.

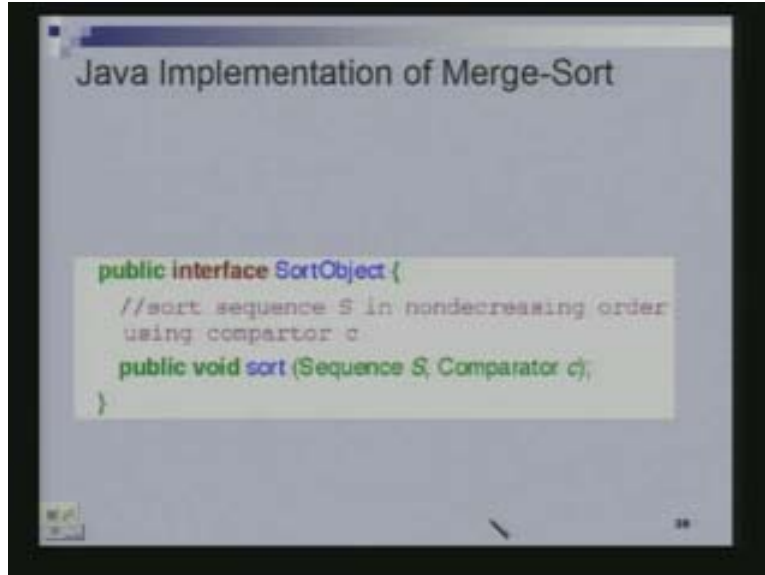
But these are three ways. Then there is a fourth method which is a recursion tree that you draw out a tree of the calls that have been made as we drew and you count at each node, how much time is being spent. So we are going to look at let's say, the repeated substitution method once again. Suppose I am trying to merge sort n elements and n is some power of two, let's say 2^b , this will help us ease the analysis a little bit. So recall the recurrences $T(n) = 1$ if $n=1$ and $T(n) = 2T(n/2) + n$. If n is more than one, so we are just going to adopt repeated substitution. I just keep writing over and over again and it's the same what I get. So I get $T(n) = 2T(n/2) + n$. So I substituted once and now I am replacing this $T(n/2)$ by $2T(n/4) + n/2$. So this is the substitution step. I am replacing this $T(n/2)$ with $2T(n/4) + n/2$. Why $n/2$ because this is $T(n/2)$. So this part ' $1/2$ ' is coming in here and this part whatever is n written here, it comes again here as plus n and now I just expand this which means I just write it as $2^2 T(n/4) + 2n$. Once again I substitute for $T(n/4)$. And I expand 2, get this and now I have to observe a pattern basically.

What is that's happening? I get 2^2 times T 4 times T $n/4 + 2n$, then 8 times $n/8 + 3n$ and so on. So in general at any step, I am going to get something like after I have done this thing, I times I get 2 to the i times T of $n/2$ to the $i + i$ times n . that's what the 3 , the 2 and 1 here and now if I choose i to be $\log n$, what do I get? I get 2 to the $\log n$ times T of n by $n + n$ times $\log n$. now T of n by n is 1 , 2 to the $\log n$ is n T of 1 is 1 . so this becomes n in all and this becomes $n \log n$. so I get a total of n plus $n \log n$ which is order $n \log n$. so that's the total time taken by this procedure. Everyone understands this. So the very simple recurrence you can just solve this using repeated substitution. Now if you are understanding this, let me ask you a question.

Does this give us the average case time? It is in all cases it will be the same time because we have written that the time to combine the time to combine is n . this is the combining time at the time to combine is always n . so to say because we need to look at our merge step more carefully, when we look at our merge step more carefully and I actually show you the code, we will see that you really need that much time. You can't do in less than that time as many elements as you trying to merge total number of elements. You are trying to merge. You really need that time. so what you have actually argued is that of n -the time for merge the number of time for merge is θ of $n \log n$. it is not ω . It's actually θ . It is always you take this kind of time. its bounded by some constant time $n \log n$ upper bounded and lower bounded also by some times. Now suppose I had instead of this one here, I had a C , what do you think will be a solution for this? Why am I doing this? [hindi] n into n plus $c \log n$. so this is by just looking at the solution you can see that this is what you would get [hindi] provided C was not that big. the one way of also thinking this entire thing is what we said a recursion tree. Let's draw out a recursion tree for this. Let me write down the total time taken in the division and the combining at this node. [hindi]

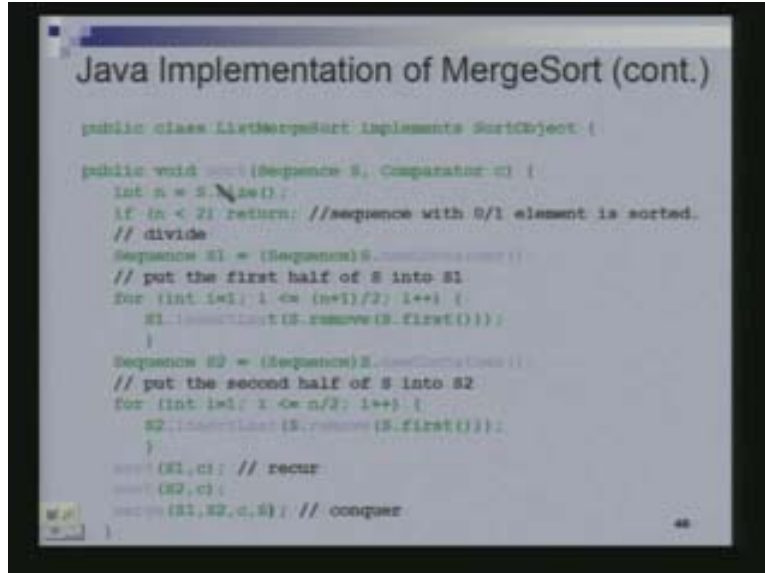
So I should write down n here. I should write down $n/2$ and so on and on. [Hindi] now what's the total time taken? It's basically the num sum of numbers written in these nodes which is n here. This level [hindi] and so on. [hindi] I have not said that these c 's are necessary constants. There could also be something will depend upon n . n which case this is how. this is another way to view this thing. You can draw out this tree and you can write the total time being done at that node in the combine at the divide step. So this is what we do in the repeated substitution method. Substitute, expand substitute, expand and keep doing that. Observe a pattern. So do it sufficient number of times. You observe a pattern and that on how you would get an expression after the i^{th} substitution and find out a value of i which gives you the value the initial condition. So to say this is what the method is.

(Refer Slide Time: 42.44)



So I can now show you a kind of a java implementation of merge sort. So these things are clear. It is a very simple algorithm. That's why I am showing it to you. Here we are also taking a comparator object. What is a comparator object? We are not just merge sorting integers. We could use this to sort anything. So we are looking at a java implementation of a merge sort. So this would require the sort object. So you define an interface called sort object which is implementing a method called sort and this method requires a sequence and a comparator. Why comparator because we said we are not interested in sorting integers. We could be sorting anything. So if I give you tow objects, this comparator is going to tell me whether one is less than the other, greater than the other and so on. Depending upon what you want to do, the sorting on you will design a comparator. So the comparator typically has three functions. 'is less than', 'is equal to', 'is greater than' and you could also have 'is less than' or 'equal to also' or you could just these two do the implementation and all of these three functions are essentially Boolean. They will say true or false.

(Refer Slide Time: 44.31)



```
Java Implementation of MergeSort (cont.)

public class ListMergeSort implements SortObject {

    public void sort (Sequence S, Comparator c) {
        int n = S.size();
        if (n < 2) return; //sequence with 0/1 element is sorted.
        // divide
        Sequence S1 = (Sequence)S.clone();
        // put the first half of S into S1
        for (int i=1; i <= (n+1)/2; i++) {
            S1.addLast(S.remove(S.first()));
        }
        Sequence S2 = (Sequence)S.clone();
        // put the second half of S into S2
        for (int i=1; i <= n/2; i++) {
            S2.addLast(S.remove(S.first()));
        }
        sort(S1, c); // recur
        sort(S2, c);
        merge(S1, S2, c, S); // conquer
    }
}
```

So this is basically the merge sort. You have this sort. So you have the sequence comparator. Let's say 'n' is the size of the sequence. If n is less than two which means they are just zero or one elements. Nothing to be done. It is already sorted. Otherwise you create a new sequence. S 1 is a new sequence and you are going to insert the first half of s into S 1. So let's say we have an operation called insert last which will insert an element into the sequence S 1. This is an operation. This is the method on the class sequence or a data type. Similarly we create a new sequence called S 2 and we put the second half of s into S 2. These are how you implemented your methods. You could have done it this way. You could have done it in some other way. But let's say we have such methods. Then you can break it up in this manner and now you just need to call sort. Sort on S 1 and S 2. This merge sort on S 1 and S 2 and then eventually you need to do this merge. So S1 and S 2 were two sorted sequence. c is your comparator and S is lets say the resulting sequence you want.

So you will have to do this step and what does this merge step contain? This is basically a two sorted sequence. You have a comparator. This is the resulting sequence. What you're saying is that while both the sequences are non-empty. You will keep doing something. What is that something? You will compare the first element of S 1 and the first element of S 2. If the first element of S 1 is less than or equal to the first element of S 2, then what is that you should add to S? You should add the smaller which is S1. you should add the first element of S 1 to S. so you are going to remove the first element of S1 from S1 and add it to s. that's the only thing you are doing here in this statement. Get the first element of S1, remove it and then insert it into S.

And you are essentially repeating this. If this is not true, then you will insert, remove the first element of S 2, insert into this and you keep repeating it. Still either of S 1 or S 2

becomes empty when one of them becomes empty. Then depending upon which one it is, if S 1 is empty then that means that you have to just copy S 2 to S. so which means that you will once again keep removing one element at a time from S 2 and inserting it at the end of S.

That's what you are doing. Removing the first element of S2 and inserting at the end of S. You keep repeating this. If S1 is empty, if S2 is empty, then that means that S1 is left with some elements. We will keep removing one element at a time from S 1 and insert it into S. With that we will end today's discussion on sorting. We learnt what merge sort is on which we also saw the divide and conquer paradigm for sorting is.