NPTEL
NPTEL ONLINE CERTIFICATION COURSE

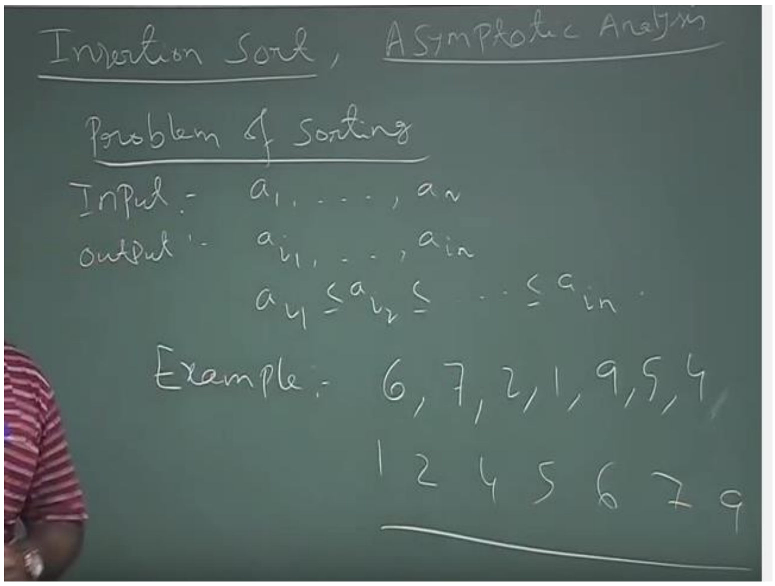
Course Name
Fundamental Algorithms:
Design and Analysis

by
Prof. Sourav Mukhopadhyay
Department of Mathematics
IIT Kharagpur

Lecture 01: Insertion Sort and Asymptotic Analysis

Okay, so we will talk about sorting problem.

(Refer Slide Time: 00:27)



Insertion Sort, Asymptotic Analysis

Problem of Sorting

Input:- $a_1, \ldots, a_n$

output:- $a_{i_1}, \ldots, a_{i_n}$

$a_{i_1} \le a_{i_2} \le \cdots \le a_{i_n}$.

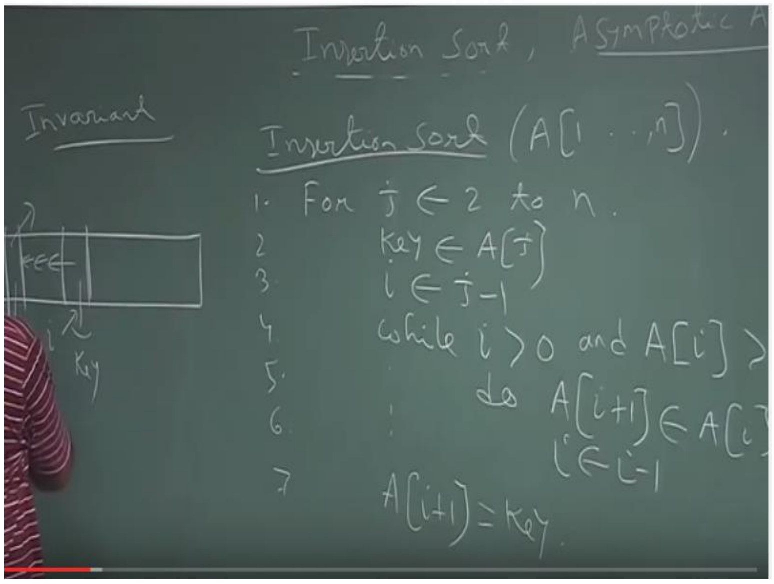
Example:- 6, 7, 2, 1, 9, 5, 4

1 2 4 5 6 7 9

So the problem of sorting, so basically we have a input of size n, say we have a array 1, 2….. a and the output will be a rearrangement or a formulation of this array, such that a_{i1}, a_{i2} less than a_{in}. So this is the rearrangement of the number in a sorted way. Now example is suppose we have given this array so 6, 7, 1, 2, 9, 5, 4, so when this is our input.

Now we want to sort it, so the output will be 1, 2, 4, 5, 6, 7, 9, sorted. So we will look at some way how we can do this. So first algorithm we will be looking at what is called insertion sort.

(Refer Slide Time: 01:48)



Insertion Sort, Asymptotic A

Insertion Sort $(A[1 \ldots n])$.

```
1. For $j \leftarrow 2$ to $n$.
2      key $\leftarrow A[j]$
3.     $i \leftarrow j-1$
4.     while $i > 0$ and $A[i] >$
5.         do $A[i+1] \leftarrow A[i$
6.            $i \leftarrow i-1$
7      $A[i+1] = $ key.
```

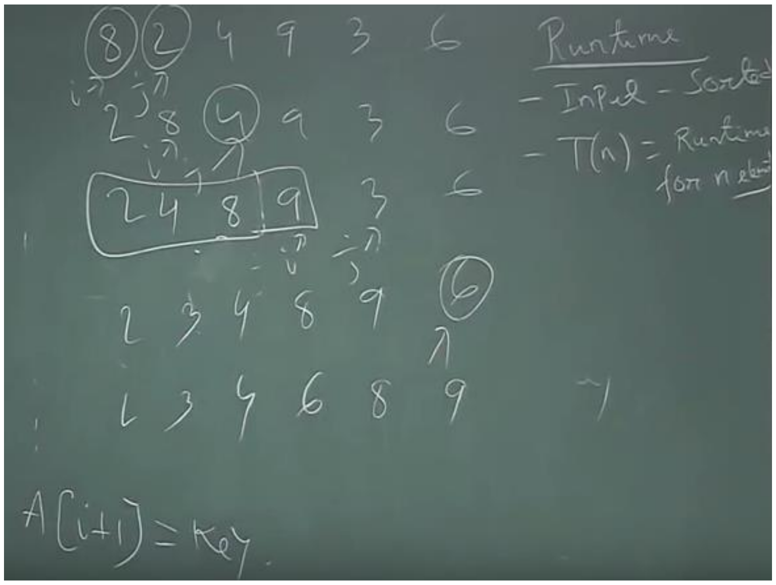
So the input is an array of size n. So we have a for loop for i = 2 to or we can say j = 2 to n, we put this in key value and if i = j-1 and we have a while loop, while i>0 and A[i] > key value.

Then we have to A[i] and this and then we reduce i/i-1 and then, so this is 5, 6, 7. So here now we put A[i+1] = key. So this is the, this is what is called pseudo code. So pseudo code means it is not a very much English description what we are doing, but it is a language, I mean it is in follow where, but it is also not a some specific syntax of the language like C language.

But if we can give this to some programmer he or she can easily implement in that particular language, so that is called pseudo code. Okay, now let us try to understand this, so what is the loop invariant, what loop invariant is, so our j is pointing somewhere here, so this is our at some point of time this is our key value which is basically is A.

Now initialize by 2, now it is trying to find your audits position like this. So by shifting this like this, so finally if we got the position we put this value over here and every time we are shifting this way, like this okay. So we can take an example to execute this.

(Refer Slide Time: 05:06)



Suppose we have these numbers 8, 2, 4, 9, 3, 6 and we want to sort this using the insertion sort. So we are starting with j=2, so this is our key value and then our i is this value j-1, i is starting with this. Now as long as i>0, i>0 and A[j], A[i], A[i] >key so if A[i] is less than k then we do not need to do anything, but if i>k then we need to find out the location for this A[i] by shifting this. So what we do, we will shift 8 here and we reduce i/1, so i/1 it will be 0, so we stop there, so we put this key here and now j is pointing here, this is why that follow.

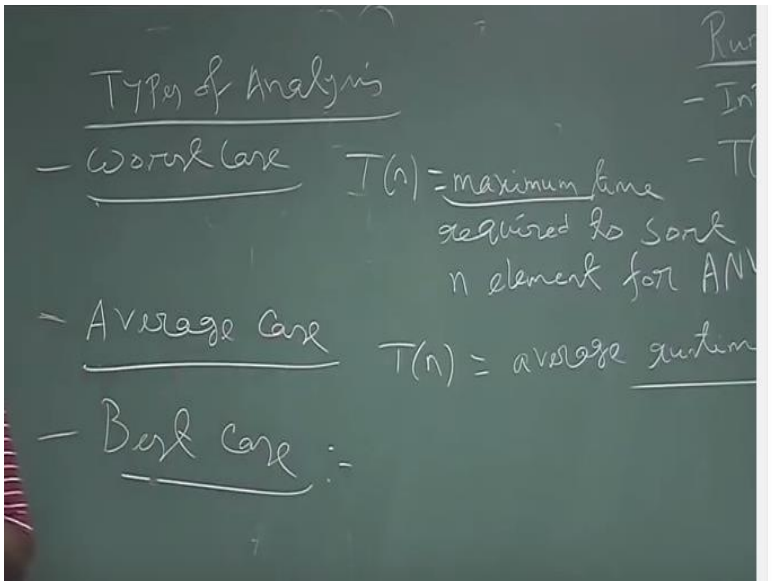
So again this is our key value and we, what do we do we just look into this, so this our now we put i is equal to this now we look at here so this i is, I mean key is Ai is greater than this so we will shift we have to shift here and then i is i-1 so that is this that is not zero but now the Ai value is less than key so we will not go for this loop so we just put A[i+1] is equal to key so 4 will be here like this.

So now j is pointing here so now this is, now this is our key so we compare this with the pervious value so every time if we observe this part is already sorted. Now we need to find out the position for this guy and this, this is greater than so cool no need to do anything so we will increase j by 1 so j is now pointing here now we compare this, so we compare so this our i now so we compare this, this is less so we shift this here, we shift here, we shift here, we shift here, and finally we put 3 here.

So 3 has to come long way to here and now 6 is point 6 is the new i this is the key so again we do the same thing, 9 will come here 8 will come here and 6 will be here, that is it j is gone, so this is the execution of this code okay. Now what is the run time of this algorithm or run time of this piece of code, so well run time will depend on few things like what is the, I mean when we are, we do not need to do many comparison here if the input is sorted, if the input is already sorted we just need to compare with the previous one that is it, but if the input is reverse sorted.

That means everybody has to come a long way to the beginning so the run time will depend on the input the pattern of the input and also it will depend on the size of the input. Suppose we need to sort 10 element and we need to sort 100 element obviously 10 element will take less than time than 100 element, so we can perimeter is this runtime by the input size, suppose T(n) is the runtime for, for n element. So either, so this is the perimeter action we are doing so your n is runtime is T(n) where n is the size of the input.

(Refer Slide Time: 09:26)


Types of Analysis
- Worst Case
T(n) = maximum time required to sort n element for ANY
- Average Case
T(n) = average runtim
- Best case :-
Run
- In
- T(

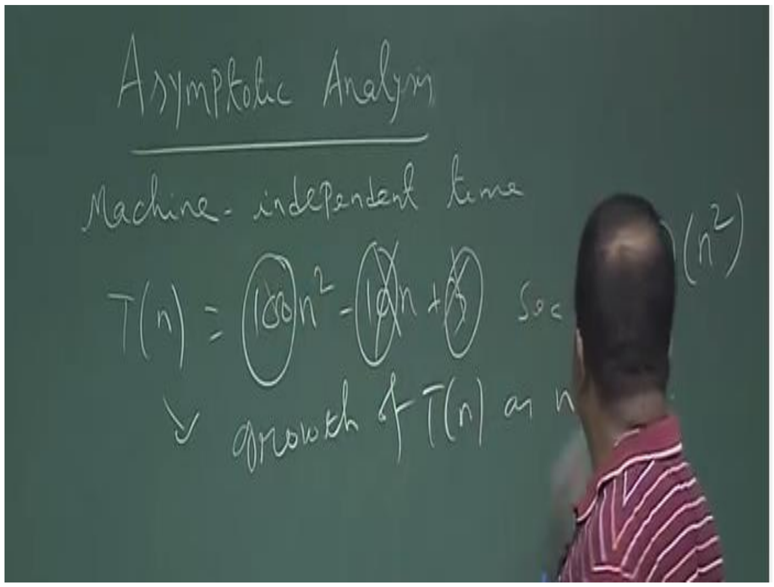
Okay, so we will do three types of analysis, analysis first one is worst case so that means T(n) is the worst case means we have to see the maximum time it is requiring for any input, say for worst case for inner sensor if the input is revere sorted then it is the worst case because everybody has to come forward to the beginning, okay.

So now this is the maximum time, time require to sort n element for any input okay, that is sort of guarantee, we can guarantee that this the upper bound, we can guarantee that our time is this much maximum, whether any put you can give no matter the input you choose, the our algorithm will perform like this, it will not be worst than it will not be better than which I am claiming, so that is the usual analysis we will do because we want the guarantee, this is sort of guarantee.

Okay now second one is average case, average case means sometimes we will follow some distribution of the input and we will see how this T(n) is, so T(n) is the average run time or expected okay, and the last one is which is really bogus best case, best case means sometimes like for insertion sort when is the best case if the input is already sorted okay so it is a bogus. I mean if we design as a sorting algorithm and if you give it to somebody to purchase and you tell okay this is my sorting algorithm and this is performing very nice for this type of input so you are declaring your pattern of the input or perform, your code will perform good.

That is the best case but nobody will believe that, people will try for all other input and see where your code is performing bad so that is why we will not prefer to go for this based case, this is sort of cheating you are cheating by providing the input, for this type of input my always good but you are hiding that usual case analysis you have to tell what is the maximum, maximum run time for your code for any input so that is the user analysis we will do. Okay, so apart from that suppose you want to do washed case analysis for insertion sort.

(Refer Slide Time: 12:46)



So we want to do washed case analysis for insertion sort. So washed case means input is reverse sorted, we have given the input which is the reverse sorted that is the washed case scenario for the insertion sort so then let T(n) be the run time where n is the size of the input, now will it depend on any other thing other than this, any other factorial continued in this run time, we have fixed already TAN we are dealing with the washed case, so any other parameter or any other factor will be affecting this run time, yeah it is the speed of the machine, the computer you are running this code.

Suppose you are running in a Pentium one computer or some old micro processor 8085 8086 and you are running this code in a band new laptop code, code to processor so obviously code to processor will take lesser time than this, so it will depend on the speed of the machine, so runtime also depend on the speed of the computer okay, so suppose I calculate my runtime and it is coming out to be this, say $100n^2$ + - 60n last three, this is my runtime, this many second and these are the factor, these are terms are coming from how much time we are spending in the add, adder, microprocessor, hitting time everything.

So we, we just calculate those and this is the exact runtime for my code and suppose my friend is having another code which is having run time like this, $2n^3 + 6n +3$, my friend also calculated using the, the time spending in the hardware and got this much second, now which one is the better, this is my code this is my friend code so if you draw this with respect to n so one will be like this, another will be like this okay. So this is, so up to this, so this is my code T_1 and this my friend code $T_{2,}$ so we will see after certain time up to a certain point my friends code looks like better.

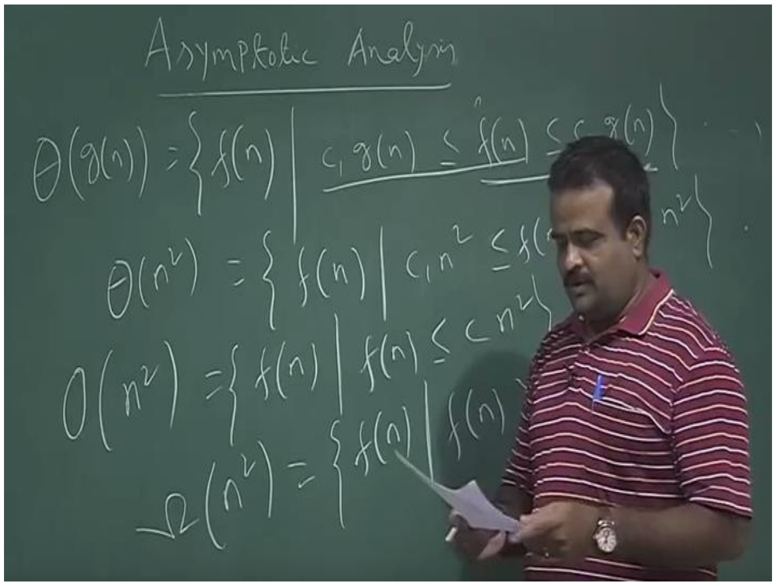
Because this constant factor is less but actually my code is better, it is having order of n^2 so that is why we need to bring a great idea which is called asymptotic notation so we, we want to get rid of this constant term which are basically missing, missing independent terms so if we how, how we can get rid of, we just ignore the constant term, we can ignore the lower term and then we ignore the leading coefficient so this is basically order of n^2 and this is order of nq, so this better and this is in the asymptotic notation.

So we will formulate define per term in my order the big the big Ɵ but this is the idea, so we want to get rid of from the missing dependent constant, missing dependent terms so that is the reason, so this would be the better code than this one. Okay, so let us define what is.

So what is we bigger of weak Θ asymptotic analysis we will do, so we will ignore the missing independent time, missing independent constant. Suppose our time is a $100n^2 - 10n + 3$ so these are the constant coming from the this mini seconds, coming from the miss independency so we want to get rid of this constant so we want to see the growth of this n as growth of Tn as intending to infinity.

So this is basically, we will denote these by order of n^2 so we will ignore the all loaded term, then we ignore the leading coefficient this will give us the big $\Theta(n^2)$. We formerly defined what is Θ.

(Refer Slide Time: 18:15)



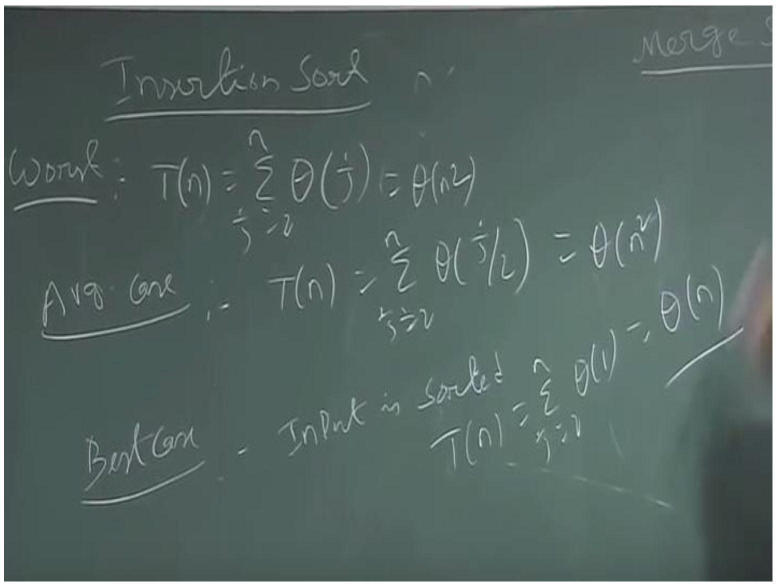
Asymptotic Analysis
$\Theta(g(n)) = \{ f(n) \mid c_1 g(n) \leq f(n) \leq c_2 g(n) \}$
$\Theta(n^2) = \{ f(n) \mid c_1 n^2 \leq f($
$O(n^2) = \{ f(n) \mid f(n) \leq c\, n^2 \}$
$\Omega(n^2) = \{ f(n) \mid f(n)$

Okay, so now, so Θ (g(n)) is the set of all function f(n) such that there exist two positive constant c_1 and c_2 and f(n) is bounded by this. So that means Θ (n^2) if g(n) is n^2 is basically set of all function f(n) such that
$c_1n^2 \leq f(n) \leq c_2n^2$.

So now this is the Θ, now O is basically, $O(n^2)$ O is basically the upper boundary if you have only this then it is call O. so $O(n^2)$ is set of all function such that, such that f(n) is ≤ some positive constant cn^2. Similarly $\Omega(n^2)$ is the lower bound, if we have only this then it called Ω of g(n) and g(n) is n^2 so this is basically set of all function such that f(n) is $\geq$ some constant $n^{2,}$ positive constant. So this means we want to look at the growth of this function as ended into infinity.

So we will now look at the few terms lesser, hello we want to see quite this T(n) is going as intending to infinity, so that is in asymptotic science.

(Refer Slide Time: 20:03)



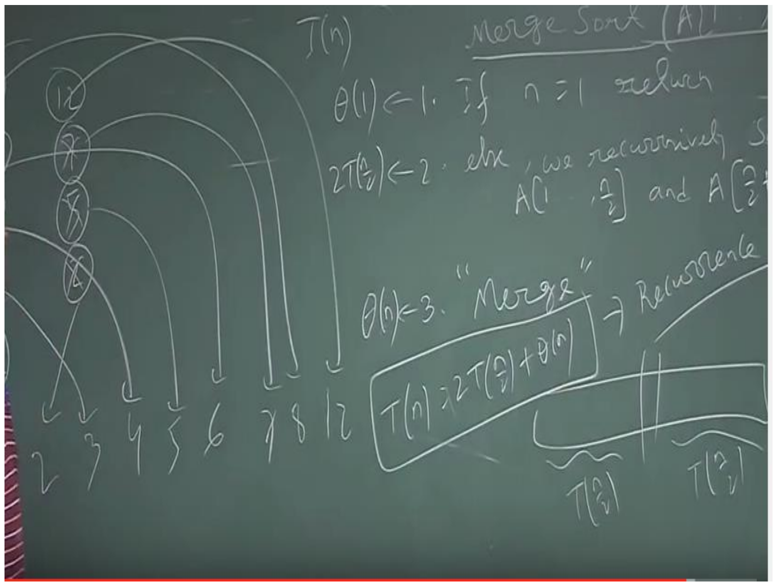
Insertion Sort
Merge S
Worst: $T(n) = \sum_{j=2}^{n} \theta(j) = \theta(n^2)$
Avg. Case: $T(n) = \sum_{j=2}^{n} \theta(j/2) = \theta(n^2)$
Best Case - Input is sorted $T(n) = \sum_{j=2}^{n} \theta(1) = \theta(n)$

So what is the worst case, so insertion or worst case means the input is reverse sorted, if the input is reverse sorted then we have a follow and everybody has to come forward. So the runtime is basically quarter of, so this is basically order of n^2. Okay, so this is the worst case, now what is the average case, so in the average case we are assuming that on an average everybody has to come half way to the beginning, I mean half way.

So T(n) will be j to n, so this is also will be order of n^2, this is the average case for insertion sort. But for average case we need to take the forward distribution of the input and then you need to do the proper expectation of that run time. But this is in a informal way to say the average case for insertion sort. Now when is the best case for insertion sort, best case is the if the input is already sorted, that means, that means you are comparing each element once only, so it is basically T(n) is basically then ∑ of, so this is a best case for insertion sort.

Now we look at sorting algorithm where we will see the average case is better than n^2 which is n log algorithm that is marge sort. So we have an array, upsize in, so in the marge sort what we do.

(Refer Slide Time: 22:25)



So if n=1 we return because all the one element so one element is already sorted nothing to do. Otherwise, we go into the middle, we go into the middle n/2, I mean ceiling we have to put to make it integer and then we sort this and sort this recursively. Okay, so this is a recursive call so else, we recursively, recursively sort, we apply the same marge sort on A 1 to n/2 and A n/2 last 1 to n.

So we have basically we have sorted this array, this array. Now once we have two sorted array now we need to merge these to get the whole sorted array, so we apply here a merge subroutine which will take two sorted array, sub array and which will give us the whole sorted array. So what is the merge subroutine, we can, we have two sorted array say 8, 6 ,4, 3, 1 say, and we have another sorted array 12, 9 like 12, 7, 5, 2 say, we have two sorted array, now we will explain the merge subroutine, so what we choose, we choose the minimal smallest element of these two sorted array. So smallest element will be the first element, so we point this smallest element of these two sorted array and we take a auxiliary array of size in to store this.

Now we choose the smallest among, which is the smallest one among the smallest, so 1 is the smallest so we output 1 and next we point to the next smallest element of that array. Now we compare again that these two smallest and 2 is the smallest, so we output 2 and we choose the next smallest. So we compare these two smallest, 3 is the smallest so we output 3 and we point to the next smallest so we compare these two, 4 is the smallest.

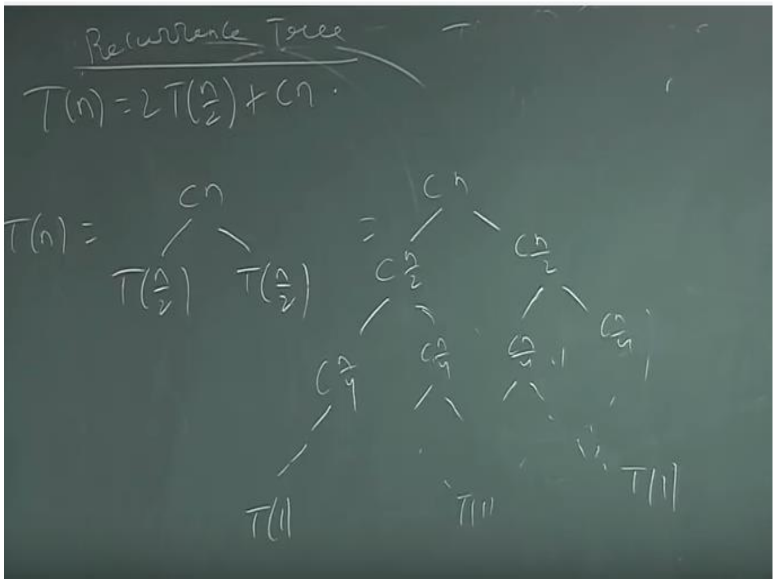
So 4 will be the output and we choose the next smallest, so if we compare these two smallest 5 is the smallest 5 be the output here and we point to the next smallest. So now 6 is the smallest 6, so 7 and 8 so 7 is the smallest so 8 and 12, 8 is the smallest. So only one so this is 12, so this is the, and for these we need to have array of size and to store this, okay. So this is the execution of the merge subroutine.

Now how much time we are spending here? Suppose we have a roughly n/2, n/2 element so we are comparing the each element one time. So it is basically a linear time algorithm, okay. But here we need a extra array, that to be noted, okay. Now what is the time complexity for marge sort, suppose T(n) be the time for this so how much time we are spending here? It is the signaling constant time.

And how much time we are spending here? So we are doing two recursive call, we are calling the marge sort here, marge sort here, so this size is n/2 so time will be T(n/2) this is also T(n/2) but here we, we may use the lower sling or upper sling but this is equal to 2T(n/2), so the total time is $T(n) = 2\ T(n/2) + \theta\ (n)$. So this is what is called recurrence of this, recurrence of this code, of this marge sort.

Now the question is how we can solve this recurrence? So one way to see the solution we can use what is called recursive tree or recurrence tree.

(Refer Slide Time: 27:26)



Okay. So this is our recurrence T(n) = 2 T(n/2) + cn that θ we can just put it at cn, so now what we do so T(n) is the basically, can be written as cn, T(n/2), T(n/2). So these are the time required to solve these two sub problems and this is the combined time.

Now again if we assume the same recurrence we will use again, so this is a share problem of size(n/2), again we are dividing this problem into (n/4), (n/4) like this, so this will be cn → c(n/2) →T(n/4), T(n/4), c(n/2) →T(n/4), T(n/4), so T(n) is basically sum of all these notes, so this way we continue, so c(n/4) until we reach to a T(1), if there is only one element we stop there, that is the T(1).

C(n/4) like this every branch we will stop at T(1), c(n/4) sorry this is c(n/4) like this. So this is c(n/4) like this, every branch is stopping at T(1), okay. Now the we have that total time T(n) is the sum of this all time so this is you can do the level like sum cn, cn and so everywhere it is cn, so what is the height of tree? Height of the tree is log (n) because it is a complete binary tree, so the time is, so the time for marge sort is n log(n).

So this is the time complexity for marge sort and this we are achieving using the this recurrence tree, in the next class we will see some other method to solve the recurrence other than the recursive tree like marge sort method, and so we will discuss in the next class, thank you.