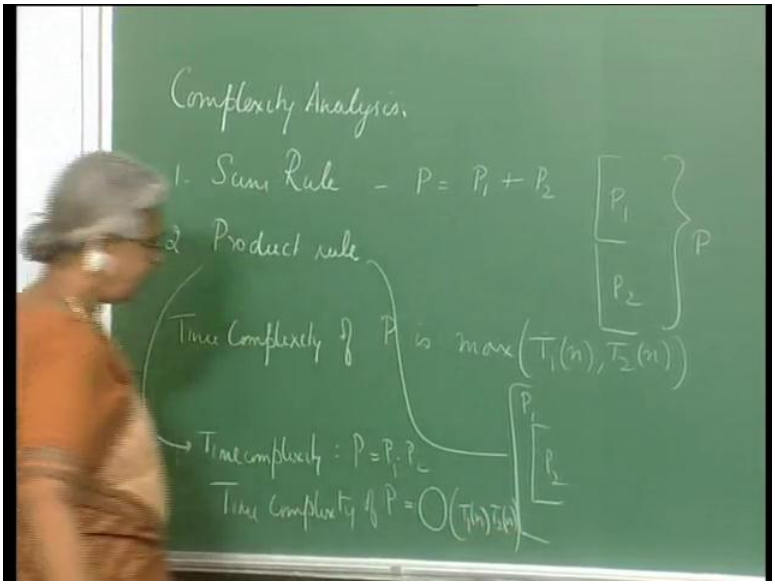
Programming, Data Structures and Algorithms
Prof. Shankar Balachandran
Department of Computer Science and Engineering
Indian Institute Technology, Madras

Module – 03
Lecture - 26
Example of computing time complexity using sum/product rules
Computing time complexity in a recursive function,
Example: Factorial Example: Compute time complexity of sorting and searching (linear versus binary)

(Refer Slide Time: 00:16)

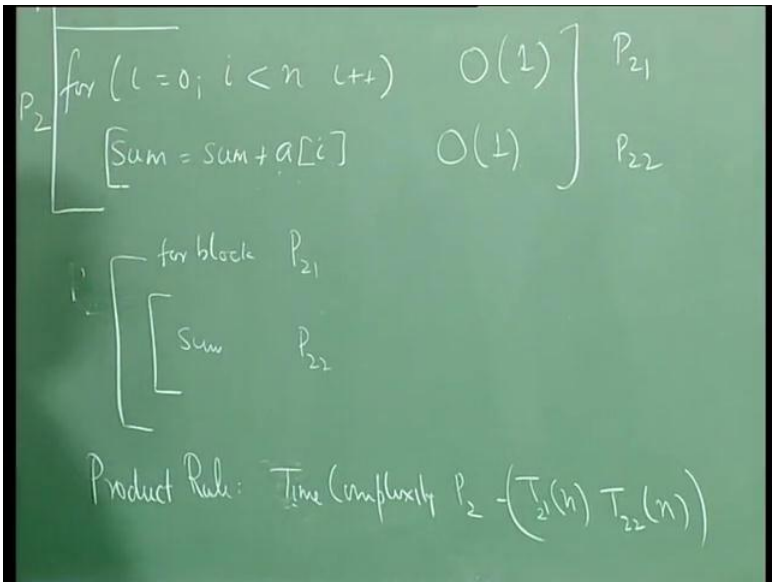


So far what we have done is we talked about two rules for complexity analysis very two very simple rules; one rule is what is called the sum rule, and other is call the product rule? What is the sum rules state? It my program P is made up of two program segments; P 1 and P 2. What do we mean by this have P 1 followed by P 2, and this is my entire program P, then the time complexity of the program P is max of be time complexities of P 1 of n comma P 2 of n. We have already saw where n comes from, n comes from the input to the program, it may be size or the value of the input itself.

Now the second rule is a product rule. What is the product rule state? It is says that if I have a program P 1 like this, and within this I have a program P 2. What is the meaning of this? It tells me that P 2 is executed P 1 times, then the product rule the time complexity is, time complexity that is what is that we are saying the program is made up

of P 1 times P 2, and therefore the time complexity of P - P is equal to the product of P 1 of n into P 2 of n, this what we saw in the last lecture. So, we are two rules: one is sum rule, and other one which is the product rule. Now what we will do is will go through an example and see how these rules can be applied. So, what do I have there, I have a simple program let me write just re write it here it.

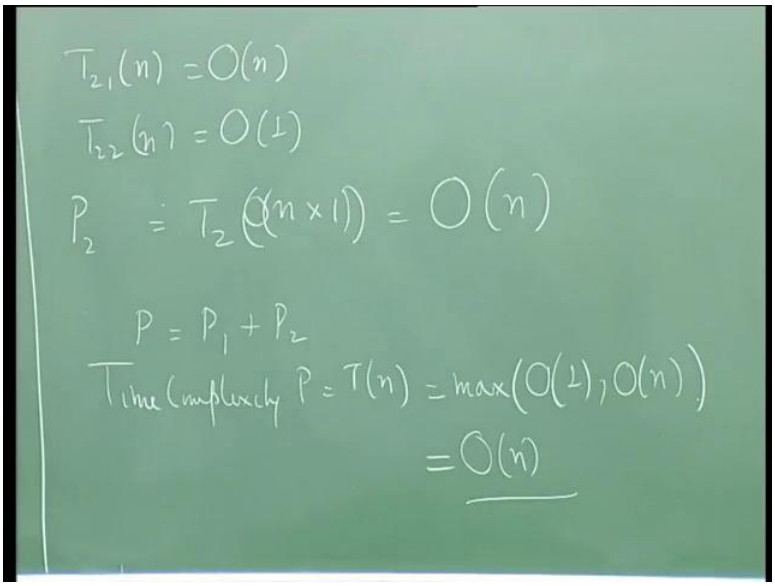
(Refer Slide Time: 02:4)



It simply computes the some of the elements in an array. So, it is says sum equal to 0 and for i equal to 0, i less than n, i plus plus. We compute sum is equal to sum plus a of phi, a is the array over here, we computing the some of n elements in the given array a. Now let us look at this program. Sum is equal to 0 is an assignment statement, and if I look at this entire program here I would say this is made up of P 1 and P 2. Now when I look at P 2 by itself notice that this is again made up of there is a loop here; that means, this statement is executed n times. So, here in P 2 to analyze the time complexity of P 2, I need to apply the product rule. To analyze the time complexity of P 1, I am simply use the, but some rule. So, to for this one for the example, if I look at this what is this statement here, i is being initialized, i less than n, i plus plus. So, what ((Refer Time: 04:07)), the worst case I will do three operations; initialization, comparison, and increment. So, this once time complexity is order one, because if max of all this all of them are constant time operations, and this is also constant time operation, but what are

we seen the P 2 now. P 2 is made up of this segment, that is this for loop is executed, this executes this statement n times. So that means what, the time complexity is of P 2 is n times means what, if I look at the if let us say P 2 is made up of two parts; P 2 1 and P 2 2, where P 2 corresponds P 2 1 corresponds to the for block and within that I have this sum which is computed. That means, what P 2 1 and P 2 2, this is what it corresponds 2; P 2 2 is executed P 2 1 times. Therefore, now I have to apply the product rule. What is the product rule tell me now? The product rule tells me that the time complexity of P 2 of P 2 is the time complexity of P 1, time complexity P 2 1 times the time complexity of P 2 2. So, since the for block executes n times, it is time complexity is n, notice that the increment is you know i increases by 1 during every iteration. Therefore, this loop executes n times.

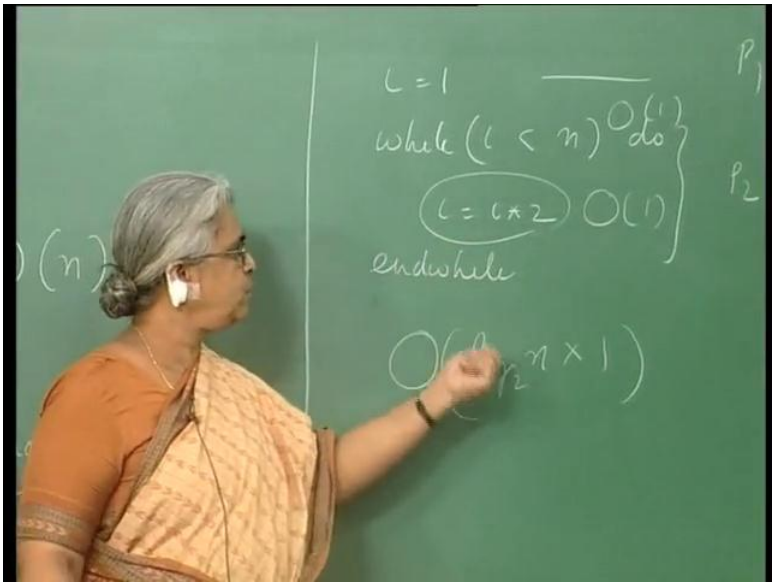
(Refer Slide Time: 06:01)



Therefore we have T 2 1 is equal to order n, and we already saw T 2 2 it is the max of these two is order 1. Therefore, the time complexity of the program segment P 2 T 2 is equal to n into 1 which is order of n into 1 which is equal to order of n. Now what do we have? We have the program segment P being made up of P 1 plus P 2, and therefore the time complexity T is now of T which is equal to T of n, because now I can apply the sum rule. Therefore, this becomes max of order one which comes from here program segment T 1, and order n, and therefore the time complexity is order of n. So, this is an example

of time complexity. So, what have we done? We have used both the sum and the product rules to compute the time complexity. Let us now this is fine as for as programs with that are having regular for loops are concerned.

(Refer Slide Time: 07:45)

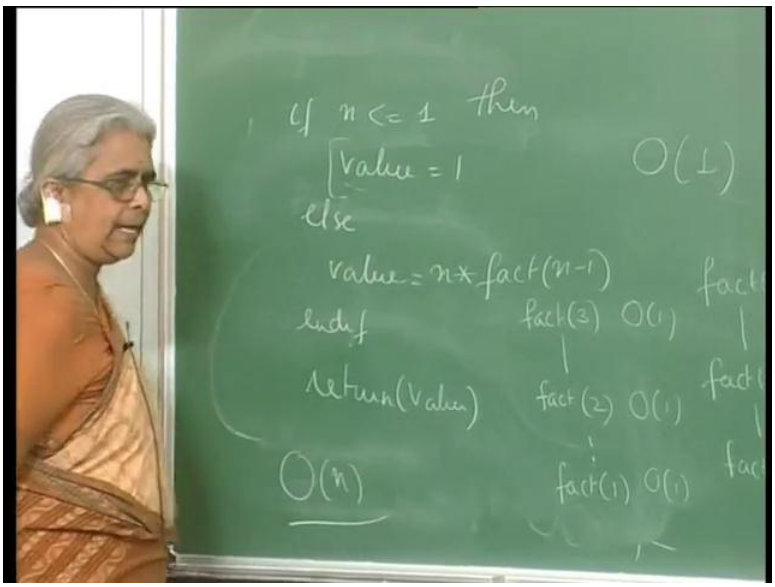


Now, if we looking at something else, let me leave you with this example. Suppose I have i equal to 1 while i less than n, do I am writing some ((Refer Time: 07:57)) here, i is equal to i star 2 end while, now again there is of the form, this is program P 1, this is P 2, again P 2 have the loop, but how many times that this loops executes, basically notice that i is getting double every time. So, if I say n is equal to, let us say 8 then i equal to the first time I have i equal to 1, then the second time I have equal to 2, i equal to 4 and the third time i equal to 8. So, if we look at it, it basically goes through this, this part of the program is executed only three times, and therefore in terms of n if you look at it; that means, how many times is a loop executed, it is only executed login times. That means, now and the cost of this statement over here is order one, this is also order one, but the loop is executed log n times, therefore the time complexity becomes order of log n. So, this is how you compute time complexities using the product, and the sum rules.

Now this is all fine as long as there is a loop like this which you can you know all that we have done is we have kind of un ruled this loops ((Refer Time: 09:19)), how many

times if the loop the executed, this is what we are doing here? And multiplying the number of times the loop is executed with a cost of the operation within the loop, there is how you defined time complexity over here. Now let us look at another example. This is slightly more complicated example, what happens when we have recursion, here is an example with recursion a computing, the factorial of a given number, let me rewrite this over here is saying. So, is basically like this the int fact over here, and what are we doing here? Again we can apply the same which but the only difference over here is that, we need to find out how many times the recursive call is made? And that will decide the cost of the computation.

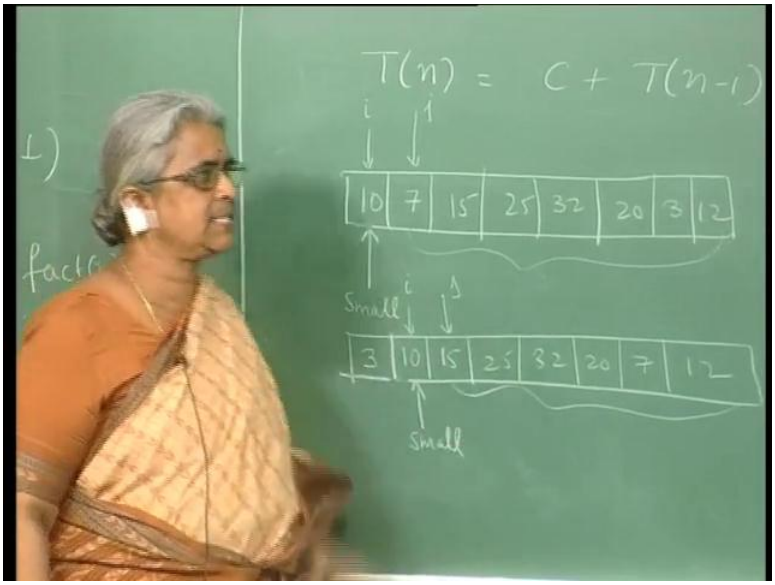
(Refer Slide Time: 10:20)



So, if you look at it over here. So, what is it that I have we have fact, and we are saying some if n less than or equal to 1 then we are returning 1; values set to one in that is what we have written. That means, that is the only statement that is executed in the recursive function else value equals n star fact of n minus 1, and if here and we are returning value, this is what the function is return. So, let us look at this recursive function? What is the various thing that what have been done this statement is very harmless, what do we have here? We making one comparison here, and the cost of the operation, therefore this will be atmost order 1; that is then n becomes less than or equal to 1, the cost of the computation is only order 1, but when n is not equal to1, let say n is greater than 1 then

what is that that we have? We have n star fact of n minus 1. That means, what I have fact of n, now this is calling fact of n minus 1 which is calling fact of n minus 2 and so on. Let us say I want to compute the factorial of 3, then the fact of three is going to incur; that means, what is that I am doing, within this now there is one multiplication over here. The cost of the multiplication is essentially going to be order 1, and the assignment for that matter is going to call order one here, and fact two again order 1, and so on fact to one which is again order 1. So that means, what to compute the factorial of 3, I do 3 times order one operations. So, the recursive call essentially tells me that is going to three times order 1, n is a number over here which is this input, therefore the time complexity of this is going to be order n. And what we can do is what we have written there basically if we look at the particular loop here, which as order of 1 plus plus T of n minus 1. This is the way we write the time complexity for recursive functions. Let see what we do over here. So, what we do is to give you a nice simple way of doing it. So, we are saying there is some constant cost over here plus the cost of T of n minus 1, where this is the cost of the recursion. If n is greater than 1, it is equal to d just one some other constant cost, if n less than or equal to 1. So in general define at the ith iteration, ith recursion then T of n and what we done i's i time c plus T of n minus i, n is greater than n. Then i equal to n minus 1 what is happening, then there is only one c of… So, basically what happening i cost c of n minus 1 plus d. d is the cost of the last call to the recursion. And therefore, the time complexity becomes order n. I must one new this is not a nice algorithm to compute the factorial, because it is two expansive, but never the less this illustrates the problem of how complexity analysis can be done for recursive functions. So, this is…

(Refer Slide Time: 14:05)



So, basically in when you have recursive function the most important point is that, we write it we say T of n will take one more example, and a little while and we write T of n and both on the left hand side, and the right hand side in this particular computational just C plus T of n minus 1. So, you will find T of n is equal to define in terms of T of n minus 1 and so on, and you can solve this, you learn more about solving the time complexity for recursive functions in the next course, an algorithms where it will regress way of finding the time complexities will be given.

So, we leave it here and now what I want do is, I want to take few more examples on complex, so this is so that I am sure that your absolutely clear about what is being done and let us look at this example of sorting and elements. So, I have a set of n elements by this; 10 7 15 25 32 20 and 3 12, that is given as input to my sorting algorithm and I want to get this output which is a sorted array.

On the other hand if I am searching what am I doing, I am given sorted array, and I am let us say I am given the key equal to 12, and output should give me index of the where this element can be form. So, these let us look at these two problems. So, let us look at

sorting. Will take a very simple selection sort algorithm, what the selection sort algorithm? It is a quite an inefficient version of the selection sort algorithm that I have written, all it is doing is it take is array over here, and what we does is let us look at this example again I have an array with something like this. What do I have? I have 10 7 15 25 32 23 and 12.

So, what is it do? When I look at a first loop, let us look at this I will ((Refer Time: 16:09) what is doing, its assigning something call small to this element, i is pointing here and j is now pointing here. Then what it is do? It goes through this entire list and keeps on exchanging then the swap function I just use the swap function from the start very efficient to do it which should actually find the minimum in the array from I plus 1 to n and replace exchanging the 2 of them. So, what is it do now it comes keep on exchange this, 7 7 will be exchange then I can 7 is in the beginning here, 7 and 15 is find 7 is smaller than 15 remain as it is. Then it will compare 7 with 25, 7 with 32 and then comes with 3 here, and ultimately what we will have it is something like this at the end of the first loop.

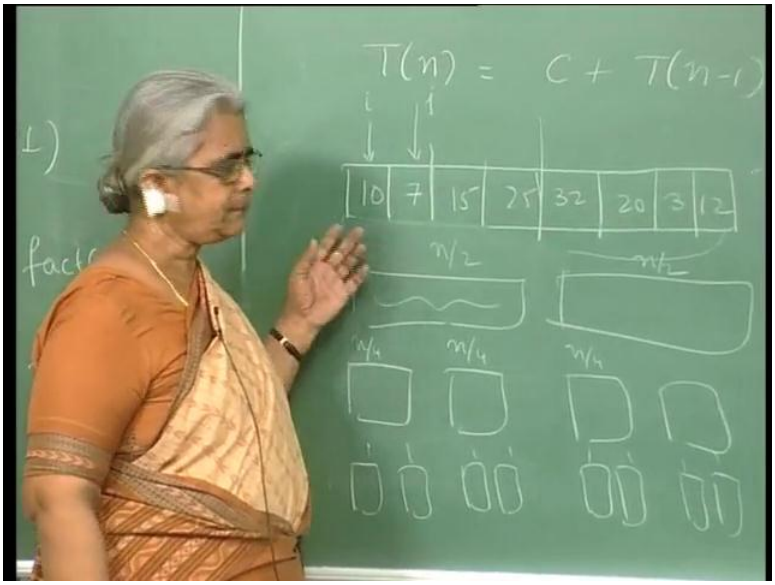
So, I would have here 3, I will have 10 here, I have 15 here, 25, 32, 20, 7 and 12 after the first pass. So, what are we done one execution, that is one's I have gone through the entire array, the most minimum element is that the beginning. Next what we do, we make this small and this is i, this is j, and then we compare again find the smallest element smallest element is put he, and this is repeated until we are exhaust the array. Now when I look at the time complexity this program, this program is of the form P 1 into P 2 into P 3. What is P 3 now? P 3 corresponds to this int this loop over here. So, there are two for loops; one for loop is inside the another for loop, each for loop is executed n times. So, now let us look at...

So, the for loop with j as index is inside the for loop with the index i, P through corresponds to the segment inside the second for loop, P 1 and P 2 corresponds to the i and j loops respectively. So, that means what do I mean over here going back to this. So, this corresponds to P 3, this corresponds to P2, and this corresponds to P 1. And we will P 1 again, there is one there is a some rule small plus this loop. Therefore, the time complexity become an order n now by now your quite familiar with this, this loop

execute n times, the cause of this operation is order one. therefore, this is order n that is within the for loop, and then this for loop is again execute n times therefore, this total time complexity this comparison operation cause order one. So, this becomes within this for example, if I look at the body of the ith for loop, the time complexity of the ith for loop is basically order means, our body of the ith for loop is order n.

Therefore, order n multiplied again this executed n times, therefore the time complexity of this becomes order n square which is what i summarized here. The for loop of j index is inside the for loop with the index i, quickly correspond to the segment inside the second for loop; P 1 and P 2 corresponds to i and j loops respectively. So, basically you are executing two for loops, therefore the time complexity is order n square. Now let me leave you with another function which was a nice is a which is a interesting and efficient more efficient sorting algorithm then the input permutation is very quite random, and this is called the quick sort algorithm. The quick sort algorithm you learn on the algorithm scores and what is interesting over here is, this is a recursive function and what does the quick sort algorithm do is simply take this an array over here divides it into two parts, and then recursively sorts the element, and what happens is goes on until that is divides into two parts and I am divides into two parts, what is it ensure? That elements on the left side of the middle element as smaller and the elements on the right side of the middle elements are larger, the first part is will do that. The next part what will do recursively do this again for this part, and in the best case if you a middle element is actually the median in the array at every time.

(Refer Slide Time: 20:57)

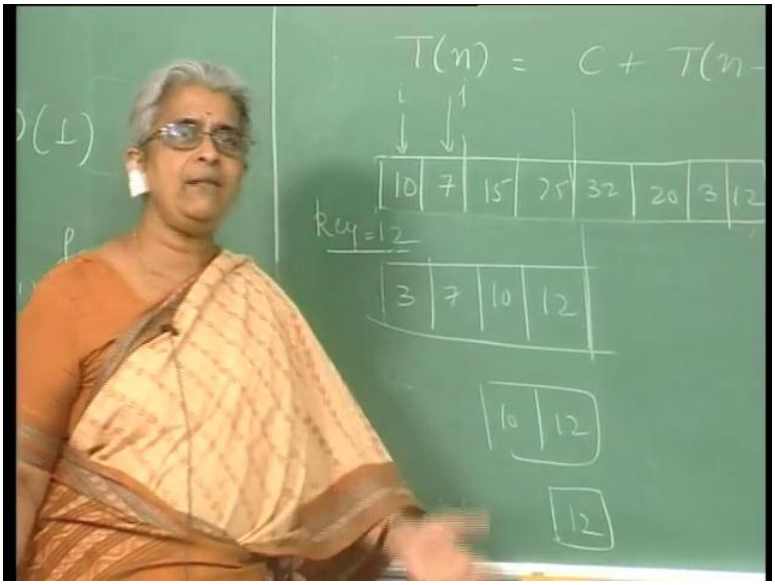


What we will find is that the depth of the recursion notice that in the program quick sort is being called again and again it may be atmost, so I have two labels like this, next it will become items of two elements long, finally you will have items which are one element long. So, the depth of the recursion is only log n. But in each recursive call all the n elements of process, because you have one part here, this is n by 2 element being process, this is n by 2 elements being processed, and this is again n by 4 n by 4 and so on in this particular case. Finally, you have one element being the process, this the best case. Why do I say called is best case, that is an every recursive call, we assume that the middle element corresponds to the median element in the array. Then the number of recursive calls, there are required is at most log n to the base 2, but in each recursive call one call is doing n by 2 elements here, and other call because the recursion recursive call to n for example, that is makes two recursive calls over here, both on the left and the right, therefore always n elements are looked at, therefore every time doing some n comparison. So, n into log n is the time complexity of this algorithm. So, this have you do these operations. All assignment statement, expressions, and so on so forth, the essentially cost order one. So, each recursive call divides the array into this is the best case that I am talking about, in each call all the n number are compared recursion terminates when the array size becomes 1. And therefore, the number is a recursive calls is log n in the best case, and order n log n is a time complexity for the best case of this

algorithm, I encourage you to go back and verify this. I also want to verify that the time complexity only given for the best case is also true for the average case; average case ((Refer Time: 23:04)) about is little more involved, you will find that in the worst case what can happen is that? The array is divide into two parts such that; one part is the size n minus 1 and other part is the size 1 and if it happen recursively then the depth becomes n and you will get a n squared worst case algorithm, I want you to verify this case. Now let us look at searching, now let us do simple thing is next do linear search. Now this is again a short of inefficient algorithm and just I assuming that.

I have a set up n elements which are not necessarily sorted, and all I am going to do is I am going to search through this I have an array like this, and search through the array to find whether there the whether given element is present or not, this is very straight forward. So, what is that we are doing over here, this the initializing flag found to 0, and while less than or equal to n and and not found, if array of i equals k found equals 1, otherwise return a found. So, in the worst case the time complexity that is if the element is the last element I am looking for 12 as I have looked here, then linear search essentially searches through the entire array, and determines whether the element is there or not, therefore the time complexity can be ordered n, it is a very, very straight forward algorithm. Now what I am going to do is am going to do one more algorithm ((Refer Time: 24:40)) which is called binary search, and in binary search that as you assumption is that we have a sorted array, and in this example what would be my sorted array, this could let us assume that use the sorting algorithm which I have already have and I am going to perform search on this search on this.

(Refer Slide Time: 25:00)



So, this will be 3 7 10 12 15 25 20 25 and 32. So, let us see this the output that I have. Now I want a search for this element 12. So, what is done in binary search is somewhat difference. So, because it is already shorted, all that we need to do is we divide the array into two, and find out now what is the given, the key is given the difference is I look for the element key equal to 14. Then I compare with where this key might be present, key equal to 12 is work it you mean. Now I check whether it can be in the first part of the array or the second part of the array, then what I do is I search depending upon which will part it my belong to, then we can do this recursively again again divided into two parts, that is what that algorithms is doing. So, what is doing is, if key key is greater than array of middle then it searches from middle plus 1 to upper, otherwise it searches from lower to middle.

So, what is initially middle now, what is lower now, lower is the starting point of the array, upper is the end point of the end index should be array. So, it is searches in the appropriate portion for the key. So, what is happening is every time the search space is getting reduced into half. So, that I am looking at 12, the next time I am only searching in this part. Next time I am searching only in this part, and then finally I am searching in this. So that means what the array size which was 8 long becomes 4, 2 and 1; and when I ((Refer Time: 26:49)) 1, then if there element is equal to the key that I am searching for I

have found the element and this the essentially the idea mind research. So, now, if you look at it, what is the time complexity of it, notice that in the first case I come made one comparison, every recursive call does only one comparison. Therefore, that is constant time, therefore the total time complexity of this is, because in the recursion I am running it up to log n times in to 1, and therefore the time complexity is order of log n to the base 2 for the binary search algorithm. And this is what I have summarize to become.