Introduction to Problem Solving and Programming

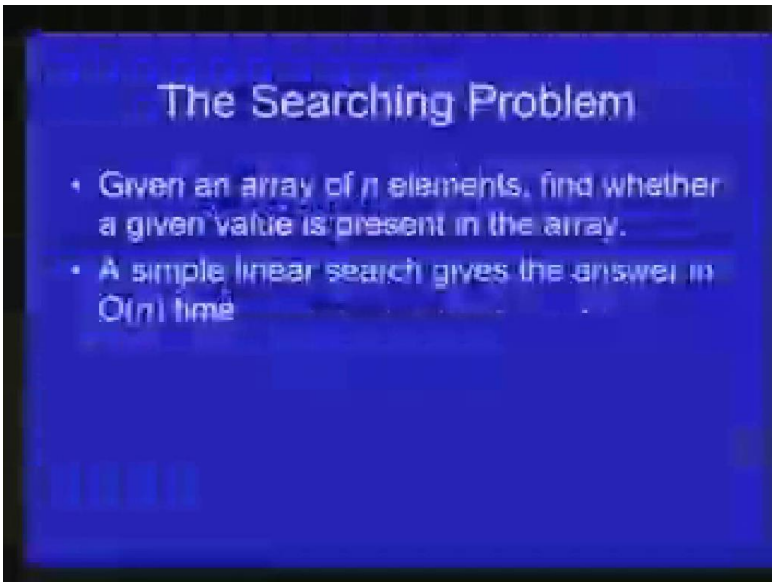
Prof. Deepak Gupta
Department of Computer Science Engineering
Indian Institute of Technology, Kanpur

Lecture No. # 22

(Refer Slide Time: 00:27)

In today's lecture, we will talk about two very important problems which find applications in most application areas. These are the problems of searching and sorting, I will explain what these problems are and will try to develop and analyze some algorithms for these problems.

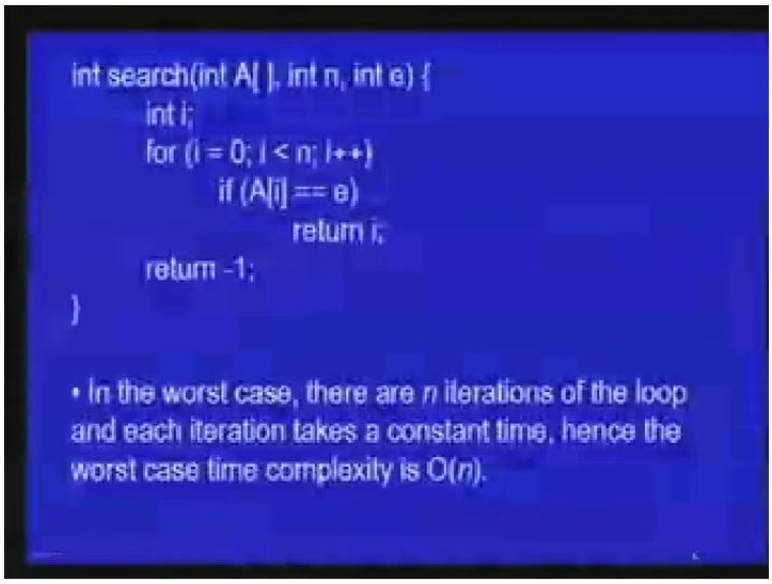
(Refer Slide Time: 00:38)



So, let us talk about the searching problem first. The problem is very simple we will given an array of n element. These could be of any type really and we need to find out whether another given value is present in the array or not. And let us say if it is present in the array we return the index at in the array at which the element is present or if it is not present in the array we can return minus 1.

Now this problem is off course very simple as you might imagine all one as to do is to look at each element of the array one by one, and compare it with the given element and if there is a match if the two values are equal, and we have found the element in the array and we know what place in the array it occurs and we exhaust all the elements of the array then clear the element is not in the array. So, this algorithm is called linear search and clearly it takes time proportional to and to find the answer in the worst case, because in the worst case we may have to look to all the elements in the array to say to find before finding a match or before saying let the element does not exist in the array.

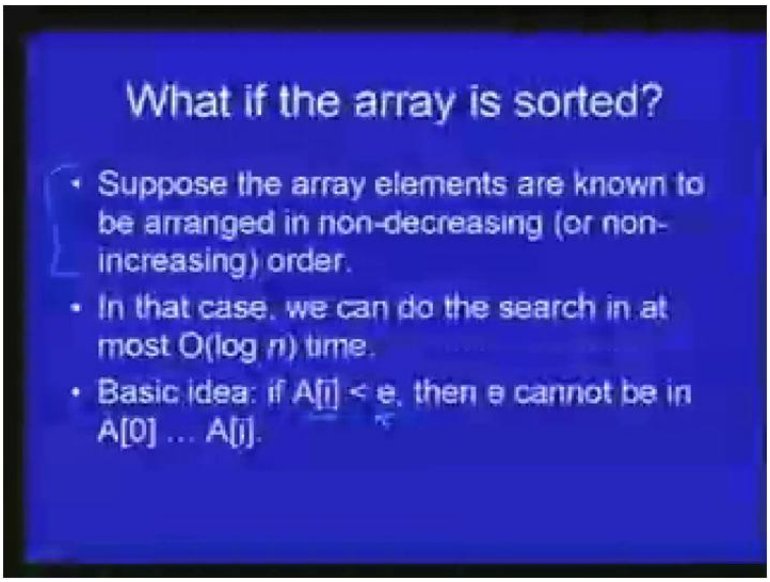
(Refer Slide Time: 01:47)



Let us quickly see the c function for implementing this algorithm. So, this is the function search, it takes as argument an array and the size of the array there are assume to be an element in the array and e is the element that we are searching for in the array A from A 0 to A of A of n minus 1. So, the algorithm is very simple we have a loop that trans for i from 0 to n minus 1 that is for each index in the array. And in each iterations we compare A i with e and if the value of A i is equal to the value of e, we return the index at which we found this element that is i and if this loop finishes that is (( )) of the loop; that means, that this return statement never executed and that happens because we never found a match and so therefore, we return minus 1 to indicate that the element e is not in the array.

So, you see that in the worst case there will be n iterations of this call loop and each iteration takes a constant time, and therefore clearly worst case time complexity of the algorithm is order n. So, the question is can we do better than order n for searching and the answer is no. If there is nothing known about the elements of the array because in the worst case we do need to look at all elements of the array at least once. So, therefore, we must do type we must do effort proportional to n, but if you know something about the array elements then we can do better.

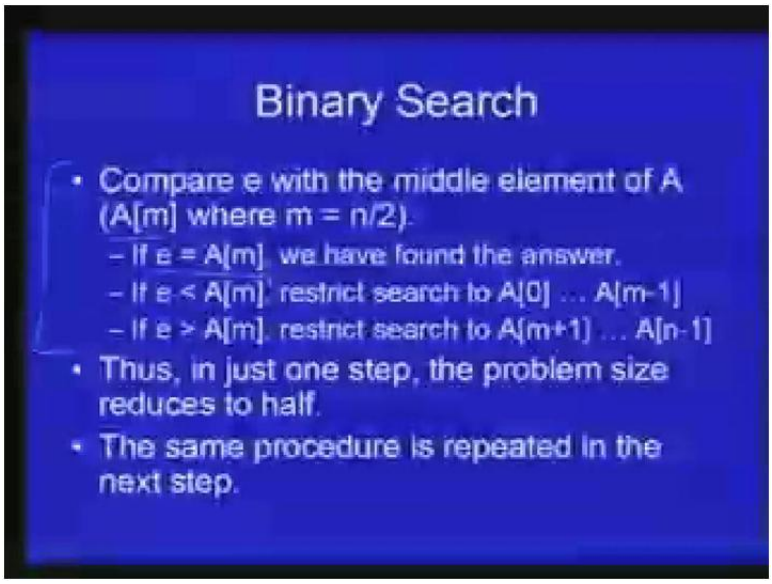
(Refer Slide Time: 03:19)



Suppose the array is sorted - that is the array elements are known to be arranged in non-decreasing or non-increasing order. Now if that is the case and that happens in many situation that the array is already in certain fashion then we can actually do the search in at most order log n time. Now how do we do that the basic idea is very simple is we compare the ith element of A with e and e happens to be greater than A i; that means, that e cannot be present from A 0 to A i, because we know that A i is less than e. And since if they are assuming that the elements are arranged in the non-decreasing order - that means, A 0 to A minus 1 they must also be less than e.

So therefore, we need to now focus our attention for searching only to the part of the array which is to the right of A i - that is from A i plus 1 to A n minus 1, and similarly if A i happens to be greater than e; that means, that e cannot be present in A i plus 1 to A n minus 1 because the elements are assumed to be arranged in non decreasing order. So, if A i is greater than e then A i plus 1 A i plus 2 etcetera are also greater than e and therefore, we do not need to look at all these elements to look at e.

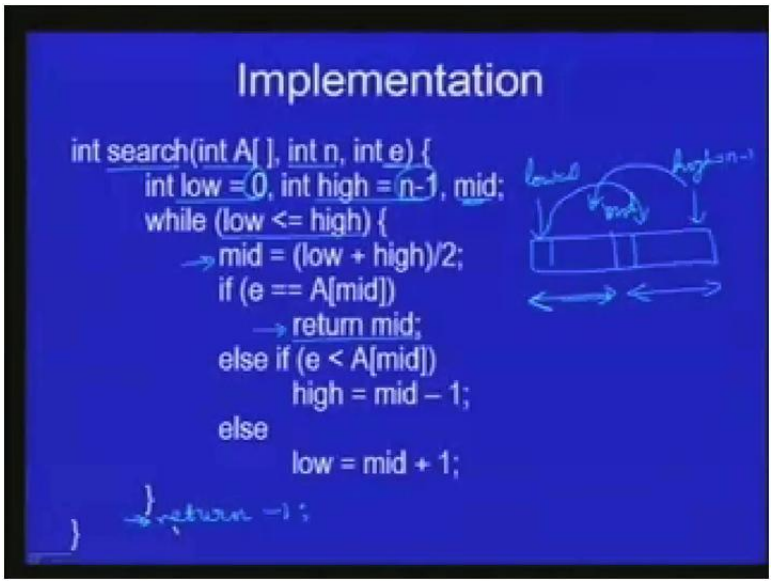
(Refer Slide Time: 04:42)



So, that the basic idea of the well known the binary search algorithm. And this is how the algorithm works. So, in each steps we compare e with the middle element of the part of the array that we are looking for that we are looking at. So, we basically compare e with A m where m is equal to n by 2 - that is the middle element of the array. If A e and m are equal then clearly we have found e in the array and we have found at index m, so m is the answer. If e is less than m, if e is less than A m then we restrict our search to A 0 to A m minus 1, because we know that e cannot be anywhere from A m to A of n minus 1. And similarly, if e is greater than A m then we restrict our search to be to the second half of the array that is from A of m plus 1 to A of n minus 1.

And so, in one step itself the problem are reduces to half, because in the next step we have to consider only half the array, and similarly after the second step will have only one-fourth of the array left that we have to search for e in. And if you keep repeating this process, no in each step we keep reducing the part of the array where we have to look at that we have to look at my half and. So, it is easy to see that in log n steps the array size will become will will reduce to 1 or 0 in which case the problem is trivially solvable in constant amount of time.

(Refer Slide Time: 06:22)



So, let us now implement this as the c function. So, again the header for the function is same we are given the array A beside n of the array A - that it has the elements from A 0 to A n minus 1 total of n element, and e is value that we are looking for. This variables low and high indicate what part of the array, we are currently focusing our attention on - that is this is the part of the array in which we need to search for e and as we as we saw when we discuss the algorithm in every step this part of the array gets reduced to about half of this size - that it was in the last step.

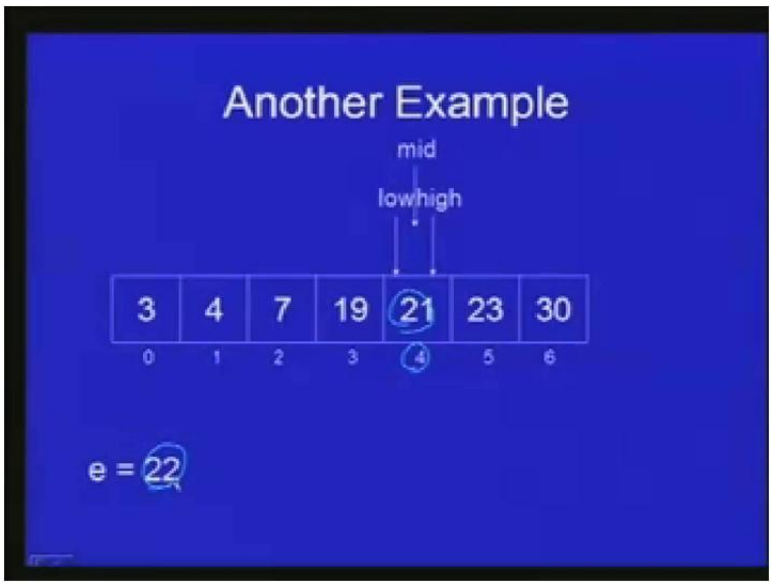
To begin with the entire array it has to be looked at, and therefore low is to be pointing very first element of the array that is low is equal to 0 and high is the index of the very last element of the array, which is equal to n minus 1. So, that is why low is initialize to 0 and high is initialize to 1 because we have to look at entire array from this is 0 to n minus 1. The variable mid will be used to compute the index of the middle element of the part of the array that we are currently looking at.

Now, the number of element that we need to look at is high minus low plus 1 clearly and so if low becomes greater than high then the number of elements that we that they are in the part of the array that we are looking at will become 0. So, in which case will found the answer if the size of the array that we need to look at becomes 0; that means e cannot be anywhere in the array.

So, we know the answer is, answer that we need to return is minus 1. So, therefore, as long as low less than equal to high which keeps repeating process. We compute the middle index of this array fragment and if e is equal to the middle element we return this index, otherwise if e is less than A of mid then so this is mid. So, if e happen to be less than A of mid; that means, e is to be find found somewhere in this part of the array. So, which means that high should the set two mid minus 1 where as low should remains same.

So, that in next step we focus only on this part of the array where as if e is greater than A mid then we need to focus our attention on this part of the array in the next step which means that low should become mid plus 1 whereas high should remain the same and when the loop terminate that can only terminate and if you reach the statement we should have A return minus 1 here which i omitted by mistake. So, this loop terminates and reach the statement; that means, we did not find the element anywhere in the array because otherwise we would have return from the function using this return statement and so if you come out of the loop then we know that the element e is not in the array and so we return minus 1. So, let us see A couple of example using animations to see how this algorithm is working.

(Refer Slide Time: 09:45)

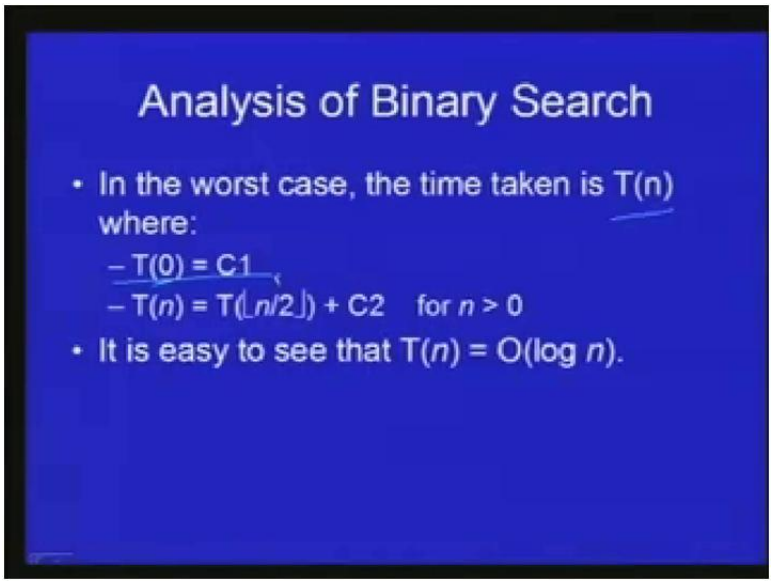


So, let us say this is the given array 3, 4, 7, 19, 21, 23, 30 and what we are looking for is 4 which of course we know it present here. So, initially low is 0, mid is 6, there are 7

elements in the array, I am sorry high is 6, because there are seven elements in the array and mid is 6 plus 0 by 2 which is equal to 3, so mid is pointing here. So, we compare e with a mid, now e is 4 and a mid is 19 and so e is lesser, and therefore in this step high moves to mid minus 1 and then mid becomes the new value of mid becomes low plus high by 2 which is 0 plus 2 by 2 which is 1. And so in the next step, we again compare a mid with e and this time they are equal because a mid is 4 which is equal to the value of e we found the answer and in this 1 gets it (( )).

Let us look at another example in this case we have same array, but we are searching for the element 22. So, now, 22 is larger than a mid which is 19 and so low will move to mid plus 1 and mid will correspondingly change, so low has become 4 and high is 6. So, the new value of mid is 6 plus 4 by 2 which is 5 and this time again we compare compare a mid with e which is 22 and we find 22 is lesser than 23. So, high will become mid minus 1, mid is recomputed low and high has become equal, but still the loop will continue because that the loop condition was while less than equal to high, low is still equal to high. So, this condition is true. So, mid is recomputed and it will turn out to be 4 again because low and high are both 4. So, we compare 21 which is a mid with the value of e which is 22 and e is greater than this; so that means, that low will become now mid plus 1 and one that happens in the next then the loop terminates because the value of high value of low has actually become higher than the value of high which means that the fragment of the array that we are now looking at has size 0, and an array of size 0 clearly cannot contain an element, and so the loop terminate and we return failure to find the element in the array.

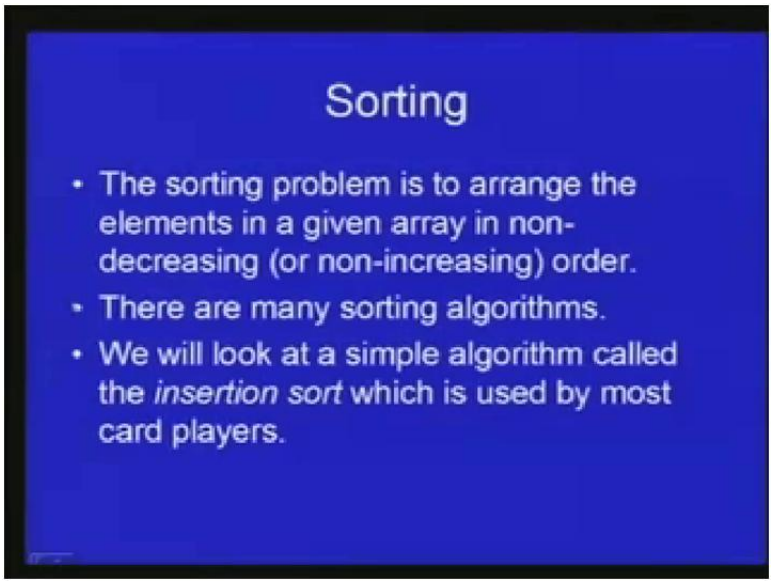
(Refer Slide Time: 12:17)



Let us now analyze this algorithm that we developed. So, let us say the time taken in the worst case is T n is the function of n which is array size and as for all recursive function as, this case we can again come up with the occurrence relation for T n, because in every step we are reducing the problem size by high by half. So, n is 0 which means low is actually greater than high then takes a constant amount of time C 1, otherwise T n is T of n by 2 plus C 2 because if the array size is n then in the next step the array size reduces to at most the floor of n by 2. There is not really a recursive function, off course we could have implemented the same algorithm recursively as well, but it is still easy to express the time taken as occurrence relation, we have done that.

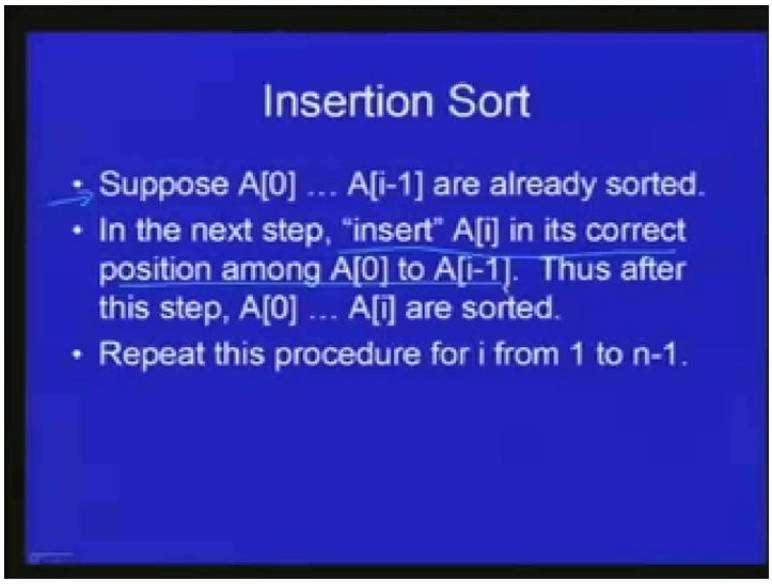
And this occurrence relation off course should be similar to you from the last lecture when you looked at the square and multiply algorithm. This is exactly same occurrence relation and as you saw last time we can solve it to obtain the fact T n is order log n which means that the time being taken by this algorithm e proportional to log n instead of n. And number of element of the array that we are looking at in the worst case is the log n instead of looking at all the n element.

(Refer Slide Time: 13:54)



Let us now look at now the second problem that we want to discuss today and that is the problem of sorting. The problem of sorting is simply to arrange a given array of element in some order may be the non-decreasing order or non-increasing order. We will consider only the non-decreasing order clear with algorithm will remain the same only the direction of comparison will really change and it does not really matter. There are number of sorting algorithm which have been developed for this problem what we will look at today is very simple algorithm called the insertion sort not the most efficient algorithm known for this problem. This is the algorithm which is commonly used by card players, when they have a set of cards in their hand and they need to sorted, take the next card and insert it in the right place in the cards prior to that. So, that is what the basic idea for the insertion sort algorithm also is so at any given step.

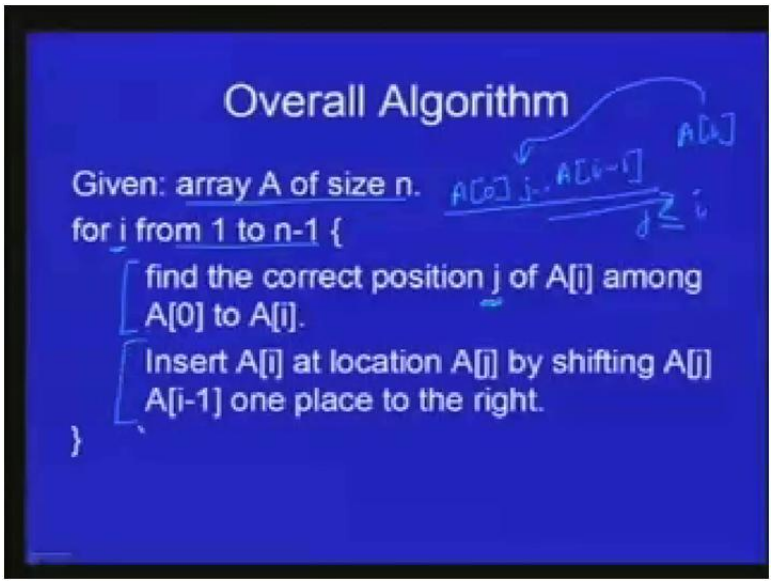
(Refer Slide Time: 14:47)



Let us assume that the elements A 0 to A i minus 1 are already sorted and to begin with of course if we set i to 1 then A 0 a single element is off course always sorted, if an array A sequence of one element; obviously, trivially sorted. So, to begin with this is true for i equal to 1, but as some intermediates step let us assume that A 0 to A i minus 1 are correctly sorted, but the rest of the element are not really in the correct places.

So, in this particular steps what we are going to do is to make sure that the size of the sorted sub sequence, becomes 1 larger that is at the end of these steps will achieve the condition that A 0 to A i are correctly sorted. So, to do that what we are going to do is to insert A i in its correct position among A 0 to A i minus 1, so that after this step A 0 to A i are correctly sorted and this procedure is repeated from i for i 1 to n minus 1. After the first step, the first two elements would get sorted, after the second step the first three elements would be sorted and so on. And so after the n minus 1 eth step, the all the n elements of the array would get sorted.

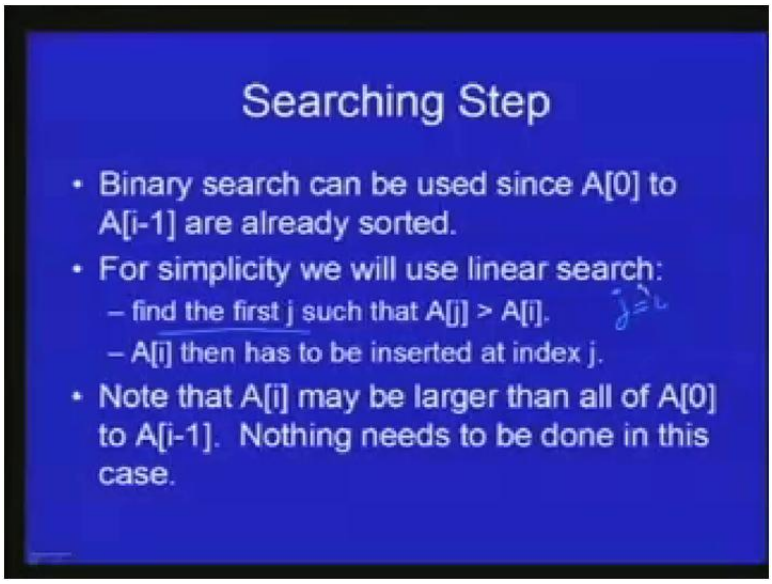
(Refer Slide Time: 16:10)



Before we write the C implementation of this algorithm, let us look at the algorithm at high level. So, we are given size n and what we do is we run A loop for i from 1 to n minus 1 as (( )), and in each iteration of loop we assume that A 0 to A i minus 1 are correctly sorted and A i is going to be inserted somewhere in its right place in the correct sorted order within in the sequence. So that the end at the end of this step the element A 0 to A i are all correctly sorted. So we need to first find where what is the correct position of A i in this sorted sequence, let us call that position as j that is A i should go at A j where j is less than or equal to i. And we insert A i at the location A j by shifting all elements which are to the right of A j by 1 position and then putting A i at the location A j this will become clear once we look at the details of the algorithm.

So, there are two sub steps here; one step to do this searching if this correct position and the second step to do this insertion.

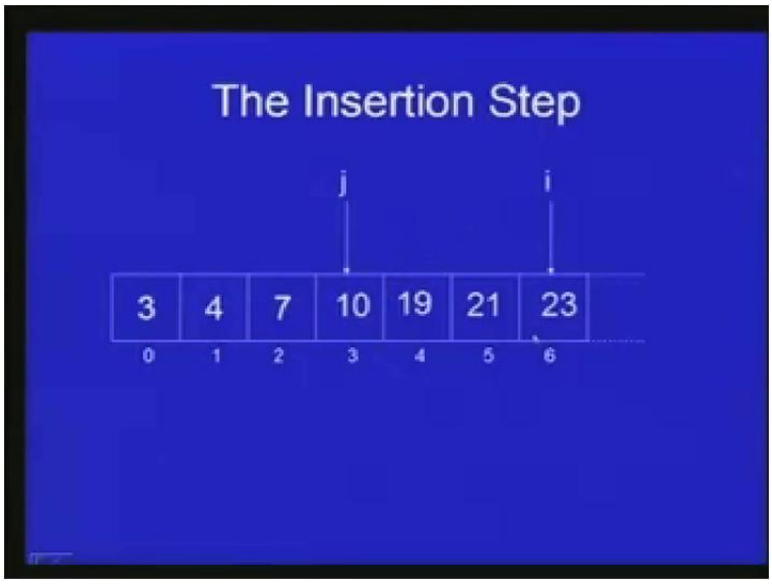
(Refer Slide Time: 17:31)



So, for the searching step we can use binary search because we already know that A 0 to A i minus 1 are already sorted, but in the program that we will write I will not do that I will use the simpler linear search, and I will leave it as an exercise for you to implement that use binary search which we have already seen. So using linear search essentially what we will do is we will try to find first j such that A of j is greater than a i which means that A of j minus 1 was less than equal to A i, but A j greater than A i. And therefore the correct position for A i in the sequence is at the position at the index j in the array. And so, A i would be inserted in the next step next sub step at the index j.

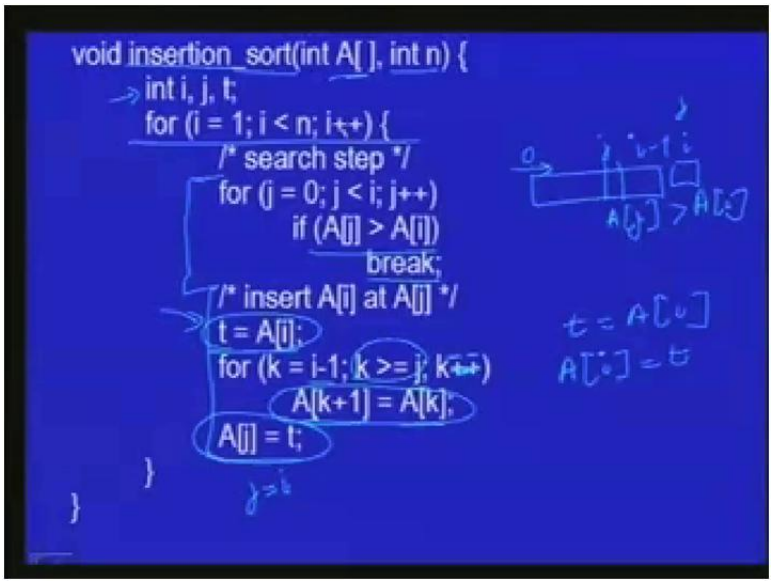
Note that it may turn off that j is equal to i now j is equal to i will happen when A i is larger than all the element A i minus 1 and that is it happens by chance that we sorted just first i minus 1 element and i eth element was already in its right place. So, in that case of course nothing is to be done, but you must ensure that are algorithm work correctly even in this case.

(Refer Slide Time: 18:54)



(( )) example of the insertion step. So, let us assume that the sequence is already sorted. So, i is 6 which means that the sixth element of the array has to be inserted at the right place and A 0 to A 5 are already sorted as you can see in this example. Now j is 3 in this example because 10 should really appear at this location. Now the way the insertion will happen is just move out this 10 and store at in it temporary variable and then after that shift this element one step to the right again shift the element one step to the right and shift the next element one step to the right. So, that now 10 can be safely stored here because the old value stored here has been shifted here, the value has been stored here has been shifted here the value has been stored here has been shifted here, and so 10 comes here, so that is the insertion steps.

(Refer Slide Time: 20:00)



Now ready to look at the implementation of the algorithm. So, we have the function insertion sort which has no return value. Takes an array A and an integer n and sorts array A. What does this function is going to modify these values of elements of A, but and it does not does not return the new value, but that alright because the array as you know is past as the starting address of the array is is really past and when element of that array is accessed or modified. It is accessed using its address, and so therefore, the change is reflected in the array of the calling function. So, we have this local variable i, j and t.

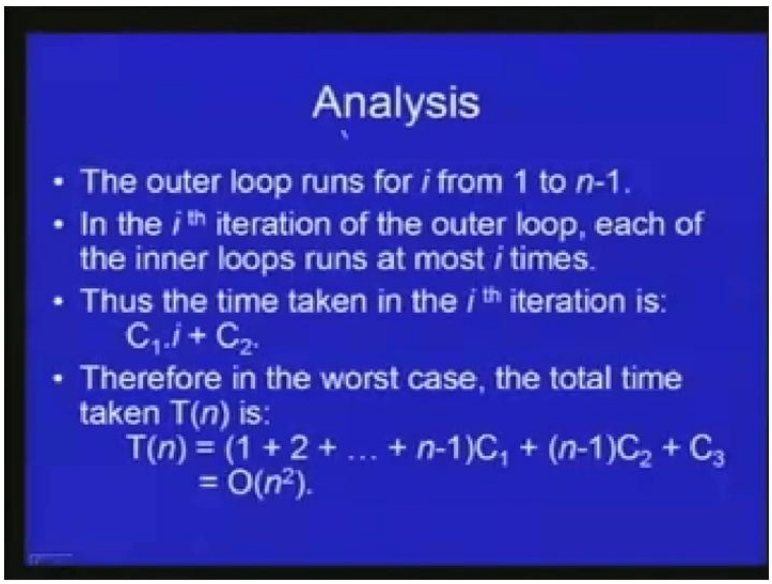
Now this is the outer loop which runs for i from 1 to n minus, and for each value of i, we have to first do the search. The search is quite easy. So, we start so essentially what we have to do is for j from 0 to i minus 1. We have to find the first value of j such that A j is greater than A of i, because A of i is what we are trying to insert here. So, we start from 0 and we proceed towards A of i minus 1 and as soon as we find A j such that A j is greater than A i we break from the loop.

So, when we come here either A j is greater than A i and that is why we broke from the loop or it is possible that j is equal to i. If j is equal to i; that means, that A i is already in its correct position, it is already larger than all of A 0 to A i minus 1 and nothing really needs to be done in this case, so this is the insertion step. Note that this step happens even when i is equal to j, but as will see that it works (( )) even when A i is equal to j. So,

the first move the value of A i to a temporary variable and then for all location of the array starting from i minus 1 going back right up to j. This should be k minus minus instead of plus plus. We moved value of A k to the location A k plus 1, and so all the element of the array starting from i minus 1 and going back up to j they get shifted 1 place right and then finally, the location A j is given value t which was the original value of A i.

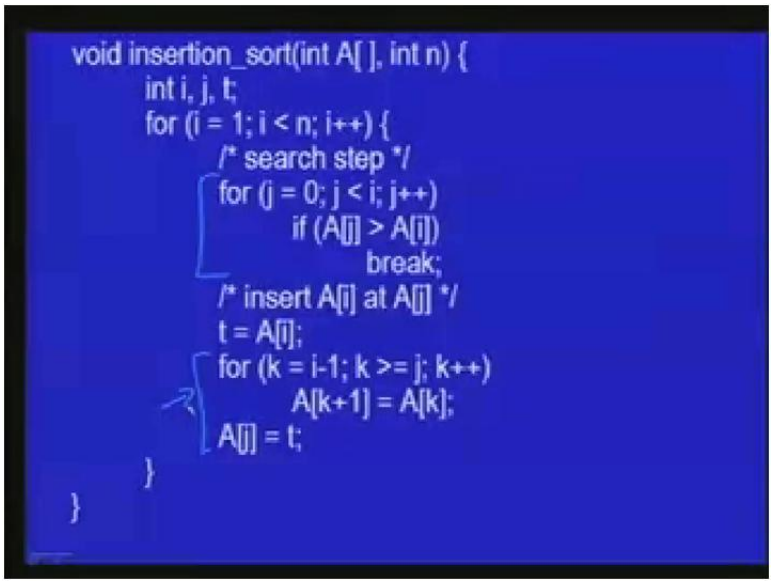
Note that if i happens to be equal to j, (( )) happens is that t will be assigned A i and then this loop will not run at all, because the initial value of k is i minus 1; j is equal to i, so i minus 1 is not greater than equal to i. So, this this condition will become false in the very first iteration, and so therefore, this loop body will not execute at all when j is equal to i, and so therefore, all that will happen will be these two assignment, t assigned A i and A j assigned t. Now since j is equal to i, all that happens is that t assigned A i and again A i assigned t which has no effect really because the same value is assigned back into A i. So that is we want it to happen. So, when the outer loop terminates when it has run for all values of i from n 1 to n minus 1, the array sorted and the function returned.

(Refer Slide Time: 24:01)



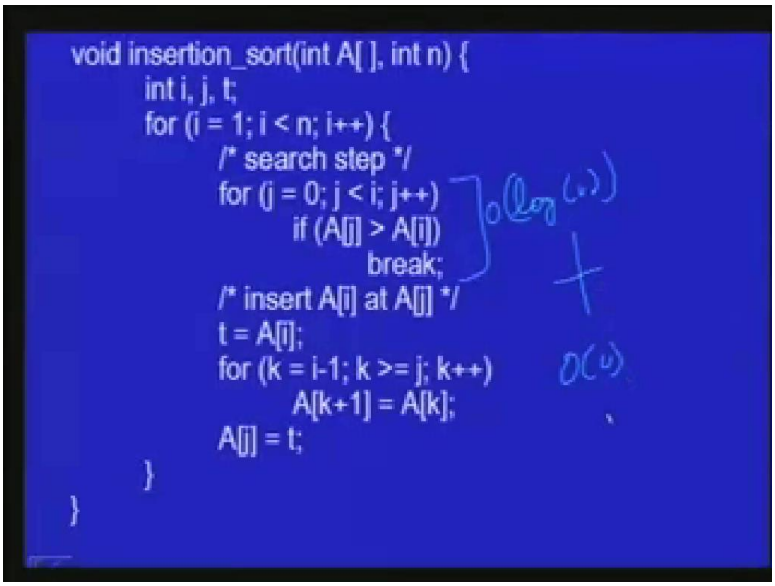
Let us now, analysis this algorithm to find its time complexity. The outer loop as we know runs for i from 1 to n minus 1 for different values of all values of i from 1 to n minus 1. And let us look at one particular iteration of the outer loop, so let us look at the iteration where for some particular value of i - that is i eth iteration of the outer loop.

(Refer Slide Time: 24:27)



Now in the i eth iteration of the outer loop. This loop as well as this loop will run at most i times, because we may have to do this i times because the element which is greater than i may not be present at all in which case we need to look at all elements from A 0 to A i minus 1. And the worst case for this step happen when when the value of k turns out to be 0 itself because then all elements from A 0 to A i minus 1 need to be shifted. So, a total of i element get shifted. So, and if you ,if you think actually about this, the number of time that this loop will execute plus number of times this loop will execute will actually always be equal to i. So, therefore, the time taken in the ith iteration is in the worst case is C 1 i times plus C 2 - that it is proportional i plus a constant.

(Refer Slide Time: 25:34)



Note that if you had the search as a binary search step instead of using linear search if you used binary search over here then the amount of time taken in this step in the worst case would be log i and the amount of time taken in the worst case over here would be order i, but if you add this two up that would be still order i. So, which means that the amount of time taken in the i eth iteration would not really change in the order notation, and therefore, the overall time taken would also not really change in the order notation, even though using binary search is off course slightly more efficient.

So, for each iteration of the for the i eth iteration of the loop, the time taken by the iteration is proportional to i or we have denoted as C 1 i plus C 2, and therefore the total time taken is T 1 times 1 plus C 2 for i equal to 1 plus C 1 times 2 plus C 2 for i equal to 2 and so on up to i equal to n minus 1. And so add up all that C 1 will have a coefficient of 1 plus 2 plus up to n minus 1, and C 2 will have a coefficient of n minus 1, and the C 3 times might be some constant mode of time required in addition to the outer loop, for example, for initialization and over head of the function called and so on and so forth.

So, now, if you this session and this is an arithmetic series, and if you add this up, its turns out be A n times n minus 1 by 2, which is a constant times n square plus lower ordered terms, and so the entire thing is constant times n square plus lower ordered terms, so and therefore in the (( )) notations, this is order n square. So, this is the end of

today’s lecture, in the next lecture we will talk about a new data type in c that we have looked at so far which is the structure type.