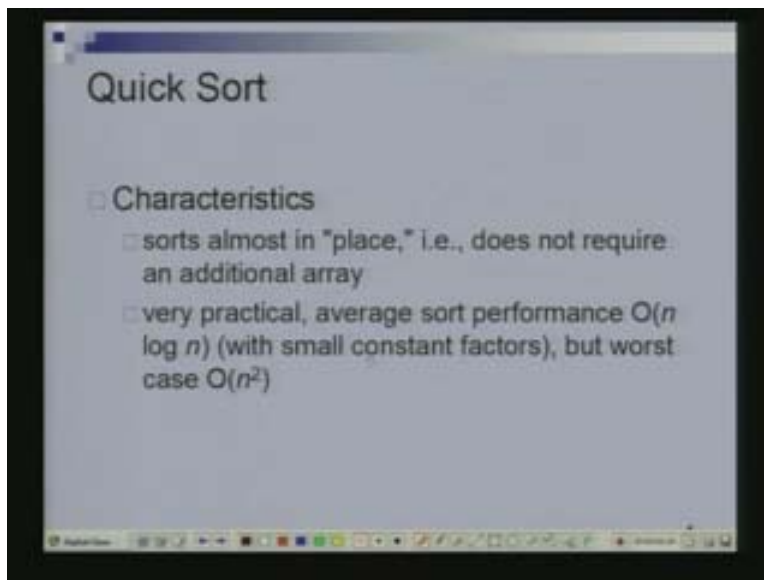


**Data Structures and Algorithms**  
**Dr. Naveen Garg**  
**Department of Computer Science and Engineering**  
**Indian Institute of Technology, Delhi**  
**Lecture – 10**  
**Quick Sort**

Today we are going to talk about quick sort. This is the second sorting algorithm we are discussing in this series of lectures. The first one was insertion sort for which we had argued a worst case running time of  $O(n^2)$ . Today the quick sort algorithm which we are going to look at is also going to have a worst case running time of  $O(n^2)$  but we will argue that on an average it takes only about  $n \log n$  time. It is a quick algorithm, in practice it is very fast with very small constants.

(Refer Slide Time: 01:35)



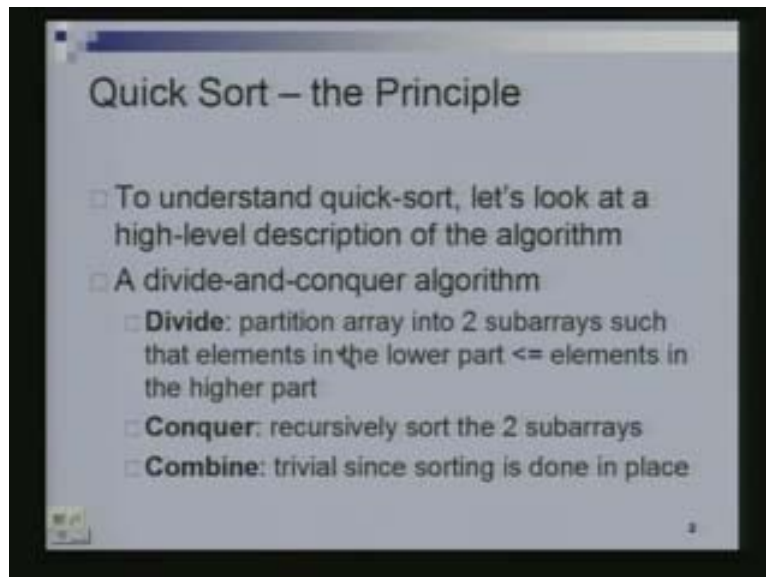
Another property of this algorithm is that it is in “place” sorting algorithm. What is an in “place” sorting algorithm? An algorithm is called in “place” if you do not require any additional memory to do the sorting. We will assume that  $n$  numbers are given in an array, may be we need memory for 1 or more variables but we will not need any additional memory to do the sorting. That is called the in place sorting and that is typically good thing to have. Because its space is at a premium, especially when you are trying to sort a large collection of numbers.

This algorithm falls into a paradigm which is called divide and conquer. I am not going to say too much about this because we are going to see a lot of divide and conquer algorithms in this course. At a very high level, the idea behind divide and conquer is the following that you are given a certain problem that you want to solve. You divide the problem up into 2 or more pieces. You solve the problem for those smaller pieces.

The divide step in the case of quick sort would be partition our array. We will take the array which stores the n numbers and partitioned it; break it up into two parts. One is we will call it as the lower part and the other we will call the higher part. The property of the lower and the higher part would be that every element in the lower part, every number in the lower part would be less than every number in the higher part.

How does this help us if we do such a partition? If I sort the lower part now and I sort the higher part and I put all elements of the lower part first and follow it up with all elements of the higher part then the entire thing is sorted. I have sorted the lower part and I have sorted the higher part, every elements here is less than the element in the higher part. So the entire thing is sorted. That is the combining part and in this case it is trivial because nothing needs to be done. But quite often in divide and conquer algorithm you have to do something to do the combine and we will see examples of these later. So we will go one step at a time. We will first understand how this partition is done.

(Refer Slide Time: 4:10)



So we will give an algorithm to do the partitioning which will be a linear time procedure. So the partitioning is done around a pivot element. I will take one of these elements as a pivot and everything which is smaller than the pivot will be my lower half that is lower part of the array. And everything which is larger than the pivot will be the larger part. This is the procedure to do the partitioning. It takes the parameters, the array A and p, r are the limits of the array. The p here refers to this location and r refers to this location and this could be part of a larger array (Refer Slide Time: 05:10). This just says that partition the sub array from p to r.

We will see what it will do at the end of the procedure. I am going to take A [r] which would be the element 10. I will put it into x which means x is going to be my pivot element. So x would contain 10. What is this doing? The i is getting the value p-1 so I is assigned to something before p. So this location is r and this location is p. And i is getting

p-1 and j is getting r+1 and so which means i and j are just before and after that is the start and the end of the sub array that we are interested in. And now we are just going to go through this loop. The while TRUE which means we just continue going through this loop, till you break out to the loop. What is this doing? This is saying repeat j is j-1 until A [j] is less than or equal to k. So we are looking at this index and we are saying keep decrementing it till I reach a location which is less than or equal to 10. What was x? The pivot 10. So keep decrementing it, so I decrement j till I reach a location which has content less than or equal to 10. So already I reached such a location so I stopped decrementing.

(Refer Slide Time: 08:40)

**Partitioning**

Linear time partitioning procedure

```

Partition(A, p, r)
01 x ← A[r]
02 i ← p-1
03 j ← r+1
04 while TRUE
05   repeat j ← j-1
06     until A[j] ≤ x
07   repeat i ← i+1
08     until A[i] ≥ x
09   if i < j
10     then exchange A[i] ↔ A[j]
11   else return j
  
```

Diagram 1: Initial array [17, 12, 6, 19, 23, 8, 5, 10]. Pivot x=10. i points to index 0, j points to index 7.

Diagram 2: j moves left to index 6 (value 5). i moves right to index 1 (value 12). Array: [10, 12, 6, 19, 23, 8, 5, 17].

Diagram 3: i moves right to index 2 (value 6). j moves left to index 5 (value 8). Array: [10, 5, 6, 19, 23, 8, 12, 17].

Diagram 4: i moves right to index 3 (value 8). j moves left to index 4 (value 23). Array: [10, 5, 8, 8, 23, 19, 12, 17].

Now I go to the next loop where am incrementing i till I reach a locations which is greater than or equal to 10, so already I reached a location which is greater than or equal to 10. What am I going to do? Recall that I want everything in the left part to be less than 10 and everything in the right part to be more than 10. So these are in some sense culprits because this is more than 10 and it is in the left part and this is less than or equal to 10 and it is in the right part. So we will like to swap them and that is what we will do here. Exchange A[i], A[j] means just swap these contents. So I swapped these contents. So I will denote blue as everything which is in the left part and orange would be everything which is in the right part. So everything in the left part would be less than or equal to 10 which was my pivot element and everything in the right part which is the orange part will be greater than or equal to 10.

Recall, i is here at this point and j is here and once again I am going to keep decrementing j till I find an element which is smaller than 10. So I found one already and I keep incrementing i till I find an element which is larger than 10. So that actually I will find here and once again I will swap these things. I swapped them and I continue with j, so j will keep moving till I find something which is smaller than 10. So actually I found immediately one which is smaller than 10 at this location and it will keep moving till I

find an element which is larger than 10. So this is not larger so actually I will end up till this location 19 and I will once again swap 8 and 19. I swap 8 and 19 and I get this. This is already smaller so this is also marked blue (Refer Slide Time: 07:38- 08:30).

Now j would continue decrementing till I find something which is smaller than 10. The j is searching for something which is smaller than 10. So it will come to this position, this is the first element which is smaller than 10 (Refer Slide Time: 08:45). So it comes here and i will continue incrementing till I find something which is larger than 10. So it comes here and now i and j have crossed each other, which means our job is done. So if i is less than j then you do an exchange, if i is more than j which means they crossed each other then you exit which means return the procedure. What we are returning? We are returning the value of j which tells us about the boundary of the two halves.

So this is my left half that is my left half is from p to j and my right half is from j+1 to r because am only returning j and not i. So j+1 to r is my right half. How much time does this procedure takes? It is order n. Is this clear that its taking order n time and no more because [Hindi Conversation] at every step we are decrementing j. How many times can j be decremented? It is at most 10 times the size of the array. And how many times can i be incremented? At most the size of the array, so this loop is done at most n times, this loop is done utmost n times and every time we increment or decrement we might have to do one exchange in the worst case so this is also never be done more than n times (Refer Slide Time:10:16).

(Refer Slide Time: 12:30)

**Partitioning**

Linear time partitioning procedure

```

Partition(A, p, r)
01 x ← A[r]
02 i ← p-1
03 j ← r+1
04 while TRUE
05   repeat j ← j-1
06     until A[j] < x
07   repeat i ← i+1
08     until A[i] > x
09   if i < j
10     then exchange A[i] ↔ A[j]
11   else return j
  
```

The diagrams show the array [17, 12, 6, 19, 23, 8, 5, 10] being partitioned around the pivot 10. The pivot is highlighted in yellow. The first diagram shows the initial state with i at index 0 and j at index 7. The second diagram shows j moving left to index 1. The third diagram shows i moving right to index 2. The fourth diagram shows the final state after partitioning, with the array [10, 5, 6, 8, 23, 19, 12, 17] and i at index 3 and j at index 3.

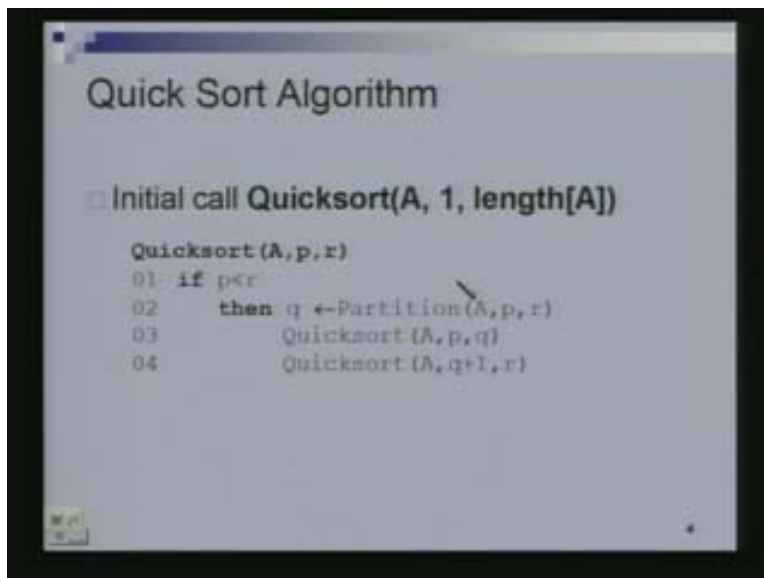
It is a simple way. Why do we do it in this manner, you could also have done it in a slightly different manner. After all we are taking the pivot and saying, compare every element with the pivot and put everything which is smaller in the first few locations and everything which is larger in the next few locations. If you try to do it, some other way you will require more memory. One way you could do is take one element at a time and

put it in some other array and then copy that array back into this place. That would be one way of doing it. But that will take more memory, so we wanted to do it in in-place as to keep all the elements in this array and not to take up any additional memory space.

We can do this partitioning in in-place that is no additional memory and in linear time which is the best pursue. I am saying let us look at this repeat until loop over all the while loops. Let me just look at this one statement, the total number of times this is executed not in one iteration of the while loop but all while loops put together. How many times is this statement executed? At most  $n$  times because I can decrement it only at most  $n$  times. That is not possible, because for one particular run of the while loop I might decrement it say for 3 times. Then for the next one I might decrement it for another 5 times and so on. But the sum over all will not be more than  $n$  because [student Conversation] At most  $n$ , that is all I am interested in. The total number of times this step is executed and this step is executed and this is executed is order  $n$ .

This is my complete quick sort algorithm, I did my partitioning so I have to do quick sort on lets say this array, array  $A$  between limits  $p$  and  $r$ . When I do an initial call for quick sort I will do 1 to length of  $A$ , whatever is the length of  $A$ . So in general this would be the thing, so  $p$  to  $r$ . I need to do it only if  $p$  is less than  $r$ , if  $p=r$  then there is nothing to be done or if  $p$  is more than  $r$  then again it does not make any sense. So if  $p$  is less than  $r$  then i do something. What do I do? I first find the partition of this part  $p$  to  $r$  then I invoke the previous procedure.  $A, p, r$  what does it do? It rearranges the part of the array between  $p$  and  $r$  such that everything which is less than the pivot is in the initial part of the sub array and everything which is more than the pivot is in the later part of the sub array.

(Refer Slide Time: 16:23)



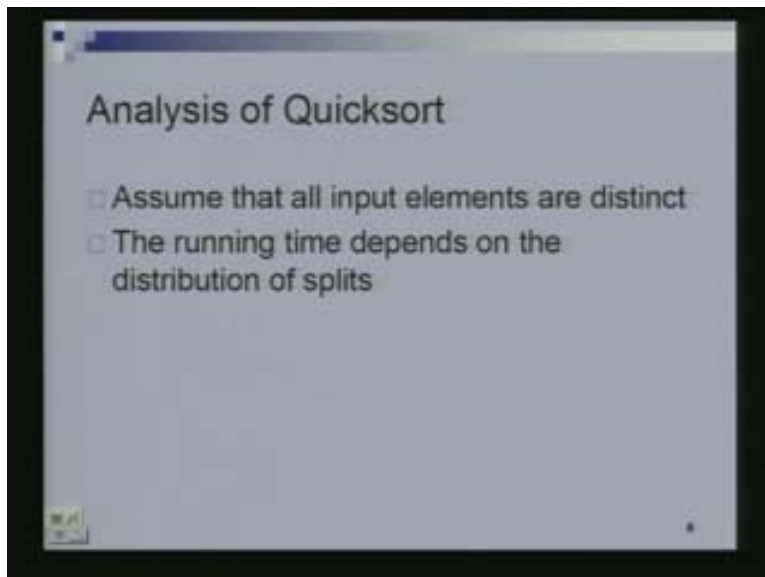
And what does it return? It returns the demarcating lines, so the lower half is going from  $p$  to  $q$  and the larger half is going from  $q+1$  to  $r$ . So I need to sort the lower half and the upper half separately. So I recursively invoke quick sort on the lower half which is going

from  $p$  to  $q$  and on the larger half which is going from  $q+1$  to  $r$ . The while TRUE means just keep doing the while loop forever. Where does this while loop stop? It will stop when you return out of this, it's like an exit out of this while loop, like a break statement.

So when this condition is met that  $i$  is more than  $j$  then you will return  $j$  and go out of this entire partition procedure itself. So which means that you also go out of the while loop. So when you call quick sort recursively, note that I do not have to copy the array  $A$ , it is the same array that is used. We might have to create more copies of the variables, actually we would only be creating additional copies of these parameters that we are passing. But that is the space created on this stack. We are ignoring this but we are not taking any additional memory for the elements that we have. They are all sitting in a single array. What element should be partitioned around? What should be our pivot element?

Did every one understand the quick sort? You took a pivot element and you partitioned the array around the pivot element and you said let me sort this left half and let me sort this right half and then I am done. Then how do you solve the left and right half? Repeatedly by the same procedure. Since we have a notion of a left half and a right half therefore we need to write a quick sort procedure in this manner, because this will specify the limits of the sub array that you are sorting  $p, r$ . Let us try and analyze how much time does quick sort takes? We have only seen that the partition procedure takes order  $n$  times that is linear time but we do not know how much time quick sort takes. So the time taken by quick sort will depend upon how the split is happening. What do I mean by that now? I should not say left half but it is left part. So how many elements end up in the left part and how many elements end up in the right part? That is what we should find out and that will determine how much time quick sort is taking. Let's see why.

(Refer Slide Time: 17:00)

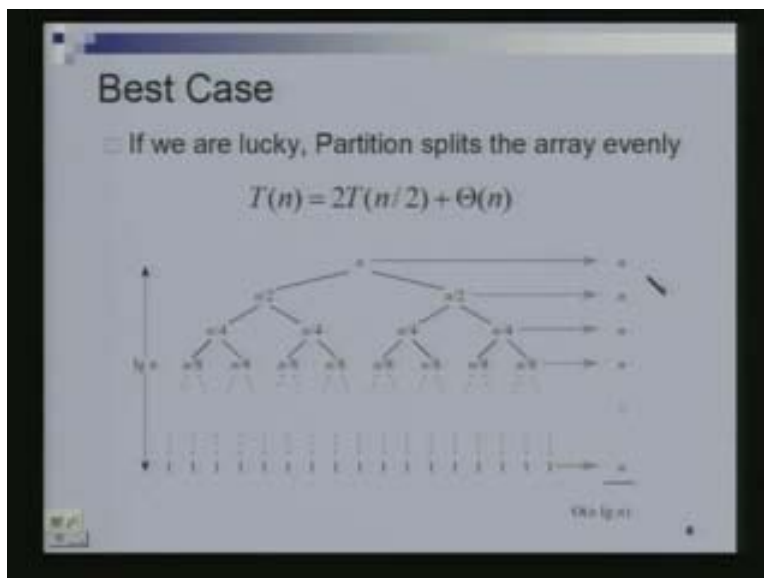


So if at every point, suppose the following was happening. At every point we were actually dividing the array up in to two equal halves which means that I started of with  $n$  elements, this here [Hindi Conversation] Somehow or I was lucky or whatever it is, my pivot was such that it was the median lets say. Which means that half the elements were less and half were more. So half the elements ended up in the left half and half elements ended up in the right part. Then when I did a quick sort on this again I was lucky, again I picked a pivot such that it divided up the thing in to two parts.

And when I did my quick sort on this again I was lucky it just divided up in to two equal parts and so on. Suppose I was lucky at every step, then how much time am I taking? Let's see. How much time did I take to divide this array up into two parts that is  $n/2$  and  $n/2$ ? It is  $n$  times, the size of this array. We saw the partition procedure took order  $n$  time. How much time did I take to divide this array of size  $n/2$  into two parts that is  $n/4$  and  $n/4$ ? The  $n/2$  here and  $n/2$  here so that it becomes  $n$ . Then to divide this up into two parts that is  $n/8$  and  $n/8$ , I again took  $n/4$  here  $n/4$  here  $n/4$  and  $n/4$  and so that is also  $n$ . So in each level of this tree that I have drawn am taking order  $n$  time to do the partition. And how many levels are there in this tree of mine? It is  $\log n$  because eventually you will end up with one and there would be  $\log n$  such levels. So the total time taken is order  $n \log n$ .

Why have I written theta of  $n$  and not  $o$  of  $n$ ? Can you also say that the partition will take at least omega  $n$  time or at least will take  $n$  time? Yes, because when did we say that we would stop? When our  $i$  and  $j$  would inter change, so  $i$  has to go at least till some part and  $j$  has to go, so the total number of times I will increase  $i$  or decrease  $j$  is at least 10. Total number of times I will increase  $i$ , I am not saying that number of times I will increase  $i$  is at least half, it is not  $n/2$ . May be I increased  $i$  only  $n/4$  times but then it means I decreased  $j$  at least  $3n/4$  times. So the sum is at least  $n$ . I need at least  $n$  times which mean I need utmost  $m$  times and I need at least  $n$  times, so the exact time is really some constant time, theta of  $n$ . So I can actually say equality.

(Refer Slide Time: 20:31)



What happens in general, if you are not lucky? What is the worst case?  $n$  squared, when we partition [Hindi Conversation]. One side gets one element and the other side gets  $n-1$  elements. We could have such a situation,  $T(1) + T(n-1)$ . Again I am writing the recurrence relation. Time to quick sort  $n$  elements equals the time to partition. That was the very first step of our quick sort procedure plus the second step of our quick sort procedure was quick sort the left part lets say the left part has one element in it, so  $T(1)$  plus the time to quick sort the right part which has lets say  $n-1$  elements, it is  $T(n-1)$ .

So this is our recurrence and let us solve this recurrence in exactly the same manner that we did earlier. So I am assuming  $T(1)$  as zero. So this is just  $T(n-1) + \theta(n)$ . If you have one element there is nothing to be done, it is sorted. So this is just  $T(n-1)$  plus  $\theta(n)$ . What is  $T(n-1)$ ? It is  $T(n-2) + \theta(n-1)$ . So  $T(n-2) + \theta(n-1)$  and  $T(n-2)$  is  $T(n-3) + \theta(n-2)$  and so on. So it has essentially become  $\theta(k)$  where  $k$  going from one through  $n$ . What all I am saying is, this is then equal to  $T(n-2) + \theta(n-1) + \theta(n)$ , which is  $T(n-3) + \theta(n-2) + \theta(n)$  plus this term which is equal to  $T(n-4) + \theta(n-3) + \theta(n)$  plus this entire thing and so on (Refer Slide Time: 23:54). So this is what you get which is basically  $\theta(n^2)$ . It is  $n$  squared,  $k$  going from one through  $n$ , sum of  $k$  is just  $n$  squared. That is why you get  $\theta(n^2)$ .

If this  $\theta$  is bothering you, just replace it by  $c$ , some constant times  $n$ . So this would be the worst case. So the best case is when you do a half split and the worst case is when you know it is a skewed split. So this is what the worst case look like pictorially,  $n$  divided into 1 and  $n-1$ ,  $n-1$  divided into 1 and  $n-2$ ,  $n-2$  divided into 1 and  $n-3$  and so on. And what will be the height of this tree now? It is  $n$ . In each step you are taking time  $n$ ,  $n-1$ ,  $n-2$  and so on. All the way down to 1 so that makes it  $n$  squared.

(Refer Slide Time: 23:46)

**Worst Case**

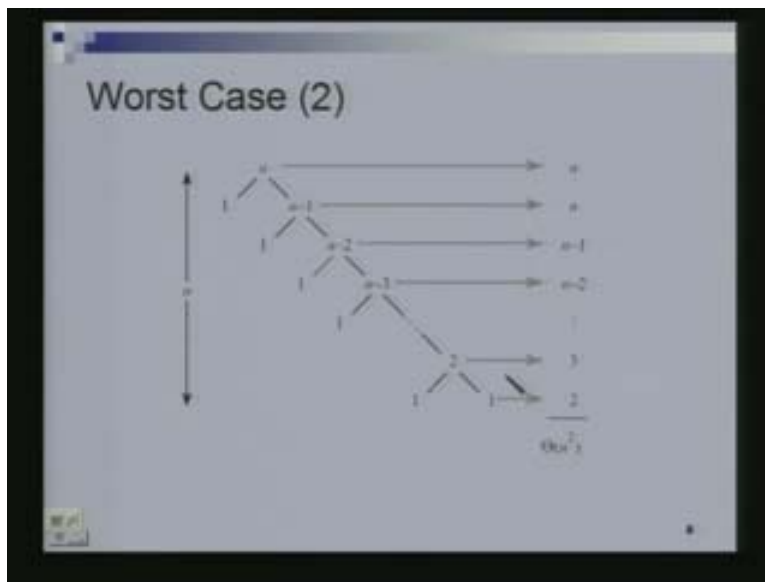
- What is the worst case?
- One side of the partition has only one element

$$\begin{aligned}
 T(n) &= T(1) + T(n-1) + \theta(n) \\
 &= T(n-1) + \theta(n) = T(n-2) + \theta(n-1) + \theta(n) \\
 &= \sum_{k=1}^n \theta(k) \\
 &= \theta\left(\sum_{k=1}^n k\right) \\
 &= \theta(n^2)
 \end{aligned}$$



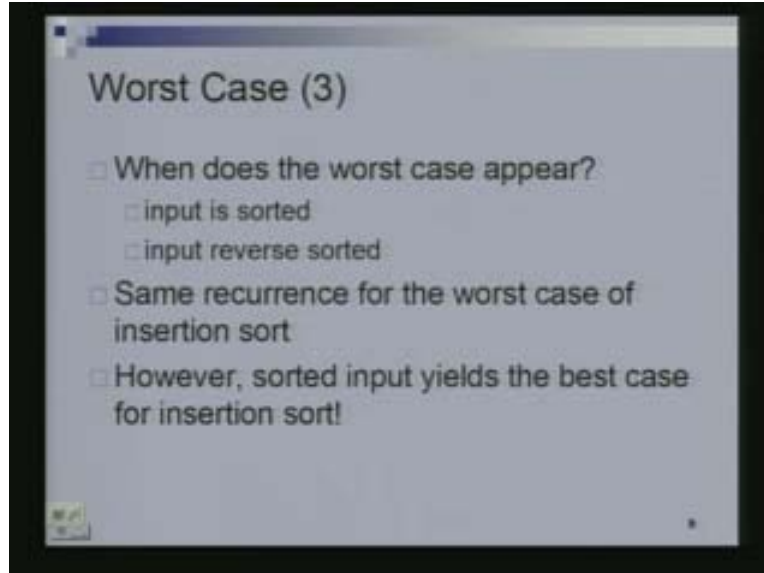
When does the worst case appear? Suppose we were doing a following scheme, we were saying let me take the last element as the pivot always. That is what we did to begin with, let me take the last element as the pivot. So if my input is sorted already lets say in increasing order. Then I took the last element as the pivot. How many elements will be in my lower part? The  $n-1$  elements in my lower part and only one element in my upper part because there is no element larger than the last element. And then once again to sort the lower part I took the last element as the pivot. So once again it will get divided in this manner.

(Refer Slide Time: 24:16)



So the worst case would happen when the input is already sorted in ascending order or in descending order. Even in descending order you will take  $n$  squared time. Because when it is in descending order you took the last element as the pivot, it is the smallest element [Hindi Conversation] there will be one element, right half will have all the  $n-1$  elements in it. And you keep doing this and this is what will happen. Similar kind of a thing happened in insertion sort. In insertion sort we said the worst case would happen when it is in descending order. Because if you recall in insertion sort we were taking an element and figuring out the best place to put that element. And we would go from the end to find the best place. So if it is sorted in decreasing order then the best place is always the front of the array. So you will have to go all the way to the front of the array at every step. It is again  $1+2+3+4$  and so on all the way up to  $n$ . So you will get again the same  $n$  squared.

(Refer Slide Time: 26:35)



But in insertion sort if the array was sorted in increasing order then how much time do you take? Then it is the best case because you do not have to move back anymore. Every element is in the right place, so it just takes constant amount of time. That is the comparison with insertion sort. But here both, whether it's sorted increasing or sorted decreasing you might end up with something like  $n^2$  time. So worst case seems to occur more often.

Let us continue with this analysis. We saw [Hindi Conversation] if the split was half and half at every step, then we are lucky and we get  $n \log n$  time. Suppose it was not half and half but it was one tenth, nine tenth that is 10 percent of the elements end up on one side and 90 % of the elements **change out**. Suppose this was happening at every stage. You will not call this lucky because it's not half and half but it's still good which we will argue now. At the first stage this is what happens,  $n/10$  elements on one side and  $9n/10$  elements on the other side. How much time did I take to do this partition?  $n$  time, then this  $n/10$  got divided into one tenth on one side and nine tenth on the other side. One tenth means  $n/100$  on one side and nine tenth of this is  $9n/100$  on the other side. And this  $(9/10)n$  also gets divided into  $9n/100$  on one side and nine tenths of this guy which is  $81/100n$  on this side. How much time did it take to partition this into this and this (Refer Slide Time: 27:45)  $n/10$  number of elements and so for this the total time is still  $n$ .

(Refer Slide Time: 29:56)

The slide is titled "Analysis of Quicksort". It contains the following text and diagram:

□ Suppose the split is 1/10 : 9/10

$$T(n) = T(n/10) + T(9n/10) + \Theta(n) = \Theta(n \log n)!$$

The diagram shows a recursion tree for the recurrence relation  $T(n) = T(n/10) + T(9n/10) + \Theta(n)$ . The root node is labeled 'n'. It has two children: a left child labeled 'n/10' and a right child labeled '9n/10'. The 'n/10' node has two children, each labeled 'n/100'. The '9n/10' node has two children: a left child labeled '9n/100' and a right child labeled '81n/100'. The '9n/100' node has two children, each labeled '9n/1000'. The '81n/100' node has two children: a left child labeled '81n/1000' and a right child labeled '729n/1000'. The tree continues to branch out, with nodes becoming smaller as they go down. On the left side of the tree, there are vertical arrows indicating the height of the tree. The top arrow is labeled 'log\_{10/9} n' and the bottom arrow is labeled 'log\_{10/9} n'. On the right side, there are horizontal arrows indicating the width of the tree at each level. The top arrow is labeled 'n' and the bottom arrow is labeled 'n'. The slide number '18' is visible in the bottom right corner.

So similarly the total time for every level will continue to be  $n$ . But now how many levels do I have? Let's figure out that. Let's just look at the largest number we have, as we go down one level at a time. We want all the numbers go down to one, so the largest number is the one which we have to say it goes down to a one. So at this route we have  $n$ , the largest number at this step will be  $9n/10$  because the other number is smaller. What will be the largest number at this step? It will be in the nine tenths of this guy which will be the largest one, so that will be  $81/100n$ .

The largest at the next step would be the nine tenths of this guy [Hindi Conversation] that will be the largest guy at this step which will be this and so on. So the largest number at every level is decreasing by a factor of  $9/10$ . How many times can I decrease before it gets down to one? At most  $\log$  of  $n$  to the base  $10/9$  because it is decrementing by a factor of  $9/10$ ,  $10$ 's at every step. When it was decrementing by half at every step, we were saying  $\log n$  base two. So if we just work out the math it will be  $\log n$  to the base  $10/9$  which is order  $\log n$ . It is just some constant, it is different constant times  $\log n$ . So once again the height is order  $\log n$ , so the total time taken is order  $n \log n$ . Because at each level we are taking a total time of  $n$  and number of levels is  $\log n$ .

But even better, this is we are only providing an upper bound. Even if this split was in this strange manner one tenth and nine tenth or any constant fractions, there is nothing sacrosanct about one tenth and nine tenth. I could have said  $36/37$ , even that is ok, it is not a problem. Even then we will be able to argue  $\log n$ . There is nothing spectacular or special about  $1/10$ . The important thing is we are saying a constant fraction of numbers go on one side. We cannot afford to say only one number go on one side that becomes bad for this. When only one number go on one side or only two numbers go on one side then we will end up taking  $n$  squared time. But if we say a constant fraction one tenth or one hundredth or one thousandth or one millionth the height would still be  $\log n$ .

We will do a formal analysis starting from the next slide, but just to give you a little more motivation on this. Suppose we alternate the lucky and unlucky cases, even then you can prove a  $\log n$  depth. So what was our unlucky case? 1 and  $n-1$ , then what was the lucky case?  $n-1$  divided into two,  $n-1/2$  and  $n-1/2$ . How many operations did I take here to split  $n$  and how many operations did I take here to split  $n-1$ ? So in total I took  $2n-1$  operation here and after  $2n-1$  operations or comparisons I managed to split it into  $n-1/2$  and  $n-1/2$  and 1 with  $2n-1$ .

(Refer Slide Time: 32:51)

**An Average Case Scenario**

- Suppose, we alternate lucky and unlucky cases to get an average behavior

$L(n) = 2L(n/2) + \Theta(n)$  lucky  
 $U(n) = L(n-1) + \Theta(n)$  unlucky  
 we consequently get  
 $L(n) = 2(L(n/2-1) + \Theta(n/2)) + \Theta(n)$   
 $= 2L(n/2-1) + \Theta(n)$   
 $= \Theta(n \log n)$

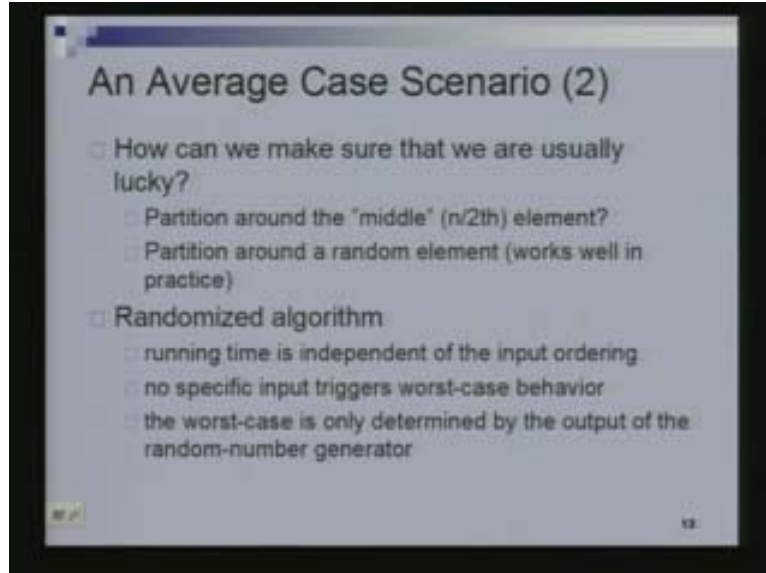
$L(n) = 2L(n/2) + \Theta(n)$  lucky  
 $U(n) = L(n-1) + \Theta(n)$  unlucky  
 we consequently get  
 $L(n) = 2(L(n/2-1) + \Theta(n/2)) + \Theta(n)$   
 $= 2L(n/2-1) + \Theta(n)$   
 $= \Theta(n \log n)$

$n \rightarrow \Theta(n)$   
 1       $n-1$   
 $(n-1)/2$      $(n-1)/2$

$n \rightarrow \Theta(n)$   
 $(n-1)/2+1$      $(n-1)/2$

So now I can again bring the same tree. I will get a recurrence like this. I can think of it as this with theta  $n$  or in particular  $2n$  comparisons I managed to split it into  $n-1/2$  on one side and lets say  $n-1/2+1$  on the other side. Actually I managed a three way split because the 1 is coming here. So once again we will have a depth of only  $\log n$  and I took  $2n$  in each step, so its  $2n$  times  $\log n$ . We will not worry too much about this lets try and do this formally. So it seems that for many scenarios we will get a  $\log n$ . We want to argue that this really is the case?

(Refer Slide Time: 35:14)



So what is the best thing to do? We said we will be lucky when we always partition half and half. The best thing to do would be to find a median element. Pick the median element as the pivot and that will break up my array into two equal parts and that would be great. How do I find a median element? Sort the numbers and then find the median element. That would be one strategy except that sorting is what we are trying to do in that first place. So finding median element is not straight forward, you will see a procedure for doing it in your next algorithms course. [Student Conversation: not clear-some what close to the median by dividing the array into some odd number of small array then sorting these, as in like write it into small arrays of size five and sort them and then take the median of these. Now I repeat this till I get one number, that it is close to the median and we will take  $\log n$  steps.] You can actually compute a median element in linear time but it is fairly an involved procedure which you will learn later.

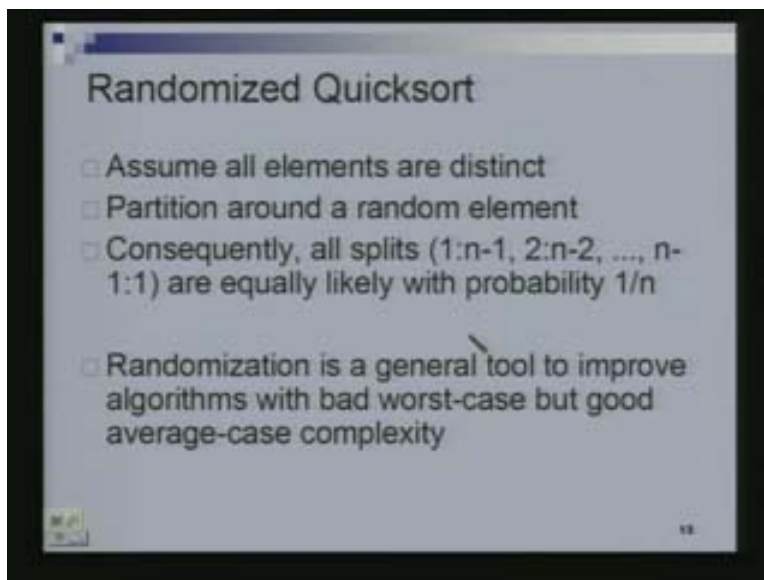
For now what you will do is, since you want to find the median element and you cannot find the median element. So you will just pick a random element and declare that. What we are going to do is we are just going to pick a random element as our pivot. We said we do not want to pick a specific element as the pivot. Why? Because if I always want to pick the last element as the pivot then if my sequence is in decreasing order or increasing order then I will struck with an  $n$  square running time. So I will just pick a random element as the pivot. So this is what we call a randomized algorithm.

What is a randomized algorithm? An algorithm which is basically making some kind of random choices and we will analyze what this algorithm does. We will analyze the running time of this algorithm. So this is what a randomized quick sort is. We will assume all elements are distinct. We partition around a random element. A pivot is a random element and we just pick any element at random with the same probability. So what kind of splits we can get? We can get all kinds of different splits, if I have  $n$  elements I can get a split of  $1n$   $n-1$ ; I can get a split of  $2n$   $n-2$  and so on. What will be the

probability of these splits? They all will be equal. They will all be with probability one over  $n$ . Did you understand why they would all be equal? Because it is a random element, each element can be picked with equal probability.

So if I pick the **five**  $n$  elements and I pick the tenth smallest element then I will get a 9 versus  $n-10$  split or a 10 versus  $n-10$  split. But what is the probability of picking the tenth smallest element? It is  $1/n$ , because I could have picked any element. So I do not know what I am picking. The tenth smallest element is as likely as the 11<sup>th</sup> smallest element which is as likely as the 12<sup>th</sup> smallest element. The probability of each one of them is the same. We will see more examples of randomization in this course and it is a useful tool for designing algorithms.

(Refer Slide Time: 36:44)



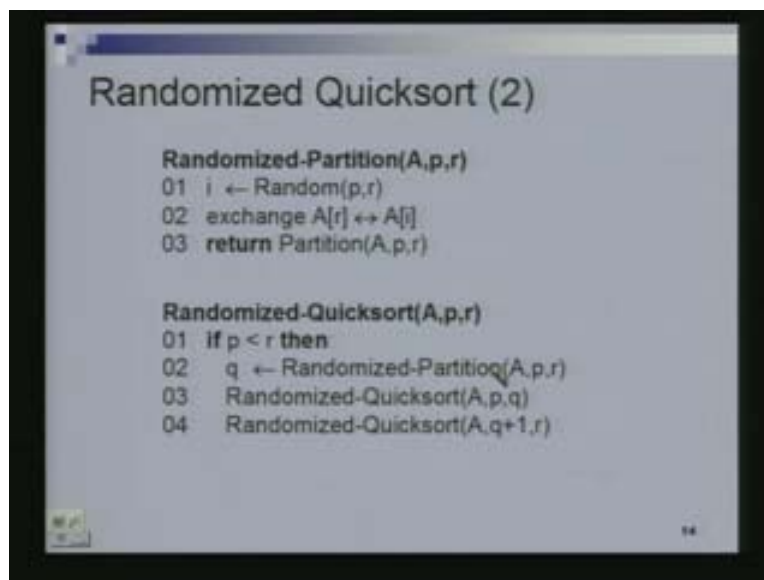
So this is what our randomized quick sort is going to look like. So we are only going to modify the partitioned procedure and call it randomized partition instead. Once again we are trying to partition the array  $A$  between  $p$  and  $r$ , the sub array between locations  $p$  and  $r$ . What is random  $p$ ,  $r$ ? Random  $p$ ,  $r$  generates a random number between  $p$  and  $r$ . Lets say that number is  $i$ , between  $p$  and  $r$  means including  $p$  and  $r$ . Lets say that number is  $i$  and we are just going to exchange  $i$  and the last element. Why am I doing this? Because this partition procedure, if you recall was taking the last element as the pivot. Now I want to put my pivot at the last location, so I just exchange the pivot element with the last location and then I call my partition procedure, the same partition procedure as before. So this becomes my randomized partition procedure.

And what does randomized quick sort do now? Instead of calling partition it calls randomized partition, the rest is the same. Did you understand what it is? Since our choice of pivot is crucial and we do not know really how to choose the pivot. We just pick a random element as the pivot. **No, it would not be a random choice.** So what is the difference between random choices here? So he has a very good question, he says if I

partition it around the last element, why is that not a random choice? That is not a random choice because if I give you a specific input which is lets say increasing order then on that input you are going to take  $n^2$  time. If you partition around the last element.

Now in randomized quick sort we are not partitioning around the last element. We are picking a random element to partition. When I pick a random element to partition then given the same sorted sequence as input, you are not going to take  $n^2$  time. In fact how much time are you going to take? We do not know how much time we are going to take. That is the interesting thing. Why we do not know how much time we are going to take? We do not know what the pivots are going to be, they are randomly selected.

(Refer Slide Time: 41:00)



So it might happen that today you run the algorithm and it take some time and tomorrow you run the same algorithm and it takes a different time. Because it depends upon what random numbers are selected and those random numbers selected decide the pivot and the pivot as you see is crucial in deciding how much time we are taking. Because if that pivot was a nice one which was splitting the things evenly and if it is going to take less time and if the pivots were turning out to be bad ones fairly skewed, then I am going to take more time. So we are going to see all of that. So now if you do not know how much time the algorithm is going to take and it is going to take some time today and some time tomorrow, then what do we analyze? What is that we can say? The average time or expected, which is also called the expected time.

What is the average we are doing over? So are we averaging over all possible inputs? Sequence of random numbers is generated in some sense. So the way to think of it is, fix an input. You are not going to change the input; we will run the algorithm today and tomorrow and day after and so on till the end of this course. And then you are going to compute the time and take the average and that will be the expected time for sorting that specific input sequence.

(Refer Slide Time: 41:35)

The slide is titled "Randomized Quicksort Analysis". It contains the following text:

- Let  $T(n)$  be the expected number of comparisons needed to quicksort  $n$  numbers.
- Since each split occurs with probability  $1/n$ ,  $T(n)$  has value  $T(i-1)+T(n-i)+n-1$  with probability  $1/n$ .
- Hence,

$$T(n) = \frac{1}{n} \sum_{j=1}^n (T(j-1) + T(n-j) + n-1)$$
$$= \frac{2}{n} \sum_{j=0}^{n-1} T(j) + n-1$$

Let  $T(n)$  denote the expected number of comparisons required by quick sort.  $T(n)$  is a function of  $n$ , the number of comparisons required to sort  $n$  numbers will depend upon **how many numbers you have depend upon it**. Let us recall what quick sort does. Quick sort first partitions, if it has to partition  $n$  numbers no matter what the pivot is. It is always going to require the same number of comparisons. It is always going to require no more than  $n-1$  comparisons. You can think of the partition process as every number is compared against the pivot and then all those that are less than the pivot are put on one side and all those that are more than pivot are put on the other side. So we always require  $n-1$  comparisons.

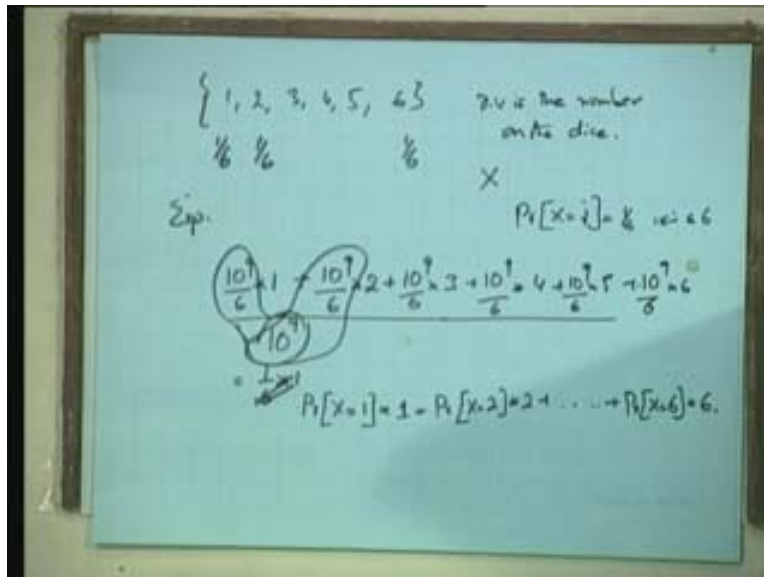
Every number has to be compared against the pivot otherwise we will not be able to decide whether it goes on the left side or on the right side. So this is the part of the partition. Depending upon what the pivot was, if the pivot was the  $i$ th smallest element then on one side how many elements am I going to get? Let's say  $i-1$  on one side and  $n-i$  on the other side and the  $i$ th element lets say I leave it at the right place. I have to quick sort those  $i-1$  elements, I have to quick sort those  $n-i$  elements. How much expected time am I taking to quick sort those  $i-1$  elements?  $T(i-1)$  is the expected time I take to quick sort these and this is the expected time I take to quick sort the  $n-i$  elements (Refer Slide time:42:53) So I will take this much amount of time in all. But what is  $i$  here? The  $i$  was the fact that the pivot was the  $i$ th smallest element that happens with the probability of one over  $n$ . So I am going to take this much time with the probability of one over  $n$ .

I am going to take time  $T(7-1) + T(n-7) + n-1$  with the probability of one over  $n$ . I am going to take time  $T(13-1) + T(n-13) + n-1$  again with the probability of one over  $n$  and so on. So the expected time taken is basically this sum (Refer Slide Time: 43:45) this quantity summed over all choices of, so I have replaced  $i$  by the  $j$  here. So this  $i$  is the same as this  $j$  here and each one of them is being picked with the probability of one over  $n$ . This is how you compute expectation. Expectations, so for instance just to give you an



examples those who are forgetting your expectations, I roll a dice. What is the expected value I see? How does one compute this? Each of the outcomes is equally likely, each occurs with the probability of one over six. I see one over one with the probability of one over six, two with the probability of one over six, three with the probability of one over six and all of that. So the expectation is one over six times one plus one over six times two plus one over six times three and so on. Whatever value we get that is the expected value you would see. What should I repeat?

(Refer Slide Time: 48:12)



Throw a dice so what are the possible values we get? It is 1, 2, 3, 4, 5, and 6. It is an unloaded dice, so each appears with the same probability. What is the expected value? What is the random variable? Random variable is the number that comes, so the random variable is the number on the dice. This random variable lets call it x, this random variables takes six different values. Each value takes the probability of one over six. Probability  $x = i = 1/6$ , i between 1 and 6. What is the expectation? Think of the expectation as just the following. You keep throwing this dice and keep recording the outcome. Keep doing this forever and then just take the average. Suppose you threw this dice one billion times. How many times are you going to see a one? One billion by six times. This is what the probability means, the probability that it is 1/6.

If you do the experiment sufficiently many times then this fraction of the times you are going to see this particular outcome. [Hindi Conversation] times your outcome was one. If I am looking at the sum of the outcomes this will be one sum. How many times do you see a two? How many times do you see a three and so on? So let me just complete this which is exactly the same as saying one by six into one, this quantity here.

What is this quantity? (Refer Slide Time: 47:32) This is just the probability of the event that x i equals one. So this is 1/6 or I could even have written it as probability, the random variable takes the value one times this one which is the value of the random

variable plus the probability  $10^9/6$  is the probability that it takes the random variable. The  $10^9/6$  by  $10^9$  is just the probability that it takes the value two. So probability  $x = 2$  into 2 and so on. And probability  $x = 6$  into 6. So that is what expectation is. One way of thinking of expectation is, if a certain random variable is taking a set of discrete values then you just compute the value of the probability with which it takes the value, times the value summed over all the possible choices is the expectation of the random variable.

So let's revert to our slides. This is going to be taking this value with the probability of one over n. And so we have done just the probability times the value summed over all the possible choices which is j varying from one through n. And again as j varies from one through n, this quantity varies from T(0) to T(n-1) and this quantity also varies from T(n-1) to T(0) which means every term T(0) through T(n-1) appears twice. So I can write this part of the sum as this and this n-1, we are summing it n times and then dividing it by n and so it's just plus n-1 separately. And this is a recurrence which we saw in the last class. It was the recurrence for the expected number of comparisons required to insert a randomly chosen permutation of n elements in a binary search tree.

(Refer Slide Time: 49:45)

The slide is titled "Randomized Quicksort Analysis" and contains the following text:

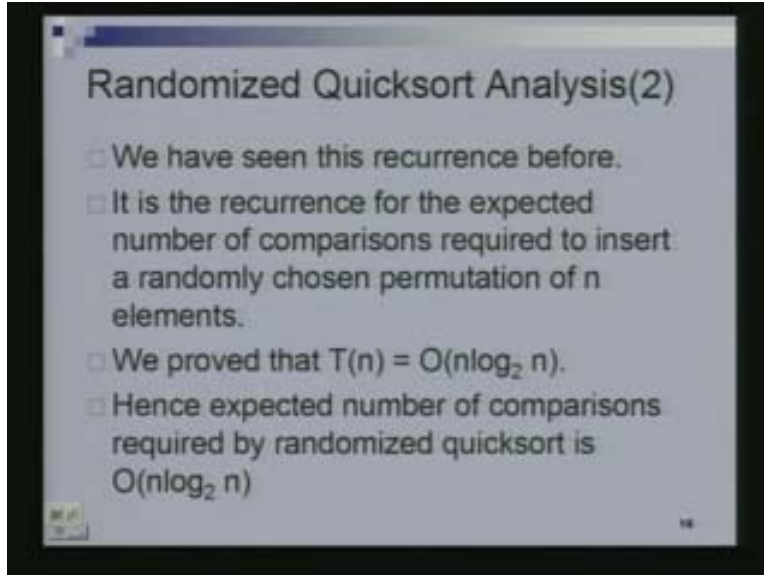
- Let  $T(n)$  be the expected number of comparisons needed to quicksort  $n$  numbers.
- Since each split occurs with probability  $1/n$ ,  $T(n)$  has value  $T(i-1)+T(n-i)+n-1$  with probability  $1/n$ .
- Hence,

$$T(n) = \frac{1}{n} \sum_{j=1}^n (T(j-1) + T(n-j) + n-1)$$

$$= \frac{2}{n} \sum_{j=0}^{n-1} T(j) + n-1$$

And what did we prove in the last class, we solved this recurrence and we showed the solution is  $n \log n$ . Hence the expected number of comparisons required by randomized quick sort is  $n \log n$ , the same recurrence. We solved it before and we are just using that fact. This is the analysis which we did and this is the expected number of comparisons required by quick sort. So let's quickly summarize the time taken by quick sort. Worst case time we said was  $n$  squared, the best case time was  $n \log n$ . We did not prove it formally? Why it should be  $n \log n$  and why it can not be less, but intuitively you can understand that the best thing happens when the two are roughly equal and then we argued that it will take  $n \log n$ . And the expected is once again  $n \log n$  and this behavior is similar to what we also in the last class.

(Refer Slide Time: 50:41)

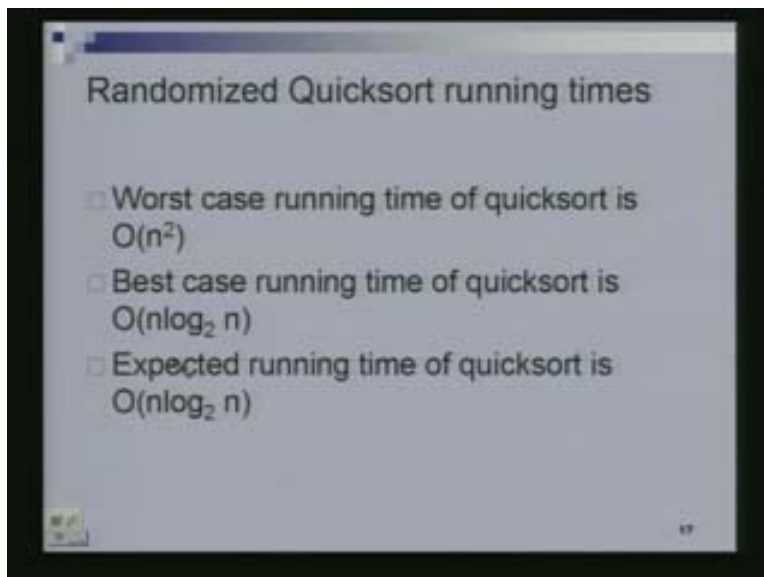


Randomized Quicksort Analysis(2)

- We have seen this recurrence before.
- It is the recurrence for the expected number of comparisons required to insert a randomly chosen permutation of  $n$  elements.
- We proved that  $T(n) = O(n \log_2 n)$ .
- Hence expected number of comparisons required by randomized quicksort is  $O(n \log_2 n)$

When I am trying to insert  $n$  elements into a binary search tree, what is the worst case time? It is  $n$  squared, it happens when my sequence is sorted. What is the best case time? It is  $n \log n$  again and if the first element I was inserting was roughly the median and the second element I was inserting was  $n/4$  and so on. And the expected time was  $n \log n$ . The crucial difference between what we are doing today and what we did in the last class. This is something that I have already said but let's just recap. What we have done today is that the running time of quick sort depends upon the numbers that are getting generated. If the same random numbers were generated then the running time would be the same for a given input.

(Refer Slide Time: 51:19)



Randomized Quicksort running times

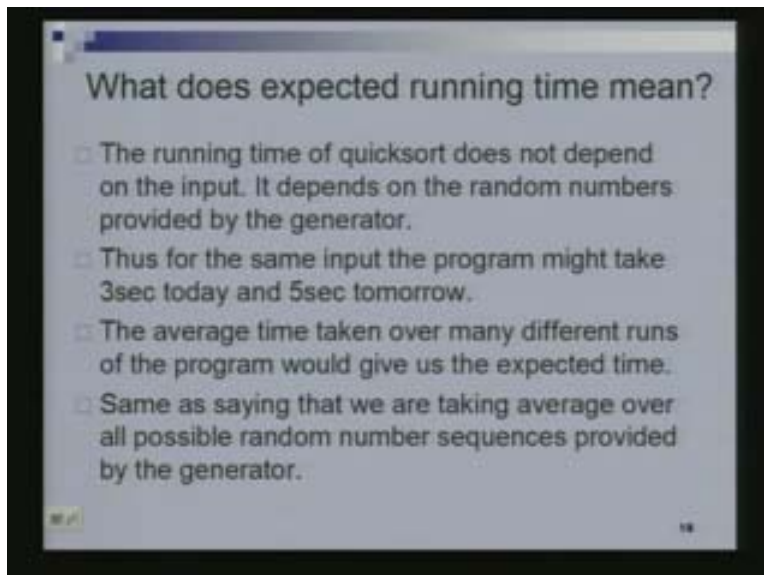
- Worst case running time of quicksort is  $O(n^2)$
- Best case running time of quicksort is  $O(n \log_2 n)$
- Expected running time of quicksort is  $O(n \log_2 n)$

So the same thing I said, I fix an input the running time could be some value today, it could be some value tomorrow because the random numbers generated could be different. What we have done is we have taken expectations over these different random numbers that have been generated, by saying that today we will compute what the value is, tomorrow we will compute what the value is so on and take the average. We are doing this for one fixed input and we said if I fix an input and I compute this value it turns out to be  $n \log n$ . That is what we did.

And no way we used the fact that, what the input was really. Come to think of it, no matter what the input is, your expected time is turning out to be  $n \log n$ . So somehow this is actually what some of the slides were also saying before. When we said we would take a specific element as the pivot lets say the last element as the pivot. We saw that the running time would depend upon what input was given to us. If the input was sorted we would take  $n^2$  time, but if the input was such that the last element was the median element then once again we would get this half and half split and so on.

So if I had taken a specific element as the pivot always then my running time would depend upon what the input sequence was or what the input order was. But I got around that somehow. I said let me randomly pick my pivot element and now what has happened? The running time does not really depend upon my input is, it depends upon what the random number choice is, which in expectation will give me a running time of only  $n \log n$ . [Student Conversation: So how does that make things if better we do not make any fix element.....] the algorithm time is independent upon the input. How does, no. So what we are trying to say here is that when we say it is independent upon the input, we are saying that no matter what input the adversary gives us. You are trying to may be beat my algorithm.

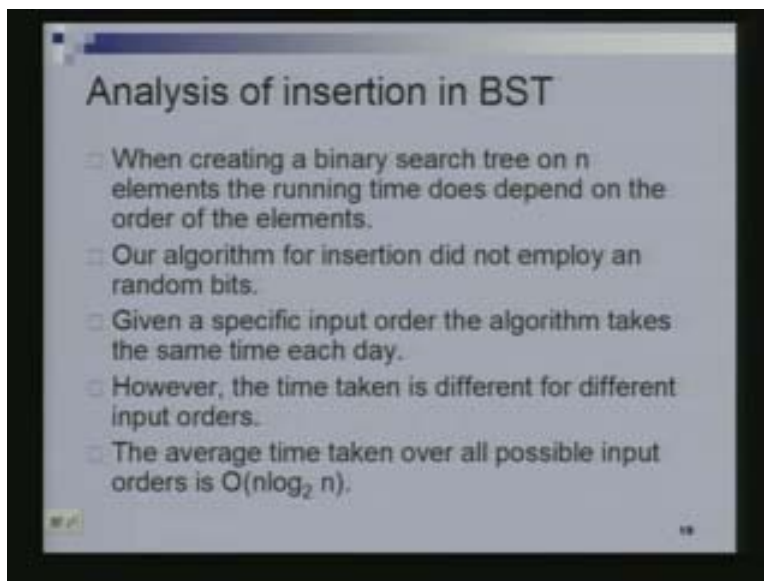
(Refer Slide Time: 56:27)



You want my algorithm to take as much time as possible. So the point here is that you cannot come up with any such sequence. No matter what sequence you come up, what sequence of numbers you come up, my algorithm will take a time which in expectation is  $n \log n$ , which quite often will turn out to be  $n \log n$ . [Student Conversation: I want to work somewhere. So there nobody is going to... We have to make an algorithm Particular kind of input may be if you look at this quantity that you want to make it independent of the input, then this kind we are making a random for that kind of input]

If we knew what kinds of inputs we were getting then perhaps it makes sense to design the algorithm for those kinds of inputs. But this is not what we are doing here because we are not designing the algorithm for a specific input sequence or specific kinds of inputs. We are saying we want to be able to consider all possible inputs and in doing that we want an algorithm which really does not depend upon what the input is, its behavior is independent of that. What did we do in the last class for binary search tree? And I want to make this difference very clear. For the binary search tree we were doing an average over what? We said take a particular input sequence and it is going to take a specific amount of time, whether you run it today or you run it tomorrow its going to take the same amount of time. That was not a randomized algorithm it was a very specific algorithm were no random choices being made. If you take the same amount of time no matter when you run it, but if I take a different input sequence it would take a different amount of time and if I take a third input sequence it would take a third different amount of time and so on.

(Refer Slide Time: 57:08)



And there what we were doing was, lets take the average over all input sequences that is over all possible  $n$  factorial different permutations. We took the average over all of them and then we got  $n \log n$ . Did you understand the difference? The recurrence is the same but there is a vast difference between these two things that we have done. One was what you call an average case analysis; we looked at all possible inputs that can be there of

numbers one through  $n$ , there are  $n$  factorial different permutations. So there are  $n$  factorial different inputs possible, we looked at all of them; we computed the time taken by the algorithm for each one of those inputs and took the average. Today on the other hand ours was a randomized algorithm, our algorithm was taking different times depending upon what the random numbers were. And today we were taking the average over the random numbers that were getting generated and not over the inputs, the input was fixed. With that I am going to end today's lecture. We saw quick sort and we did the expected time analysis for randomized quick sort.