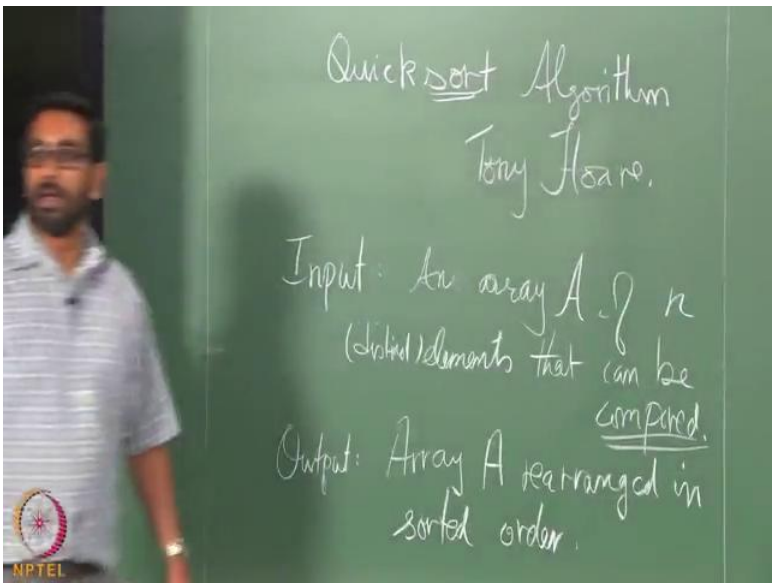
Algorithms for Big Data
Prof. John Ebenezer Augustine
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Lecture – 07
Randomized Quicksort

So, this we are going to talk about quick sort, which will be an example of an algorithm that will help us to understand how the randomize algorithms are developed? How they are analyzed? And this will also be an exercise for you in terms of learning how random variables worked? How probability theory helps the randomize algorithms? In that sense it is going to be a little exercise in it is nice pickers having taken an algorithms course. You are probably already familiar with quick sort the deterministic version. What we are going to do now is try to understand the randomized version, but let us remind ourselves what we are talking about.

(Refer Slide Time: 01:10)



We are talking about the quick sort algorithm and this was an algorithm invented in the sixties by Tony Hoare and it turns out to be extremely good and its variance of quick sort are used extensively in a variety of context, but quick sort the way it work does not necessarily give you very good asymptotic running time bounds, if you expected to be deterministic.

We will look at that more carefully, but for now let us remind ourselves how this

algorithm works and just to recall, we are talking about sorting which means that the input is any array A of elements; any array A of elements that can be compared. So, the easiest way to think about this is an array of integers and you can; given two integers compare. It could also easily work for comparison of strings, if you want to compare it alphabetically so on and so forth.

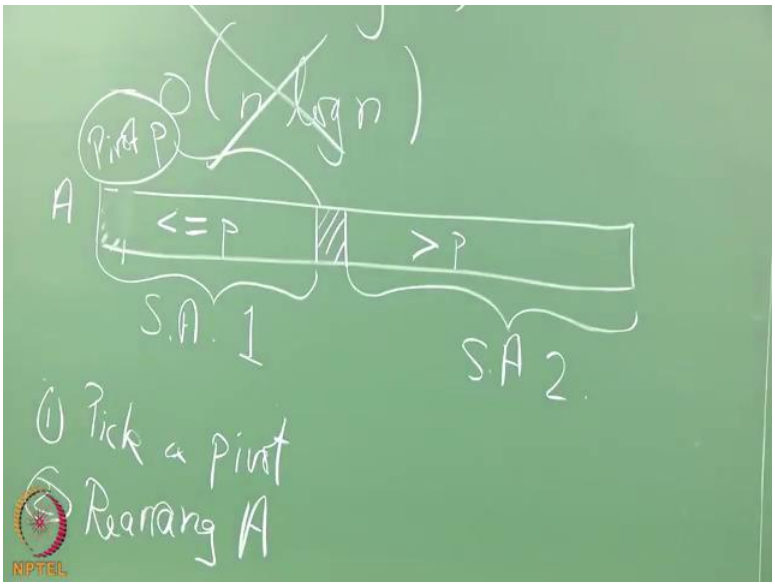
So, all we want is that these elements be comparable and for simplicity, let us assume that are these are distinct elements. This is not crucial requirement can be easily done away with, but for simplicity we are going to assume that and as you of course, might know the output is typically, let us even say that the algorithm and the array A itself, but it is, it should be rearranged, so that it is in sorted order.

This is the requirements for any sorting algorithm and let us thinks about few things or re recall a few things that we already know about the sorting algorithms. One is we are focusing on comparison based sorting. So, a fundamental bound that we have to respect is that the worst case any sorting algorithm is going to require omega of n log n time and here n is the number of elements. So, you can think, there is no hope of getting an algorithm that is going to run faster than n log n. In other words, an algorithm that runs in little o of n log n in this context simply that is not interest.

So, we know that and we have to leave by this bound. Let us look at how quick sort works in you can think of it as working in 3 steps, but this actually recursion involved. So, at high level you consider the full array, this is your array A. The first step is pickup the vector well. This is our array A let us pickup pivot element and without loss of generality, for now we are going to pick the first element. Let say this is the pivot element and then the second step would be rearrange the array A such that elements that are less than the pivot and let us call this pivot p rearrange this arrays, so that elements that are less than the pivot come to the left of the pivot. So, this anything that is or let say less than or equal to p gets placed to the left of the pivot and then you place the pivot p.

This now, we are rearranging the array and then all elements that are greater than p are placed to the right of the pivot and now after this rearrangement we can split the array into two sub arrays.

(Refer Slide Time: 06:45)



This is sub array number 1 and this is sub array number 2 and all we have to do now is recursively apply this algorithm on the 2 sub arrays. So, this provides us with a way to sort the algorithm and it is an easy exercise to see that this algorithm is going to be sorted and, so that will be left as an exercise for you to prove that this algorithm works and. Let us now think about how good this algorithm is notice that this algorithm works by this divide and conquer paradigm. Each time the problem the input is this array A as when we recurse, we are working on a smaller part of the array and each time you recurse the problem size gets broken into a smaller piece and then we know how to put the solutions together and will the final sorted array.

So, this divide and conquer can go wrong. How could this go wrong, if you are not familiar with this now, might be a good point in time to pause and just think about why this algorithm could go not wrong, but good operate very slowly think about what could not happen. So, pause for a little while and come back if you need to, now let us just to complete the story let see what could go wrong. So, we choose the pivot to always be the first element in the array.

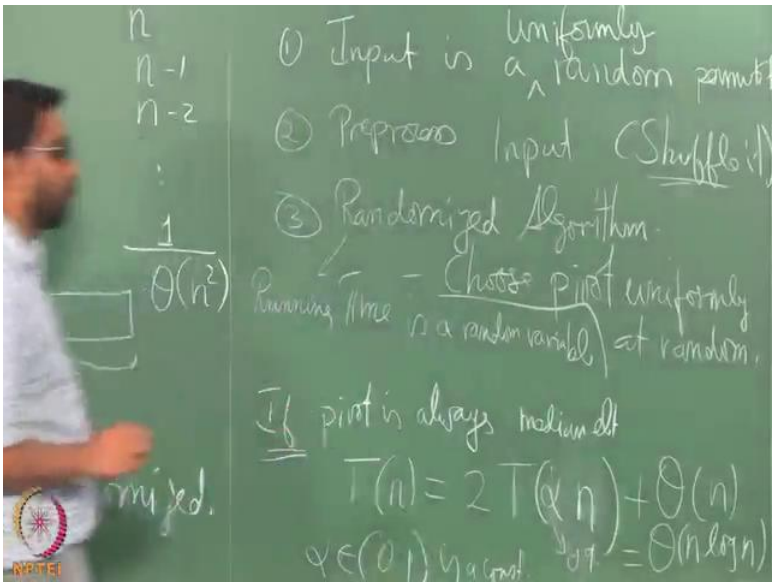
If the array is already sorted what is going to happen every time you choose the pivot, this going to be no element to the left of this pivot all the elements are going to be the up to the right which means that the one side of the problem will be of size 0, the other side the problem will be of size n minus 1. So, yes the problem size reduces, but the problem

size does not reduce significantly. So, this will roughly correspond to; while the first instance you are dealing with n to you will spend about theta of n or let us just approximate it n time steps to rearrange this array the first in a first recursion.

In the next one, it will be your problem size would not have reduced much less than that it would have been come n minus 1, n minus 2 and so on. Up to 1 and this will add up to something theta of n square. So, this unfortunately does not give us good running time. In particular, it is far away from what is potentially achievable. We know that n log n is a lower bound, but we are getting a worse case running time of theta of n square and that is in under this form of input and so that is something that we would like to fix. We would like to understand this the quick sort algorithm maybe make a few modifications to achieve our goal which is n log n, but there is unfortunately no hope of doing that with as long as we insist on a deterministic choice of the pivot.

So, that is again a good exercise for any deterministic choice of pivot can you design an input array sequence, so that the worst case, the running time will be theta of n square. Now, that is a good exercise for you as well. So, think about, but much of this that situation occurs because the input has a specific form and the algorithm does something very specific deterministic to be precise and that just does not work out well for us. How do we break out of this bad situation and that is where you would help for us to look at this from randomize perspective. Now, we will have to think about how to apply randomization? There are few ways to think about it. Let me find out one way to think find them out one.

(Refer Slide Time: 11:36)



Let us assume that one way to bring in randomization is to assume that the input is a random permutation. If you want to be a little bit more pedantic noise, we can add uniformly random permutation, what this means is that given n items there are n factorial possible permutations and among those n factorial possible permutations, we have chosen one uniformly at random.

So, each permutation occurs with probability 1 over n factorial and if this is the case then let us intuitively thinks about, what happens when we apply this simple quick sort algorithm. Well, when the first pivot is chosen it is going to be very low probability. There it will be one pivot that does a very bad job of splitting the problem size into two and problem into two sub problem. So, first the pivot is likely to be somewhere in the middle and if the pivot is likely to be somewhere in the middle then the two problems or the two sub problems that we will get will be of roughly equal size and that is significantly helpful. Let us reason that out a little bit more carefully. So, it is somehow to get a feet wet. Assume, magically that the pivot is exactly the median each time.

If somehow magically the pivot always ends up being the median element in the sub array that is being worked on, then the running time is given by the following recurrence, let say t of n is the running time when the problem size is n. The nice thing is since you are going to use the median, to split the problem we are going to get two sub problems each one of them is going to be no more than the sealing of n by two plus you will have

to spend some time arranging them the smaller elements to the left of the pivot, larger elements to the right of the pivot and so on.

So, that is going to be theta of and this we know is another exercise for you, is going to be theta of n log n. If you can do this perfectly of course, this is very unlikely if and this only works if the pivot is always the median element that is not going to happen, but the intuition is very important because now when we come back to this random permutation, the intuition is even this intuition is actually quite strong. It does not even have to be exactly splitting the problem into half, it could even. For example, this n log n would work even if the problem gets split into some fraction.

So, let say the problem gets split into some alpha n, where let say alpha is a constant between 0 and 1 open interval and alpha is a constant and it must be a constant and if you think about this, what this means is that the split need not be perfect or it has to be a fractional split. So, even if you can ensure that the pivot is somewhere in the middle, somewhere between say the tenth percentile and the ninetieth percentile that is also fine because then the two problem size is the larger of the two. Problems will not exceed 90 percent that will be corresponded alpha equal to 0.9 as long as alpha is a fixed constant within this open interval were fine.

We will still be able to get theta of n log n. So, that gives a lot of hope because this when we use randomization would likely to get the element. The pivot element to be somewhere in the middle, so that gives us a hope that this will work and it is in fact, correct, but the problem with this is you cannot always assume that the input is going to be uniformly at random because the context the application context could mean that you are sorting an array that is almost sorted already. You are just correcting the few places where it is not sorted in which case this algorithm is going to be; you cannot make the assumption that the input is uniformly at random. So, this is a small fixed to it

So, the fix is the following you do not make the assumption that the input is uniformly at random, you do that pre-processing step. In other words, shuffle it here is another small assignment for you think of a way to shuffle an items. So, that the output is a random permutation of those n items. So, that is the nice little homework for you as well. So, if you can shuffle it even if the actual input is a bad input, for example, even if the actual input is, already sorted when you shuffle it you are going to essentially mimic input

being uniformly at random. So, that is a possibility, but this whole requires you to have written a pre-processing step.

Let us see, if we can avoid that. So, the third probability is that you actually do the randomization within the algorithm. So, the algorithm is no longer going to pick the pivot as the first element in the array neither is it going to pick the element always is the last element or any deterministic way. Instead, what it is going to do is it is going to consider all the n items in the array and choose a pivot uniformly at random and then when you get to the sub problem. Well, these are your options for the potential pivots, you choose one uniformly at random at each recursive step, you choose a pivot uniformly at random that is randomization within the algorithm.

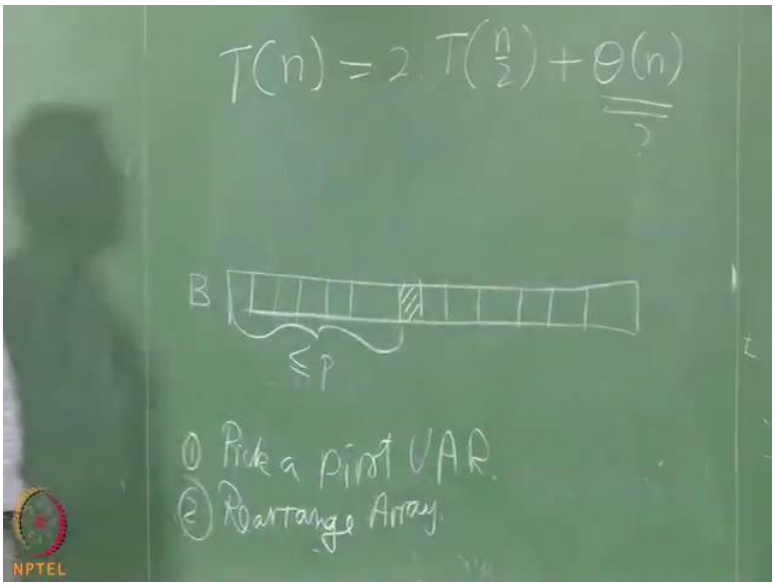
So, essentially if you think about it all of these three options are essentially equivalent might take a little bit of effort to figure out that this three is also essentially equivalent to the other two options, but I will let you think about it at your own time. So, what we are going to focus on now is to take this option 3 and analyze it and show that it actually has a running time that matches this n log n at lower bound, but keep in mind that when you randomize the algorithm the running time is no longer a deterministic quantity.

The running time is a random variable and that is something to keep in mind. So, you cannot really say in give a direct bound on the running time, but rather we are going to focus on is the expectation of this running time. We going to give bound not on the running time directly, but the expectation of the running time, that is let us see, how we can arrive with that.

Student: Sir, could you please explain me the contribution of theta of n in the running time of this random variable.

So the question that is being asked is the contribution of theta of n within the running time.

(Refer Slide Time: 21:20)



So, represent this is what you are asking about t of n equal to that is just approximated to n by 2 plus theta of n and I represent your concern is this theta of n, how this is arrived? Let us recall the algorithm, pick a pivot and the second step this is where it matters array and if you recall, we rearrange the array. So, that the elements smaller than the pivot are to the left elements that are larger to the pivot or to the right.

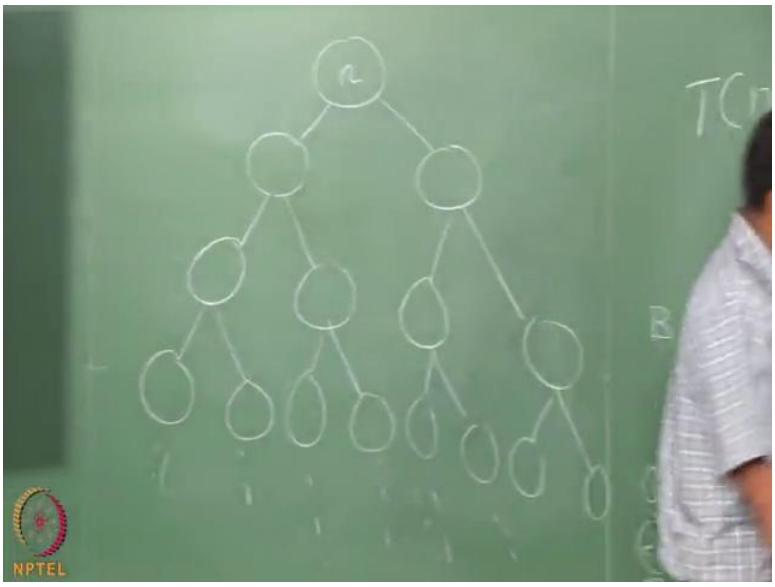
So, how will that happen? So, once you have the array what you have to do is put all the elements to the smaller that are the smaller than the pivot, first into the array; is one way to do it. Let me give you a simplest way that I can think of you know put all run through the list of elements in the array. You can think of this for simplicity, you can think of this as another array B in which you put all the elements that are smaller than the array as smaller than the pivot first.

So, you start filling in the array, but you are going through the entire list in array A to fill in this array B. Once, you have exhausted the array A these are all the items that are smaller than or equal to p and then you put the pivot then again whatever is left whatever is not already been used over here, you are going to fill it up over here this step of rearranging the items and putting them into this array B is what gives you this theta of n. You have to go through the entire list of array elements to be able to create this array B. There are some fancy things you can do to simplify, but this is the idea all right.

So, now coming back to the randomized algorithm, think about how we chose the pivot.

We choose the deterministically in the earlier case, we want to do, we going to pick the pivot uniformly at random, then we are going to rearrange the array and then third steps will be to recurse. Now, the operation of this algorithm can be thought of in this following manner in this sort of a divide and conquer tree.

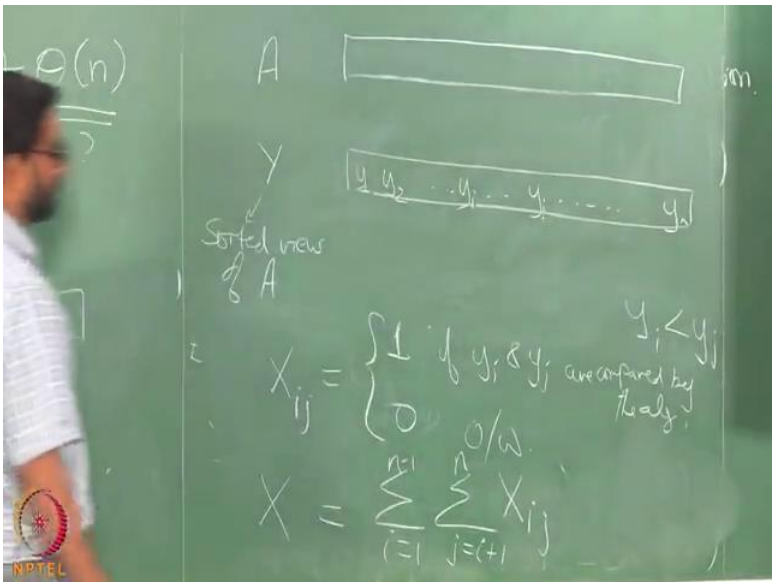
(Refer Slide Time: 24:24)



Problem size to begin with is n and then chooses a pivot uniformly at random. Then whatever happens to be on the left side of the pivot becomes the left sub problem, whatever happens on beyond the right side of the pivot happens to be the right sub problem and then again, now for each of these sub problems you recurse which means you again have to choose, a pivot is basically this node here in this tree corresponds to a sub array and you further sub divide and divide and conquer and so on.

So, we want to get a handle on the running time of this algorithm and as you may be aware, in a lot of sorting algorithms we do not really compute the actual running time, but we rather compute the number of comparisons. So, we are going to be concerned about the number of comparisons. We are going to approximate the running time by the number of comparisons which means that we need to think about a way to break down or understanding of the total number of comparisons into smaller pieces that we can handle and then we can put them together. So, if you are wondering how this will works this is where the notion of linearity of expectation will come into play. So, let us think through this a bit carefully.

(Refer Slide Time: 26:09)



So, what we are going to do, now remember our input array is this array A for the purpose of analysis, let us consider the array, let us call it y, y is nothing, but the sorted view of basically an array A that is already sorted. Obviously, if you know the array y you can just output that. So, you do not know array y, but what we are going to do is just analysis for the purpose of analysis, you are going to look at array Y. So, y comprises y 1, y 2, y i y j and these are all sorted. Now, therefore, y i is less than or equal to y j and now what we are in this the sorted view.

So, let us assume that they are all distinct elements. So, y i must be therefore, strictly less than y j in order to get a handle on the total running time. Let us define a very simple random variable corresponding to these two elements y i and y j. We are going to call a random variable X ij and this X ij can take one of the values it can be either 1 or 0, it is a 1 if y i and y j are compared by the algorithm and because of randomisation there is no guarantee that y i and y j will be compared take maybe compared they may not be compared.

So, if it is compared then X ij is going to take the value 1. If it is not compared X ij is going to take the value 0, otherwise and if you think about it this algorithm is only going to compare two items at most once. So, if you want to handle on the total number of comparisons, let us denote that by the random variable x that is essentially equal to this double summation.

This is essentially the easiest way to think about it, for the moment, ignore these two summations. This is X ij is just compare it is going to be 1 when y i and y j are compared, but then that is just one comparison you have to account for every possible comparison and how do we do that well that is where this double summation comes in i runs from 1 to n minus 1 and the moment you fix an i, you do not want to compare, you do not want to consider j that is prior to i you want to consider and because you want every unique pairing.

So, we want to consider j values that are larger than i. So, once i is fixed j runs from i plus 1 all the way to n and with these two summations taken over X ij's we will get very possible i j combination with i not be equal to j. Which you have questions; yes please.

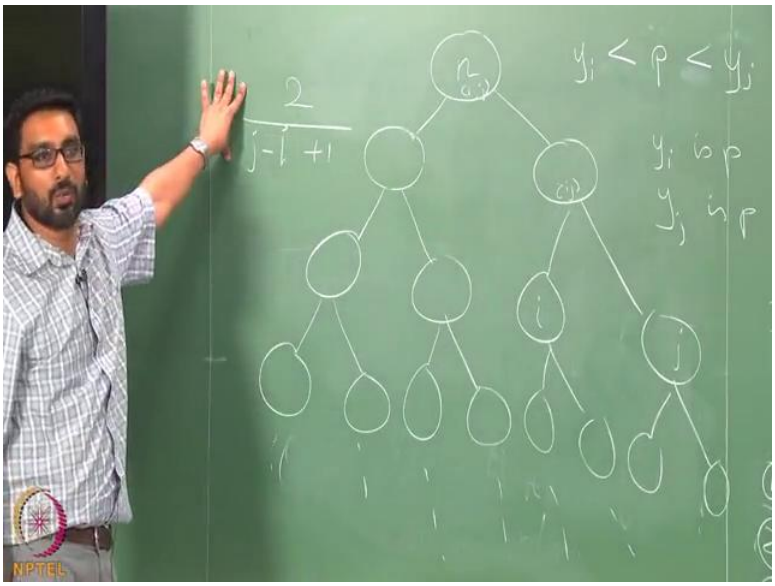
Student: Sir, actually you said about this comparison within y i and y j.

Right.

Student: Is it like any one of them will be a randomly selected pivot?

We have to discuss that, your question is a naturally laid in because now when we consider the values at x I, when X ij will be 1 and when X ij will be 0 that will depend on whether y i or y j were choose as pivots right now y i and y j are just two elements within this sorted view. So, the question therefore, is what values can X ij take and it is well, 1 or 0, but when will it take the value 1 that is the question. So, think about go back to this picture where we show the recursion. So, let us fix i and j. So, this i and this j as fixed we want to know when these two elements are going to be compared because when these two elements are compared X ij value is going to be 1.

(Refer Slide Time: 31:29)



So, initially i and j are both inside the same array that is being considered and may be they will continue to be in the same sub array. So, maybe they both go into the right sub array here.

Eventually, this at the end it is only going to be singletons elements in the leaf. So, at some point i and j will have to split. So, let say without loss of generality that i's goes to the left here and j goes for right here. So, something happened over here in this sub problem that is split i and the j the y i went to the left sub problem and y j went to the right sub problem, what happened why would that spilt happen? Well, it happened because a pivot was chosen and that pivot was had a value that is somewhere between y i and y j.

This is when a pivot of this form was chosen that pivot was compared with y i and it was put on left sub problem and pivot was chosen compared with y j and y j was put in the right sub problem and that is why, i and j got spilt into the two sub problems and if let us consider the case where the pivot was strictly between y i and y j and not neither y i nor y j.

If that is the case upon to this point in time the pivot was always fully to the left of y i or fully to the right of y j which means that y i and y j themselves were never compared and even at this point when they were split, since the pivot is fully between in particular neither y i nor y j y i was never really compared with y j y i was compared with the pivot

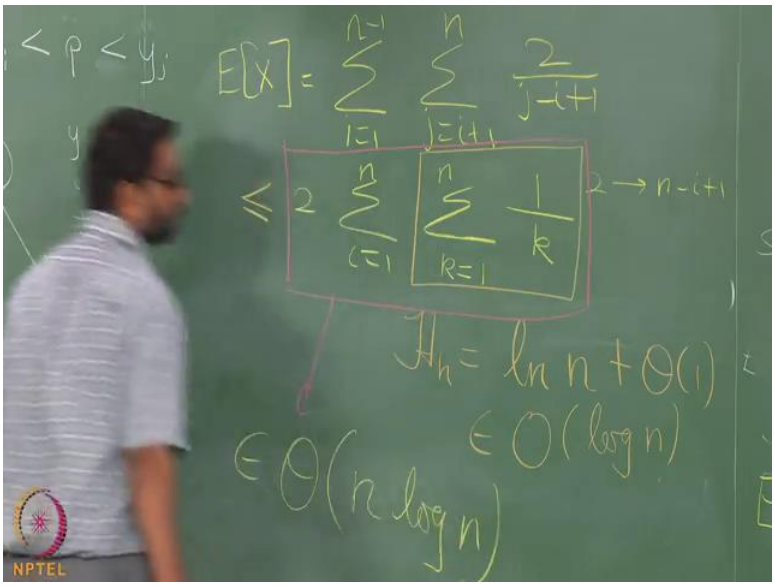
and y j was compared with the pivot, but these two were not compared. So, every time the pivot is chosen in the middle excluding the 2 y i and y j's, then no comparison take place the only way a comparison can take place, if either y i is p or y j is the pivot these are the only two possibilities under which these two elements y i and y j can be compared and what is the probability of that happening well how many elements are there between y i and y j well.

So, in this picture if you think about it is y i y i plus 1 and so on up to y j. So, this is going to be j minus i plus 1 possible element in that range. Out of them there are 2 elements if chosen as the pivot will imply that y i and if the pivot is chosen either as y i or y j when the two elements will be compared. So, the two cases, two possibilities under which they will be compared, this is the probability with which X ij is equal to 1 and it is going to be 0 otherwise.

So, we can write that p r and if you think about this is also the expected value of X ij because X ij can be 1 with this probability and the probability 0 with the remaining probability. So, i expected value of this random variable X ij will be nothing, but 2 over j minus i plus 1 and just the little intuition if the two elements y i and y j are far apart the denominator is going to be large in which case the probability that there will be compared is small if they are close to each other, then that probability increases, so that you should think about why that is the case and if you think about how that has a bearing on the rest of the argument.

Now, the expectation of this x i j, but we really want is there expectation of this random variable x which captures the total number of comparisons we want e of x that is the expectation in which is e of this whole quantity and nice think is when it is expectations you can apply expectations over the sum of the number of random variables you can apply the linearity of expectation. So, you can simply take the expect inside you can write it this way and if you can write this way and we already know a formula for this.

(Refer Slide Time: 37:18)



So, let us try to simplify this whole thing e of 1 to n and going to actually substitute the x value here remember e of X ij is 2 over j minus i plus 1 and that is equal to let us simplify things a little bit. So, let me be a bit sloppy here 2 and here I am going to run i equal to 1 all the way to n.

Now, this will only increase the quantity on the right hand side, but I do not care about it I want an upper bound on the running expected running time. So, then let us see what we are going to do here is. Now, when we used a different variable for iteration, here I am going to let this run, let see what is the smallest value that this denominator can take this denominator can take the smallest value, it can take is 2 because j is definitely larger than 1 by at least 1. This quantity is at least 1 plus a 1. So, this summation will if you think of the denominator it will be 2 then it will be 3, 4 and so on and the largest denominator can value can be is when j equals n, in which case the denominator will be n minus i plus 1.

So, the denominator will run from 2 to n minus i plus 1. So, we are going to simply that. So, we are going to run the denominator from 2 to n minus i plus 1 and write it as 1 by k and simplify this further. So, this is running up to n minus i plus 1, I am just going to make it run all the way up to n and an even let say I started at k equal to 1, remember of all this I can do because I am looking for an upper bound and I am being a bit sloppy here, but to make the analysis simpler.

So, this quantity is nothing, but the nth harmonic number this is denoted h n and this is

equal to lon n plus theta of 1, this is the known quantity and so essentially this is O of log n and this what is within this box is an O of log n quantity and we are summing it n times plus times 2. So, as a result this whole quantity belongs to theta of n log n. This is of by the way theta. So, this whole quantity is now theta of n log n.

So, what we have able to show here is at the expected number of comparisons, which is the surrogate for the running time the expected number of comparisons is at most theta of n log n this is. So, what this tells us is that on expectation randomized quick sort is going to have a running time that is of the order of n log n, there still can be bad cases that it could be times when the algorithm runs a lot slower than that, but it is going to actually be rare.

So, in a subsequent lecture when we talk about Chernoff bounds, we will look at the same quick sort. We will actually try to prove that the quick sort is not just good on expectation, but it is actually good with high probability. We are going to actually be able to say something stronger about this random variable. We are still going to focus on this particular random variable, but what we are going to be able to say is that this random variable x will be of the order of n log n with high probability with overwhelmingly high probability, but that will require a little bit of knowledge on some other probability theory tools that we will look at in a subsequent lecture. So, any questions, no questions so, well done.

Thanks.