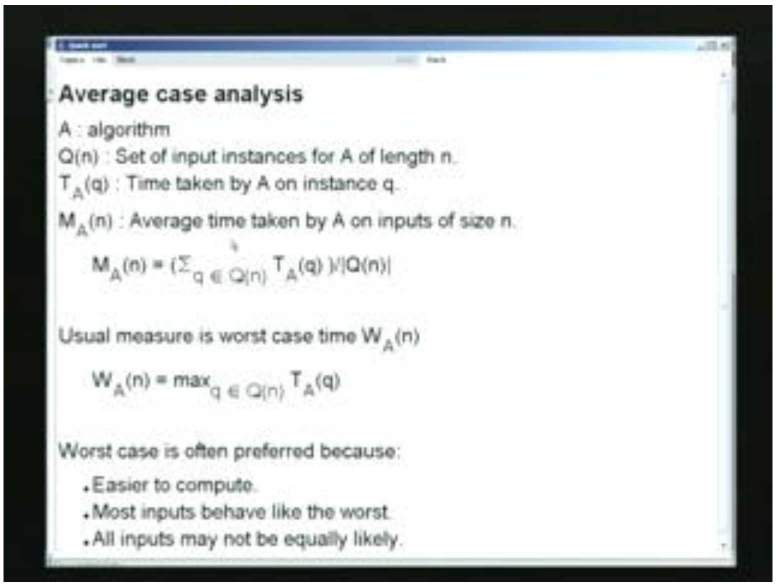
Design and Analysis of Algorithms
Prof. Abhiram Ranade
Department of Computer Science Engineering
Indian Institute of Technology, Bombay

Lecture - 22
Average Case Analysis of Quicksort

Welcome to another lecture on Design and Analysis of Algorithms. Our topic for today is Average Case Analysis of Quick Sort. Let us begin by discussing Average Case Analysis.

(Refer Slide Time: 01:07)


Average case analysis

A : algorithm

Q(n) : Set of input instances for A of length n.

$T_A(q)$: Time taken by A on instance q.

$M_A(n)$: Average time taken by A on inputs of size n.

$$M_A(n) = (\Sigma_{q \in Q(n)} T_A(q))/|Q(n)|$$

Usual measure is worst case time $W_A(n)$

$$W_A(n) = \max_{q \in Q(n)} T_A(q)$$

Worst case is often preferred because:

- Easier to compute.
- Most inputs behave like the worst.
- All inputs may not be equally likely.

Let suppose A be an algorithm. Q is the set of input instances of A. And let us make it, make Q be a function of n. So, Q of n is a set of instances of instances for algorithm A of length n. T sub A of q is the time taken by A on instance q. Given this, you can define the average time taken by A on inputs of size n, which we might write down as M sub A of n. M could be interpreted as mean for example.

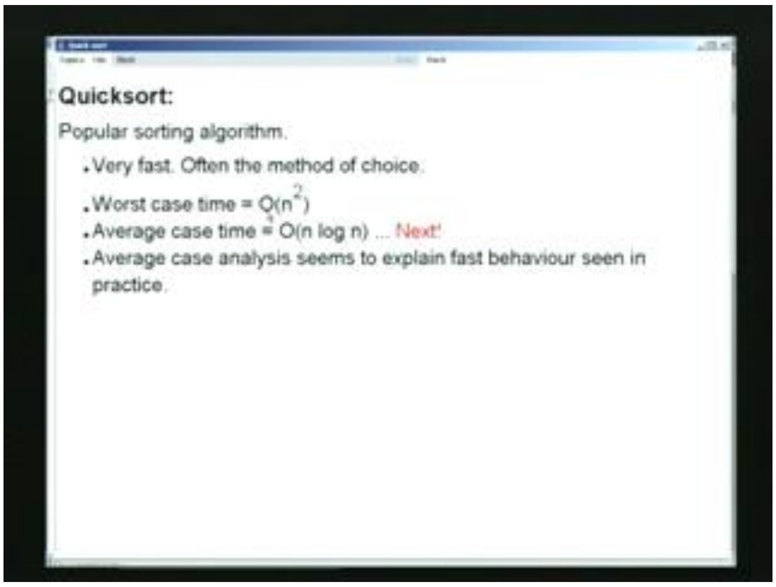
And M sub A of n is defined as sum over all instances in the set Q of n. Or sum over all instances of size n of the time taken by those instances, divided by the total number of instances. So, this is the usual definition of what we mean by an average. This is not the most popular definition of course. The definition we usually use is the so called worst case measure. Here the worst case time of an algorithm A on problems of size n, is defined as the maximum over all q of this.

The maximum time taken by any input instance of size n is defined as, the usual measure or the worst case time. There are several reasons for doing this. Worst case is usually easier to compute than this average. When we talk about the average, in some ways we have to talk about all input instances. Whereas, often it is easier to deduce what the worst instance is going to be. And, then we can just worry about that. For many algorithms, most of the inputs behave like the worst input anyway.

So, in which case it does not really matter, it is not really necessary to take the average in any case. Very often may be perhaps, the average case is easy to compute if at all. And it might still not be preferable, it might still not be very popular because, in practice we do not know which input instances are likely to appear more frequently. If some instances appear more frequently, then in this mean expression we would have to wait those instances more heavily.

Therefore, if we just take the mean, then that is not an indication of what might happen in practice. And therefore, again we do not really focus so much on the average case analysis. Worst case on the other hand might be conservative, but at least we know that it is conservative. And therefore, at least we can give some guarantees. Our topic for today is quick sort however, where average case analysis turns out to be quite useful and quite interesting.

(Refer Slide Time: 04:14)



Quicksort:

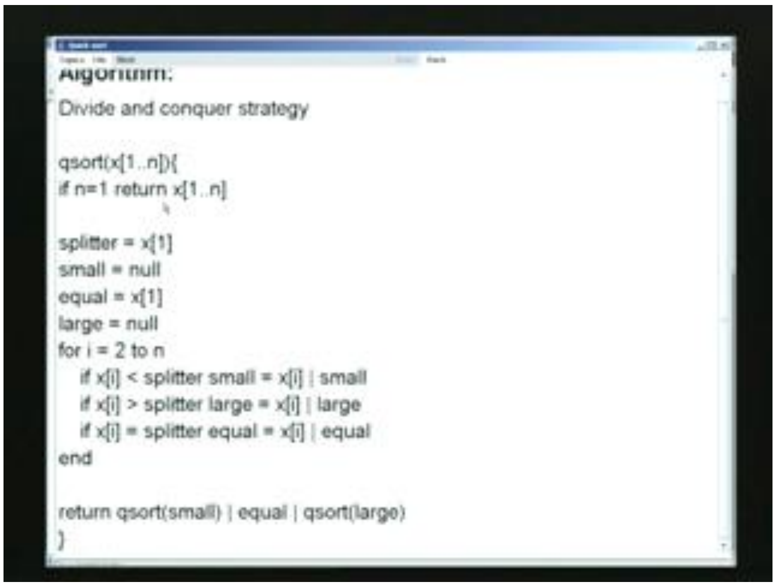
Popular sorting algorithm.

- Very fast. Often the method of choice.
- Worst case time = $O(n^2)$
- Average case time = $O(n \log n)$ ... Next!
- Average case analysis seems to explain fast behaviour seen in practice.

So, let me say a few things about quick sort. Quick sort is a popular sorting algorithm, perhaps the most popular and the most commonly used in practice. It is very fast. And as I said, it is often the method of choice. The worse case time of quick sort is O of n square. The average case time on the other hand is O of n log n and will see this quite soon.

So, in some sense the excellent performance in practice might be better explained by the fact, that the average case time is O of n log n rather than by focusing on the worse case time. So, let us now take a look at this algorithm. Quick sort is based on divide and conquer strategy. And the algorithm is something like this.

(Refer Slide Time: 05:08)



Algorithm:

Divide and conquer strategy

```
qsort(x[1..n]){
if n=1 return x[1..n]

splitter = x[1]
small = null
equal = x[1]
large = null
for i = 2 to n
    if x[i] < splitter small = x[i] | small
    if x[i] > splitter large = x[i] | large
    if x[i] = splitter equal = x[i] | equal
end

return qsort(small) | equal | qsort(large)
}
```

So, as input we take an array x 1 through n by writing x 1 through n, I simply mean that x is an array whose length is n. This is an array in which we have keys. We can think of these keys for the minute as some integers perhaps. And our idea and the goal of quick sort is to sort them. That is let us say that the smallest keys have to come at the beginning and the largest ones have to go towards the end.

We begin quick sort by looking at the best case first. So, the best case just checks whether, this is an element this array has only one element. If it only has one element, then the array is sorted trivially. And therefore, we just return that array. Otherwise, we pick something which we will call it splitter. And that splitter is chosen to be the first element of this x. The first key is the splitter. Then, we built three lists.

A list which will call small is going to be small, which contains all elements of x which are smaller than the splitter. A list which we will call equal, which will contain all elements of x which are equal to x 1 and so, we begin by putting x 1 into equal. I should perhaps said less over here. But, will not we are not worrying, we are not very careful about this. And we will not be very careful about this throughout the course.

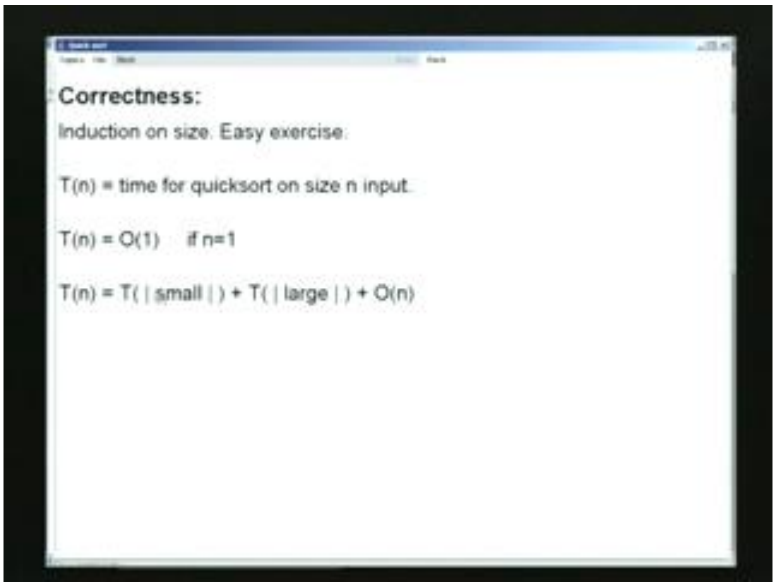
So, we will I just tell you that, we have just made equal just single list, a list with a single element. And we will also construct a list, which we will call large. And large will contain all the elements of x, which are larger than x 1. So, right now it has been initialized to null and small has also been initialized to null. This loop is simply going to build up the lists, as we just described. So, first step if so for every element other than the first.

We check whether it is smaller than splitter in which case, we add that element to small. If it is equal to splitter, then we add it to equal. If it is greater than splitter, we add it to large. So, at the end of this loop all the elements have been put into the proper lists. So, now it is simply a matter of recursion. So, small is a list which contained all small elements. So, we call qsort or quick sort in this list.

So, as a result we will get these elements sorted. Now, these elements are all guaranteed to be smaller than the elements in the list equal. And those in turn, are guaranteed to be smaller than the list in large. But we do not append large immediately over here, we quick sort it. So, as a result we have a long list which is made up by appending three lists. But, which in turn is guaranteed to be sorted.

So, this is how quick sort works. So, as I said it is divide and conquer strategy, the division part is where the interesting work happens. And, then there is a conquer part and then the combined part is trivial. Correctness is quite obvious here. You can do a induction on size, if you want to prove it formally. And I will leave that as a easy exercise. So, now you wanted to analyze this algorithm.

(Refer Slide Time: 08:36)



Correctness:

Induction on size. Easy exercise.

T(n) = time for quicksort on size n input.

T(n) = O(1)   if n=1

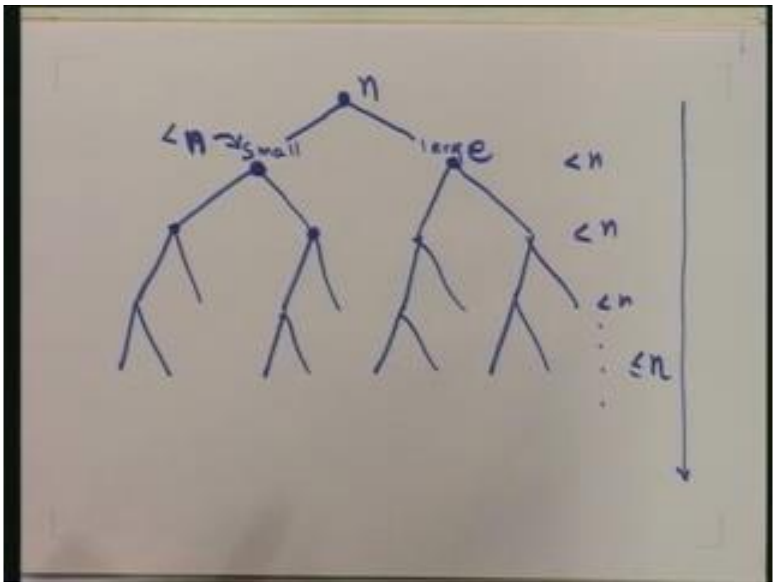
T(n) = T(| small |) + T(| large |) + O(n)

Let me use T of n to denote the time for quick sort, on size and input. So, right now I have only written n over here. But, I will actually have a specific input in mind. Just for the minute. Later on we will worry about average cases or the worst cases or whatever. So, right now let us say this is for a particular instance. So, T of n is the time taken by that particular instance. So, how do we analyze this? Well.

Usually if we write something like this, we try to establish the recurrence. So, of course no matter what input instance we feed. If it has length of only 1, then the time taken is constant. So, that is what is you write down first. Then, we need to find out how the recursion happens. So, let us just go back to the algorithm. So, over here the recursion happens by calling quick sort on small and calling quick sort on large.

And before that, we have a loop which runs about n times. So, the result we have O of n time for the loop. And T of large or T of the cardinality of large is the time taken for that instance, for further for evoking quick sort on the list large. And T of small, is the time taken for invoking quick sort in the list small. Now, we can analyze this using the recursion tree. So, this was our basic recurrence. So, let us draw a recursion tree corresponding to this.

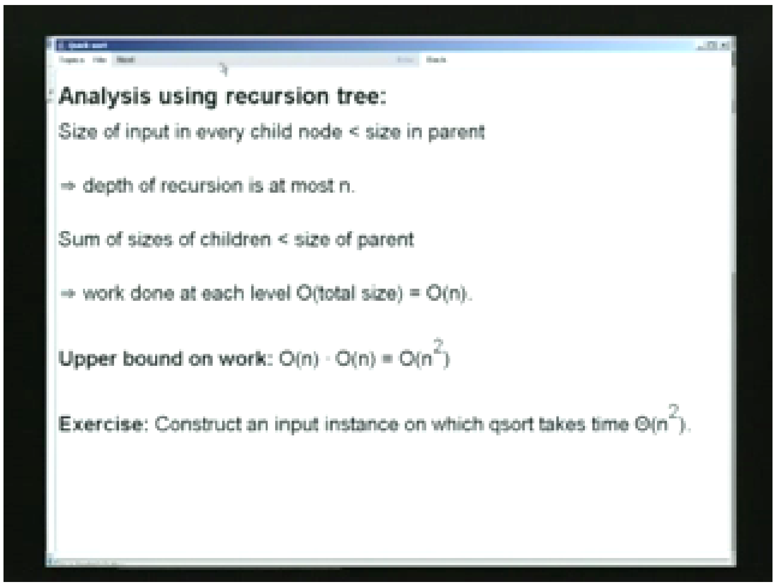
(Refer Slide Time: 10:28)



So, we start off with the problem of size n. And, then as per this recurrence we break the problem into two pieces. One is the small list and the other is the large list. So, this is the small part and this is the large part. This is a problem of size n. This is the small part, this is the large part. If this is the small part this is the large part, then we are going to call quick sort recursively on these. So, furthermore this problem will get split, this problem will get split.

Of course, if the problem is going to get split, may be one part one side could be smaller than other side could be larger and so on. And may be one of the lists could be empty in which case of course, this whole thing terminates. So, in general it is going to keep on splitting. May be once in a while, a list terminates and things keep on going in this manner. So, what do we know all about this.

(Refer Slide Time: 11:38)



Analysis using recursion tree:

Size of input in every child node < size in parent

⇒ depth of recursion is at most n.

Sum of sizes of children < size of parent

⇒ work done at each level O(total size) = O(n).

Upper bound on work: $O(n) \cdot O(n) = O(n^2)$

Exercise: Construct an input instance on which qsort takes time $\Theta(n^2)$.

Well, here is the first observation. So, if this node has size n, then we know that in this node the number of keys which are going to be present is definitely going to be less than n, in this node. Or in fact, in this node as well, so what does that mean? So that means that as I go down from here along any branch, the size of the instance has to decrease. So, which means I cannot go down too far.

So, I start the instance of size n. It has to decrease therefore, that means this height has to be utmost n. That is the first observation. The second observation is that, if I look at any node its children have a certain size. But, that size adds up to something strictly smaller than this. So, if I look at the size over here, it is n the size over here has to be less than n. The size over here in fact, has to be smaller than this for these two.

And for these two, it has to be smaller than this. So, this also has to be less than n. This also has to be less than n. So, if I look at any level, the size of that the sum of the sizes of the problem at that level have to be at most n. But now, if we go back to our problem our algorithm at each inside the body of each invocation, we do work or we do work proportional to the size of the problem.

So, which means corresponding to each node over here we are going to do work, which is proportional to its problem size. So, now if I look at the total work done here, it is going to be O of n. Because, it is going to be proportional to this problem size this problem size added up, it is going to be proportional to O of n or it is going to be O of n.

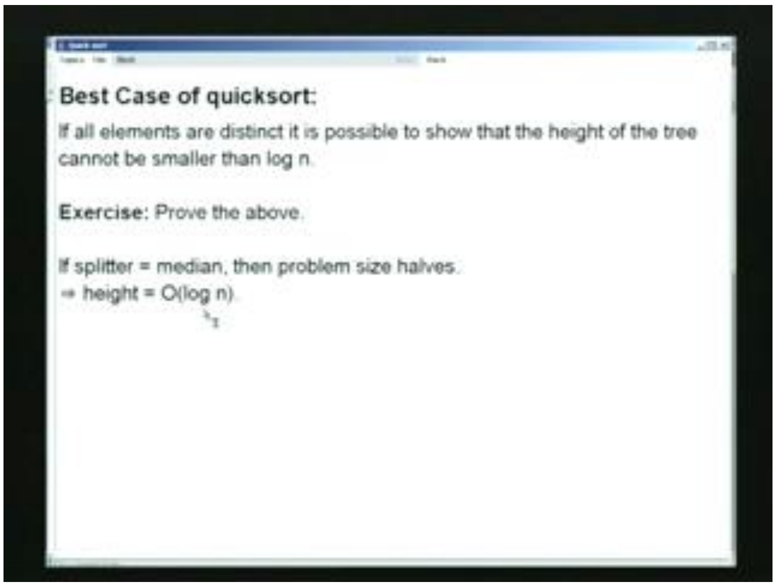
Similarly, here also it is going to be O of n. At every level, it is going to be at most n at most proportional to n.

So, now we have an upper bound on the work. Because, there are n levels at most and at each level the work is O of n. And therefore, the total work has to be O of n square. So, this is the upper bound on quick sort. Now, I will leave it as an exercise for you to construct an input instance for which, quick sort actually takes time n square. So, this is actually fairly easy. Let me give you a hint, think of a sorted list.

What if the input instance is already sorted? But, the key question is that this is the worst time. But, will it be the most time, will it take this long usually or is this some kind of an unusual case. So, if you come back to the recursion tree to this tree, then we know that at every level the work is going to be at most n. So, the real question that we want to ask is, will the tree be of a large height or will the tree have a small height because, if the tree has small height then our total work will be less.

So, in fact that is what we will see is, going to happen quite frequently. So, we did the analysis of the worst case. So, let us ask what the best case is going to be? So, clearly the best cases are the one in which tree is as small as possible.

(Refer Slide Time: 15:41)



Best Case of quicksort:

If all elements are distinct it is possible to show that the height of the tree cannot be smaller than log n.

Exercise: Prove the above.

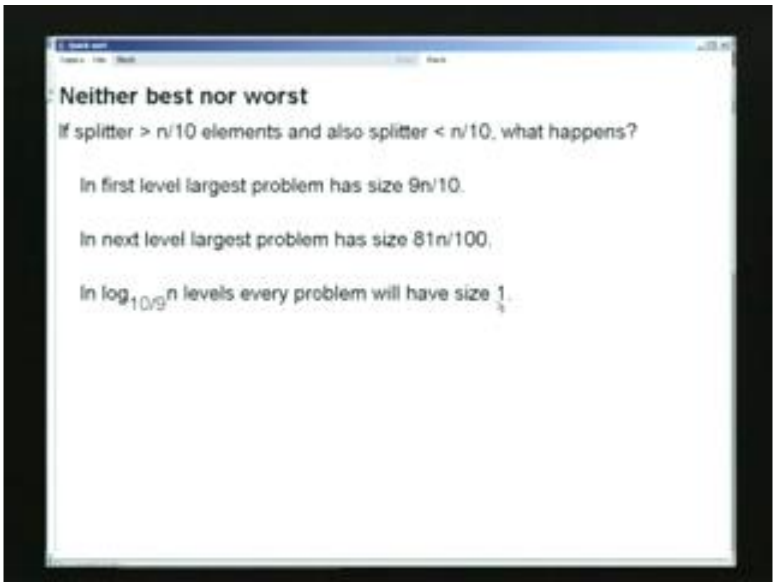
If splitter = median, then problem size halves.
⇒ height = O(log n).

If the elements that we are trying to sort are all distinct, then I will claim that the height cannot be smaller than log n. Why is that? Well, I am going to leave this as an exercise.

But, again let me give a hint. So, we said as we go down the tree height must decrease. But, we also said that the sum of the nodes over here, the size over here plus the size over here must be smaller than this. But, if everything is distinct it will only be one less than this. So, if it is one less than this, then you should be able to argue that it would not decrease too fast either.

So in fact, you should be able to argue that it essentially halves at each step. And therefore, the total height will be something like log n. So, what happens in the best case? So, in the best case it turns out. That the total time taken will be O of log n, O of n log n. And in fact, there is a very simple situation in which the best case will happen, which is this? If the splitter is equal to the median, then the problem size halves. And, then the height becomes O of log n.

(Refer Slide Time: 16:57)



Neither best nor worst

If splitter > n/10 elements and also splitter < n/10, what happens?

In first level largest problem has size 9n/10.

In next level largest problem has size 81n/100.

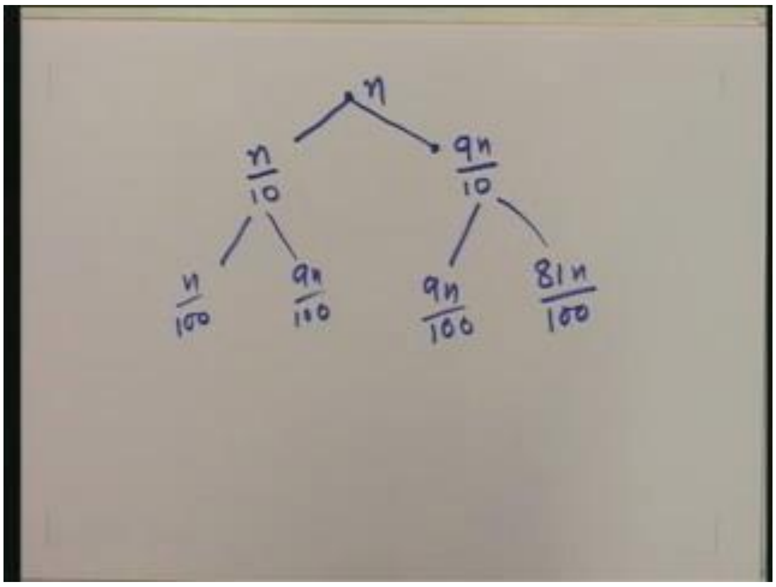
In $\log_{10/9} n$ levels every problem will have size 1.

So, if the height is O of log n that I have taken as n log n, and that is the best. So, we consider two cases one case in which the splitter goes, somewhere in the middle. Another case in which the splitter was extreme, the splitter was the smallest element. Well, that was supposed to be a homework exercise. But, suppose we take splitter, the splitter happens to be the smallest element. Then, the list would be split very unevenly.

So, let us consider a case which is somewhere in between. So, in this the splitter is say larger than n over 10 elements in the list and is also smaller than n over 10 elements. So, it could be somewhere in the middle. So, of course this is an artificial case. But, you can imagine that this will happen frequently enough because, after all if I pick an element from a list, it is likely to be somewhere in the middle.

So, let us say this happens. Let us say that, every time I pick a splitter it satisfies a property like this one. What happens then? Well, let us go back to the recursion tree. So, let us redraw this recursion tree.

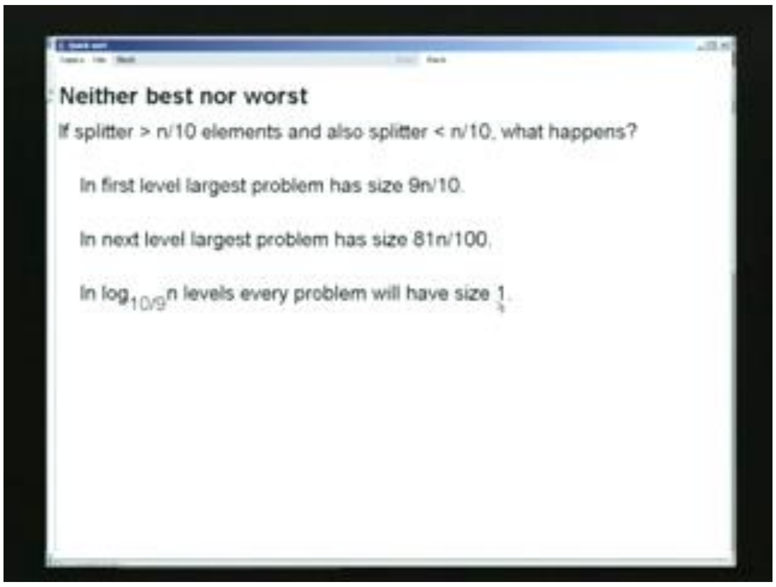
(Refer Slide Time: 18:18)



So, I start with an n node problem an n key problem. Now, I am going to pick a splitter such that, it is larger than n over 10 elements. So, if I consider, what is the most uneven distribution? What is the size? Well on one side I could get something like a list of n over 10 elements. On this side, I could get a list of say 9 n over 10 elements. So, this is good because, this list is going to shrink and its going to terminate quickly.

The height is going to be small over here. This on the other hand, might appear to be a problem, because here the height has not reduced. That the size has not reduced. If the size has not reduced, then it will keep on going in this manner. And may be the height of the tree may be large. But, what we argued was the work done in this algorithm is, the height of the tree is at most the height of the tree multiplied by n because, n is the work at each level.

(Refer Slide Time: 19:23)



Neither best nor worst

If splitter > n/10 elements and also splitter < n/10, what happens?

In first level largest problem has size 9n/10.

In next level largest problem has size 81n/100.

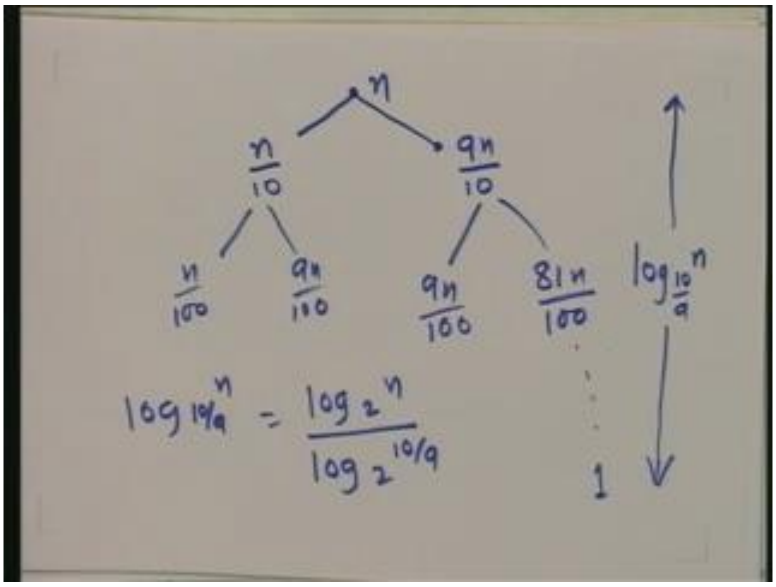
In $\log_{10/9} n$ levels every problem will have size 1.

So, let us see what happens? So, in the first level as we have pointed out, the largest problem size will be 9 n by 10. It could be smaller than that. So, it could be say half half. But, that is actually not so bad. That means, the third tree height will be actually small. So, this is trying to force the tree height to be large. And therefore, it is trying to force quick sort to take large, to take long time. So, we are sort of looking at we said that we are looking at neither the best nor the worst cases.

But, we are sort of erring on the side of the worst within this region. So, in the first level problem, largest problem level size is 9 n by 10. What happens next? Again, we assume that the problem will split in the ratio 1 is to 9. So, this will become say something like n by 100 and 9 n by 100. This will become something like 90 n by 100 and 81 n by 100. So, as you can see this rightmost branch will keep on having the largest size.

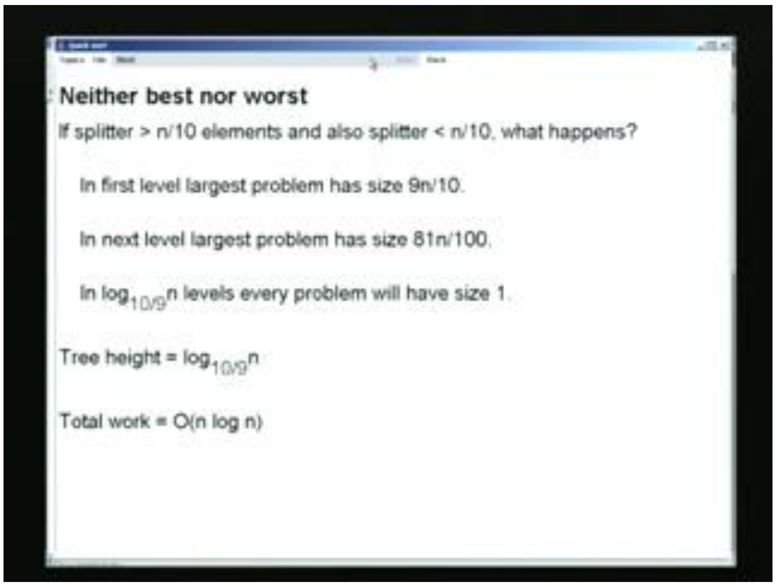
So, what will happen? At each in each step, the size of the largest problem drops down by a factor 9 by 10. And therefore, we can conclude that log of n to the base 10 by 9, the problem size will even on this right most branch will drop down to 1. And even to do that, I will take log of n to the base 10 by 9 levels. So, this is good news in the sense that, even when I am looking at a split which is lopsided.

(Refer Slide Time: 21:25)



The number of levels, the height of the tree is still going to be about log. Well, it is going to be log not to the base 2. But, to the base 10 by 9 and let me just remind you that, log n to the base 10 by 9 is simply equal to log of n to the base 2 divided by log of 10 by 9 to the base 2. So, this is still only a constant. And therefore, this is still O of log n. So, the height given in this case is O of log n, the height of the tree. The tree height is of log n. And therefore, the total work is n log n.

(Refer Slide Time: 22:08)



Neither best nor worst

If splitter > n/10 elements and also splitter < n/10, what happens?

In first level largest problem has size 9n/10.

In next level largest problem has size 81n/100.

In $\log_{10/9} n$ levels every problem will have size 1.

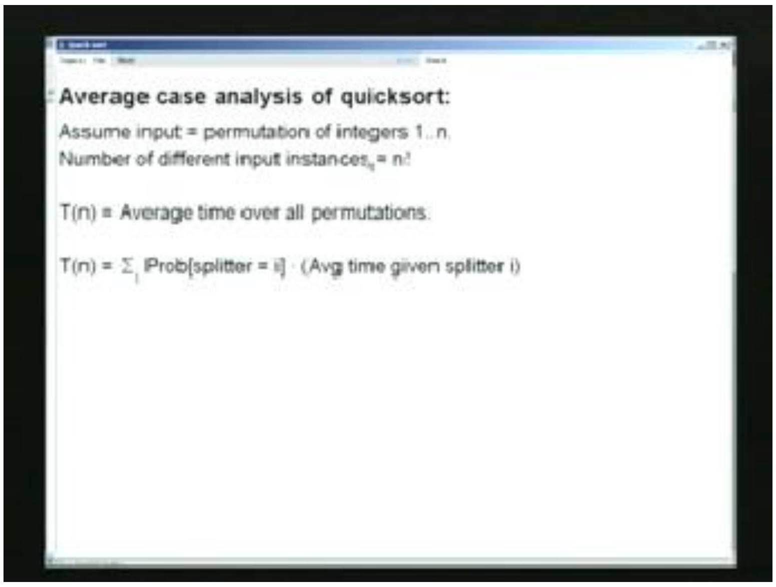
Tree height = $\log_{10/9} n$

Total work = O(n log n)

So, even in this middle case we have seen that the total work is about n log n. So, that is the sort of the first intuition as to why quick sort should work? Quick sort may be works while in practice. Because, unless the splitter comes from too large or too small, the two sub problems that we create will be reasonably balanced and not too lopsided. And if they are not too lopsided, then the height of the tree height of the recursion tree will not be too large.

Next we are going to actually do sort of a very systematic analysis, of the average time taken by the quick sort. We are going to do this in two ways. In one way, we are going to derive the recurrence. And we will not really solve the recurrence, but I will indicate to you how that recurrence could be solved. And it will turn out that, the solution of the recurrence is n log n. And, then I will indicate somewhat more elegant way using, which we can also derive n log n.

(Refer Slide Time: 23:22)



Average case analysis of quicksort:

Assume input = permutation of integers 1..n.

Number of different input instances$_n$ = n!

T(n) = Average time over all permutations.

T(n) = Σ_i Prob[splitter = i] · (Avg time given splitter i)

So, when I talk about average case, I have to define what are the possible inputs? So, in this case I am going to assume that, for this particular analysis I am going to assume first of all that all the inputs are distinct. All the inputs, the numbers the elements the keys which are given to us are all distinct. And if they are all distinct, I might as well assume that they are integers 1 through n for each of the n keys.

But of course, they will not be given to me as 1 through n, but they will be given to me as some permutation of 1 through n. So, now I will state exactly what my allowed inputs are. So, my allowed inputs are any possible permutation of the integers 1 through n. So, there are n factorial possible permutations. There are that many input instances for my algorithm. So, my question will be, what is the average time taken by my algorithm over all these input instances or over all these permutations?

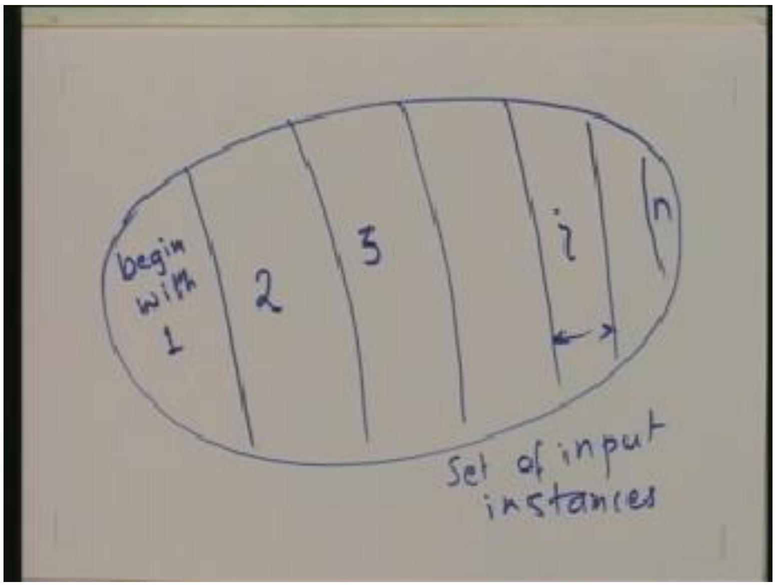
And of course, I would like you to express it as a function of n. So, now I am going to express I am going to look at our analysis. And I am going to figure out, how we can estimate this. So, although I have been talking about different, about taking averages I can also think of this in terms of probabilities. So, I can think of this as follows. So, I have been given a set of input instances. I have constructed a set of input instances.

And I am picking one of those instances at random. And I am doing this, giving equal probability to every input instance. So, there are n factorial instances possible. Each one has equal probability or in other words each one has probability whenever we assign. So I am picking one of those. And I could also be asking under this choice, what is the expected time for that for the instance that I pick? Which is of course, the same thing asking what is the time taken, what is the average of all the times?

So, now this average can be estimated by grouping the instances into separate groups. And, then calculating the average within each group and then multiplying by, essentially by the size of the group or by the probability of picking that group. So, here is how you are going to do it. So, in the first step of the algorithm, we pick a splitter. There are n keys and the keys are going to be numbers in the range 1 through n. So, there is going to be some probability that, the splitter is going to be one of these keys.

It is to be even any one of those keys. So in fact, let us assume that we always pick a splitter at the first element which is in fact, what the algorithm did. So, in that case the question is. So, we are splitting all our input instances into those permutations first in which the splitter in which I appears in the first place. And within that group, we are taking the average time. So, let me draw this picture out here.

(Refer Slide Time: 26:59)

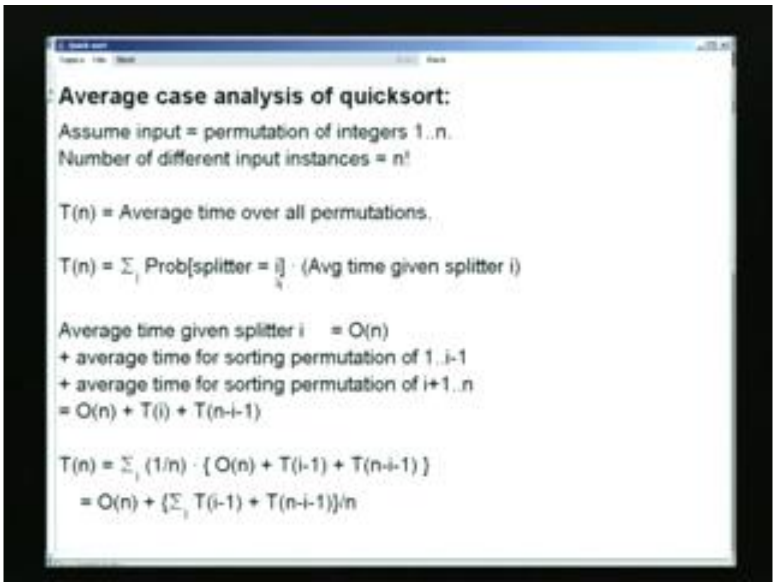


So, here is our set of input instances. So, I am breaking it into pieces. So, these are input instances which begin with 1. That is, they have 1 in the first place. These are input instances which begin with 2. These are input instances which begin with 3. And somewhere over here are input instances which begin with i. And of course, at the end there are instances which begin with n. So, I am going to pick a group.

And, then I am going to pick an element from it. Or I can ask, what is the average time taken for this group? And if all this groups are identical, then I can just take this average or I will have to wait with the size of this group. So, that is exactly what I have done over here. So, I have taken the average time for this group which is what is written over here? Average time given that splitter is equal to i.

But, given that splitter is equal to i is the same thing as saying, that the first element of the list is i. So, I am in this region of my input space. And since, I want the average over the entire space, I just want to I just multiply by the probability that the splitter is equal to i. Or the fraction which indicates how many instances are there in this group as compared to the entire group. So, this is what I get.

(Refer Slide Time: 28:42)



Average case analysis of quicksort:

Assume input = permutation of integers 1..n.

Number of different input instances = n!

T(n) = Average time over all permutations.

$T(n) = \Sigma_i$ Prob[splitter = i] · (Avg time given splitter i)

Average time given splitter i = O(n)

+ average time for sorting permutation of 1..i-1

+ average time for sorting permutation of i+1..n

= O(n) + T(i) + T(n-i-1)

$T(n) = \Sigma_i (1/n) \cdot \{ O(n) + T(i-1) + T(n-i-1) \}$

$= O(n) + \{\Sigma_i T(i-1) + T(n-i-1)\}/n$

Now, what is the average time given that the splitter is i? Well, if we go back to our algorithm here. So, I pick a splitter over here. Then, I am going to have this loop anyway. So, if I am solving a problem of size n, I will do n work in any case. And, then I will have my inputs split into two lists or three lists. But, only two of which will be interesting. So, average time given splitter i is going to be O of n for that loop to take, loop to do its work plus the average time for sorting the small set.

But, what is the small set? It is the permutation of the elements of integers 1 to i minus 1. And the average time for sorting permutation of elements i plus 1 through n. Because, that is what quick sort does. It splits into groups, it sorts the first group, takes the equal elements in which case in this case there is only one equal element which is i. Sorts the last group and then concatenates them together.

So, in addition to sorting the time will require is O of n. So, you might require O of n time also for concatenation. But, in any case we have written O without actually mentioning the constant. And therefore, this is fine. Or we might have a clever data structure in which case, we do not need this O of n time. But, in any case we need the O of n time for the loop. So, this is perfectly fine.

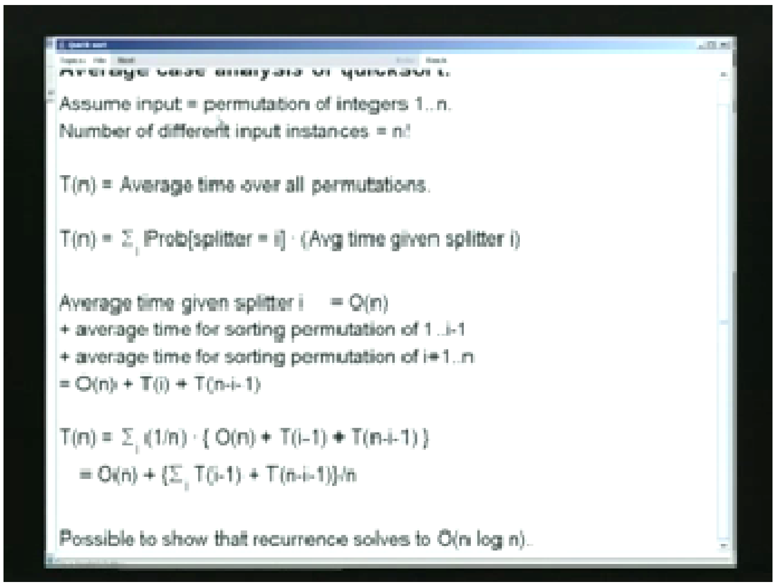
So, now you have the average time for sorting permutation of 1 through i minus 1 and then the average time for sorting permutation of i plus 1 through n. Here is the important part. So, the first time we picked the splitter to be i and then we constructed this group.

But, the key observation has to be, that the numbers the order in which these numbers will appear is not going to be particularly biased. So, we know that within the group that we selected, i is going to appear as the first element.

Since we are dealing with all possible permutations, the other elements would appear equally likely in the first space in this group or in the second space in this group or in the third space in this group. So, this group will have all possible permutations of 1 through i minus 1 as well. So, if it has all possible permutations of 1 through i minus 1, then the time average time for sorting it will be T of i or other T of i minus 1.

It does not really matter, T of i over here. The time over here is going to be i plus 1 through n or it is going to be T of n minus i. So, I think. So, what do we get from this? Well this expression has to be put in over here and as a result we get something like this.

(Refer Slide Time: 32:03)



Assume input = permutation of integers 1..n.
Number of different input instances = n!

T(n) = Average time over all permutations.

T(n) = Σ_i Prob[splitter = i] · (Avg time given splitter i)

Average time given splitter i = O(n)
+ average time for sorting permutation of 1..i-1
+ average time for sorting permutation of i+1..n
= O(n) + T(i) + T(n-i-1)

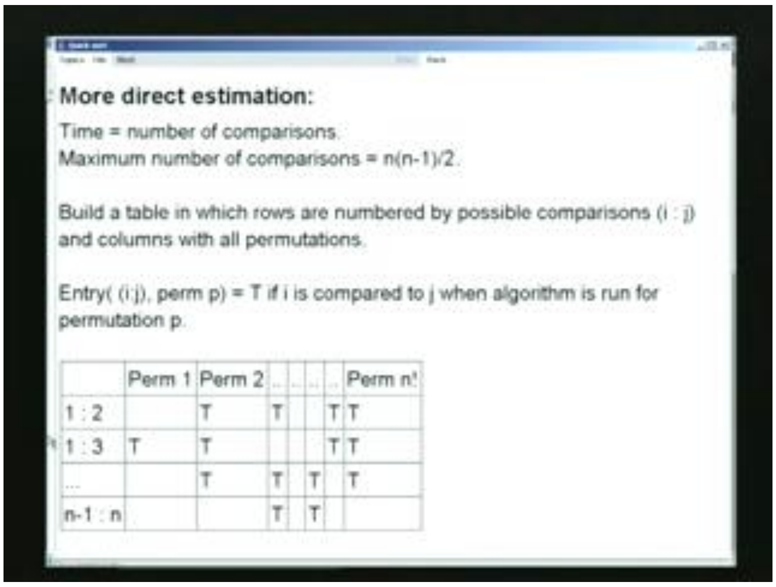
T(n) = Σ_i (1/n) · { O(n) + T(i-1) + T(n-i-1) }
= O(n) + {Σ_i T(i-1) + T(n-i-1)}/n

Possible to show that recurrence solves to O(n log n).

T of n is equal to sum over i of this probability that, the splitter is i. There are n choices for i and since we are considering all possible permutations. Everyone is equally likely to appear in the first place. And therefore, the probability that i appears in the first place is just 1 over n. So, this is 1 over n and this we just established is this. And that is what I have written over here.

I just remarked that this should have been i minus 1. And that is what I have put in over here. Now, this recurrence can actually be solved. It is a little bit tedious algebraically, but you can certainly solve it by recursion induction. Since I am telling you that, the solution is n log n. So, that will establish that the average case of quick sort is n log is O of log n.

(Refer Slide Time: 33:02)



More direct estimation:

Time = number of comparisons.
Maximum number of comparisons = n(n-1)/2.

Build a table in which rows are numbered by possible comparisons (i : j) and columns with all permutations.

Entry((i:j), perm p) = T if i is compared to j when algorithm is run for permutation p.

	Perm 1	Perm 2	..	..	..	..	Perm n!
1 : 2		T	T			T	T
1 : 3	T	T				T	T
...		T	T		T		T
n-1 : n			T		T		

Now, we are going to do we are going to consider an alternate method for solving this. So, this is going to be much more direct. We are not going to write recurrences. We are just going to do some interesting counting. So, here we will focus on the comparisons performed by the algorithm. So, after all the important operation in all of this, is comparison. So, if you go back to the loop let us just take a look at that. We did other work as well.

Say we added elements into lists. But, corresponding to every such operation there is a comparison operation going on as well. So, certainly if we bound the number of comparisons, then that will give us a good indication of the time taken by the entire algorithm. So, that is exactly what we are going to do. So, we are going to estimate what is the number of comparisons performed by the algorithm on the average.

And we will show, then that is going to be something like O of n log n. Well, let us first determine what is the maximum number of comparisons possible? So, the maximum number clearly is n into n minus 1 upon 2. This is, if every key is compared with every other key. And of course, if the input is the worst case input, then something like this actually happens. But, this will not this will, but if the input is some permutation, then every key will not get compared with other key.

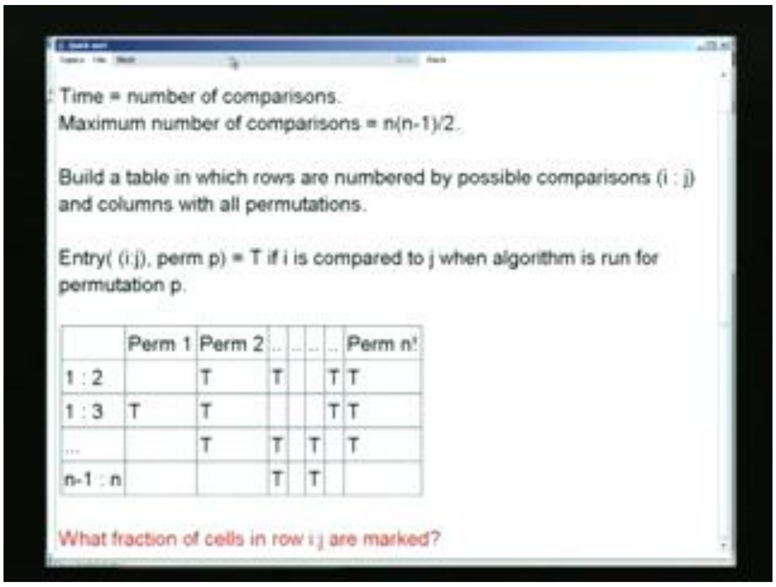
So, just to see clearly what is going on, I am just going to describe a table which shows what happens for different input instances. So, a table this table will have rows. And there will be a row corresponding to every possible comparison. So, our keys are integers in the range 1 through n. And for every i and j, we will have a row. So, I compare j that will be the label of that row. And in that row, we will have information about whether i and j are compared in every possible input instance.

And in fact, the columns will be the input instances. So, the entries are going to be indexed by two indices, one is i colon j. Well, this itself is a complicated index and the other is this permutation P. So, here for example, is a table. Of course, I have just made up the entries, just to tell you what this table might look like. So, the rows are labeled i colon j. So, starting with 1 column 2, 1 compare 2, 1 compare 3 and so on to n minus 1 compare n.

So, during the execution whether it is or not 1 is compared to 2, when permutation 1 is input is going to be written out here. So, you have left a blank over here. And that just says that node, that node will not be compare. It is just an example. On the other hand, 1 and 3 will be compared, when permutation 1 is the input. Similarly, if permutation 2 is the input then 1 and 2 will get compared, 1 and 3 will get compared and may be some other things will also get compared.

Similarly, there will be other permutations for which this will be the pattern of comparison, this will be and so on. So, there are n factorial possible input permutation. So, we have n factorial possible columns. And for each possible comparison, we have a row. And their intersection says that, whether that comparison actually happens in the corresponding execution.

(Refer Slide Time: 37:00)



Time = number of comparisons.
Maximum number of comparisons = n(n-1)/2.

Build a table in which rows are numbered by possible comparisons (i : j) and columns with all permutations.

Entry((i:j), perm p) = T if i is compared to j when algorithm is run for permutation p.

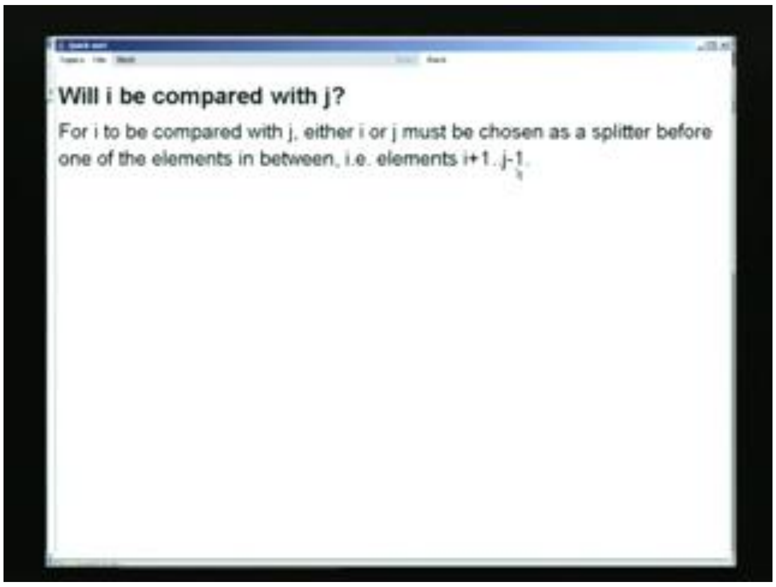
	Perm 1	Perm 2	..	...	...	..	Perm n!
1 : 2		T	T			T	T
1 : 3	T	T				T	T
...		T	T		T		T
n-1 : n			T		T		

What fraction of cells in row i:j are marked?

The key question is, Are there many T cells in this or are most of the cells blank? What we really want to know is, what fraction of the cells in the column are marked? Or what is the average number of cells which are marked in a given column? We are not going to answer this question directly. We will begin by asking, what is the fraction of cells which are marked in any row?

And interestingly, that will tell us something about what happens in columns as well. Say, if I go to a particular row of this table or the row which has labeled i colon j, the question that I am asking is, Is i going to be compared with j in the first permutation or in the first input instance or in the second input instance in the third input instance and so on.

(Refer Slide Time: 38:12)

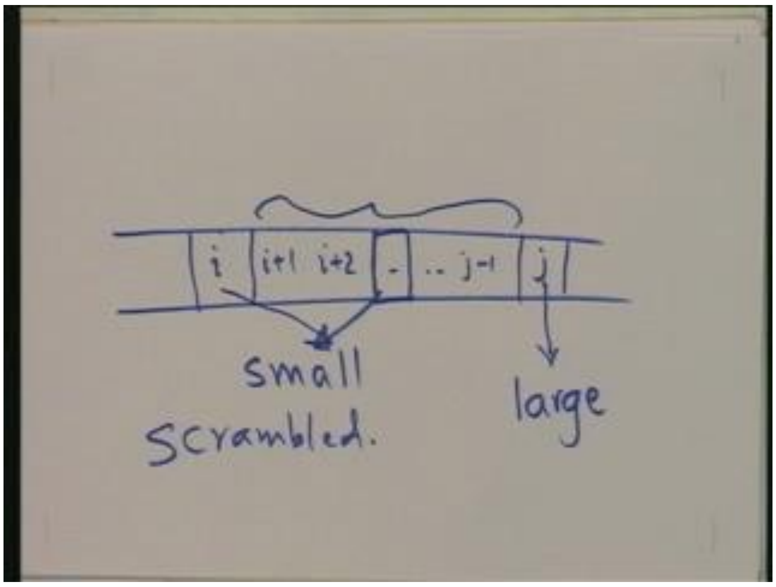


Will i be compared with j?

For i to be compared with j, either i or j must be chosen as a splitter before one of the elements in between, i.e. elements i+1..j-1.

So, here is the key observation for i to be compared with j either i or j must be chosen as a splitter before, one of the elements between that is elements i plus j or j minus 1 gets splitter. So, let me explain this a little bit.

(Refer Slide Time: 38:33)



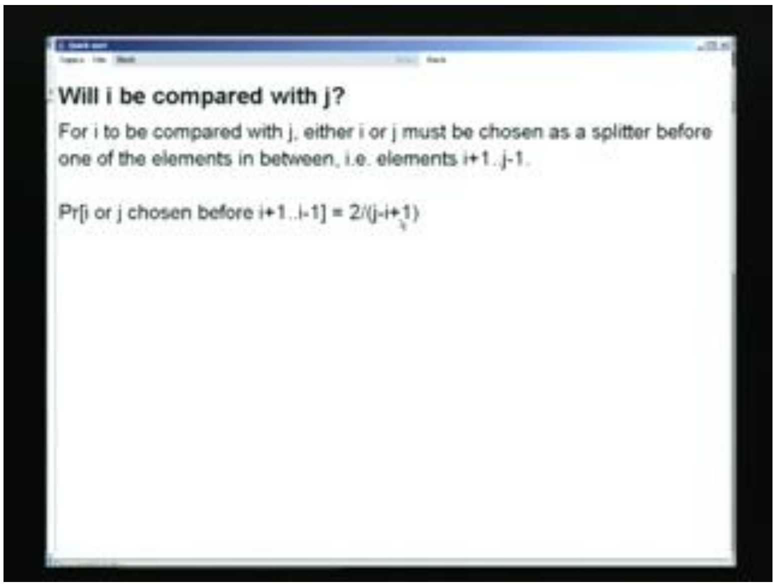
So, here is i here is j and there are some elements in between. Well, I know i plus 1 i plus 2 all the way till j minus 1. So, these are the elements that I am considering. Of course, they will not appear in my input instance in this order. They will be in my input instance, they will be scrambled up. But, I am just thinking of them as sitting in a line. Now, suppose some element over here gets picked up as a splitter, what happens?

If this element is picked as a splitter, then this element is compared with everything else. If everything else is compared with it, then this I will get input in the small list. So, i will go into the small list. j on the other hand will get input in the large list. But, remember that once an element goes into this list and another element goes into another list, there is no question of comparing them subsequently.

So, if any of the elements in between over here get picked as splitters, before any of these two elements get picked. Then, we know for sure that these elements will go into separate lists. And therefore, they will not be compared. On the other hand, before these elements have been picked suppose i gets chosen, what happens then? Well, then i is going to be compared with everything larger than it, or certainly everything which has not been found which is in the current list.

But, if nothing in this has been selected as a splitter, then this had been better been in the current list. And therefore, j will get compared with i and vice versa. If j gets picked first, then i will get compared because j will be compared with everything over here. So, which means that, these two elements must get split as splitters before these, inner elements are picked. So, what is the probability of that happening?

(Refer Slide Time: 40:59)



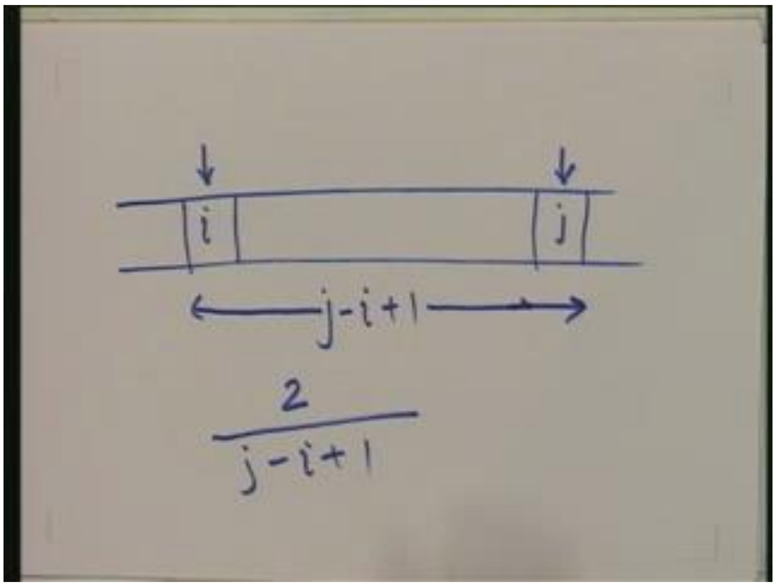
Will i be compared with j?

For i to be compared with j, either i or j must be chosen as a splitter before one of the elements in between, i.e. elements i+1..j-1.

Pr[i or j chosen before i+1..i-1] = 2/(j-i+1)

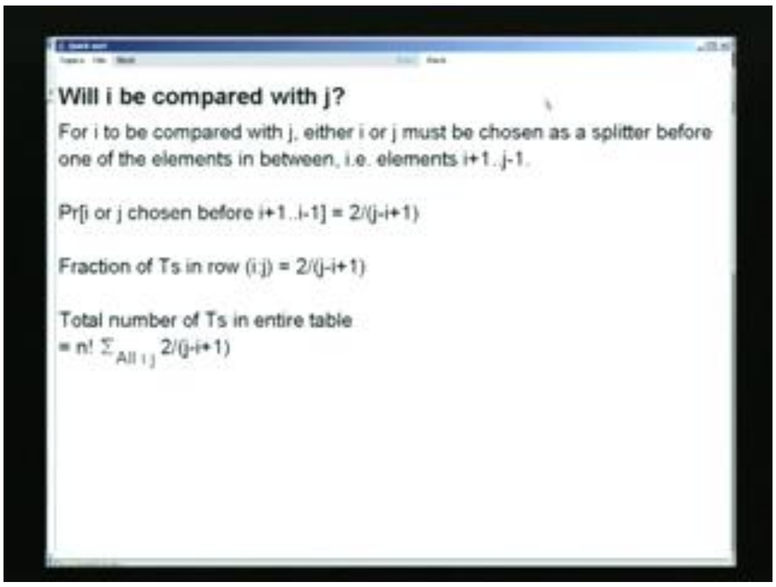
So, I claim that probability of i or j being chosen before i plus 1 or before elements i plus 1 through j minus 1 is in fact, 2 minus 2 upon j minus i plus 1.

(Refer Slide Time: 41:22)



So, here is i here is j. So, how many elements are these in total? These are j minus i plus 1 elements. And out of these, the comparison happens only if this is picked or this is picked. So, there are two cases which are good out of j minus i plus 1 cases. And therefore, that is the probability. So, now actually things are very, very simple.

(Refer Slide Time: 41:59)



Will i be compared with j?

For i to be compared with j, either i or j must be chosen as a splitter before one of the elements in between, i.e. elements i+1..j-1.

Pr[i or j chosen before i+1..i-1] = 2/(j-i+1)

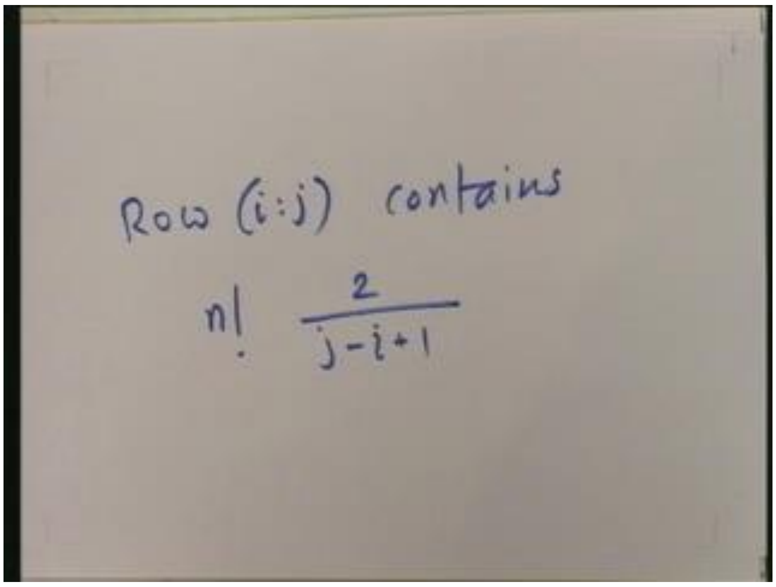
Fraction of Ts in row (i:j) = 2/(j-i+1)

Total number of Ts in entire table
= n! $\Sigma_{\text{All } i,j}$ 2/(j-i+1)

So, the fact that i or j is probability that i or j is chosen, before i plus 1 through j minus 1 is this just tells us something very simple. It tells us that the fraction of T s in this row is just this. Because, that is what the probability is. We are going to pick a row at random. And we know that, 2 upon j minus i plus 1 fraction of the time we get a T or the comparison happens.

So that means, in other words the number of columns the fraction of the number of columns in which T s appear, is just going to be this much. So, what does that tell us? So, it tells us that the total number of T s in the entire table is going to be sum over all the rows of this multiplied by n factorial. Let me explain that a bit slowly. So, from this what can I conclude?

(Refer Slide Time: 43:03)

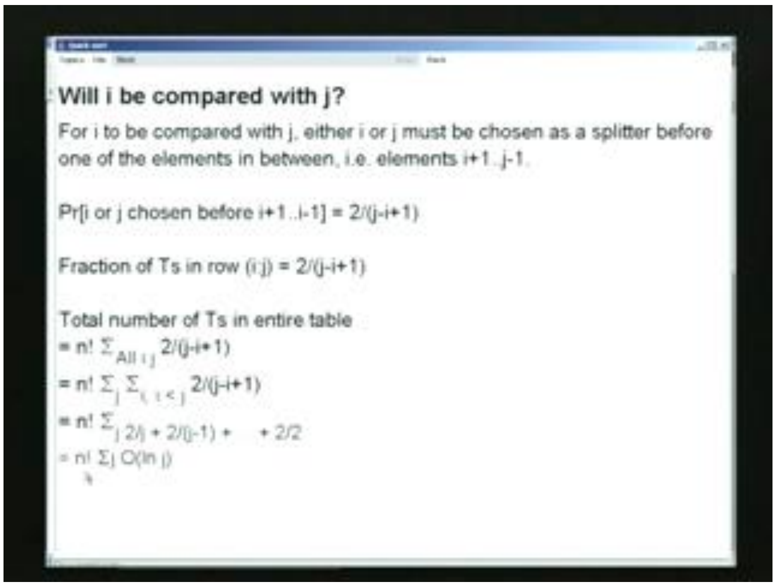


Row (i:j) contains

$$n! \; \frac{2}{j-i+1}$$

I can conclude that, in row i colon j contains n factorial times 2 upon j minus i plus 1 T s, what T s represent? Where comparisons happen, whether comparisons happen or not. But, if I want over the entire table I just have to sum over all possible rows. So, this is what I have written out here.

(Refer Slide Time: 43:33)



Will i be compared with j?

For i to be compared with j, either i or j must be chosen as a splitter before one of the elements in between, i.e. elements i+1..j-1.

Pr[i or j chosen before i+1..i-1] = $2/(j-i+1)$

Fraction of Ts in row (i:j) = $2/(j-i+1)$

Total number of Ts in entire table

$= n! \sum_{\text{All } i j} 2/(j-i+1)$

$= n! \sum_{j} \sum_{i,\ i<j} 2/(j-i+1)$

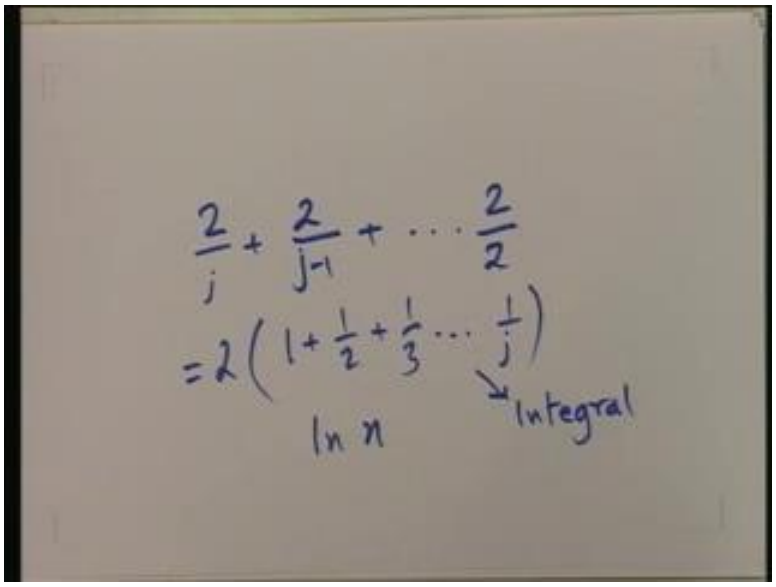
$= n! \sum_{j} 2/j + 2/(j-1) + \quad + 2/2$

$= n! \sum j\ O(\ln j)$

Except that the n factorial has taken outside because, it does not depend on what row I am looking at. Well, this expression can be written out slightly differently. So, all possible labels i j, I can now classify as all possible levels in which j is a second element. And, then the first element has to be smaller. And therefore, it is sum over i is less than j of this expression.

But, what is that, so summation over i of i less than j of this expression, well. What is the first term? So, i begin from 1 and so, first term is simply 2 upon j. And next term is 2 upon j minus 1 and so up on until 2. But, what is this? So, this is let me write it down again.

(Refer Slide Time: 44:37)



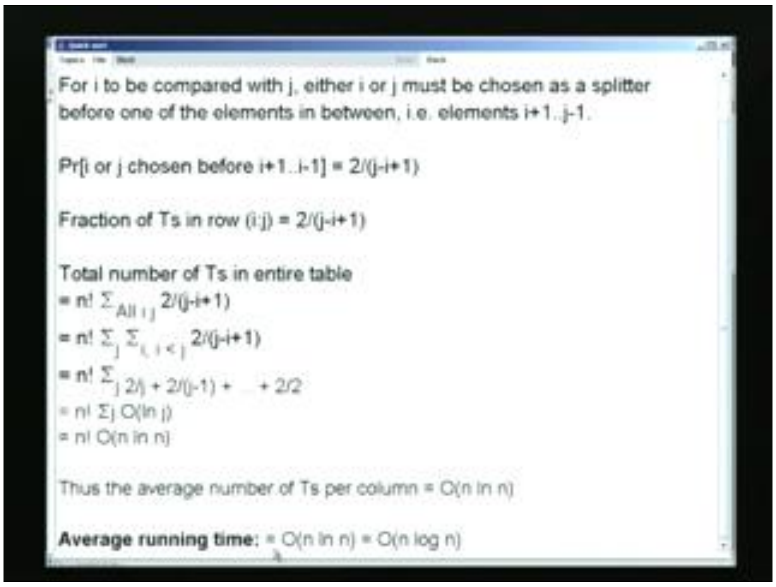
$$\frac{2}{j} + \frac{2}{j-1} + \cdots \frac{2}{2}$$

$$= 2\left(1 + \frac{1}{2} + \frac{1}{3} \cdots \frac{1}{j}\right)$$

$$\ln n \qquad \searrow \text{Integral}$$

It is 2 upon j plus 2 upon j minus 1 plus all the way till 2 upon 2 or written differently, it is 2 times 1 plus half plus one third all the way upon till 1 upon j. And this we know simply l n n by treating this sum to an integral, converting it to an integral. So, this is a good estimate or in fact, this is an upper bound.

(Refer Slide Time: 45:13)



For i to be compared with j, either i or j must be chosen as a splitter before one of the elements in between, i.e. elements i+1..j-1.

Pr[i or j chosen before i+1..i-1] = 2/(j-i+1)

Fraction of Ts in row (i:j) = 2/(j-i+1)

Total number of Ts in entire table
= n! $\Sigma_{\text{All } i,j}$ 2/(j-i+1)
= n! $\Sigma_j \Sigma_{i,\, i<j}$ 2/(j-i+1)
= n! Σ_j 2/j + 2/(j-1) + ... + 2/2
= n! Σj O(ln j)
= n! O(n ln n)

Thus the average number of Ts per column = O(n ln n)

Average running time: = O(n ln n) = O(n log n)

So, finally we have this whole thing as n factorial times sum over j of O of l n j. But, if you are going to take the sum over j, what do we get? Well, we get n l n and n. So, we get n l and n over here, what is n l and n? So, we have total number of T s in the entire table list n factorial times n l n n. So, what then is the number of T s per column or what is the average number of T s per column?

Well, how many columns are there are there? There are n factorial columns. And therefore, we divide this total number by n factorial and then we get O of n l n n. So, average running time is O of n l n n and why is that? Because T s represent the number of comparisons, and we said that the average that the time is in fact proportional to the number of comparisons. So, the average running time is going to be O of n l n n.

But, O of n l n n is simply O of n log n as well. So, here the base was the natural base was e or this was the natural logarithm. Here the base is 2, but that does not matter log of n and l n of n are within a constant factor of each other. So, let me conclude. So, I would just like to say that, a similar idea works for selection as well. So, suppose we want to select the rth smallest element, then something like this will also be fine.

Thank you.