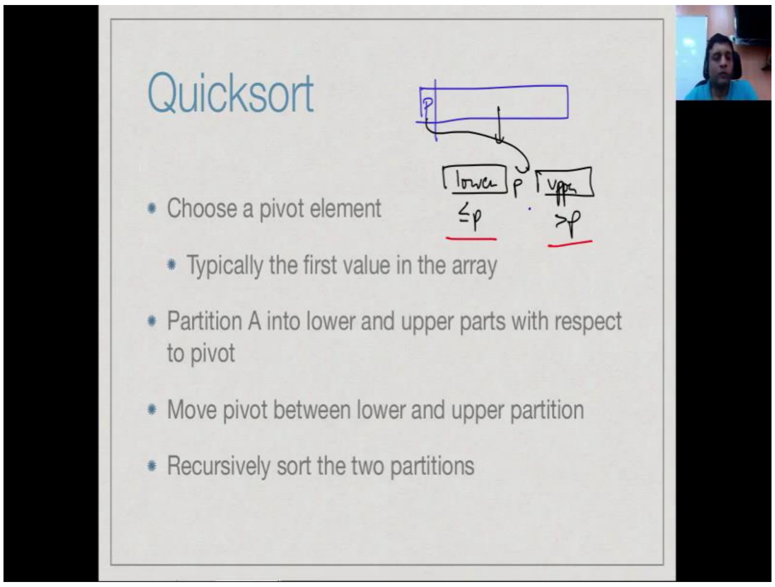
Design and Analysis of Algorithms, Chennai Mathematical Institute
Prof. Madhavan Mukund
Department of Computer Science and Engineering,

Module – 08
Lecture - 16
Quicksort: Analysis

We have seen Quicksort which is a divide and conquer algorithm which overcomes the requirement for an extra array as in merge sort. So, let us do an analysis of Quicksort.

(Refer Slide Time: 00:11)

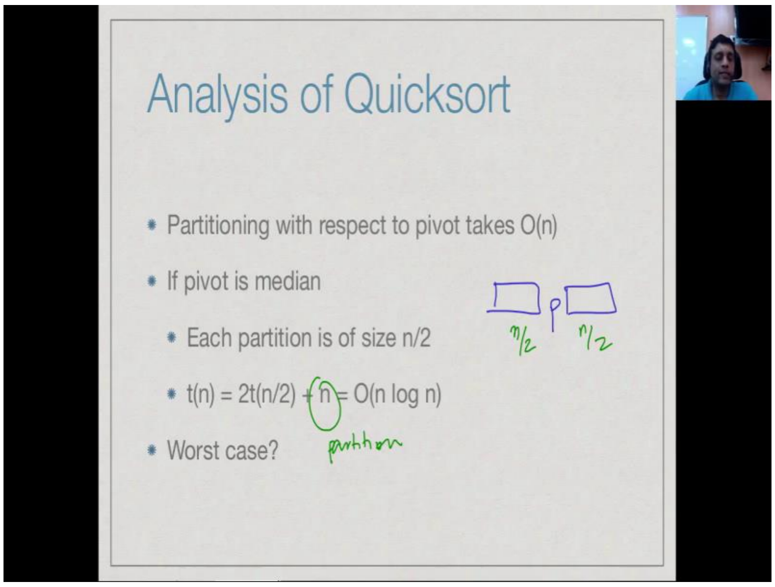


Quicksort

- Choose a pivot element
    - Typically the first value in the array
- Partition A into lower and upper parts with respect to pivot
- Move pivot between lower and upper partition
- Recursively sort the two partitions

So, remember how Quicksort works, you pick up a pivot element say typically this first element of an array. And then what you do is, you partition this into two parts such that you have a lower part which is less than or equal to p and may be have an upper part, this is bigger than p. And you move this pivot in between and then you sort this lower part and upper part separately recursively and then you do not need to do any combining step, because these two things are within the correct position with respect to each other.

(Refer Slide Time: 00:45)



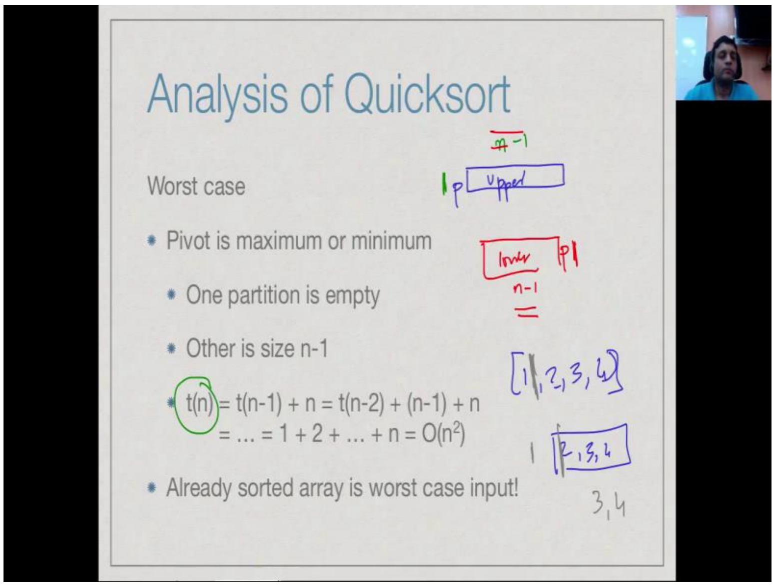
Analysis of Quicksort

- Partitioning with respect to pivot takes O(n)
- If pivot is median
  - Each partition is of size n/2
  - $t(n) = 2t(n/2) + n = O(n \log n)$
- Worst case?

So, the first thing we observed is that this partitioning actually is quite efficient. We can do it in one scan of the entire array. So, we can partition with respect to any pivot in order n time. So, the question is how bigger the recursive problems? So, if the pivot is a median then you would expect that by definition in median that these are of size n by 2. Because, the median is that element which splits the array into two parts, those half of the elements are bigger than the median, half are smaller than the median.

And if you do have this fortunate situation that the pivot is the median, then we end up with the merge sort recurrence which says that t of n takes time 2 times t n by 2 for the two parts and this is the partitioning steps. So, it is not the merge step after the recurrence, but the partitioning set before the recurrence. So, we have as we saw in merge sort, this recurrence takes order n log n if we expand it out, but the pivot is in some sense the best case.

(Refer Slide Time: 01:44)



Analysis of Quicksort

Worst case

- Pivot is maximum or minimum
  - One partition is empty
  - Other is size n-1
  - $t(n) = t(n-1) + n = t(n-2) + (n-1) + n = \ldots = 1 + 2 + \ldots + n = O(n^2)$
- Already sorted array is worst case input!

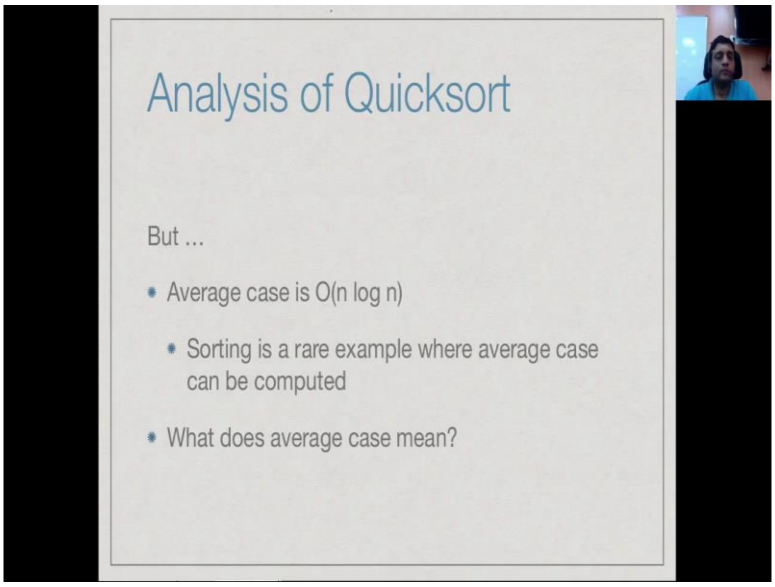
What do we thing is a worst case? When the worst case is when the pivot is an extreme value, either the smallest value or the biggest value. So, if it is a smallest value then what will happen is that everything will be bigger than the pivot. So, you will have an upper element set which has n minus 1 values, because the pivot is a smallest value and we will have nothing on this side. Symmetrically, if the pivot is a largest value in your array, then you would have everything in the lower element set.

So, this is again besides n minus 1 and the pivot would be something which is on one extreme end and there is nothing in the other side. So, now what we see is that in order to sort this array of size n, I have to then sort a smaller segment which is only n minus 1 it is more smaller than n minus 1. So, our t n takes t n minus 1 plus n, n is a time taken to partition and t n minus 1 again in the worst case will have again a pivot element which is the extreme value.

So, for example, supposing we start with the already sorted array like 1, 2, 3, 4 then what happen is that, we pick 1 as the pivot and then this, this then results in what we want to solve 2, 3, 4 and then I will pick 2 as a pivot and this results in our sort 1 it sort 3, 4 and so, on. If you have an already sorted array in some sense, the pivot is always an extreme values. So, the next step takes splits the array very bad. And of course, we expand out this t n is t n minus 1 plus n, we get the summation that we got for first selection sort and insertion sort. So, this becomes order n square.

So, the worst case of Quicksort is actually order n square, which is the same as the worst case for selection sort and insertion sort. So, why do we bother with this much more complicated algorithm Quicksort, when we already know several intuitive algorithm which have order n square.

(Refer Slide Time: 03:36)



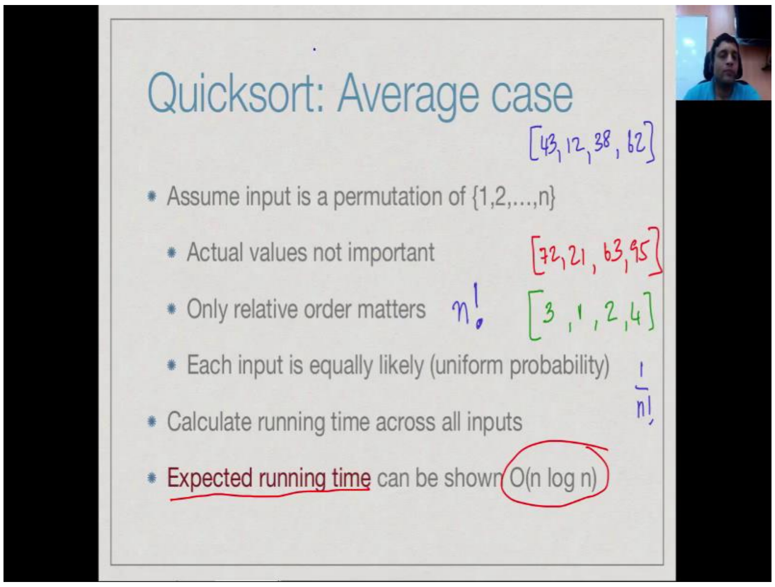
Analysis of Quicksort

But ...

- Average case is O(n log n)
  - Sorting is a rare example where average case can be computed
- What does average case mean?

So, it turns out that Quicksort we can show actually does not behave in this worst case way in a very frequent manner. So, we can actually compute in the case of Quicksort what is called the average case complexity and show that this n log n. So, we will not actually show that it is n log n, but we will try to at least explain what it means to compute the average case analysis of Quicksort. As we said in the beginning, average case is very difficult to compute. So, let us see what it involves to do this.

(Refer Slide Time: 04:09)



Quicksort: Average case

[43, 12, 38, 62]

- Assume input is a permutation of {1,2,…,n}
  - Actual values not important [72, 21, 63, 95]
  - Only relative order matters $n!$ [3, 1, 2, 4]
  - Each input is equally likely (uniform probability) $\frac{1}{n!}$
- Calculate running time across all inputs
- Expected running time can be shown O(n log n)

So, the first reason why the average case is difficult to compute is, because we need to have a way of describing all possible inputs. Now, even for a sorting algorithm all possible inputs is an infinite space, supposing I just take arrays of a fixed line, supposing I take arrays of length 4. So, I could have an array which look like 43, 12, 38 and then 62. So, this is an array with 4 elements, I could have another array of 4 elements which is say 72, 21, 63 and 95.

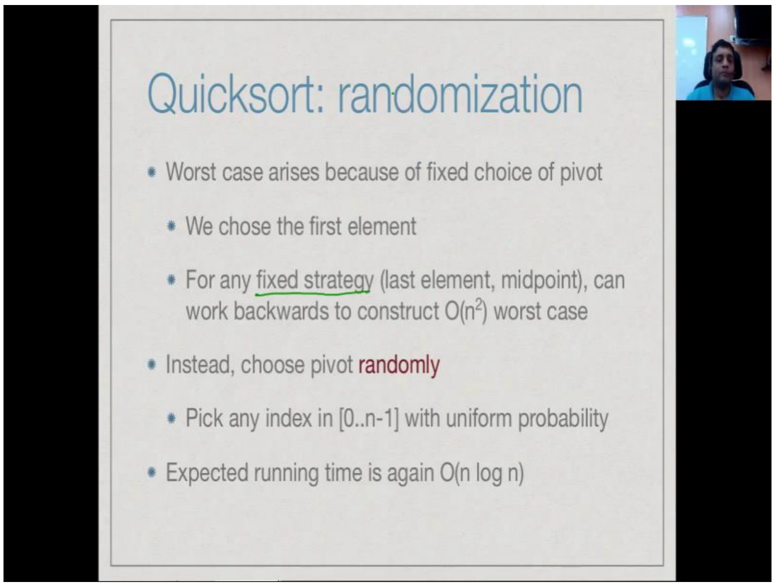
But, in this way we can continue and put any elements we want and there are infinitely many error of size 4. But, there is a commonalty between these, which says that the first element is bigger than the second element. In fact, the second element is the smallest element and so, on. So, if you look at this we can say that there are 4 elements and we think of them in order, then the smallest element is here, the second smallest element is here, the third smallest element is here and the fourth smallest element is here. So, I can actually think of this as the array 3, 1, 2, 4, because 4 elements and the 4 elements are ordered in this way.

So, the actual values are not important only the relative order matters. So, we can actually think of inputs of size and to be these kind of re orderings of 1 to n or permutations of 1 to n. Now, among these permutations we do not have any preference, any one of them would come as our input. So, we all know that there are n factorial such permutations and we say that each of them is equally likely. So, each of them has probability 1 by n factorial are occurring.

Now, we look at all these n factorial inputs of size n and see how had been our algorithm behaves. So, we will not do the actual calculation, but if you see the average, you see the actual time it is take for all the n factorial inputs, added and divide by n factorial which is what is in probability known as calculating the excepted running time. Then, you can show that this is actually order n log n.

So, we are not shown it, we are just explain what is the mathematics required in order to show this. But, in Quicksort you can prove that the expected running time across all possible random inputs equally likely inputs, this actually order n log n. So, though Quicksort has an O n squared worst case and the average it behaves like merge sort and without some of the pit falls of merge sort, it particular it does not requires the extra space in order to create a merge array.

(Refer Slide Time: 06:41)



Quicksort: randomization

- Worst case arises because of fixed choice of pivot
  - We chose the first element
  - For any fixed strategy (last element, midpoint), can work backwards to construct $O(n^2)$ worst case
- Instead, choose pivot randomly
  - Pick any index in [0..n-1] with uniform probability
- Expected running time is again O(n log n)

Now, you can actually exploit this average case behavior in a very simple manner. So, why does this worst case occur? The worst case occurs, because the pivot that we choose could be a bad pivot, as we saw if you put the first element as your pivot, then a sorted array becomes a worst case, because every time the pivot is the extreme element. On the other hand, you could take the last element and you would have the same problem, if you pick the midpoint again you can make the middle point of the array that you start with the extreme element and you can then work backwards and construct always the worst case which takes order n square.

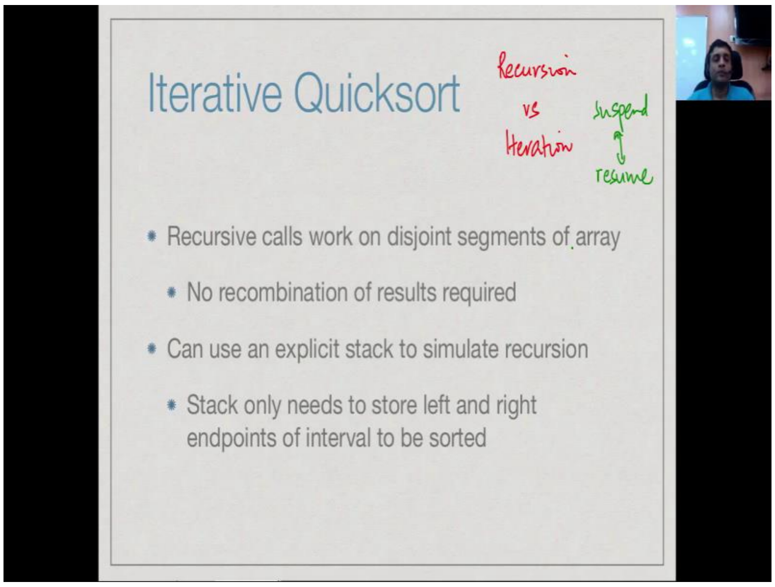
So, what we are saying is that for any fixed strategy, if I tell you in advance that I am always going to compute the position of the pivot in a fixed way, then by working backwards you can always ensure that the position in the current problem, you have a worst case that is an extreme input and reconstruct something which will take O n square for that strategy.

So, the solution is to not fix the strategy, each time I want to apply Quicksort to a recursive sub problem, I have some position 0 to n minus 1 which I need to pick as a pivot. But, rather than telling you that is going to be 0 or n minus 1 or the mid-way between 0 and n minus 1, I will say that I will choose any one of these values with equal probability.

So, think of it as, I am choosing a random number between 0 and n minus 1 equally likely or if you want to think graphically it is like passing at or throwing a die. So, a die has say six faces normally. So, if you roll a fare die you get any number between 1 and 6 will be equal likely. So, now we have an en sided die. So, we have a complex kind of object, we throw it and whichever number comes up, we pickup that as a pivot.

So, now the behavior of this algorithm is not fixed, it depends on how this die rolls. So, this is a different type of algorithm called a randomized algorithm. So, you can now implement Quicksort in a randomized speed with a very simple randomization step, namely just pick the pivot at random at each called Quicksort. And it is turns out that again you can do a similar calculation, saying that across all the possible random choices I make for the pivot, the expected running time is order n log n. So, this is a very simple, this is a kind of a dual result to the fact of the average cases n log n, you can exploit that by creating a very simple randomized strategy in order to achieve this n log n thing with good probability.

(Refer Slide Time: 09:05)



Iterative Quicksort

- Recursive calls work on disjoint segments of array
  - No recombination of results required
- Can use an explicit stack to simulate recursion
  - Stack only needs to store left and right endpoints of interval to be sorted

The other aspect that we mentioned about merge sort, which is a bit limiting, is that it is inherently recursive. Now, our solution to Quicksort avoids this duplication of space, but it is recursive. Now, it turns out in Quicksort you can actually manually make a recursive algorithm iterative. So, the point is that the recursive calls works on disjoint segments. So, what you need to remember in the recursive call is not the entire segment, but just what segment you need to work on, you do not need to combine the results.

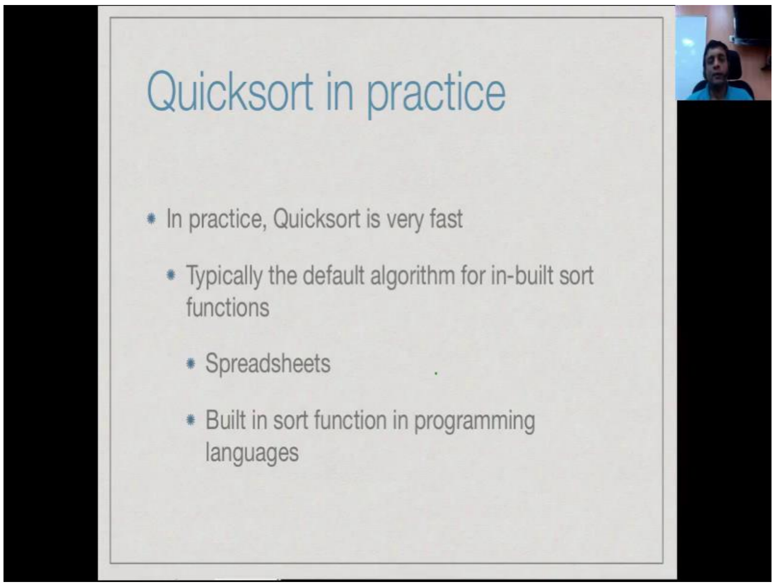
So, we will not discusses in great detail, but it turns out that you can use a stack, you can actually maintain your own stack and every time you make a recursive call, you just store in the stack, the left and right end point of what segment needs to be sorted. And in this way you can actually take the recursive algorithm that we wrote before and convert it into an iterative algorithm. Now, why you would like to do this in general list, because you have a trade off in recursion versus iteration depending on your programming language.

Because, when you make a recursive call, when you make a function call in general in a programming language, what happens is the current function that you computing has to be suspended. So, you need to suspend and receive. So, when you make a recursive call you have to put aside whatever you have and then you have to take a new set of local variables. So, in the memory of the program you have to load some new data, then you have to execute the function, you want to terminates, you have to throw that out and restore the context, you have to resume.

So, this takes some time and it take some resources and so, usually the cost of making a function call, even though we might count for it in our complexity as a basic operation is much more than doing some really arithmetic operation like addition or something. So, in particular recursion every time you make a recursive call, you have basically going and replacing something on the stack is some new frame and then putting it back and this takes time.

So, it is in general sometimes for efficiency purposes, good to convert recursion to iteration. On the other hand, this process can make the algorithm more obscure and many programming languages actually are optimizing compilers can try to do this automatically. So, may be this distinction between recursion and iteration does not always help so, much. But, it is useful to know that certain algorithms can be done both ways and certain algorithm is difficult to do one here.

(Refer Slide Time: 11:26)



Quicksort in practice

- In practice, Quicksort is very fast
  - Typically the default algorithm for in-built sort functions
    - Spreadsheets
    - Built in sort function in programming languages

So, our final remark before we leave quicksort for now. So, in practice Quicksort is very fast, as we said the worst case happens very rarely. For this reason, typically Quicksort is a default algorithm that you see that people use when you have a built in sort function. So, if you have a spread sheet and allows you to sort a column, then usually this algorithm running in your background to sort that column is Quicksort or if you have built in sort function.

For example, C, C plus plus, java all allow you to just call sort, even python just allows you to just call sort. In almost any programming language, this sort function that is available to the programmer by just a simple call is usually an implementation of Quicksort. Of course, this implementation may use various optimizations such as randomization and other things to make it faster, but at the underlined algorithm the heart of it is usually Quicksort.