

Design and Analysis of Algorithms
Prof. Madhavan Mukund
Chennai Mathematical Institute

Week - 02
Module – 07
Lecture - 15
Quicksort

So, we are now ready to look at another algorithm called Quick sort. So, quick sort was invented by a computer scientist called Tony Hoare in the early 1960's; that is about 50 years ago. And Tony Hoare is a very well-known computer scientist and he has impact one Turing about which is one of the highest achievements awarded for academic computer scientist.

(Refer Slide Time: 00:28)

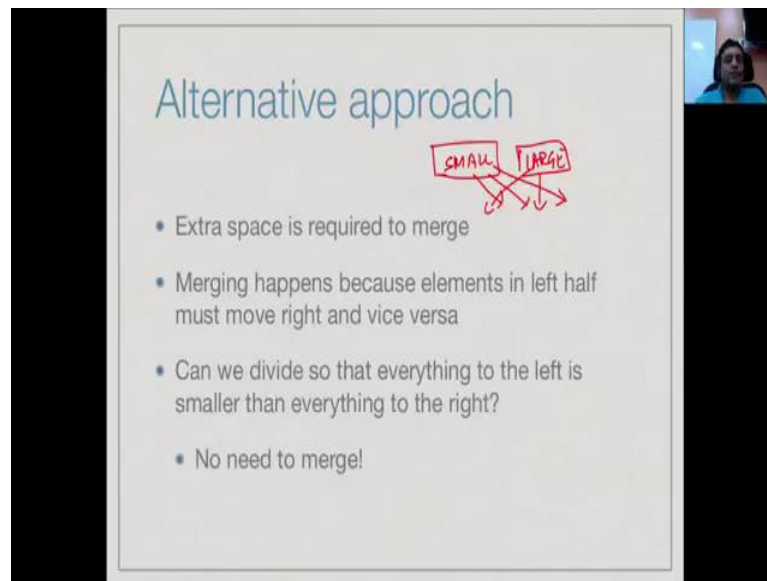


Merge Sort: Shortcomings

- Merging A and B creates a new array C
 - No obvious way to efficiently merge in place
- Extra storage can be costly
- Inherently recursive
 - Recursive call and return are expensive

So, what is the purpose of quick sort? Well, the purpose of quick sort is to overcome some of the shortcomings that we saw in the merge sort. So, one of the things that we saw in the merge sort is that because of the merge operation, we need extra storage and so, this makes merge sort a little expensive.

(Refer Slide Time: 00:42)



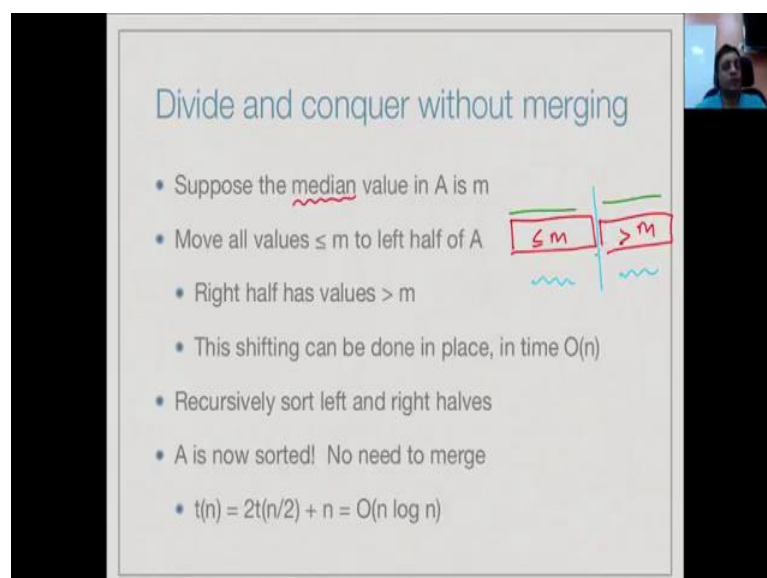
Alternative approach

- Extra space is required to merge
- Merging happens because elements in left half must move right and vice versa
- Can we divide so that everything to the left is smaller than everything to the right?
- No need to merge!

Handwritten diagram: Two boxes labeled 'SMALL' and 'LARGE' with arrows pointing from 'SMALL' to 'LARGE' and from 'LARGE' to 'SMALL', indicating a swap or movement between halves.

So, we also observed that the reason that we need this extra storage is because we have in the merge operation that we might be pulling out elements from both sides, when we have populating the merged array. So, basically some elements in the right might be smaller than some elements in the left and this is what results in merging. So, can we divide everything. So, that this does not happen, everything on the left is smaller and everything of the right is larger, is it possible to do a divide and conquer in this fashion?

(Refer Slide Time: 01:16)



Divide and conquer without merging

- Suppose the median value in A is m
- Move all values $\leq m$ to left half of A
- Right half has values $> m$
- This shifting can be done in place, in time $O(n)$
- Recursively sort left and right halves
- A is now sorted! No need to merge
- $t(n) = 2t(n/2) + n = O(n \log n)$

Handwritten diagram: A vertical line with a box on the left labeled ' $\leq m$ ' and a box on the right labeled ' $> m$ ', representing the partitioning of an array around a median value m.

Well, this is the case, then what we need to do is, we need to put the middle value in the center. So, supposing we can find the median. So, remember what the median is, the median is the value such that exactly half the values in the array are bigger and half a

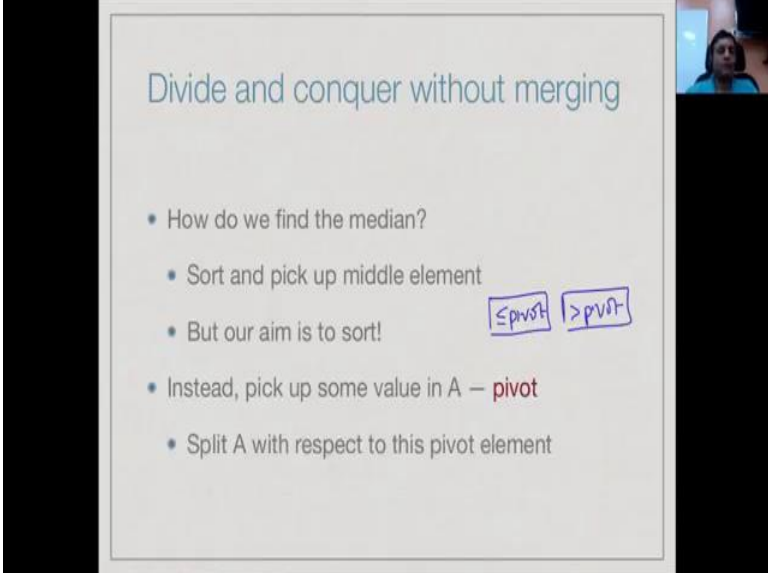
smaller. So, now we move everything which is smaller than m to the left half. So, we have a set of values here which are less than or equal to m and then we have on the right side something which is strictly greater than...

So, of course, we have to do this shifting, but the claim is that we can do this shifting in linear time and we will see a way to do this. So, assuming we can do this, pick a value m which is the median and shift everything smaller than m to the left, then this is roughly going to be the half point, because m is a value which splits the array into two parts, those just smaller or half and those bigger or half.

Now, I do this recursive thing, I sort this and I sort this and now remember this no need to merge, because everything on the left is already smaller than everything on the right. So, I can just go ahead and assume the array is sorted. So, if I do this, then by the analysis as for merge sort, I have a recurrence which has T of n is $2 \times T_{n/2} + n$ and we now this is order $n \log n$.

So, this will give us the same complexity as merge sort, but it will avoid some of the overheads involved with creating extra space. Because, when I do the recursive call here, I can easily sort this part in place and this part in place, because I do not need to refer to the other part at all when I do this solution.

(Refer Slide Time: 02:42)



The slide is titled "Divide and conquer without merging". It contains a list of bullet points:

- How do we find the median?
- Sort and pick up middle element
- But our aim is to sort!
- Instead, pick up some value in A — **pivot**
- Split A with respect to this pivot element

Handwritten notes in blue ink are present next to the third and fourth bullet points:

- Next to "But our aim is to sort!": $\leq \text{pivot}$ and $> \text{pivot}$ (each in a box)
- Next to "Instead, pick up some value in A — **pivot**": **pivot** (in red)

So, of course, there must be a catch and the catch is how do we find the median? Right at the beginning of our discussion, we said that one of the reasons that we want to sort is to do statistical things like find the median. So, if we have sorted the array, then the median

value is the middle value, but of course, our goal now is to sort the array. So, we cannot assume that we have the median, because we have already seen that sorting is an easy way to find the median. So, it is a kind of the chicken and egg problem, we cannot use the median to sort.

So, what quick sort Tony Hoare algorithms says, do not necessarily pick the medium, just picks some value in A and do what we said. So, we pick up this pivot and then you break it up into two parts, those which are smaller than the pivot and those that are bigger than the pivot. So, the pivot is just some value and the array, it need not be the median and we will see that if that is not the median, then this results in some problem in terms of the worst case complexity, but let us just ignore it.

So, we just pick some value in the array and we take all those value as smaller than that, move it to the left, all those which have bigger than that move it to the right. And then, we sort them recursively and then we guaranteed that nothing on the left needs to be combined with anything on the right after this. So, the resulting array is sorted.

(Refer Slide Time: 03:55)

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

Diagram illustrating the partitioning process:

Lower partition: $\leq \text{pivot}$

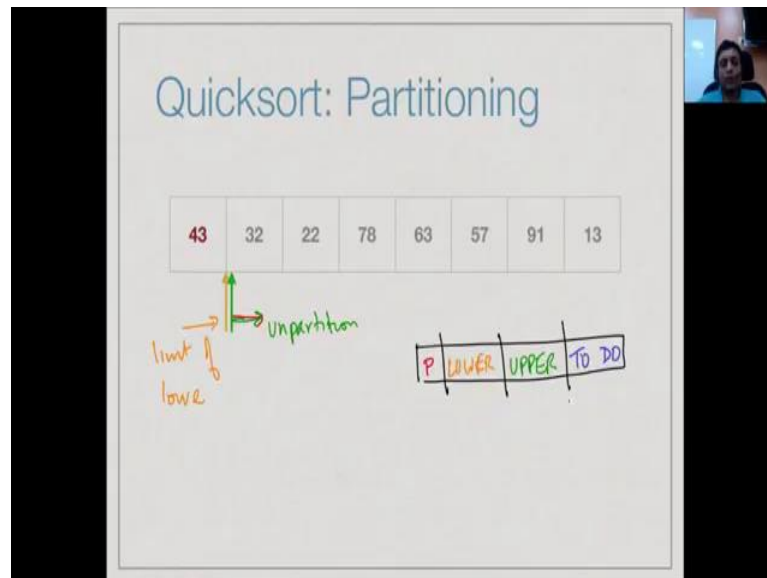
Upper partition: $> \text{pivot}$

Pivot element is shown moving between the two partitions.

So, this is quick sort, choose a pivot element. So, for example, you may just pick up the very first value in the array as implemented. Partition this array into the lower and upper part. So, the lower part is those which are less than the pivot, the upper part is that which is greater than the pivot. So, the crucial step is this partitioning, we will see how to do this partitioning, that is what we see now. Then, we move this pivot here. So, that it is in the correct place and now we recursively sort this part and this part and we are done,

because nothing needs to move.

(Refer Slide Time: 04:35)



So, here is a kind of high level description of the algorithm through an example. So, supposing this is my array, then I pick the first element at the pivot namely 43. So, with respect to 43, I now partition this array. So, that everything smaller than 43. So, what are the elements smaller than 43 here, we have 32, 22 and 13. So, these elements should come to the left and the elements which are bigger namely 78, 63, 57, 91 should go to the right.

So, I do this partitioning and how do I do this partitioning, well I kind of... So, I brought everything to the left and then after this, I recursively sort the left. So, this is no longer assumed to be sorted it, just I editing smaller than 43, this is not sorted, there are everything bigger than. Then, recursively if I assuming I can sort it, then I have sorted the entire array, because now nothing in the yellow side needs to be combined with green side, the red pivot value separates these two.

So, the first thing that we need to understand is how to do this partition. So, there are two ways to partition. So, we will look at one in detail and show some code for it and then we will look at another through an example and you will have to write the code by yourself, if you are interested. So, here is one way to partition. So, I have start with the pivot element at the left and now I have this entire range of values to the right which are unsorted.

So, I will put two indices which I have indicate in this picture with two color pointers, a

yellow pointer and a green pointer, this significance will become a little clearer once we move a couple of steps in the algorithm. So, what we do is that everything to the right of the green pointer. So, the green pointer indicates the end of the part which has already been partitioned. So, anything to the right of the green pointer is unpartitioned and the yellow pointer on the other hand is going to indicate. So, this is going to indicate the limit of the lower part.

So, I need. So, basically in general I am going to have this picture. So, I am going to have the pivot here, then I am going to have the lower part here which have already found. Then, I am going to have the upper part here, these are the elements have already scanned and partitioned and then I have the part that is to do. So, in the beginning everything is to do and there is no lower part and there is no upper part.

So, what I am saying that we will keep these pointers like this. So, this thing will point to the end of this and this thing will point to the end of this. So, this is what we want to achieve. So, we start as I said with this picture. So, what we do is, if we see something which is lower, then I extend the lower part and I am move to the next element, again we see something this is lower. So, we extend the lower part. So, if this point would be saying is that the lower part has two values 32 and 22 and the upper part is empty and everything will 78 onwards is ((Refer Time: 07:51)).

Now, I look at 78. So, 78 is bigger than 43. So, the lower part stays here and now I have a non empty upper part namely 78. Now, I look at 63, once again 63 belongs to the upper part. So, again I am move this follow, 57 again belongs to the upper part. So, I move to the follow, 91 again move to the follow. So, the first interesting thing happens when I come to 13. So, now, , when I come to 13, I find that it must going to the lower part, but the lower part is far away.

So, how do I achieve this. So, what I will do is, I know that this element to the right of the lower pointer. So, this is bigger than P and this element is smaller than P. So, one way to achieve what I need is to exchange these two values. So, I exchange 13 and 78. So, I take 13, I label it as lower, then I exchange and move both points. So, this is a forward partitioning algorithm which keeps reducing the length of the unpartitioned part, if I see something which is upper, I just move the green pointer. If I see something that is lower, I exchange that lower element with the first part of the upper thing and then I extend both partitions.

Then, finally, at this point I still do not have the final thing, but I want this pivot element to be in between these two. So, now the point is that I know that this is the last. So, what is to the right of the yellow pointer is the first upper limit and what is to the left of the yellow pointer is the last lower thing. So, I can exchange the 43 and 13 and then I get the final array partition does I want to that pivot in the middle, the lower part on the left and upper part on the right.

(Refer Slide Time: 09:40)

Quicksort: Implementation

```

Quicksort(A,l,r) // Sort A[l..r-1]

if (r - l <= 1) return; // Base case

// Partition with respect to pivot, a[l]
yellow = l+1;
for (green = l+1; green < r; green++)
    if (A[green] <= A[l]) // pivot
        swap(A,yellow,green);
        yellow++;

swap(A,l,yellow-1); // Move pivot into place

Quicksort(A,l,yellow); // Recursive calls
Quicksort(A,yellow+1,r);
    
```

The slide includes two diagrams. The first diagram shows an array with a pivot 'P' at index 'l', a yellow pointer at 'l+1', and a green pointer at 'r-1'. The second diagram shows the array after partitioning, with the pivot 'P' in its final position and the array divided into 'lower' and 'upper' sub-arrays.

So, this is how we do quick sort in general. So, in general now remember that after we do this partitioning, we are going to have to quick sort this part and quick sort this part. So, the recursive calls will be sorting different segments. So, it is useful to say for each call that I am sorting from some left limit to some right limit. So, in general quick sort will take the array and it will take two pointers, it will say sort from l to r minus 1.

Now, if this length is small, in other words, I have only one value, if r minus l is less than or equal to 1, if either are then you want value or if I had no values sort, then I do nothing. So, this is the base case. So, this is the recursive value algorithm, if the sorting, array to be sorted as only one element we do nothing. Otherwise, using the terminology of the previous example, we use the yellow to indicate the position of the yellow pointer and we use green to indicate the position of the green pointer. So, these two variables indicate the position of these two arrows.

So, remember that we start to the right to the pivot. So, initially we have at the position l, we have that pivot and r minus 1 the last terms. So, this is our pivot P. So, our initial

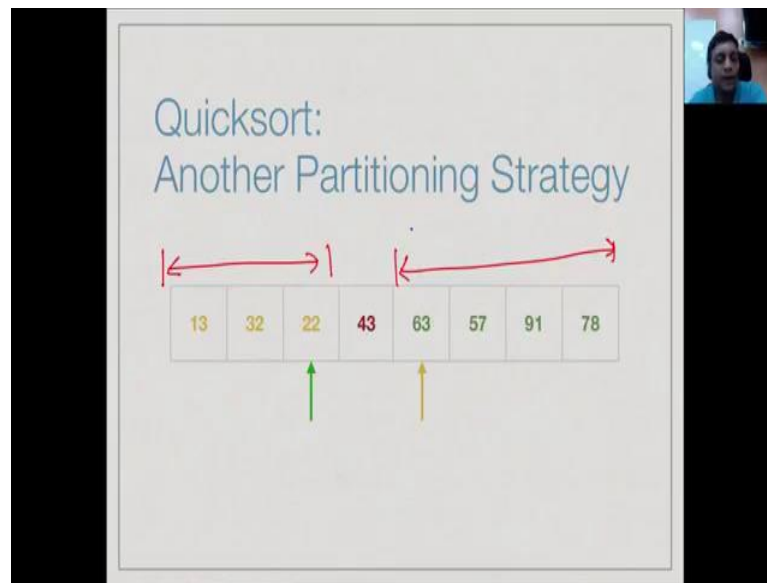
thing is to say that put both these pointers here. So, initially yellow is $l + 1$. So, that is this position and we start now moving the green. So, green starts $l + 1$ and goes until... So, if we see that the green value is smaller than the pivot. So, this is the pivot. So, A of l is the pivot.

So, the green value, the value I am looking at under the green pointers smaller, then I do this exchange. So, if I am wear somewhere and I have this green value, then what happens this is smaller, then I exchange these two values. So, that is what this is same, swap A the element at position yellow and the element at position green and A should be swap and then, I will also increment the yellow point. The green point has be incremented anyway at a P..

So, finally, after this loop I have done this partitioning to the extent where I have the pivot element, I have the lower part and the upper part. And now what I want to do is, I want to move the pivot to the center, at this point I have the yellow pointer here and the green pointer here. So, I need to take the last element here and exchange to the pivot and that is what this is going to same.

Exchange the value at position l with the value at position yellow minus 1, now I having done this, now I want to recursively sort. So, I want to sort from the beginning l up to an to the left of this yellow pointer. So, I sort from A from l to yellow, when I want to sort everything on the right. So, I sort with yellow plus 1 and sort into right. So, these are the input recursive calls. But, now the important thing is that at this point everything between l and yellow is small than the pivot, everything beyond yellow plus 1 up to r is bigger than the pivot. So, after these two sorting sub recursive calls to quick sort nothing more needs to be done, we are done.

(Refer Slide Time: 12:46)



So, as I said this partitioning strategy can also be implemented in a different way and in fact, this is the original partitioning strategy proposed by Tony Hoare. So, in this original strategy, the idea was to not start from one end and sweep until you claim all the element, but it started opposite ends. So, you start building up in some sense, then you start building up the lower side from here and you start building up the upper side from here.

And gradually the lower side grows, until it can expand a more the upper side goes and then everything is in place and then, you do the final swap as before. So, here what you do is, you start with again I will use the same color thing. So, yellow refers to lower green refers to upper. So, what I will do is, I will take the yellow pointer and keeps scanning until I find the value which is not yellow. So, 32 is smaller than 43. So, remember that we have trying to grow the yellow partition is the lower partition.

So, trying to include in the lower partition everything smaller than 43 and trying to include in the upper partition everything with the bigger than 43. So, I keep moving the yellow thing until I find the first error in some set. So, 32 is . So, I skip over it, 22 is also smaller than 43. So, I skip over it and now I reach the value 72. So, this point my partition ends here and 72 cannot be included, because it is bigger than 43.

Now, I start on right hand side and I look for the position, where I can include things in the upper thing, but the very first thing I see 13 should not be there. So, this pointer up upper limit is here. So, now, what I do is, I exchange these two values, if I exchange these two values, then this will become 13, this will become 78 and then, I will be able to

shift these two boundaries by 1. So, this is the basic step in this partitioning strategy.

So, the next step what I do is, I exchange the 13 and 78 and now I say that I have the lower thing up to here and I have an upper thing up to here. So, this is the invariant now, we have the lower thing on the left part and upper thing for the right part and in between we have the unsorted elements, but we have these two indicators, the left most unsorted the left most impartation element, the right most impartation element.

So, now, I again start doing the same thing, I move the yellow right, until I can along extend lower, here I cannot extend it anywhere, because 63 should already not be there. So, I cannot move this partition. On the other hand, the upper partition can move, because 91 is bigger. So, I will move it left, 57 is still bigger. So, I will move it left again, 63 is still bigger. So, I will move left again.

And now I find that the right partition indicator has move to the left of the left partition. So, when this exchange happens, then it terminate this partition. So, when the right boundary crosses the left boundary, when we are finish partitioning of the element, because there is nothing in between the two elements to be partitioning any more. So, once we have terminated, now we have the same problem as before which is that we want to move this element to this center, but now remember that at this point, when this thing is terminates, the right part there is pointing to the end point of the lower limit.

So, I can just exchange these two. So, I can exchange the 13 and 43. So, I take this there to move this here and then, I get my answer. So, if I move this and then, I simultaneously move the green pointer, now I have the pointed to the last of the lower elements or the pointed to the first of the upper elements and now I can apply quick sort recursively to this part to this part.

So, we will not write Pseudo code or describe this, the algorithm and more detail, but you can definitely try and work out similar way of keeping these indices moving as we did for the earlier partitioning and see if you can get it right, this is also discussed in many of the books. So, both these partition algorithm that we have in text books and you can choose which ever have find easier.

In both cases remember that they basic variant condition, there are these two markers and these two markers indicate the part which is already when partition, the limits of the lower and the upper part. And then, there is an unpartisan part and then, the unpartisan part becomes empty you have done.