



IIT Madras

ONLINE DEGREE

Programming, Data Structures and Algorithms using Python

Professor Madhavan Mukund

Quicksort

So, the last sorting algorithm that we will look at for now is called quicksort.

(Refer Slide Time: 00:14)

The image shows two screenshots of a video lecture. The top screenshot shows a slide titled "Shortcomings of merge sort" with the following bullet points:

- Merge needs to create a new list to hold the merged elements
 - No obvious way to efficiently merge two lists in place
 - Extra storage can be costly
- Inherently recursive
 - Recursive calls and returns are expensive
- Merging happens because elements in the left half need to move to the right half and vice versa
 - Consider an input of the form `[0, 2, 4, 6, 1, 3, 5, 9]`

The bottom screenshot shows a slide titled "Shortcomings of merge sort" with the following bullet points:

- Merge needs to create a new list to hold the merged elements
 - No obvious way to efficiently merge two lists in place
 - Extra storage can be costly
- Inherently recursive
 - Recursive calls and returns are expensive
- Merging happens because elements in the left half need to move to the right half and vice versa
 - Consider an input of the form `[0, 2, 4, 6, 1, 3, 5, 9]`
- Can we divide the list so that everything on the left is smaller than everything on the right?
 - No need to merge!

So, quicksort is interesting for us, because it overcomes some of the shortcomings that we have observed with merge sort. So, we saw two things which we did not like about merge sort. It was good that it took order $n \log n$ in the worst case, but it was bad because the merge function

actually forces us to use extra space to show the merge list. So, this creates an extra copy of all the space that we are using to store the original list.

And it also had this inherent recursion. We were forced to write a recursive function. And the reason that recursion is bad is because there is a certain amount of the programming language infrastructure for recursion requires you to suspend the current computation, make a call and then restore. So, there is a certain amount of, what you might call, paperwork with recursive calls. And so recursive calls tend to be more expensive in that sense. The function calls are expensive. And so you want to avoid them if you can for, if you are interested in performance, although as we said, in some cases, the most natural functions may be recursive. So, then you use recursion as a tool to write correct algorithms. But efficiency wise, it is useful to see if you can avoid recursion wherever possible.

So, why does merging require us? I mean, what is the cost of, cause of merging? So, if you think about an example like this, so these left and right halves are already sorted. But now, I cannot leave them as they are, because I have to move things here. I have to move the 1 and the 3 here. And in principle, I have to move the 4 and the 5 there, so 5 and 6. So, that in some sense, unless I do these shifting things across the two halves I will not be able to assemble the full sorted list. I cannot just keep the sorted list on the left and stick the sorted list on the right together and make a full sorted list.

So, the premise of quicksort is that what if we could do this division better. So, if we can do the division so that everything on the left is smaller than everything on the right, then I can just stick them together. I sort the left half. I sort the right half, but then the only work I need to do is to put them together kind of concatenate them. I do not need to actually do any merging. So, this is the idea behind quicksort.

(Refer Slide Time: 02:41)

Divide and conquer without merging



- Suppose the median of L is m
- Move all values $\leq m$ to left half of L
 - Right half has values $> m$
- Recursively sort left and right halves
 - L is now sorted, no merge!
- Recurrence: $T(n) = 2T(n/2) + n$
 - Rearrange in a single pass, time $O(n)$



Divide and conquer without merging



- Suppose the median of L is m
- Move all values $\leq m$ to left half of L
 - Right half has values $> m$
- Recursively sort left and right halves
 - L is now sorted, no merge!
- Recurrence: $T(n) = 2T(n/2) + n$
 - Rearrange in a single pass, time $O(n)$
- So $T(n)$ is $O(n \log n)$
- How do we find the median?
 - Sort and pick up the middle element
 - But our aim is to sort the list!
 - Instead pick some value in L — pivot
 - Split L with respect to the pivot element



So, let us think of how to do this. So, supposing you could calculate the median value. So, remember that the median value is the one which divides the list into two parts where half the values are smaller than m and half the values are bigger than m . Then what you can do is you can walk through this list and every time you see a value which is smaller than or equal to the medium, you pull it on the left hand side, every time it is bigger than you move to the right hand side. So, you create two new partitions or two new halves. But now these halves are not by position, but by value. So, those values which are smaller than the median, wherever they happen in the list will come here, those values are bigger than medium will go there.

Now, because this is the median, these will be halves. I mean, the property of the median is that it is a midpoint. So, you can expect, let us assume there are no duplicates for simplicity. Then it is very easy to see that exactly half the values will be on the left and half will be in the right. So, then now, if I take the left hand sorted and I take the right hand sorted, then everything on the left is smaller than everything on the right and each of them individually is in sorted order. So, the whole thing must be in sorted order. I do not have to merge anymore.

So, if I take a recurrence for this, because of the median property, the two parts are roughly half. So, I have to sort two lists of size half. This n cost I incur to create those halves. So, unlike in the case of merge sort, where I just have to find the midpoint and say you take everything before this and I take everything after that, I do not have to actually walk through the list at that point. I just have to give the thing and say you take this and I will take that. Here, one has to actually walk through the entire list and decide which goes left, which goes right. For each value, we have to calculate whether it is smaller than the median or bigger than the median.

So, you take order n time to decompose the problem into two parts. But then you do no work to combine them except to just concatenate. But it is still this familiar recurrence. This is the same recurrence we had for merge sort. And we know that if we got this recurrence, then we have an $n \log n$ sorting algorithm. So, this would be wonderful if you could do this. If we could find the median and then use the median to split the two lists, the problem is how do we find the median.


So, when we started our discussion of sorting, we said that one of the advantages of sorting is that it helps us find the median. So, what we can do is we can sort and pick the middle element as the median. But the problem here is that what we are trying to do is sort. So, we cannot assume that we sorted, find the median and then sort again efficiently, because we have already had to do a sort before this. So, there is no point in doing it twice. So, if we find a median through sorting, then it is not acceptable, because we are using the median for sorting. So, there is a kind of circularity in this argument. So, we cannot use the median independently of unless we can find it independently of sorting and that is hard to do.

So, instead, what quicksort tries to do is to just pick some value in L . It does not necessarily mean it is the median. It just picks a value which is traditionally called a pivot element. And what you do is you do the same thing that we did for the median, except you do with respect to

the pivot. Everything that is smaller than the pivot, you move to one side, everything that is bigger than the pivot you move to the other side. So, this is this algorithm called quicksort.

(Refer Slide Time: 06:04)

Quicksort [C.A.R. Hoare]




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

High level view of quicksort

- Input list →


43	32	22	78	63	57	91	13
----	----	----	----	----	----	----	----

- Identify pivot




Madhavan Mukund

Quicksort



Quicksort [C.A.R. Hoare]



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

High level view of quicksort


- Input list

43	32	22	78	63	57	91	13
----	----	----	----	----	----	----	----

- Identify pivot
- Mark lower elements and upper elements
- Rearrange the elements as lower-pivot-upper


32	22	13	43	78	63	57	91
----	----	----	----	----	----	----	----

13 22 32 57 63 78 91



Madhavan Mukund

Quicksort



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

High level view of quicksort

 - Input list
 - Identify **pivot**
 - Mark **lower elements** and **upper elements**
 - Rearrange the elements as lower-pivot-upper
 - Recursively sort the lower and upper partitions

43 32 22 78 63 57 91 13

32 22 13 43 78 63 57 91



So, it is due to famous computer scientists called Tony Hoare, Antony Hoare C.A.R. the A stands for Antony. So, it says choose a pivot element. So, how do you choose a pivot element? We will basically choose it based on its position. You do not know anything about the values in the list. So, maybe we just pick the first element, for instance, as the pivot element. Then what you do is you partition the list into two parts, the upper part and the lower part. The lower part has everything which is smaller than the pivot. The upper part has everything which is bigger than the pivot. And now you move the pivot in between.

So, now you have everything smaller than the pivot, then you have the pivot, and then you have everything bigger than the pivot. So, now if you sort these parts, everything is with respect to each other in the correct place. So, then you have to take these two parts which are before and after the pivot and sort them again. So, you, again you quicksort for that. So, you again break them up with a pivot to a smaller half, bigger half and so on. But everything in that will stay on the left, everything here stay on the right. That is important.

So, let us look at this kind of at a high level. So, supposing this is my input list. So, first, I have to pick a pivot. So, the usual practice, as I said, is to pick the first element is a pivot. So, let us say 43 is our pivot. Now, with respect to the pivot, I have to scan these elements from left to right and decide which ones are smaller and which ones are bigger. And from that, basically, identify the lower and the upper partition. So, here, for instance, the blue ones, 32, 22, and 13 are

the smaller ones, and the green ones, 78, 63 and 57 and 91 are the bigger ones. So, these are the lower and upper partition.

So, now, I will kind of rearrange these things so that all the lower ones go to the left, the pivot comes to the middle and the higher ones go to the right. So, this is the setup now. So, now what I have to do is I have to recursively sort this part, recursively sort that part, and I am done. Why am I done, because whatever comes out of this, in general, is going to, obviously, hopefully, be 32, 13, 22 and 32 is going to always lie to the left of the pivot and everything that comes out of this, which is hopefully going to be 57, 63, 78 and 91 will lie to the right of the pivot.

So, I no longer have to do anything with these two parts with respect to each other. I can separately solve the blue guys, I can separately solve the green guys and just let them be wherever they are and they have done their job. So, that is the point of quicksort that you do not have to do a merge step afterwards, so you can in some sense disjointly work on the blue things and disjointly work on the green things and never have to ask them to talk to each other again.

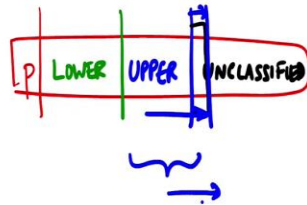
So, this is a high level view. But in order to actually implement it, we have to, this step that we did going from here to here is not obvious. It is one thing to go through and flag each element as lower or upper, but how are you going to rearrange them at the same time into this lower followed by pivot followed by the upper. So, this is really the heart of quicksort, this partitioning process, how do you partition the list into the correct sequence.

(Refer Slide Time: 09:12)

Partitioning



- Scan the list from left to right
- Four segments: **Pivot**, **Lower**, **Upper**, **Unclassified**
- Examine the first unclassified element
 - If it is larger than the pivot, extend **Upper** to include this element



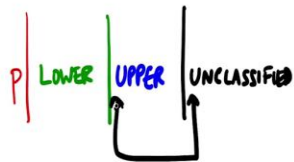
Madhavan Mukund

Quicksort

Partitioning



- Scan the list from left to right
- Four segments: **Pivot**, **Lower**, **Upper**, **Unclassified**
- Examine the first unclassified element
 - If it is larger than the pivot, extend **Upper** to include this element

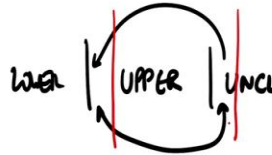


Madhavan Mukund

Quicksort

सिद्धिर्भवति कर्मजा

- Scan the list from left to right
- Four segments: **Pivot**, **Lower**, **Upper**, **Unclassified**
- Examine the first unclassified element
 - If it is larger than the pivot, extend **Upper** to include this element
 - If it is less than or equal to the pivot, exchange with the first element in **Upper**. This extends **Lower** and shifts **Upper** by one position.



Madhavan Mukund Quicksort

So, here is one partitioning strategy. It is not the only one, but it is one which is easy to remember. So, we will scan from left to right. Remember, the leftmost thing is always the pivot. So, what is left to be partitioned is everything beyond the pivot from element position one onwards. So, position 0 is the pivot. Position 1 onwards are the things that need to be scanned. So, I will kind of break it up into four parts is the pivot, then after the pivot will be all the elements which are known to be lower than the pivot, after that will be all those which are known to be upper, higher than the pivot, and then there will be the part which I have yet to seek, the unclassified ones.

So, I will be always looking at the next element as the first unclassified element and I will examine it. And now I have to decide how to adjust the lower and upper parts that I have already found. So, the claim is that if the unclassified element is larger than the pivot, I can just move. So, basically, let us try and draw a picture. So, this is my pivot. Then I will say that this is my lower segment, this is my maybe upper segment and this is my unclassified. So, this is where I am. Now, I will look at this element.

So, I will look at this element and compare it to the pivot. So, if it is bigger than the pivot, then it is easy. Then I just move this blue line to here and I have extended the upper thing and I have got one less unclassified element. If it is smaller than the pivot, then I need to put that element in the green thing. Now, naively, that would mean that I have to take all these values and shift them, because I need to make one more space in the green thing. But actually, the clever thing to do is

to just, if I have this and I want to move it into the upper thing, the clever thing to do is to just exchange this value and this value.

So, if it is less than or equal to the pivot, you exchange. So, you have the upper, you have the lower and you have this boundary and you have this unclassified. So, I am saying you just take this value and you swap it with this value. So, you are now taking a value which is lower and adding it to the lower part and you are taking a value in the upper and moving it from the beginning to the end. But the net result is that your boundary has basically shifted like this. So, your upper thing has shifted from left to right does not preserve the order. The first thing has come to the end. And in the space that you have vacated, you have moved a smaller thing. So, this is the strategy of this partition thing.

(Refer Slide Time: 12:25)

Partitioning

■ Scan the list from left to right

■ Four segments: **Pivot**, **Lower**, **Upper**, **Unclassified**

■ Examine the first unclassified element

- If it is larger than the pivot, extend **Upper** to include this element
- If it is less than or equal to the pivot, exchange with the first element in **Upper**. This extends **Lower** and shifts **Upper** by one position.

■ **Pivot** is always the first element

■ Maintain two indices to mark the end of the **Lower** and **Upper** segments

43 32 22 78 63 57 91 13

↑ ↑

Madhavan Mukund Quicksort

IIT Madras BSc Degree

सिद्धिर्भवति कर्मजा

Partitioning



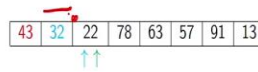
- Scan the list from left to right

- Four segments: **Pivot**, **Lower**, **Upper**, **Unclassified**

- Examine the first unclassified element

- If it is larger than the pivot, extend **Upper** to include this element

- If it is less than or equal to the pivot, exchange with the first element in **Upper**. This extends **Lower** and shifts **Upper** by one position.



- **Pivot** is always the first element

- Maintain two indices to mark the end of the **Lower** and **Upper** segments

Navigation icons: back, forward, search, etc.



Madhavan Mukund

Quicksort

Partitioning



- Scan the list from left to right

- Four segments: **Pivot**, **Lower**, **Upper**, **Unclassified**

- Examine the first unclassified element

- If it is larger than the pivot, extend **Upper** to include this element

- If it is less than or equal to the pivot, exchange with the first element in **Upper**. This extends **Lower** and shifts **Upper** by one position.



- **Pivot** is always the first element

- Maintain two indices to mark the end of the **Lower** and **Upper** segments

Navigation icons: back, forward, search, etc.



Madhavan Mukund

Quicksort

सिद्धिर्भवति कर्मजा

Partitioning



- Scan the list from left to right

- Four segments: **Pivot**, **Lower**, **Upper**, **Unclassified**

- Examine the first unclassified element

- If it is larger than the pivot, extend **Upper** to include this element

- If it is less than or equal to the pivot, exchange with the first element in **Upper**. This extends **Lower** and shifts **Upper** by one position.

43	32	22	78	63	57	91	13
----	----	----	----	----	----	----	----

- **Pivot** is always the first element

- Maintain two indices to mark the end of the **Lower** and **Upper** segments

Navigation icons: back, forward, search, etc.

Madhavan Mukund

Quicksort



Partitioning



- Scan the list from left to right

- Four segments: **Pivot**, **Lower**, **Upper**, **Unclassified**

- Examine the first unclassified element

- If it is larger than the pivot, extend **Upper** to include this element

- If it is less than or equal to the pivot, exchange with the first element in **Upper**. This extends **Lower** and shifts **Upper** by one position.

43	32	22	78	63	57	91	13
----	----	----	----	----	----	----	----

- **Pivot** is always the first element

- Maintain two indices to mark the end of the **Lower** and **Upper** segments

Navigation icons: back, forward, search, etc.

Madhavan Mukund

Quicksort



INDIAN INSTITUTE OF TECHNOLOGY MADRAS

सिद्धिर्भवति कर्मजा

Partitioning



- Scan the list from left to right

- Four segments: **Pivot**, **Lower**, **Upper**, **Unclassified**

- Examine the first unclassified element

- If it is larger than the pivot, extend **Upper** to include this element

- If it is less than or equal to the pivot, exchange with the first element in **Upper**. This extends **Lower** and shifts **Upper** by one position.



- **Pivot** is always the first element

- Maintain two indices to mark the end of the **Lower** and **Upper** segments

Navigation icons: back, forward, search, etc.

Madhavan Mukund

Quicksort



Partitioning



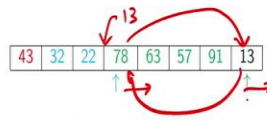
- Scan the list from left to right

- Four segments: **Pivot**, **Lower**, **Upper**, **Unclassified**

- Examine the first unclassified element

- If it is larger than the pivot, extend **Upper** to include this element

- If it is less than or equal to the pivot, exchange with the first element in **Upper**. This extends **Lower** and shifts **Upper** by one position.



- **Pivot** is always the first element

- Maintain two indices to mark the end of the **Lower** and **Upper** segments

Navigation icons: back, forward, search, etc.

Madhavan Mukund

Quicksort



सिद्धिर्भवति कर्मजा

Partitioning



- Scan the list from left to right
- Four segments: **Pivot**, **Lower**, **Upper**, **Unclassified**
- Examine the first unclassified element
 - If it is larger than the pivot, extend **Upper** to include this element
 - If it is less than or equal to the pivot, exchange with the first element in **Upper**. This extends **Lower** and shifts **Upper** by one position.
- **Pivot** is always the first element
- Maintain two indices to mark the end of the **Lower** and **Upper** segments



Navigation icons: back, forward, search, etc.

Madhavan Mukund Quicksort

Partitioning



- Scan the list from left to right
- Four segments: **Pivot**, **Lower**, **Upper**, **Unclassified**
- Examine the first unclassified element
 - If it is larger than the pivot, extend **Upper** to include this element
 - If it is less than or equal to the pivot, exchange with the first element in **Upper**. This extends **Lower** and shifts **Upper** by one position.
- **Pivot** is always the first element
- Maintain two indices to mark the end of the **Lower** and **Upper** segments
- After partitioning, exchange the pivot with the last element of the **Lower** segment



Navigation icons: back, forward, search, etc.

Madhavan Mukund Quicksort

So, let us see how it works. So, you start with this 43 this thing. So, now you first identify the pivot and you make a kind of marker of where you are two segments end. So, we are using this Python convention where that marker is one beyond. So, the blue and the green at position one means that the left and right things are empty right now, because they do not include the position of the marker.

Now, I look at 32. So, 32 is a lower element. And if I do that thing right now it does not make much sense. But if you do that thing, you will end up shifting. So, now it says the blue segment is actually from here to here and the green segment is from here to here and 22 is my next unclassified element. So, now for 22, again, it shifts. Now, for 78, because 78 is an upper thing,

the upper segment grows, for 63 the upper segment grows, for 57 the upper segment grows, for 91 the upper segment grows. Now, this is the case which illustrates what happens most clearly at this point.

So, now, 13 is smaller. So, I need to put 13 here. So, what I am claiming is I will move the 78 here and I will move the 13 here and I will move both of these pointers. That is what I am going to do. So, I am going to basically, so look at the previous picture. So, the previous picture, the blue segment was ending below 78 and the green segment was ending below 13. Now, I will move 13 to the position of 78 and 78 to the position of 13 and move both those arrows to the right. So, this is now, at the end of this pass, I now have pivot, lower and upper and nothing is unclassified, because I have moved to the end of the list.

Now, I have to move the pivot in between. So, I will use the same trick. So, to move the pivot in between I ideally would have to shift the three lower elements to the left and make a space, but I could as well take the rightmost element in that 13 and swap it with the 43. So, I will just swap the 13 and the 43 and now I have the lower elements to the left of the pivot and the upper element to the right. So, this is my partitioning strategy.

(Refer Slide Time: 14:39)

Quicksort code

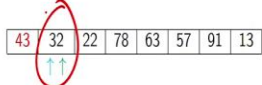



- Scan the list from left to right
- Four segments: **Pivot**, **Lower**, **Upper**, **Unclassified**
- Classify the first unclassified element
 - If it is larger than the pivot, extend **Upper** to include this element
 - If it is less than or equal to the pivot, exchange with the first element in **Upper**. This extends **Lower** and shifts **Upper** by one position.

```
def quicksort(L,l,r): # Sort L[l:r]
    if (r - l <= 1):
        return(L)
    (pivot,lower,upper) = (L[l],l+1,l+1)
    for i in range(l+1,r):
        if L[i] > pivot: # Extend upper segment
            upper = upper+1
        else: # Exchange L[i] with start of upper segment
            (L[i], L[lower]) = (L[lower], L[i])
            # Shift both segments
            (lower,upper) = (lower+1,upper+1)
    # Move pivot between lower and upper
    (L[l], L[lower-1]) = (L[lower-1], L[l])
    lower = lower-1
    # Recursive calls
    quicksort(L,l,lower)
    quicksort(L,lower+1,upper)
    return(L)
```



- Scan the list from left to right
- Four segments: **Pivot**, **Lower**, **Upper**, **Unclassified**
- Examine the first unclassified element
 - If it is larger than the pivot, extend **Upper** to include this element
 - If it is less than or equal to the pivot, exchange with the first element in **Upper**. This extends **Lower** and shifts **Upper** by one position.
- **Pivot** is always the first element
- Maintain two indices to mark the end of the **Lower** and **Upper** segments





Madhavan Mukund
Quicksort

So, with this strategy we can now write this algorithm quite clearly. So, I want to do quicksort. In general, I have to do quicksort remember on different segments. So, I do quicksort on the slice from l to r . If the slice is trivial, that is it has at most one element, then as usual, I return. Now, if this slice is not trivial, then what I do is that I will fix the pivot to be the beginning of the slice and I will fix initially, if you remember, the lower and the upper segments were the positions immediately to the right of the pivot. So, they were, if the pivot is at small l , then the lower and the upper things are at l plus 1.

Now, I will scan through all the unclassified things. If the unclassified thing is bigger than the pivot, I extend the upper segment. That is what this is. Otherwise, I will exchange the current position with the first upper thing. So, remember, lower is pointing just to the right of where the lower segment ends. So, this index lower is actually the first upper position. So, I will swap those, and then I will shift both the lower and the upper markers by 1. So, this is that partitioning step. Now, at the end of this loop, I finished partitioning.

Now, I will do this business about swapping the pivot and the last element in the lower partition and then I will move the lower partition limit by one because I have kind of shifted the whole thing down by one and now I call quicksort on the left half and I call quicksort on the right half. Notice that here I am doing it up to lower minus 1 and from lower plus 1, so I am leaving out the position where the pivot is, because I do not want to again, the pivot is already in the right place.

So, I am doing everything to the left of the pivot and everything to right of the pivot. So that is this quicksort.

(Refer Slide Time: 16:33)

Summary

- Quicksort uses divide and conquer, like merge sort
- By partitioning the list carefully, we avoid a merge step
 - This allows an in place sort
- We can also provide an iterative implementation to avoid the cost of recursive calls
- The partitioning strategy we described is not the only one used in the literature
 - Can build the lower and upper segments from opposite ends and meet in the middle

Diagram illustrating the partitioning process: A horizontal array is shown with a pivot element. Elements less than the pivot are moved to the left (L) and elements greater than the pivot are moved to the right (U). The process is iterative, with arrows indicating the movement of elements from both ends towards the center.

Summary

- Quicksort uses divide and conquer, like merge sort
- By partitioning the list carefully, we avoid a merge step
 - This allows an in place sort
- We can also provide an iterative implementation to avoid the cost of recursive calls
- The partitioning strategy we described is not the only one used in the literature
 - Can build the lower and upper segments from opposite ends and meet in the middle
- Need to analyse the complexity of quick sort

So, this is again, a divide and conquer thing, except is doing the division in a more clever way, it is partitioning the list in a careful way based on the value. So, we are avoiding this merge step. And this actually allows us to sort it in place. So, actually, you will see that that code if you run it, you will find that the list actually gets updated in place.

And we can also provide an iterative implementation, which I will discuss briefly later, to avoid the cost of recursive calls. And we have given one partitioning strategy which basically starts from the left and kind of builds both these partitions as we go along from left to right. Now, there are other strategies which start at the opposite end. So, I start with this and I will have some kind of lower here and upper here and I will grow them towards the middle. So, this is also a strategy which has been studied and implemented.

So, there is more than one way to implement quicksort. So, it is a question also, but it is really, the partitioning algorithm is usually the one where you make a mistake. So, you have to be careful that you are doing the right thing in partitioning, because after that the recursive call is very straightforward. So, now, in order to justify this we actually have to analyze the complexity and see if we have really done something which improves on the shortcomings of merge sort without sacrificing the advantages of merge sort.

