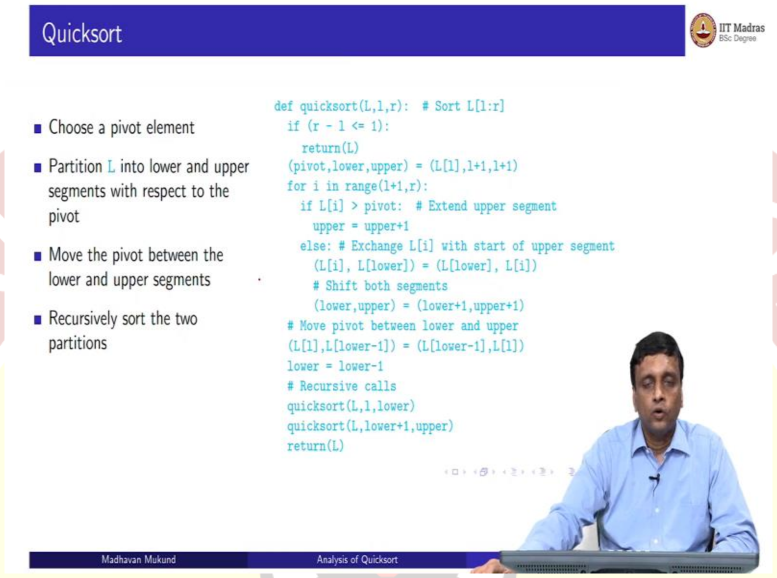
IIT Madras

ONLINE DEGREE

Programming, Data Structures and Algorithms using Python
Professor Madhavan Mukund
Analysis of Quicksort

(Refer Slide Time: 00:11)



Quicksort

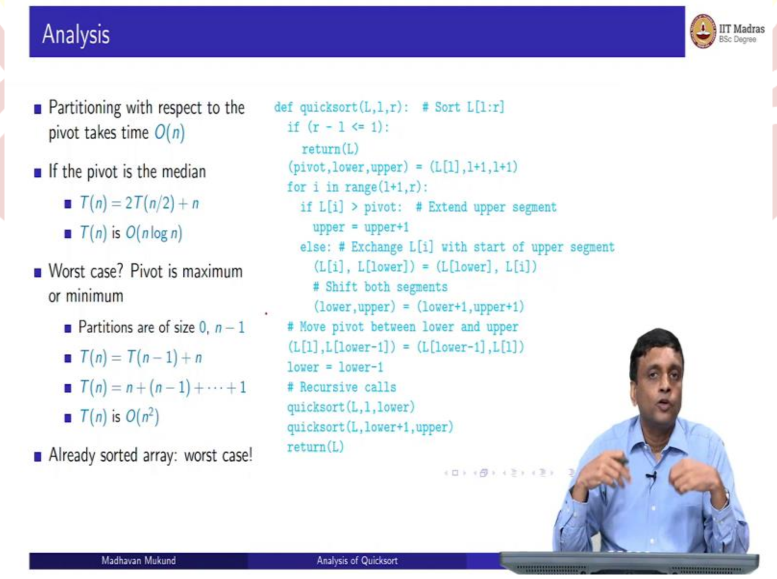
- Choose a pivot element
- Partition L into lower and upper segments with respect to the pivot
- Move the pivot between the lower and upper segments
- Recursively sort the two partitions

```
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)
```

Madhavan Mukund Analysis of Quicksort

So, let us analyze quicksort. So, remember, this is how quicksort works. It chooses a pivot, partitions it, moves the pivot between the lower and right segments, and then recursively sorts the two partitions. So, it all amounts to really asking how well this partitioning works in terms of dividing the array into two or the list into two smaller parts.

(Refer Slide Time: 00:28)



Analysis

- Partitioning with respect to the pivot takes time $O(n)$
- If the pivot is the median
  - $T(n) = 2T(n/2) + n$
  - $T(n)$ is $O(n \log n)$
- Worst case? Pivot is maximum or minimum
  - Partitions are of size $0, n-1$
  - $T(n) = T(n-1) + n$
  - $T(n) = n + (n-1) + \cdots + 1$
  - $T(n)$ is $O(n^2)$
- Already sorted array: worst case!

```
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)
```

Madhavan Mukund Analysis of Quicksort

So, partitioning itself takes linear time, because we saw that in one scan of the list, we can do the partitioning. But the real question is, what are we partition, partitioning with respect to. So, if the pivot is actually the median, then we know that the two halves are roughly equal in size. So, we know that the lower half has half the elements, the upper half has half the elements. So, we get our familiar merge sort recurrence, which is Tn is 2 of T2 times n by 2 plus the partitioning cost of n and we get an n log n algorithm.

If this were indeed the case, we would be in business because now we have got an in place algorithm, which I also claimed can be done iteratively. But it does not have the cost of the recursion and the extra space of merge sort, but it is as good theoretically as an upper bound. But unfortunately, this is not going to be the case. So, in the worst case, because you have no control over the pivot, because it is picking it up from someplace in the list, the first element of the list need not be the median. You are just picking it up without analyzing the values.

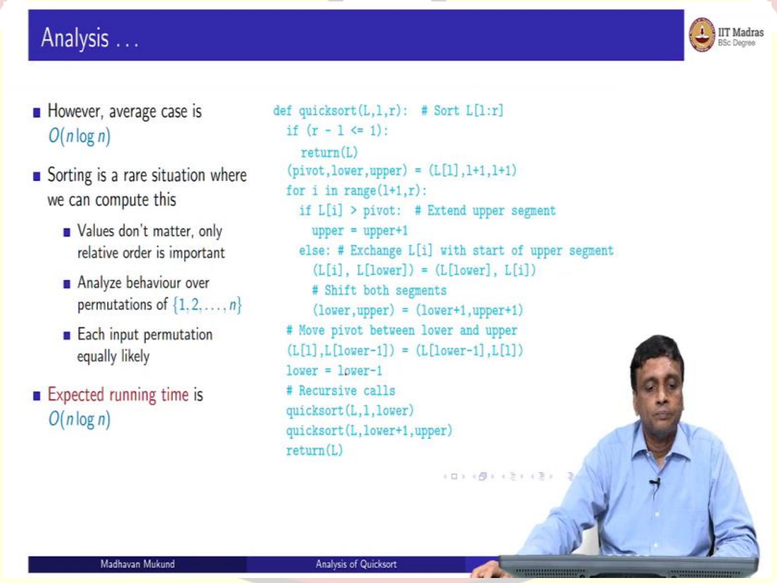
In the worst case is either the smallest or the largest value. So, if it is the smallest or the largest value, what will happen is that every other value will be either smaller than it or bigger than it. So, one of the two partitions, either everything will go into the lower part and nothing will go into the upper part or everything will go into the upper part and nothing will go into the lower part. So, worst case, your partitions will have size 0 and n minus 1, instead of n by 2, n by 2, of course, the pivot will not be there, so n will reduce. But it will reduce in this very asymmetric way, either the lower will be empty and the upper will be n minus 1 elements or the lower will have n minus 1 elements and the upper will have.

So, remember this worst case. We cannot avoid the worst case. So, this might happen. So, in this case, then the recurrence in the worst case says that to sort n, I end up having to sort the larger of the two partitions, which in this case is n minus 1 and I have spent order n work getting there. So, I have exactly the same recurrence that we had for selection, for insertion sort when we did it recursively. So, I have Tn is T n minus 1 plus n and this ends up being n square.

So, unfortunately, this very clever strategy of avoiding the merge has a worst case complexity, which is n square. And what is the worst case? Well, paradoxically, the worst case is one where in fact, the array is already sorted. So, for instance, if I am sorting it in ascending order and I give you an array in ascending order and I pick the first element as a pivot, then the first element is going to be the smallest one. So, it is going to produce an upper partition consisting all the remaining elements.

But remember, they are going to be generated in the same sequence. So, the upper partition will again be a sorted sequence with the first element is the smallest one. So, if I pick that one, again, it is going to produce an upper thing with n minus 2 sorted things and so on. So, the case which is actually bad is the case which should be good. Like we said in insertion sort I give you an already sorted sequence in the correct sequence sorted in the same sequence that I am looking for. It will actually work well. Here, it actually works very badly.

(Refer Slide Time: 03:34)



Analysis ...

- However, average case is $O(n \log n)$
- Sorting is a rare situation where we can compute this
  - Values don't matter, only relative order is important
  - Analyze behaviour over permutations of $\{1, 2, \ldots, n\}$
  - Each input permutation equally likely
- Expected running time is $O(n \log n)$

```
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)
```

Madhavan Mukund   Analysis of Quicksort

But there is some silver lining. So, actually, one can show that the average case for quicksort, we are not going to prove it, but I am just going to claim it. The average case is m log n. So, we had a discussion earlier about what this entails. Average case means we have to talk about all inputs, some distribution over the inputs, and then somehow look at the expected running time and all that. So, in this case, how does it work?

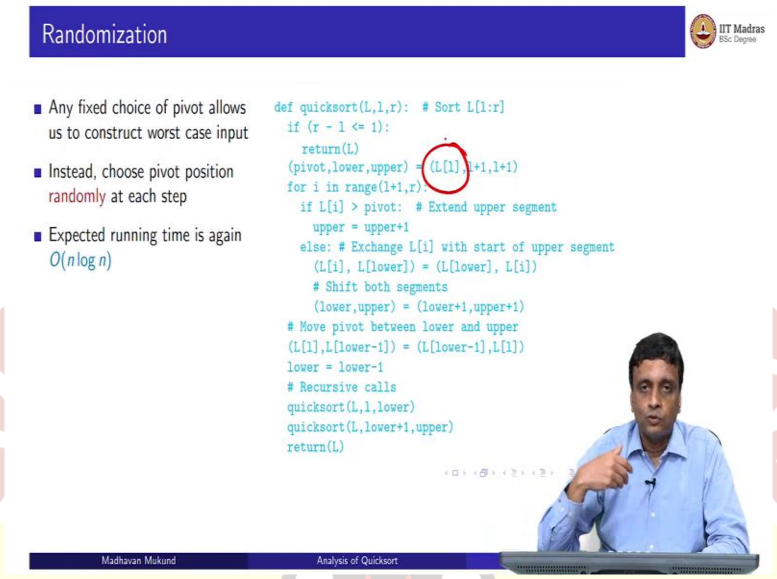
Well, the first thing that is there is that in principle you have infinitely many arrays of a fixed length. Even if I say that the length is n, there is no limit to the number of arrays or lists you can construct of length n, because you can keep changing the values arbitrarily. But when I am doing compare and swap as my sorting, it really does not matter where, if I give you a list with 1, 2, 3 and if I give a list of 10, 20, 30, it really does not matter. They are both the same list as far as the algorithm is concerned, because the first position is smaller than second position is smaller than the third position.

So, the actual values are not important. What is important is their relative order, which is the biggest one, which is the second biggest one. So, I can always think of any input of size n as being a permutation of n elements. It tells me the biggest element is in one position, the second biggest is somewhere else, and so on. It really does not matter what the values are. So, this allows me to now first bound the space over which I am calculating the probability. I can say that for input of size n, I am looking at all n factorial permutations of 1 to n. And then I have to make an assumption.

But in the case of sorting you can imagine that somebody is giving you a list to sort there is no bias, every permutation is equally likely. So, you can assume that for the probability part that each is equally likely. And then because it is exhaustive, you can actually do a count and verify this. So, that is how this thing comes out, saying that the expected running time for inputs of size n over all permutations of size n you can actually calculate its n log n, even though there are going to be some inputs which are going to be worst case n square.

So, although, so, therefore, in some sense, n square is a rare case, because n log n turns out to be the average case. So, this is not possible for most scenarios and algorithms, but for sorting it is possible. And in particular, it has been done for quicksort to show that its expected running times n log n.

(Refer Slide Time: 05:52)



Randomization

- Any fixed choice of pivot allows us to construct worst case input
- Instead, choose pivot position randomly at each step
- Expected running time is again $O(n \log n)$

```
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)
```

Madhavan Mukund Analysis of Quicksort

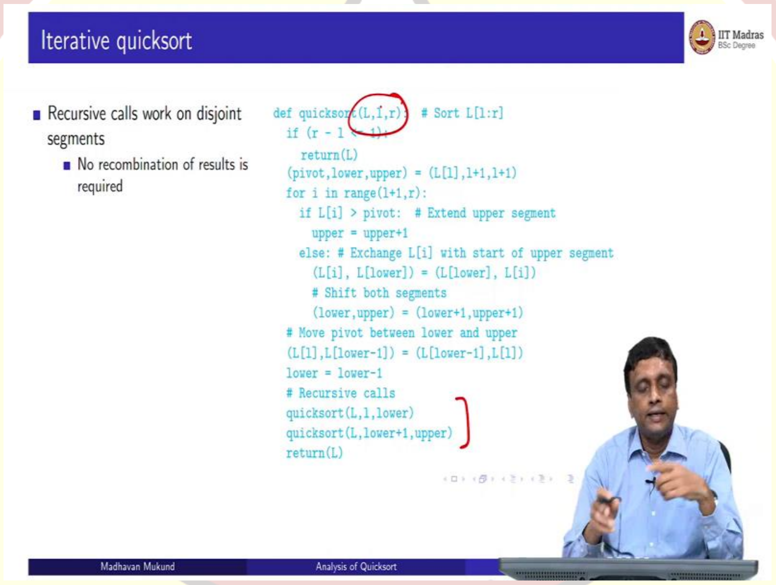
So, there is another way to beat, I mean, to exploit this to beat this worst case in the case of quicksort. So, the real problem with quicksort turns out to be that choosing the pivot using a fixed position gives us a problem. So, we saw that if the first position is our pivot, then if we put the smallest value at the first position each time, we can kind of build up an array or a list which will always give us a worst case behavior.

Supposing you say, no, I am not going to take the first position, I am going to take the last position, then I will give you a symmetric input, which will be bad for that. If you say, I am going to pick the middle position, then I can make sure that in the input I construct that the first element, the extreme element is going to be the middle, then I will run your quicksort implementation to figure out what time is left, what happened the lies and then I can put again the second minimum and the third minimum at the midpoints of the two partitions that you find.

So, I can always reconstruct a worst case and put if you have a fixed strategy for finding the pivot. So, what is the solution? The solution is that you do not have a fixed strategy. At every time when you want to run quicksort, you have to fix a pivot and then partition. But you do not fix a pivot by choosing the first element or the middle element or the last element, you kind of pick a random value between 0 and n minus 1, uniformly. So, you just generate a random number uniformly with probability 1 by n between 0 and n minus 1 and you say, okay, for now, this is the pivot, next time will be something else.

So, since you are picking the pivot at random, there is no way to, I mean, in some sense, intuitively for somebody adversarial to give you a bad input. So, if you do the calculation in this randomize sense, where each time you pick the pivot, so this pivot is not always the left-hand side of your list, but it is some random position, which you calculate with each iteration, then it turns out that again you can show that you have an expected running time of n log n. So, this is a different way of achieving that average case. And it is an easier way in some sense, because what it means is that when you actually implement it, if somebody gives you a worst case input your algorithm is not necessarily going to be stuck in that worst case.

(Refer Slide Time: 07:54)

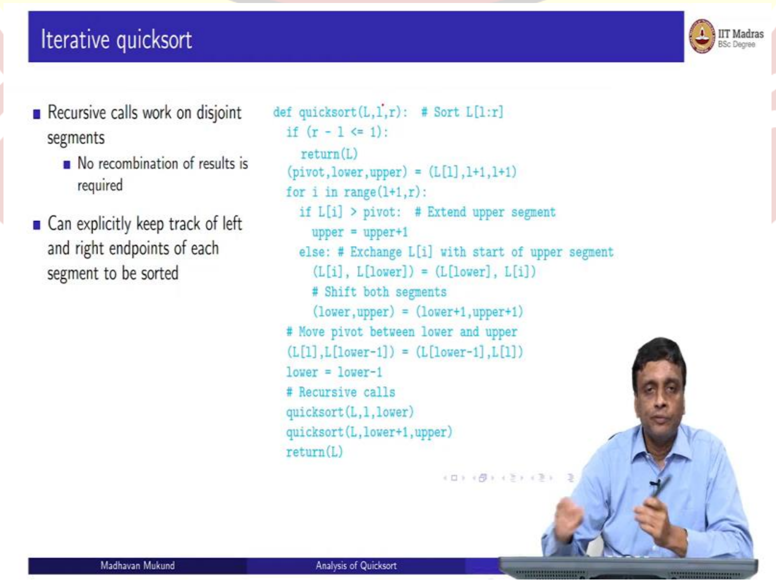


Iterative quicksort

- Recursive calls work on disjoint segments
  - No recombination of results is required

```
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)
```

Madhavan Mukund   Analysis of Quicksort



Iterative quicksort

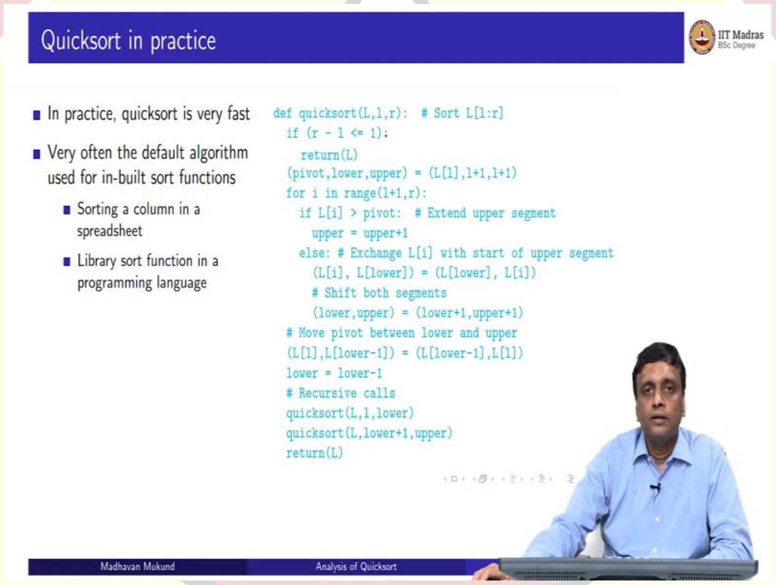
- Recursive calls work on disjoint segments
  - No recombination of results is required
- Can explicitly keep track of left and right endpoints of each segment to be sorted

```
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)
```

Madhavan Mukund   Analysis of Quicksort

We also mentioned this iterative quicksort. So, I will not go into it in much detail. But just to suffice to say that, basically these calls are happening on disjoint parts of the array. So, since I am anyway telling quicksort to work within a bounded interval from left to right, when I work on this interval it does not influence anything else, when I work in that. So, I can always rewrite this code to work iteratively on each segment between l and r minus 1. So, I would not go into the details, but you can convince yourself that this algorithm can actually be implemented iteratively. So, when we are doing this iterative thing, we have to each time when we are running quicksort, we have to know between which r we are working.

(Refer Slide Time: 08:33)



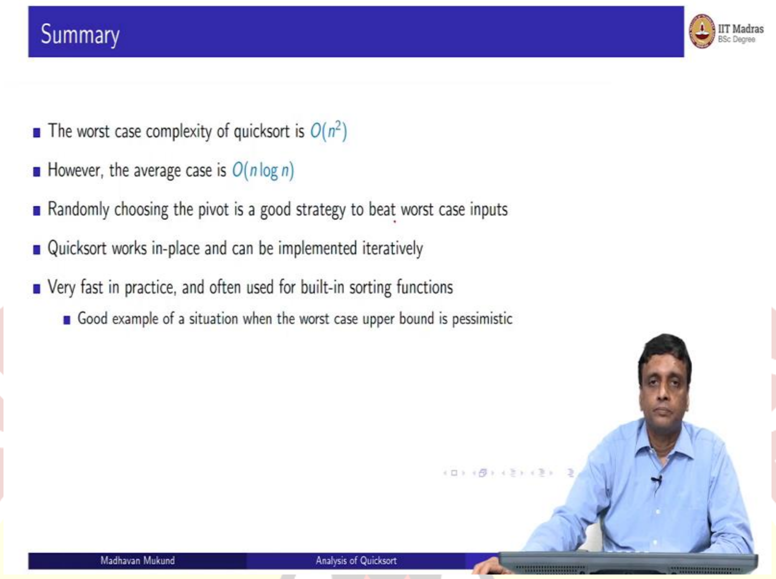
Quicksort in practice

- In practice, quicksort is very fast
- Very often the default algorithm used for in-built sort functions
  - Sorting a column in a spreadsheet
  - Library sort function in a programming language

```
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)
```

So, given all this, you might ask why are we so interested in quicksort? Well, it turns out that quicksort despite its order n square upper bound is actually very fast in practice. So, in many situations you use built in sorting. You take a spreadsheet, you take a column and you say sort it, or you might have a sort function which you can call like we have in Python, like l dot sort or sorted, so you might have a function that you can call for free in a programming language. You do not have implement it. So, in many such cases, actually quicksort is the algorithm that is used. So, that shows basically that despite the worst case it actually works this implementation without the overhead of merge sort actually makes it competitive and very efficient in practice.

(Refer Slide Time: 09:19)



Summary

- The worst case complexity of quicksort is $O(n^2)$
- However, the average case is $O(n \log n)$
- Randomly choosing the pivot is a good strategy to beat worst case inputs
- Quicksort works in-place and can be implemented iteratively
- Very fast in practice, and often used for built-in sorting functions
  - Good example of a situation when the worst case upper bound is pessimistic

So, to summarize, the worst case complexity is n square, but actually you can calculate the average is n log n. And one way to kind of achieve this average is to have a randomized strategy to choose the pivot. So, at each time you want to pivot the thing you pick a position with uniform probability between the beginning and the end. So, quicksort overcomes some of the limitations of merge sort, in that it works in place, it does not require you to construct a new array, even though it is can be implemented recursively and it also can be implemented non-recursively, it can be done iteratively.

And the main selling point of quicksort is that it is very fast in practice. So, it is often used for built-in sorting functions and it kind of illustrates the point that we made that using upper bound as a prediction of the overall behavior of an algorithm is often very pessimistic and this is one real life situation where this pessimism actually shows up.