INDIAN INSTITUTE OF TECHNOLOGY MADRAS

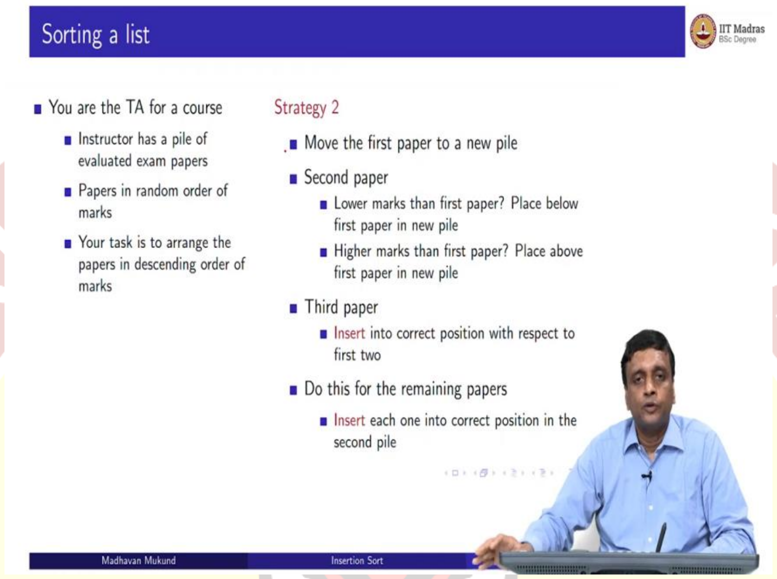
सिद्धिर्भवति कर्मजा

IIT Madras

ONLINE DEGREE

Programming, Data Structures and Algorithms Using Python
Professor. Madhavan Mukund
Insertion Sort

(Refer Slide Time: 0:09)



Sorting a list

- You are the TA for a course
  - Instructor has a pile of evaluated exam papers
  - Papers in random order of marks
  - Your task is to arrange the papers in descending order of marks

Strategy 2

- Move the first paper to a new pile
- Second paper
  - Lower marks than first paper? Place below first paper in new pile
  - Higher marks than first paper? Place above first paper in new pile
- Third paper
  - Insert into correct position with respect to first two
- Do this for the remaining papers
  - Insert each one into correct position in the second pile

Madhavan Mukund   Insertion Sort

So, we have seen one intuitive algorithm for sorting selection sort. So, here is another one. So, remember the scenario you are the TA, the instructor has asked you to arrange the graded papers in order of marks. So, the second algorithm works as follows.

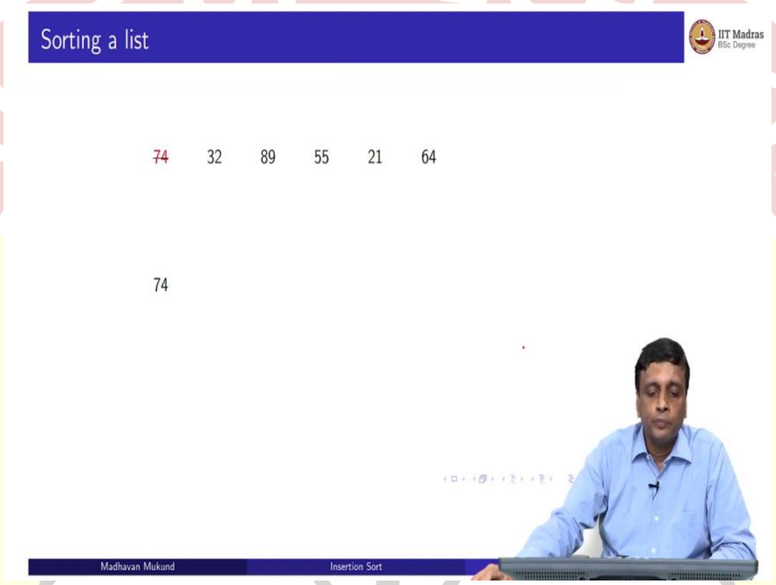
So, you look at the first paper and you put it into a new pile, so you start off a pile. And we are now going to the earlier case, you made sure that the pile was going to be sorted by making sure that you carefully put the minimum value there, now, you are just going to put the first value you see over there.

Now, this does not guarantee it is going to be sorted. At least for one, it is sorted. But now when you move the second paper, you have to put it correctly. So, if it if you want the minimum at the bottom of the pile, and the second paper is smaller, you have to slide it below. If it is bigger, you put it on top.

So, depending on the value of the second paper compared to the first paper, you have to put it in the position. Now if you come to the third paper, there are three possibilities, so you have paper 1, you have paper 2 and now paper 3 could go below, both of them go above both of them or go in between, depending on what its value is.

So, in general, you have to keep doing this for every paper for every paper, you take the new pile, which is already has been sorted because of way you created it. But now in order to maintain the fact that it is sorted, you cannot just dump it on top as you did in selections, or you have to find the correct position to insert. So, that is why this is called insertion sort because every time you have to insert the next value into the sorted list that you are creating.

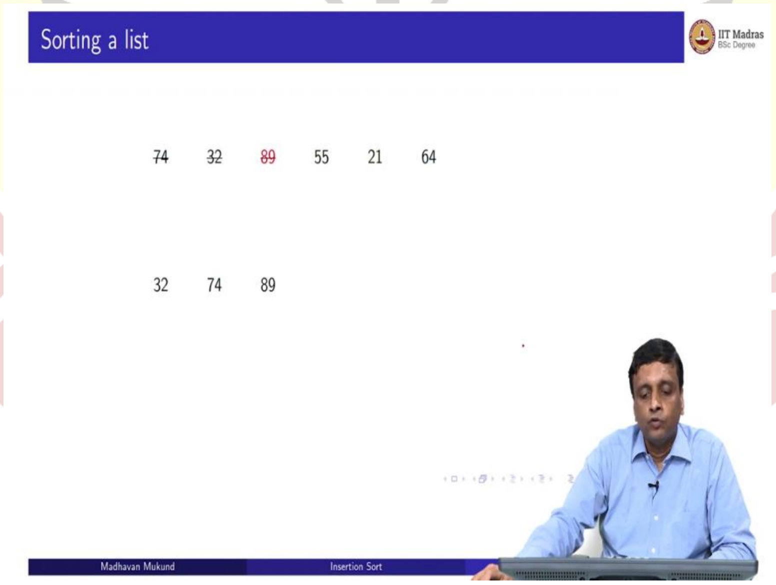
(Refer Slide Time: 1:33)



Sorting a list

~~74~~ 32 89 55 21 64

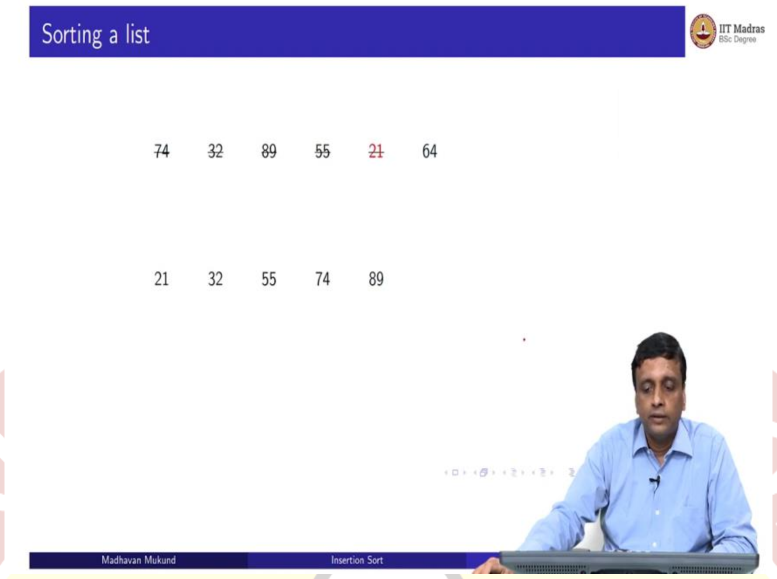
74



Sorting a list

~~74~~ ~~32~~ ~~89~~ 55 21 64

32 74 89

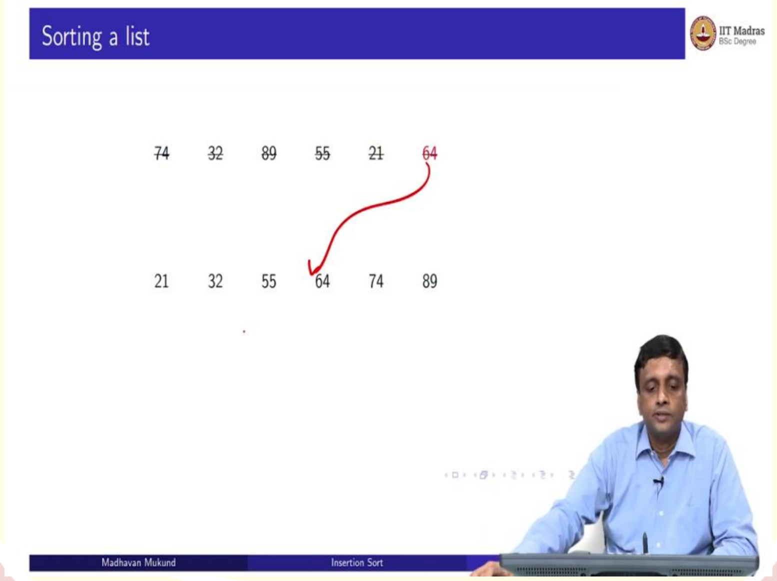


Sorting a list

~~74~~ ~~32~~ ~~89~~ ~~55~~ ~~21~~ 64

21 32 55 74 89

Madhavan Mukund | Insertion Sort



Sorting a list

~~74~~ ~~32~~ ~~89~~ ~~55~~ ~~21~~ ~~64~~

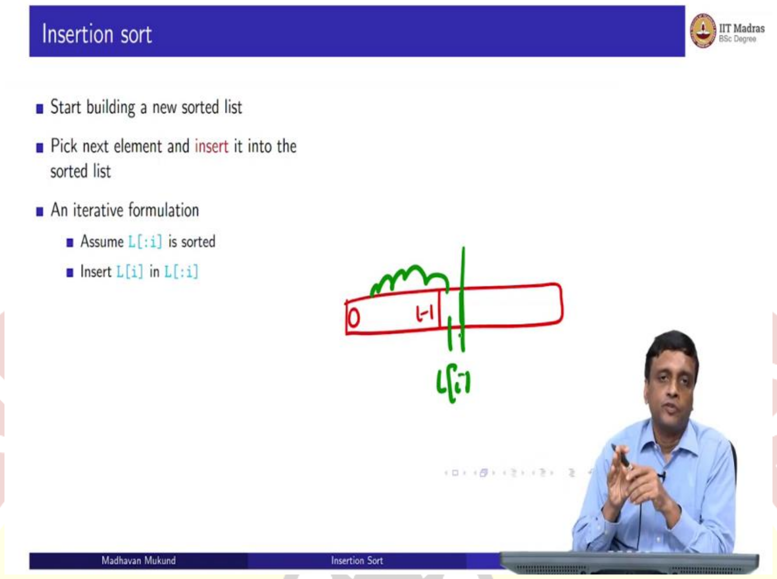
21 32 55 64 74 89

Madhavan Mukund | Insertion Sort

So, let us do this Insertion Sort now, just we will do it from left to right. So, I start from, I start building up the new list from the left, so I pick up the 74 and I create a new list. Now when I pick up the 32 because 32 smaller than 74, it has to go to the left. So my new list changes. So, 74 gets pushed 89 when I pick up next, is now bigger than everything.

So, it goes to the top of the pile. So, it just stays at the top and nothing gets disturbed. For 55, I have to move it down and push it into correct place. 21 goes all the way to the bottom and pushes everything to the right. And 64 again goes in the middle and pushes 2 elements to right. So, this is the way of insertion. So, basically 64 came and inserted itself between the 55 and the 74. Hence, Insertion Sort.

(Refer Slide Time: 2:21)



Insertion sort

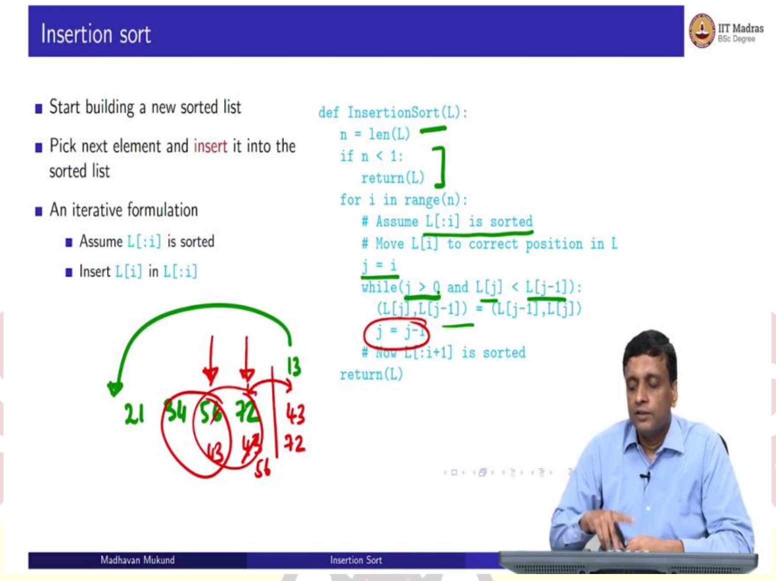
- Start building a new sorted list
- Pick next element and insert it into the sorted list
- An iterative formulation
  - Assume L[:i] is sorted
  - Insert L[i] in L[:i]

So, an insertion sort, basically you build a new list, you pick the next element and insert it into this list. So initially, you build a list of size one, which is sorted by the fact that it has only one element, every one element list is by definition sorted. Now once you have two elements by insertion, you make sure that the two element list is sorted, then the three element list is sorted and so on.

So, if you want to do this iteratively, rather than it is like our selection sort, if you do not want to actually build a new list, what you can assume is that so far, I have somehow scan from 0 to i minus 1 and I have inserted them so that they are sorted. Now I take this L[i] and I move backwards and insert it into the position, then I would have grown my sorted prefix up to here.

So, in selection sort the sorted prefix was created by swapping the minimum with the beginning of the prefix. Here, instead, what we do is we take the next element, which is not in the correct, which is not yet part of the sorted prefix, and I inserted by moving it back, so the sorted prefix will shift a little bit and make space for it.

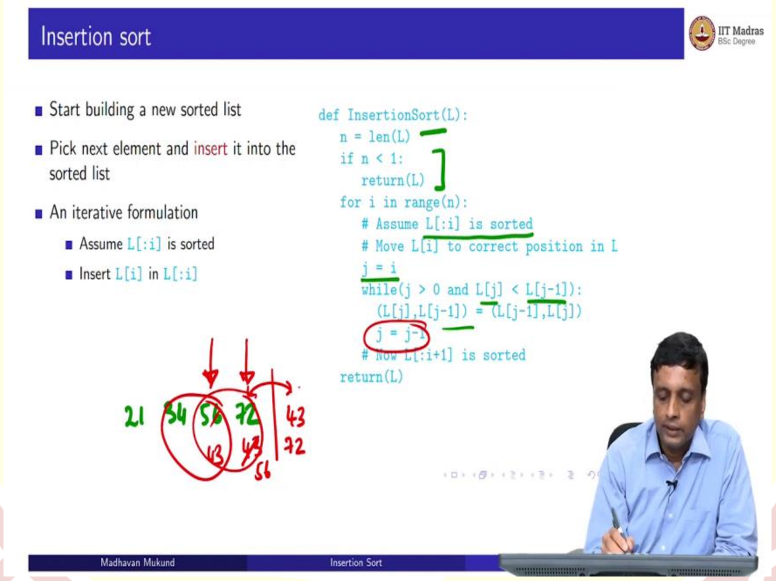
(Refer Slide Time: 3:35)



Insertion sort

- Start building a new sorted list
- Pick next element and insert it into the sorted list
- An iterative formulation
  - Assume L[:i] is sorted
  - Insert L[i] in L[:i]

```
def InsertionSort(L):
  n = len(L)
  if n < 1:
    return(L)
  for i in range(n):
    # Assume L[:i] is sorted
    # Move L[i] to correct position in L
    j = i
    while(j > 0 and L[j] < L[j-1]):
      (L[j],L[j-1]) = (L[j-1],L[j])
      j = j-1
    # Now L[:i+1] is sorted
  return(L)
```



Insertion sort

- Start building a new sorted list
- Pick next element and insert it into the sorted list
- An iterative formulation
  - Assume L[:i] is sorted
  - Insert L[i] in L[:i]

```
def InsertionSort(L):
  n = len(L)
  if n < 1:
    return(L)
  for i in range(n):
    # Assume L[:i] is sorted
    # Move L[i] to correct position in L
    j = i
    while(j > 0 and L[j] < L[j-1]):
      (L[j],L[j-1]) = (L[j-1],L[j])
      j = j-1
    # Now L[:i+1] is sorted
  return(L)
```

So, here is Insertion Sort. So, it is actually simpler to write than selection sort of you remember the amount of code there. So, what we do is that again, if we compute the list length, and if the length is 0, then we return otherwise, now we assume that we have already sorted again, like selection sort, there is an invariant that we are looking at position i, we assume that 0 to i minus 1 is already sorted.

So, we start with the ith position and now, we want to go backwards. So, if L[j] is smaller than L[j-1], then we want to swap and continue. So, supposing I had, whatever it was, so say 21, 34

56, 72 and now, I find say, the next element that I am looking at is a 43. So, what I will do is that this is my i, so I will compare these two, and then I will exchange them.

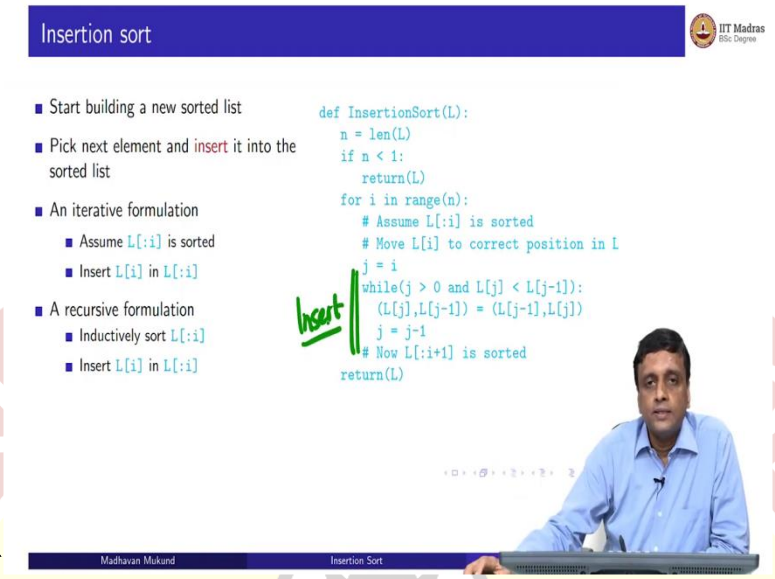
And now I will shift my attention to this position, because that is where I am. So, I will do j equals j-1, then I will compare these two and if necessary I will swap them. So, I will now make this into 56 and this into 43. And now I will move my attention to this position, and then I will compare these two positions and see and now I do not need to swap them. So, I will finish.

Now the other possibility that could happen is that I could go all the way and eventually supposing that we not 43 but say 13, then eventually it'd have gone all the way there. And then I would have reached the beginning of the list. And now if I start looking at j minus 1, it is not a valid value in Python will throw an index error. That is why I have this extra thing saying; do not try to look at the previous element if you are already at the beginning.

So, if you happen to insert and push it all the way back, then just stop, So either you stop, because you have reached the beginning, you have inserted it by pushing it all the way to the end, or you stop because there is a point beyond which it should not move anymore, because the previous element is actually this element is bigger than or equal to the previous element.

It is not smaller than so this is the insert thing. And once you have done this by moving this L, L of i backwards to the correct position, you can claim that the sorted prefix has been extended from i to plus 1. So, the invariant has been extended. And in the limit, as I go to the last position, I will have extended the prefix to the entire thing. So, the whole thing will be sorted. So, this is Insertion Sort.

(Refer Slide Time: 6:00)



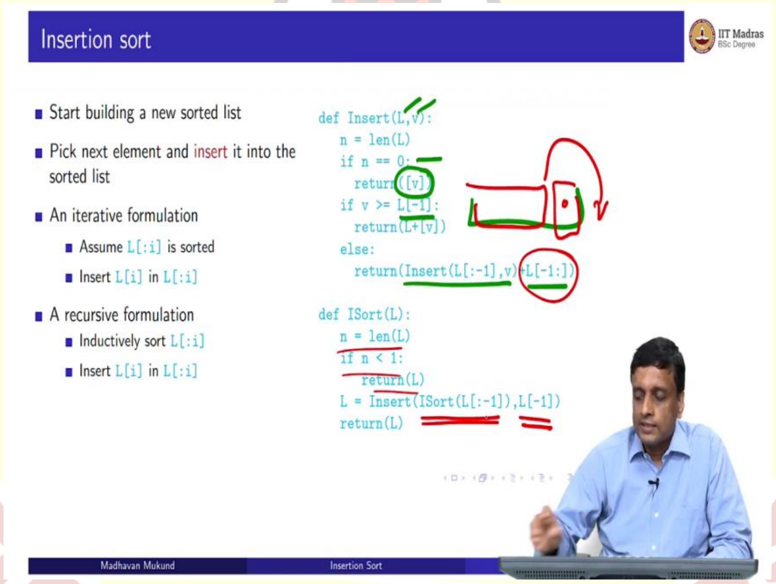
Insertion sort

- Start building a new sorted list
- Pick next element and insert it into the sorted list
- An iterative formulation
  - Assume L[:i] is sorted
  - Insert L[i] in L[:i]
- A recursive formulation
  - Inductively sort L[:i]
  - Insert L[i] in L[:i]

```
def InsertionSort(L):
  n = len(L)
  if n < 1:
    return(L)
  for i in range(n):
    # Assume L[:i] is sorted
    # Move L[i] to correct position in L
    j = i
    while(j > 0 and L[j] < L[j-1]):
      (L[j],L[j-1]) = (L[j-1],L[j])
      j = j-1
    # Now L[:i+1] is sorted
  return(L)
```

Insert

Madhavan Mukund Insertion Sort



Insertion sort

- Start building a new sorted list
- Pick next element and insert it into the sorted list
- An iterative formulation
  - Assume L[:i] is sorted
  - Insert L[i] in L[:i]
- A recursive formulation
  - Inductively sort L[:i]
  - Insert L[i] in L[:i]

```
def Insert(L,v):
  n = len(L)
  if n == 0:
    return([v])
  if v >= L[-1]:
    return(L+[v])
  else:
    return(Insert(L[:-1],v)+L[-1:])

def ISort(L):
  n = len(L)
  if n < 1:
    return(L)
  L = Insert(ISort(L[:-1]),L[-1])
  return(L)
```

Madhavan Mukund Insertion Sort

Now, we can also do this, just for illustration, recursively. So, we can assume that this thing is sorted as before, and now we can insert it, but the sorting happens inductively. So, this is also useful to just think about it, because you need some practice are also looking many problems that are actually easier to formulate recursively.

So, in this case, it is not very difficult to do it both ways. But let us look at it recursively. So first, let us look at insert. So, we have this operation here which is actually inserting, so this is actually an insert operation. So, let us see how to do this insert in a recursive way. So, I have a list L and I want to insert a value v from the right.

So, L is sorted and I am going to do exactly what I did before, I am going to insert it from the right recursively. So again, the base case is that the list is empty, I do not have anything to insert into, then the new list I get is just a list consisting of the new value v, I am inserting it. So, I have the old list plus the value v in the correct position, the old list is empty. So, the new list is just the value V in the correct position, which is the single.

Now if the value v is actually bigger than everything in the list, then I just take it at the end. So, L minus 1, remember that L minus 1 is the last value. So, if v is not smaller than the last value, in other words, is bigger than equal to the last value, then I just take L and I append value v to it and I return that list.

Otherwise, I will move in one step. So, I will recursively, this is where the recursion comes; I will insert v into the slice excluding the last element. So, I will try to insert it one step before, but I must remember to restore the last value afterwards. So I am basically bypassing so I have this. And I have checked that we must go before this.

So, I am trying to now insert it in this segment, and then add back this, that is this value. So, this is a recursive insert and once I have a recursive insert, then insertion sort is very simple. So, I take the list, if it is empty, I return otherwise, I take the last value in the list and inserted into the recursively sorted list up to the n minus 1 value. So, this is a recursive Insertion Sort.

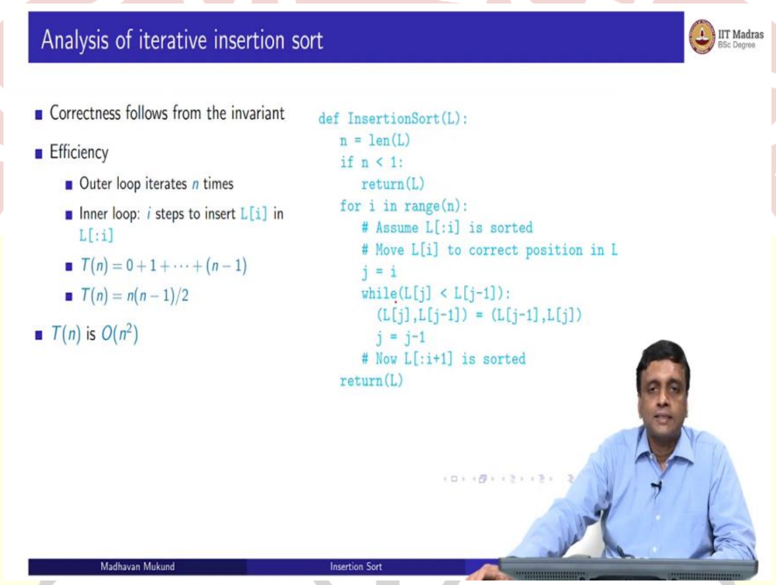
Now there is another point to note between these two versions. So, if you look at the iterative Insertion Sort like the selection sought, it is actually maintaining L and updating it in place, because all these operations exchanging Li, Lj, Lj-1, all these things do not disturb the identity of L.

So, though we are returning L. Actually L is also being updated inside the function. So this is actually sorting l as you go along. Whereas when you are doing a recursive thing, typically you are constructing a new list. So, each time you insert, you are constructing a new list. So actually, this function will not disturb it.

You will get actually a new list back and the old list does not change. Now there is this. If you look at Python, actually, there are two ways to sort a list built in there is something called L dot sort and there is a function called sorted, which takes a list. So this is an in place sort or that is it will actually take L and update it to a sorted version of itself.

But sometimes you do not want that sometimes you want to keep L but you want a sorted version of its values. So, then you should do this. So, then you will have to say L 2 is equal to sorted of L. So, L will not change but L 2 will be the sorted version. So, it is useful to have both versions. But in this case, just to highlight the recursive version of Insertion Sort will create a new list whereas the iterative version will actually update the list in place.

(Refer Slide Time: 10:01)



Analysis of iterative insertion sort

- Correctness follows from the invariant
- Efficiency
  - Outer loop iterates n times
  - Inner loop: i steps to insert L[i] in L[:i]
  - $T(n) = 0 + 1 + \cdots + (n-1)$
  - $T(n) = n(n-1)/2$
- $T(n)$ is $O(n^2)$

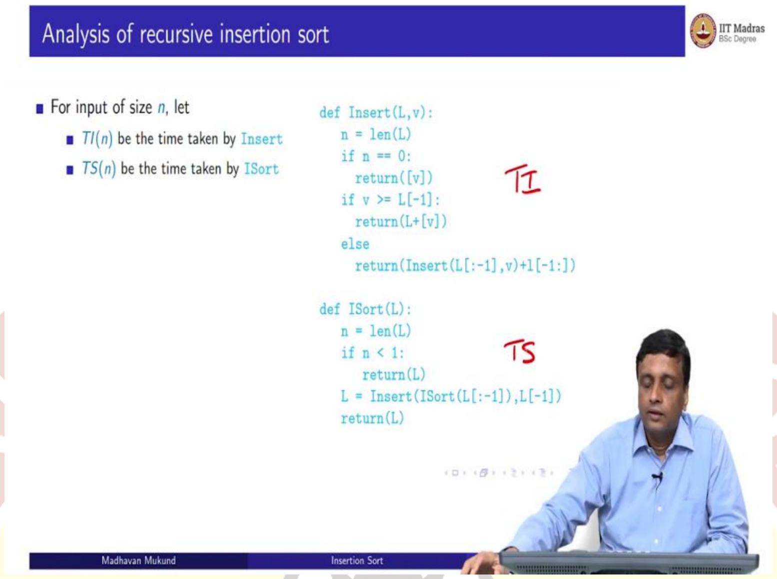
```
def InsertionSort(L):
  n = len(L)
  if n < 1:
    return(L)
  for i in range(n):
    # Assume L[:i] is sorted
    # Move L[i] to correct position in L
    j = i
    while(L[j] < L[j-1]):
      (L[j],L[j-1]) = (L[j-1],L[j])
      j = j-1
    # Now L[:i+1] is sorted
  return(L)
```

So, let us analyze these two separately so, analyses of the iterative version. So, the correctness follows from the invariant just as for selection sort. And if we look at the efficiency, well, the outer loop again, we have this outer loop, which executes n times and inside we have an inner loop and this inner loop is really this insert.

So, we have to take L of i and insert it into 0 to i -1. So, 0 to i -1 has i values. So, I have to look at each of these values in the worst case, because it might be smaller than L0. So the, it takes i steps to insert L of i into the prefix 0 to i - 1. So therefore, by the same logic, as the selection sought, the first time, I have to do no insert, so it is 0, the first element is just really in place, the second element has to be inserted with respect to the first element, the third has been inserted, the first two and so on, so is 0 plus 1 plus 2 up to n minus 1.

And using that same summation rule as we had before, because now it is from 0 to n minus 1, the summation is n into n minus 1 by 2, not n into n plus 1 by 2, but still, it is big O of n squared. So, that is how we can analyze insertions are the same complexity, essentially a selection.

(Refer Slide Time: 11:15)



Analysis of recursive insertion sort

- For input of size n, let
  - $TI(n)$ be the time taken by Insert
  - $TS(n)$ be the time taken by ISort

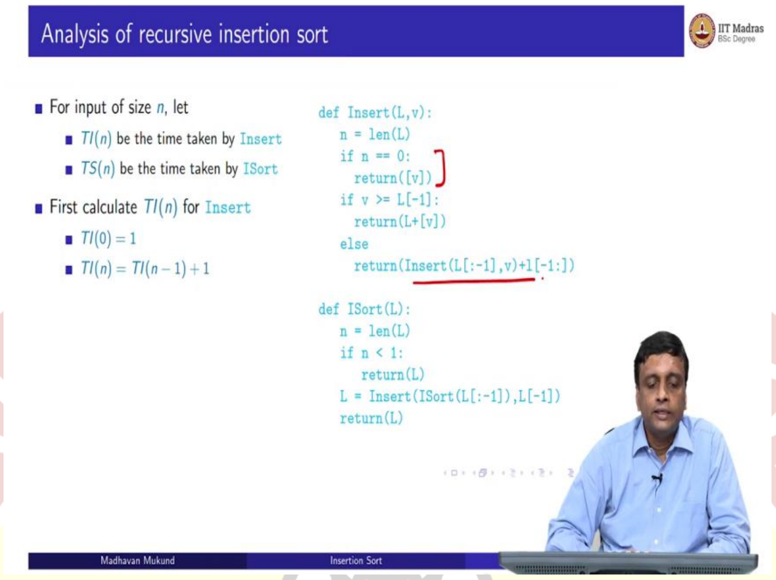
```
def Insert(L,v):
  n = len(L)
  if n == 0:
    return([v])
  if v >= L[-1]:
    return(L+[v])
  else
    return(Insert(L[:-1],v)+l[-1:])

def ISort(L):
  n = len(L)
  if n < 1:
     return(L)
  L = Insert(ISort(L[:-1]),L[-1])
  return(L)
```

So, in the case of the recursive version, so first of all, I am I am glossing over the correctness, but you have to argue that the recursive formulation is kind of correct, which is much easier to do usually than iterate, because just by looking at the structure of the function, you can claim that the recursive version is correct.

But now, if I want to do an analysis, I have to analyze two functions, I have to take the time here, which I am going to call T of i and I am going to take the time here, which is called T of S, TS. So, Ti of n is a time taken by insert for input of size n, TS of n is the time taken by insertion sought for input of size n and Insertion Sort calls insert.

(Refer Slide Time: 11:55)



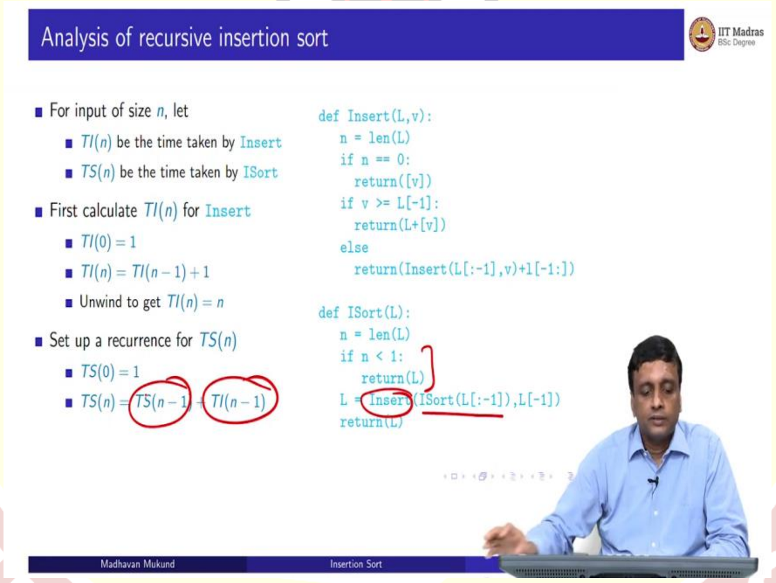
Analysis of recursive insertion sort

- For input of size n, let
  - $TI(n)$ be the time taken by Insert
  - $TS(n)$ be the time taken by ISort
- First calculate $TI(n)$ for Insert
  - $TI(0) = 1$
  - $TI(n) = TI(n-1) + 1$

```
def Insert(L,v):
  n = len(L)
  if n == 0:
    return([v])
  if v >= L[-1]:
    return(L+[v])
  else
    return(Insert(L[:-1],v)+L[-1:])

def ISort(L):
  n = len(L)
  if n < 1:
     return(L)
  L = Insert(ISort(L[:-1]),L[-1])
  return(L)
```

Madhavan Mukund Insertion Sort



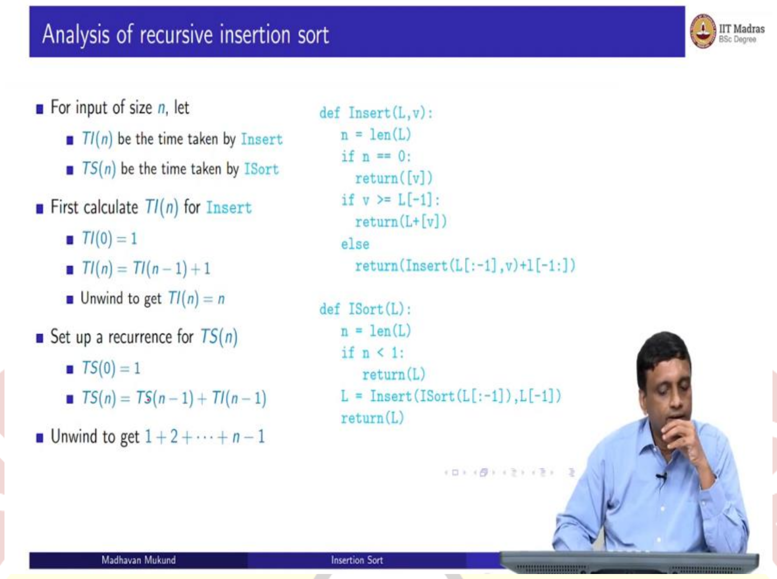
Analysis of recursive insertion sort

- For input of size n, let
  - $TI(n)$ be the time taken by Insert
  - $TS(n)$ be the time taken by ISort
- First calculate $TI(n)$ for Insert
  - $TI(0) = 1$
  - $TI(n) = TI(n-1) + 1$
  - Unwind to get $TI(n) = n$
- Set up a recurrence for $TS(n)$
  - $TS(0) = 1$
  - $TS(n) = TS(n-1) + TI(n-1)$

```
def Insert(L,v):
  n = len(L)
  if n == 0:
    return([v])
  if v >= L[-1]:
    return(L+[v])
  else
    return(Insert(L[:-1],v)+L[-1:])

def ISort(L):
  n = len(L)
  if n < 1:
     return(L)
  L = Insert(ISort(L[:-1]),L[-1])
  return(L)
```

Madhavan Mukund Insertion Sort


Analysis of recursive insertion sort

- For input size n, let
  - $TI(n)$ be the time taken by Insert
  - $TS(n)$ be the time taken by ISort
- First calculate $TI(n)$ for Insert
  - $TI(0) = 1$
  - $TI(n) = TI(n-1) + 1$
  - Unwind to get $TI(n) = n$
- Set up a recurrence for $TS(n)$
  - $TS(0) = 1$
  - $TS(n) = T\$(n-1) + TI(n-1)$
- Unwind to get $1 + 2 + \cdots + n - 1$

```
def Insert(L,v):
  n = len(L)
  if n == 0:
    return([v])
  if v >= L[-1]:
    return(L+[v])
  else
    return(Insert(L[:-1],v)+L[-1:])

def ISort(L):
  n = len(L)
  if n < 1:
    return(L)
  L = Insert(ISort(L[:-1]),L[-1])
  return(L)
```

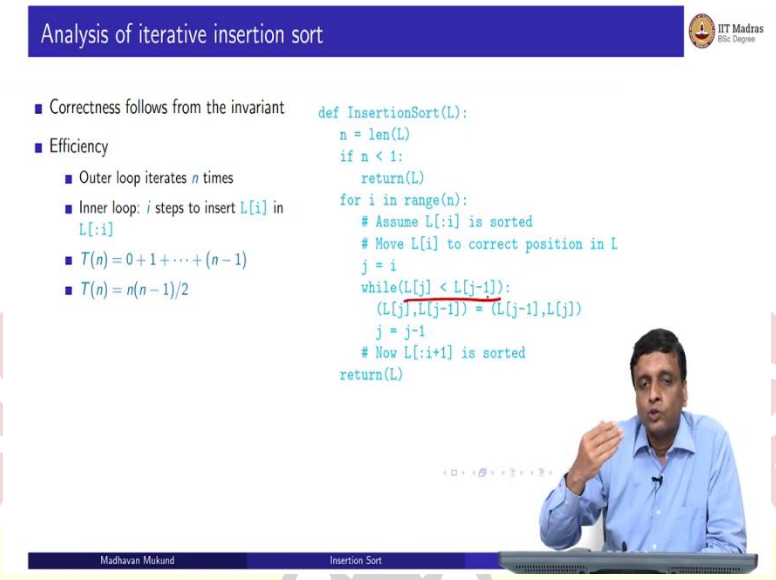
Madhavan Mukund　Insertion Sort

So, let us look at insert first. So again, if it is 0, it immediately returns the list containing only the new value of v. So, T of Ti of 0 is one and Ti of n in the worst case is going to have to call itself with n minus 1 values, and then do some work to append back the value that I skipped over. So, T i of n is going to be Ti of of n minus 1 plus 1 and if you unwind this, you will get 1 plus 1 plus 1 n times.

It is not very difficult; you can just check for yourself that it goes down and get you get 1 plus 1 plus 1 n times. So, Ti of n nd this is the intuitive thing that we said earlier. Also to insert in a list of i elements, we said you need i steps. So, this is saying to insert in a list of n elements, you need n steps.

What about T S? Well, TS again, when it is 0, you return immediately. Otherwise, you first insert and you sort, you have to do both. So, you have to sort an n minus 1 elements and you have to insert an n minus 1 elements. And so, you can unwind this and you can figure out that this thing is actually going to be 1 plus 2 plus n minus 1 and so this is going to give me n squared.

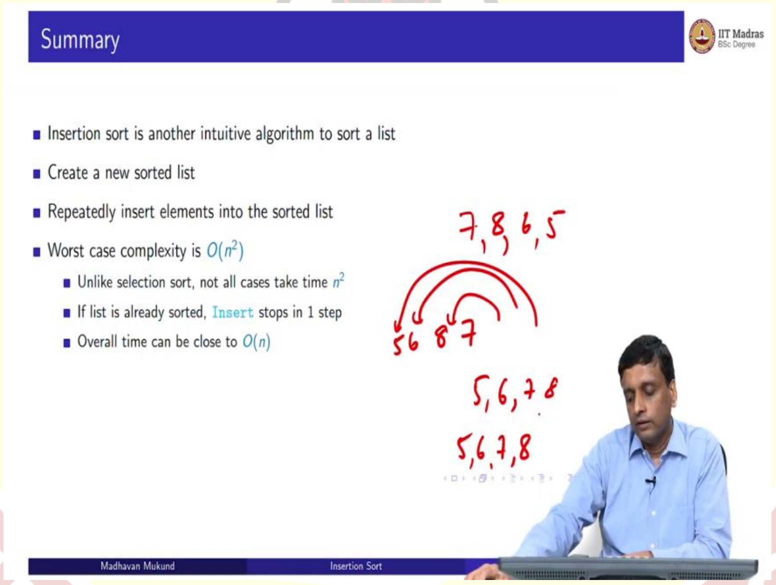
(Refer Slide Time: 13:13)


Analysis of iterative insertion sort

- Correctness follows from the invariant
- Efficiency
  - Outer loop iterates n times
  - Inner loop: i steps to insert L[i] in L[:i]
  - $T(n) = 0 + 1 + \cdots + (n-1)$
  - $T(n) = n(n-1)/2$

```
def InsertionSort(L):
  n = len(L)
  if n < 1:
    return(L)
  for i in range(n):
    # Assume L[:i] is sorted
    # Move L[i] to correct position in L
    j = i
    while(L[j] < L[j-1]):
      (L[j],L[j-1]) = (L[j-1],L[j])
      j = j-1
    # Now L[:i+1] is sorted
  return(L)
```

Madhavan Mukund Insertion Sort


Summary

- Insertion sort is another intuitive algorithm to sort a list
- Create a new sorted list
- Repeatedly insert elements into the sorted list
- Worst case complexity is $O(n^2)$
  - Unlike selection sort, not all cases take time n^2
  - If list is already sorted, Insert stops in 1 step
  - Overall time can be close to $O(n)$

Madhavan Mukund Insertion Sort

So, to summarize insertion sort and selection, sort of both intuitive algorithms and these are both algorithms that we use quite naturally. For instance, insertion sort is typically the kind of algorithm you use, if you if you ever play cards, you have to pick up a card and put it into the correct place in the hand that you have, usually use Insertion Sort, it is already sorted, you find the correct place to put it in.

So here, in both cases, you create a new sorted list. But in this case, unlike the earlier one, where you actually the sorted list was automatically built up in sorted order here, you have to correctly insert it in the correct place. So, the worst case complexity is order n squared. But unlike selection sort, not all cases take time n squared.

Because if you look at this insert operation, especially in the iterative cases, very clear. If you look at the iterative insert operation, notice that if this element is already bigger than the previous one, then I will not go into that loop at all. So essentially, for each element, if it is already in ascending order, the next element will be bigger than the previous element.

So, I will not do this while at all. So, you could actually get almost a linear time algorithm out of this depending on how the input is originally. So, Insertion Sort, performance can be wildly different depending on whether you have an already sorted input, which is a good case. But it has to be sorted in the way if it is ordered in the opposite way.

Supposing you are trying to sort in ascending order and it is given to you in descending order, then you are going to spend a lot of time because it needs time, you are going to have to kind of move it back to the beginning. So, supposing I have 7, 8, 6 as my lists, then I will put 7 then I will have to bring 8 in here.

Then I will bring 6 in here, and I will bring 5 and 8, so then I am going to do a lot of work, but if it was 5, 6, 7, 8, Then I am going to put 5 then 6 is going to come after that, then 7 is going to come after that 8 is going to come after that. So, depending on the order, it may be much more efficient than the worst case of n squared.