

# IIT Madras

ONLINE DEGREE

**Computational Thinking**  
**Professor Madhavan Mukund**  
**Department of Computer Science**  
**Chennai mathematical Institute**  
**Indian Institute of Technology, Madras**  
**Pseudocode for recursion**


So, let us look at this concept of Recursion where the procedure or a function is defined in terms of a simpler version of itself.

(Refer Slide Time: 00:23)

**Inductive definitions**

- Many computations are naturally defined inductively
  - Base case: directly return the value
  - Inductive step: compute value in terms of smaller arguments
- Factorial
  - $n! = n \times (n-1) \times \cdots \times 2 \times 1$   $n-1!$
  - $0!$  is defined to be 1
  - `factorial(0) = 1`
  - For  $n > 0$ , `factorial(n) = n * factorial(n-1)`

Pseudocode: Recursion 2/6


 **IIT Madras**  
ONLINE DEGREE

**Inductive definitions**

- Many computations are naturally defined inductively
  - Base case: directly return the value
  - Inductive step: compute value in terms of smaller arguments
- Factorial
  - $n! = n \times (n-1) \times \cdots \times 2 \times 1$
  - $0!$  is defined to be 1
  - `factorial(0) = 1`
  - For  $n > 0$ , `factorial(n) = n * factorial(n-1)`

```
Procedure Factorial(n)
  if (n == 0) {
    return(1)
  }
  else {
    return(n * factorial(n-1))
  }
End Factorial
```

Pseudocode: Recursion 2/6

 **IIT Madras**  
ONLINE DEGREE

## Inductive definitions

- Many computations are naturally defined inductively

- Base case: directly return the value
- Inductive step: compute value in terms of smaller arguments

### Factorial

- $n! = n \times (n-1) \times \dots \times 2 \times 1$
- $0!$  is defined to be 1
- $\text{factorial}(0) = 1$
- For  $n > 0$ ,  $\text{factorial}(n) = n * \text{factorial}(n-1)$

```
Procedure Factorial(n)
  if (n == 0) {
    return(1)
  } else {
    return(n * factorial(n-1))
  }
End Factorial
```

### Recursive procedure

- $\text{factorial}(n)$  is suspended till  $\text{factorial}(n-1)$  returns a value

factorial(-)

-1 factorial(-2)



So, recursion is naturally associated with what are called inductive definitions. So, in many cases the computation that we want to perform is defined inductively in terms of a base case a simple case where we know the answer and an inductive step where we compute the value in terms of smaller arguments.

So, most common example of this is the factorial function. So, remember that  $n$  factorial is  $n$  multiplied by  $n$  minus 1. So,  $n$  multiplied by all the numbers smaller than  $n$ , so  $n$  into  $n$  minus 1 so 6 factorial be 6 into 5 into 4 into 3 into 2 into 1. Now, mathematically 0 factorial is defined to be 1. So, one way of interpreting factorial is it is a number of ways of arranging  $n$  things. So, you have  $n$  choices for the first position  $n$  minus 1 choices for the second position and so on.

So, the number of different ways in which you can say supposing you have a sequence in which you have some  $n$  students in the class and you want to line them up. So, how many different sequences of students how many different lines up line up of students can you make well you can choose any one of the  $n$  students to be the first position of the remaining  $n$  minus 1 anyone can be in the second position.

So,  $n$  into  $n$  minus 1 then any  $n$  minus 2 in the third position and so on so  $n$  factorial represents this number the number of ways in which you can rearrange a set of  $n$  elements as a sequence. So, 0 is defined to be 0 factorial is defined to be 1. So, if I have nothing I can in principle rearrange it only one way by not having a having an empty set.

So, in this case 0's are base case so we can define factorial of 0 to be 1 and now comes the inductive part factorial of  $n$  we can write it explicitly as  $n$  into  $n$  minus 1 and  $n$  minus 2 and so on but if we look at this remaining portion here this is clearly  $n$  minus 1 factorial. So, this is the same function applied to a smaller number. So, if we knew how to compute  $n$  minus 1 factorial then we could compute it and just multiplied by  $n$  so we can have this inductive thing which has that factorial of  $n$  is  $n$  times factorial of  $n$  minus 1.

So, this is the heart of an inductive definition so we can translate this inductive definition to a procedure which uses this inductive structure. So, here is a procedure factorial which uses this inductive structure. So, it takes the argument  $n$  for which you want to compute the factorial and it says well okay if this is 0 then this is the base case and I return the value 1 because 0 factorial is 1 and if it is not 0 then what I do is I compute factorial for  $n$  minus 1 and then I multiplied by the current number  $n$  and that is my answers.

So, I return  $n$  into factorial of  $n$  minus 1. So, this is what is called a recursive procedure factorial is calling itself but with a different value and the point is that that value is decreasing and because we are talking about natural numbers, they will keep decreasing until it hit 0 and once I hit 0, I have a base case for which I get an explicit answer. In terms of computation what is happening is that when I come to this point say factorial if I say I am trying to do factorial of 3.

So, then it will come here and say I need factorial of 2. So, at this point factorial of 3, the computation says it is 3 times something and that something is factorial of 2, so, I cannot compute factorial of 3 until I know what is factorial of 2, so I will enter back and create start this procedure again with 2.

So, it will come down and say it is 2 times something and that something is factorial of 1. So, it will go back and start again with 1. So, the factorial 3 was suspended because I was trying to find factorial of 2, factorial 2 got suspended because I am trying to find factorial of 1, factorial of 1 will get suspended because I will need factorial of 0, but then magically when I asked for factorial of 0 the function will directly return back without suspending itself again.

So, factorial of 0 will come back as 1 then factorial of 1 will say okay so 1 times factorial 0 so is 1 times 1 so factorial 1 will return 1 factorial 2 will now get the answer back from factorial of 1

and it will say okay now factorial of 2 is 2 times 1 so it will return 2 and finally factorial of 3 will say factorial of 2 is 2 then factorial 3 is 3 times 2.

So, we have this notion of suspending and then resuming once we get the answer back, but of course for this answer to come back we must stop somewhere. So, just to understand what is happening supposing I say factorial of minus 1 in this particular example, then what will happen is factorial of minus 1. It is say  $n$  is not 0 so then I must take minus 1 times factorial of minus 2. So, factorial of 2 minus 2 will come and say minus 2 is not 0 so I must do minus 2 times factorial of minus 3, but now you can see that this is not going to terminate because minus 3 will call minus 4 minus 4 call minus 5 but in the negative direction if I keep going I can keep on going, there is no smallest negative integer.

So, this particular function procedure for factorial works only if  $n$  is positive so you have to be very careful when you write the recursive definitions to make sure that when you call it with an argument the base case will actually be reached in all circumstances. So, you will never get stuck in this endless computation which never terminates. So, this makes sense for numbers and there are other functions that you can think of with numbers which are inductively defined. But this kind of recursive approach to computing is not restricted to numbers.

(Refer Slide Time: 06:15)

Inductive definitions on lists

- Inductive functions on lists
  - Base case: Empty list
  - Inductive step: Compute value in terms first element and rest
- Sum of numbers in a list
  - If  $l == []$ , sum is 0
  - Otherwise, add first(l) to sum of rest(l) list

Pseudocode: Recursion 3/6

IIT Madras  
ONLINE DEGREE

## Inductive definitions on lists

- Inductive functions on lists
  - Base case: Empty list
  - Inductive step: Compute value in terms first element and rest
- Sum of numbers in a list
  - If  $l == []$ , sum is 0
  - Otherwise, add  $\text{first}(l)$  to sum of  $\text{rest}(l)$

```
Procedure Listsum(l)
  if (l == []) {
    return(0)
  }
  else {
    return(first(l) +
           Listsum(rest(l)))
  }
End Listsum
```



Pseudocode: Recursion

3/6



## Inductive definitions on lists

- Inductive functions on lists
  - Base case: Empty list
  - Inductive step: Compute value in terms first element and rest
- Sum of numbers in a list
  - If  $l == []$ , sum is 0
  - Otherwise, add  $\text{first}(l)$  to sum of  $\text{rest}(l)$
  - Can also add  $\text{last}(l)$  to sum of  $\text{init}(l)$

```
Procedure Listsum2(l)
  if (l == []) {
    return(0)
  }
  else {
    return(last(l) +
           Listsum2(init(l)))
  }
End Listsum2
```



Pseudocode: Recursion

3/6



So, here is another example that we can do we can do inductive definitions on lists. So, what is the corresponding thing to 0 for a list? So, 0 is a kind of value which is nothing for a number and the corresponding nothing for a list is an empty list. So, supposing we have a function which we know how to compute for the empty list and then for the inductive step we decompose the list into the first element and the rest. So, the rest of the list is a smaller list. So, the length of that list is decreasing so eventually when I keep doing this the length of that list will become 0 and I will have be able to compute the base case for the empty list. So, that is the idea.

So, here is an example supposing I want to have a list of numbers and I want to add up all the elements in the list. So, of course we can do a for each and just keep incrementing sum as we go



along but the point is to show that this can be done inductively and therefore implemented recursively.

So, the observation is that if I have no numbers then the sum is 0 so if the length of the list is 0 or if the list is empty list then the sum is 0. If there are numbers in the list then what I can do is take the first number in that list, which if you remember we can get by this first of l and I can add it to the result of applying the same procedure to the rest of the list. So, remember that when I take first of l, I get the first number in the list and the rest of l is everything else. So, if I 1, 2, 3, 4, 5 then first of l will be 1 and the rest of l will be the list 2, 3, 4, 5 so this is a list and this is a value.

So, it takes the number at the beginning of the list and adds it to the rest of the list. So, here is corresponding recursive procedure for this. So, it takes list sum which takes l as input, it checks if l is the empty list and if so it returns the sum as 0 and if it is not the empty list then you pick the first element and add it to the result of computing the list sum for the rest of the elements and return that value. So, notice one nice thing here is we do not have this intermediate value sum, which I have to start with initializing into 0 and then, so that happens implicitly through this recursive call.

So I have eliminated this one temporary name called sum which I have to start with 0 and then add up as I go through and so on but more important than that this although this is a very simple function this sometimes gives you a nice insight into the structure of the function. So, of course, we have two ways of decomposing a list. So, the first way we have you seen is to take the first element and the rest but we also have a function which we have assumed which case the last element and the remaining part is called the initialization or the in it.

So, we can also write this other version of this list thing which says if l is empty as before return 0 otherwise, you take the last element and add it to what happens by adding up the beginning of the list which is in it of l. So, we are both equivalent ways of writing an inductive summation function for a sum of a list of numbers.


(Refer Slide Time: 09: 30)

### Insertion sort

- Build up a sorted prefix
- Extend the sorted prefix by inserting the next element in the correct position

```
Procedure InsertionSort(l)
  sortedList = []
  foreach z in l {
    sortedList = SortedListInsert(sortedList, z)
  }
  return(sortedList)
End InsertionSort
```

Pseudocode: Recursion 4/6




### Insertion sort

- Build up a sorted prefix
- Extend the sorted prefix by inserting the next element in the correct position

```
Procedure InsertionSort(l)
  sortedList = []
  foreach z in l {
    sortedList = SortedListInsert(sortedList, z)
  }
  return(sortedList)
End InsertionSort
```

```
Procedure SortedListInsert(l, x)
  newList = []
  inserted = False
  foreach z in l {
    if (not(inserted)) {
      if (x < z) {
        newList = newList ++ [x]
        inserted = True
      }
    }
    newList = newList ++ [z]
  }
  if (not(inserted)) {
    newList = newList ++ [x]
  }
  return(newList)
End SortedListInsert
```

Pseudocode: Recursion 4/6



So, let us look at little more interesting example so we have seen this procedure to sort a list using insertion sort. So, what we did in this insertion sort is that we imagine that we had a list and we picked out the first element of the list and said okay, that is a sorted list. Then we take the second element of the list and say I want to insert it in the correct position with respect to the first element. So, now I have a sorted list of length 2 then I take the third element and I insert it into the correct position.



So I have sorted length of list 3. So, I keep building up the sorted prefixes. So, a prefix of length 1, prefix of length 2 so at a prefix of length  $k$ , then I take the  $k$  plus 1th element in my original list and insert it.

So, the procedure that we had was this one so we started with an empty. So, we are given an input list  $l$ . We start with an empty sorted list and now for each  $z$  in  $l$  that is I am going through my list from beginning to end, I inserted  $z$  into the sorted list. So, initially it was empty. So, the first element just goes there, the second element will go either before and after depending on whether it is to be smaller or bigger and so on and finally I return the list that I construct and of course this whole thing relies on this procedure called sorted insert which we wrote here with said that basically it will walk through the list and find the first position where it needs to be inserted.

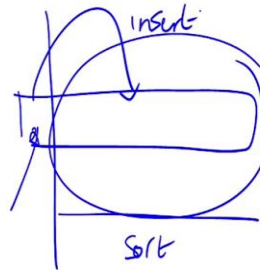
So we have already seen this procedure before I just wanted to remind you what it looked like but we do not have to repeat it now. So, this just takes a sorted list  $l$  and it takes a new value  $x$  which must be inserted and finds the correct place to insert it and returns a new list with  $x$  inserted in the correct position.

So, this is what we could call an iterative version of insertion sort that is one by one we construct these prefixes and we keep inserting into them. But now we are looking at this inductive notion or recursive notion of functions. So, let us think how we might want to recursively apply insertion sort.

(Refer Slide Time: 11:31)

### Insertion sort, inductively

- List of length 1 or less is sorted
- For longer lists, insert `first(l)` into sorted `rest(l)`



Pseudocode: Recursion

5/6



### Insertion sort, inductively

- List of length 1 or less is sorted
- For longer lists, insert `first(l)` into sorted `rest(l)`

Procedure InsertionSort(l)

```
if (length(l) <= 1) {  
  return(l)  
}  
else {  
  return(  
    SortedListInsert(  
      InsertionSort(rest(l)),  
      first(l)  
    )  
  )  
}
```

InsertionSort

Procedure SortedListInsert(l,x)

```
newList = []  
inserted = false  
foreach z in l {  
  if (not(inserted)) {  
    if (x < z) {  
      newList = newList ++ [x]  
      inserted = True  
    }  
  }  
  newList = newList ++ [z]  
}  
if (not(inserted)) {  
  newList = newList ++ [x]  
}  
return(newList)  
End SortedListInsert
```

Same

End SortedListInsert

Pseudocode: Recursion

5/6



सिद्धिर्भवति कर्मजा

## Insertion sort

- Build up a sorted prefix
- Extend the sorted prefix by inserting the next element in the correct position

```

Procedure InsertionSort(l)
  sortedList = []
  foreach z in l {
    sortedList =
      SortedListInsert(sortedList, z)
  }
  return(sortedList)
End InsertionSort
    
```

```

Procedure SortedListInsert(l, x)
  newList = []
  inserted = False
  foreach z in l {
    if (not(inserted)) {
      if (x < z) {
        newList = newList ++ [x]
        inserted = True
      }
    }
    newList = newList ++ [z]
  }
  if (not(inserted)) {
    newList = newList ++ [x]
  }
  return(newList)
End SortedListInsert
    
```



Pseudocode: Recursion

4 / 6



So, any list which has at most one element either it is empty or it has one element is already sorted and if I have a list which has more than one element. So, I have a list which has more than one element then what I can do is I can pick up the first element then I can sort this. So, this is my inductive thing I sought a list of length  $n$  minus 1 and then I insert the first element into the sorted list.

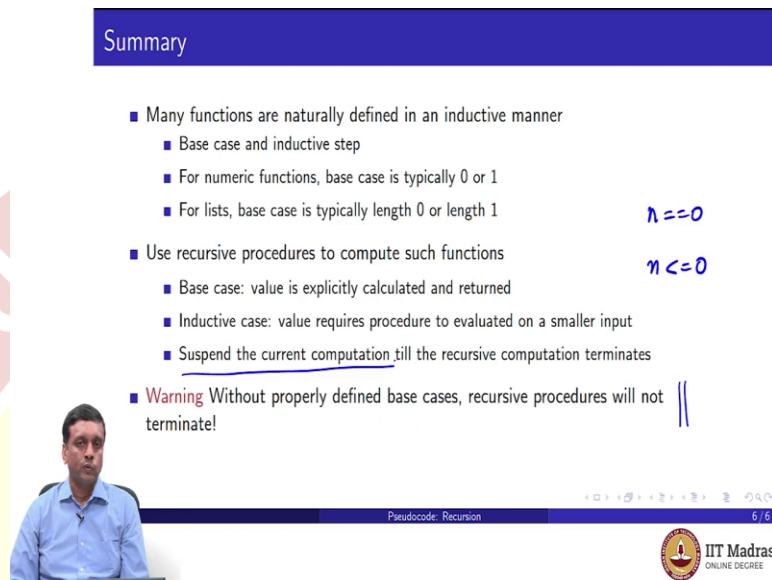
So, here the base cases that I do not need to do this, I only have one element or less and the inductive thing is I have at least two elements then I take the second element onwards recursively sorted and then insert the first element in the correct position. So, I insert first of  $l$  into the sorted version of rest of  $l$ . So, the right hand side is the same, this sorted list insert is the same as before. But what is new is this part. So, what now this says is that if the length of  $l$  is less than or equal to 1 then I just return  $l$  as it is, it is already sorted.

Otherwise I take insertion sort and I apply to rest of  $l$ . So, remember that insertion sort returns me back a sorted version of whatever it gets. So, this particular function will return me back the first the last  $n$  minus 1 elements in sorted order then I take the first element and I apply this function. So, I do a sorted list insert take the first element put it there and this will now be the sorted list as a whole.

So, there is no loop and there is no there is so we do not have for example this this temporary sorted list which we accumulate and so on and so there are many things which are nice and of course this could be done by inserting as before last of  $l$  into the sorted version of in it of  $l$ , just

as we did for sum instead of taking the first element and sorting the rest and putting this into that I can take the last element sort the beginning and put the last into the first so both are equivalent.

(Refer Slide Time: 13:43)



**Summary**

- Many functions are naturally defined in an inductive manner
  - Base case and inductive step
  - For numeric functions, base case is typically 0 or 1
  - For lists, base case is typically length 0 or length 1
- Use recursive procedures to compute such functions
  - Base case: value is explicitly calculated and returned
  - Inductive case: value requires procedure to be evaluated on a smaller input
  - Suspend the current computation till the recursive computation terminates
- **Warning** Without properly defined base cases, recursive procedures will not terminate!

$n == 0$   
 $n <= 0$

Pseudocode: Recursion 6 / 6

IIT Madras  
ONLINE DEGREE

So, what we have seen is that in many cases the function that we want to compute over a set of values can be defined inductively in terms of the base case and the inductive step and typically for numbers if this is a number calculation then the base case is 0 or 1 and if it is a list type of it is a structure then the base case is the empty structure we could also do it for dictionaries, but we have not done it here. We can take out one value of a dictionary do it for the rest and then put this back in for example.

So, you can take empty case and as we saw with the sorting sometimes the base case could be a singleton list, a list which has only one value and then we have the inductive case. The important thing that we have to remember is that you must make sure the base case is reached remember what we did when we did the factorial of a negative number then the base case called missed because you already started smaller than 0.

So, starting smaller than 0 we are going to get even further from 0. So minus 1 takes us to minus 2 take us to minus 3 and so on we never going to get to 0 and the thing will not terminate. So without a properly defined base case or without a clear indication as to what are the legal values that you want to get in that factorial function for example, we could have just changed  $n$  equal to

equal to 0 we could have to change it to  $n \leq 0$  that means see in a sense there is no meaning to saying minus 1 factorial. How do you take minus 1 people and rearrange them.

So, we could just arbitrarily decide that factorial of any negative number like factorial of 0 is 1 and just make that the base case then that will work for any integer. Otherwise you have to be careful to say do not use this function unless you are calling it with positive integers and computationally what happens is that when we hit this recursive procedure call we have to suspend what we are doing now keep all the values as they are and wait for this procedure called to return a value to us.

So it is as though now we suspend this go to another place and start computing this procedure with whatever we have passed it so it will create a new copy of all the internal variables that it needs create that if necessary it may call itself again and so on but because there is a base case is going to eventually come to a call which will return a value and then we come back and we you know through each sequence of these calls we will restore the values and finally we will get the value that we need. So this is how recursive procedures work.

