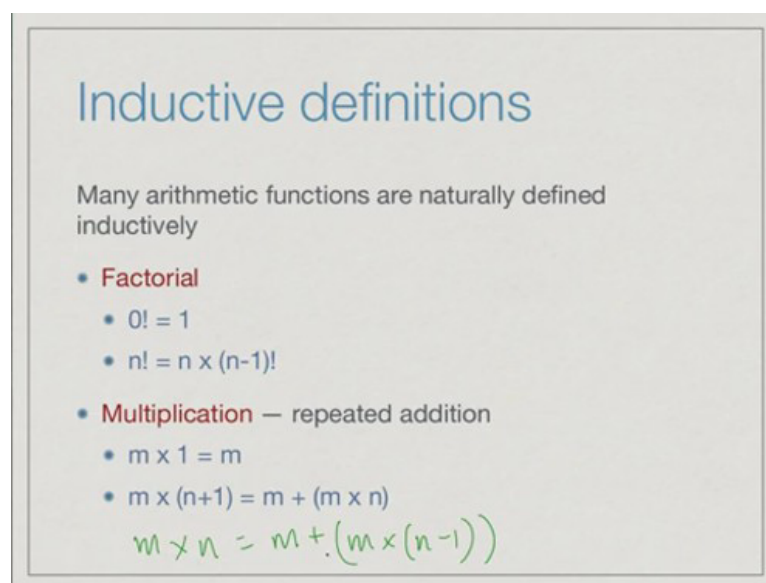


Programming, Data Structures and Algorithms in Python
Prof. Madhavan Mukund
Department of Computer Science and Engineering
Chennai Mathematical Institute, Madras

Week - 03
Lecture - 18
Recursion

For the last lecture of this week, we will look at recursive functions.

(Refer Slide Time: 00:05)



Inductive definitions

Many arithmetic functions are naturally defined inductively

- **Factorial**
 - $0! = 1$
 - $n! = n \times (n-1)!$
- **Multiplication** — repeated addition
 - $m \times 1 = m$
 - $m \times (n+1) = m + (m \times n)$

$m \times n = m + (m \times (n-1))$

Recursive functions are typically based on what we call inductive definitions. So, in arithmetic many functions are naturally defined in an inductive way. Let us explain this by some examples. The first and most common example of a function defined inductively is **the** factorial function. So, we say that zero factorial is 1 and then, we say that n factorial can be obtained from n minus 1 factorial by multiplying by n .

Remember that n factorial is nothing but n into n minus 1 into n minus 2 product all the way down to 1. So, what we are observing is that after n , what appears can be rewritten as n minus 1 factorial. Inductively n minus 1 factorial can be extended to n factorial by multiplying by the value n . So, we can also do this for other functions. You may remember or you may not that multiplication is actually repeated addition when I say m times n , I mean m plus m plus m plus m , n times.

So, how do we define this inductively well we say that m times 1 is just m itself and m times n plus 1 is m plus inductively applying multiplication to n . We could equivalently write this. If you want to be symmetric with the previous case as m times n is m plus m times n minus 1, the same thing. What you are saying is that you can express m times n in terms of m times n minus 1 and then adding it. So, in both these cases what we have is that we have a base case.

(Refer Slide Time: 01:50)

The slide is titled "Inductive definitions ...". It contains two bullet points:

- Define one or more **base** cases
- Inductive step defines $f(n)$ in terms of smaller arguments

 Below the bullet points, the Fibonacci series is written in green: $\text{Fib: } 1, 1, 2, 3, 5, 8$. Below that, the base cases and the inductive step are written in red:

$$\text{Fib}(1) = \text{Fib}(2) = 1$$

$$\text{Fib}(n) = \text{Fib}(n-1) + \text{Fib}(n-2)$$

We have like 0 factorial or m times 1, where the values are given to us explicitly and then, we have an inductive step where f of n is defined in terms of f of n minus 1 and in general, it can be defined in terms of even more smaller arguments. So, one example of that is the Fibonacci series.

If you have seen the Fibonacci series, the Fibonacci series starts with 1 2 3 5 and so on and this is obtained by taking the previous two values and then adding. So, the general rule for Fibonacci is that the first value is equal to the second value is equal to 1 and after the second value Fibonacci of n is Fibonacci of n minus 1 plus Fibonacci of n minus 2. In general a recursive or inductive definition can express the value for n in terms of 1 or smaller values of the function for smaller inputs.

(Refer Slide Time: 02:49)

Recursive computation

- Inductive definitions naturally give rise to recursive programs

```
def factorial(n):  
    if n == 0:  
        return(1)  
    else:  
        return(n * factorial(n-1))
```

$0! = 1$
 $n! = n(n-1)!$

Our interest in inductive definitions is that an inductive definition has a natural representation as a recursive computation. Let us look at factorial. Here is a very simple python implementation of factorial as it is defined, it checks the value n and says that n is 0, then the return 1 otherwise return the value n times the computation recursively of factorial n minus 1.

(Refer Slide Time: 03:33)

Recursive computation

- Inductive definitions naturally give rise to recursive programs

```
def multiply(m,n):  
    if n == 1:  
        return(m)  
    else:  
        return(m + multiply(m,n-1))
```

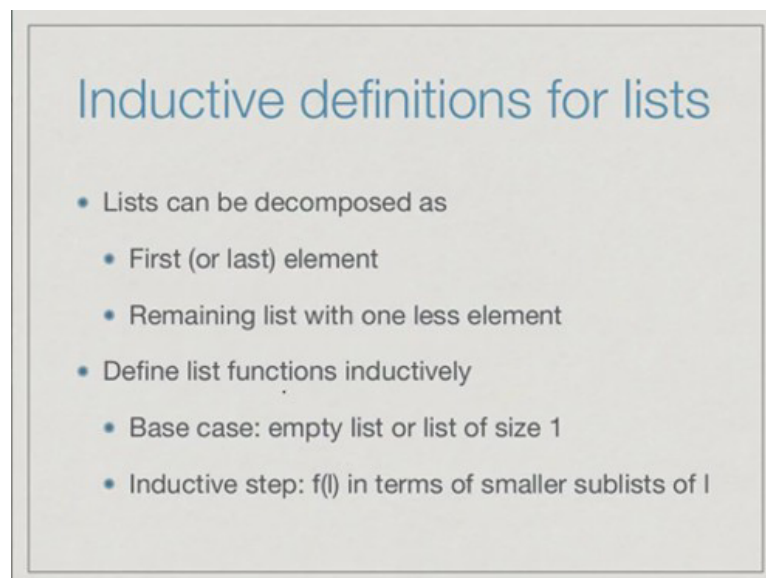
$m \cdot 1 = m$
 $m \cdot n = m + m(n-1)$

This is very clearly what we said before; it says zero factorial is 1 and otherwise if n is not 0, n factorial is n times n minus 1 factorial. So, this is exactly what we wrote before

directly translated as recursive computation. We can say the same for multiplication. You can say if you want to multiply m by n , if n is 1, we return m otherwise we add m to the result of multiplying m by n minus 1.

Again we had written that before as m times 1 is n and m times n is m plus m time n minus 1. If you have an inductive definition, it immediately gives rise to a recursive function which computes same definitions. The advantage is that we do not have to spend much time arguing that this function is correct because it directly reflects the inductive definitions, the mathematical definitions of the function we are trying to compute.

(Refer Slide Time: 04:14)



Inductive definitions for lists

- Lists can be decomposed as
 - First (or last) element
 - Remaining list with one less element
- Define list functions inductively
 - Base case: empty list or list of size 1
 - Inductive step: $f(l)$ in terms of smaller sublists of l

Now, what may be less obvious is that we can do the same thing for structures like list. A list has an inductive structure a list can be thought of as being built up from an empty list by adding one element at a time. So, we can think of decomposing a list, reversing the step. So, we start building a list from an empty list by adding one element at a time say we add them to the left. We add the last element and we add the second last element and so on.

But conversely we can say that given a list we can decompose it by taking say the first element and looking at that first element and the remaining list after removing the first element which has one less element. This gives us our induction we have a smaller structure on which we can try to express a function and then we can combine it with the

first element to get the value for the larger thing. So, we will have a base case where the function is defined either for the empty list or for the simple list of size 1 and in the **inductive** step f of l will be defined in terms of smaller sublists of l .

(Refer Slide Time: 05:15)

The slide is titled "Inductive definitions for lists". It contains a bullet point "• Length of a list" followed by a Python function definition:

```
def length(l):  
    if l == []:  
        return(0)  
    else:  
        return(1 + length(l[1:]))
```

Handwritten in pink on the slide is a diagram of a list `[x, , , ,]`. A vertical arrow points from the first element `x` down to the number `1`. A horizontal line underlines the rest of the list `[1:]`.

Again this is best understood **through** some simple definitions suppose we want to compute the length of the list l . Well the base case if the list is empty it has zero length. If l is equal to `[]` - l is equal to the empty list, we return 0; otherwise what we do is we have a list consisting of a number of values. So, we pull out this first value and we say it contributes one to the length and now inductively we compute the length of the **rest**, right.

We return 1 plus the length of the slice starting at position one. This is an inductive **definition** of length which is translated into a recursive function and once again by just looking at **the** structure of this function, **it** is very obvious that it computes the length correctly because this **is** exactly how you define length **inductively**.

(Refer Slide Time: 06:00)

Inductive definitions for lists

- Sum of a list of numbers

```
def sumlist(l):  
    if l == []:  
        return(0)  
    else:  
        return(l[0] + sumlist(l[1:]))
```

$[x_1, x_2, \dots, x_n]$
 $(x_1) + [x_2, \dots, x_n]$

Now here is another function which does something similar except instead of computing the length, it adds up all the numbers assuming that list is a list of numbers. Again if I have no numbers to add, if I have an empty list, then the sum will be 0 because I have nothing to put into this sum.

On the other hand, if I do have some numbers to add, well the sum consists of taking the first value and adding it to the rest. If I have a list called x_1, x_2 up to x_n , then I am saying that this is x_1 plus the sum of x_2 up to x_n . I can get this sum by this recursive or inductive call. Then, I just add this value to that.

(Refer Slide Time: 06:45)

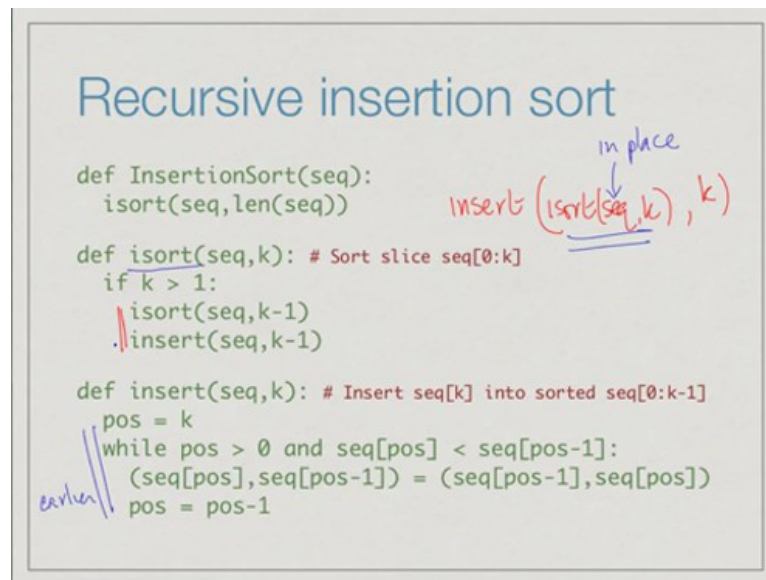
Recursive insertion sort

- Base case: if list has length 1 or 0, return the list
- Inductive step:
 - Inductively sort slice $l[0:\text{len}(l)-1]$
 - Insert $l[\text{len}(l)]$ into this sorted slice
len(l)-1

Insertion sort which we have seen actually has a very nice inductive definition. If we have a list of size 0 or size 1, it is already sorted. We do not have to do anything, so this becomes the base case.

On the other hand, if we have a list of length two or more, we inductively sort the slice from the beginning up to, but excluding the last position. This is slice from 0 to length of the list minus 1, then we take the last position and then, this should be minus 1. So, we take the value at the last position and we **insert it** into the sorted slice. We insert the last position into the inductively sorted slice excluding the last position.

(Refer Slide Time: 07:36)



Here is a recursive definition of insertion sort in python. The natural thing in python or in any other thing would be to say that we want to insert the last position into the result of sorting the sequence up to, but excluding the last position, but the way we have written **our** insertion sort; this function does not return a list. It sorts this thing in place. This call would not have the correct type because insert will take a sequence and a position and insert this value at this position to its left. So, we write **it** now as two separate things.

First of all we have an overall insertion sort which takes a sequence and it will call this **auxilliary** function which says: sort this sequence up to this position. So, isort sorts the slice from 0 up to k minus 1. So, what **does** isort say? isort checks if is the base case if k is 0 or k is 1. If I am sorting up to the first position or I am not sorting anything at all, right - then I will just return the sequence. This is telling me sort from 0 to k minus 1.

If it is 0, then I have an empty list. If I have 0 to 1, then I have a list of one position, **it is** only if I **have** 0 to 2 that I have at least two elements. And if so, what I do is I sort k minus 1 **positions** and I insert the last position into this sequence. What does insert do? Insert **does** exactly what we did when we did the regular insertion sort. It **sets** a position variable or name to the last position and walks left and keeps swapping. So long as it has not reached the left hand side h and so long as it finds something to the left which is strictly bigger than the one that you looking at. So, this is what we did earlier.

What is new is this part which is this recursive **call, it** says sort the sequence up to this position, recursively using the same isort **but** change the index from k to k minus 1 and then insert the current value into this sequence.

(Refer Slide Time: 09:51)

```
>>> import sys
>>> sys.setrecursionlimit(10000)
>>> l = list(range(1000,0,-1))
>>> InsertionSort(l)
>>> l = list(range(5000,0,-1))
>>> InsertionSort(l)
>>> []
```

So, as before let us run this in python. Here is the code isort rec dot **py** which contains **this** recursive implementation of insertion sort. So, if I now import this, then as before if we say l is a range of 500 values say, **in** descending order, then if we apply insertion sort to this, then l produces **the** ascending order 1 to 500.

Last time we said that for n squared sort, we should look at larger **values**. Supposing we take range 1000, now if we take range 1000, then something surprising happens. We get an error message from python saying that it could not sort this because it **reached** something called the maximum recursion depth. So, what happens when we make a recursive call is that the current function is suspended. We try to sort something of length 1000. It will try to insert **the 1000th element** into the result of sorting the first 999. So, we suspend the first call and try to sort 999. This in turn will try to insert **the 999th element** into sorting the first 998. So, it will suspend that and try to sort the first 998.

At each time we make a recursive call, the current function is suspended and a new function is started. So, this is called the depth of recursion. How many functions have **we** suspended while we are doing this process? Now unlike some other languages, python imposes a limit on this and the limit as we can see is clearly less **than** 1000 because we

are not able to sort a list of 1000 using this particular mechanism. So, how do we get around this? Well, first of all, let us try and see what this limit is.

We know that we cannot sort 1000, but we could sort 500. So, can we sort 750 for example, it turns out that we can sort 750. Now, it will turn out that, for instance, we can sort 900. So, we can actually find this limit by doing what we did earlier - binary search. We can keep halving although I did not strictly halve after 750. I know it fails at 1000, it does not fail at 750, I should have tried 875, but I will spare you this binary search and I can show you that if I use 997, then it will work, but I use 998, then it will fail. So, somewhere around 998 is the recursion limit that python uses by default.

Now, fortunately there is a way to get around this. So, you can set this recursion limit and the way you do it is the following. You first have to import a bunch of functions which are called system dependent functions by saying import sys and then, in sys there is a function called set recursion limit and we can set this to some value bigger than this say 10000. Now, if we for instance ask it to sort a list of 1000, then it does not give us error. Same way, I could even say 5000 because 5000 will also only create the same kind of limit because it is well under 10000.

Remember that in this we are basically doing recursion exactly the number of times as there are elements because we keep inserting, inserting, inserting and each insertion requires us to recursively sort the things to its left. That is why we get the stack of nested recursions, but the thing to remember is that by default, python has an upper bound on the number of nested recursions which is less than 1000, if you want to change it you can by setting this recursion limit explicitly, but python does not allow you to set it to an unbounded value. You must give it an upper bound. So, you cannot say let recursion run as long as it needs to. You have to have an estimate on how much the recursion will actually take on the inputs you are giving, so that you can set this bound explicitly.

Now, the reason that python does this is because it wants to be able to terminate a recursive computation in case you have made a mistake. A very common mistake with recursive computation, it is a bit like we said for while loops, if we never make the condition false, a while loop will execute forever. Same way if we do not set the base case correctly, then a recursive computation can also go on forever. The way that python

stops this and forces you to go and examine the code is by having a recursion limit saying beyond a certain depth, it will refuse to execute the code.

(Refer Slide Time: 14:51)

Recursion limit in Python

- Python sets a recursion limit of about 1000

```
>>> l = list(range(1000,0,-1))
>>> InsertionSort(l)
...
RecursionError: maximum recursion depth
exceeded in comparison
```

- Can manually raise the limit

```
>>> import sys
>>> sys.setrecursionlimit(10000)
```

So, what we have seen is that we have this recursion limit and we can raise it manually by importing this sys module and setting the set recursion limit function inside sys.

(Refer Slide Time: 15:00)

Recursive insertion sort

- $T(n)$, time to run insertion sort on length n
 - Time $T(n-1)$ to sort slice `seq[0:n-1]`
 - $n-1$ steps to insert `seq[n-1]` in sorted slice
- Recurrence
 - $T(n) = n-1 + T(n-1)$
 $T(1) = 1$
 - $T(n) = n-1 + T(n-1) = n-1 + ((n-2) + T(n-2)) = \dots = (n-1) + (n-2) + \dots + 1 = n(n-1)/2 = O(n^2)$

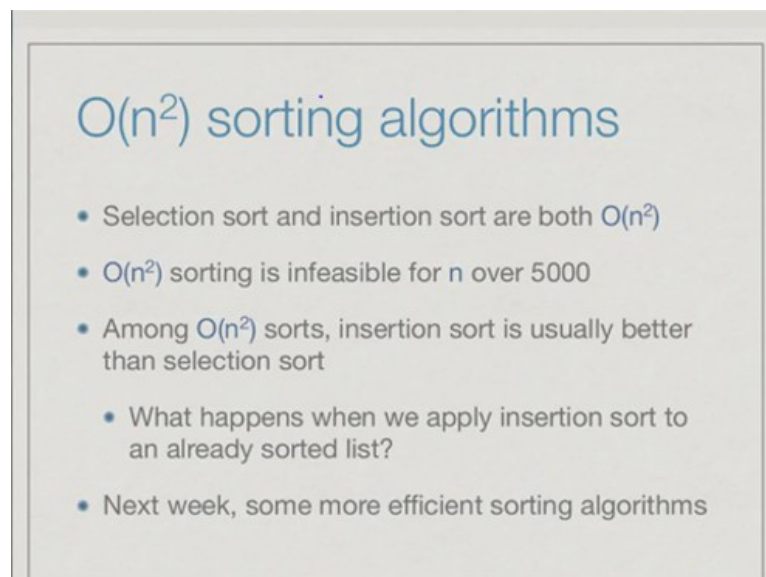
Handwritten notes: $T(n-k)$ and $= T(1)$ are written in red above the recurrence formula.

So how would we analyse the complexity of recursive insertion sort? So as before, we use T of n to denote the time it takes to run insertion sort on an input of length n . Insertion sort at the highest level consists of two steps. We first have to sort the initial

slice of length $n - 1$ and by definition, this will take time T of $n - 1$ and then, we need $n - 1$ steps in the worst case to insert the last position into the sorted slice. This gives rise to what we call a recurrence. We saw this when we were looking at how to analyse binary search which was also a recursive algorithm. We have that T_n in general is $n - 1$ plus T of $n - 1$ and T of 1 when we come to the base case is 1.

As with the binary search, we can solve this by expanding or unwinding the recurrence. So, we have T of n is $n - 1$ plus T of $n - 1$. If we take T of $n - 1$ and apply the same definitions, we get $n - 2$ plus T of $n - 2$. Then, we take T of $n - 2$ and apply the same definition, we get $n - 3$ and T of $n - 3$ and this will keep going on until we get T of $n - k$ is equal to T of 1. In other words when k becomes n after n steps, we will have 1. So, we will have $n - 1$ plus $n - 2$ down to 1 which is the same thing we had for the iterative version of insertion sort - n into $n - 1$ by 2 and this is order n^2 .

(Refer Slide Time: 16:41)



$O(n^2)$ sorting algorithms

- Selection sort and insertion sort are both $O(n^2)$
- $O(n^2)$ sorting is infeasible for n over 5000
- Among $O(n^2)$ sorts, insertion sort is usually better than selection sort
 - What happens when we apply insertion sort to an already sorted list?
- Next week, some more efficient sorting algorithms

We have seen two order n^2 sorting algorithms both of which are very natural intuitive algorithms which we do by hand when we are giving sorting tasks to perform - selection sort and insertion sort. We have also seen that both of these will not work in general if we have large values of n and not even so large; if we have values of n over 5000. But we also saw in the previous two lectures that insertion sort is actually slightly better than selection sort because selection sort, the worst case is always present because

we always have to scan the entire slice in order to find the minimum value element to move into the **correct** position where insertion sort will stop as soon as it finds something which is in the correct order.

So if we have a list which is already sorted, then insertion sort will actually work much better than n^2 , but we cannot rely on this, **and** anyway we **are counting** worst case complexity. So, we have to take the fact that both of these will in general not work **very well for** lists larger than 5000 elements.

What we will see next week is that there are substantially more efficient **sorting** algorithms which **will** allow us to sort much larger **lists than** we can sort using selection sort or insertion sort.