PYTHON FOR COMPUTATIONAL PROBLEM SOLVING

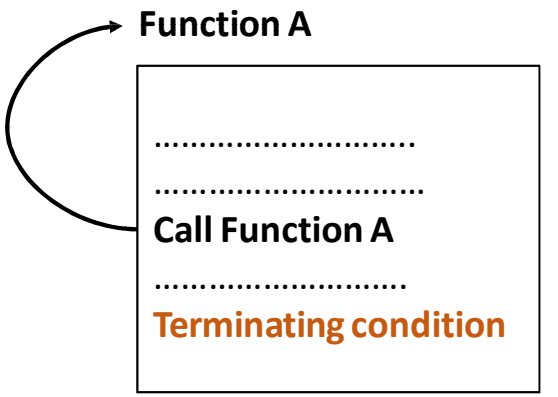
Recursion

Prof. Sindhu R Pai
PCPS Theory Anchor - 2024
Department of Computer Science and Engineering

PES UNIVERSITY

Recursion

- There are many problems for which the solution can be **expressed in terms of the problem itself.**
- The problem is solved by **breaking it down into simpler versions of itself** and working on those, **until you get to a "base case" or stopping point**.
- In programming, this is represented by a **function calling itself** over and over until it solves a particular problem.



Recursive Function Definition

PES UNIVERSITY

Recursion

Every recursive problem can be divided into a base case and a recursive case.

- **Base Case:** This is the simplest subproblem the problem can be reduced to. In a recursive solution, **the base case is the one case you know the direct solution for**.

- **Recursive Case:** Any case that isn't the base case is automatically a recursive case. We **don't know the solutions** for these cases; instead, we **split them further into simpler cases** until we reach the base case.

PES UNIVERSITY

Recursion – Basic Example

Take the example of opening a set of Russian nesting dolls.

Base Case: The doll doesn't open. This means we've reached the smallest doll, and represents the stopping condition for our recursion.

Recursive Case: The doll opens. This means we still haven't finished opening the set, and we need to perform some action here called the recursive step.

Recursive Step: If you have a doll that can open (i.e., recursive case), then open it and see if the doll inside it can open. This is the action we perform on the recursive case to get it closer to the base case.

PES UNIVERSITY

Recursion – Basic Example

We can illustrate this using pseudocode as follows:

```
function OpenDoll(doll):
    # Base Case
    if doll is the smallest doll:
        print("This is the smallest doll. Stop opening.")
    # Recursive Case
    else:
        print("Opening doll to find a smaller one inside.")
        OpenDoll(smaller_doll_inside)   # Recursive Step

# Start the recursive process by opening the largest doll
OpenDoll(largest_doll)
```

PES UNIVERSITY

Need for Recursion

- Used to simplify complex problems into smaller, easier problems
- Solving problems with repeated patterns
- Handling unknown levels of depth (number of steps to a problem)
- Mathematical and algorithmic problems

Criteria for Implementing Recursion

- Recursive functions must have a **base case or stopping condition**
- Every recursive call to the function **must take a step towards reaching the base case**

Example: Computation of factorial of a number

factorial(0)=1
factorial(1)=1*1
factorial(2)=2*1
factorial(3)=3*2*1
factorial(4)=4*3*2*1
.
.
.
factorial(n)=n*(n-1)*(n-2)*……..*1

factorial(n)=n * factorial(n-1)

The complete definition of the factorial function is,

$$\text{factorial}(n) = \begin{cases} 1 & \text{if } n=0 \\ n*\text{factorial}(n-1) & \text{otherwise} \end{cases}$$

PES UNIVERSITY

Implementation of Recursion

- Recursion is implemented using stack because activation records are to be stored in LIFO order (last in first out).
- An activation record of a function call contains arguments, return address and local variables of the function.
- Stack is a linear data structure in which elements are inserted to the top and deleted from the top.

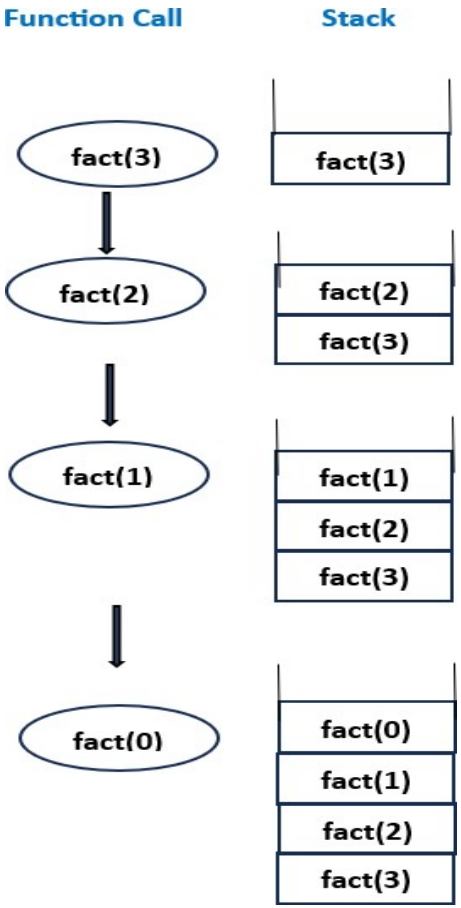
PES UNIVERSITY

Implementation of Recursion

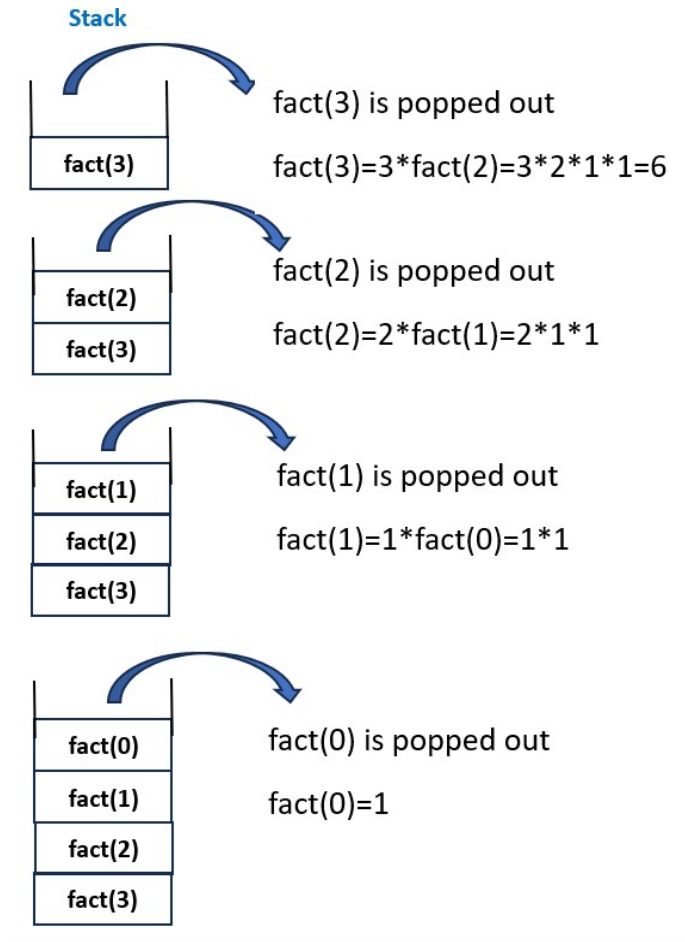
- Consider the example of computing the factorial of a number.

```
def fact(n): #Recursive Function
        if n == 0 : #terminating condition
                res = 1
        else:
                res = n * fact(n - 1)
        return res
```

- Suppose we want to compute **fact(3)**. The above function gets executed using a stack.



- When the function call is made recursively, the activation record for each call will be placed on the top of the stack.
- Initially, fact(3) is called which recursively calls fact(2), fact(1) and fact(0); the activation record for each of these calls gets inserted to top of the stack.



- For n=0 i.e. base case (Termination Condition), the function call stops and fact(0) is popped out from the top of the stack.
- It returns 1, then the recursion backtracks and solves the pending function calls. These are popped out of the stack as they get computed.

PES UNIVERSITY

Stack Memory Allocation

- In Python, function calls and the references are stored in **stack memory.**
- Allocation happens on contiguous blocks of memory – referred as ***Function call Stack***.
- The size of memory to be allocated is known to the compiler and whenever a function is called, its variables get memory allocated on the stack.
- Any local memory assignments such as variable initializations inside the particular functions are stored temporarily on the function call stack, where it is deleted once the function returns.
- This allocation onto a contiguous block of memory is handled by the compiler using predefined routines.

PES UNIVERSITY

Example 1– Compute factorial of a number

```
def fact(n): #Recursive Function
        if n == 0 : #terminating condition
                res = 1
        else:
                res = n * fact(n - 1)
        return res
print(fact(5))
print(fact(0))
```

Output:

```
120
1
```

PES UNIVERSITY

Example 2 – Compute GCD of two numbers

```
def gcd(m, n): #Recursive Function
        if m == n : #terminating condition
                res = m
        elif m > n :
                res = gcd(m -n, n)
        else:
                res = gcd(m, n - m)
        return res
print("GCD : ", gcd(65, 91))
```

Output:

GCD : 13

PES UNIVERSITY

Example 3 – Generate Fibonacci Series upto n terms

```
def fib(n):    #Recursive Function
        if n <= 1:
            #terminatin
            g condition
            return n
        else:
            return(fib(n-1) + fib(n-2))

n_terms=int(input("Enter the number of terms for

Fibonacci Series\n"))

for i in range(n_terms):
        print(fib(i))
```

Output:

```
Enter the number of terms for
Fibonacci Series
 5
0
1
1
2
3
```

PES UNIVERSITY

Example 4 – Solving Tower of Hanoi Puzzle

```
def TowerOfHanoi(n , src, aux, dest): #Recursive Function if
    n==1: #terminating condition
        print ("Move disk 1 from source",src,"to destination",dest)
        return
    TowerOfHanoi(n-1, src, dest, aux)
    print ("Move disk",n,"from source",src,"to destination",dest)
    TowerOfHanoi(n-1, aux, src, dest)

n=int(input("Enter number of disks\n"))
TowerOfHanoi(n,'A','B','C')
```

Output:

```
Enter   number
of disks 3
Move disk 1 from source A to destination C
Move disk 2 from source A to destination B
Move disk 1 from source C to destination B
Move disk 3 from source A to destination C
Move disk 1 from source B to destination A
Move disk 2 from source B to destination C
Move disk 1 from source A to destination C
```

PES UNIVERSITY

Example 5 – To add two numbers using recursion

```
def add(x, y): #Recursive Function
    if(y == 0):    #terminating condition
        return x
    return add(x, y - 1) + 1

print("Sum =", add(10, 20))
```

Output:

Sum = 30

PES UNIVERSITY

Example 6 – To subtract two numbers using recursion

```
def subtract(x, y): #Recursive Function
    if(y == 0):          #terminating condition
        return x
    return subtract(x-1, y-1)

print("Result =", subtract(10, 20))
```

Output:

Result = -10

PES UNIVERSITY

Example 7 – To multiply two numbers using recursion

```
def product(a,b): #Recursive Function
    if(a<b):
        return product(b,a)
    elif(b!=0):
        return(a+product(a,b-1))
    else:                #Stopping point
        return 0

print("Product =",product(10,20))
```

Output:

Product = 200

PES UNIVERSITY

Example 8 – To divide two numbers using recursion

```
def divide(x, y):    #Recursive Function
    if(x < y):            #terminating condition
        return 0
    else:
        return 1 + divide(x - y, y)

print("Result:", divide(20, 5))
```

Output:

Result: 4

PES UNIVERSITY

Recursion vs. Iteration

Recursion	Iteration
Function calls itself	Set of program statements executed repeatedly
Implemented using Function calls	Implemented using Loops
Termination condition is defined within the recursive function	Termination condition is defined in the definition of the loop
Leads to infinite recursion, if does not meet termination condition	Leads to infinite loop, if the condition in the loop never becomes false
It is slower than iteration	It is faster than recursion
Uses more memory than iteration	Uses less memory compared to recursion

Note: When a problem can be solved both recursively and iteratively with similar programming effort, it is generally best to use an iterative approach.

THANK YOU

Department of Computer Science and Engineering

Dr. Shylaja S S, Director, CDSAML & CCBD, PESU
Prof. Sindhu R Pai – sindhurpai@pes.edu
Prof. Sowmya Shree P
Ack: Teaching Assistant – Advaith Sanil Kumar