Department of Computer Science and Engineering
PES University, Bangalore, India

Python For Computational Problem Solving (UE22CS151A)

Mr. Prakash C O
Asst. Professor,
Dept. of CSE, PESU
coprakasha@pes.edu

Recursive Function

There are many problems for which the solution can be expressed in terms of the problem itself. There are some classical examples from Mathematics. Successive differentiation of trigonometric functions like sine or cosine yields the same function itself. There are special functions like Legendre polynomial and Bessel's function which express the relationships using recursion.

A function is said to be a recursive if it calls itself. For example, let's say we have a function fact(n)and in the body of fact(n)there is a call to the fact(n).

A recursive function is a function defined in terms of itself via self-referential expressions. This means that the function will continue to call itself and repeat its behavior until some condition is met to return a result.

All recursive functions share a common structure made up of two parts:

1. base case and
2. recursive case.

To demonstrate this structure, let's write a recursive function for calculating n!:

1. **Decompose the original problem into simpler instances of the same problem**. This is the recursive case:

```
n! = n x (n-1)!
n! = n x (n-1) x (n-2)!
n! = n x (n-1) x (n-2) x (n-3)!
.
.
n! = n x (n-1) x (n-2) x (n-3) .... x 3!
n! = n x (n-1) x (n-2) x (n-3) .... x 3 x 2!
n! = n x (n-1) x (n-2) x (n-3) .... x 3 x 2 x 1!
n! = n x (n-1) x (n-2) x (n-3) .... x 3 x 2 x 1 x 0!
```

2. **As the large problem is broken down into successively less complex ones, those subproblems must eventually become so simple that they can be solved without further subdivision.** This is the base case:

   Here, 0! is our base case, and it equals 1.

Example: Recursive function for calculating n! implemented in Python:

In [3]:
```
def fact(n):
        if n == 0 :
                res = 1
        else:
                res = n * fact(n - 1)
        return res
```

Note:

1. In (recursive) programming, we require two steps.
   - Express solution for a problem in terms of the problem itself, but of a smallest size.
   - Express solution to a base case without recursion.
2. Every recursive function must have a base case/condition that stops the recursion or else the function calls itself infinitely.

Example 1:

We may express factorial of n as n times factorial of (n – 1) if n > 0 and we can express factorial of 0 as 1. Here is such a program.

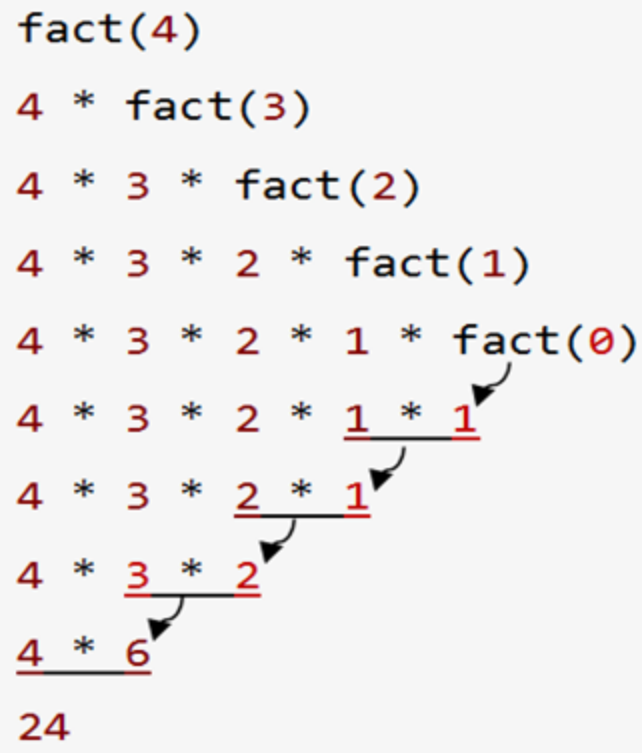
In [4]:
```
# recursion
# factorial:
#       n! = n x (n-1)!  for n > 0
#     n! = 1  for n = 0

def fact(n):
        if n == 0 :
                res = 1
        else:
                res = n * fact(n - 1)
        return res

print(fact(5)) # 120
print(fact(0)) # 1
```

```
120
1
```

Each time the function is called a new activation record is created. Please check how this function works using the tool python tutor.



In [4]:
```
# Program to find nth fibonacci number (with recursion)
def fib(n):
    #print("Calculating F", "(", n, ")", sep="", end=", ")
    # Base case
    if n == 0:
        return 0
    elif n == 1:
        return 1
    # Recursive case
    else:
        return fib(n-1) + fib(n-2)

print('fib(6) =',fib(6))
```

```
fib(6) = 8
```

In [7]:
```
# Program to find nth fibonacci number (without recursion)
def fib(n):
     a, b = 0, 1
     for i in range(2,n) :
         #print(a, end=' ')
         a, b = b, a+b
     print(b)
fib(5)
```

```
3
```

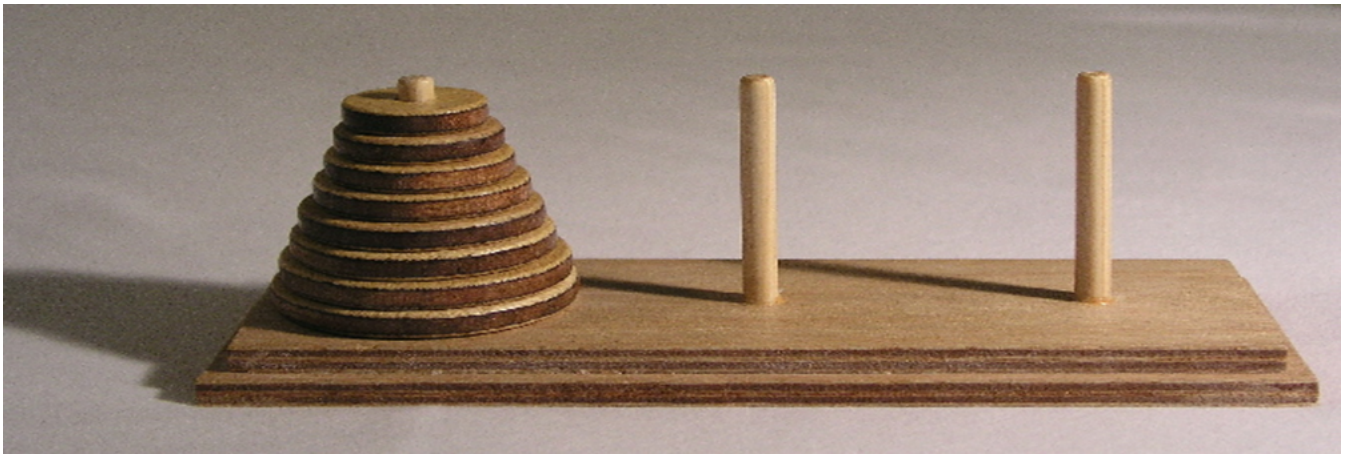
In [10]:
```
# Program to find fibonacci series up to n.
def fib(n):
     a, b = 0, 1
     while a < n:
         print(a, end=' ')
         a, b = b, a+b
     print()
fib(1000)
```

```
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

Recursive Python function to solve tower of hanoi

Rules:

- Only one disk can be moved at a time.
- Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e. a disk can only be moved if it is the uppermost disk on a stack.
- No disk may be placed on top of a smaller disk.



A, B, C are the name of rods/pegs

- A - Source rod/peg
- B - Auxiliary rod/peg
- C - Destination rod/peg

Disk names are numeric, higher the number larger the size

In [11]:
```
def TowerOfHanoi(n , from_rod, to_rod, aux_rod):
    if n > 0:
        TowerOfHanoi(n-1, from_rod, aux_rod, to_rod)
        print("Move disk",n,"from rod",from_rod,"to rod",to_rod)
        TowerOfHanoi(n-1, aux_rod, to_rod, from_rod)

n=3  # Number of disks
TowerOfHanoi(n, 'A', 'C', 'B')
```

```
Move disk 1 from rod A to rod C
Move disk 2 from rod A to rod B
Move disk 1 from rod C to rod B
Move disk 3 from rod A to rod C
Move disk 1 from rod B to rod A
Move disk 2 from rod B to rod C
Move disk 1 from rod A to rod C
```

Advantages of Recursion

- Recursive functions make the code look clean and elegant.
- A complex task can be broken down into simpler sub-problems using recursion.
- Sequence generation is easier with recursion than using some nested iteration.

Disadvantages of Recursion

- Sometimes the logic behind recursion is hard to follow through.
- Recursive calls are expensive (inefficient) as they take up a lot of memory and time.

- Recursive functions are hard to debug.

References:

1. function_recursion_functional_programming.docx – Prof. N S Kumar, Dept. of CSE, PES University.
2. https://www.w3schools.com/python
3. https://docs.python.org/

In [ ]: