PES UNIVERSITY
Department of Computer Science and Engineering (UG Studies)
PES University, Bangalore, India

Introduction to Computing using Python (UE19CS101)

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:

```
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.

```
# name : 1_recursion.py
# 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
```

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

```
fact(4)
4 * fact(3)
4 * 3 * fact(2)
4 * 3 * 2 * fact(1)
4 * 3 * 2 * 1 * fact(0)
4 * 3 * 2 * 1 * 1
4 * 3 * 2 * 1
4 * 3 * 2
4 * 6
24
```

Example 2:

Let us look at another example of recursion – the greatest common divisor. This is supposed to be the oldest algorithm ever known – traced to 300 BC. This is called Euclid's algorithm.

It states as follows.

```
Algorithm gcd(a, b)
if two numbers a and b are equal then
       that itself is the gcd
else if a > b then
     subtract b from a and find gcd of (a – b, b)
else
    subtract a from b and find gcd of (a,  b – a)
```

Here is the equivalent Python code.

Observe that the recursive calls are made two different limbs of the if statement. At runtime, the activation record will remember to which part of the code it should return once the function terminates.

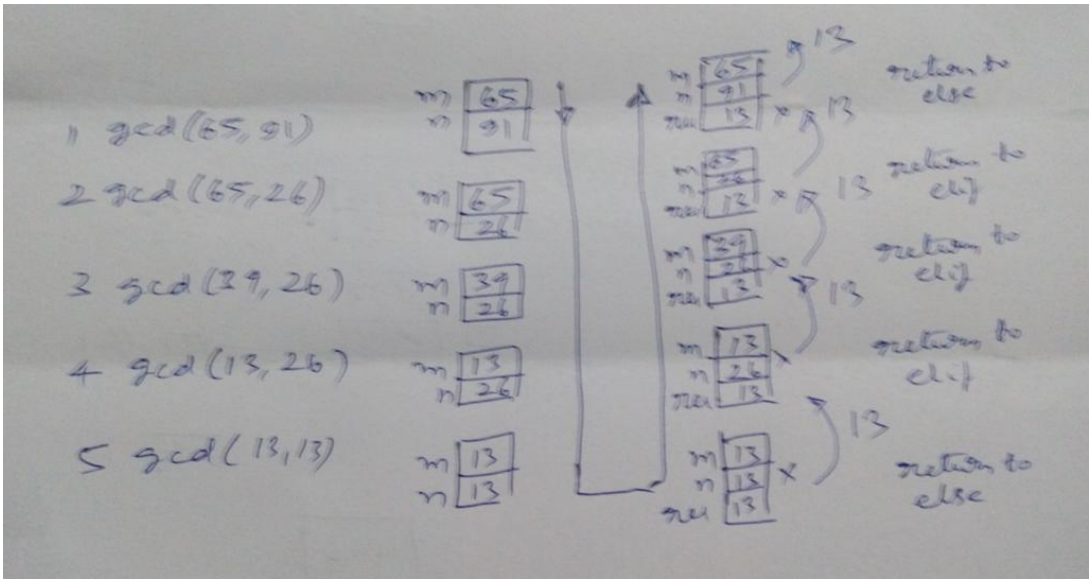
```
# name: 2_recursion.py
# Euclid's algorithm to find the greatest common divisor

def gcd(m, n):
      if m == n :
            res = m
      elif m > n :
            res = gcd(m - n, n)
      else:
            res = gcd(m, n - m)
      return res

print("gcd : ", gcd(65, 91))

# 1. gcd(65, 91)
# 2. gcd(65, 26)
# 3. gcd(39, 26)
# 4, gcd(13, 26)
# 5. gcd(13, 13)
```

```
Output: gcd : 13
```

Try this in the Python tutor and observe that the activation records get created and get destroyed. Observe how the res variable of the function gets populated. Also observe how the cursor jumps back to different recursive calls once the earlier call is terminated.

The following diagram illustrates

- call sequence
- activation record creation, population and destruction
- return value propagation
- return control



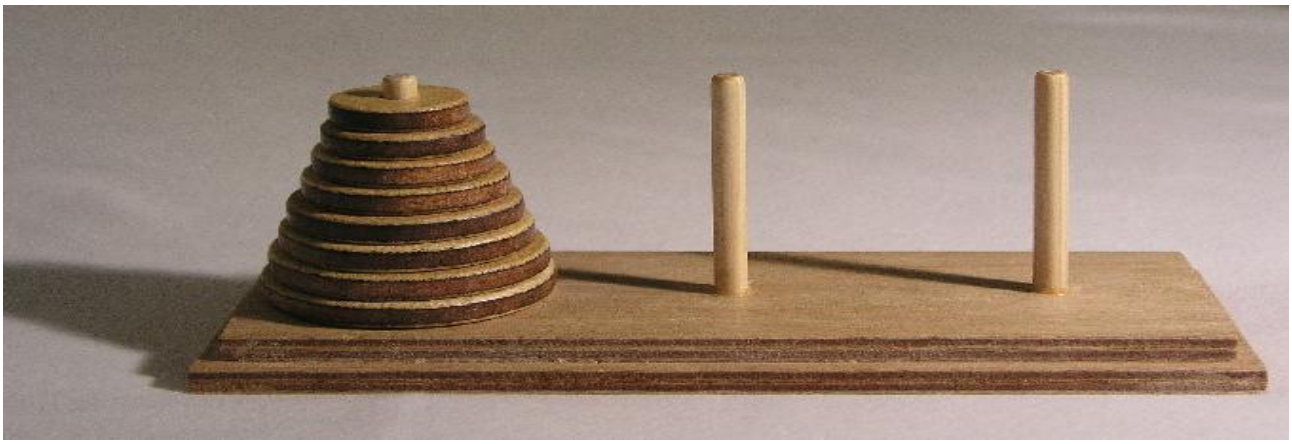
Example 3:

```
# 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(5) =',fib(5))

# 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(15)

# 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
```

Example 4:

Let us discuss one very interesting puzzle – called **tower of Hanoi**. It is also known as Brahma's riddle.

```
Check this link : https://en.wikipedia.org/wiki/Tower_of_Hanoi
```



Problem Statement: There are three pegs. There are a number of disks in the first peg. The task is to move the disks from the first to the third using second if necessary. There are two rules in the game. We can move only one disk at a time. We cannot place a biggest disk over a smaller disk.

It seems Brahma created 3 diamond sticks and put 64 golden disks in the decreasing order of size at Varanasi. It seems he asked the priests of Varanasi to move the disks from the first to the third using second if necessary. It seems that priests can move one disk per second. It seems that Brahma has said that the world will perish if all the disks are moved. How come the world has not perished so far? How do we solve the problem?

The solution is extremely simple. If there is no disk(say n = 0), then nothing to move. This is the base case or the escape hatch.

If there are number of disks(say n), then somehow move the (n – 1) disks from the first peg to the second peg using third peg if necessary, then move the bottom disk from the first peg to the third peg. Then move (n – 1) disks from the second peg to the third peg using first if necessary.

So, the number of moves for varying values of n shall be:

```
n = 1 => moves = 1
n = 2 => moves = 3
n = 3 => moves = 7
…
moves for n disks = 2 ** n - 1
```

If you calculate the time required, it will turn out to be a huge number.

```
18446744073709551615 seconds = 584942417355 years.
```

Our priests are very slow. Can we do this task on our computers for n = 64? Our computers at a speed of about 5 GHz – can do so many operations in one second. How much time will our computers take to solve this assuming that a move can be made in one clock cycle?

It turns out to be : 117 years !!

These problems require time proportion to the power of n. There are many such problems which can be solved only for a small value of n in a reasonable type.

So, our computers are not really powerful. Are they?

```
# 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

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')
```

Here is the output for n = 3.

```
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
```

Example 5:

There are three different implementations of a recursive function in this program. You may want to closely observe all of them.

The first one find mimics the in operator.

Given a list and an element, it returns True or False.

So, what is the logic?

If the list is empty, then return False

else if the zeroth element matches the given element, then return true

otherwise try find again on the slice of the list starting from position 1.

This is not a very efficient program as slicing of lists will cause copying the elements.

The second version of find uses two techniques.

a) **Wrapper function**: The client calls a function with two arguments as in the earlier case. It in turns calls another function with an extra argument.

b) **Function tracks the state**: In each call, the function indicates the position in the list which should be used to compare. The value is builtup over each call.

The third version uses a default parameter – so that the wrapper function is avoided. This tip came from one of my students during my lecture.

```
#name : 4_recursion.py

"""
def find(mylist, x) :
      if not mylist:
            return False
      elif mylist[0] == x :
            return True
      else:
            return find(mylist[1:], x)


x = [11, 33, 22, 44]
print(find(x, 22))
print(find(x, 5))
"""

"""

def find_(mylist, x, i) :
      if i == len(mylist):
            return -1
      elif mylist[i] == x :
            return i
      else:
            return find_(mylist, x, i + 1)

def find(mylist, x) :
      return find_(mylist, x, 0)

x = [11, 33, 22, 44]
print(find(x, 22))
print(find(x, 5))
"""
def find(mylist, x, i = 0) :
      if i == len(mylist):
            return -1
      elif mylist[i] == x :
            return i
      else:
            return find(mylist, x, i + 1)

x = [11, 33, 22, 44]
print(find(x, 22))
print(find(x, 5))
```

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/**