DATA STRUCTURE

Dr. Rania Baashirah

r.baashirah@ubt.edu.sa

Mobile: +966-559009202

Recursion in Python

- Recursion
- Recursion is a method of solving problems that involves breaking the problem into smaller and smaller subproblems until we get to small enough problem that can be solved trivially.
- In programming
- Another approach of programming whereby a method directly or indirectly **calls itself**
- How to stop recursion?
- using a condition called "base case". It is the case that can return the result directly without a recursive call.

Summation of a list

$$\text{total} = (1 + (3 + (5 + (7 + 9))))$$

$$\text{total} = (1 + (3 + (5 + 16)))$$

$$\text{total} = (1 + (3 + 21))$$

$$\text{total} = (1 + 24)$$

$$\text{total} = 25$$

Example: Factorial

- The factorial is defined in this way:

$$n! = \begin{cases} 1 & n = 0 \\ 1 \times 2 \times \cdots \times n & n > 0 \end{cases}$$

  This can be computed by a loop.

- We can also define the factorial as:

$$n! = \begin{cases} 1 & n = 0 \\ n \times (n-1)! & n > 0 \end{cases}$$

  So, if we know how to compute the factorial of $n-1$, we know how to compute the factorial of n.

Factorial

- All what I Know that factorial(1) is 1, thus I can calculate any factorial based on this rule.
- To see how the calculation is done. Trace factorial(4):

```
Factorial(4)     = 4 * factorial(3)
                 = 4 * (3 * factorial(2))
                 = 4 * (3 * (2 * factorial(1)))
                 = 4 * (3 * (2 * 1))
                 = 4 * (3 * 2)
                 = 4 * 6
                 = 24
```

Factorial

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

#----Test ----

f = factorial(5)
print(f)
```

each recursive method MUST have a condition called **"base case"** to determine the ending point of recursive call

The Three Laws of Recursion

- All recursive algorithms must obey three important laws:
  1. A recursive algorithm must have a base case.
  2. A recursive algorithm must change its state or input and move toward the base case.
  3. A recursive algorithm must call itself, recursively.

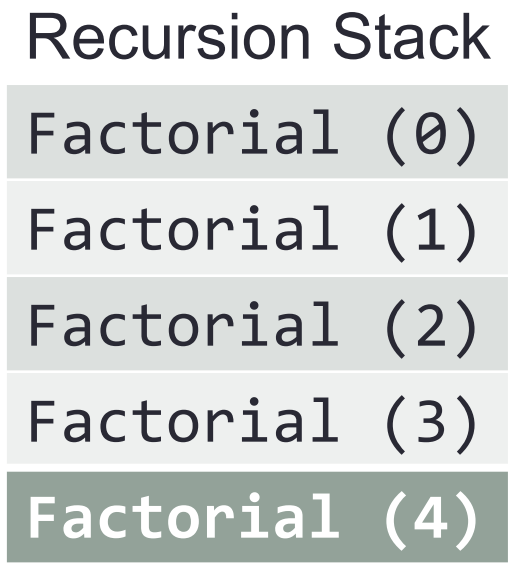
Example

- How many recursive calls are made when computing Sum [2,4,6,8,10]
- Answer?

Stack Overheads in Recursion

- We need to pay attention to two important results:

1. The depth of recursion
2. The stack overhead in recursion



- Every recursive call keeps a copy of the recursive method in a special memory space → called (Recursion Stack).

- If the recursion stack got full then it will cause the freeze of the program
  - sometimes this error called → ***"stack overflow"***.

Is it a Good idea to write a Recursive Function?

- Recursion enables us to write a program in a natural way → it helps programmers to generate realistic looking scenery for computer generated movies.

- The speed of the recursive program is slower because of **“Stack Overhead”**

- In a recursive program you have to specify recursive conditions, terminating conditions, and recursive expressions

Summation of a list

- Here is both implementations of getting a summation of a list. First is the iterative code and the second is using recursive code.

$$\text{total} = (1 + (3 + (5 + (7 + 9))))$$

$$\text{total} = (1 + (3 + (5 + 16)))$$

$$\text{total} = (1 + (3 + 21))$$

$$\text{total} = (1 + 24)$$

$$\text{total} = 25$$

```
def sumList(lst):
    total = 0
    for item in lst:
        total +=  item

    return total

def RecursiveSumList(lst):
    if len(lst) == 1:
        return lst[0]
    else:
        return lst[0] + RecursiveSumList( lst[1:] )

#----Test ----

L = [1,3,5,7,9]
print( sumList(L) )
print( RecursiveSumList(L) )
```

GCD in Recursion

- gcd(a,0)= a
- gcd(a,b)= gcd(b, a mod b)

```
def RecursiveGCD(a,b):
    if (b == 0):
        return a
    else:
        return RecursiveGCD(b, a % b )
r = RecursiveGCD(24,12)
print (r)
r = RecursiveGCD(565,15)
print (r)
```

Fibonacci in Recursion

- Feb(0) = 0
- Feb(1) = 1
- Feb(n) = Feb(n-1) + Feb(n-2)

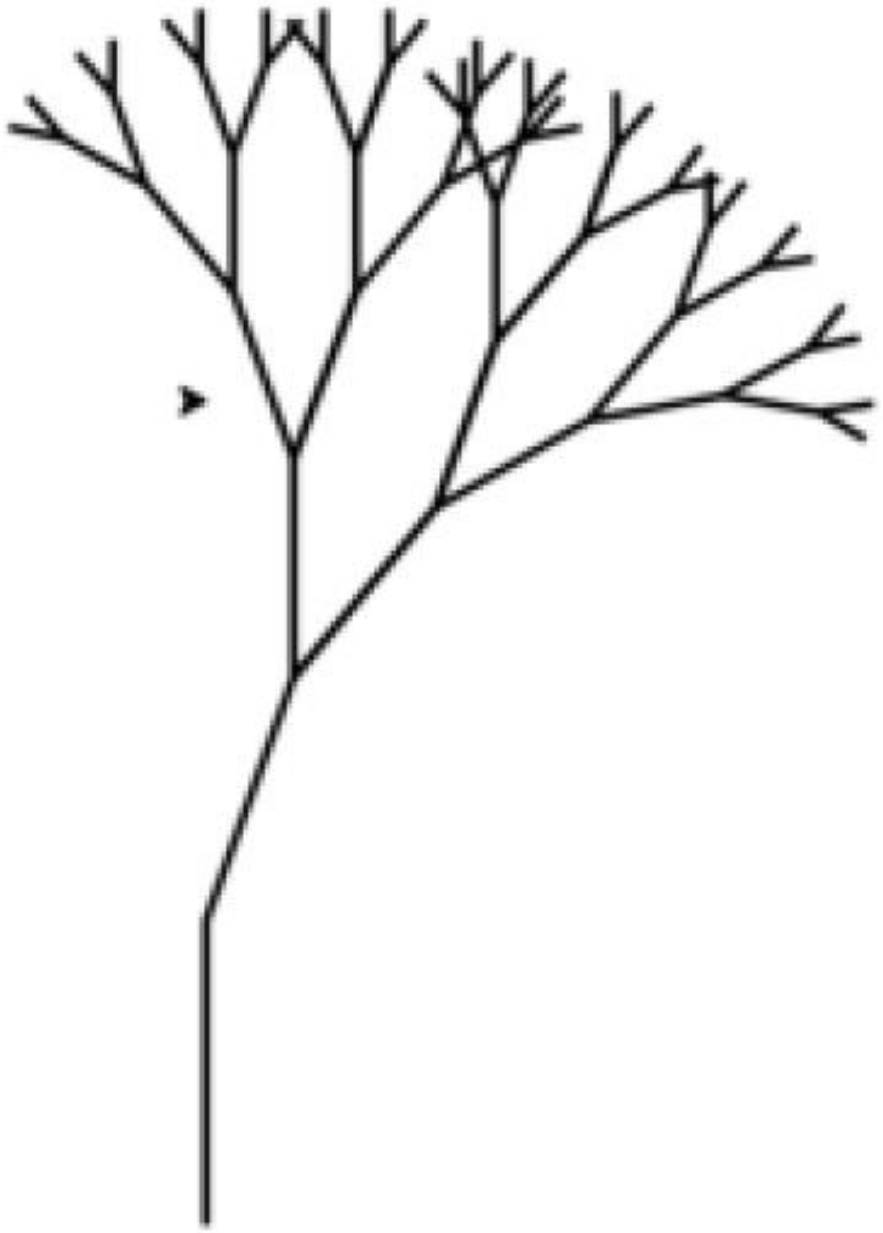
```
def feb(n):
    if n == 0:
        return 0

    if n == 1:
        return 1

    return feb(n-1) + feb(n-2);

#----Test ----

f = feb(10)
print(f)
```

Visualizing Recursion

- We can see the mental model to visualize what is happening in recursion functions
- We use Python's turtle graphics model → **Turtle**
- The idea is that the turtle will move forward, backward, left, right ..etc.
- You can change the color of the ink when the turtle moves its tail.

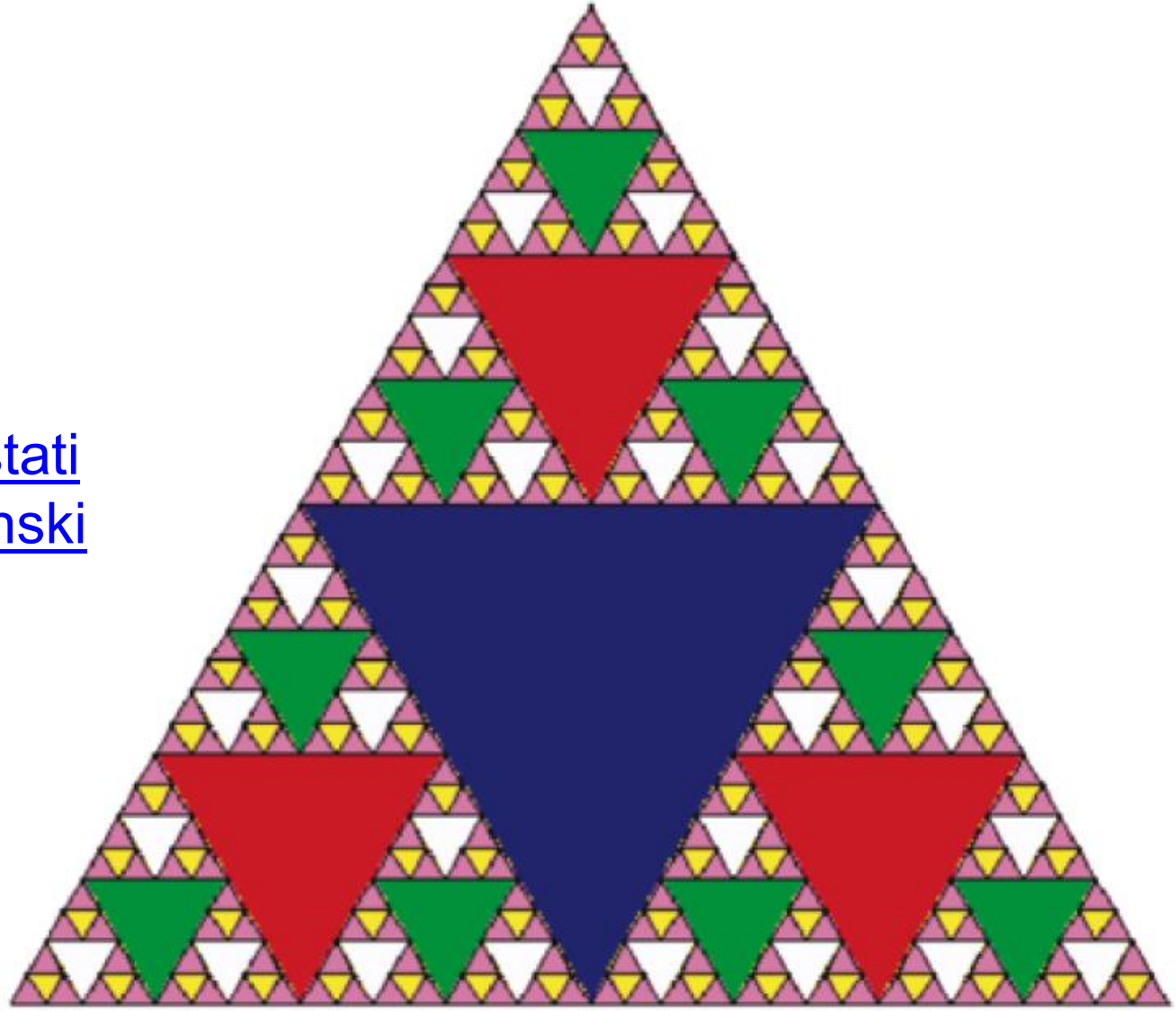
The First Half of the Tree

- http://interactivepython.org/courselib/static/pythonds/Recursion/pythondsintro-VisualizingRecursion.html



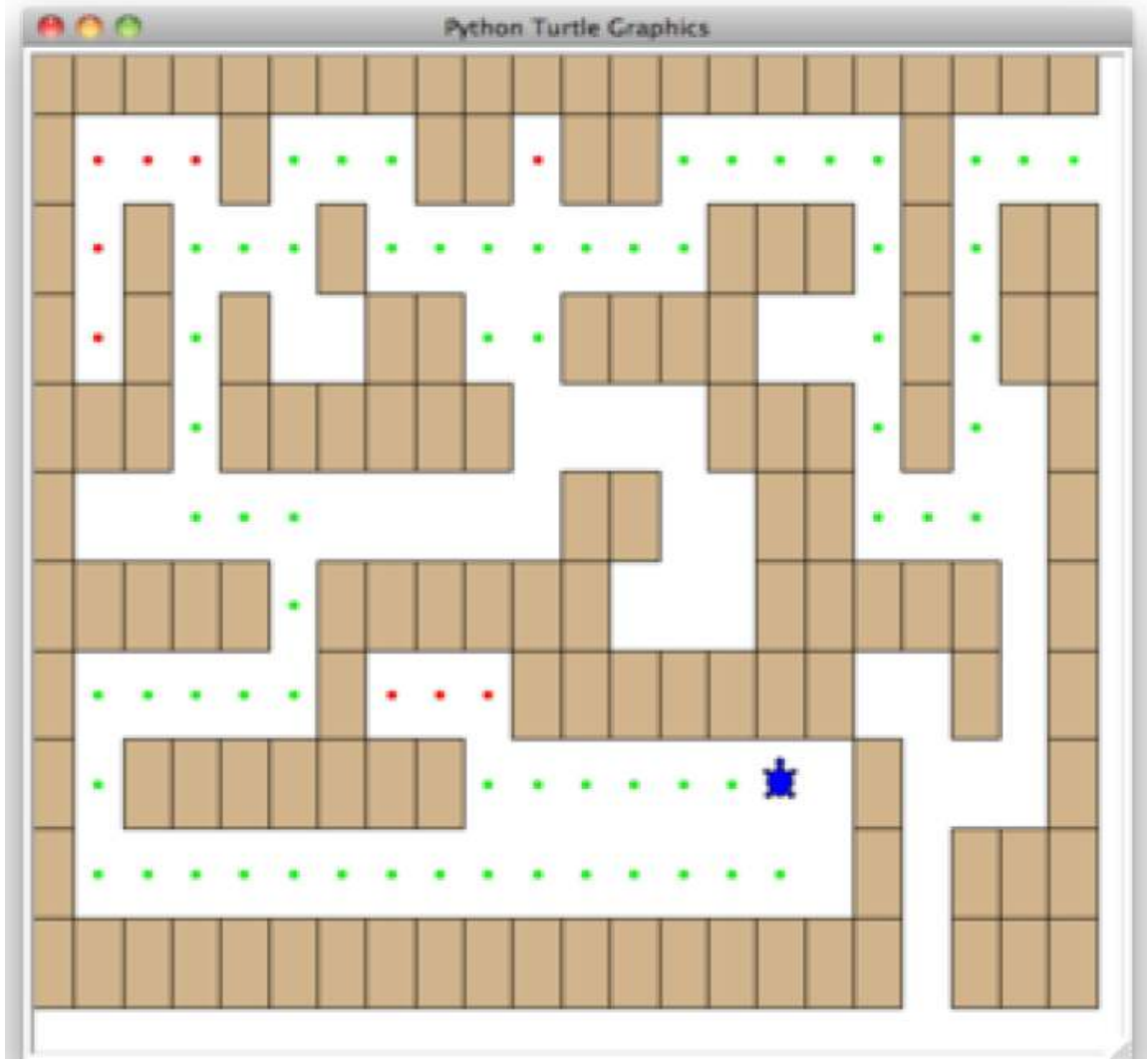
The Sierpinski Triangle

- http://interactivepython.org/courselib/static/pythonds/Recursion/pythondsSierpinskiTriangle.html



The Finished Maze Search Program

- http://interactivepython.org/courselib/static/pythonds/Recursion/pythondsSierpinskiTriangle.html



Recursion Examples

- Write a recursion function to find the sum of every number in an integer number.
  - Example: n=9182 => SumDigits = 9+1+8+2 = 20

```
def sumDigits(n):
    if n < 10:
        return n

    lastDigit = n % 10
    rest = n//10
    return lastDigit + sumDigits(rest)

#----Test ----
print( sumDigits(9182) )
```

Types Of Recursion

- Linear Recursion
- Binary Recursion
- Tail Recursion
- Exponential Recursion
- Nested Recursion
- Mutual Recursion