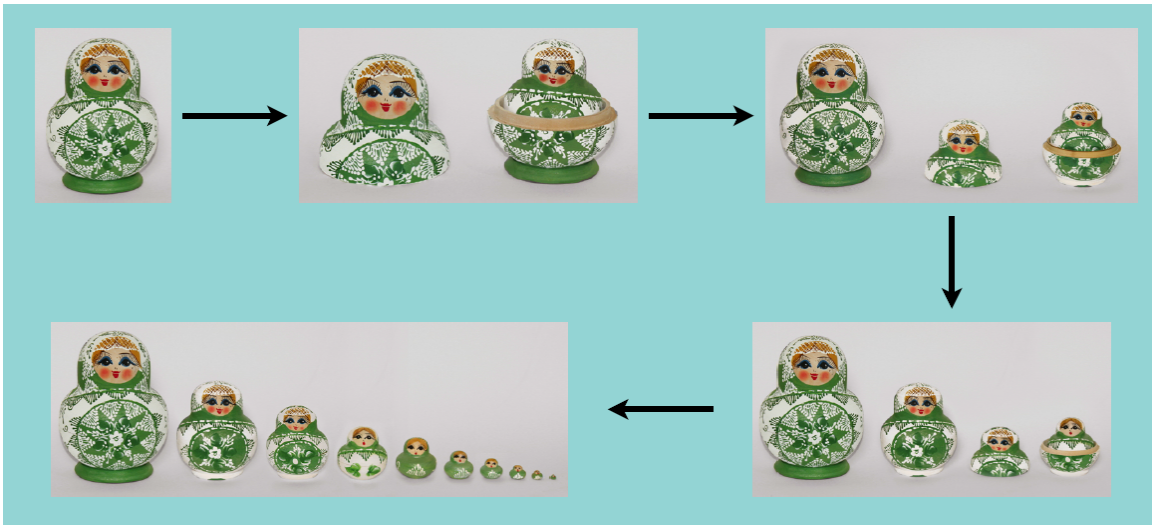
What is Recursion?

Recursion = a way of solving a problem by having a function calling itself



- Performing the same operation multiple times with different inputs
- In every step we try smaller inputs to make the problem smaller.
- Base condition is needed to stop the recursion, otherwise infinite loop will occur.

Why Recursion?

1. Recursive thinking is really important in programming and it helps you break down big problems into smaller ones and easier to use

- If you can divine the problem into similar sub problems
- Design an algorithm to compute nth…
- Write code to list the n…
- Implement a method to compute all.
- Practice
  - when to choose recursion?

2. The prominent usage of recursion in data structures like trees and graphs.
3. interview
4. It is used in many algorithms (divide and conquer, greedy and dynamic programming)

How Recursion works?

1. A method calls it self
2. Exit from infinite loop

```
def recursionMethod(parameters):
    if exit from condition satisfied:

        return some value

    else:

        recursionMethod(modified parameters)
```

How Recursion works?

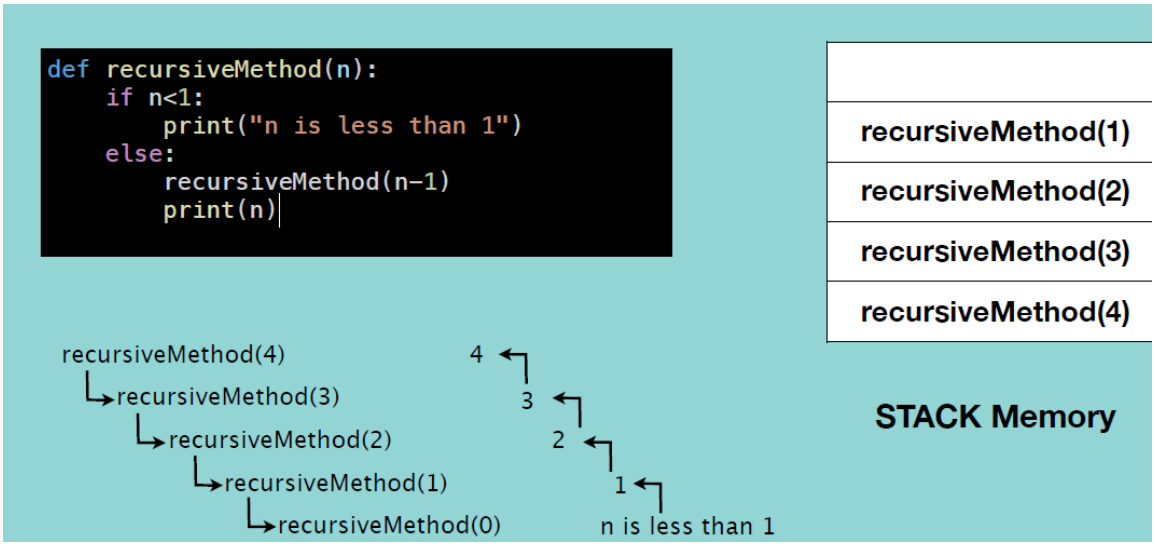
In [3]:
```
def firstMethod():
    secondMethod()
    print("I am the first Method")
def secondMethod():
    thirdMethod()
    print("I am the second Method")
def thirdMethod():
    fourthMethod()
    print("I am the third Method")
def fourthMethod():
    print("I am the fourth Method")
firstMethod()
```

```
I am the fourth Method
I am the third Method
I am the second Method
I am the first Method
```

How Recursion works?

In [5]:
```
def recursiveMethod(n):
    if n<1:
        print("n is less than 1")
    else:
        recursiveMethod(n-1)
        print(n)
recursiveMethod(5)
```

```
n is less than 1
1
2
3
4
5
```



When to Use/Avoid Recursion?

When to use it?

- When we use memoization in recursion
- When we can easily breakdown a problem into similar subproblem
- When we are fine with extra overhead (both time and space) that comes with it
- When we need a quick working solution instead of efficient one
- When traverse a tree

When we use memoization in recursion

When avoid it?

- If time and space complexity matters for us.
- Recursion uses more memory. If we use embedded memory. For example an application
- Recursion can be slow

Fibonacci numbers - Recursion

Fibonacci sequence is a sequence of numbers in which each number is the sum of the two preceding ones and the sequence starts from 0 and 1

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89...

Step 1 : Recursive case - the flow

5 = 3 + 2 **f(n) = f(n-1) + f(n-2)**

Step 2 : Base case - the stopping criterion

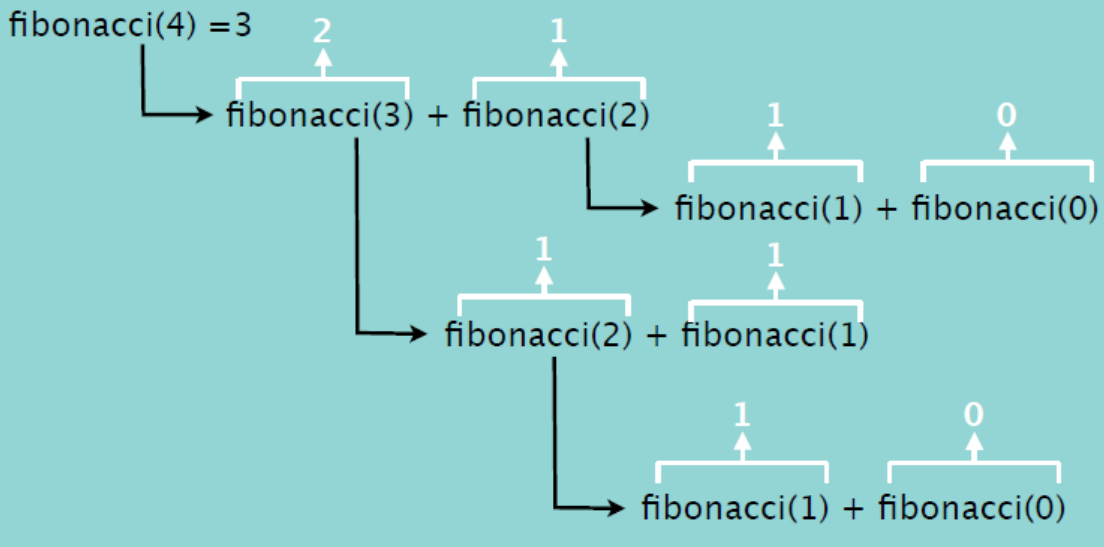
- 0 and 1

Step 3 : Unintentional case - the constraint

- fibonacci(-1) ??
- fibonacci(1.5) ??

In [1]:
```
def fibonacci(n):
    assert n >=0 and int(n) == n , 'Fibonacci number cannot be negative nu
    if n in [0,1]:
        return n
    else:
        return fibonacci(n-1) + fibonacci(n-2)
fibonacci(12)
```

Out[1]: 144



In [2]:
```
#### Russian Doll recursive function ###

def openRussianDoll(doll):
    if doll == 1:
        print("All dolls are opened")
    else:
        openRussianDoll(doll-1)


openRussianDoll(4)
```

```
All dolls are opened
```

In [13]:
```
## Recursion vs Iterarion###

def powerOfTwo(n):
    if n == 0:
         return 1
    else:
        power = powerOfTwo(n-1)
        return power * 2

print(powerOfTwo(5))

```

```
32
```

In [14]:
```
n=int(input("enter number"))
i = 0
power = 1
while i < n:
    power = power * 2
    i = i + 1
print(power)
```

```
enter number5
32
```

In [5]:
```
 ## Factorial###


def factorial(n):
    assert n >= 0 and int(n) == n, 'The number must be positive integer or
    if n in [0,1]:
        return 1
    else:
        return n * factorial(n-1)
factorial(10)
```

Out[5]:
```
3628800
```

In [6]:

```
def fibonacci(n):
    assert n >=0 and int(n) == n , 'Fibonacci number cannot be negative nu
    if n in [0,1]:
        return n
    else:
        return fibonacci(n-1) + fibonacci(n-2)

print(fibonacci(7))
```

```
13
```