Recursion

- PoPI

Recursion: Motivation

- In some problems, it may be natural to define the problem in terms of the problem itself.
- Recursion is useful for problems that can be represented by a simpler version of the same problem.
- Example: the factorial function

```
6! = 6 * 5 * 4 * 3 * 2 * 1
```

- We could write:

```
6! = 6 * 5!
```

Recursion: Motivation (cont)

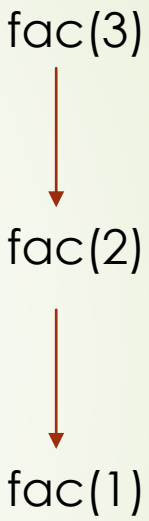
- The factorial function is only defined for *positive* integers. So we should be a bit more precise:

```
n! = 1          (if n is equal to 1)
n! = n * (n-1)!   (if n is larger than 1)
```

- Another example: Fibonacci numbers
  - 0th Fibonacci number is 0 — fib(0) = 0
  - 1st Fibonacci number is 1 — fib(1) = 1
  - n-th Fibonacci number is the sum of (n-1)th and (n-2)th — fib(n) = fib(n-1)+fib(n-2)
    - 2nd Fibonacci number is the sum of the 0th and 1st — fib(2) = fib(0) + fib(1)
    - 3rd Fibonacci number is the sum of the 1st and 2nd — fib(3)=fib(2)+fib(1)
    - …

Factorial Function: Implementation

```
def fac(numb):
    if numb ==1: return 1
    else:
        fac_numb_minus_1 = fac(numb-1)
        result = numb * fac_n_minus_1
        return result
```



Visualise in Python tutor: https://goo.gl/wj2DHs

Factorial function: Recurvise and Nonrecursive Implementations

- Recursive

```
def fac(numb):
    if numb <=1: return 1
    else:
        fac_numb_minus_1 = fac(numb-1)
        result = numb * fac_n_minus_1
        return result
```

- Iterative

```
def fac(numb):
    product = 1
    for i  in range(1,numb+1):
        product = product * i
    return product
```

- For numb = 10, which implementation will consume more memory?

Price of recursion

We have to pay a price for recursion:

- calling a function consumes more time and memory than adjusting a loop counter.
- high performance applications (graphic action games, simulations of nuclear explosions) hardly ever use recursion.

In less demanding applications recursion is an attractive alternative for iteration (for the right problems!)

- many search and sort problems
- combinatorial problems: e.g., print all 0/1 strings of length n on Snakify

Infinite Recursion

```
def fac(numb):
    fac_numb_minus_1 = fac(numb-1)
    result = numb * fac_numb_minus_1
    return result
```

- Just as loops (while) recursion can proceed infinitely
- We forgot the if condition and the branch that would make fac return 1
- A recursive function must contain at least one non-recursive branch.
- The recursive calls must eventually lead to a non-recursive branch.

Fibonacci Numbers

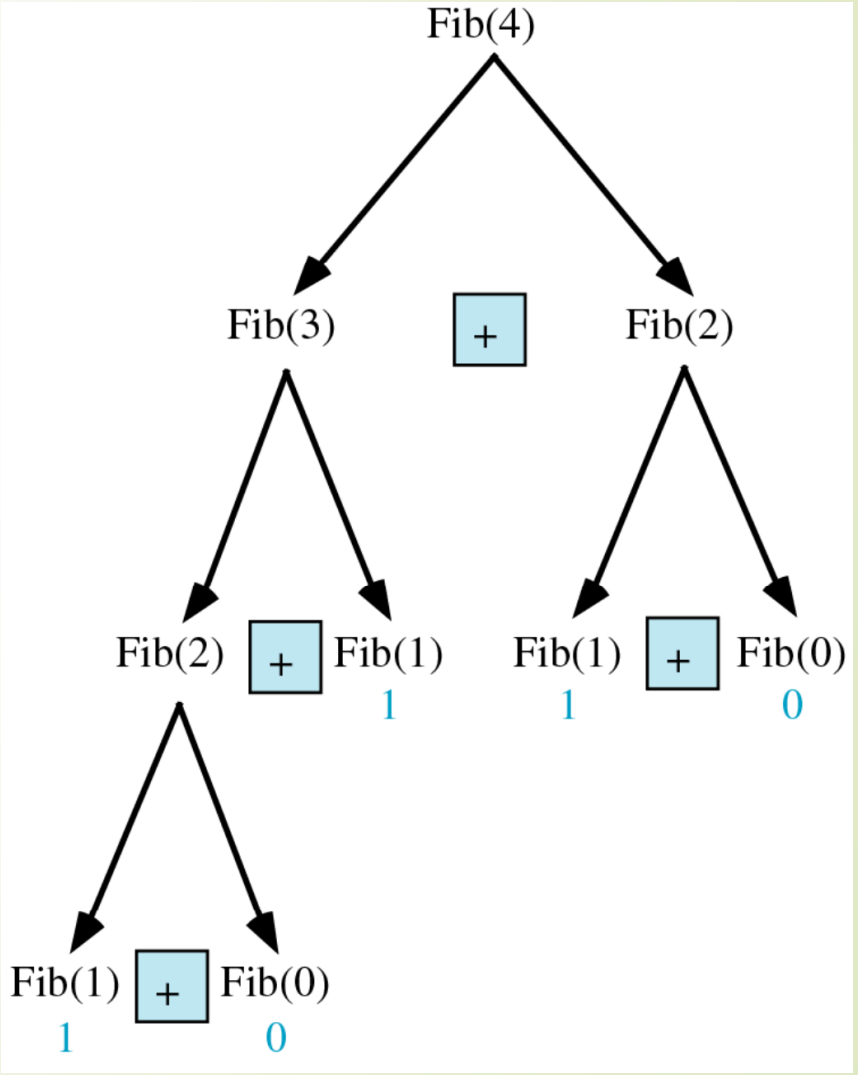
- fib(0) = 0, fib(1) = 1
- fib(n) = fib(n-1) + fib(n-2)

```
def fib(n):
    if n == 0: return 0
    elif n==1: return 1
    else:
        previous_fib_number = fib(n-1)
        previous_previous_fib_number = fib(n-2)
        result = previous_fib_number + previous_previous_fib_number
        return result
```

“Strategy” to compute Fibonacci Numbers

```
def fib(n):
    if n == 0: return 0
    elif n==1: return 1
    else:
        previous_fib_number = fib(n-1)
        previous_previous_fib_number = fib(n-2)
        result = previous_fib_number + previous_previous_fib_number
        return result
```

- https://goo.gl/F5ia3P

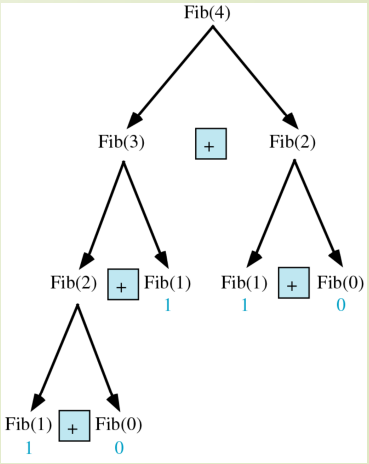


Iterative Version of Fibonacci Numbers

- Take the following iterative implementation (this is not the optimal one, you probably had a better one on Snakify)

```
def fib(n)
    all_fib_numbers = list()
    all_fib_numbers.append(0)
    all_fib_numbers.append(1)
    for i in range(2, n+1):
        all_fib_numbers.append(all_fib_numbers[i-1] + all_fib_numbers[i-2])
    return all_fib_numbers[n]
```

- Memory consumption of this code is not much better than of recursive version
- Running time of this code is however much shorter!



What does this recursion do?

```
def mystery_function(i, lst)                #lst is a list and i is an index in it
    if i == len(lst)-1: return lst[i]
    else:
        mystery_variable = mystery_function(i+1, lst)
        if lst[i] > mystery_variable: return lst[i]
        else: return mystery_variable


        mystery_function(i,lst) returns the maximal element of lst[i:]
        mystery_function(0,lst) returns the maximal element of lst


A = [1, 3, 2]
print(mystery_function(0,lst))
```