Recursion : A function calling itself.

1, Direct Recursion
2, Indirect Recursion

```
def fun1 :
    print ('fun 1')

def fun2:
    print ('Bef fun 1')
    fun1()
    print ('After fun 1')

print (Before fun2)
fun2()
print (After fun 2)
```

⇒

```
Bef fun 2
" fun 1
fun1
After fun 1
" fun2
```

1,
```
def fun1():
    fun1()
```

2,
```
def fun1():
    fun2()

def fun2():
    fun1()
```

⇒ if there is no base case, recursion will lead to stack overflow / segmentation fault.
or
Wrong Base Case,
↳ also updation of Base Case variable.

↳ Advantages of Recursion :

① clean & simple code
② Easy to solve tough problems

↳ Disadvantages of Recursion :

① We can solve every recursive problem through iteration & vice versa
② Recursion requires greater space
③ Greater time requirements beoz of func calls and return overhead.
↳ As fn calls remain in call stack, till we reach base case

Application of Recursion :

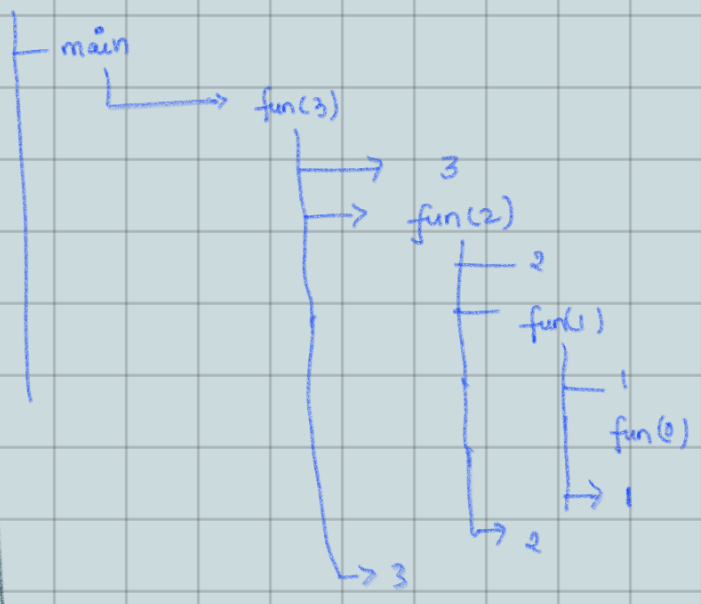
- → Algo
  - ↳ DP
  - ↳ Back tracking
  - ↳ divide & conquer
- → Problems / Questions
  - ↳ Towers of Hanoi
  - ↳ DFS Based Traversals
    - ↳ DFS of graph
    - ↳ INORDER, PREORDER, POSTORDER. Traversal of tree.

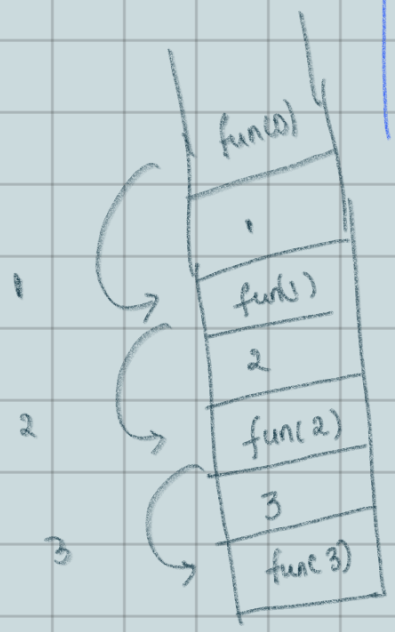
→ Recursion Output practice

①
```
def fun(n):
    if n==0:
        return
    print(n)
    fun(n-1)
    print(n)

fun(3).
```

Output :





② def fun(n):

if n==0:

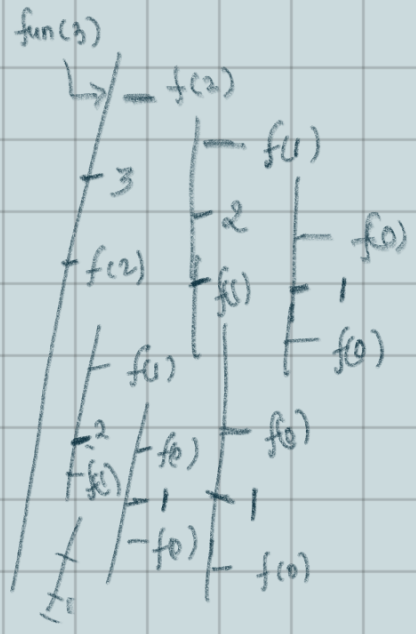
return

fun(n-1)

print(n)

fun(n-1)

fun(3).



1, 2, 1, 3, 1, 2, 1

③

def fun(n):

if n==1:

return 0;

else:

return 1 + fun(n/2)

fun(16)

fun(16)
↳ 1 + fun(8)
↳ 1 + fun(4)
↳ 1 + fun(2)
↳ 1 + f(1)
0

⇒ 4

④

def fun(n)

if n==0:

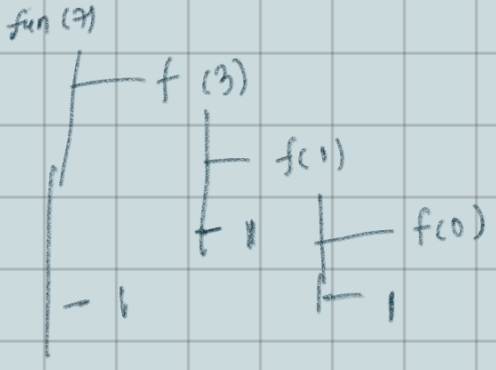
return

fun(n/2)

print(n%2)

fun(7)

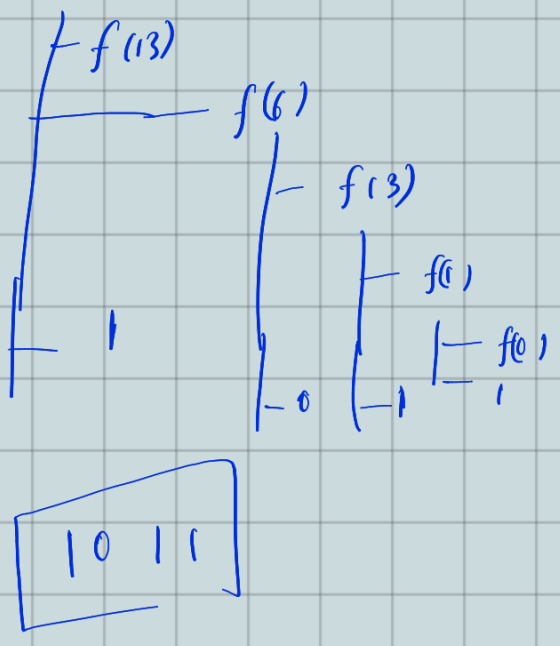
⇒



output = 111

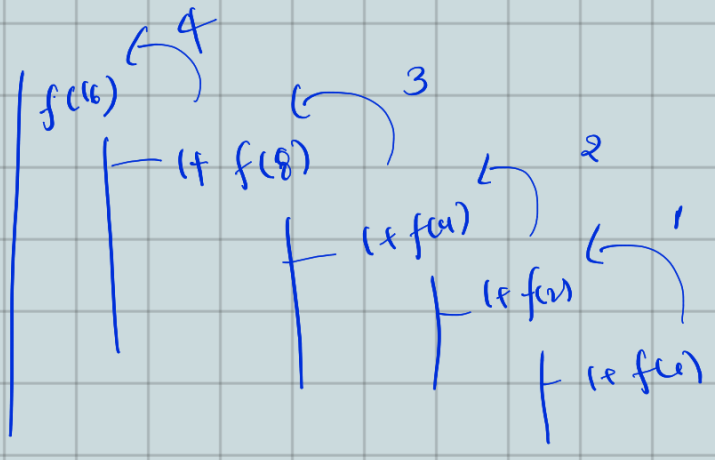
⑤ Print Binary Equivalent of n using recursion.

```
def fun(n):
    if n==0:
        return
    print(n%2)
    fun(n//2)
```



⑥ $\log_2 n$

```
def fun(n):
    if n==1
        return 0
    return 1 + fun(n/2)
```



⑦ Powel of numbers

```
def fun(base, exponent):
    if exponent==0:
        return 1
    elif exponent < 0:
        return 1/fun(base, -exponent)
    else:
        return base * fun(base, exponent-1)
```

→ Print n to 1

fun(4) ⇒ print 4 & call 3
↳ 3 & call 2 - - -

```
def fun(n):
    if n==0:
        return
    Print(n)
    fun(n-1)
```

⇒ O(n), O(n)

→ Fibonacci

0 1 1 2 3 5 8 13 21 - - —

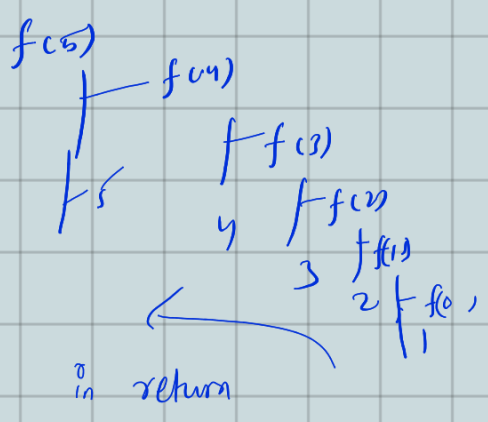
O(2^n, n)

```
def fun(n):
    if n <=1
        return n
    return fun(n-1) + fun(n-2)
```

→ print 1 to n

first we need to make fn call
Then print

```
def fun(n):
    if n==0:
        return
    f(n-1)
    print(n)
```

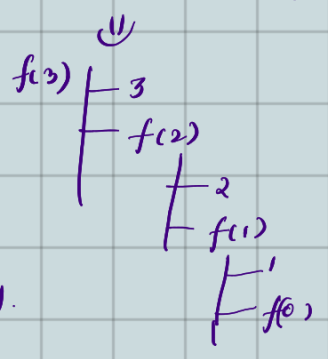


- Tail Recursion :

  by printing from n to 1 as the parent func has nothing more to do it will be faster by

  takes lesser time than

  printing 1 to n.



- A func is said to be tail recursive if last things that happens is recursive call.

- Tail Recursion is faster because the caller / call stack doesn't need to save the state.

- The changes that modern compilers does to not to save state is ...

```
def fun(n):
start:
    if n==0
        return
    print(n)
    n = n-1
    goto start;
```

} Compilers does this.

⇒ All this is called tail call elimination

⇒ They do this optimization, if it wasn't done, we would need O(n) Aux space for recursive func calls we need to save those states and resume those states.

All that recursive head is gone.

⇒ The recursive func is turned into iterative by using the goto label ..

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

Not tail recursive

⟹ Tail recursive conversion by adding params

```
def f(n, K):
    if n==0 | n==1:
        return K
    return f(n-1, K*n)
```

→ Natural num sum.

1 + 2 + 3 + 4 + 5 → 15

```
fun(n):
    if n < 1:
        return n.
    return n + f(n-1)
```

```
fun(n, K):
    if n < 1:
        return K
    return fun(n-1, K+n)
```

→ palindrome check using recursion:

```
def ispalindn(str, start, end):
    if start >= end:
        return True
    return (str[start] == str[end]) and
           ispalindrome(str, start+1, end-1)
```

→ Sum of digits:

```
def sum(n, k):
    if n <= 0:
        return k
    return sum(n//k, k + n%.10)
```

→ num of digits

```
def count(n, k):
    if n <= 0:
        return k
    return count(n/10, k+1)
```

- Array Sorted / Not.

[1, 2, 3, 4, 5]

[1, 2, 4, 3, 5]

```
def func(arr, index):
    if index == len(arr) - 1:
        True
    if arr[i] > arr[i+1]:
        False
    return isSorted(arr, i+1).
```

[1, 2, 4, 3, 5]

i i

i i

False returned.