Lesson 12b: Recursion Exercises

Recursive Algorithms

Recursion:

when a procedure/function calls to itself (directly or indirectly)

Direct Recursion

```
def functionA (…):

    …
    functionA(…)
    …
```

Indirect Recursion

```
def functionA (…):

    …
    functionB(…)
    …

def functionB (…):

    …
    functionA(…)
    …
```

Recursive Algorithms

Any recursive function must be based on a recursive definition of the problem and must cover two cases:

- Basic case: returns a solution directly, without calling itself again.
- General case (recursive): Returns the solution after calling itself again with a new value of the input parameters which simplifies the initial problem.

Exercise: Palindrome

Palindrome: A word that reads the same backward as it does forward. For example: kayak, level, civic

Write the recursive function **is_palindrome** that checks is a word is a palindrome.

1. Define the problem recursively from the:
   - Basic case
   - General case

2. Implement the recursive function.

Exercise: Palindrome

1. Define the problem recursively from the:
   - **Basic case**: An empty string and a string consisting of a single character are inherently palindromic
   - **General case**: A string of length two or greater is a palindrome if it satisfies both of these criteria:
     – The first and last characters are the same.
     – The substring between the first and last characters is a palindrome.

2. Implement the recursive function.

Exercise: Palindrome

```
def is_palindrome(word):
    """Return True if word is a palindrome, False if not."""
    if len(word) <= 1:
        return True
    else:
        return word[0] == word[-1] and is_palindrome(word[1:-1])
```

```
>>> is_palindrome("")
True
>>> is_palindrome("a")
True

>>> is_palindrome("foo")
False
>>> is_palindrome("racecar")
True
```

Exercise: Exponentiation

Write the recursive function **PowerN** that computes x^n for a positive integer $n>=0$

1. Define the problem recursively from the:
   - Basic case
   - General case

2. Implement the recursive function.

3. How to modify the function for allowing negative exponents/powers (when n<0) ?

Exercise: Exponentiation

Write the recursive function **PowerN** that computes x^n for a positive integer $n>=0$

1. Define the problem recursively from the:
   - Base case

$$n = 0 \ \rightarrow \ X^0 = 1$$

   - General case

$$n > 0 \ \rightarrow \ X^n = X \cdot X^{n-1}$$

Exercise: Exponentiation

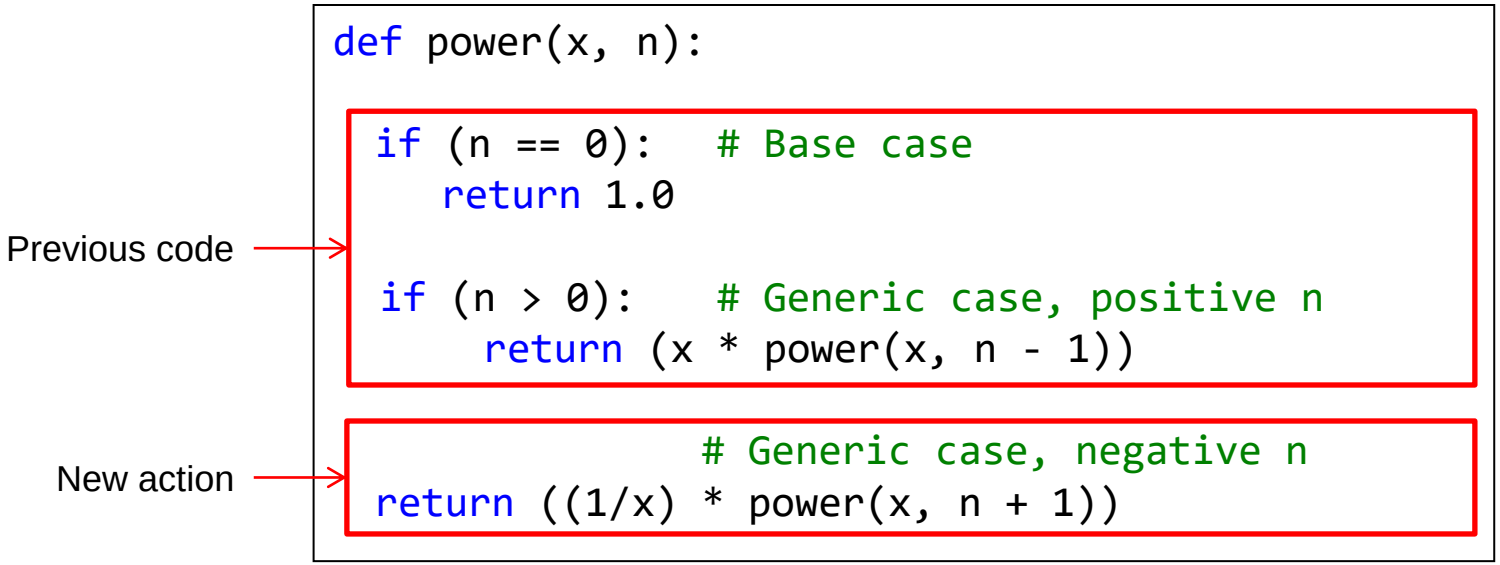
2. Implement the recursive function:

```
def powerN(x,  n):

     if (n == 0):   # Base case
          return 1.0
                         # Generic case
     return (x * powerN(x, n - 1))
```

3. How to modify the function for allowing negative powers (when n<0) ?

Exercise: Power N

3. How to modify the function for allowing negative powers (when n<0) ?



Exercise: Fibonacci series

Write the function **Fibonacci** of integer **n**. Implement 2 versions:

- Iterative version
- Recursive version

Exercise: Fibonacci series

Iterative Version

```
def fibonacci(n):

     f=1
     ant1 = 1
     ant2 = 1

     for k in range(2,n+1):
          f = ant1 + ant2
          ant2 = ant1
          ant1 = f

     return fib
```

Exercise: Fibonacci series

Recursive Version

Fibonacci series: 1,1,2,3,5,8,13,…

- Each element is the sum of the previous two elements

Formal definition:

$$a_0 = 1$$
$$a_1 = 1$$
$$a_n = a_{n-1} + a_{n-2}$$

Recursive definition:

Base case: n = 0 → Fibonacci (0) = 1

n = 1 → Fibonacci (1) = 1

Generic case:

n > 1 → Fibonacci (n) = Fibonacci (n-1) + Fibonacci (n-2)

Double recursivity: Fibonacci series

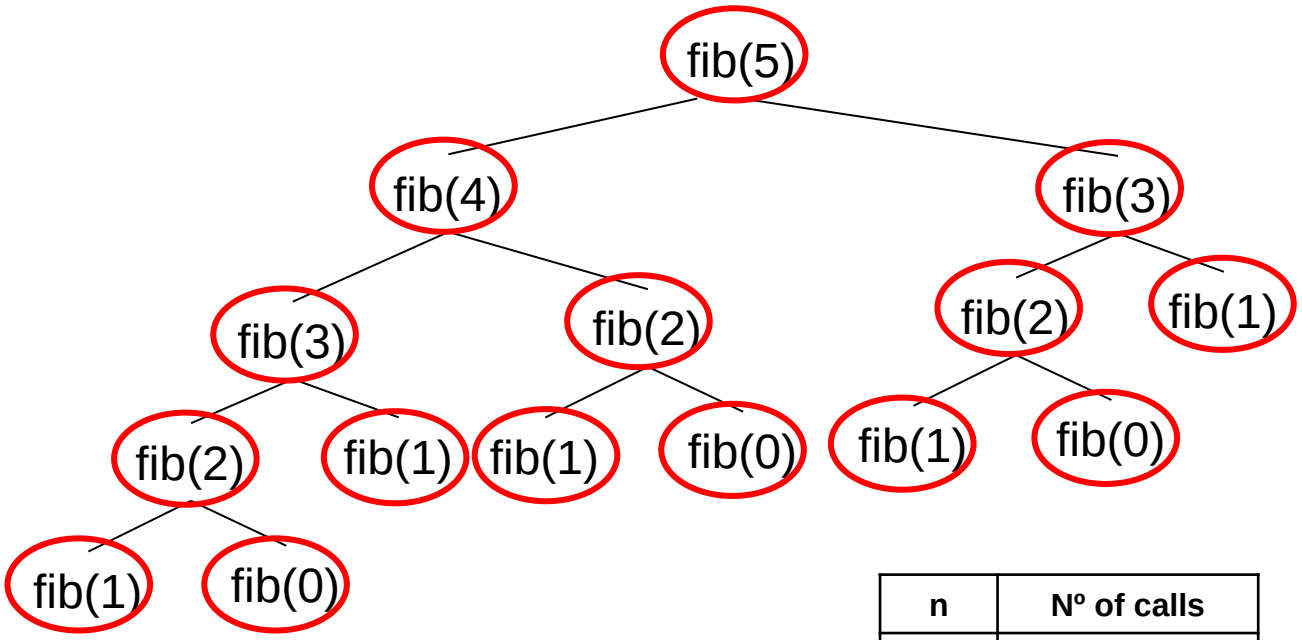
Fibonacci series: 1,1,2,3,5,8,13,…

- Each element is the sum of the previous two elements

```
def fibonacci (int n):

     if ((n == 0) or (n == 1)):         # Base case
           return 1
                                        # Generic case
     return fibonacci(n-1) + fibonacci(n-2)
```

Analysis of recursive algorithms

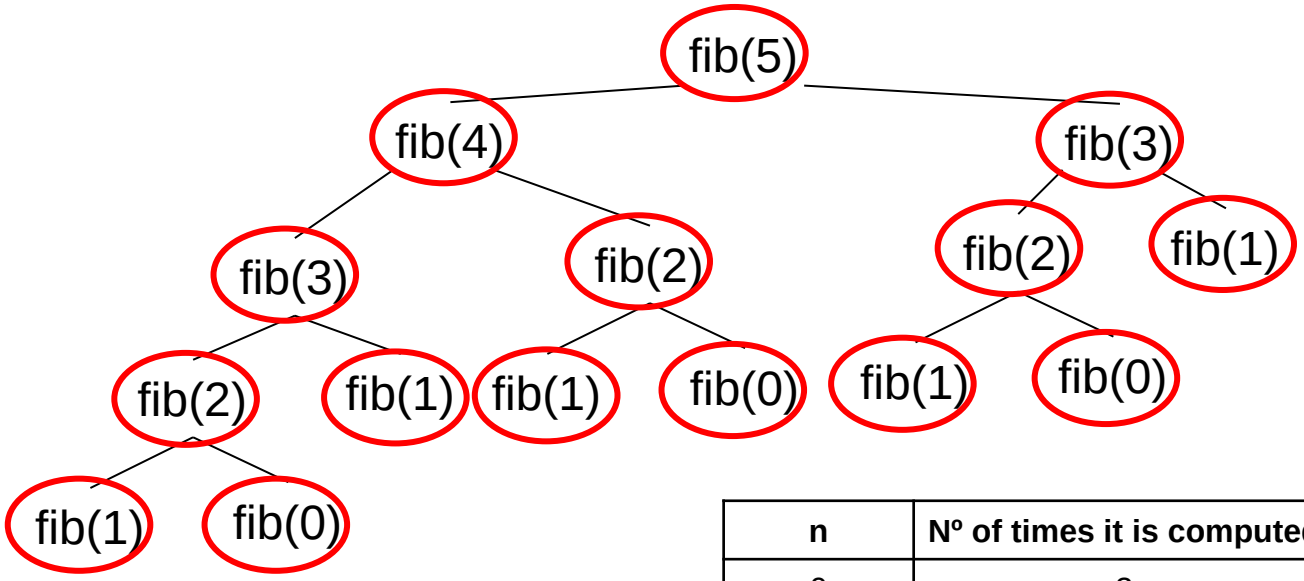
How many calls are necessary to compute `fibonacci(n)`?



n	Nº of calls
2	2
3	4
4	8
5	14

Analysis of recursive algorithms

And how many repeated calculations are performed to calculate `fibonacci(n)`?



n	Nº of times it is computed
0	3
1	5
2	3
3	2
4	1
5	1

Fibonacci series: iterative version

Iterative “equivalent” version

```
def fibonacci (n):

     fib = 1
     ant2 = 0
     while (n>0):

          ant1 = fib          # ant1 = fib(n-1)
          fib = fib + ant2     # fib = fib(n-1) + fib(n-2)
          ant2 = ant1         # ant2 = fib(n-2)
          n = n-1

     return fib
```

n	Nº of repetitions of the loop
2	2
3	3
4	4
5	5
…	…
50	50

Analysis of recursive algorithms

How many loop repeats are done to calculate `fibonacci(n)` in the iterative version?

- Number of repetitions: **O(n)**

How many calls are made to calculate `fibonacci(n)` in the recursive version?

- Number of repetitions: **$O(2^n)$**

Comparison Recursive vs Iterative:

- Recursive version is clearer
- Iterative version is more efficient: fewer loop repetitions than recursive calls (especially when **n** grows)

n	Nº of recursive calls	Nº of loop repetitions
2	2	2
3	4	3
4	8	4
5	14	5
…	…	…
50	$\approx 2x10^{10}$	50

Analysis of recursive algorithms

Factorial(n): Recursive vs Iterative version

- Number of repetitions in the iterative version → **O(n)**
- Number of recursive calls → **O(n)**

```
def fact (n):
    if ((n == 0) or (n == 1)):
    # Base case
        return 1
    # Generic case
    return n*fact(n-1)
```

```
def factorial (n):
    fac = 1
    while (n > 0):
        fac = fac * n
        n = n-1
    return fac
```

Comparison Recursive vs Iterative:

- Equal number of calls/repetitions
- Iterative version is more efficient: higher cost (time and memory) of a recursive call compared to a loop repetition.

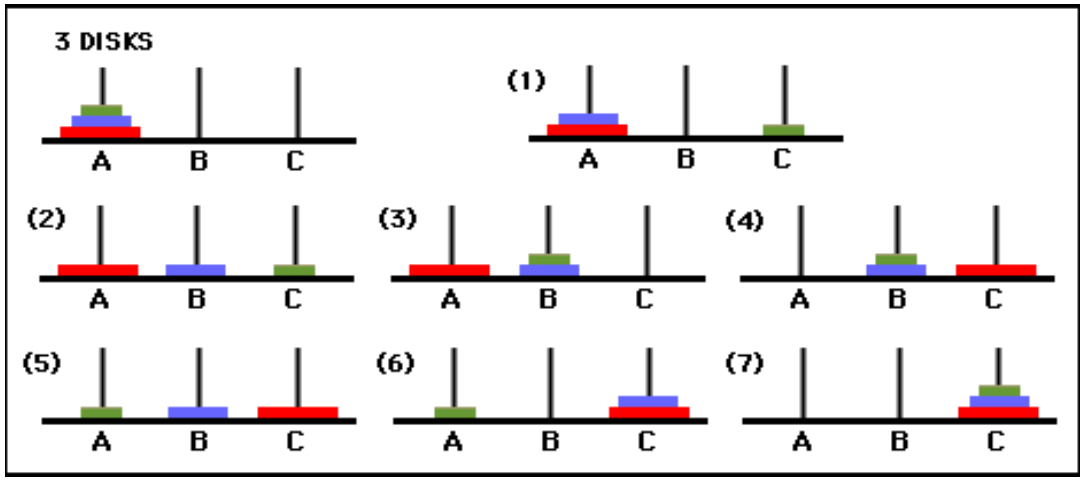
n	Nº of recursive calls	Nº of loop repetitions
2	2	2
3	3	3
4	4	4
5	5	5
…	…	…
50	50	50

Exercise: Tower of Hanoi

The Tower/s of Hanoi is a mathematical game or puzzle consisting of three rods and a number of disks of various diameters, which can slide onto any rod. The number of disks determines the level of complexity. It is typically used for solving recursive algorithms.

The puzzle begins with the disks stacked on one rod in order of decreasing size, the smallest at the top, thus approximating a conical shape. The objective is to move the entire stack from the first rod (source rod) to the last rod (target rod), obeying the following rules:

- Only one disk may 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 or on an empty rod.
- No disk may be placed on top of a disk that is smaller than it.



Exercise: Tower of Hanoi (curiosity)

There is a legend that explains that the towers, with 64 disks, were created in a temple at the beginning of the world. The temple monks must solve the puzzle. According to the legend, when the last move of the puzzle is completed, the world will end.

If the legend were true, and if the priests were able to move disks at a rate of one per second, using the smallest number of moves, it would take them $2^{64} - 1$ seconds or roughly 585 billion years to finish, which is about 42 times the current age of the universe.

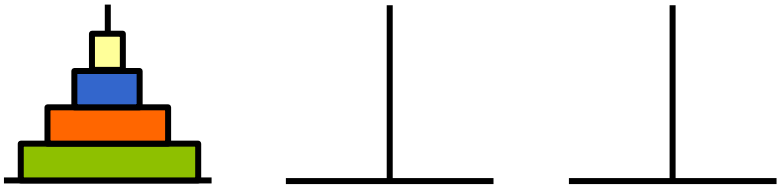
In comparison to the magnitude of this number, the Earth is about 4.5 billion years old, and the Universe is about 14 billion years old, only a small fraction of that number.

Exercise: Tower of Hanoi

https://www.mathsisfun.com/games/towerofhanoi.html

Exercise: Tower of Hanoi

Let’s solve the problem for 4 disks



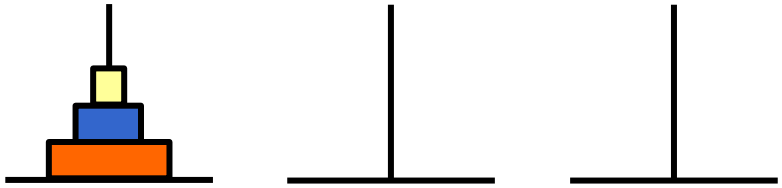
We assume that we know how to solve the problem for 3 disks:

How we can use this info to solve the problem for 4 disks?

1. Move the top 3 disks to the auxiliary tower
2. Move the lower disk to the target tower
3. Move the top 3 disks to the target tower

Exercise: Tower of Hanoi

Let’s solve the problem for 3 disks



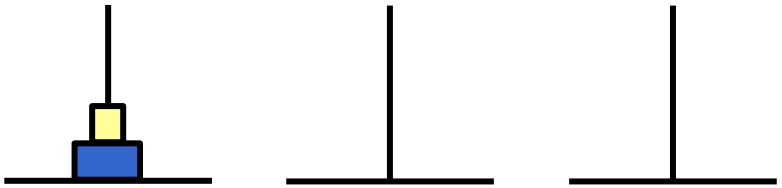
We assume that we know how to solve the problem for 2 disks:

How we can use this info to solve the problem for 3 disks?

1. Move the top 2 disks to the auxiliary tower
2. Move the lower disk to the target tower
3. Move the top 2 disks to the target tower

Exercise: Tower of Hanoi

Let’s solve the problem for 2 disks



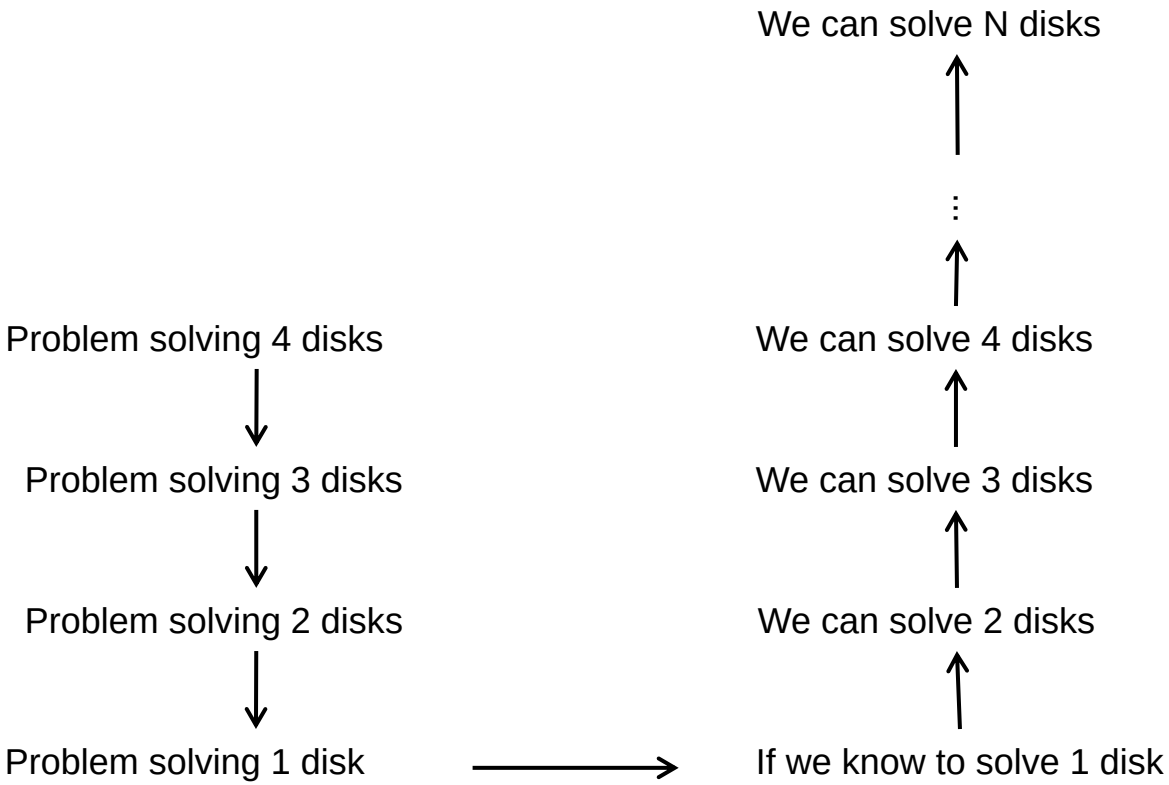
We assume that we know how to solve the problem for 1 disk:

1. Move the top disk to the auxiliary tower
2. Move the lower disk to the target tower
3. Move the top disk to the target tower

Exercise: Tower of Hanoi

Example of a recursive problem:

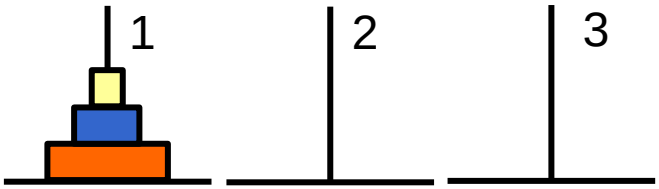
The solution is based on the solution of the same problem that we want to solve, but in a simpler version



Exercise: Tower of Hanoi

```
def THanoi(n , source, target, aux):

    if n==1:
        print("Move disk 1 from the source", source, “to the target rod", target)
        return

    THanoi(n-1, source, aux, target)

    print("Move disk ", n, “from the source", source, “to the target rod", target)

    THanoi(n-1, aux, target, source)
```

Exercise: Tower of Hanoi



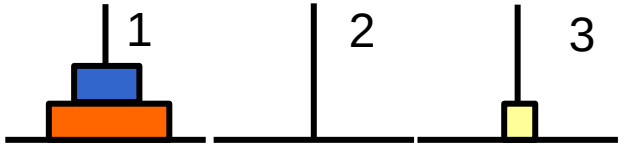

```
def THanoi(n , source, target, aux):
    if n==1:
        print("Move disk 1 from source",source,"to target", target)
        return
    THanoi(n-1, source, aux, target)
    print("Move disk",n,"from source",source,"to target", target)
    TorresHanoi(n-1, aux, target, source)
```

THanoi(3 , 1, 3, 2)

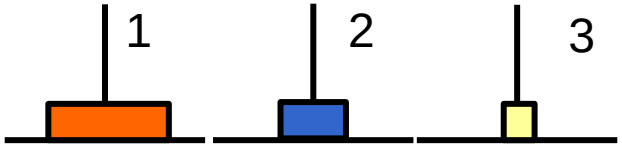
THanoi(2 , 1, 2, 3)

THanoi(1 , 1, 3, 2)

"Move disk 1 from source 1 to target 3"

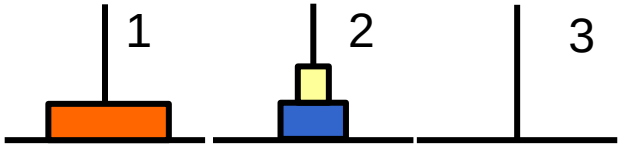


"Move disk 2 from source 1 to target 2"



THanoi(1 , 3, 2, 1)

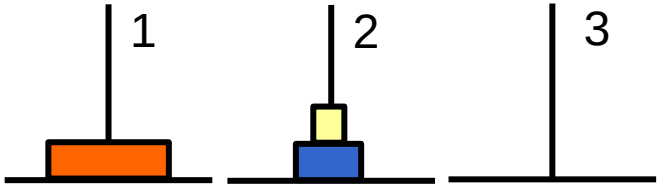
"Move disk 1 from source 3 to target 2"



"Move disk 3 from source 1 to target 3"

THanoi(2 , 2, 3, 1)

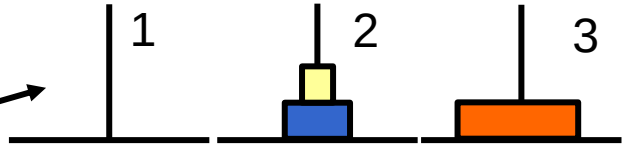
Exercise: Tower of Hanoi




```
def THanoi(n , source, target, aux):
    if n==1:
        print("Move disk 1 from source",source,"to target", target)
        return
    THanoi(n-1, source, aux, target)
    print("Move disk",n,"from source",source,"to target", target)
    TorresHanoi(n-1, aux, target, source)
```

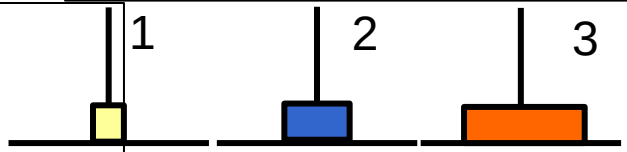
THanoi(3 , 1, 3, 2)

THanoi(2 , 1, 2, 3)
"Move disk 3 from source 1 to target 3"
THanoi(2 , 2, 3, 1)

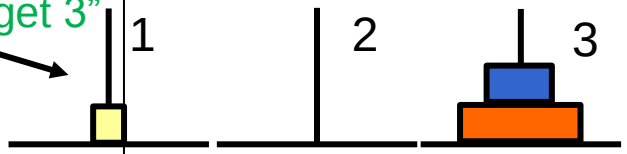


"Move disk 1 from source 2 to target 1"

THanoi(1 , 2, 1, 3)

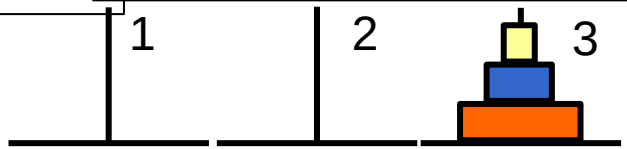


"Move disk 2 from source 2 to target 3"



THanoi(1 , 1, 3, 2)

"Move disk 1 from source 1 to target 3"



Recursion: Summary

Recursion: it is a programming technique in which a function calls to itself

Conditions for applying recursion:

- The problem can be solved by reducing the problem into simpler versions of the same problem (divide and conquer).
- The problem must have 1 or more base cases that are easy to solve.

Advantages:

- Elegant solutions and simpler solutions (easy to understand).
- Ideal for those "almost" unsolvable problems with iterative structures.

Disadvantages:

- Less eficient: it usually requires more computation time and RAM (memory)
- Some solutions require to repeat some computations (e.g. Fibonacci)
- Computers are prepared for iterative solutions (compilers convert recursive algorithms into iterative ones).