Programming For Data Science

Part Seven:
Functions, functions, functions!

Giulio Rossetti
giulio.rossetti@isti.cnr.it

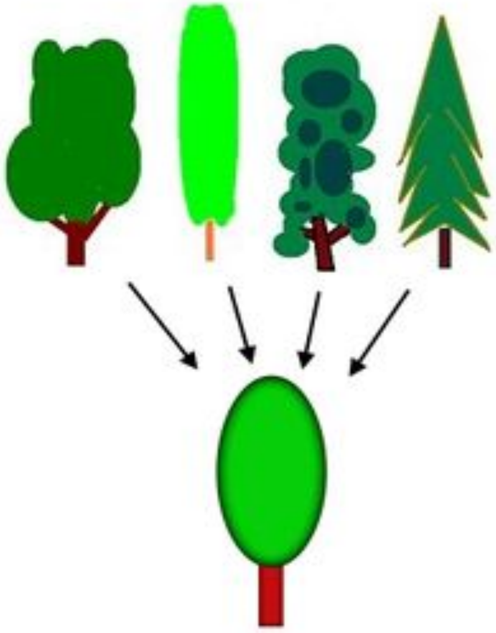
Credits: Giuseppe Prencipe
giuseppe.prencipe@unipi.it

Functions….why?

Manageability

Use code without knowing the specifics

Code useful in different situation via parameters



Generalisation

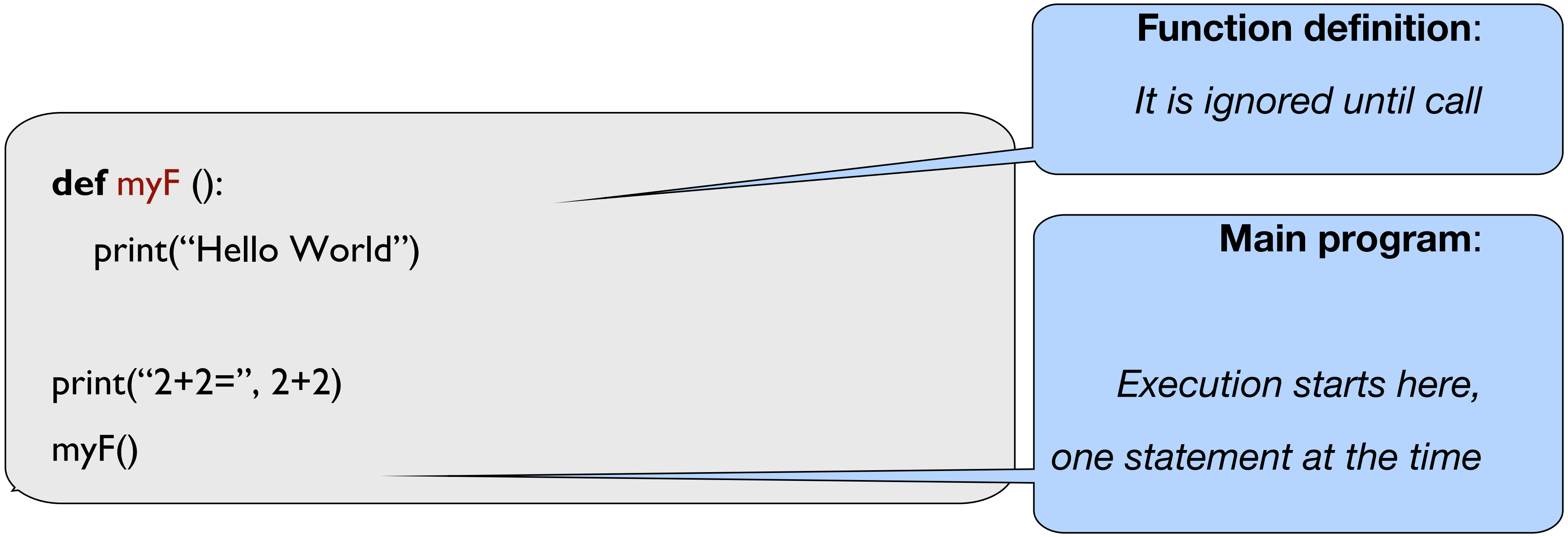
Create

- Use general conventions for name
- **Definition before use**
    - Conventions: place definitions at the top

```
def name (parameters):
    statements
```

Create

- Use general conventions for name
- **Definition before use**
    - Conventions: place definitions at the top



```
def myF ():
   print(“Hello World”)


print(“2+2=”, 2+2)
myF()
```

Function definition:

It is ignored until call

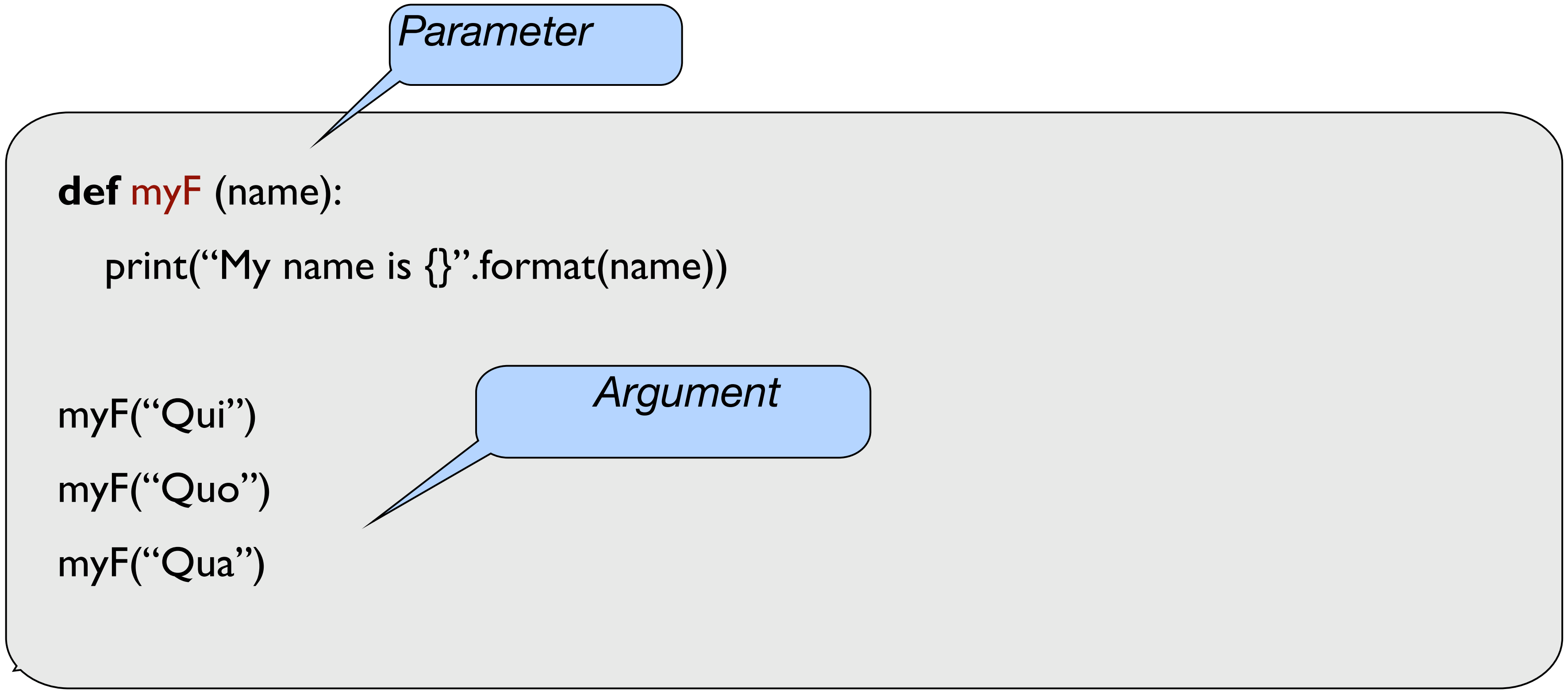
Main program:

Execution starts here,

one statement at the time

Parameters

- Parameters in functions are **local** to the function
- A value for the parameter is provided when the function is called



```
def myF (name):
    print(“My name is {}”.format(name))

myF(“Qui”)
myF(“Quo”)
myF(“Qua”)
```

Parameters

- Parameters in functions are **local** to the function
- A value for the parameter is provided when the function is called

```
def myF (name, lastname):
    print(“My name is {} {}”.format(name, lastname))


myF(“Paperon”, “de’ Paperoni”)
```

Careful with types

- You might check the values, before using them
  - **isinstance(var, type)**
  - For now we can assume types to be OK

```
# assume we have a variable a
if isinstance(a, int)
    print(“integer”)
if isinstance(a, float)
    print(“float”)
if isinstance(a, str)
    print(“string”)
else:
    print(“!!!!”)
```

Default values

It is possible to have default values for parameters

```
def myF (name, lastname = “None”, nick = “Avenger”):
    print(“My name is {} {} aka {}”.format(name, lastname, nick))


myF(“Paperon”, “de’ Paperoni”)
myF(“Spiderman”)
myF(“Spiderman”, lastname = “”)
```

Return value

- A function communicates to the outside using the keyword **return**
  - Ends the execution of the function and returns the control to the main program

```
def mul (x,y):
    return x*y

a = mul(5, 5)
print(a)
```

Return value

If no return is reached during the execution of the function, at the end it returns anyway, with no value

```
def mul (x,y):
    if (x>0 and y>0):
        return x*y

a = mul(5, 5)
print(a)
a = mul(5, -5)
print(a)
```

What do you get here?

return vs print

It should be clear, right?

```
def return3 ():
    return 3


x=2**return3()
print(x)
```

vs

```
def return3 ():
    print(3)


x=2**return3()
print(x)
```

What do you get here?

Multiple return value

Multiple values can be returned

```
def  multiple_op(x, y):
    return x+y, x*y


a, b = multiple_op(2,3)
print(“result: {} and {}”.format(a, b))
```

Other facts

Functions can call other functions

```
def mul(m, n):
        return m*n

def  multiple_op(x, y):
    return x+y, mul(x,y)

a, b = multiple_op(2,3)
print(“result: {} and {}”.format(a, b))
```

Other facts

Functions can be defined within other functions

```
def  multiple_op(x, y):
    def mul(m, n):
        return m*n
    return x+y, mul(x,y)

a, b = multiple_op(2,3)
print(“result: {} and {}”.format(a, b))
```

Scope

Scope refers to visibility (within block and nested ones)

Program level

```
f = 101
print(f)

def some_function():
    f = “I am having so much fun!”
    print(f)

some_function()
print(f)
```

Function level

```
101
I am having so much fun!
101
```

Scope

Python is ‘easy’ with scope

```
for j in range (3):
    print(“I am having so much fun….{}”.format(j))

some_function()
print(j)
```

Block level

Out of the block!

```
I am having so much fun….0
I am having so much fun….1
I am having so much fun….2
2
```

Scope

….but not so ‘easy’

Function level

```
def some_function():
    for j in range (3):
        print(“I am having so much fun….{}”.format(j))

some_function()
# print(j)
```

j not defined here!

```
I am having so much fun….0
I am having so much fun….1
I am having so much fun….2
```

Scope

….but not so ‘easy’

Global

```
value = 2
```

Modified

```
def add_value(x):
    value = 4
    print(x + value)
```

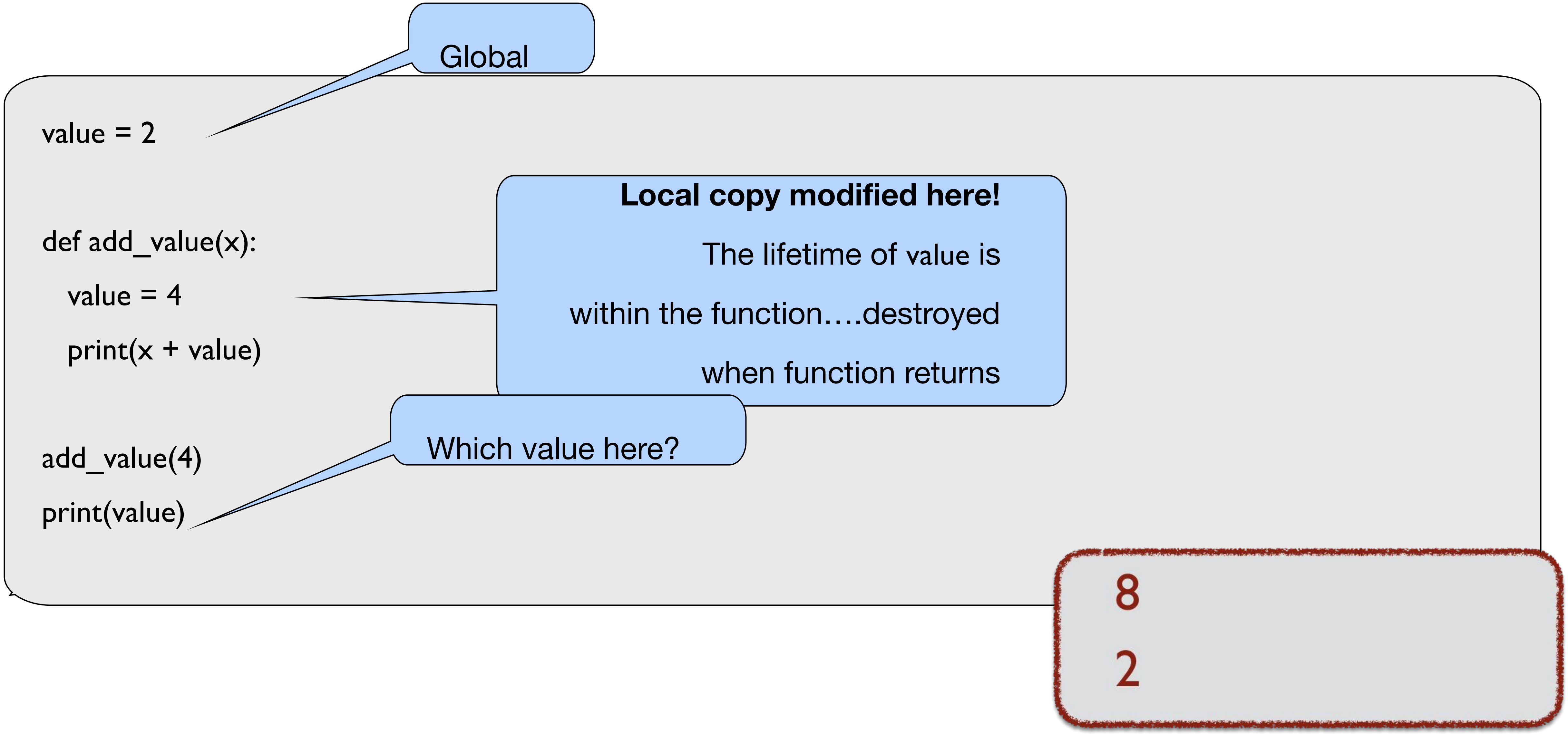
Which value here?

```
add_value(4)
print(value)
```

```
8
2
```

Scope

….but not so 'easy'



```
value = 2


def add_value(x):
   value = 4
   print(x + value)


add_value(4)
print(value)
```

Global

Local copy modified here!

The lifetime of value is

within the function….destroyed

when function returns

Which value here?

```
8
2
```

Scope

….but not so ‘easy’

```
value = 2


def add_value(x):
   value = 4
   print(x + value)


add_value(4)
print(value)
```

In general, **functions do not have to take into account variables that exist outside the function**, as any variable that they create is local to the function

Local vs global variables

- Variables from the main program are *global*
- Variables visible only to functions are *local*
- **Good practice**: do not have functions to use global functions
  - It would make it dependent on the main program
  - Use `return` to communicate with the main

Decomposition

- When solving a problem, always think to decomposition

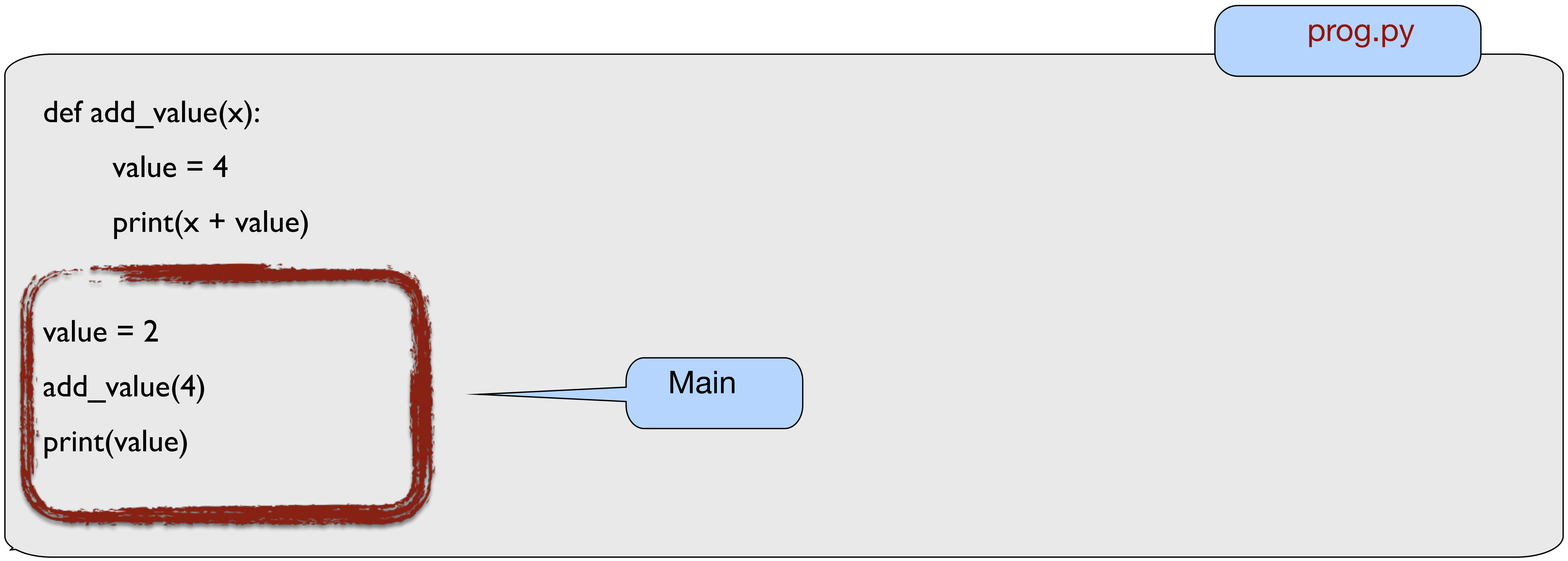
- Function are the right tool to achieve this
  - **Read in depth Section 8.4**

Decomposition

Modules

- .py files
- Same folder of your main or default Python folder for modules

Modules



```
def add_value(x):
        value = 4
        print(x + value)

value = 2
add_value(4)
print(value)
```

Modules

prog.py

```
def add_value(x):
        value = 4
        print(x + value)

def main():
    value = 2
    add_value(4)
    print(value)


if __name__ == '__main__':
    main()
```

Main

Before executing the program, the interpreter defines few special variables

If the file is executed as a whole program, __name__ is set to the value __main__ and main() executed

Otherwise main() is ignored

Modules

prog.py

```
def add_value(x):
        value = 4
        print(x + value)

def main():
    value = 2
    add_value(4)
    print(value)


if __name__ == '__main__':
    main()
```

If prog.py used as module, this can be
used for testing purposes

Before executing the program, the interpreter defines few special variables

If the file is executed as a whole program, __name__ is set to the value __main__ and main() executed

Otherwise main() is ignored

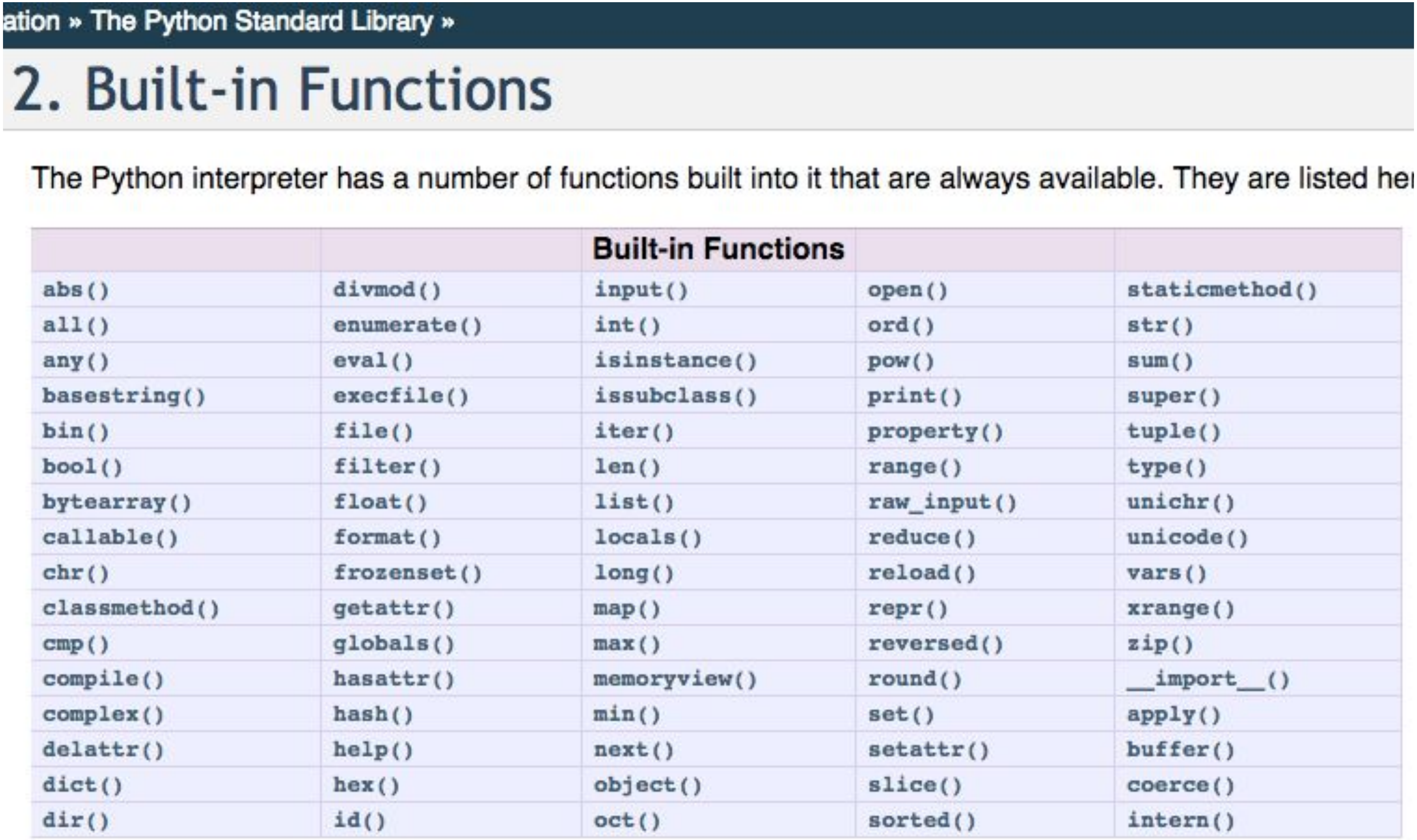
Modules — an example

prog.py

```
def add_value(x):
        value = 4
        print(x + value)

def main():
    value = 2
    add_value(4)
    print(value)


if __name__ == '__main__':
    main()
```

```
import prog

prog.add_value(2)
```

```
from prog import add_value

add_value(2)
```



ation » The Python Standard Library »

2. Built-in Functions

The Python interpreter has a number of functions built into it that are always available. They are listed he

		Built-in Functions		
`abs()`	`divmod()`	`input()`	`open()`	`staticmethod()`
`all()`	`enumerate()`	`int()`	`ord()`	`str()`
`any()`	`eval()`	`isinstance()`	`pow()`	`sum()`
`basestring()`	`execfile()`	`issubclass()`	`print()`	`super()`
`bin()`	`file()`	`iter()`	`property()`	`tuple()`
`bool()`	`filter()`	`len()`	`range()`	`type()`
`bytearray()`	`float()`	`list()`	`raw_input()`	`unichr()`
`callable()`	`format()`	`locals()`	`reduce()`	`unicode()`
`chr()`	`frozenset()`	`long()`	`reload()`	`vars()`
`classmethod()`	`getattr()`	`map()`	`repr()`	`xrange()`
`cmp()`	`globals()`	`max()`	`reversed()`	`zip()`
`compile()`	`hasattr()`	`memoryview()`	`round()`	`__import__()`
`complex()`	`hash()`	`min()`	`set()`	`apply()`
`delattr()`	`help()`	`next()`	`setattr()`	`buffer()`
`dict()`	`hex()`	`object()`	`slice()`	`coerce()`
`dir()`	`id()`	`oct()`	`sorted()`	`intern()`

(Suggested) Exercises (1)

- Write and test a function isEven()
- Write and test a function isOdd(), which determines whether a number is odd, by calling the function isEven() and inverting its result
- Write and test a function getFraction() that gets the fractional part of a float (i.e., the decimals)
- Write and test a function that returns the number of days in A years, B months and C weeks?
- Define a function called distance_from_zero, with one argument. If the type of the argument is either int or float, the function returns the absolute value of the input. Otherwise, the function returns “Nope”

(Suggested) Exercises (2)

- **All** exercises in Chapter 8 of the reference book
- In particular:
  - 8.2
  - 8.4
  - 8.6
  - 8.7

Programming For Data Science

Part Eight:

Alla fiera dell'est!

— or how to wash dishes using recursion —

Giulio Rossetti
giulio.rossetti@isti.cnr.it

Credits: Giuseppe Prencipe
giuseppe.prencipe@unipi.it

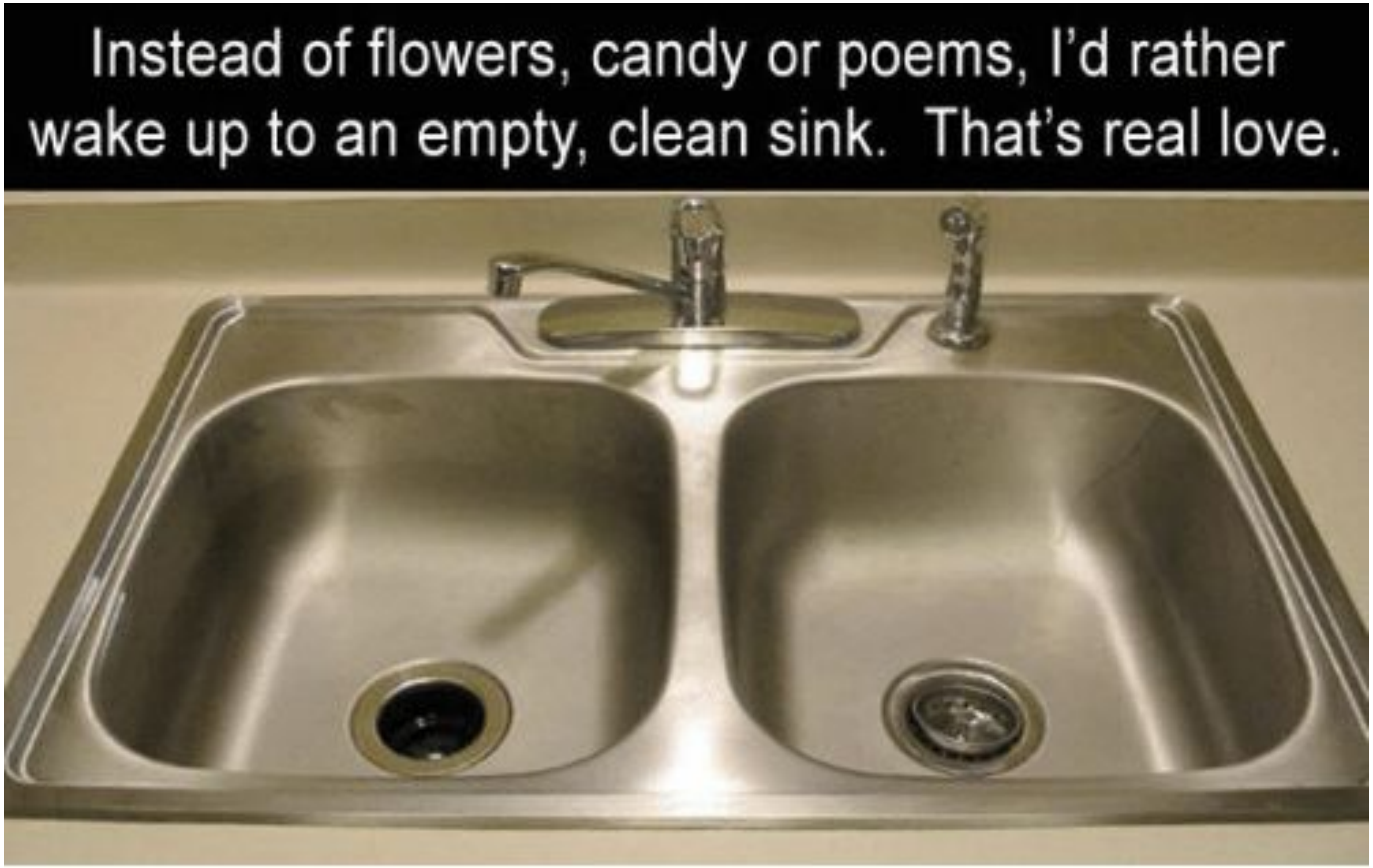
Recursion

Definition: an *object* is called recursive if it is possible to *express it in terms of itself*

Let’s see few examples….

Wash your dishes!

If the sink is empty, that’s it!

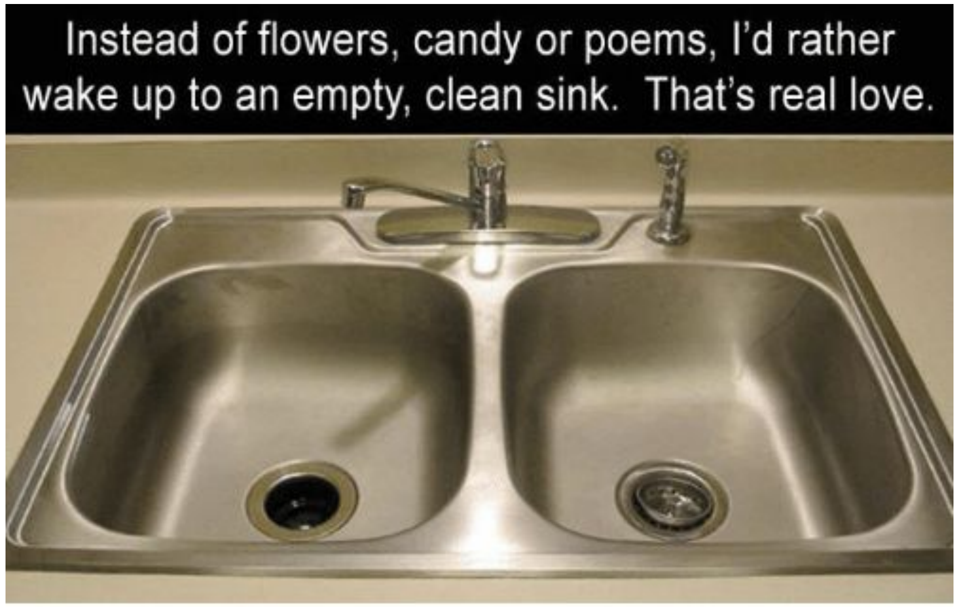

Instead of flowers, candy or poems, I’d rather wake up to an empty, clean sink. That’s real love.

Wash your dishes!

If it's not empty:

- Wash the first dish
- Ask someone else to wash the other dishes….
  - Each person works on a smaller set
  - To wash 20 dishes, I wash one, and ask someone else to wash 19

Wash your dishes!

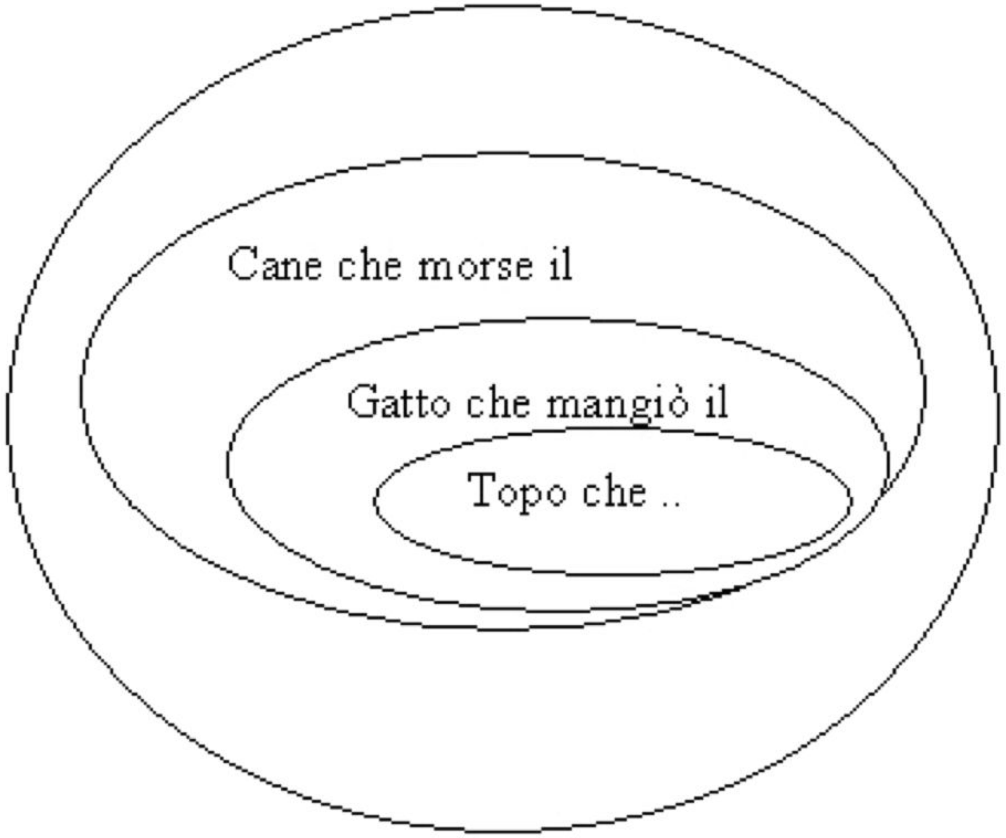

Instead of flowers, candy or poems, I'd rather
wake up to an empty, clean sink. That's real love.

If it’s not empty:

- Wash the first dish
- Ask someone else to wash the other dishes….
- Until sink is empty!

Alla fiera dell'est

Alla fiera dell'Est
Per due soldi un topolino
mio padre comprò
E venne il gatto che
si mangiò il topo che
al mercato mio padre comprò
E venne il cane che
Morse il gatto che
si mangiò il topo che
al mercato mio padre comprò
E venne il bastone che
Picchiò il cane che
Morse il gatto che
si mangiò il topo che
al mercato mio padre comprò
E venne il fuoco che
Bruciò il bastone che
Picchiò il cane che
morse il gatto che
si mangiò il topo che
al mercato mio padre comprò
Acqua … Bue ... Macellaio ... Angelo della Morte ...



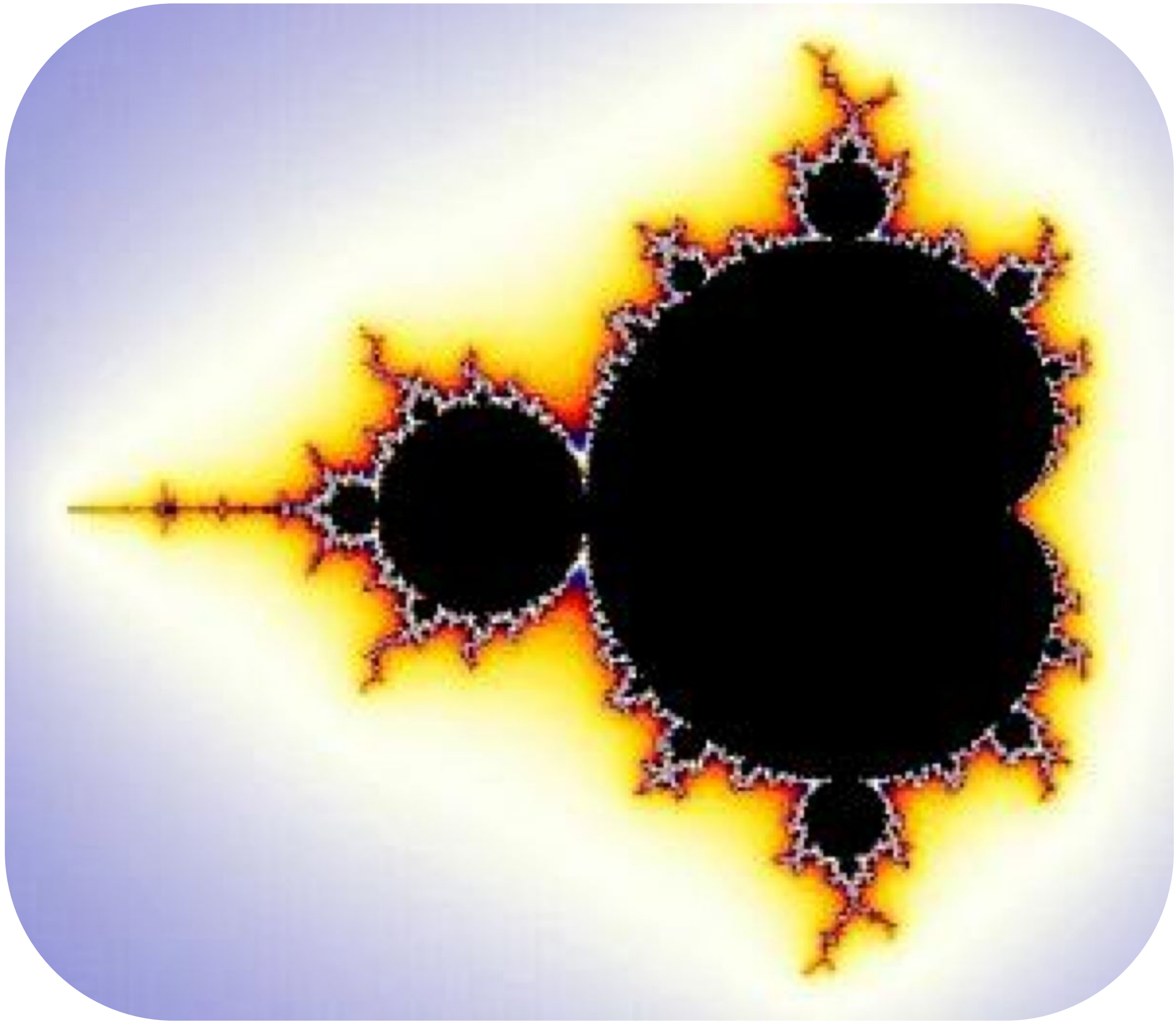
To generate a verse,
you generate two shorter verses

Visive recursion

It is possible to visualise recursion

- Matrioska
- Chinese box

Fractal



Math

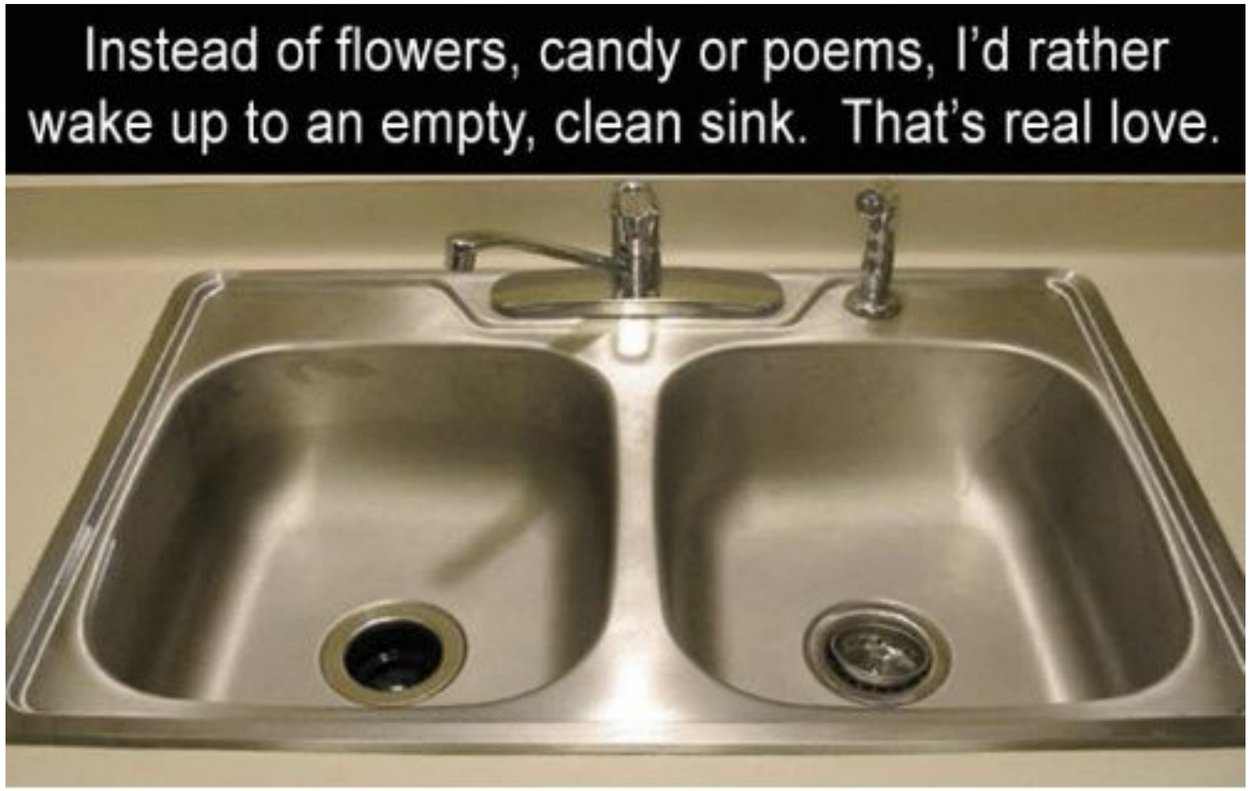
- In math, a recursive function is a function that calls itself
- Factorial is a classical example
  - n! = n * (n-1) * (n-2) * …. * 1 = n * (n-1)!

Recursion

2 main ingredients

- Base case

Sink is empty



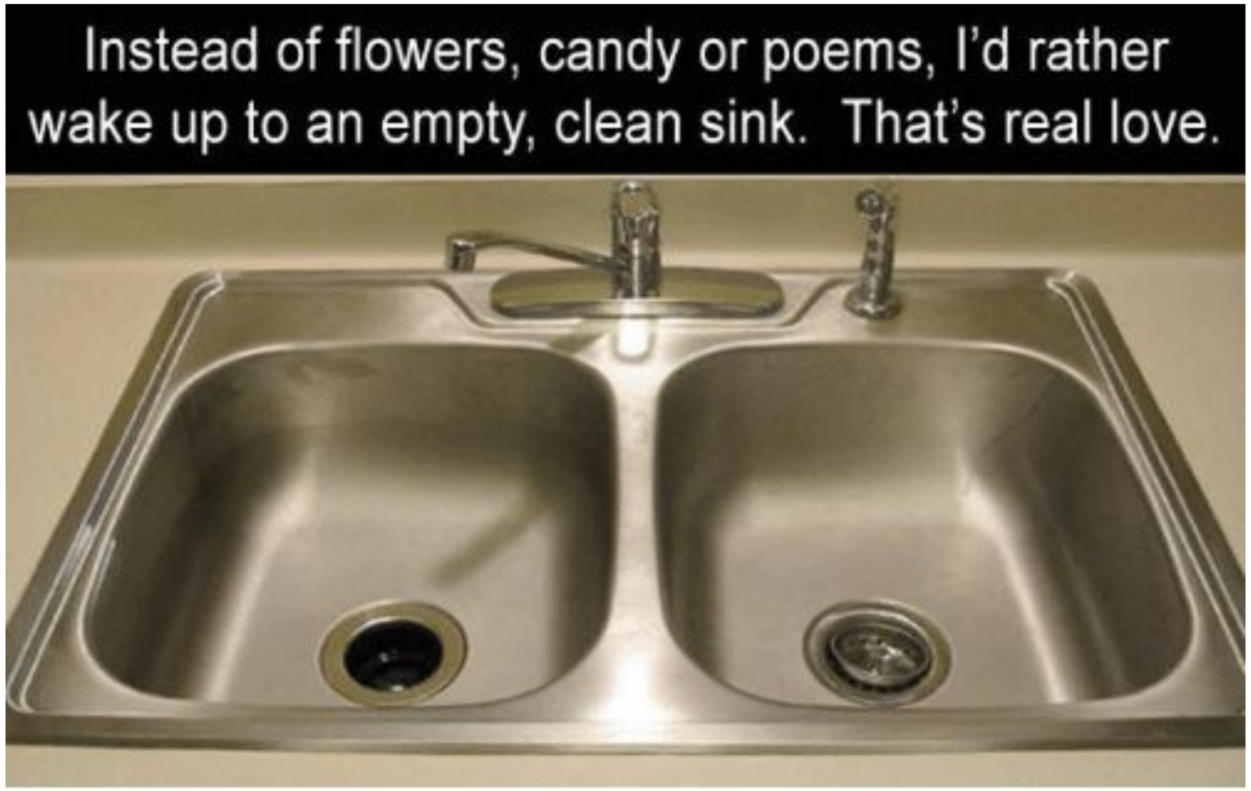
Instead of flowers, candy or poems, I'd rather wake up to an empty, clean sink. That's real love.

Recursion

2 main ingredients

- Base case
- Sub-Problem (decomposition)

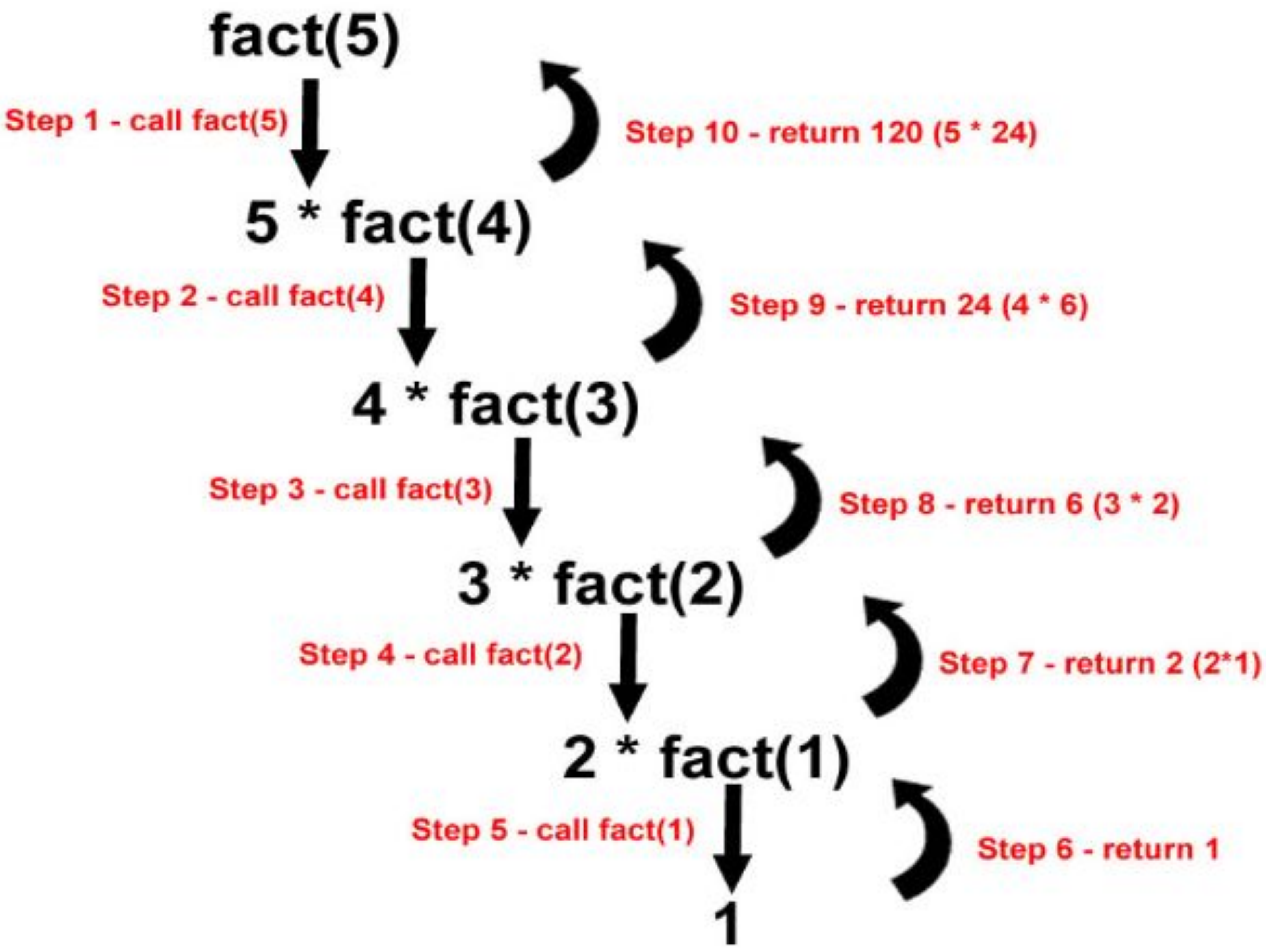
Sink is empty


Instead of flowers, candy or poems, I'd rather wake up to an empty, clean sink. That's real love.

To wash 20 dishes, I wash one, and ask someone else to wash 19

An example: factorial

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

print( factorial (5))
```



fact(5)

Step 1 - call fact(5)

5 * fact(4)

Step 2 - call fact(4)

4 * fact(3)

Step 3 - call fact(3)

3 * fact(2)

Step 4 - call fact(2)

2 * fact(1)

Step 5 - call fact(1)

1

Step 6 - return 1

Step 7 - return 2 (2*1)

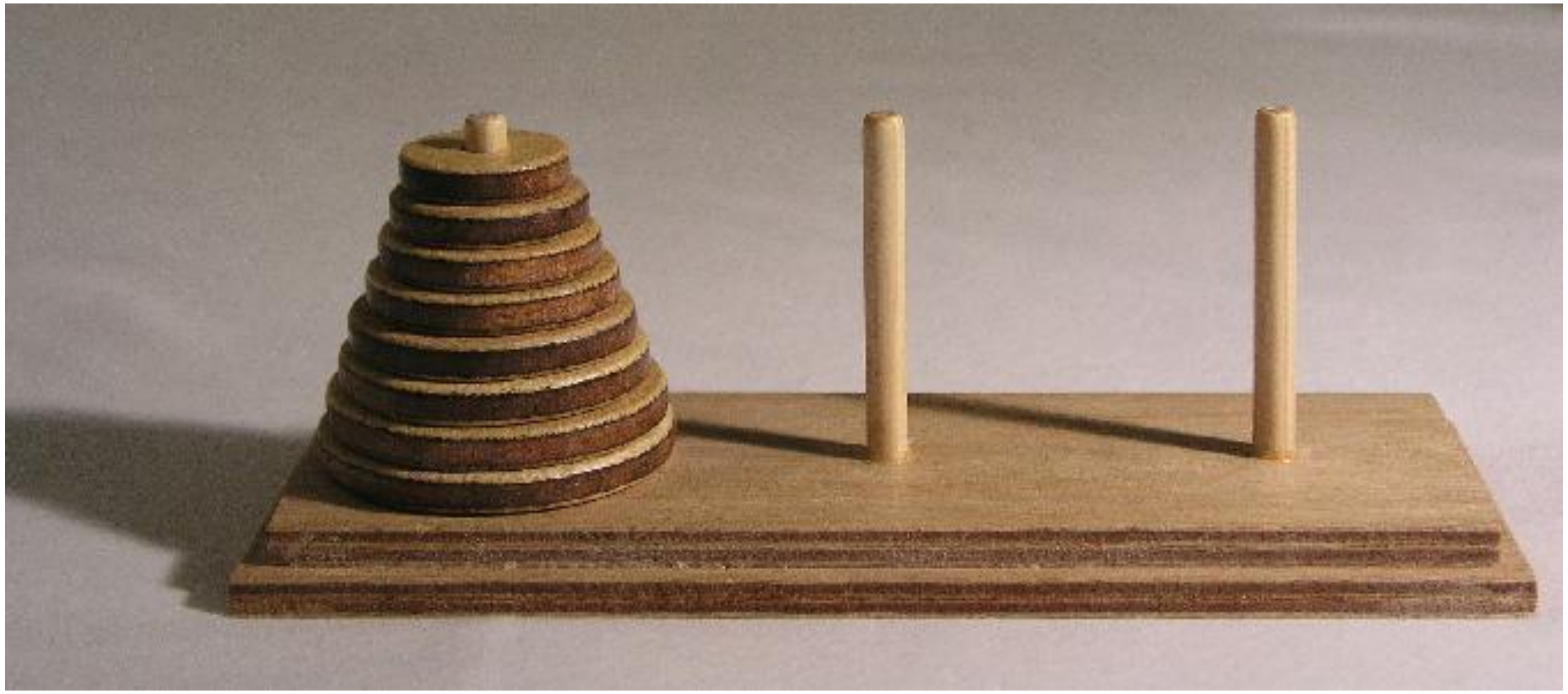
Step 8 - return 6 (3 * 2)

Step 9 - return 24 (4 * 6)

Step 10 - return 120 (5 * 24)

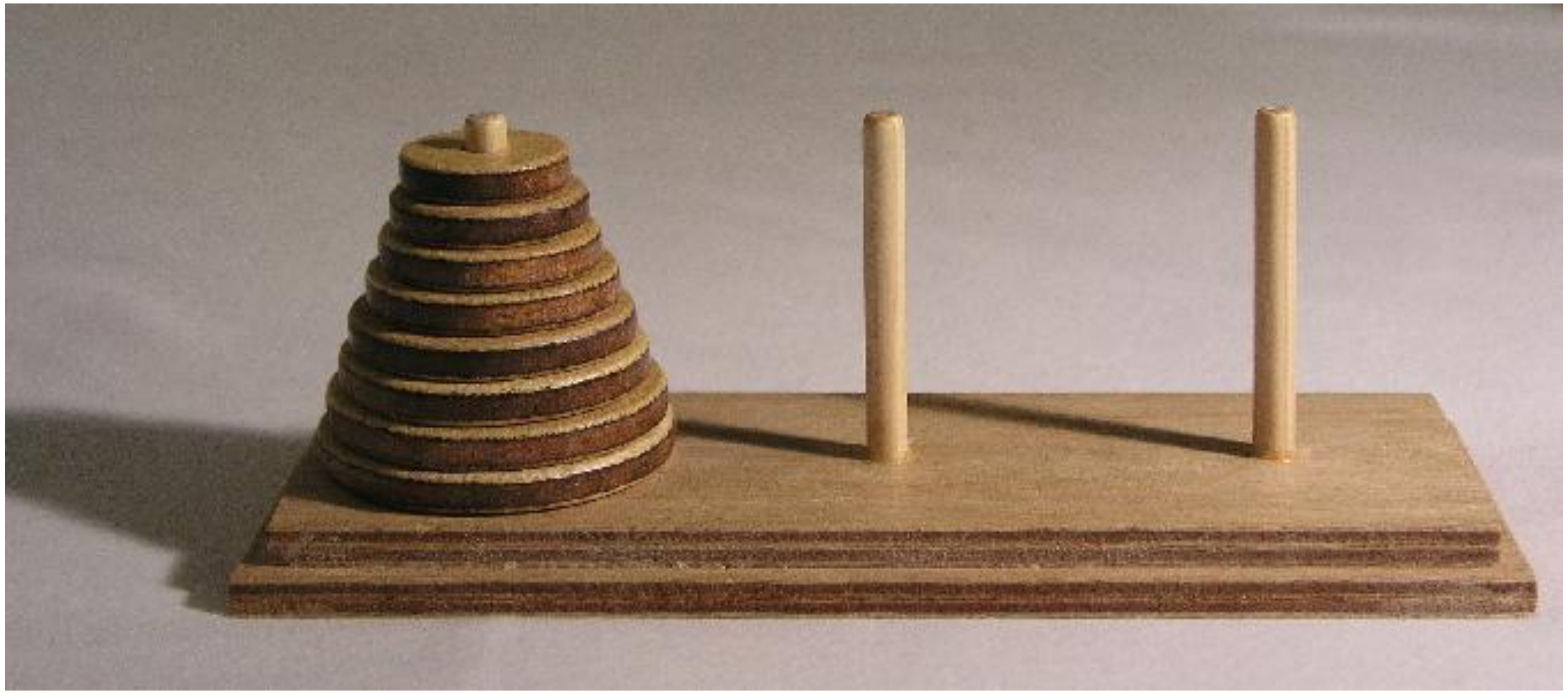
An example: Hanoi

- The objective of the puzzle is to move the entire stack to another rod, obeying the following simple rules:
    - Only one disk can 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.
    - No disk may be placed on top of a smaller disk.



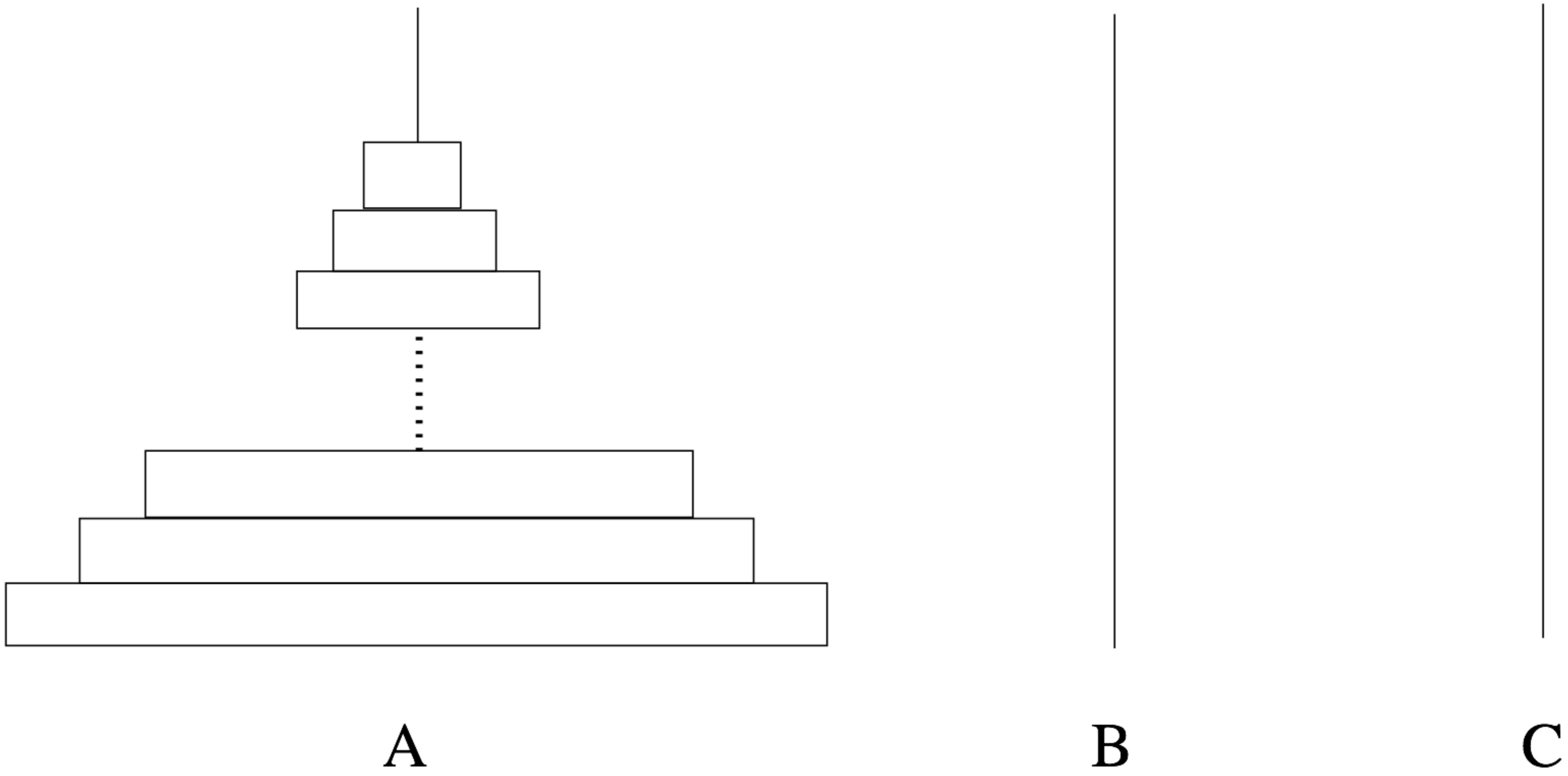
An example: Hanoi

The puzzle was invented by the French mathematician Édouard Lucas in 1883. There is a story about an Indian temple in Kashi Vishwanath which contains a large room with three time-worn posts in it, surrounded by 64 golden disks. Brahmin priests, acting out the command of an ancient prophecy, have been moving these disks in accordance with the immutable rules of Brahma since that time. The puzzle is therefore also known as the Tower of Brahma puzzle. According to the legend, when the last move of the puzzle is completed, the world will end.



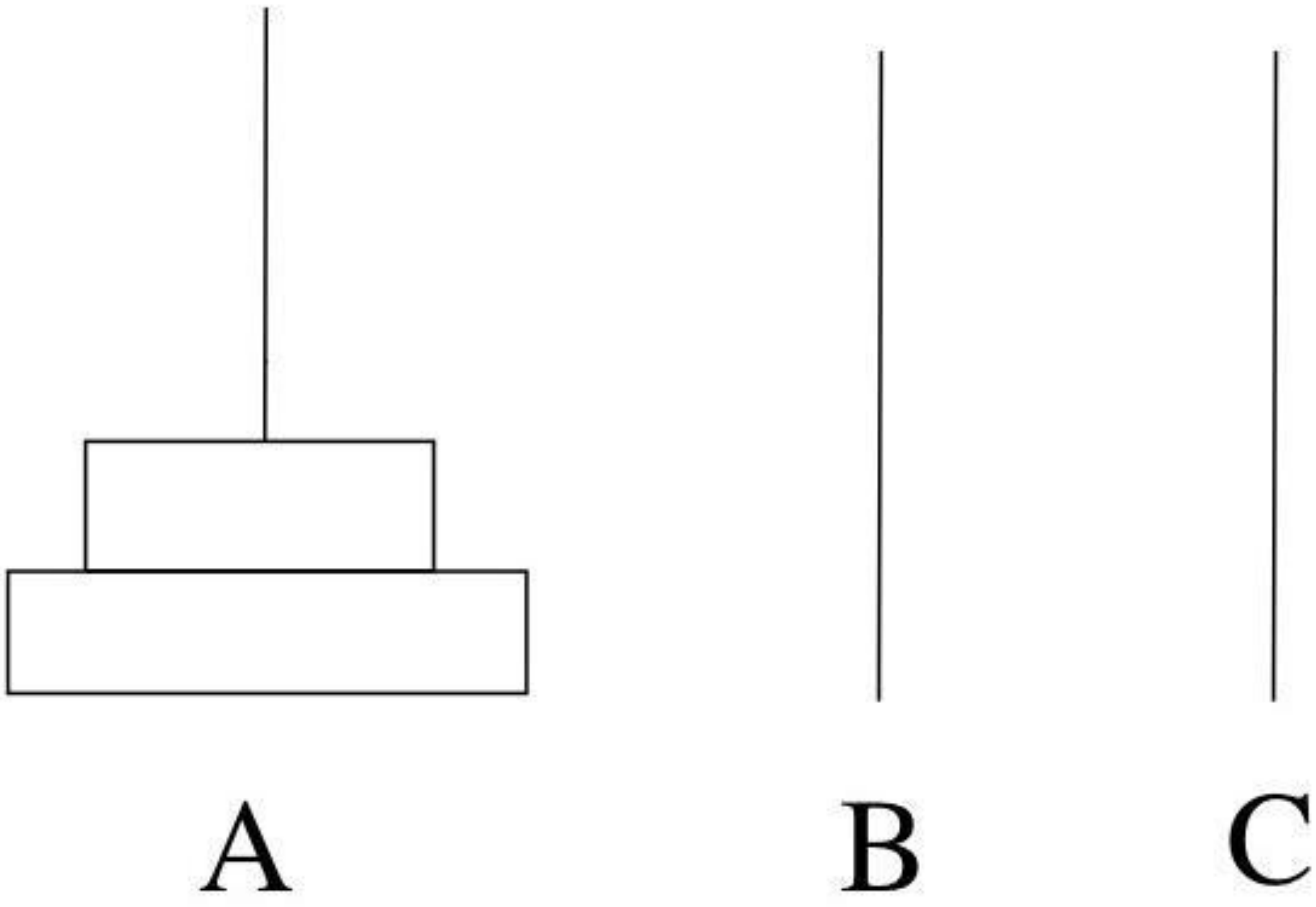
An example: Hanoi

- For N=1, easy!
  - Move the only disk from A to C, and that’s it!



An example: Hanoi

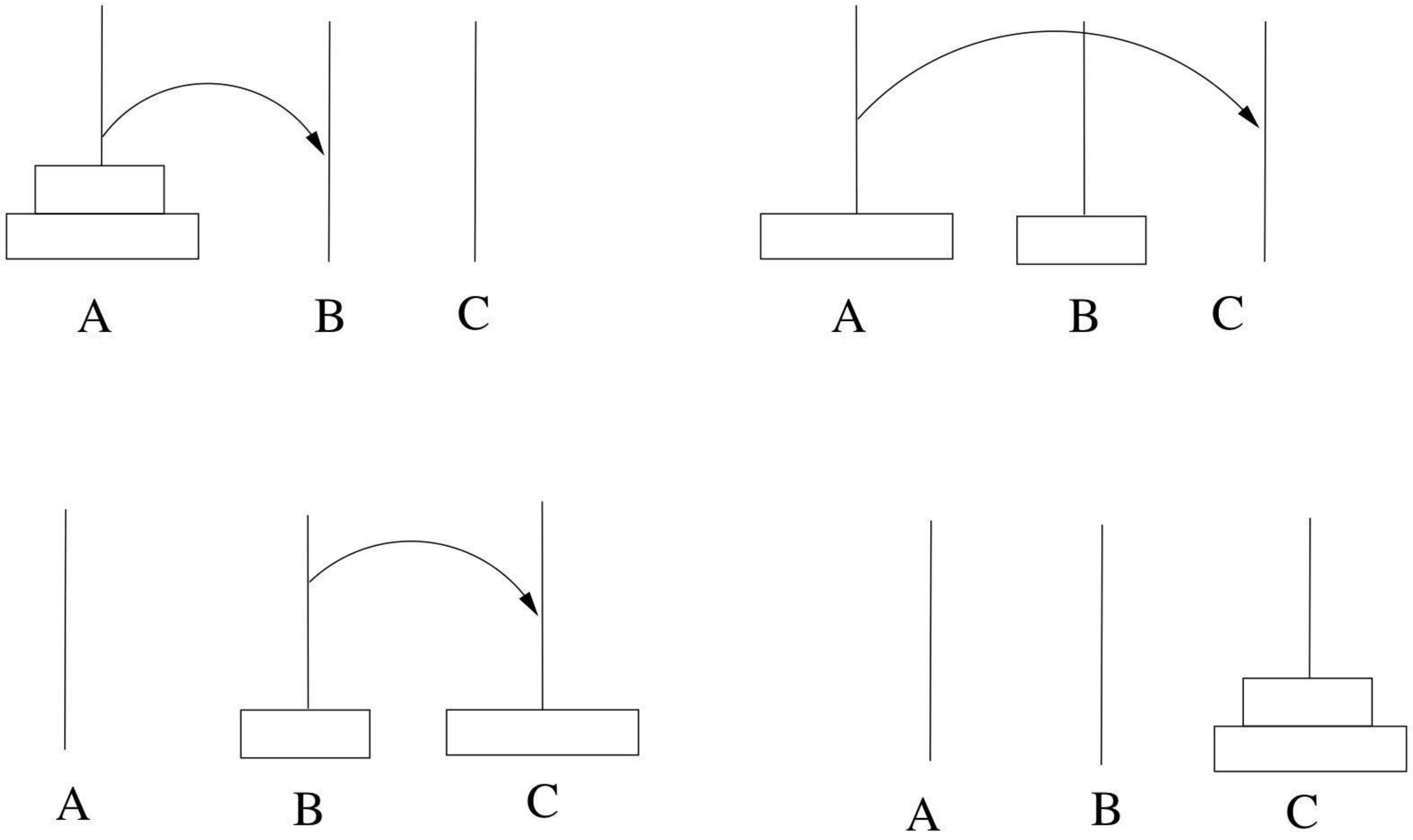
Knowing the solution for N=1, can we solve it for N = 2?



An example: Hanoi

Knowing the solution for N=1, can we solve it for N = 2?

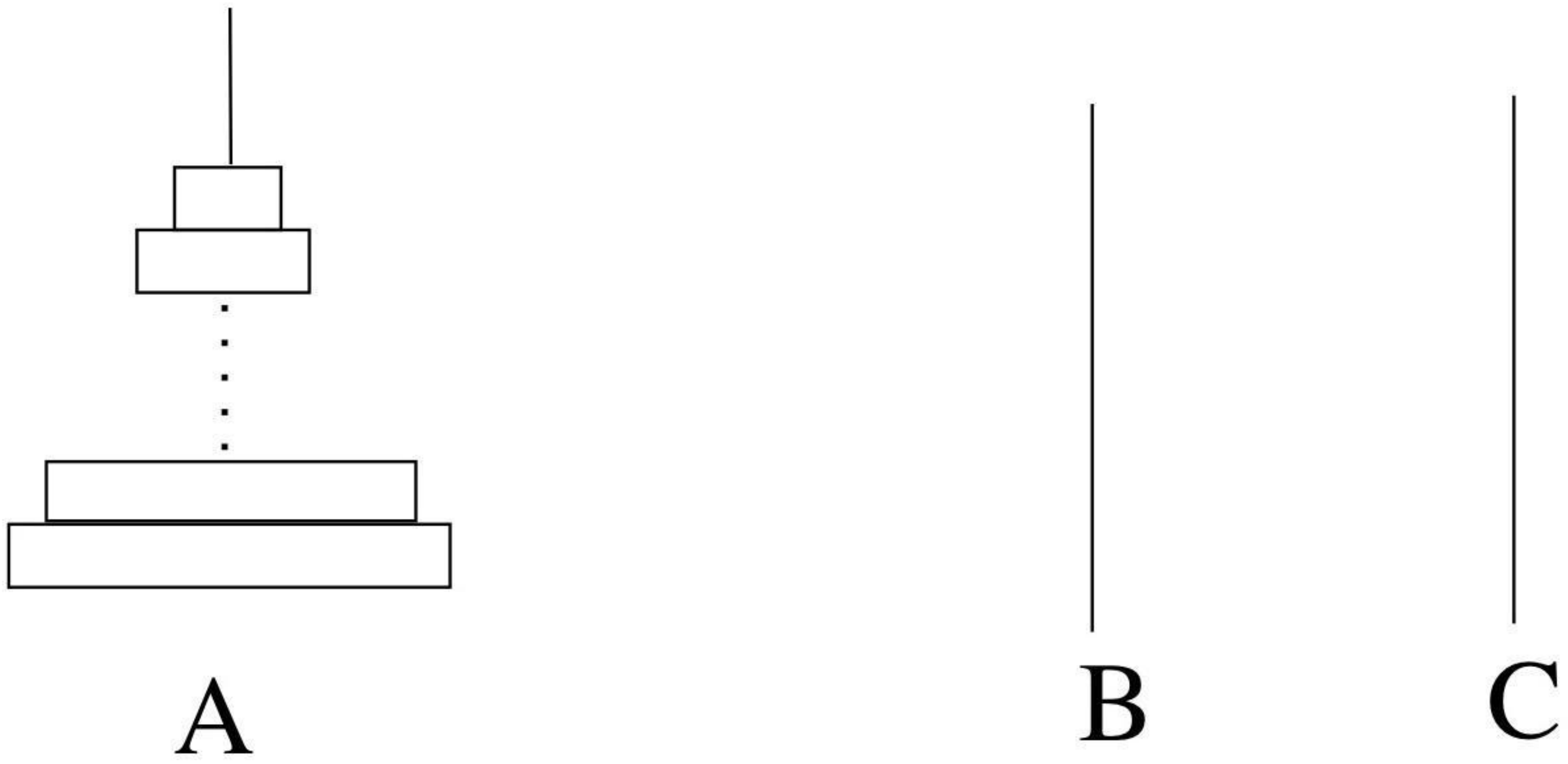
- Use B as support



An example: Hanoi

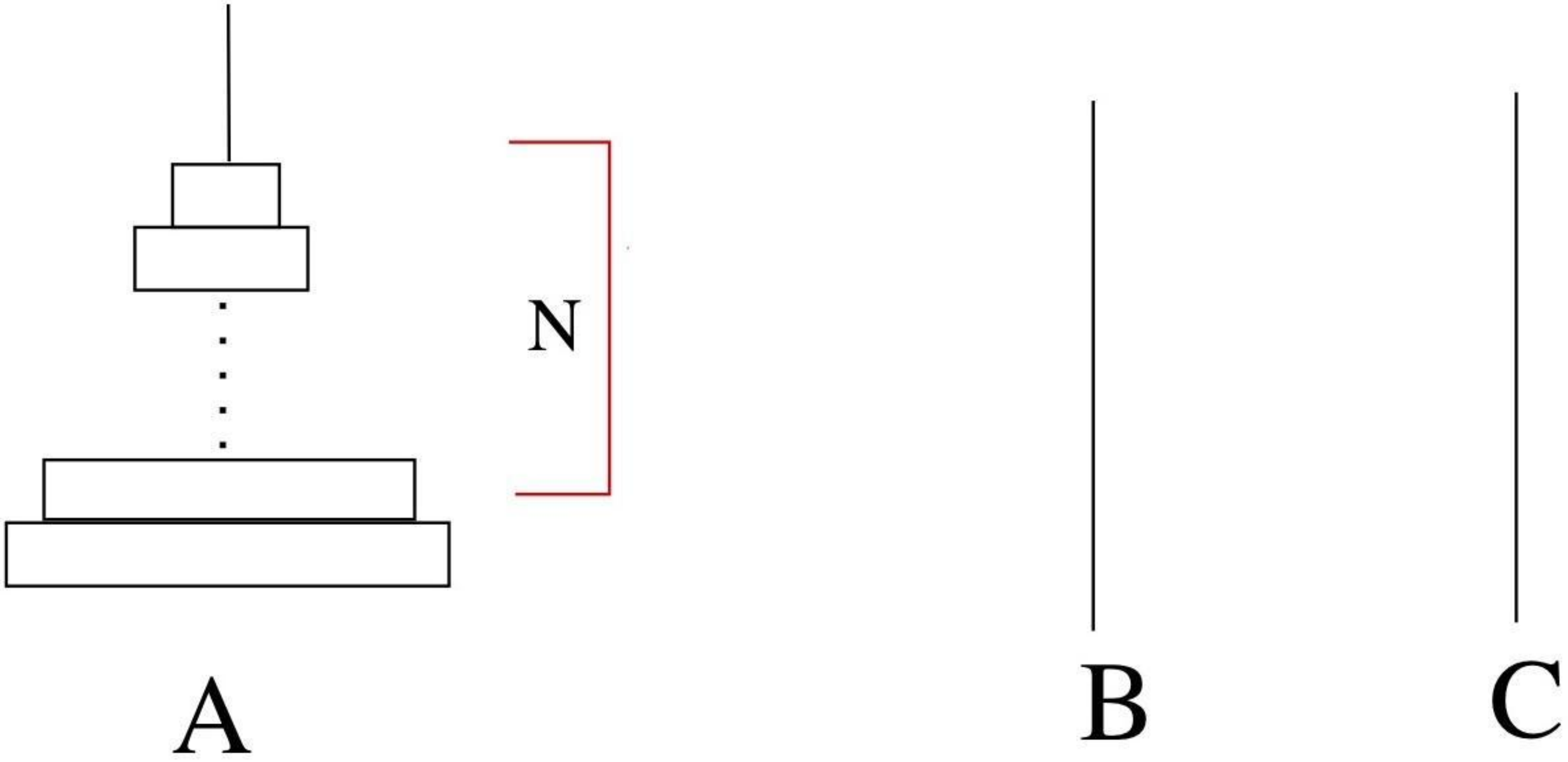
Can we generalise for N > 2?

○ Hint: use recursive argument!



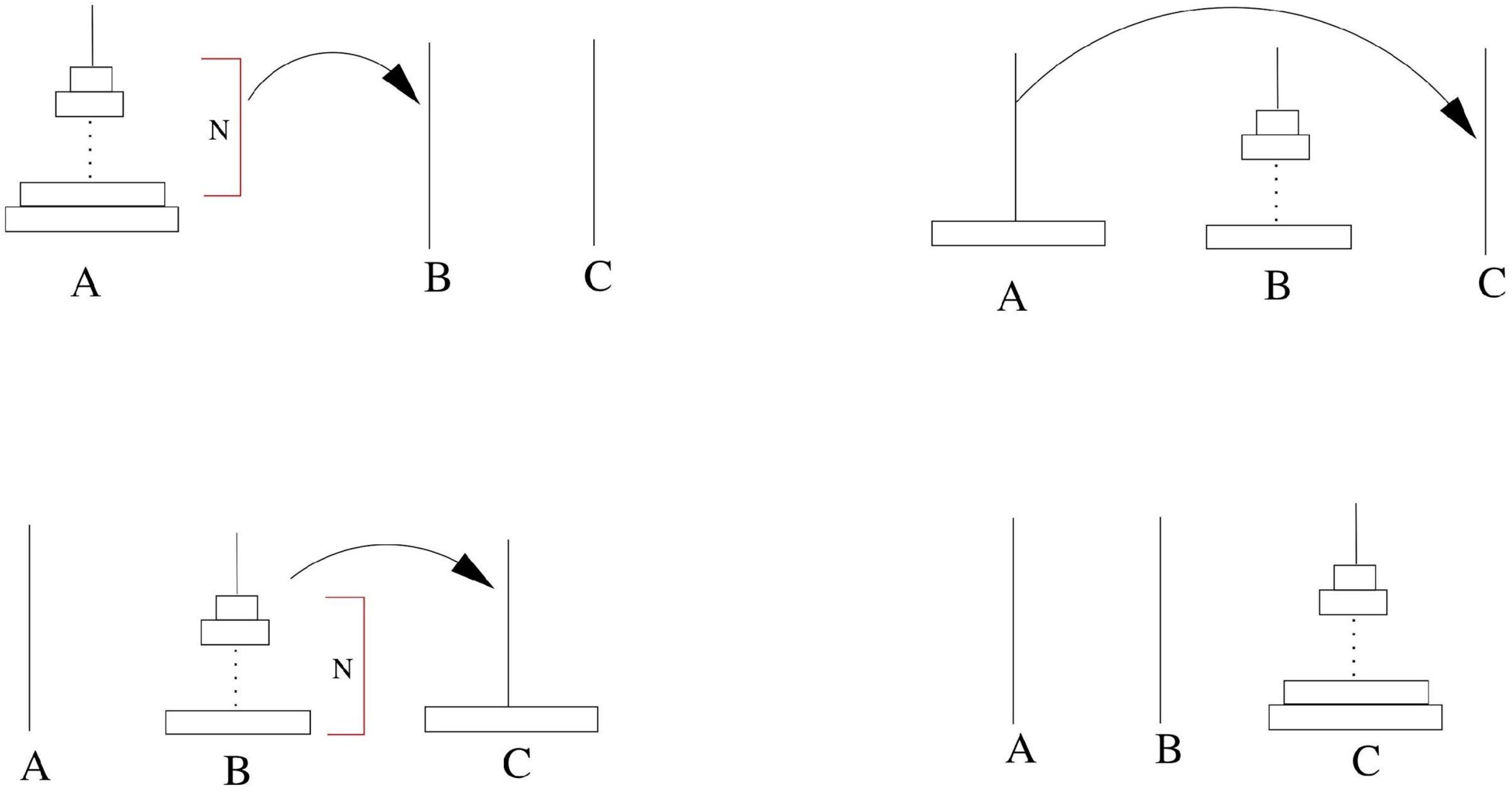
An example: Hanoi

- Can we generalise for N > 2?
  - If we could solve it for N, can we solve it for N+1?



An example: Hanoi

- Can we generalise for N > 2?
  - If we can solve it for N, we can for N+1!

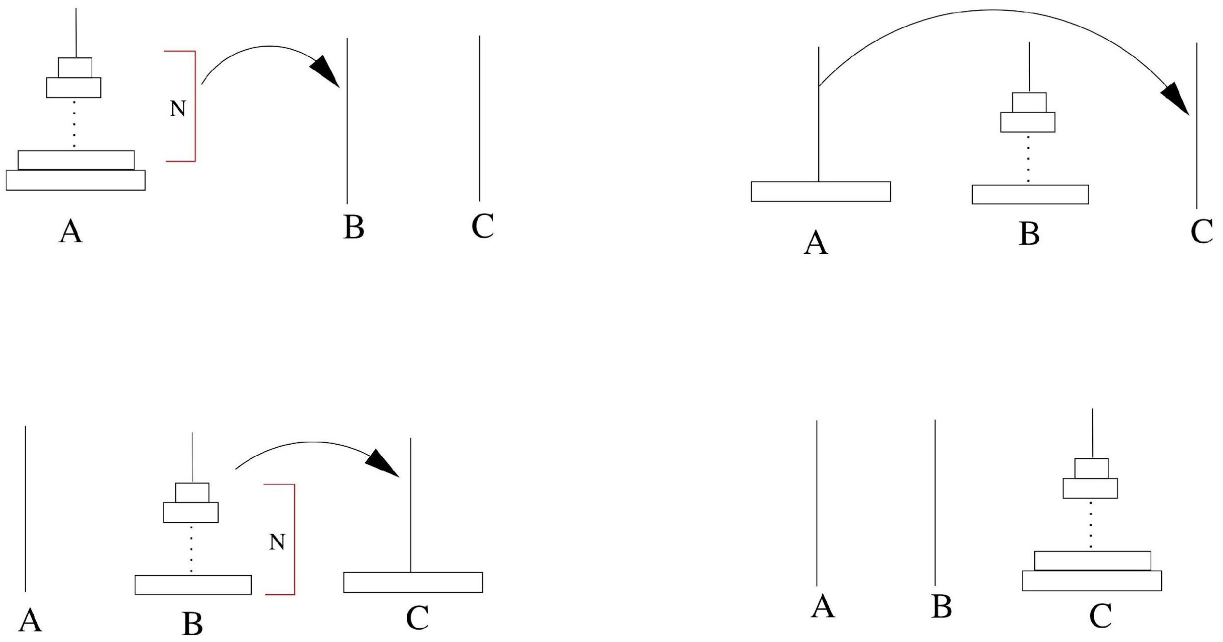


An example: Hanoi

move n disks from

A to C, using B

```
def hanoi(n, A, B, C):
    if n ==1:
        move from A to C
    else:
        hanoi(n-1, A, ?, ?)
        move from A to C
        hanoi(n-1, ?, ?, ?)
```

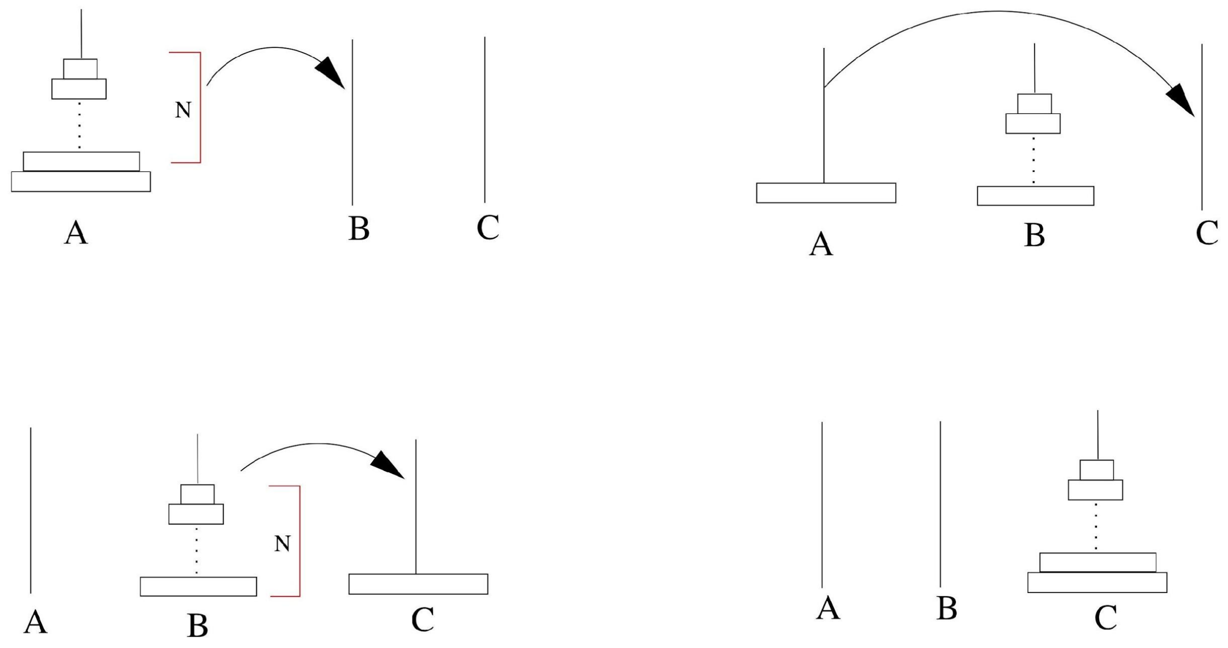


An example: Hanoi

move n disks from

A to C, using B

```
def hanoi(n, A, B, C):
    if n ==1:
        move from A to C
    else:
        hanoi(n-1, A, C, B)
        move from A to C
        hanoi(n-1, B, ?, ?)
```

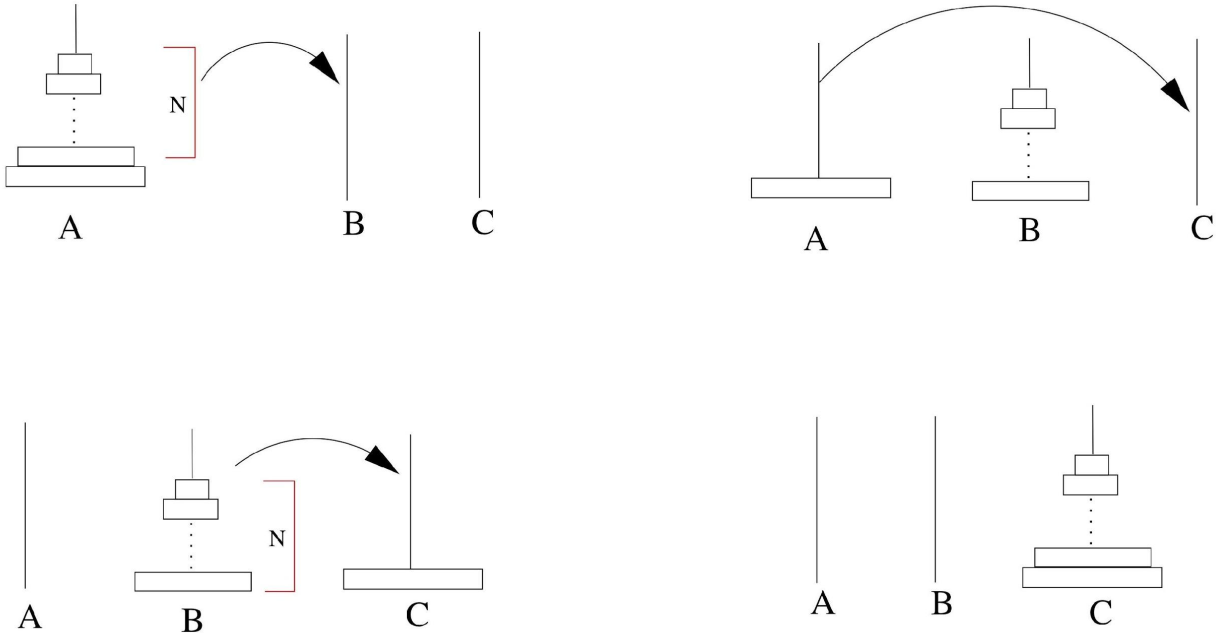


An example: Hanoi

move n disks from

A to C, using B

```
def hanoi(n, A, B, C):
    if n ==1:
        move from A to C
    else:
        hanoi(n-1, A, C, B)
        move from A to C
        hanoi(n-1, B, ?, ?)
```

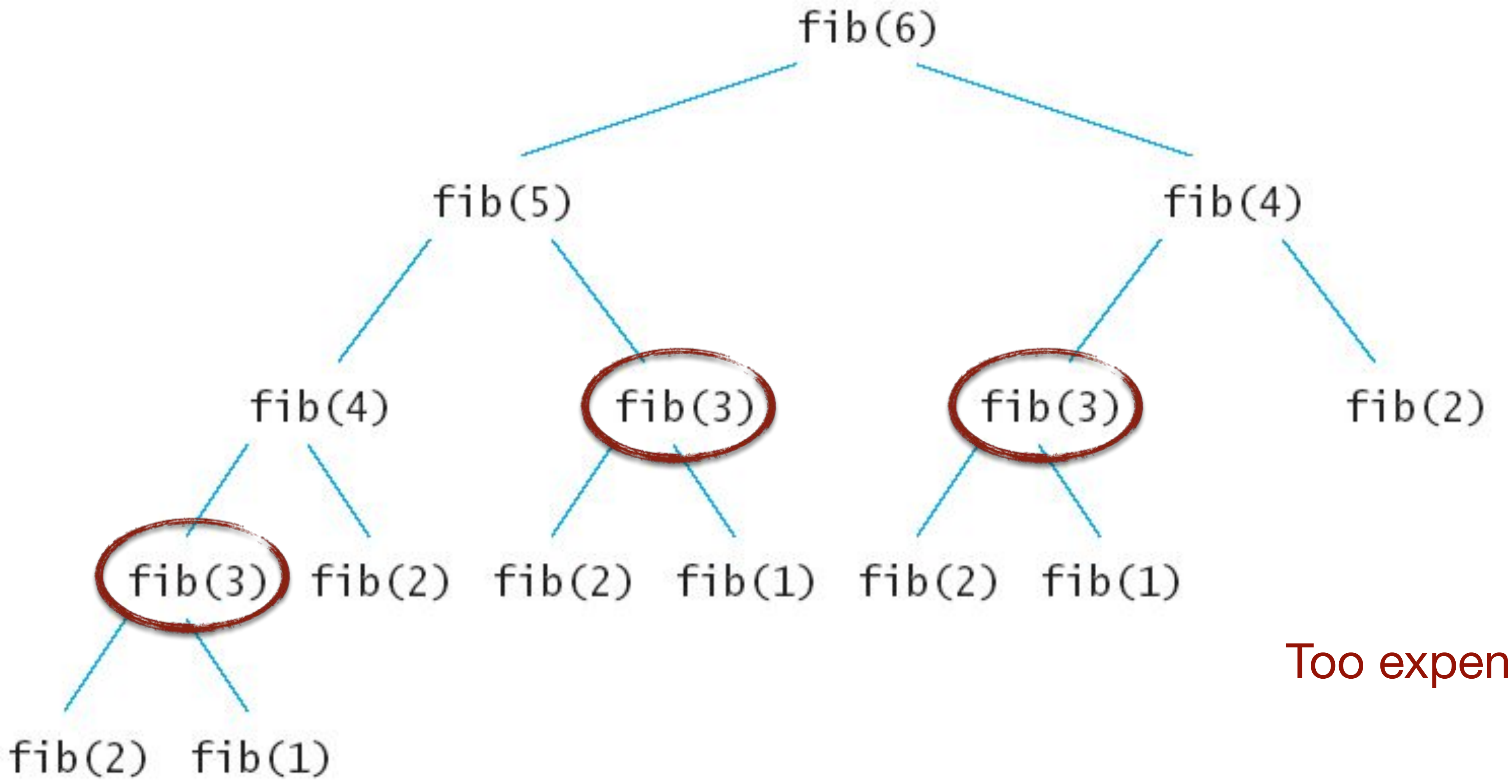


580 billions of years for 64 disks!

- 1 move, if N == 1
- 2^N - 1 moves, if N > 1

An example: Fibonacci

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



Too expensive!

An example: Fibonacci

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

Too expensive!

Recurrence Equation

$$C(n) = C(n-1) + C(n-2) = [C(n-2) + C(n-3) + 1] + [C(n-3) + C(n-4) + 1] = \ldots =\sim ((1+\sqrt{5})/2)^n$$

Exponential!

An example: Fibonacci

```
def fib(n):
    a, b = 0, 1
    for i in range(0, n):
        a, b = b, a + b
    return a
```

Iterative solution!

C(n) = ??

An example: Fibonacci

```
def fib(n):
    a, b = 0, 1
    for i in range(0, n):
        a, b = b, a + b
    return a
```

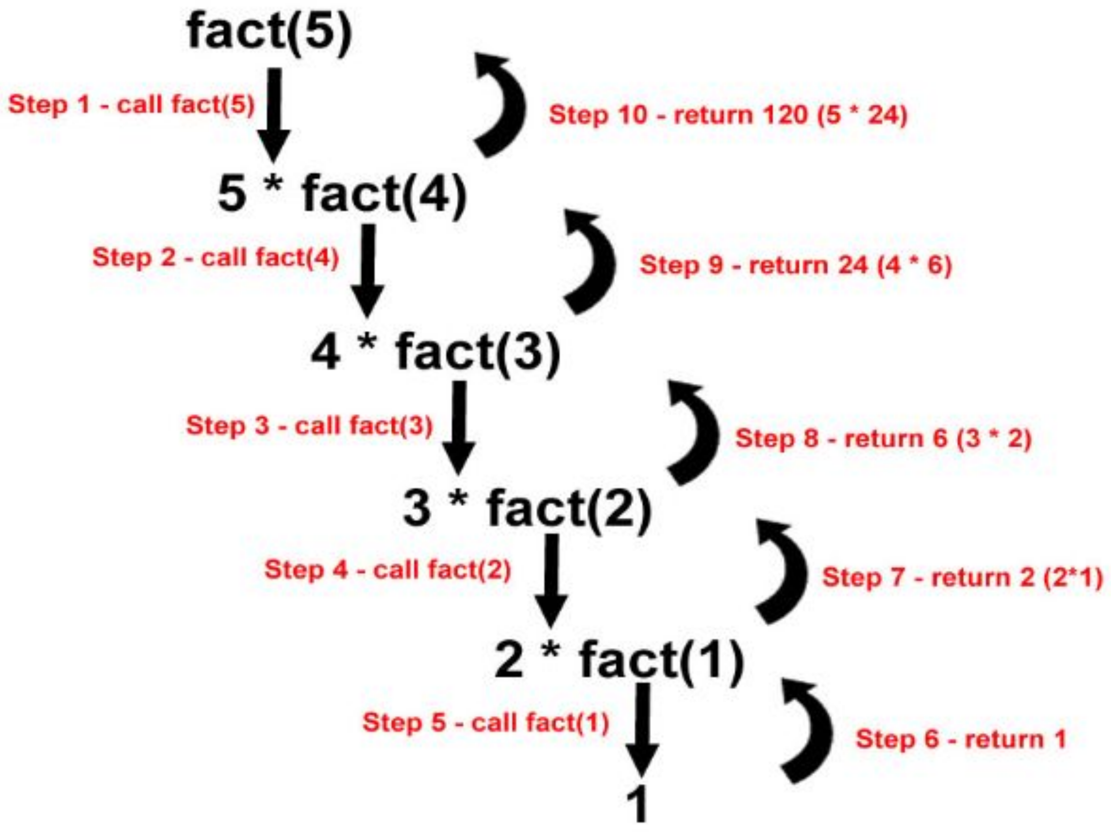
Iterative solution!

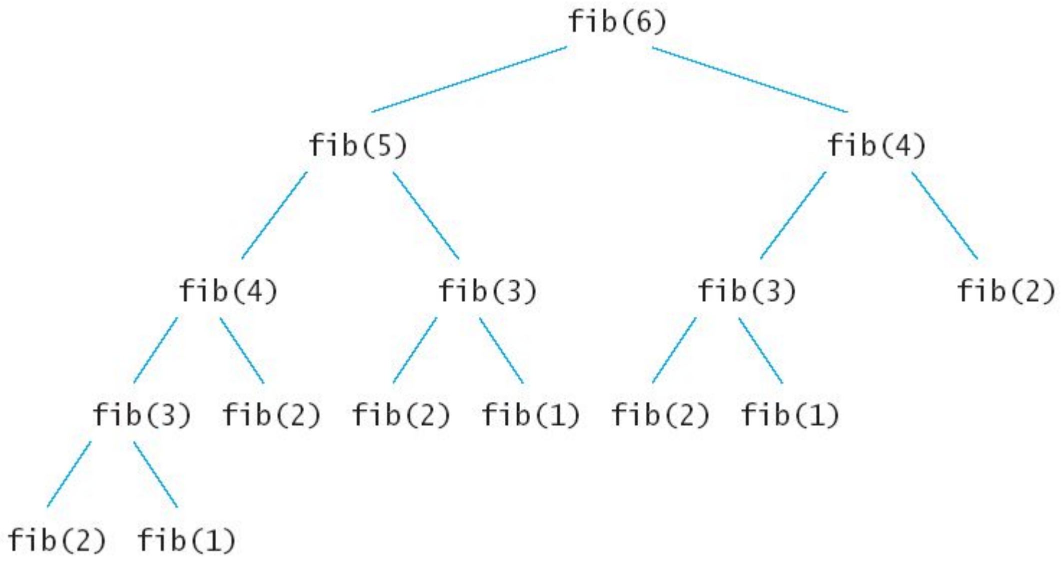
C(n) = ~n

Much faster (and better)!

Recursion vs Iteration

- Recursion is expensive
- Always check for an iterative solution
- Any recursive process can be implemented as an iterative process
    - Sometime it is too much complex





(Suggested) Exercises

- Write and test a function that, given an integer n, calculates the sum of numbers from 1 to n using recursion

- Write and test a recursive function that, given integers x and y, computes x^y

- Write and test a recursive function to count the number of digits of a given positive number (*hint*: keep dividing by 10)

- Write and test a recursive function to find the sum of digits of a given positive number (*hint*: divide by 10 and use modulo operator)

- **All** exercises in Chapter 9 of the reference book