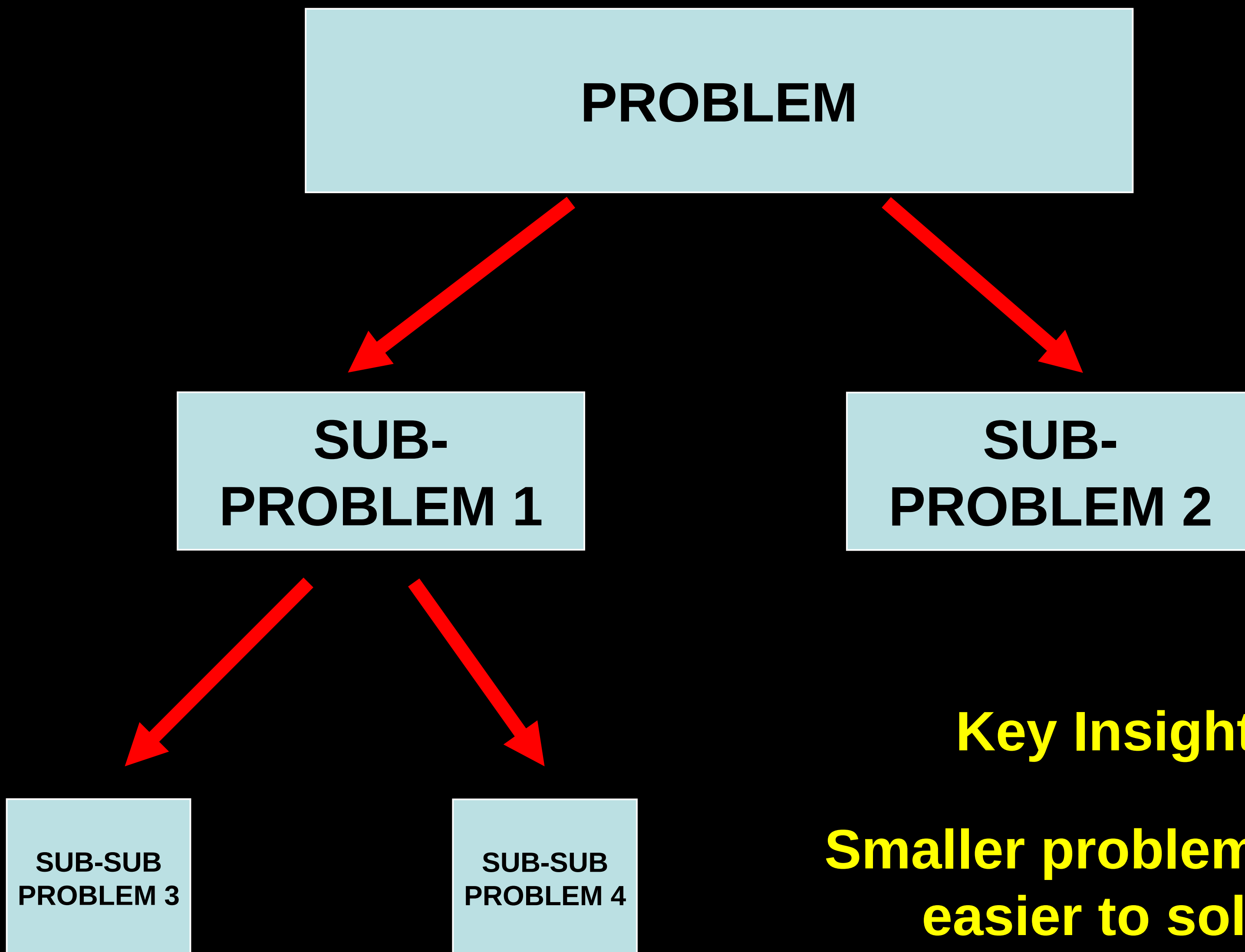
Lecture 5

Recursion

Divide-and-conquer

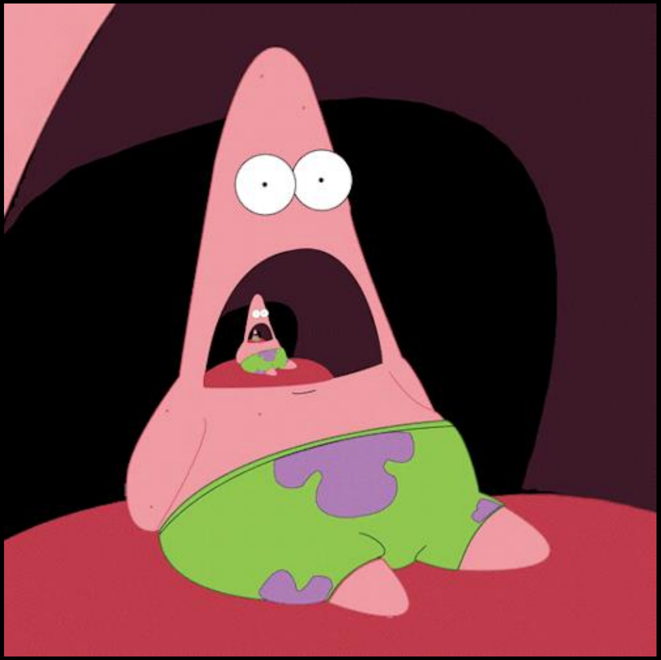


Key Insight:

Smaller problems are easier to solve

Recursion

Smaller child problem(s) has same structure as the parent



Factorial

$$n! = n * (n-1) * (n-2) *\ldots *1$$

$0! = 1$

$1! = 1$

$2! = 2 * 1= 2$

$3! = 3 * 2 * 1 = 6$

$4! = 4 * 3 * 2 * 1 = 24$

Classic Example

Consider the factorial function:

$$n! = n * (n-1) * (n-2) \ldots 1$$

Rewrite:

$$n! = \begin{cases} n * (n-1)! & \text{if } n > 1 \\ 1 & \text{if } n = 1 \end{cases}$$

Factorial

$$n! = \begin{cases} 1 & \text{if } n = 1 \\ n * (n-1)! & \text{if } n > 1 \end{cases}$$

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

Recursion

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

terminating condition/ base case

recursive call

Function that calls itself is called a <u>recursive</u> function.

Recursive process

```
factorial(6)
6 * factorial(5)
6 * 5 * factorial(4)
6 * 5 * 4 * factorial(3)
6 * 5 * 4 * 3 * factorial(2)
6 * 5 * 4 * 3 * 2 * factorial(1)
6 * 5 * 4 * 3 * 2 * 1
6 * 5 * 4 * 3 * 2
6 * 5 * 4 * 6
6 * 5 * 24
6 * 120
720
```

Note the build up of pending operations.

We care about <u>two</u> key considerations:

1. Time
2. Space

Time: how long it takes to run a program

Space: how much memory do we need to run the program

Recursive process

```
factorial(6)
6 * factorial(5)
6 * 5 * factorial(4)
6 * 5 * 4 * factorial(3)
6 * 5 * 4 * 3 * factorial(2)
6 * 5 * 4 * 3 * 2 * factorial(1)
6 * 5 * 4 * 3 * 2 * 1
6 * 5 * 4 * 3 * 2
6 * 5 * 4 * 6
6 * 5 * 24
6 * 120
720
```

pending/deferred
operations takes up space

Recursive process

```
factorial(6)
6 * factorial(5)
6 * 5 * factorial(4)
6 * 5 * 4 * factorial(3)
6 * 5 * 4 * 3 * factorial(2)
6 * 5 * 4 * 3 * 2 * factorial(1)
6 * 5 * 4 * 3 * 2 * 1
6 * 5 * 4 * 3 * 2
6 * 5 * 4 * 6
6 * 5 * 24
6 * 120
720
```

```
factorial(3)
3 * factorial(2)
3 * 2 * factorial(1)
3 * 2 * 1
3 * 2
6
```

Time $\propto$ #operations

Linearly proportional to *n*

Time complexity (order of growth): O(*n*)

Factorial: Linear recursion

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

```
(factorial 4)
      ↓
(factorial 3)
      ↓
(factorial 2)
      ↓
(factorial 1)
```

Fibonacci numbers

Leonardo Pisano Fibonacci (12th century) is credited for the sequence:

0, 1, 1, 2, 3, 5, 8, 13, 21, …

Note that each number in the sequence (except the first two) is simply the sum of the previous two.

Fibonacci numbers

Fibonnacci numbers can be expressed as the following

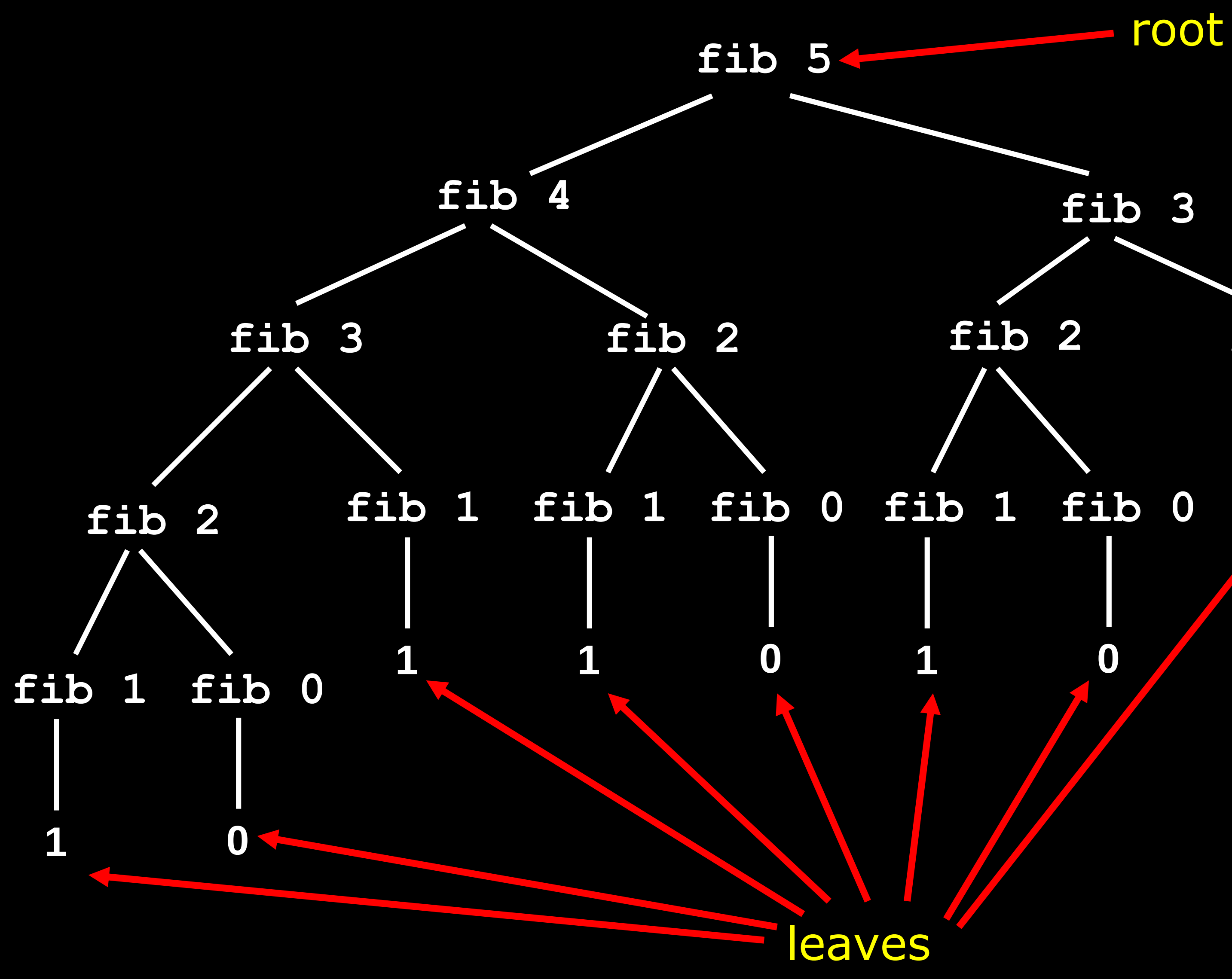
$$f(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ f(n-1) + f(n-2) & \text{if } n > 1 \end{cases}$$

Fibonacci in Python

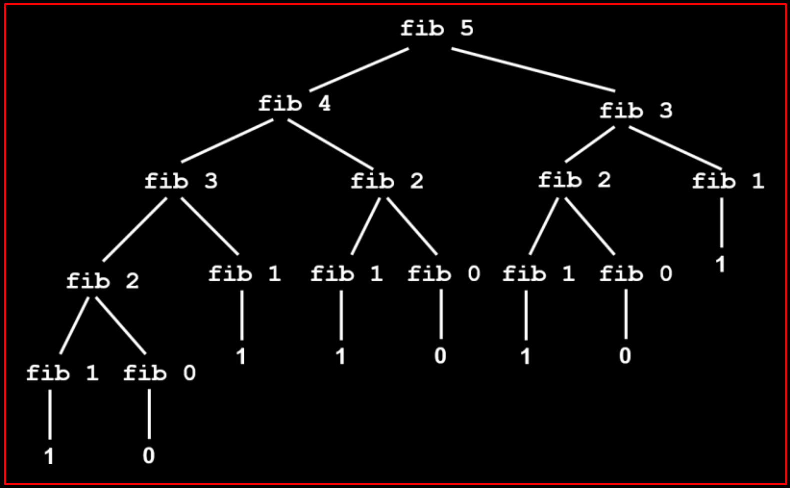
$$f(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ f(n-1) + f(n-2) & \text{if } n > 1 \end{cases}$$

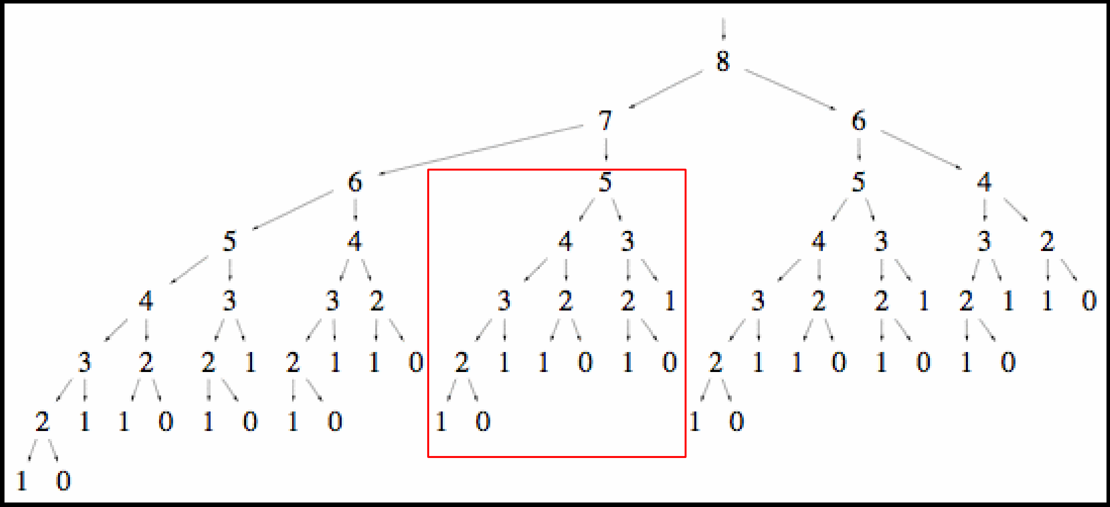
```
def fib(n):
    if (n == 0):
        return 0
    elif (n == 1):
        return 1
    else:
        return fib(n - 1) + fib(n - 2)
```

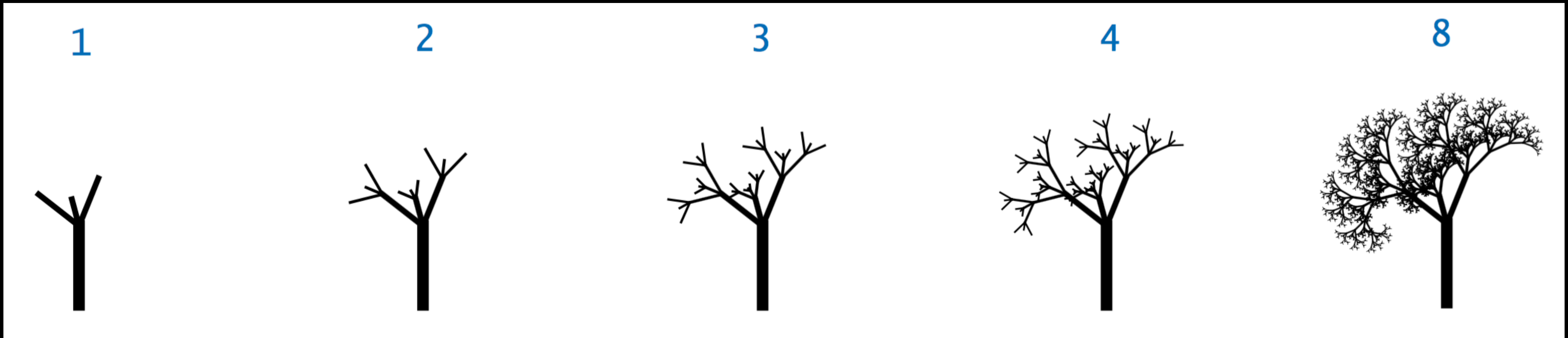
Tree recursion



Tree recursion







Tree recursion

- Time taken is proportional to number of leaves, i.e.
  exponential in *n*.

Time complexity (order of growth): O(2^n)

Mutual recursion

```
def ping(n):
    if (n == 0):
        return n
    else:
        print("Ping!")
        pong(n - 1)

def pong(n):
    if (n == 0):
        return n
    else:
        print("Pong!")
        ping(n - 1)
```

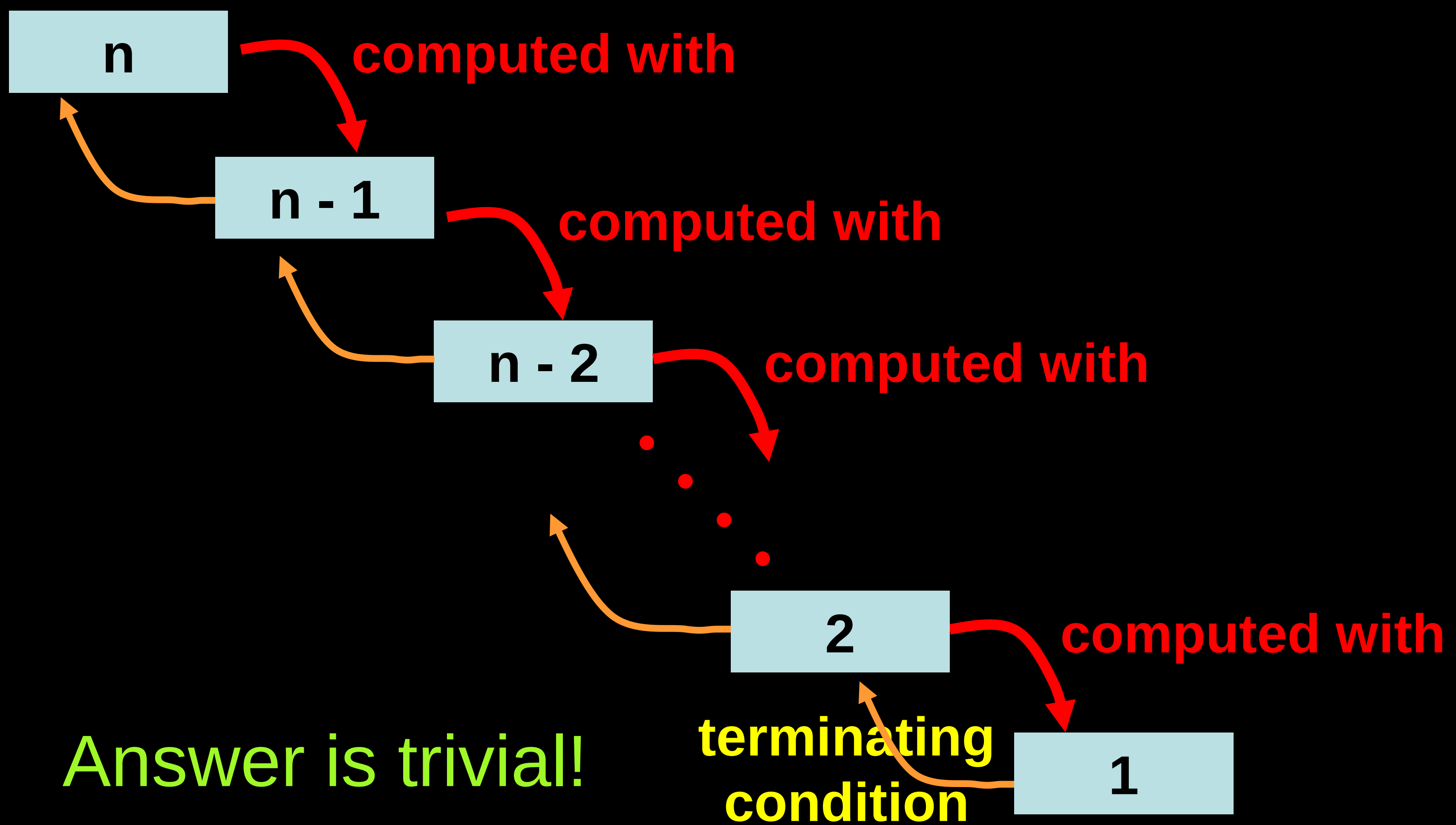
```
ping(10)

Ping!
Pong!
Ping!
Pong!
Ping!
Pong!
Ping!
Pong!
Ping!
Pong!
```

Recursion

- Solve the problem for a simple (base) case
  - Typically $n=0$ or $n=1$
- Express (divide) a problem into one or more smaller similar problems
  - Assume you know how to solve the problem for n-1. How can you use this information to solve the problem for n?

Recursion



Answer is trivial!