Prepared by Julian García, based on material by
David Albrecht and María García de la Banda

Lecture 21
Recursion

FIT 1008
Introduction to Computer Science

MONASH University
Information Technology

Objectives for this lecture

- To re-visit the concept of **recursive algorithm**
- To understand how to **implement** recursive algorithms.
- To be able to reason about **recursive algorithms**

Recursive algorithms

- Solve a Large problem by solving subproblems
  - ➡ of the same kind as the original.
  - ➡ simpler to solve

Each **subproblem** is solved with the **same algorithm** …

… **until** subproblems are so "<u>simple</u>" that they can be solved without further reductions (**base case**)

Candidate problems for recursion.

1. Must be possible to **decompose** them into simpler **similar problems**

2. **At some point**, the problems must become so simple that can be solved with **no further decomposition**

3. Once all subproblems are solved, the solution to the original problem can be computed by combining these solutions

General recursive structure

Simpler / Smaller

```
def solve(problem):
    if problem is simple:
        Solve problem directly
    else:
        Decompose problem into subproblems p1, p2,…
        solve(p1)
        solve(p2)
        solve(p3)…
        Combine the subsolutions to solve problem
```

Base cases

Recursive calls

That of a **function** that **calls itself** (directly or via others).

Factorial: a recursive approach?

Examples:

```
0! = 1
1! = 1
2! = 1*2
3! = 1*2*3
4! = 1*2*3*4
…
(n-1)! = 1*2*3*4*…*(n-1)
    n! = 1*2*3*4*…*(n-1)*n
```

1. Must be possible to **decompose** them into simpler **similar problems**
2. **At some point**, the problems must become so simple that can be solved with **no further decomposition**
3. Once all subproblems are solved, the solution to the original problem can be computed by combining these solutions

```
n! = (n-1)!*n
```

Subproblems: smaller, <u>converge</u> to base case.

Factorial: a recursive approach

```
def factorial(n):
    if n == 0:  # base case
        return 1
    else:
        return n*factorial(n-1) # recursive call
```

Convergence

Combination

Key idea for complexity of recursion:

- <u>How many recursive calls do we make in each version?</u>
- <u>How much work do we do per call?</u>

Terminology

Arity:

- Unary: a single recursive call (all previous code)
- Binary: two recursive calls (recursive sorts, later…)
- n-ary: n recursive calls

Direct vs Indirect recursion:

- Direct: recursive calls are calls to the same function (all previous examples)
- Indirect (or mutual): recursion through two or more methods (e.g., method p calls method q which in turn calls p again)

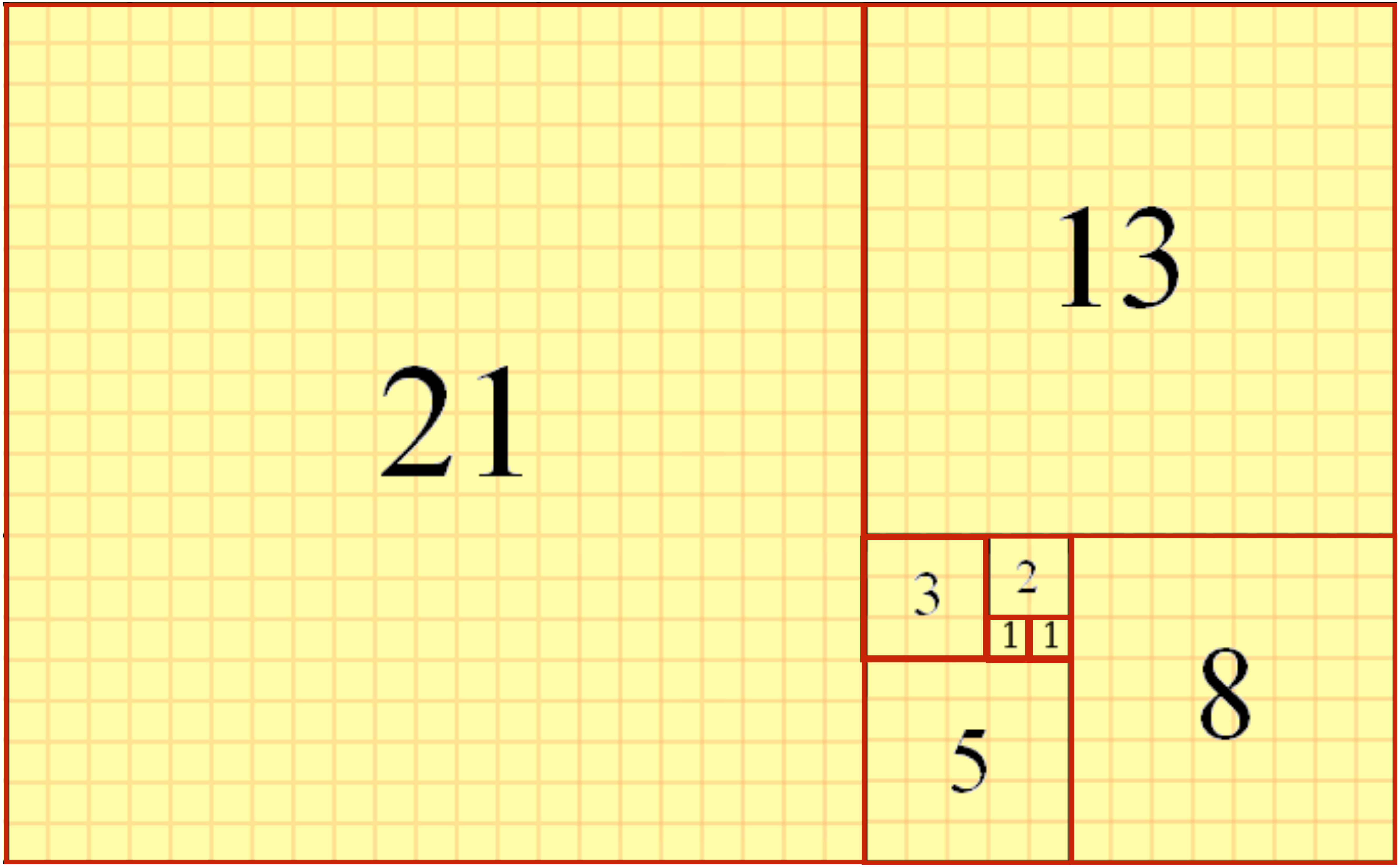
Tail-recursion:

- Where the result of the recursive call is the result of the function. Nothing is done in the “way back”. Closest to iteration - can be transformed *without “storing”*. Useful for compiler optimisation.

Tail recursive version of factorial

```python
def factorial(n):
    return factorial_aux(n,1)


def factorial_aux(n, result):
    if n == 0:
        return result
    else:
        return factorial_aux(n-1, result*n)
```


21
13
3
2
1
1
8
5

n = 0	**0**
n = 1	**1**
n = 2	**1**
n = 3	**2**
n = 4	**3**
n = 5	**5**

Fib(5)

Fibonacci

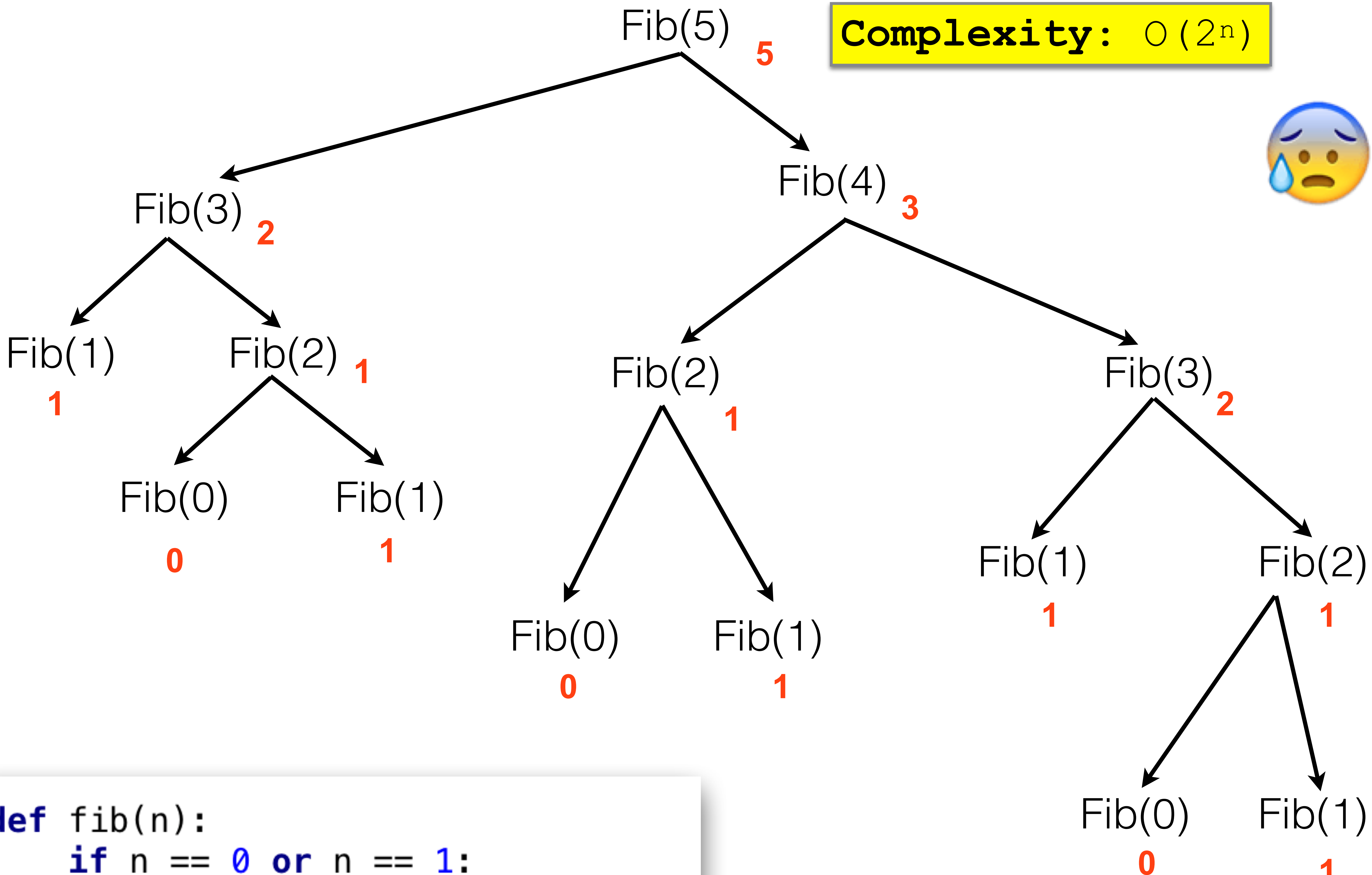
n = 0	0
n = 1	1
n = 2	1
n = 3	2
n = 4	3
n = 5	5

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

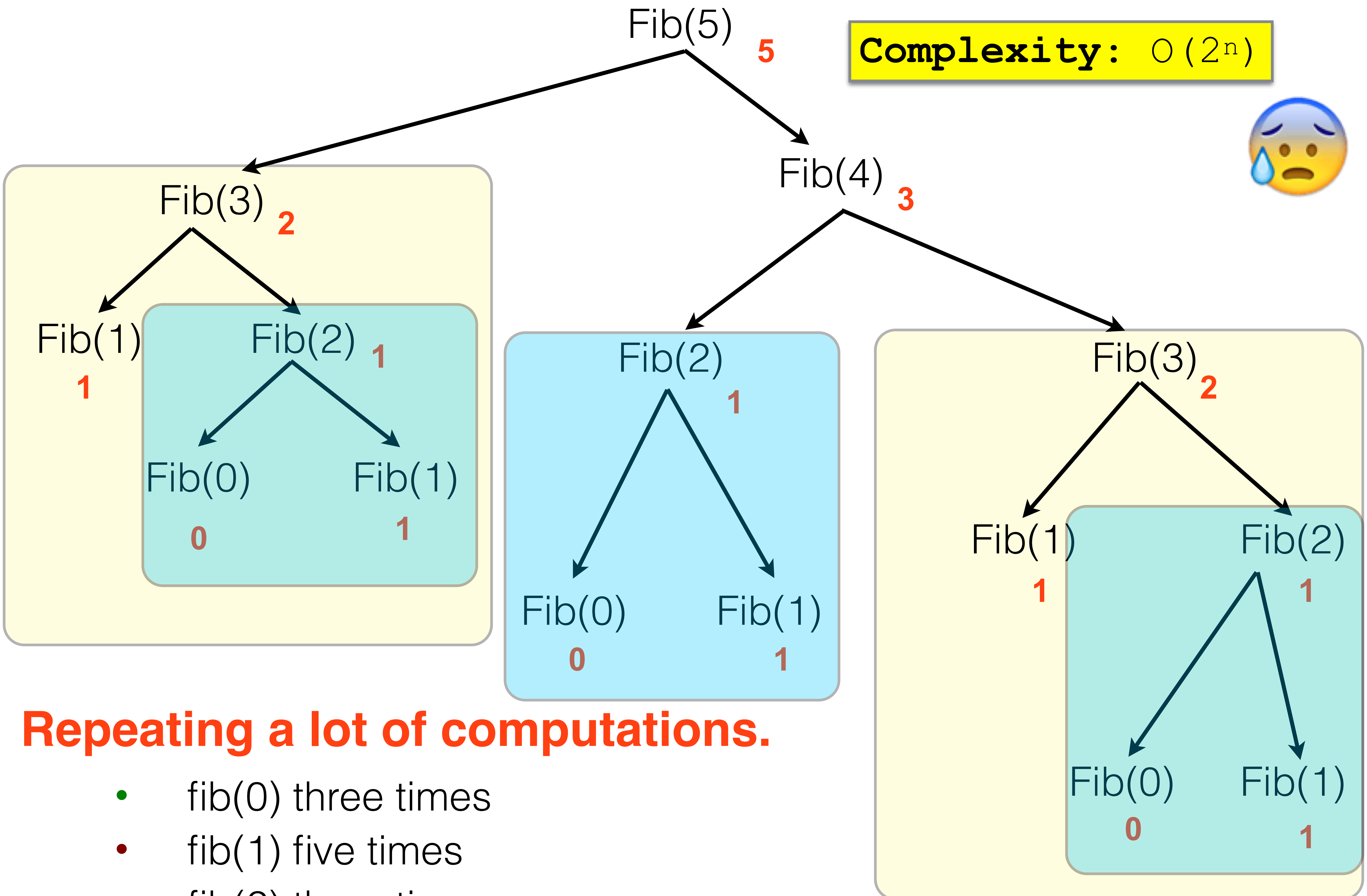
First.

Second.

Complexity: $O(2^n)$




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



Repeating a lot of computations.

- fib(0) three times
- fib(1) five times
- fib(2) three times
- fib(3) two times

```
                 before_last   last
fib(7)
fib_aux(7, 0, 1)
fib_aux(6, 1, 1)
fib_aux(5, 1, 2)
fib_aux(4, 2, 3)
fib_aux(3, 3, 5)
fib_aux(2, 5, 8)
fib_aux(1, 8, 13)
fib_aux(0, 13, 21)
```

Observations

- n decreases each call
- before_last = last
- last = before_last +last
- base case n =0
- return before_last

0	1	2	3	4	5	6	7	8
0	1	1	2	3	5	8	13	21

```python
def fib(n):
    return fib_aux(n, 0, 1)


def fib_aux(n, before_last, last):
    if n == 0:
        return before_last
    else:
        return fib_aux(n-1, last, before_last+last)
```

```python
def fib(n):
    return fib_aux(n, 0, 1)


def fib_aux(n, before_last, last):
    if n == 0:
        return before_last
    else:
        return fib_aux(n-1, last, before_last+last)
```

fib(5)

fib_aux(5, 0, 1)

fib_aux(4, 1, 0+1)

fib_aux(3, 1, 1+1)

fib_aux(2, 2, 2+1)

fib_aux(1, 3, 3+2)

fib_aux(0, 5, 5+3)

return 5

Complexity: O(n)

What happens with Recursion at the Low Level?

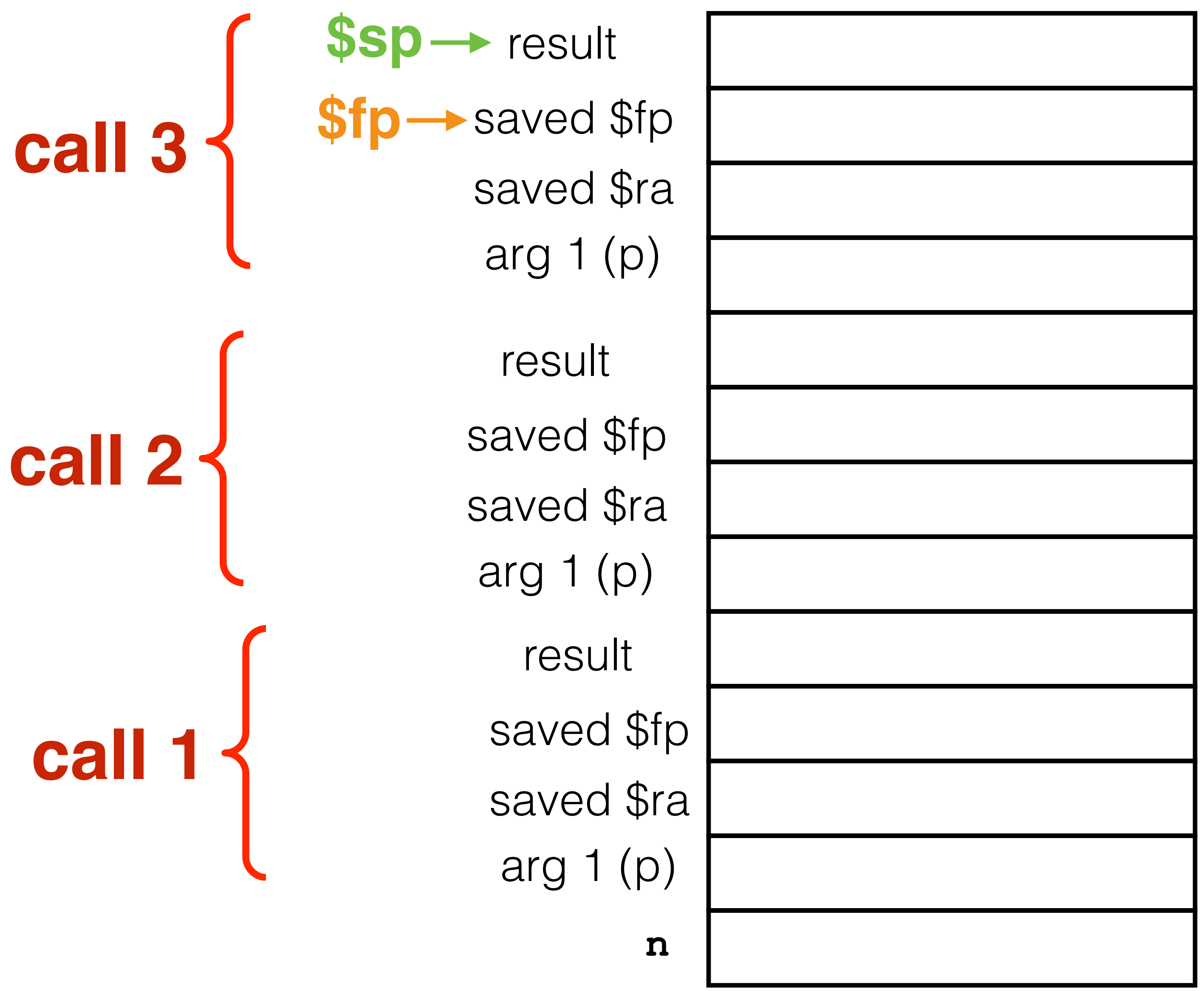
MIPS **Recursion**:

a function calling a function

(simply happen to be the same functions)

↓

We do not need to learn anything new.


call 3
call 2
call 1
$sp
$fp
result
saved $fp
saved $ra
arg 1 (p)
result
saved $fp
saved $ra
arg 1 (p)
result
saved $fp
saved $ra
arg 1 (p)
n

Tail-recursion:

- Where the result of the recursive call is the result of the function. Nothing is done in the “way back”. Closest to iteration - can be transformed *without “storing”*. **Useful for compiler optimisation.**

MIPS & Tail recursion

Fibonacci: clarity vs efficiency

- First recursive version is:
  - Clear
  - **Descriptive**
  - Inefficient

- Tail recursion with accumulators:
  - Not so clear at first
  - **More Efficient**
  - It can be easily transformed into an iterative version

Straightforward recursion is natural, but not necessarily the most efficient.

Summary

- Recursive algorithms and their implementation
- Complexity of recursive algorithms
- Reasoning about recursion