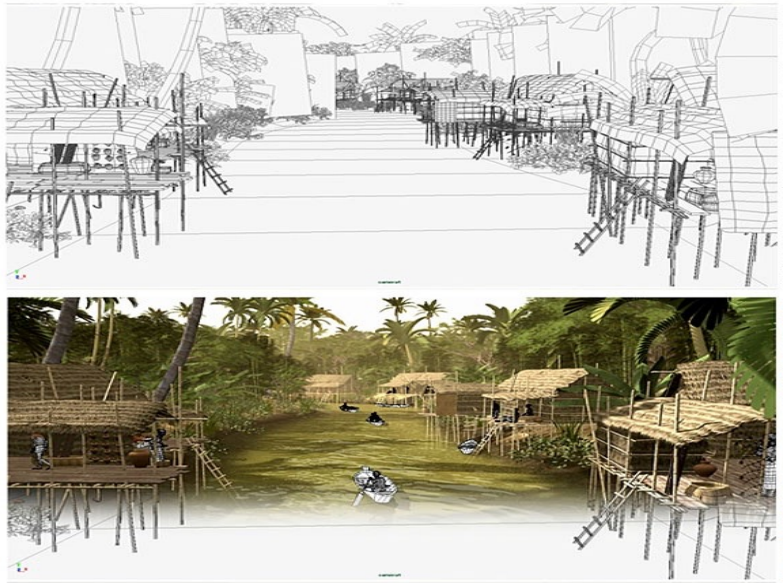
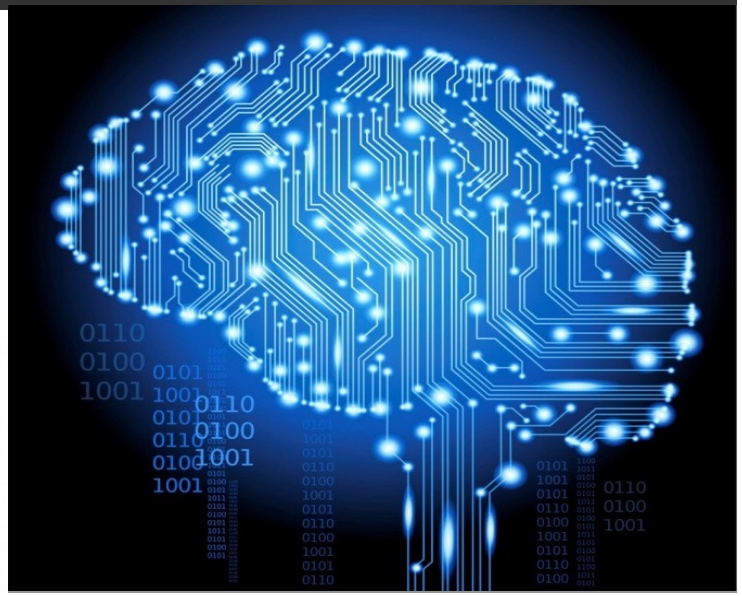
MONASH University

Information Technology

Recursion II

Prepared by Maria Garcia de la Banda
Updated by Brendon Taylor





Objectives for this lecture

- **To learn a bit about recursive notation**
- **To explore more complex recursive algorithms**
- **To continue exploring the relationship between iteration and recursion**
- **To learn about transforming complex recursion into iteration by:**
  - Using accumulators
  - Using a stack

MONASH University

Recursion notation

Some Recursive Notation

- **Unary, binary, *n*-ary recursion:**
  - Unary: a single recursive call (all previous code)
  - Binary: two recursive calls ("find a route" example)
  - ***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:
    - That is: nothing is done in the "way back"
  - Closest to iteration: can be straightforwardly transformed into it

Is it tail recursive?

```
def factorial(n: int) -> int:
    if n == 0:
        return 1
    else:
        return n*factorial(n-1)
```

Nope! In the way back we multiply

- **But we can make it tail recursive by using an accumulator**

```
def factorial(n: int) -> int:
     return factorial_aux(n,1)

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

Accumulates the result (multiplies when going “in”)

Result of the recursive call is the result of the function

MONASH University

From Recursion to Iteration: Accumulators

Binary recursion: Fibonacci

- **Some things are easier to define recursively**
  - They are not simple iterations
- **Mathematical definition for Fibonacci:**

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

- **Defined almost identically in most programming languages**

Fibonacci': binary recursive approach

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

Why not just `n==0`?

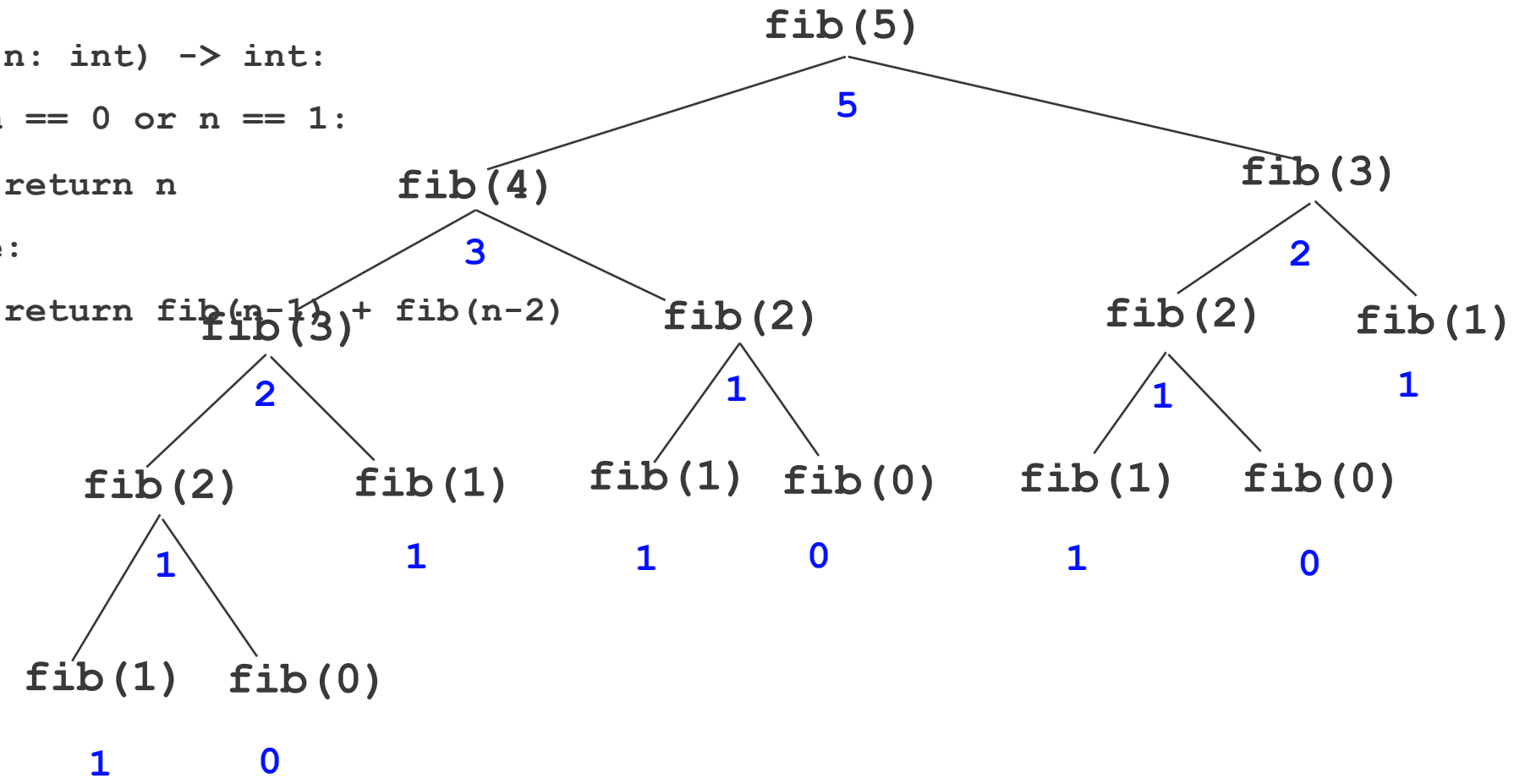
This is why: `n-2`

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

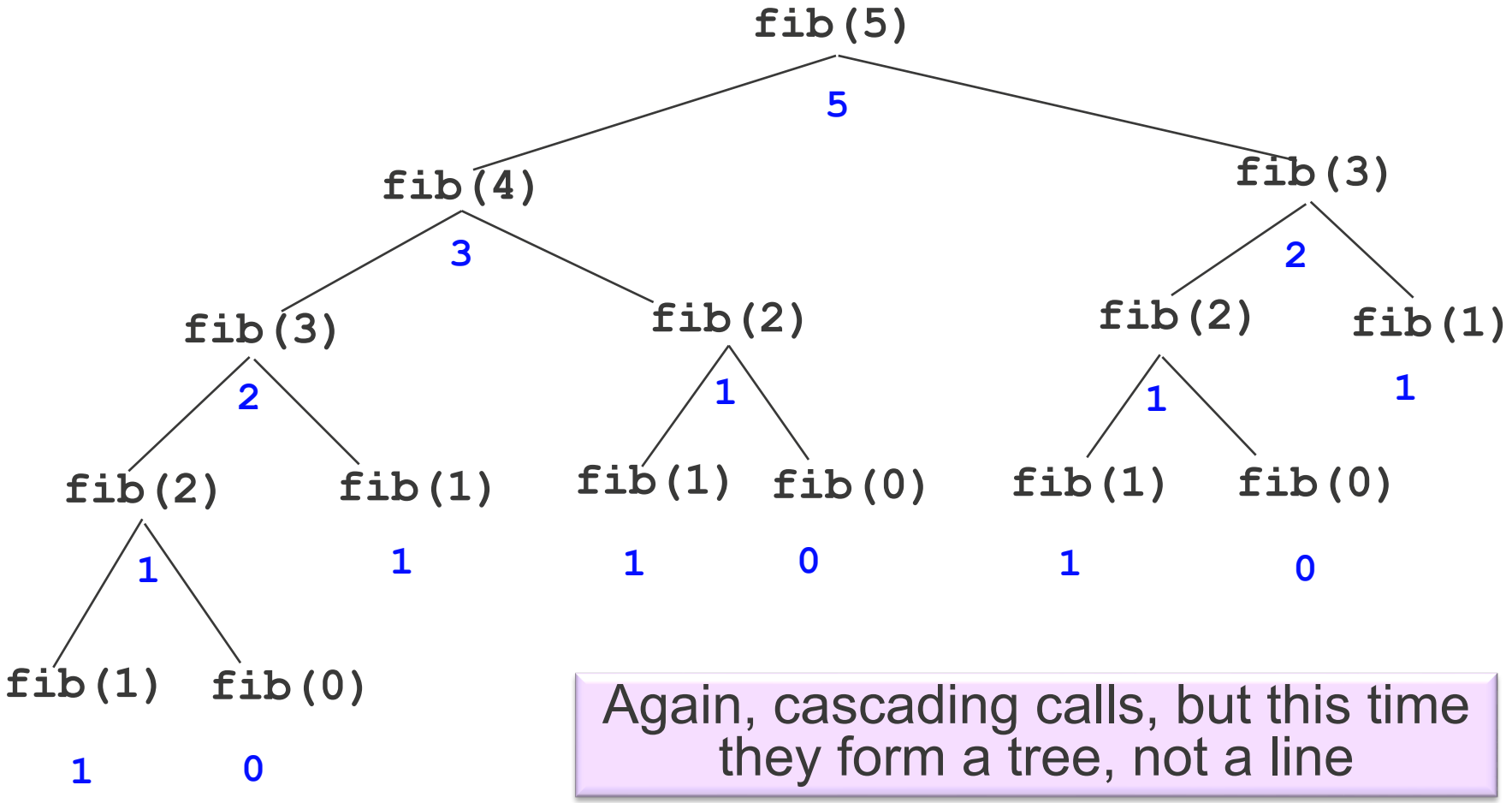
What would the execution be like?

fibonacci's execution: call tree

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



fibonacci's execution: call tree

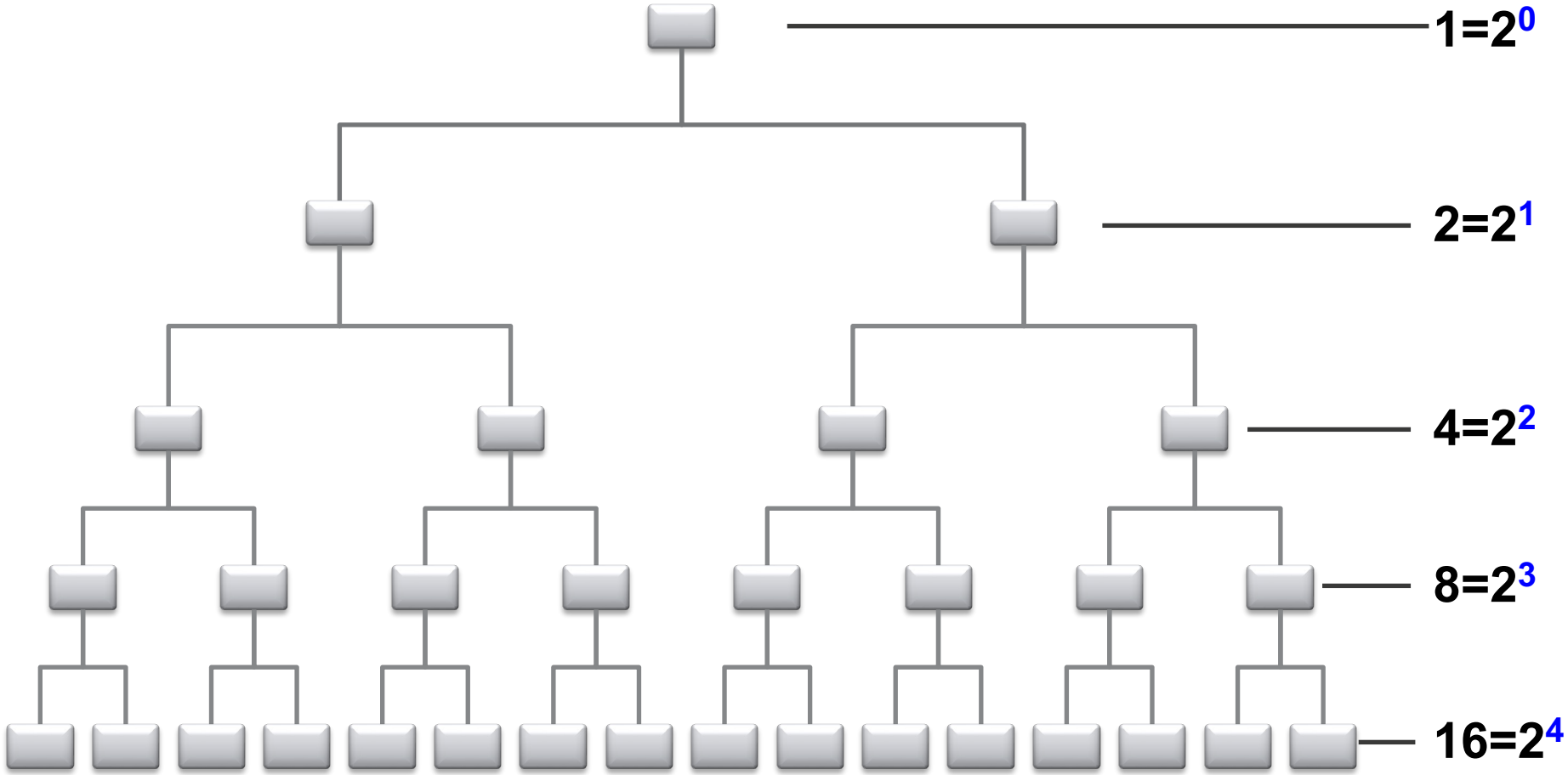


Again, cascading calls, but this time they form a tree, not a line

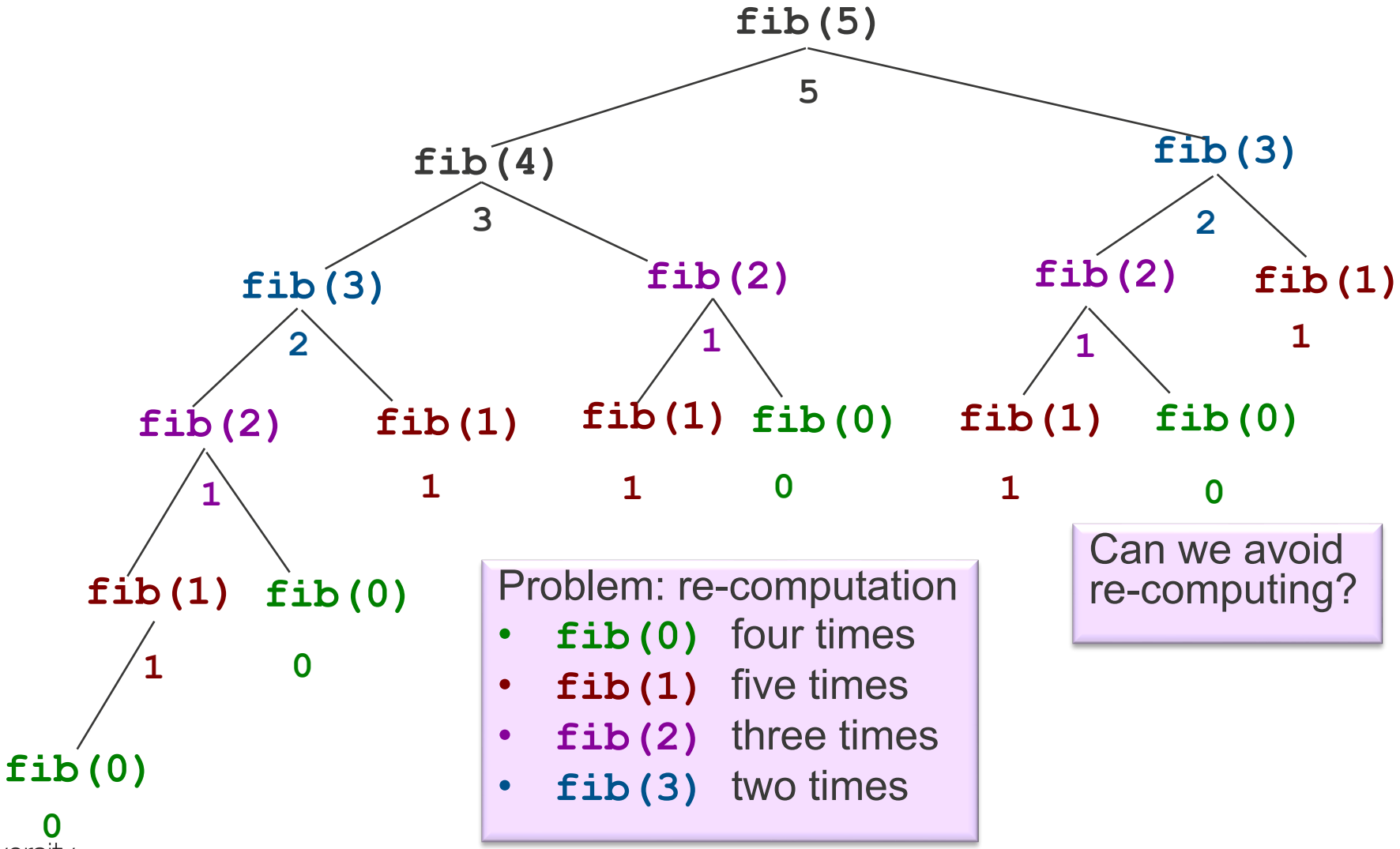
Complexity? $O(2^n)$

Why 2^n: think about the call tree

- n is the DEPTH of the binary tree



Fibonacci's binary recursion: needed?

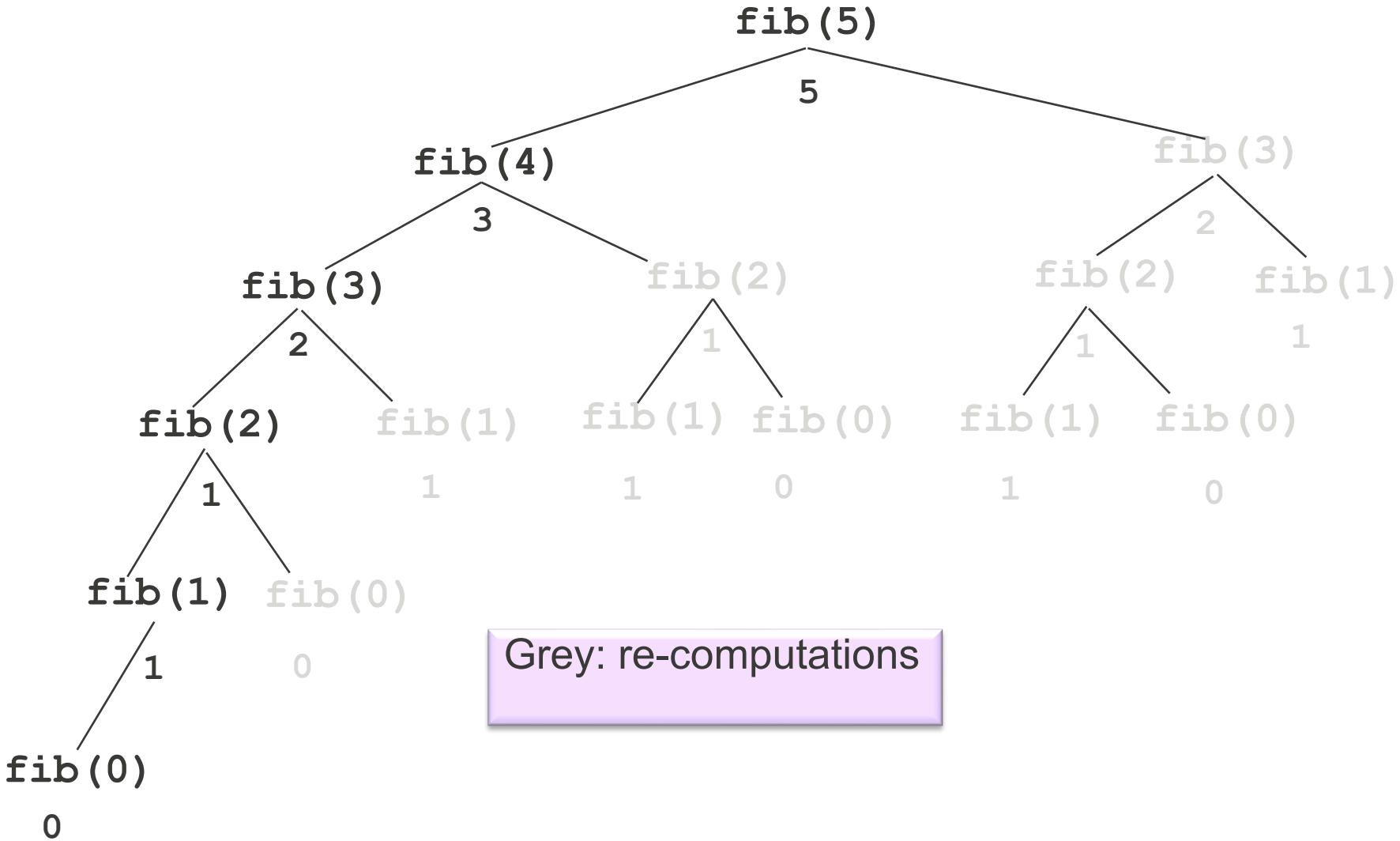


Problem: re-computation

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

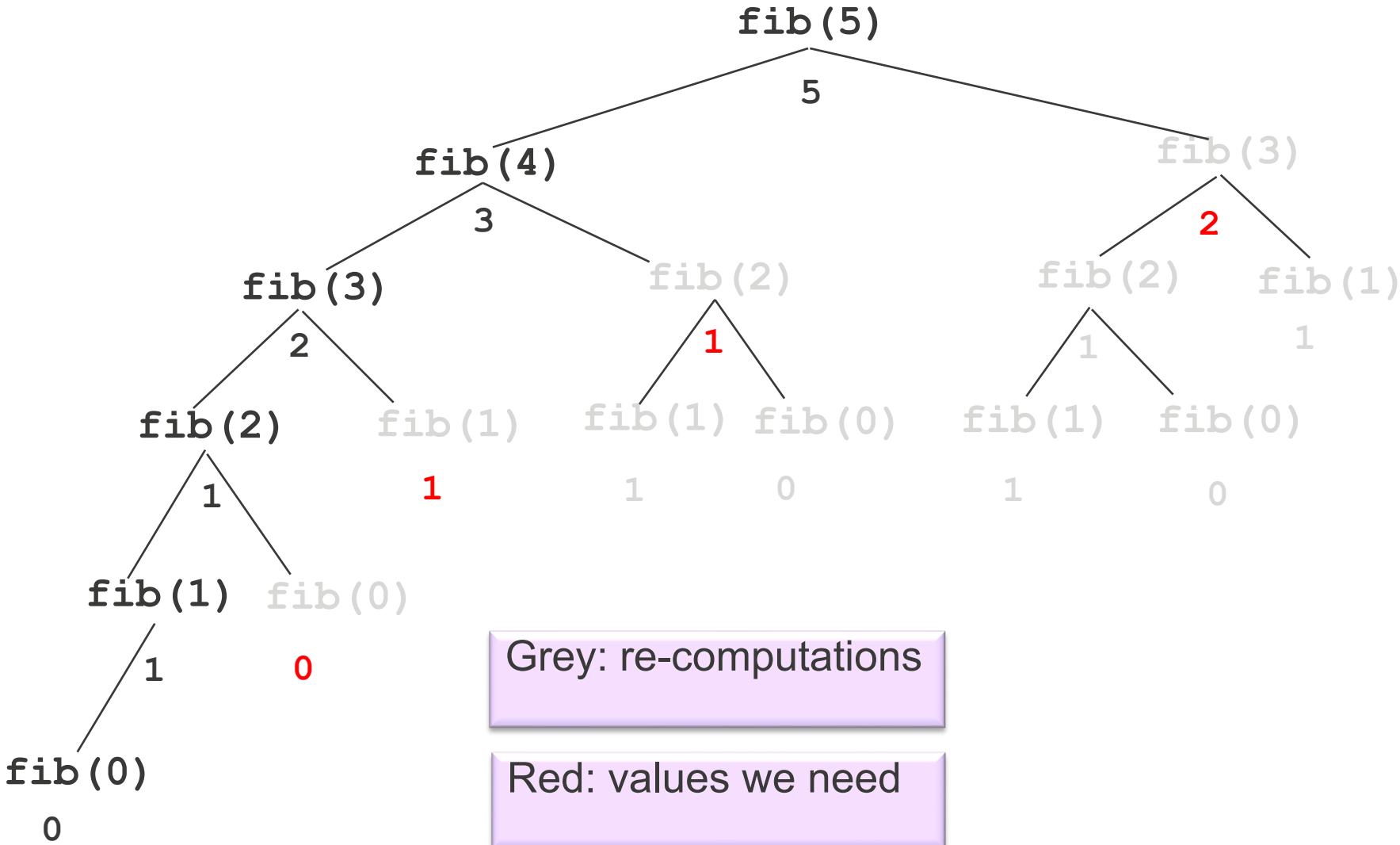
Can we avoid re-computing?

Fibonacci's binary recursion: needed?



Grey: re-computations

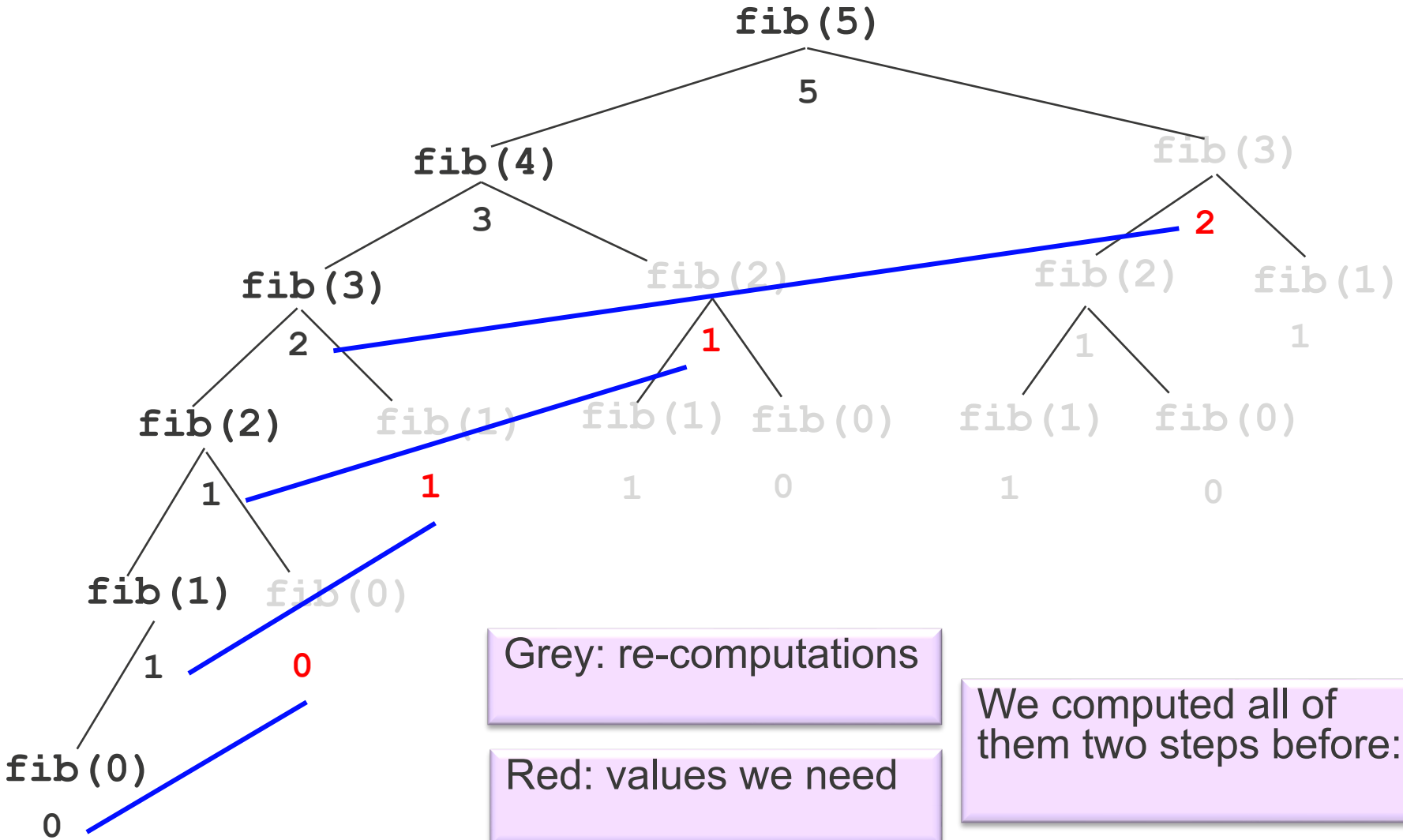
Fibonacci's binary recursion: needed?



Grey: re-computations

Red: values we need

Fibonacci's binary recursion: needed?

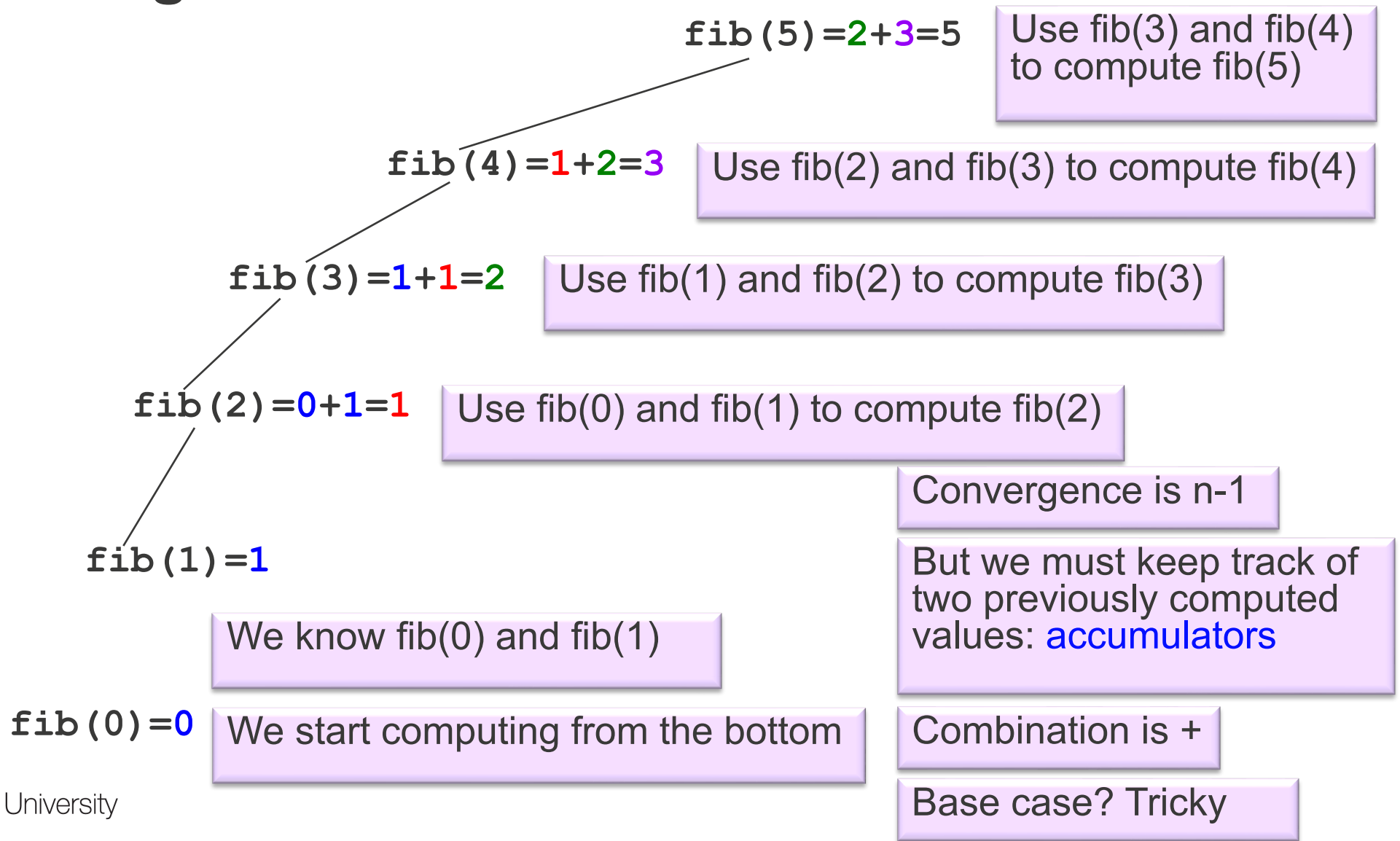


Grey: re-computations

Red: values we need

We computed all of them two steps before:

Re-thinking fibonacci's execution



Fibonacci: recursion with accumulators

```
def fib(n: int) -> int:
    return fib_aux(n, 0, 1)


def fib_aux(n: int, fm2: int, fm1: int) -> int:
    if n == 0:
        return fm2
    else:
        return fib_aux(n-1, fm1, fm2+fm1)
```

We start carrying the fib(0) and fib(1) with us

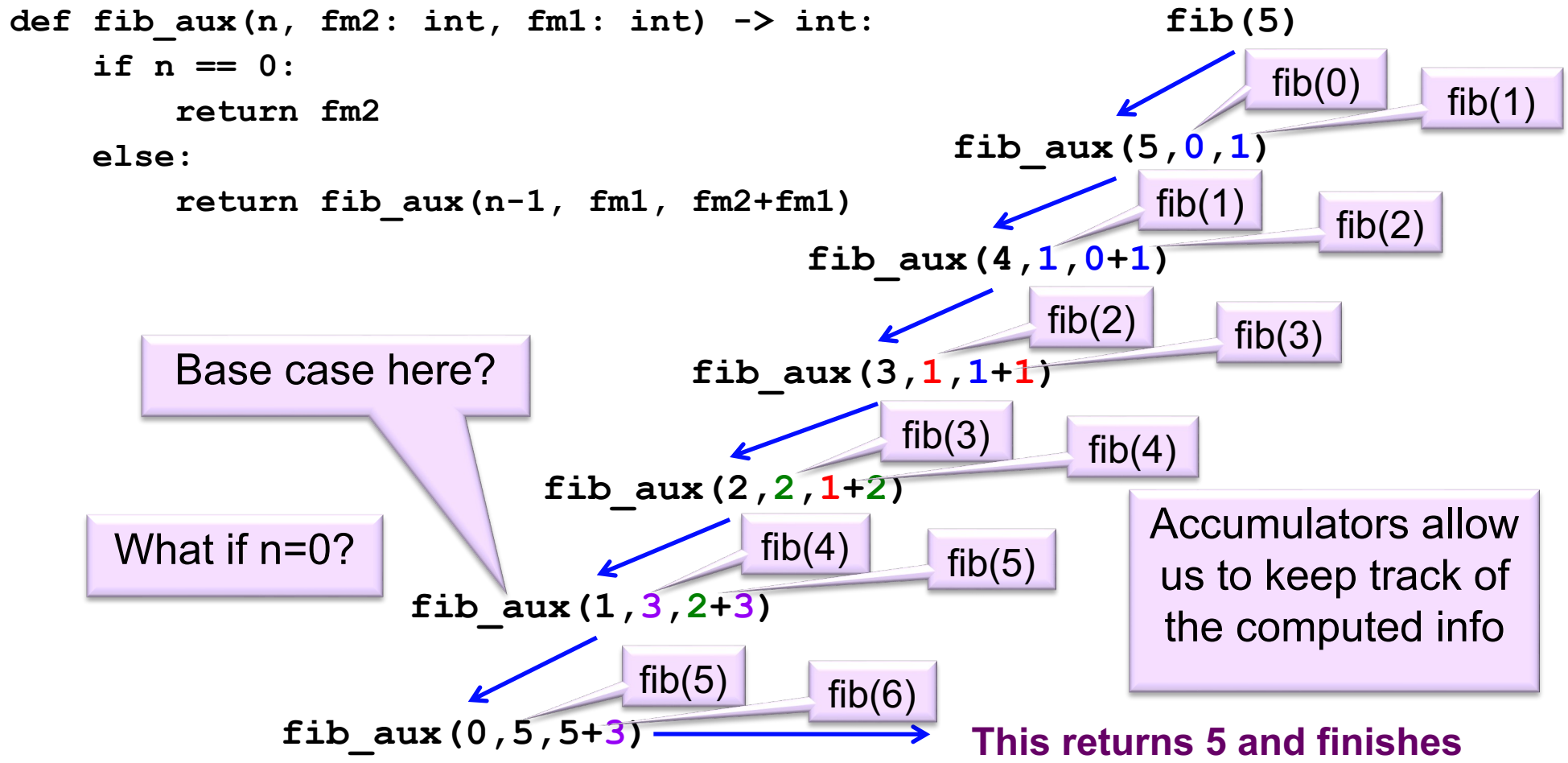
These two are the accumulators

The last one computed

The one before last

How is the new call tree?

```
def fib_aux(n, fm2: int, fm1: int) -> int:
    if n == 0:
        return fm2
    else:
        return fib_aux(n-1, fm1, fm2+fm1)
```



Fibonacci: clarity vs efficiency

- **First recursive version is:**
  - Very clear
  - Valuable as a description of mathematical properties
  - Poor as a solving algorithm
- **Second recursive version (using accumulators) is:**
  - Much less clear
  - Much more efficient
  - It can be easily transformed into an iterative version

Straight recursion specification might be very natural, but it is not always the most efficient

MONASH University

From Recursion to Iteration: Stack

Example: from recursion to iteration

- **Consider a method which computes X^n**
- **It can be done iteratively: a for loop on n**
- **It can also be done recursively. If you have:**
  - X^8 can be done as $X^4 * X^4$ ($=X^{4+4}=X^8$)
  - X^{12} can be done as $X^6 * X^6$ ($=X^{6+6}=X^{12}$)
  - X^{25} can be done as $X * X^{12} * X^{12}$ ($=X^{1+12+12}=X^{25}$)
- **Thus, compute $X^{n//2}$ (convergence by `n//2`) and if**
  - n is even: multiply the result by itself
  - n is odd: multiply the result by itself and by X

integer division

Combination of sub-problem solutions

- **Base case: n=0 return 1**

Example: from recursion to iteration

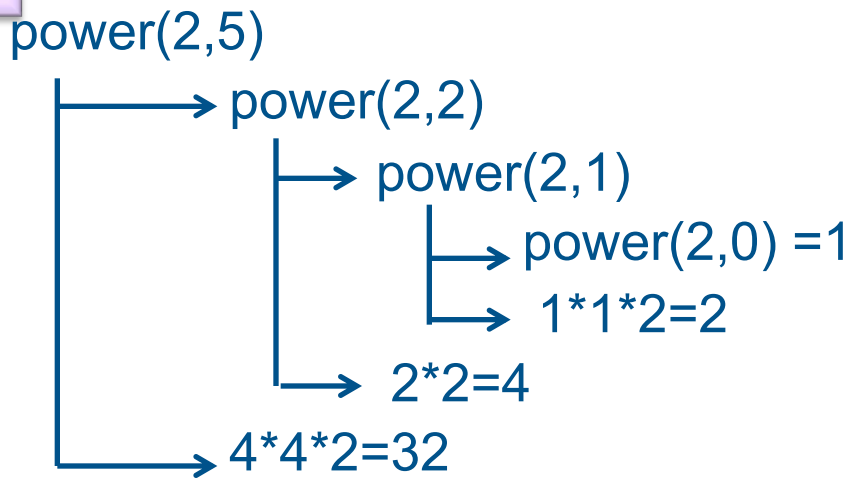
```
def power(x: int, n: int) -> int:

    tmp = 1
    if n > 0:
        tmp = power(x, n//2)
        if n % 2 == 0:
            tmp = tmp*tmp
        else:
            tmp = tmp*tmp*x

    return tmp
```

convergence

combination



Tail recursive?

No! the multiplication is done afterwards

Recall: recursion with the Runtime Stack

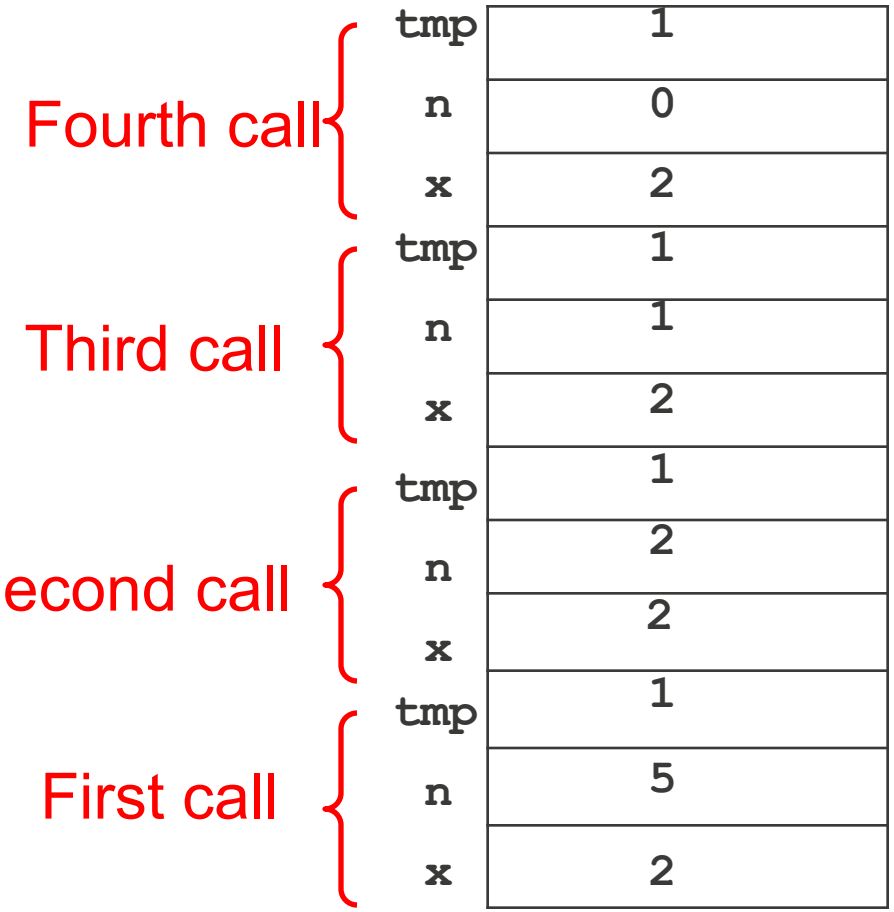
- **The system implements recursion by using the runtime stack**
- **Each recursive call reserves a new stack area** — push
  - For the parameters and local variables
- **Control is then given to the function**
  - To modify its variables according to its definition
- **Each time a call finishes** — pop
  - Area is removed after transferring the return value to its place

From recursion to iteration

```
def power(x: int, n: int) -> int:

    tmp = 1
    if n > 0:
        tmp = power(x, n/2)
        if n % 2 == 0:
            tmp = tmp*tmp
        else:
            tmp = tmp*tmp*x

    return tmp
```



Runtime stack (simplified, no $ra, $fp)

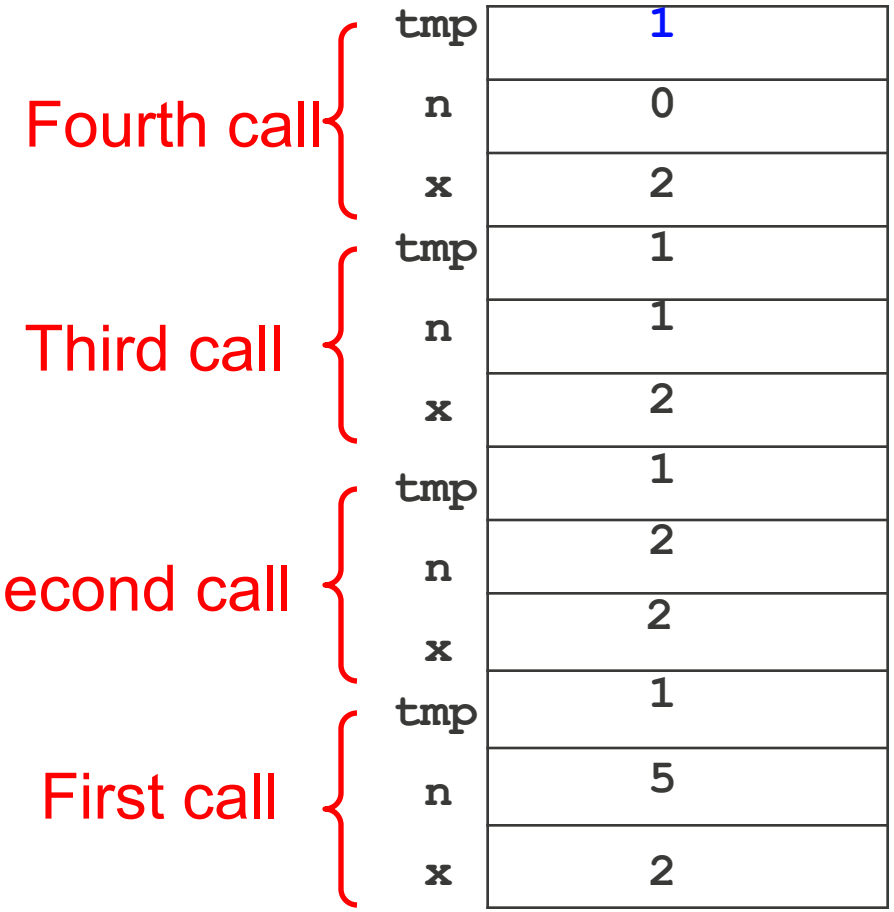
From recursion to iteration

```
def power(x: int, n: int) -> int:

    tmp = 1
    if n > 0:
        tmp = power(x, n/2)
        if n % 2 == 0:
            tmp = tmp*tmp
        else:
            tmp = tmp*tmp*x

    return tmp
```

Right before the fourth call returns



Runtime stack (simplified, no $ra, $fp)

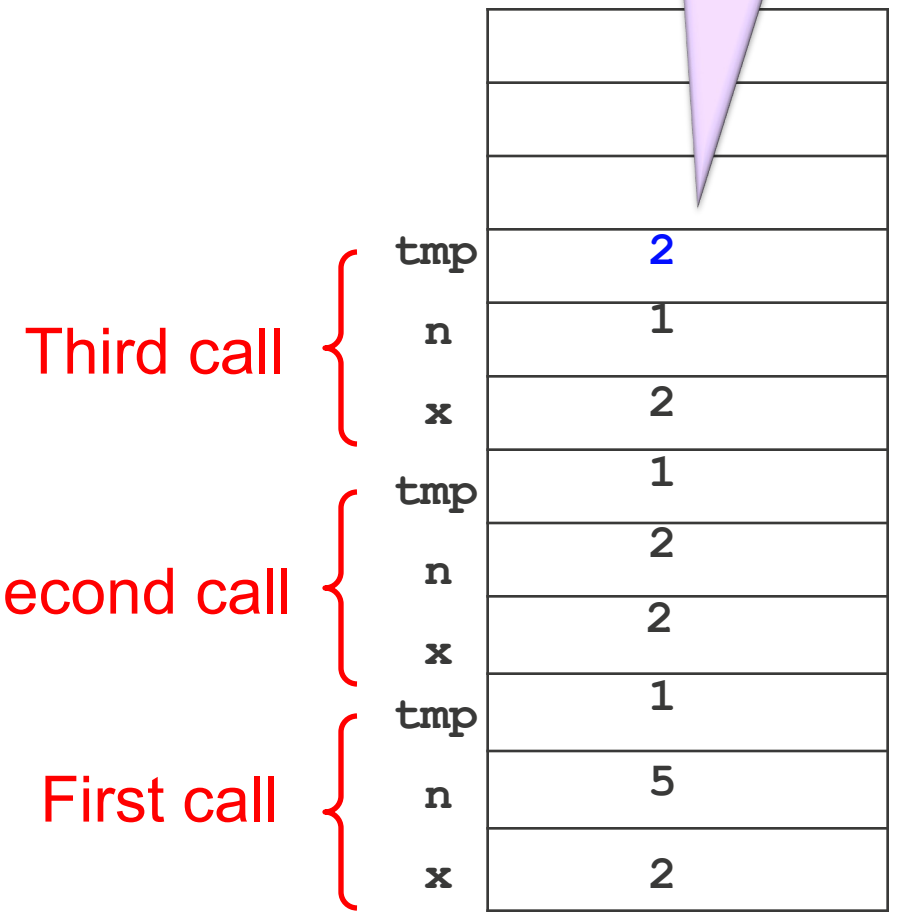
From recursion to iteration

```
def power(x: int, n: int) -> int:

    tmp = 1
    if n > 0:
        tmp = power(x, n/2)
        if n % 2 == 0:
            tmp = tmp*tmp
        else:
            tmp = tmp*tmp*x

    return tmp
```

Right before the third call returns



Runtime stack (simplified, no $ra, $fp)

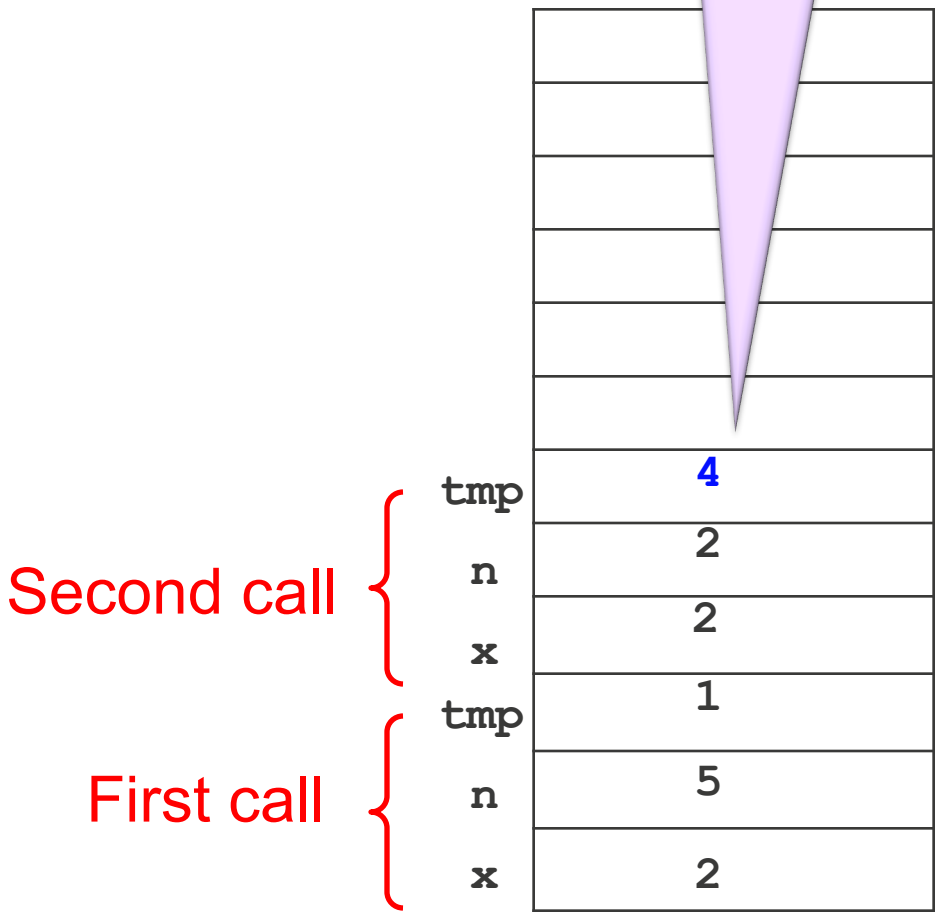
From recursion to iteration

```
def power(x: int, n: int) -> int:

    tmp = 1
    if n > 0:
        tmp = power(x, n/2)
        if n % 2 == 0:
            tmp = tmp*tmp
        else:
            tmp = tmp*tmp*x

    return tmp
```

Right before the second call returns



Runtime stack (simplified, no $ra, $fp)

From recursion to iteration

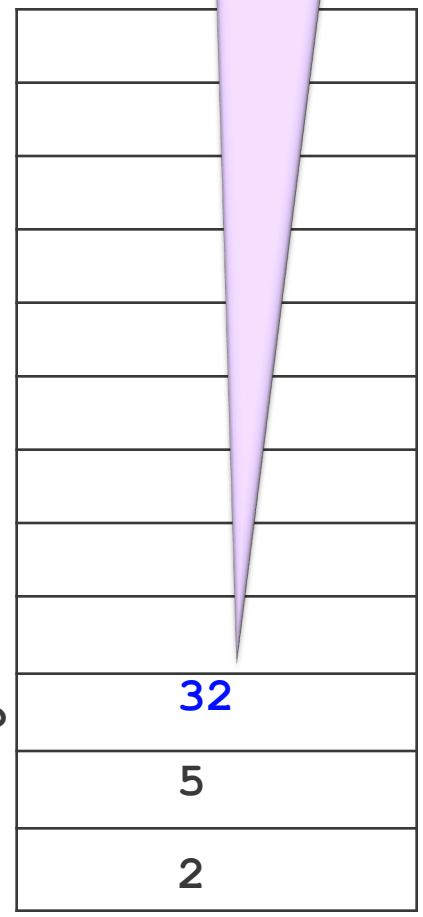
```
def power(x: int, n: int) -> int:

    tmp = 1
    if n > 0:
        tmp = power(x, n/2)
        if n % 2 == 0:
            tmp = tmp*tmp
        else:
            tmp = tmp*tmp*x

    return tmp
```

Direct recursion is easier to transform into iteration

Right before the first call returns



First call

Runtime stack (simplified, no $ra, $fp)

Direct iterative stack simulation :

```
def power(x: int, n: int) -> int:
    the_stack = Stack(n)
    while n > 0:
        the_stack.push(n)
        n = n//2

    tmp = 1
    while not the_stack.is_empty():
        if the_stack.pop() % 2 == 0:
            tmp = tmp*tmp
        else:
            tmp = tmp*tmp*x

    return tmp
```

Direct recursion: first push the converging argument

Then pop it and operate on it

Advantages/Disadvantages of Recursion

- **Advantages:**
  - Some times it is more natural (e.g., Fibonacci)
  - Easier to prove correct (due to its mathematical foundations)
  - Easier to analyse (due to its mathematical foundations)
- **Disadvantages:**
  - Run-time overhead (for tail-recursion, this will depend on the quality of the compiler)
  - Memory overhead (fewer local variables versus stack space for function call)

Print the elements of a list in reverse

- **Write a recursive method to print the elements of an iterable list in reverse order (don't modify it): you have `iter()`, `has_next()`, `next()`**

```
def print_reverse(array: ArrayList) -> None:
    it = iter(array)
    print_reverse_aux(it)
```

```
  def print_reverse_aux(it: ListIterator) -> None:
    if it.has_next():
        element = next(it)      # store element
        print_reverse_aux(it)   # print the rest in reverse
        print(element)          # then print it
```

The auxiliary function is not really needed since the iterator also has an __iter__ method (which returns itself). Thus, this also works:

```
def print_reverse(an_iterable):
    it = iter(an_iterable)
    if it.has_next():
        element = next(it)
        print_reverse(it)
        print(element)
```

Print the elements of a list in reverse

- **Do you remember how we wrote the method iteratively?**

```
def reverse(input_string: str) -> None:
    stack_size = len(input_string)
    the_stack = ArrayStack(stack_size)

    for item in input_string:
        the_stack.push(item)

    while not the_stack.is_empty():
        print(the_stack.pop())
```

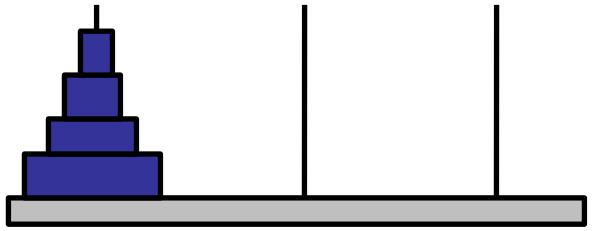
This is what the system stack does for us automatically

Summary

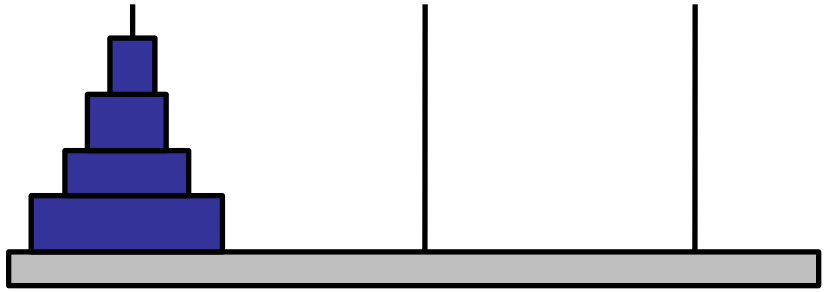
- A bit of notation (unary, binary, n-ary, direct/indirect, tail recursion)
- More complex recursions
- Role of runtime stack in recursion
- Iteration to recursion: straightforward
- Recursion to iteration: might need accumulators or a stack
- Recursion versus iteration: advantages and disadvantages

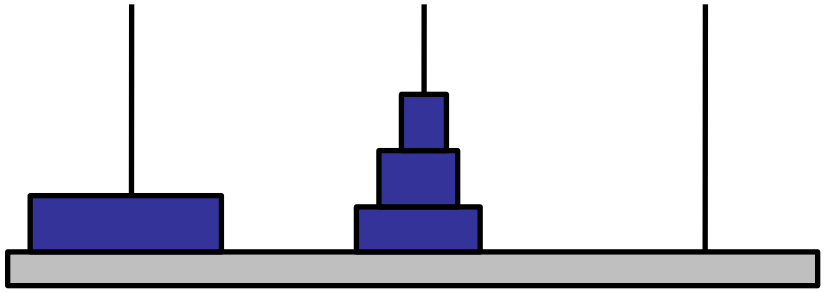
Advanced: Tower of Hanoi

- **Problem :**
  - move **the n discs** from one peg to another, according to the rules
- **Key observations:**
  - bottom disc must have an empty peg to move to
    - it can't go on top of another disc
  - so **top n-1 discs** must be placed on a single peg which is *different* to the peg the bottom disc is on, AND *different* to the peg the bottom disc has to move to.
  - So **top n-1 discs** have to be moved to another peg, then bottom disc is moved, then the **top n-1 discs** are moved on top of the bottom disc again.
- **And don't forget the base case:**
  - one disc: just move it!

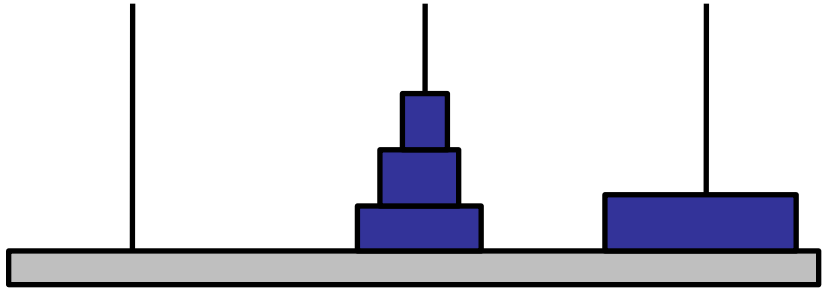


The approach

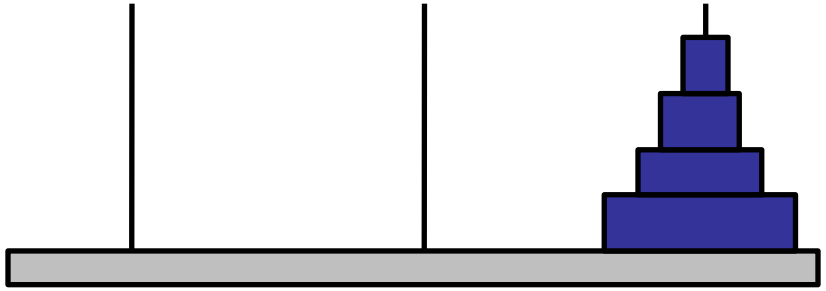




Moved n-1 discs from current peg to middle peg



Moved bottom disc from current peg to final peg



Moved n-1 discs from middle peg to final peg

Tower of Hanoi: recursive solution

```
def tower_Hanoi(number_discs: int, from_peg: int, to_peg: int) -> None:
    intermediate_peg = the_other_peg(from_peg, to_peg)

    if number_discs == 1:
        print(str(from_peg) + " -> " + str(to_peg))
    else:
        tower_Hanoi(number_discs-1, from_peg, intermediate_peg)
        print(str(from_peg) + " -> " + str(to_peg))
        tower_Hanoi(number_discs-1, intermediate_peg, to_peg)
```

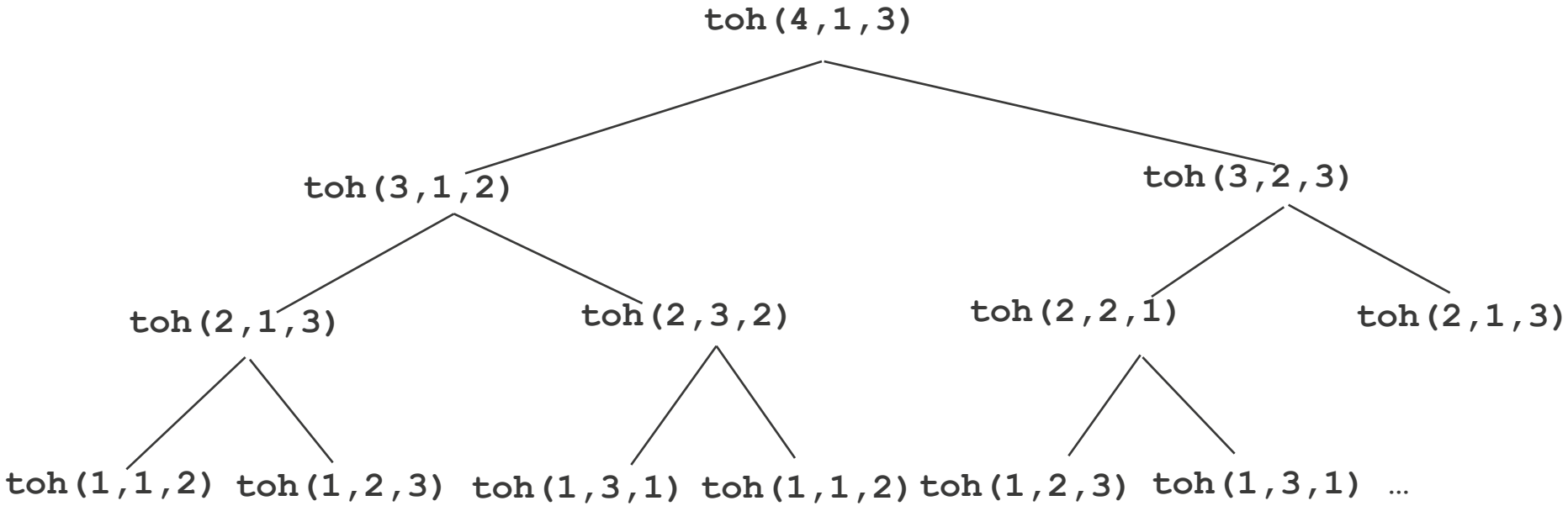
Assume we've written a simple function to return whichever number of 1, 2, 3 is not given to it as an argument

Test for base case

Recursive calls

Convergence

Hanoi's execution: call tree



Complexity? $O(2^n)$

Tower of Hanoi: recursive solution

Output for `towerOfHanoi(4, 1, 3)` should be:

```
1 -> 2
1 -> 3
2 -> 3
1 -> 2
3 -> 1
3 -> 2
1 -> 2
1 -> 3
2 -> 3
2 -> 1
3 -> 1
2 -> 3
1 -> 2
1 -> 3
2 -> 3
```

Tower of Hanoi: some challenges

- **Find an iterative solution**
- **Generalise to *n* pegs**