# Recursion Examples

# Recursion (review)

# Recursion (review)

Scenario: You are waiting in line for a concert. You can't see the front of the line, but you want to know your place in the line.

| | |
|---|---|
| The person at the front, knows they are at the front! | Base case |
| You ask the person in front of you: “what is your place in the line?” | Recursive call |
| When the person in front of you figures it out and tells you, add one to that answer. | Use the solution to the smaller problem |

# Iteration vs. Recursion

- Iteration and recursion are somewhat related
- Converting **iteration to recursion** is formulaic, but converting **recursion to iteration** can be more tricky

**Iterative**

```
def fact_iter(n):
    total, k = 1, 1
    while k <= n:
        total, k = total*k, k+1
    return total
```

$$n! = \prod_{k=1}^{n} k$$

Names: n, total, k, fact_iter

**Recursive**

```
def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n-1)
```

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1)! & \text{otherwise} \end{cases}$$

Names: n, fact

# Sum Digits

Let's implement a recursive function to sum all the digits of `n`. Assume `n` is positive .

```
def sum_digits(n):
        """Calculates the sum of
    the digits `n`.
        >>> sum_digits(8)
        8
        >>> sum_digits(18)
        9
        >>> sum_digits(2018)
        11
    """
        "*** YOUR CODE HERE
***"
```

1. One or more **base cases**
2. One or more **recursive calls** with simpler arguments.
3. **Using the recursive call** to solve our larger problem.

# Sum Digits

```
1   def sum_digits(n):
2       """Calculates the sum of the digits n
3       >>> sum_digits(8)
4       8
5       >>> sum_digits(18)
6       9
7       >>> sum_digits(2018)
8       11
9       """
10      if n < 10:
11          return n
12      else:
13          all_but_last, last = n // 10, n % 10
14          return sum_digits(all_but_last) + last
```

# Order of Recursive Calls

# Cascade

Demo

Goal: Print out a cascading tree of a positive integer `n`.

```
>>> cascade(486)
486
48
4
48
486
>>> cascade(48)
48
4
48
>>> cascade(4)
4
```

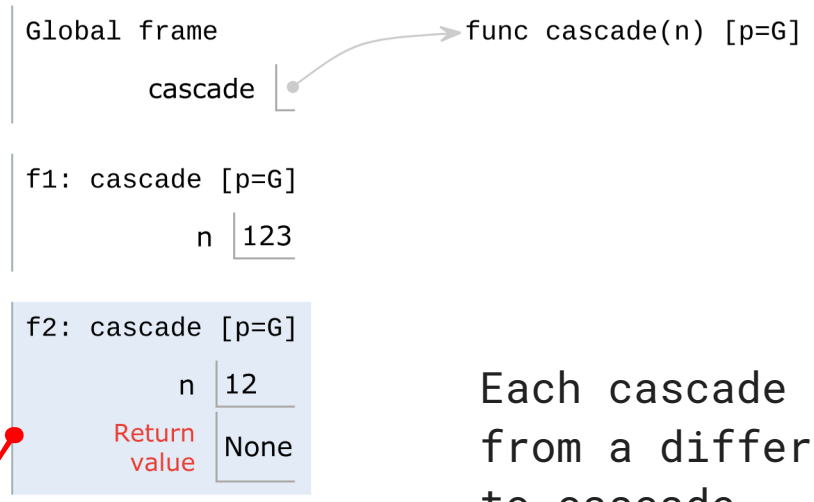
Ideas:

- If `n` is a single digit, just print it out!
- Otherwise, let `cascade(n // 10)` take care of the middle and `print(n)` around it

```
1 def cascade(n):
2     if n < 10:
3         print(n)
4     else:
5         print(n)
6         cascade(n // 10)
7         print(n)
```

# The Cascade Function

```
1  def cascade(n):
2      if n < 10:
3          print(n)
4      else:
5          print(n)
6          cascade(n // 10)
7          print(n)
8
9  cascade(123)
```




**Output**

123
12
1
12

Base case

Each cascade frame is from a different call to cascade.

Until the **Return value** appears, that call has not completed.

Any statement can appear **before** or **after** the recursive call.

Demo

# Two Implementations of Cascade

```
def cascade(n):
    if n < 10:
        print(n)
    else:
        print(n)
        cascade(n // 10)
        print(n)
```

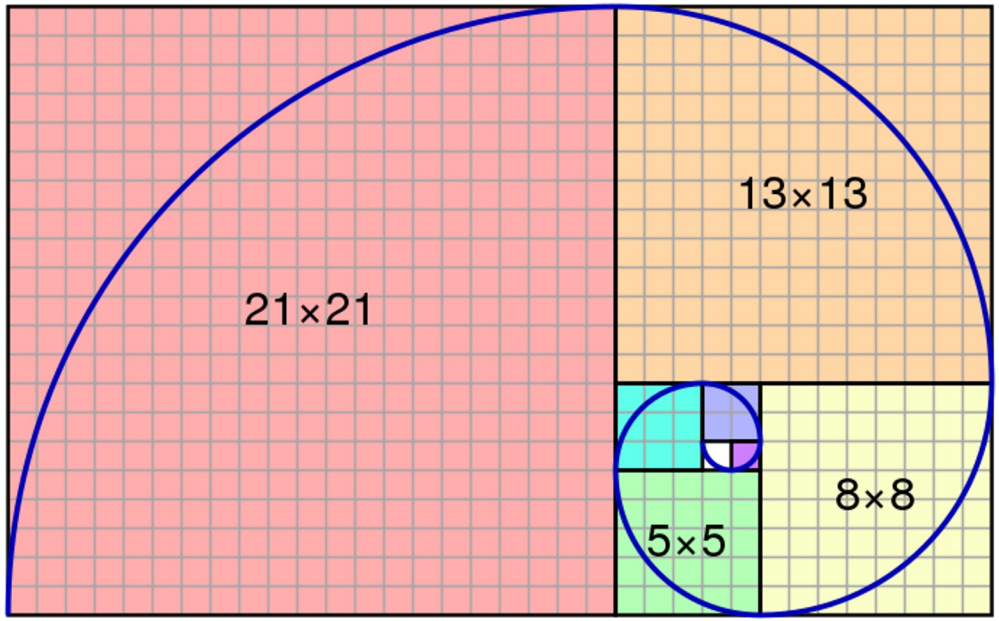
```
def cascade(n):
    print(n)
    if n >= 10:
        cascade(n // 10)
        print(n)
```

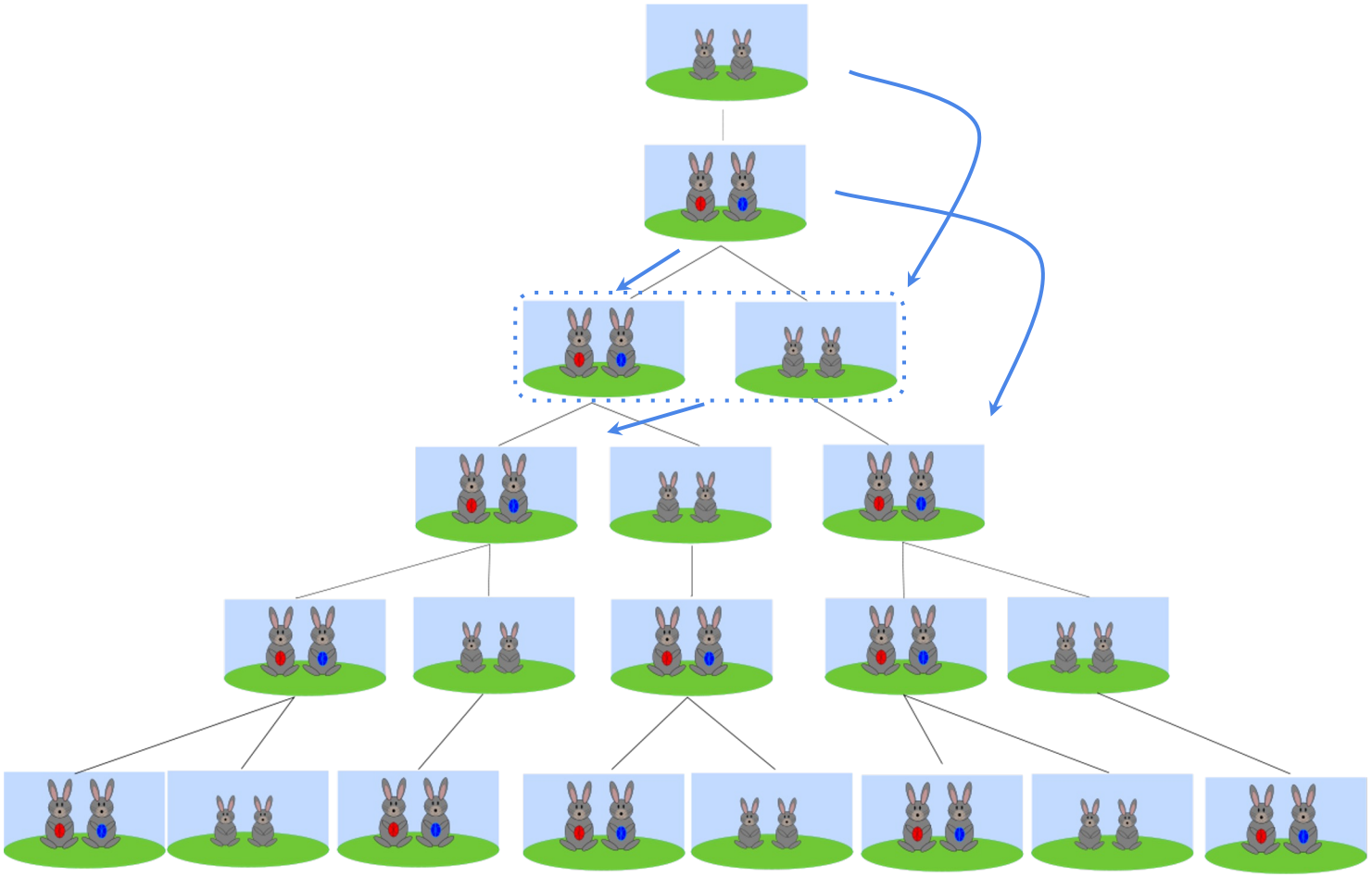
- If two implementations are equally clear, then shorter is usually better
- In this case, the longer implementation might be more clear
- When learning to write recursive functions, put the base case/s first

# Fibonacci

# Fibonacci Sequence

| n | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | ... | 30 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| fib(n) | 0 | 1 | 1 | 2 | 3 | 5 | 8 | 13 | 21 | ... | 832040 |



fib(1) == 1

fib(2) == 1

fib(3) == 2

fib(4) == 3

fib(5) == 5

fib(6) == 8

Fibonacci's rabbits

Demo

# Fibonacci

Goal: Return the `n`th Fibonacci number.

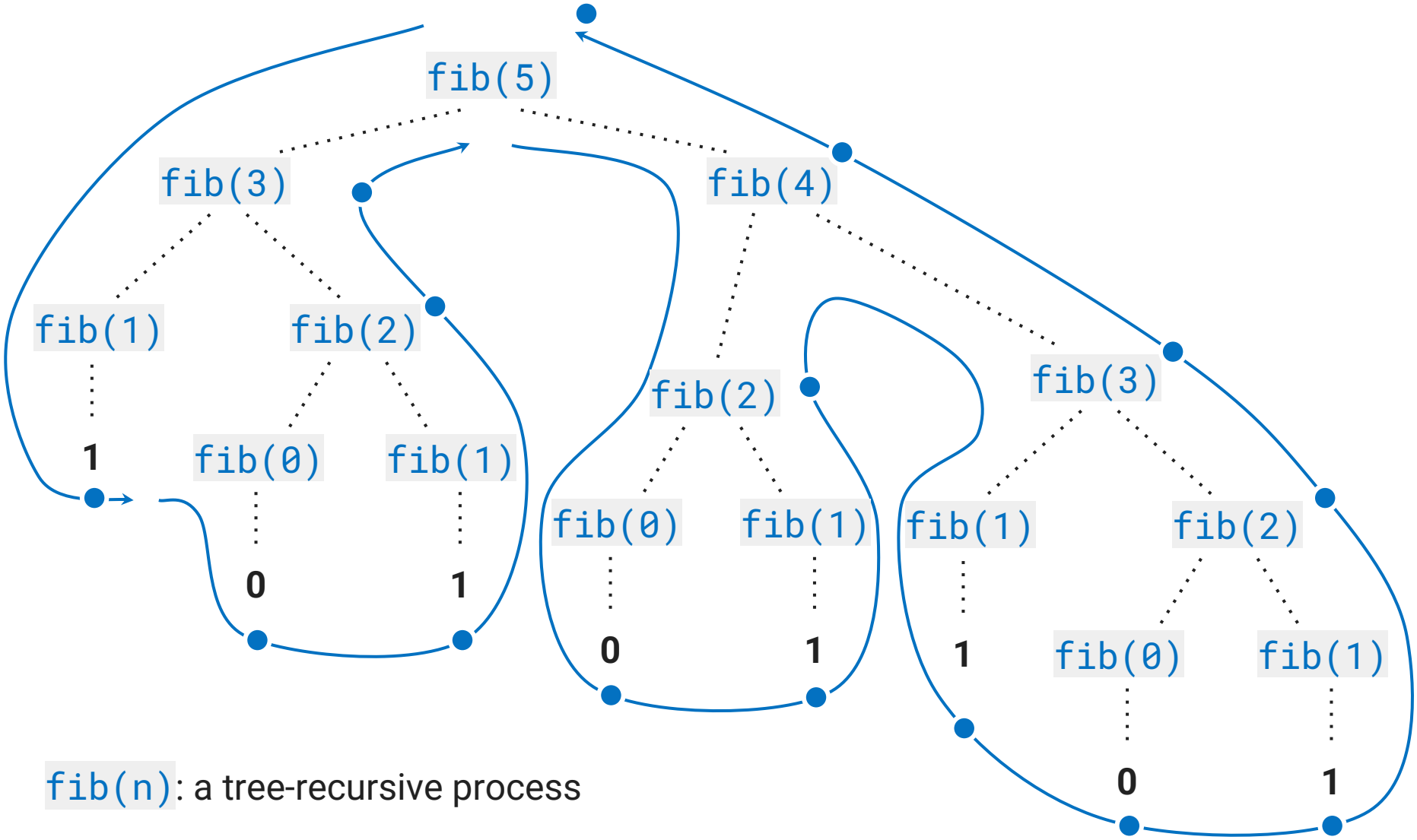
| `n` | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | ... | 30 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| `fib(n)` | 0 | 1 | 1 | 2 | 3 | 5 | 8 | 13 | 21 | ... | 832040 |

Ideas:

- The first two Fibonacci numbers are known; if we ask for the 0th or 1st Fibonacci number, we know it immediately
- Otherwise, we sum up the previous two Fibonacci numbers

```
1 def fib(n):
2   if n == 0:
3       return 0
4   elif n == 1:
5       return 1
6   else:
7       return fib(n - 2) + fib(n - 1)
```

# Fibonacci Call Tree



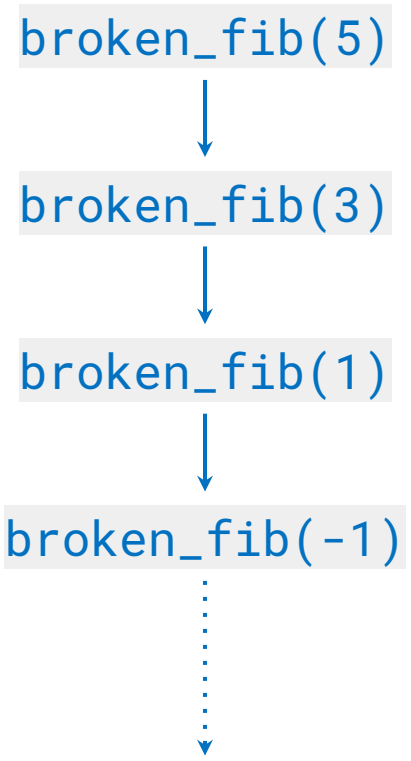

`fib(n)`: a tree-recursive process

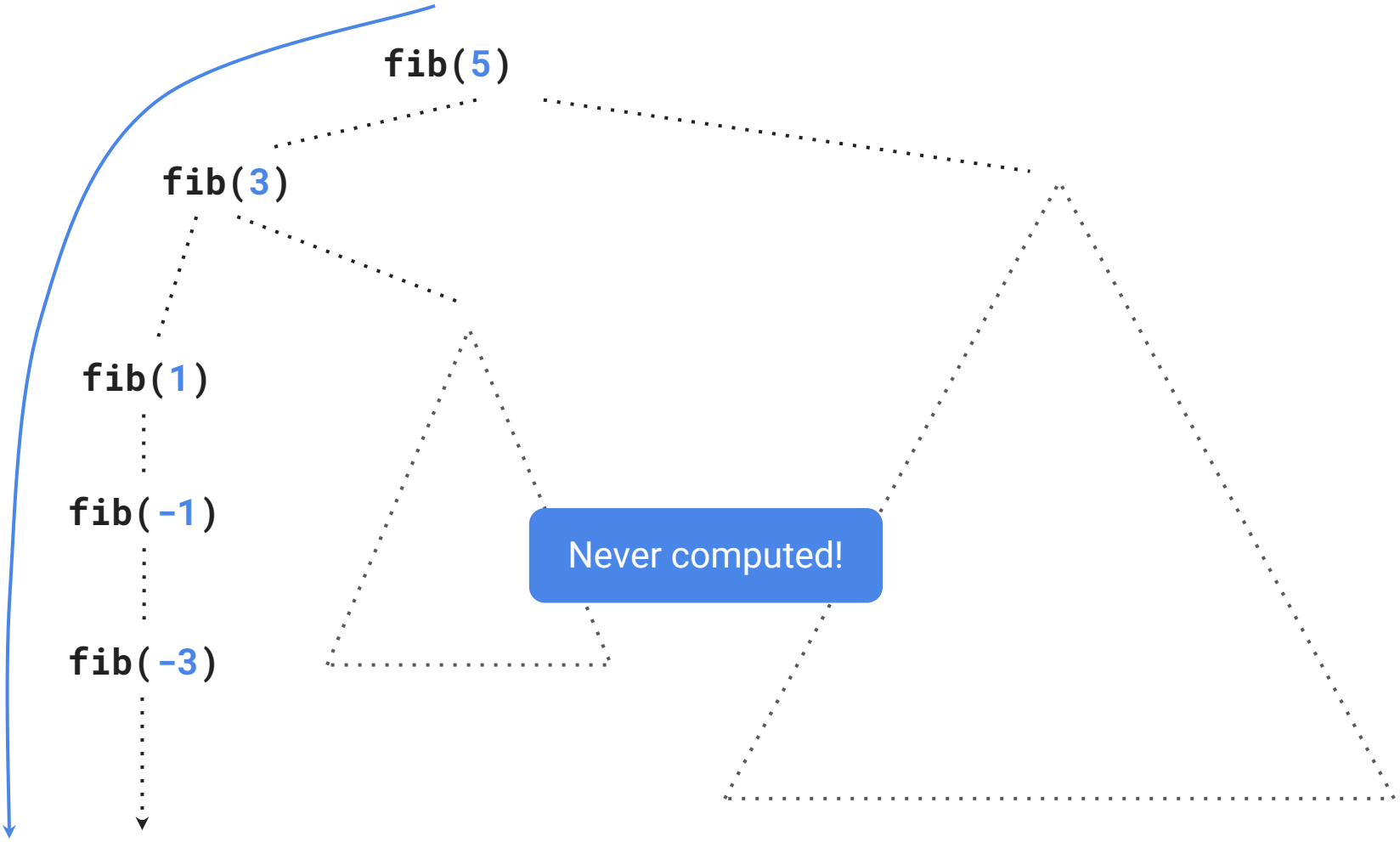
# Broken Fibonacci

```
1 def broken_fib(n):
2       if n == 0:
3          return 0
4       # Missing base case!
5       else:
6          return broken_fib(n - 2) +
                  broken_fib(n - 1)
```

a. Wrong value
b. Error

```
>>> broken_fib(5)
Traceback (most recent call last):
...
RecursionError: maximum recursion
depth exceeded in comparison
```






Broken `fib(n)`

# Counting Partitions

# Count Partitions

Goal: Count the number of ways to give out `n` ( > 0) pieces of chocolate if nobody can have more than `m` (> 0) pieces.

*"How many different ways can I give out 6 pieces of chocolate if nobody can have more than 4 pieces?"*

```
>>> count_part(6, 4)
9
```

Largest Piece: 4

2 + 4 = 6
1 + 1 + 4 = 6

Largest Piece: 3

3 + 3 = 6
1 + 2 + 3 = 6
1 + 1 + 1 + 3 = 6

Largest Piece: 2

2 + 2 + 2 = 6
1 + 1 + 2 + 2 = 6
1 + 1 + 1 + 1 + 2 = 6

Largest Piece: 1

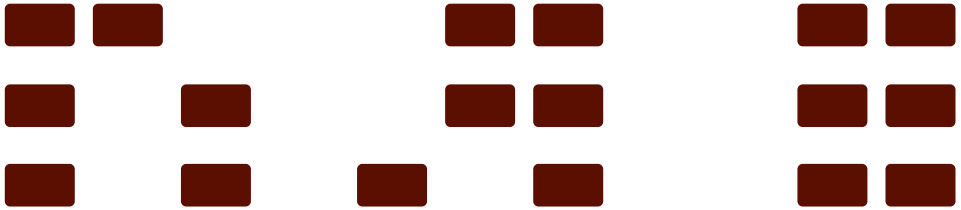
1 + 1 + 1 + 1 + 1 + 1 = 6

# Count Partitions

2 + 4
1 + 1 + 4

3 + 3
1 + 2 + 3
1 + 1 + 1 + 3

2 + 2 + 2
1 + 1 + 2 + 2
1 + 1 + 1 + 1 + 2



1 + 1 + 1 + 1 + 1 + 1

# Count Partitions

Ideas:

Find simpler instances of the problem

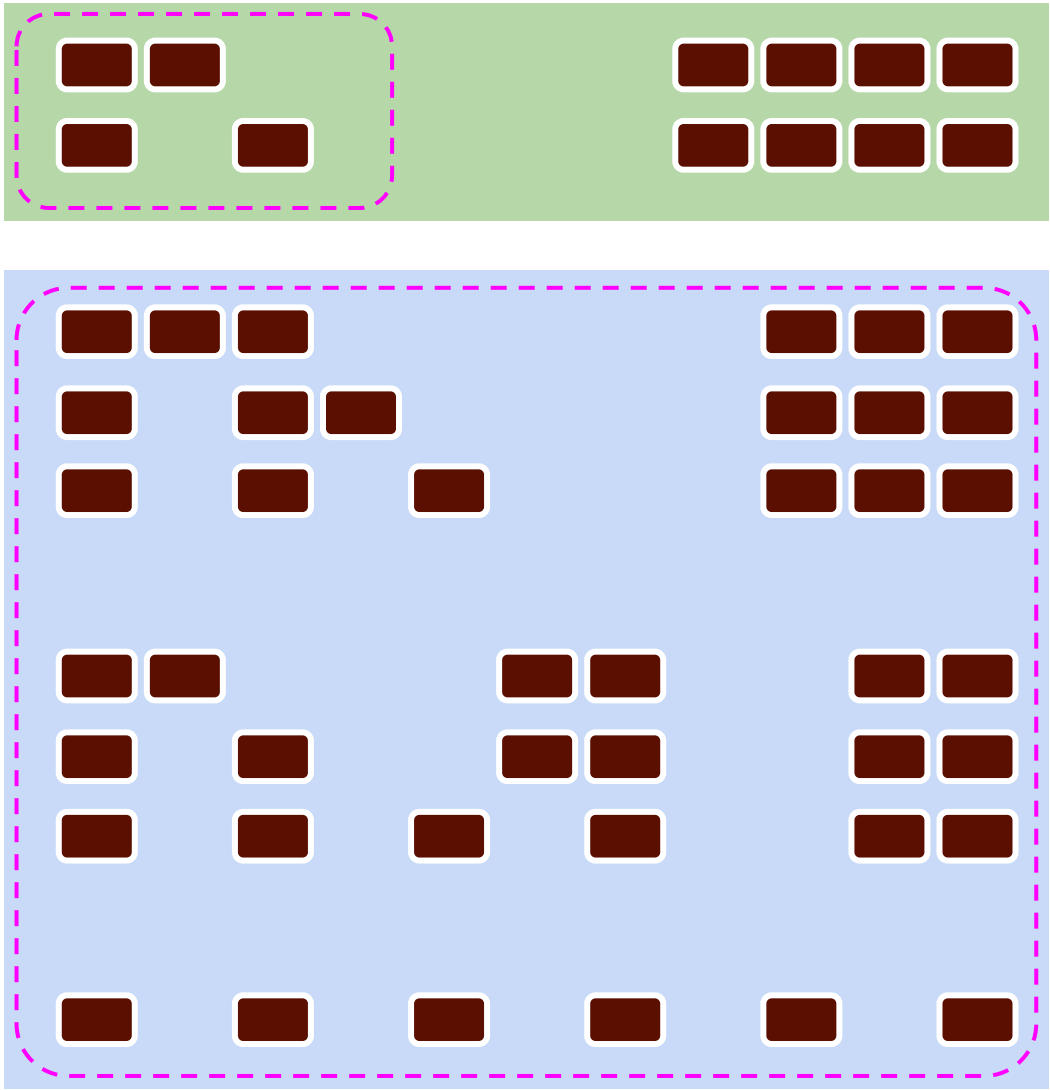
Explore two possibilities:

- Use a 4
- Don't use a 4

Solve two simpler problems:

- `count_part(2, 4)`
- `count_part(6, 3)`

Sum up the results of these smaller problems!



# Count Partitions

Ideas:

Find simpler instances of the problem

Explore two possibilities:

- Use a 4
- Don't use a 4

Solve two simpler problems:

- `count_part(2, 4)`
- `count_part(6, 3)`

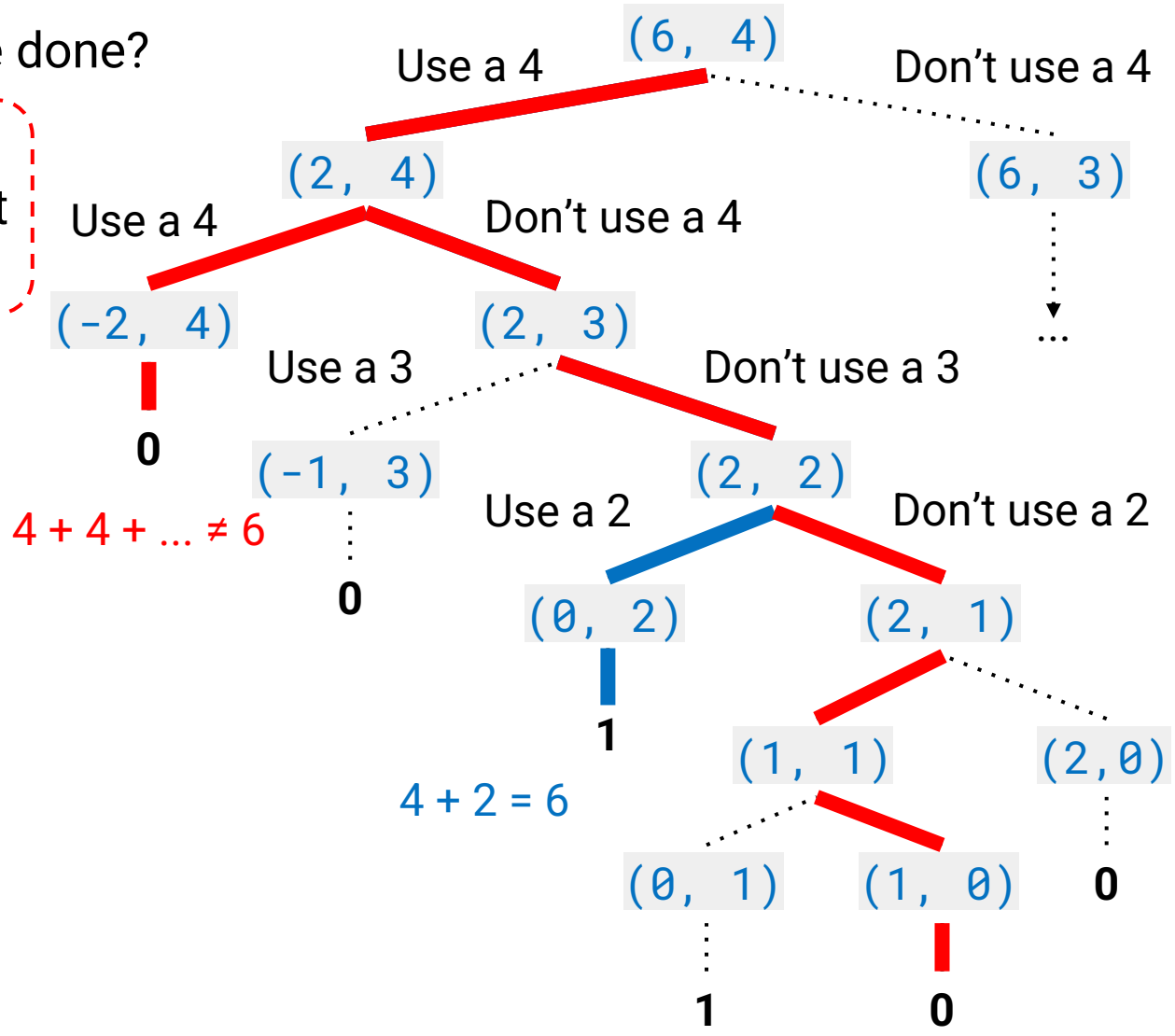
Sum up the results of these smaller problems!

```
1 def count_part(n, m):
2       if
3       else:
4         with_m = count_part(n-m, m)
5         wo_m = count_part(n, m - 1)
6         return with_m + wo_m
```

# Count Partitions

How do we know we're done?

- If `n` is negative, then we cannot get to a valid partition
- If `n` is 0, then we have arrived at a valid partition
- If the largest piece we can use is 0, then we cannot get to a valid partition



# Count Partitions

Demo

Ideas:

Explore two possibilities:

- Use a 4
- Don't use a 4

Solve two simpler problems:

- `count_part(2, 4)`
- `count_part(6, 3)`

Sum up the results of these smaller problems!

```
def count_part(n, m):
    if n == 0:
        return 1
    elif n < 0:
        return 0
    elif m == 0:
        return 0
    else:
        with_m = count_part(n-m, m)
        wo_m = count_part(n, m - 1)
        return with_m + wo_m
```

How do we know we're done?

- If `n` is 0, then we have arrived at a valid partition
- If `n` is negative, then we cannot get to a valid partition
- If the largest piece we can use is 0, then we cannot get to a valid partition

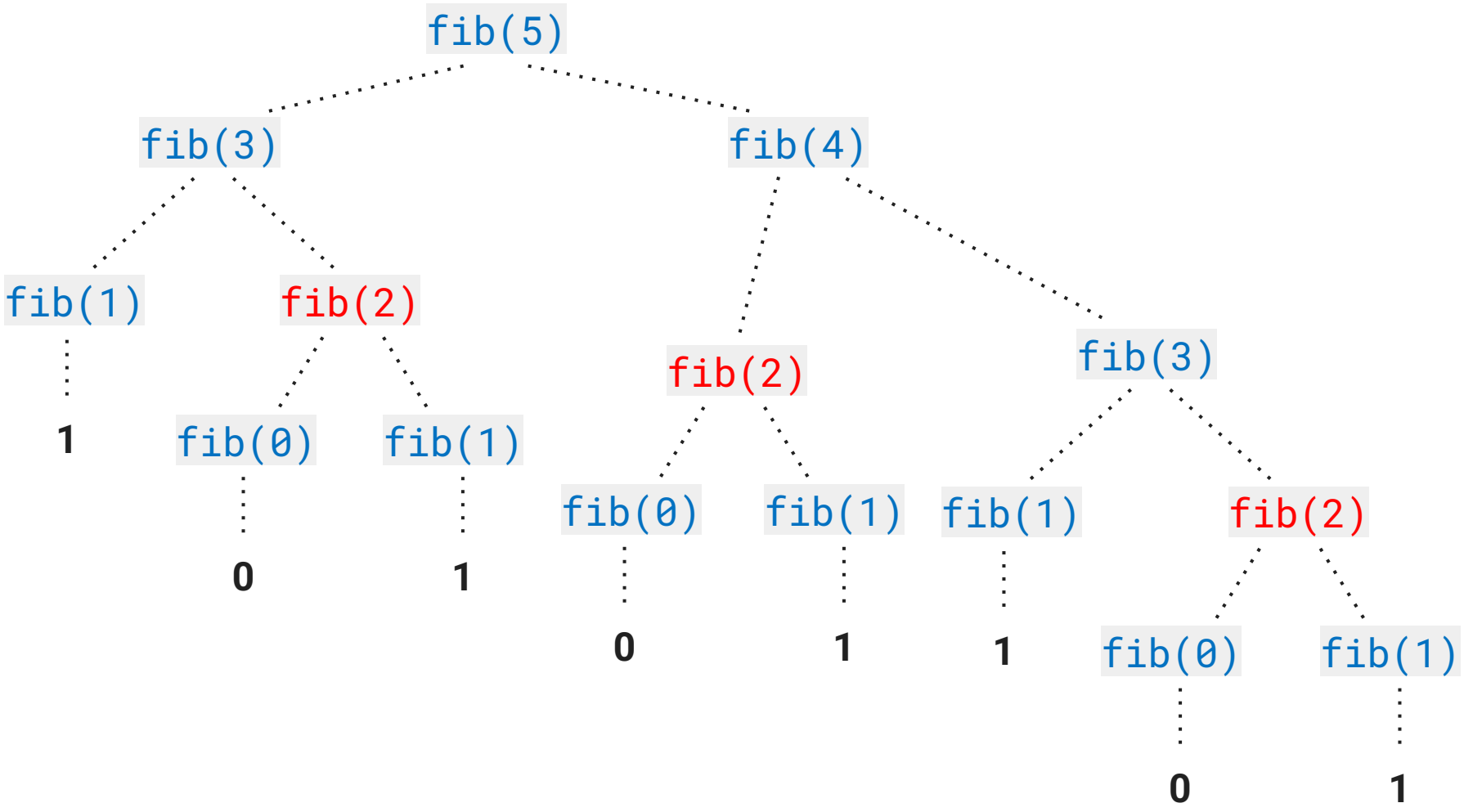
# Takeaways

- Tree recursion allows you to **explore different possibilities**
- Oftentimes, the recursive calls for tree recursion represent different choices
  - One such choice is “do I use this value, or do I try another?”
- Sometimes it is easier to start with the recursive cases, and see which base cases those lead you to

# If Time - Speeding Up Recursion (Teaser for the ~Future~)

# Back to Fib

Demo



# Basic Idea to Improve:

```
1 def better_fib(n):
2       if n == 0:
3          return 0
4       elif n == 1:
5          return 1
6       elif already called better_fib(n):
7          return stored value
8       else:
9          store & return better_fib(n - 2) + better_fib(n - 1)
```

# Summary

- **Recursion** has three main components
  - **Base case/s**: The simplest form of the problem
  - **Recursive call/s**: Smaller version of the problem
  - Use the solution to the smaller version of the problem to arrive at the solution to the original problem
- When working with recursion, use *functional abstraction*: assume the recursive call gives the correct result
- **Tree recursion** makes multiple recursive calls and explores different choices
- Use doctests and your own examples to help you figure out the simplest forms and how to make the problem smaller