Recursion

Src: Several slides in this deck borrowed from Kelly Rivers, Margaret Reid-Miller, CMU

Learning Objectives

- Define and recognize **base cases** and **recursive cases** in recursive code
- Read and write basic **recursive code**
- Trace over recursive functions that use **multiple recursive calls** with Fibonacci numbers

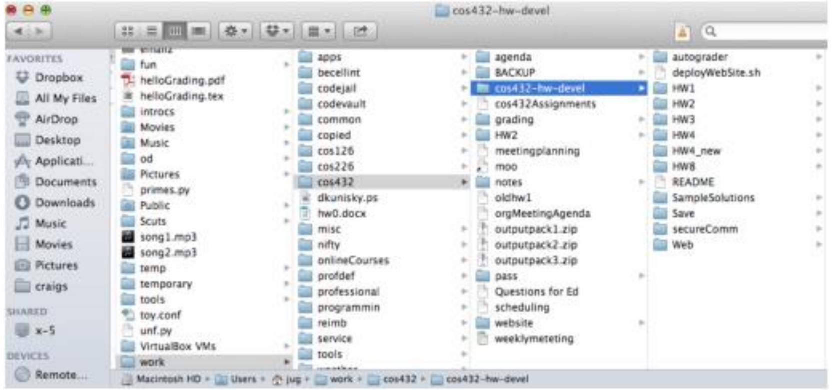
Concept of Recursion

Concept of Recursion

Recursion is a concept that shows up commonly in computing, and in the world.

Core idea: an idea X is recursive if X is used **in its own definition**.

Example: fractals; nesting dolls; your computer's file system



Recursion in Algorithms

When we use recursion in algorithms, it's generally used to implement **delegation** in problem solving, sometimes as an alternative to iteration.

To solve a problem recursively:

1. Find a way to make the problem **slightly smaller**
2. Delegate solving that problem to someone else
3. When you get the smaller-solution, **combine it** with the remaining part of the problem

Example Iteration vs. Recursion

How do we add the numbers on a deck of cards?

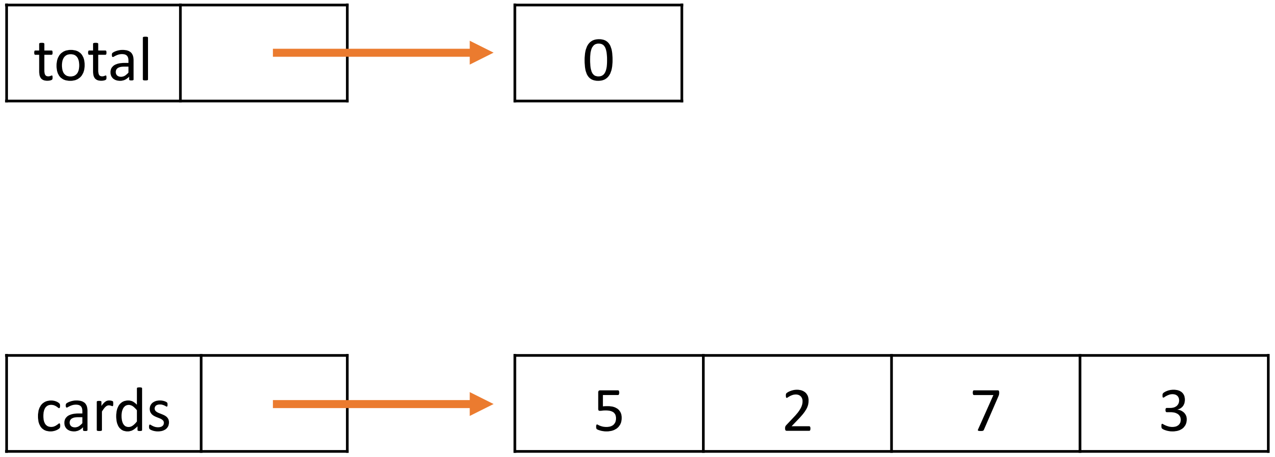
Iterative approach: keep track of the total so far, iterate over the cards, add each to the total.

Recursive approach: take a card off the deck, **delegate adding the rest of the deck to someone else**, then when they give you the answer, add the remaining card to it.

Implementing Iteration

Let's look at how we'd add the deck of four cards using **iteration**.

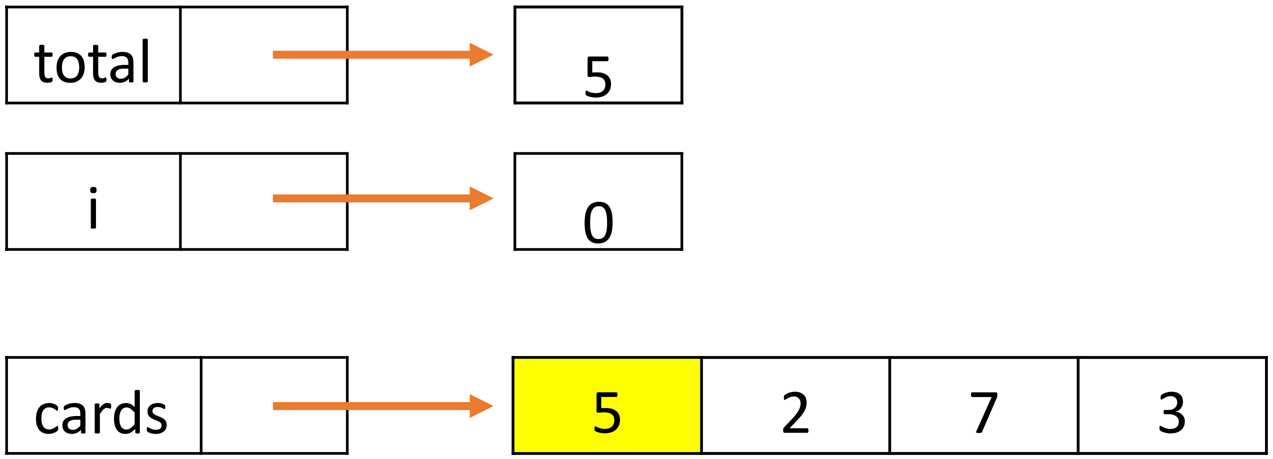
Pre-Loop:



Implementing Iteration

Let's look at how we'd add the deck of four cards using **iteration**.

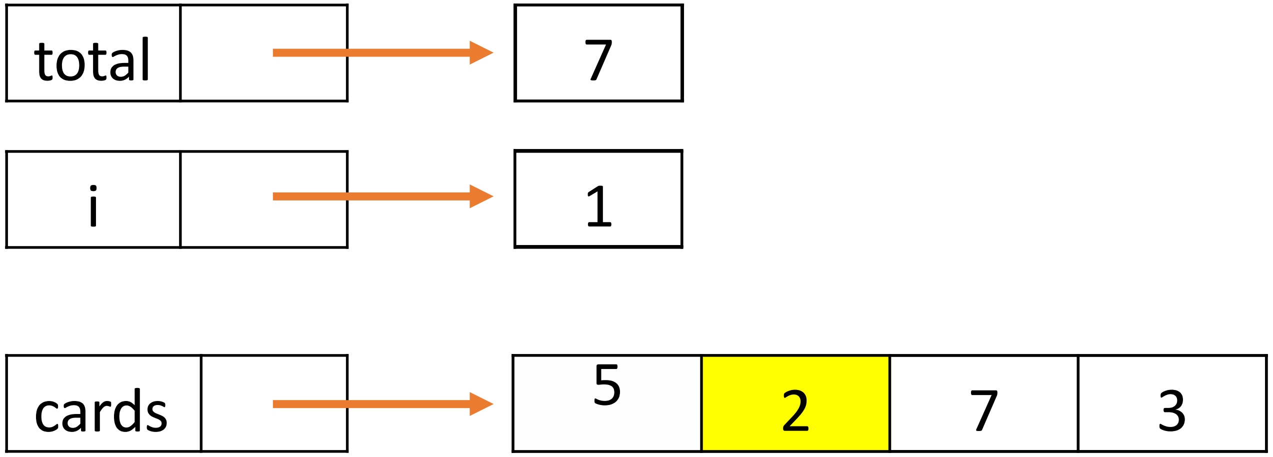
First iteration:



Implementing Iteration

Let's look at how we'd add the deck of four cards using **iteration**.

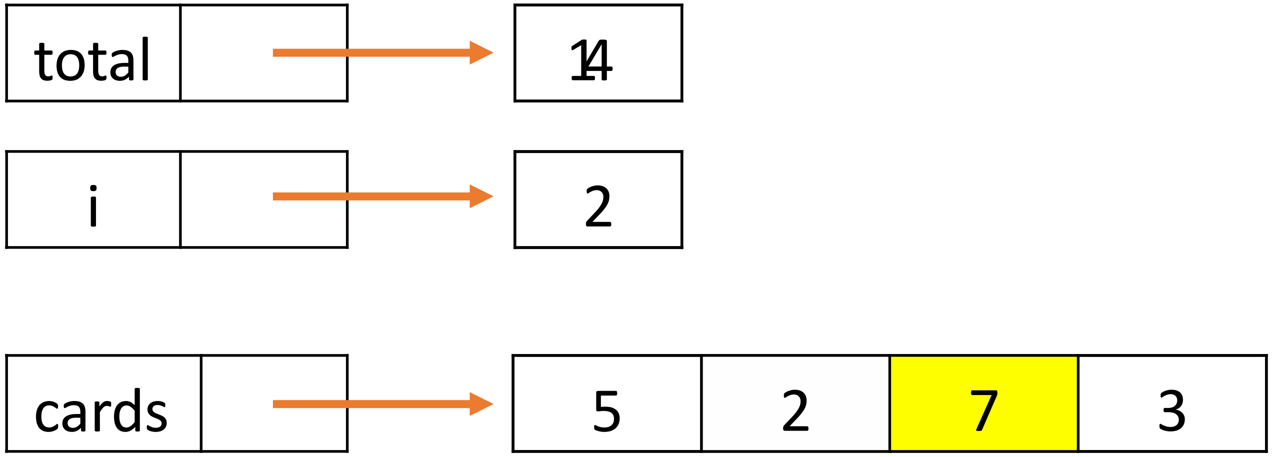
Second iteration:



Implementing Iteration

Let's look at how we'd add the deck of four cards using **iteration**.

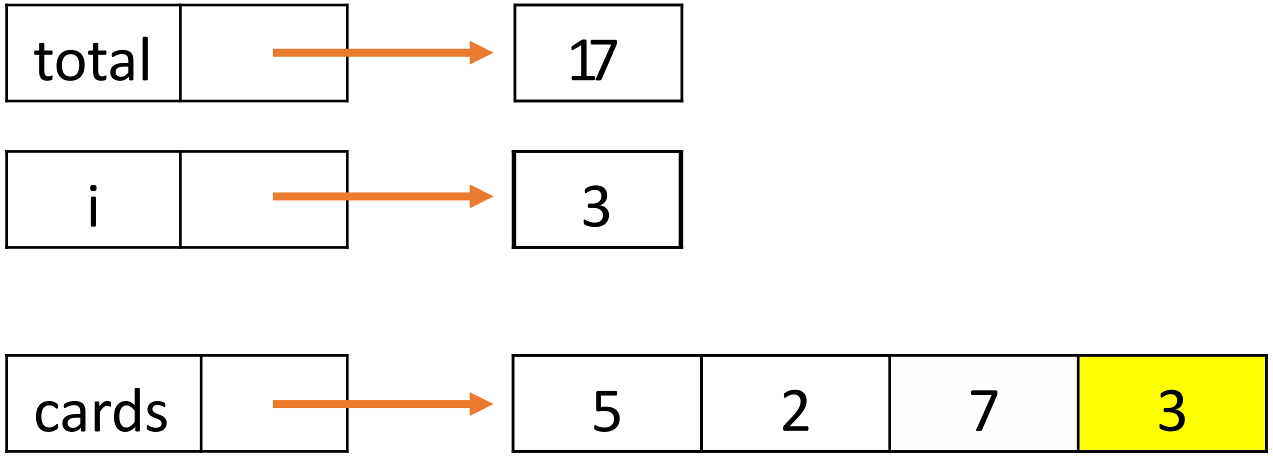
Third iteration:



Implementing Iteration

Let's look at how we'd add the deck of four cards using **iteration**.

Fourth iteration:



And we're done!

Iteration in Code

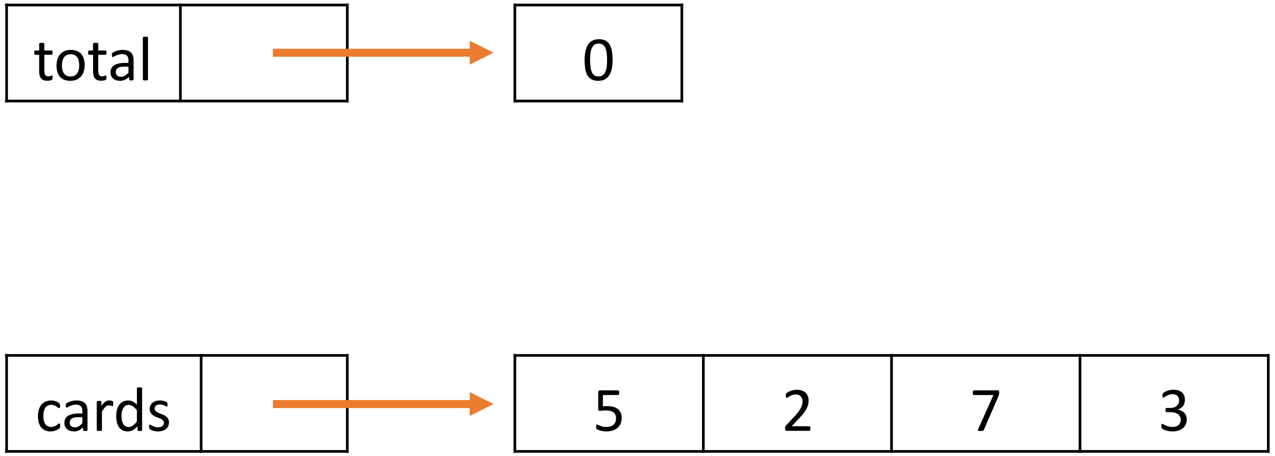
We could implement this in code with the following function:

```
def iterativeAddCards(cards):
    total = 0
    for i in range(len(cards)):
        total = total + cards[i]
    return total
```

Implementing Recursion

Now let's add the same deck of cards using **recursion**.

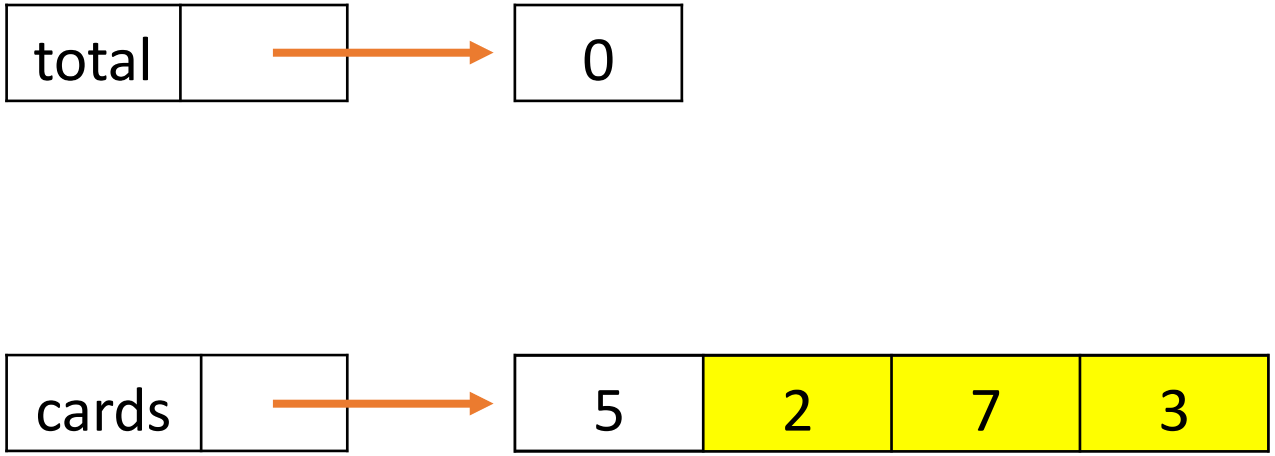
Start State:



Implementing Recursion

Now let's add the same deck of cards using **recursion**.

Make the problem smaller:

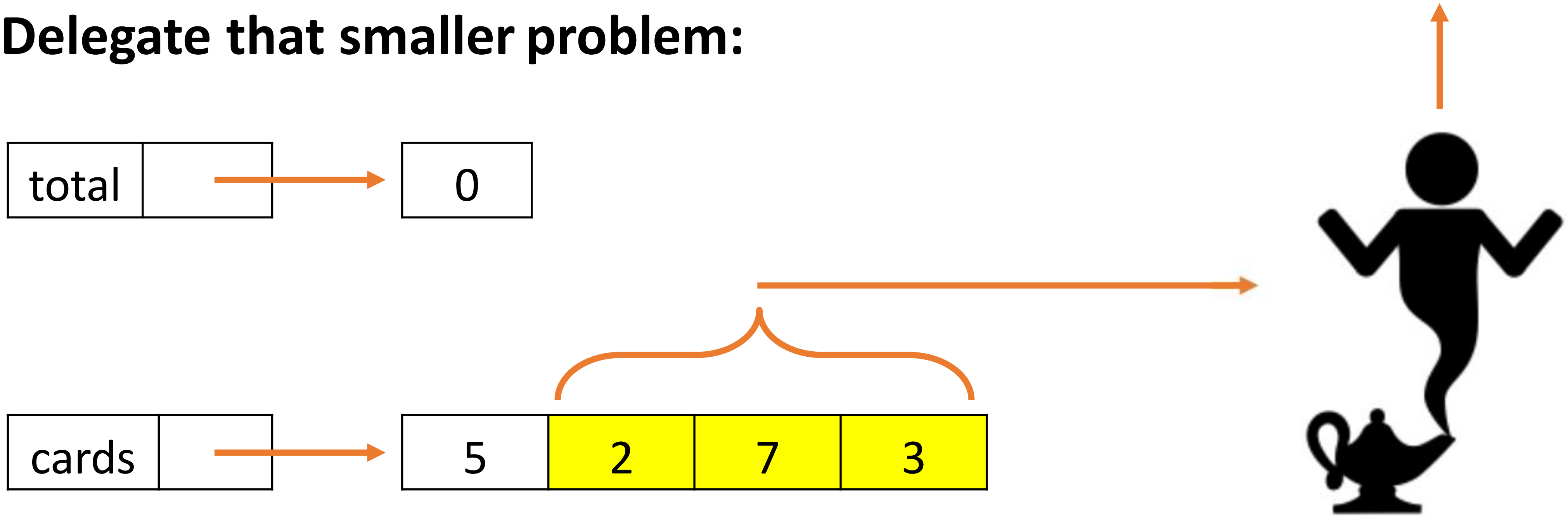


Implementing Recursion

Now let's add the same deck of cards using **recursion**.

Delegate that smaller problem:

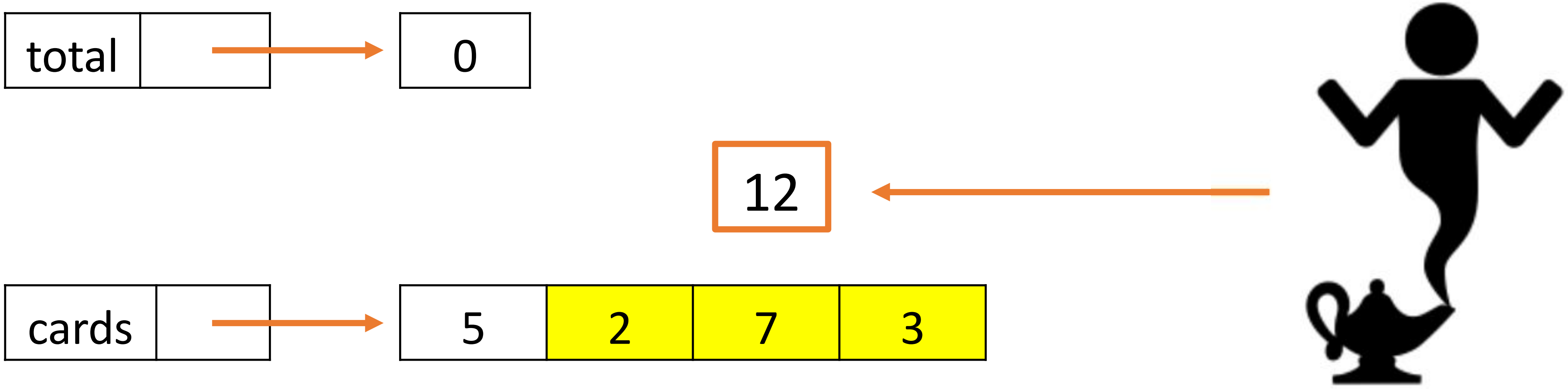
This is the Recursion Genie. They can solve problems, but only if the problem has been made slightly smaller than the start state.



Implementing Recursion

Now let's add the same deck of cards using **recursion**.

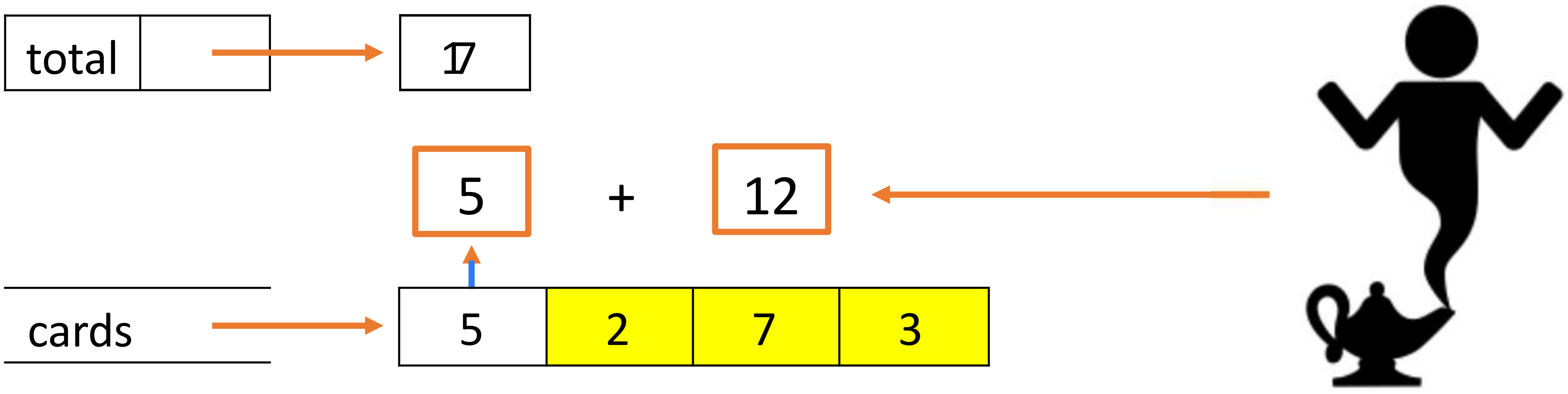
Get the smaller problem's solution:



Implementing Recursion

Now let's add the same deck of cards using **recursion**.

Combine the leftover part with the smaller solution:



And we're done!

Recursion in Code

Now let's implement the recursive approach in code.

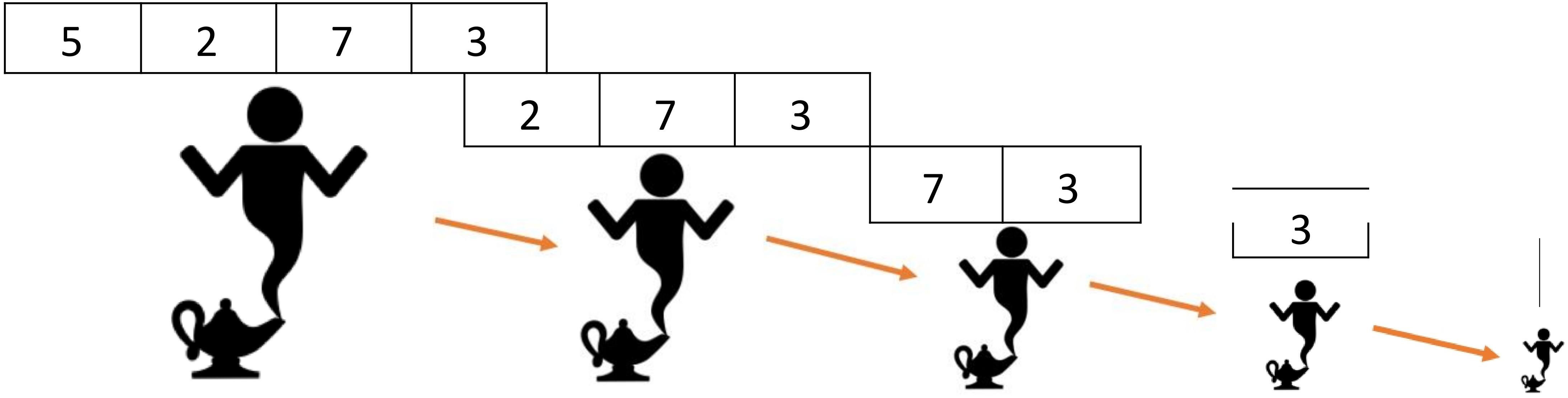
```
def recursiveAddCards(cards):
    smallerProblem = cards[1:]
    smallerResult = ??? # how to call the genie?
    return cards[0] + smallerResult
```

Base Cases and Recursive Cases

Big Idea #1: The Genie is the Algorithm Again!

We don't need to make a new algorithm to implement the Recursion Genie. Instead, we can just **call the function itself** on the slightly-smaller problem.

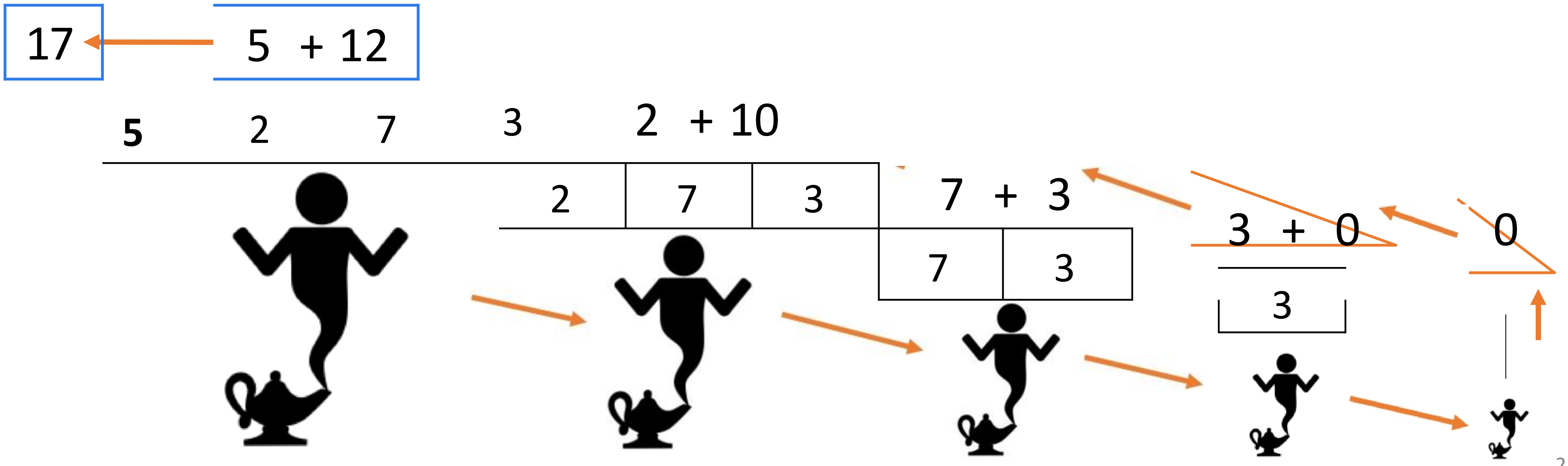
Every time the function is called, the problem gets smaller again. Eventually, the problem reaches a state where we can't make it smaller. We'll call that the **base case**.



Big Idea #2: Base Case Builds the Answer

When the problem gets to the base case, the answer is immediately known. For example, in adding a deck of cards, the sum of an empty deck is 0.

That means the base case can solve the problem **without delegating**. Then it can pass the solution back to the prior problem and start the chain of solutions.



Recursion in Code – Recursive Call

To update our recursion code, we first need to add the call to the function itself.

```
def recursiveAddCards(cards):
    smallerProblem = cards[1:]
    smallerResult = recursiveAddCards(smallerProblem)
    return cards[0] + smallerResult
```

Recursion in Code – Base Case

We also need to add in the **base case**, as an explicit instruction about what to do when the problem cannot be made any smaller.

```
def recursiveAddCards(cards):
    if cards == [ ]:
      return  0
    else:
        smallerProblem = cards[1:]
        smallerResult = recursiveAddCards(smallerProblem)
        return cards[0] + smallerResult
```

Every Recursive Function Includes Two Parts

The two big ideas we just saw are used in **all** recursive algorithms.

1. **Base case**(s) (non-recursive):
   One or more simple cases that can be solved directly (with no further work).

2. **Recursive case**(s):
   One or more cases that require solving "simpler" version(s) of the original problem.
   By "simpler" we mean smaller/shorter/closer to the base case.

Identifying Cases in `addCards(cards)`

Let's locate the base case and recursive case in our example.

```
def recursiveAddCards(cards):
    if cards == [ ]:
        return 0
    else:
        smallerProblem = cards[1:]
        smallerResult = recursiveAddCards(smallerProblem)
        return cards[0] + smallerResult
```

base case

recursive case

Python Tracks Recursion with the Stack

Recall back when we learned about functions, how we used the **stack** to keep track of nested operations.

Python also uses the stack to track recursive calls!

Because each function call has its own set of **local variables**, the values across functions don't get confused.

Trace the Stack

```
recursiveAddCards([5, 2, 7, 3])
↓
recursiveAddCards([2, 7, 3])
↓
recursiveAddCards([7, 3])
↓
recursiveAddCards([3])
↓
recursiveAddCards([ ])
```

return
0
Call 5 - [ ]
Call 4 - [3]
Call 3 - [7, 3]
Call 2 - [2, 7, 3]
Call 1 - [5, 2, 7, 3]
Stack

Trace the Stack

`recursiveAddCards([5, 2, 7, 3])`	17
`recursiveAddCards([2, 7, 3])`	12
`recursiveAddCards([7, 3])`	10
`recursiveAddCards([3])`	3
`recursiveAddCards([ ])`	0

Stack

Programming with Recursion

Recipe for Writing Recursive Functions

Thinking of recursive algorithms can be tricky at first. Here's a recipe you can follow that might help.

1. Write an `if` statement (Why?)

    2 cases: base (may be more than one base case) and recursive

2. Handle simplest case - the base case(s)

    No recursive call needed (that's why it is the **base case!**)

3. Write the recursive call

    Input to call must be slightly simpler/smaller to move towards the base case

4. **Be optimistic:** Assume the recursive call works!

    Ask yourself: What does it do?
    Ask yourself: How does it help?

General Recursive Form

In fact, most of the simple recursive functions you write can take the following form:

```
def recursiveFunction(problem):
    if problem == ???: # base case is the smallest value
        return ____ # something that isn't recursive
    else:
        smallerProblem = ??? # make the problem smaller
        smallerResult = recursiveFunction(smallerProblem)
        return ____ # combine with the leftover part
```

Example: factorial

Assume we want to implement factorial recursively. Recall that:

`x! = x*(x-1)*(x-2)*...*2*1`

We could rewrite that as...

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

What's the **base case**?

`x == 1`

Or maybe `x == 0`...

What's the **smaller problem**?

`x - 1`

How to **combine it**?

Multiply result of `(x-1)!` by `x`

Writing Factorial Recursively

We can take these algorithmic components and combine them with the general recursive form to get a solution.

```
def factorial(x):
    if x == 0: # base case
        return 1 # something not recursive
    else:
        smaller = factorial(x - 1) # recursive call
        return x * smaller # combination
```

Sidebar: Infinite Recursion Causes `RecursionError`

What happens if you call a function on an input that will never reach the base case? **It will keep calling the function forever!**

Example: `factorial(6.5)`

Python keeps track of how many function calls have been added to the stack. If it sees there are too many calls, it raises a `RecursionError` to stop your code from repeating forever.

If you encounter a `RecursionError`, check a) whether you're making the problem smaller each time, and b) whether the input you're using will ever reach the base case.

Activity: `power(base, exp)`

You do: assume we want to recursively compute the value of $base^{exp}$, where `base` and `exp` are both non-negative integers. We'll need to pass both of those values into the recursive function.

What should the **base case** of `power(base, exp)` check in the if statement?

When you have an answer, fill in the blanks.

Example: power(base, exp)

Let's write the function!

```
def power(base, exp):
    if ____________: # base case
        return ________
    else: # recursive case
        smaller = power(_______, ________)
        return ______________
```

Example: `power(base, exp)`

We make the problem smaller by recognizing that $base^{exp} = base * base^{exp-1}$.

```
def power(base, exp):
    if exp == 0: # base case
        return 1
    else: # recursive case
        smaller = power(base, exp-1)
        return base * smaller
```

Example: countVowels(s)

Let's do one last example. Recursively count the number of vowels in the given string.

```
def countVowels(s):
    if ____________: # base case
        return ________
    else: # recursive case
        smaller = countVowels(_______)
        return ______________
```

Example: `countVowels(s)`

We make the string smaller by removing one letter. Change the code's behavior based on whether the letter is a vowel or not.

```
def countVowels(s):
    if s == "": # base case
        return 0
    else: # recursive case
        smaller = countVowels(s[1:])
        if s[0] in "AEIOU":
            return 1 + smaller
        else:
            return smaller
```

Example: countVowels(s)

An alternative approach is to make **multiple recursive cases** based on the smaller part.

```
def countVowels(s):
    if s == "": # base case
        return 0
    elif s[0] in "AEIOU": # recursive case
        smaller = countVowels(s[1:])
        return 1 + smaller
    else:
        smaller = countVowels(s[1:])
        return smaller
```

Multiple Recursive Calls

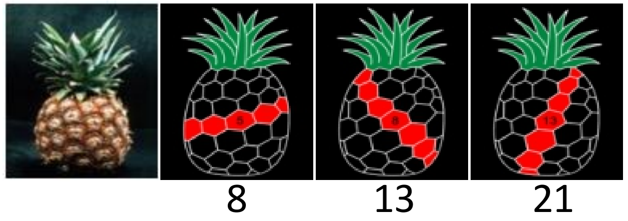
Multiple Recursive Calls

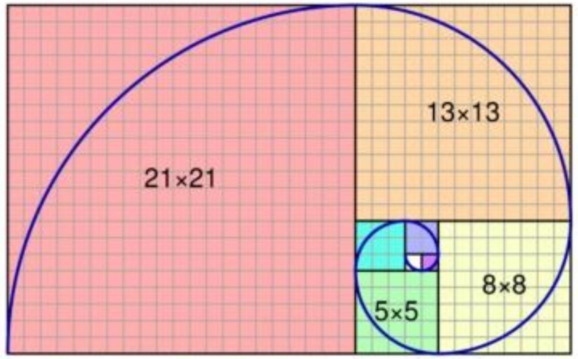
So far, we've used just one recursive call to build up our answer.

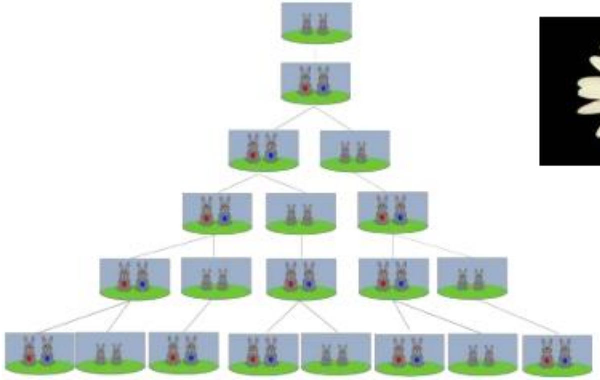
The real **conceptual** power of recursion happens when we need more than one recursive call!

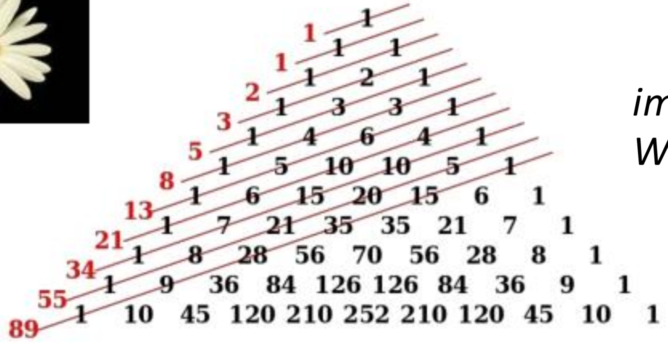
Example: Fibonacci numbers

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, etc.









images from Wikipedia

Code for Fibonacci Numbers

The Fibonacci number pattern goes as follows:

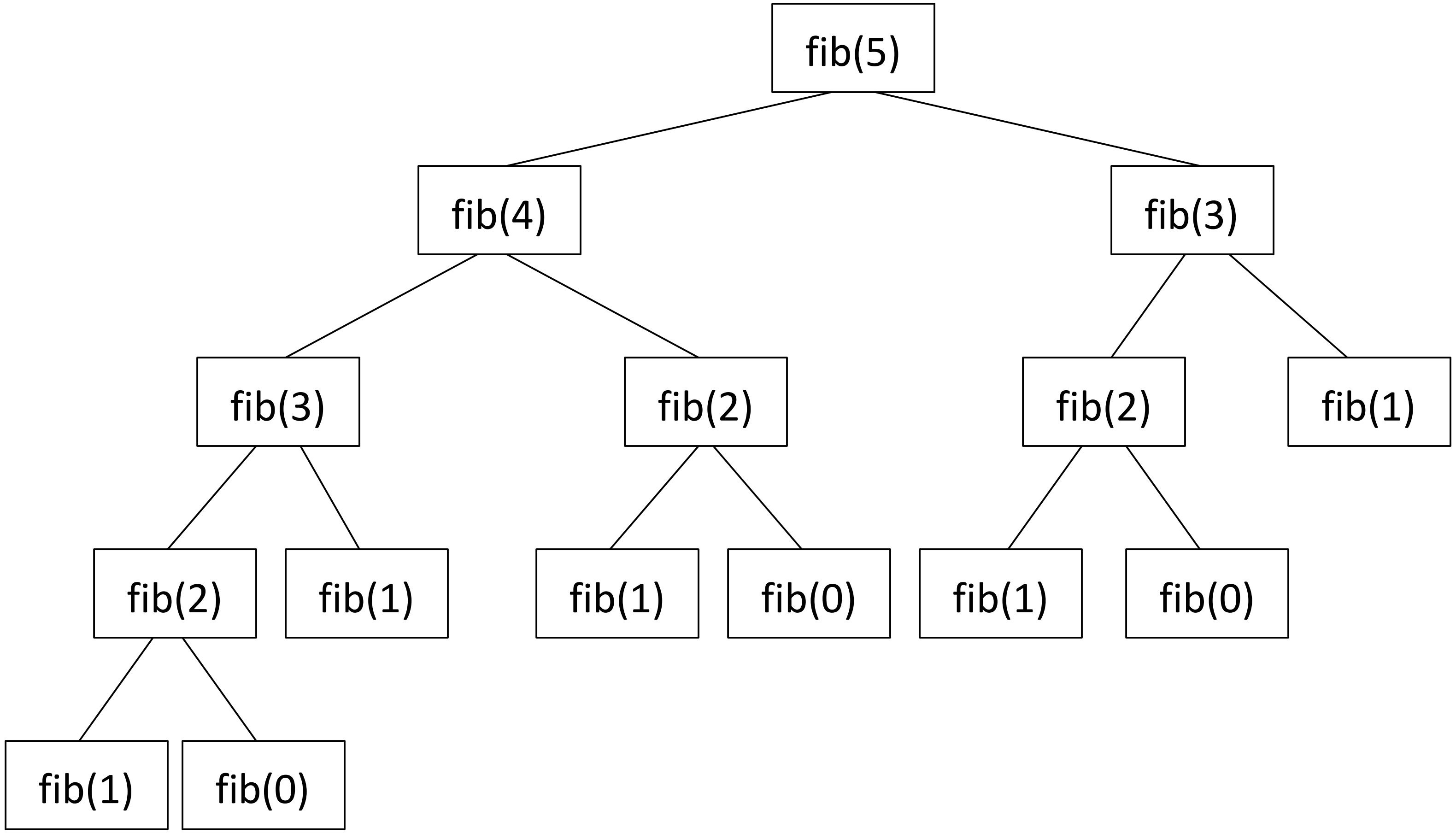
F(0) = 0
F(1) = 1
F(n) = F(n-1) + F(n-2), n > 1

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

Two recursive calls!

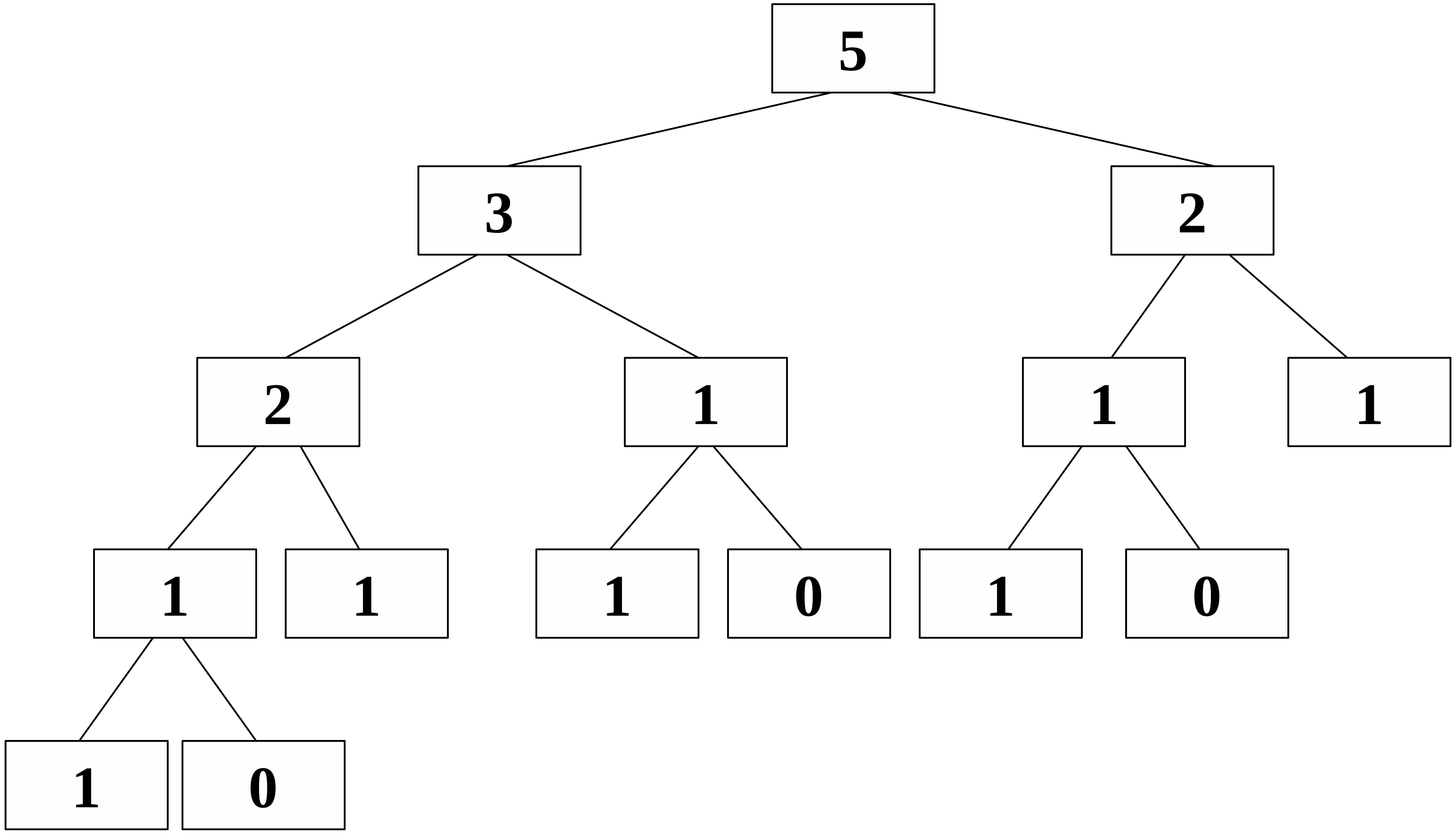
Fibonacci Recursive Call Tree

fib(0) = 0
fib(1) = 1
fib(n) = fib(n-1) + fib(n-2), n > 1



Fibonacci Recursive Call Tree

fib(0) = 0
fib(1) = 1
fib(n) = fib(n-1) + fib(n-2), n > 1



Demonstration!

- Download and run the python script ShowFactorial.py
- TODO:
  - What happens if you make the following function calls in the script?
    - Factorial(9.5)
    - FactorialR(9.5)