Welcome to Data C88C!

Lecture 07: Recursion
Wednesday, July 2nd, 2025
Week 2
Summer 2025
Instructor: Eric Kim (ekim555@berkeley.edu)

Announcements

- Due dates
  - Lab01, Lab02, HW01 due: Tues July 1st, 11:59 PM PST
    - With +1 day auto extension, due tonight at midnight!
  - Lab03, HW03 due: Sun July 6th, 11:59 PM PST
- Reminder: office hours are active, schedule + Zoom links found here: [link]
  - My first office hours is right after this lecture: Wednesdays, 4pm-5pm.

Important University Deadlines

- **Add/Drop deadline: Thursday July 3rd**
  - Advice: if you are feeling extremely behind AND you don't think that you can catch up, consider taking this course another semester
    - Or: take an alternate course like Data 8 / CS 10 first
  - Temperature Check: by end of Week 02, feel comfortable with writing and understanding Python code
    - ex: know how to define and call functions, how to write while loops
    - If you're still struggling with Python syntax by the end of Week 02, you are behind and will likely struggle in this course without significant correction to your study habits.
- **Change grade option deadline: Friday August 1st**
  - eg letter grade -> Pass/No Pass
- For more info, read this Ed post: [link]

Lecture Overview

- Recursion

Recursive Process

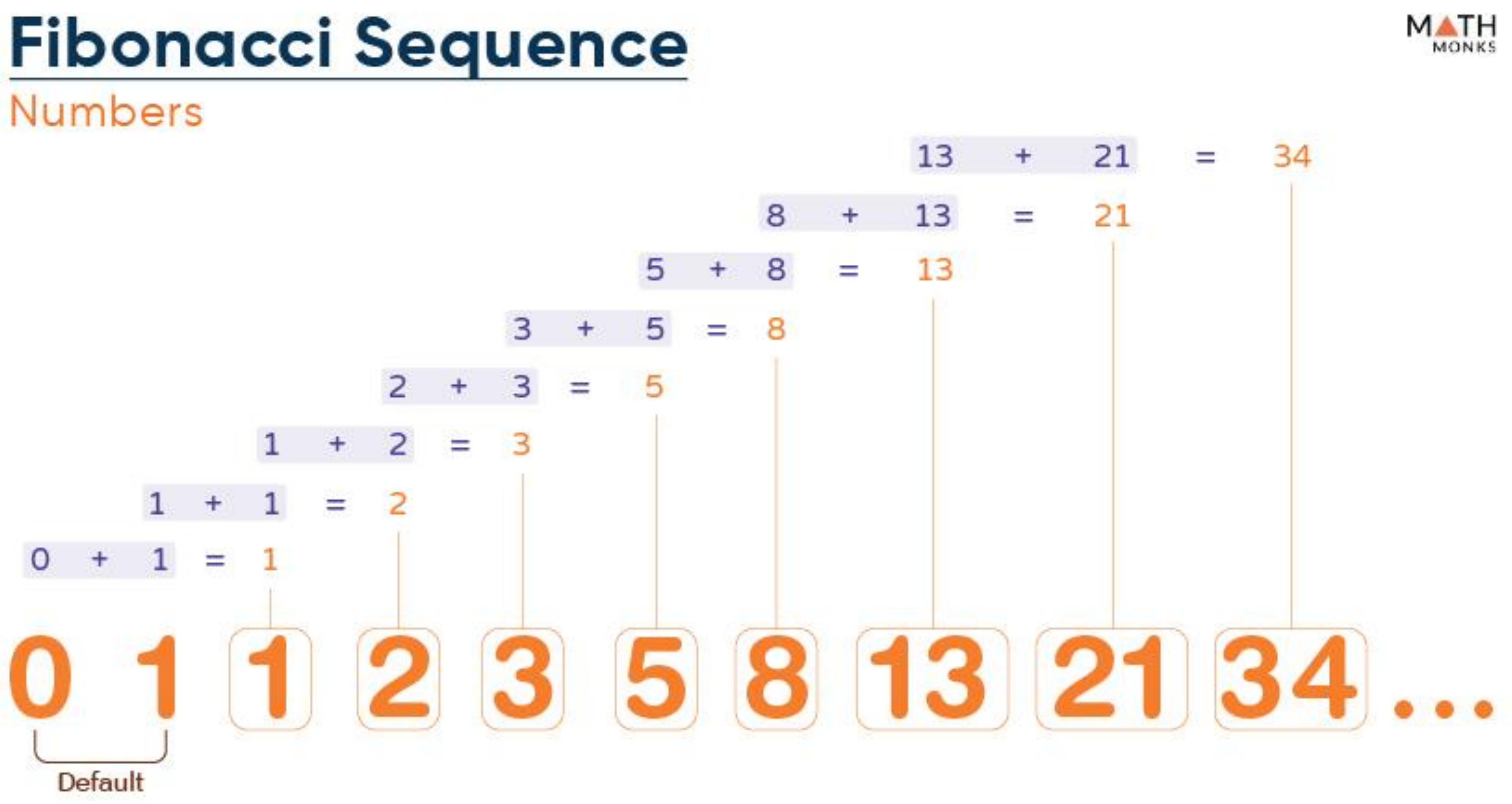
Key idea: recursive functions are useful when your problem can be defined in terms of **smaller self-similar** sub-problems ("recursive structure")

(1) **Divide** – Break the problem down into smaller parts.
(2) **Invoke** – Make the recursive call.
(3) **Combine** – Use the result of the recursive call in your result.
(4) **Base cases** – identify the "smallest" subproblem(s)

Example: fibonacci numbers.

"To compute the n-th fibonacci number, sum the (n-1)-th and (n-2)-th fibonacci numbers."

(In math) fib(n) = fib(n - 1) + fib(n - 2), where fib(0) = 1, fib(1) = 1.



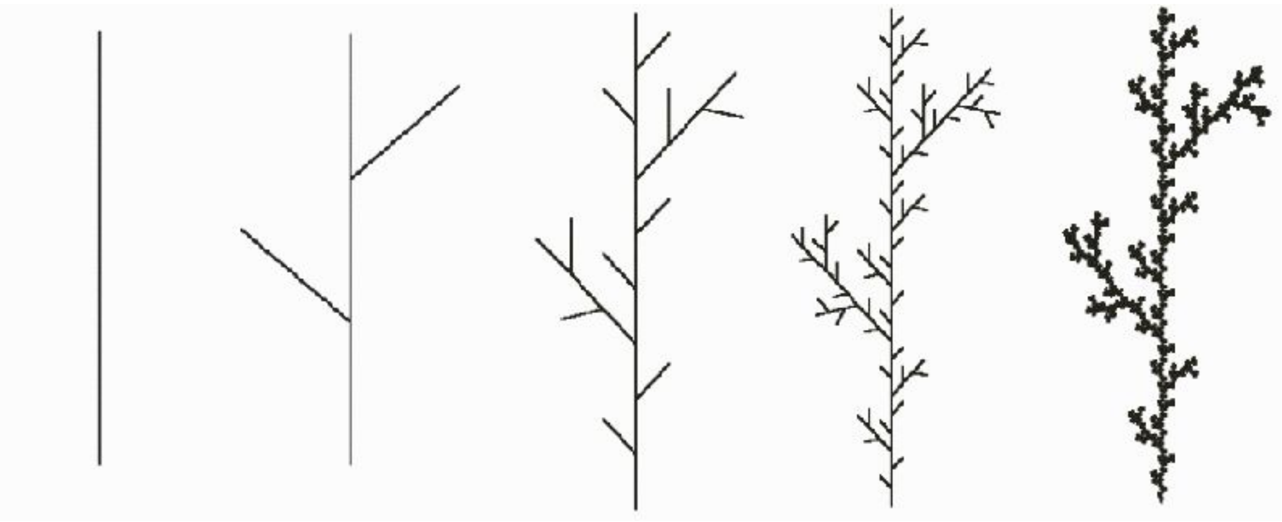
Source: https://mathmonks.com/fibonacci-sequence

Examples of recursive processes

A visual form of recursion known as the Droste effect. The woman in this image holds an object that contains a smaller image of her holding an identical object, which in turn contains a smaller image of herself holding an identical object, and so forth. 1904 Droste cocoa tin, designed by Jan Misset

Recursively defined art [link]

Generating artificial plants/trees via **recursively-defined rules**



(Step 1) Start with a line segment.
(Step 2) Replace the line segment with 5 line segments as pictured, each 1/3 the length of the original.
(Step N) Replace each segment in step *n*-1 with a reduced copy of the step *n*-1 figure.

Source: http://guyhaas.com/bfoit/itp/RecursionInNature/RecursionInNature.html

Figure 6 shows the compounding of some of the inflorescences. These pictures were all done with simple recursion.

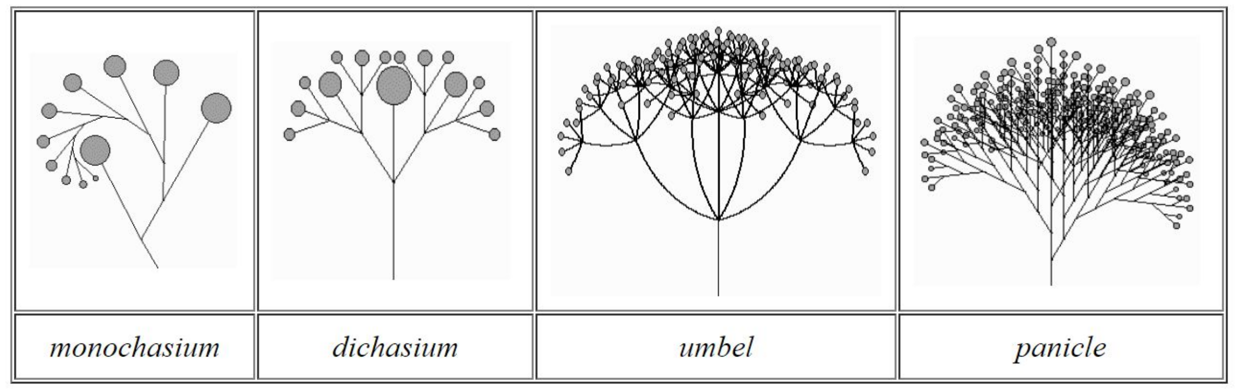


Figure 6: *Compound inflorescences*

Figure 7 shows some imaginary inflorescences obtained by using random numbers to vary segment lengths and angles and taking artistic liberties with the above.

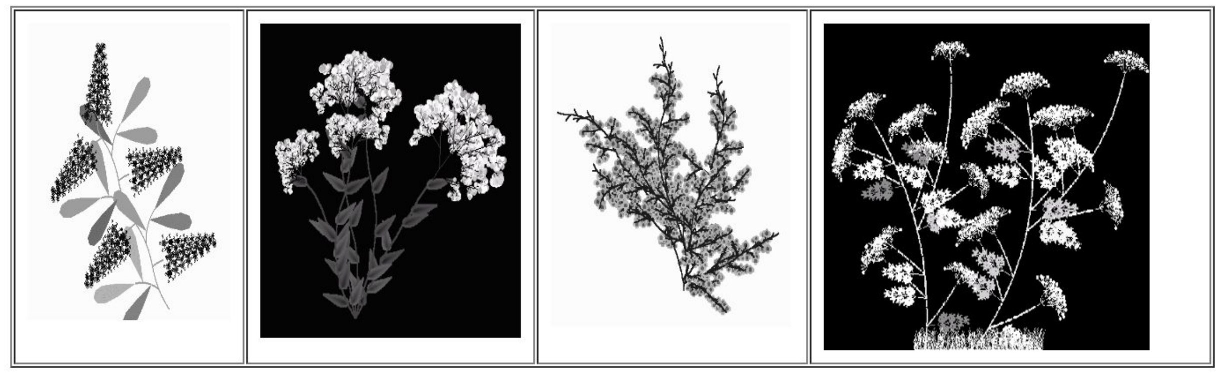
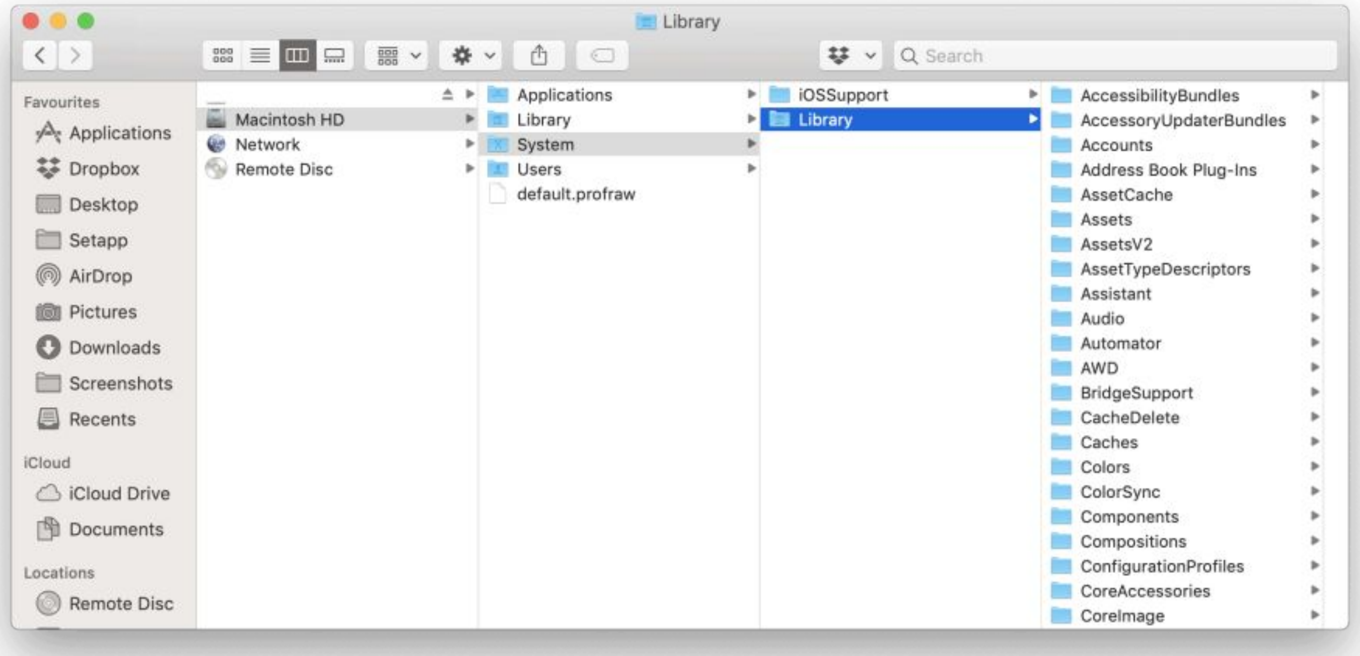


Figure 7: *Imaginary inflorescences*

Recursive file search:
- Scan the current directory
  - If the current entry is a **file**: see if its filename matches our query
  - If the current entry is a **directory**: recursively search in this directory for the query



Source: https://mac-optimization.bestreviews.net/how-to-restore-system-files-on-macos/

Recursive Process

(1) **Divide** – Break the problem down into smaller parts.
(2) **Invoke** – Make the actual recursive call.
(3) **Combine** – Use the result of the recursive call in your result.
(4) **Base cases** – what is the answer to your "smallest" subproblem(s)?

Example: computing factorial.

"To compute 5!, first compute 4!, then multiply by 5."

```
5! = 5 * 4!
= 5 * 4 * 3!
= 5 * 4 * 3 * 2!
= 5 * 4 * 3 * 2 * 1!
= 5 * 4 * 3 * 2 * 1
```

Divide + Invoke: fact(5) needs to call fact(4).

Combine: multiply fact(4) by 5.

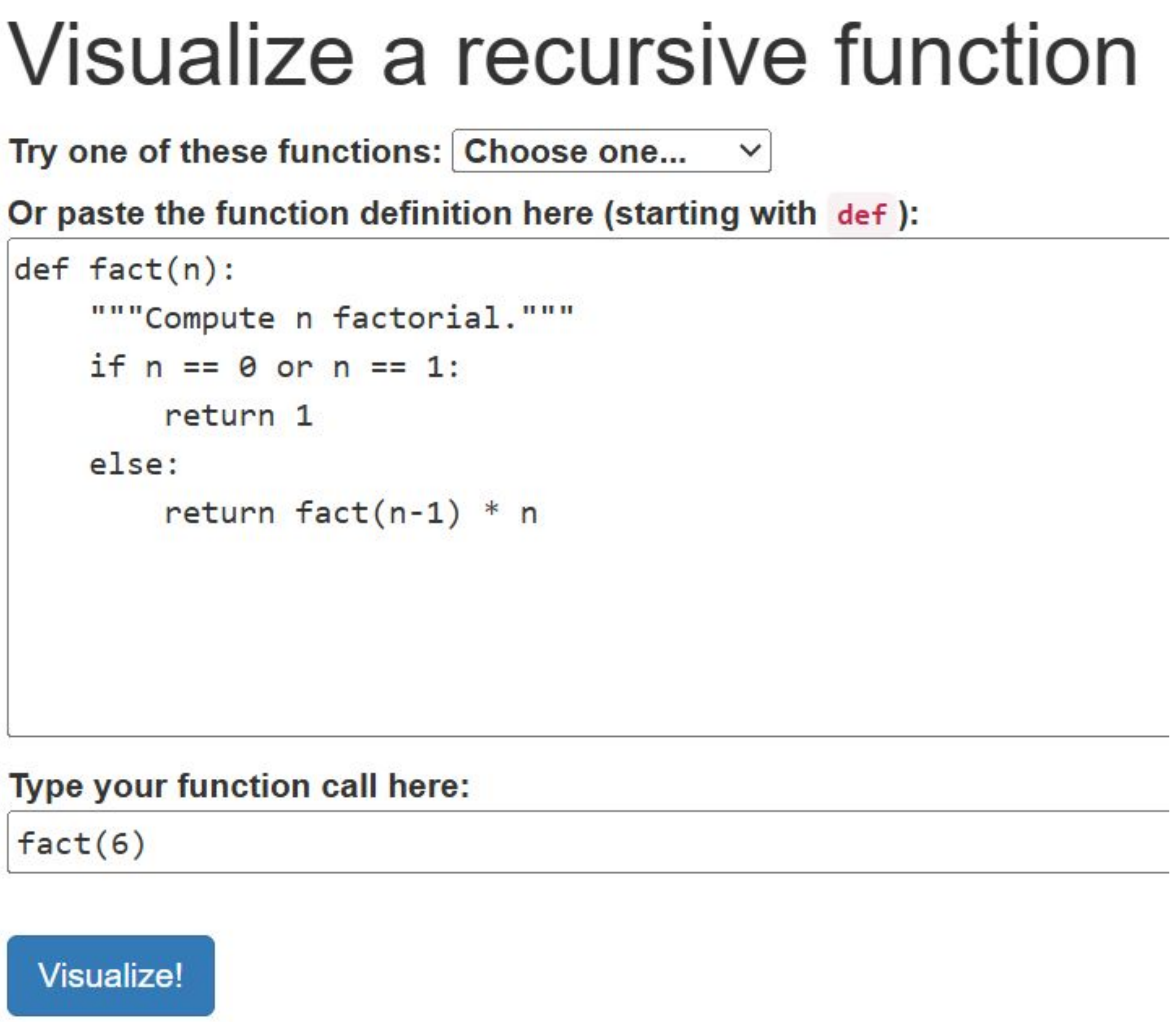
Base cases: fact(1) = 1, fact(0) = 1

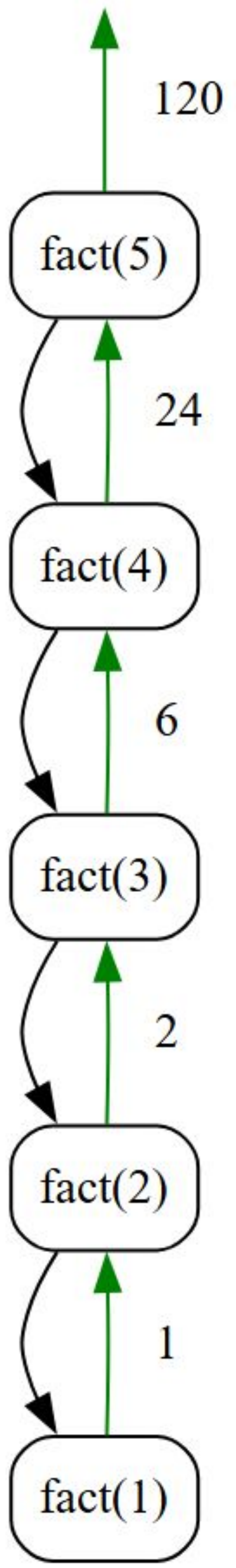
```python
def fact(n):
    if n == 0 or n == 1:
        return 1
    return fact(n - 1) * n
```

(Demo PythonTutor: [link])

(optional) Recursion Visualizer: factorial

- Handy tool for visualizing recursive function call graphs: https://www.recursionvisualizer.com/
- Ex: factorial(6) [link]





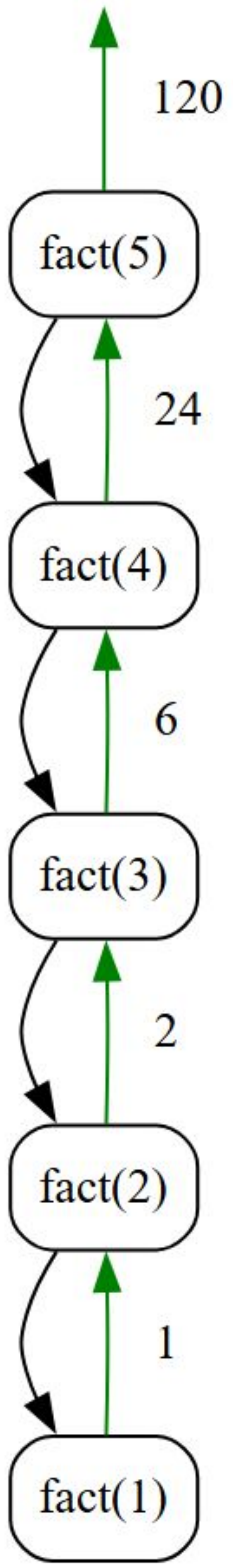
Discussion Question: Factorial Two Ways

```
def fact(n):
    """Compute n factorial.

    >>> fact(5)
    120
    >>> fact(0)
    1
    """
    if n == 0 or n == 1:
        return 1
    else:
        return fact(n-1) * n
```

This version computes fact(5) by these steps:

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



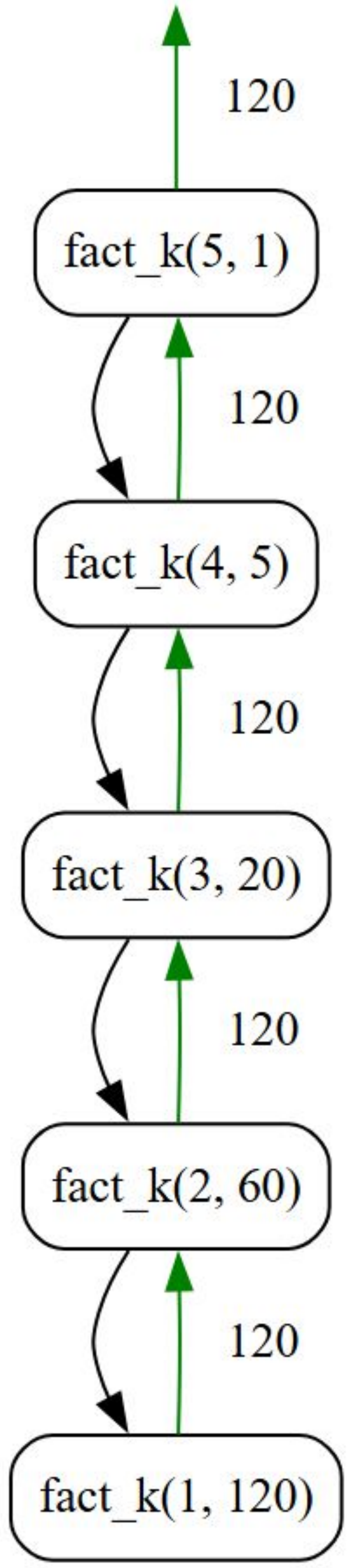
Question: Rewrite fact(n) so that the result of fact(5) is instead computed using the following steps:

```
  5 (1 * 5)
 20 (1 * 5 * 4)
 60 (1 * 5 * 4 * 3)
120 (1 * 5 * 4 * 3 * 2)
```

Trick: store intermediate result in `acc` argument

```
def fact_k(n, acc):
    """Compute n factorial times k.

    >>> fact_k(5, 1)
    120
    >>> fact_k(5, 10)
    1200
    >>> fact_k(0, 10)
    10
    """
    if n == 0 or n == 1:
        return acc
    return fact_k(n - 1, acc * n)
```



RecursionVisualizer: [link]

Recursive Process: forgetting the base case?

(1) **Divide** – Break the problem down into smaller parts.
(2) **Invoke** – Make the actual recursive call.
(3) **Combine** – Use the result of the recursive call in your result.
(4) **Base cases – what is the answer to your "smallest" subproblem(s)?**

Question: what if we forget the base case?

Example: computing factorial.

"To compute 5!, first compute 4!, then multiply by 5."

```
5! = 5 * 4!
= 5 * 4 * 3!
= 5 * 4 * 3 * 2!
= 5 * 4 * 3 * 2 * 1!
= 5 * 4 * 3 * 2 * 1
```

Divide + Invoke: fact(5) needs to call fact(4).

Combine: multiply fact(4) by 5.

Base cases: fact(1) = 1, fact(0) = 1

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

def fact_new(n):
    return fact_new(n - 1) * n
```

Answer: infinite recursion!
(Aside) In reality: you'll get a "`RecursionError: maximum recursion depth exceeded`", to learn more, see: [link]

Self-Reference

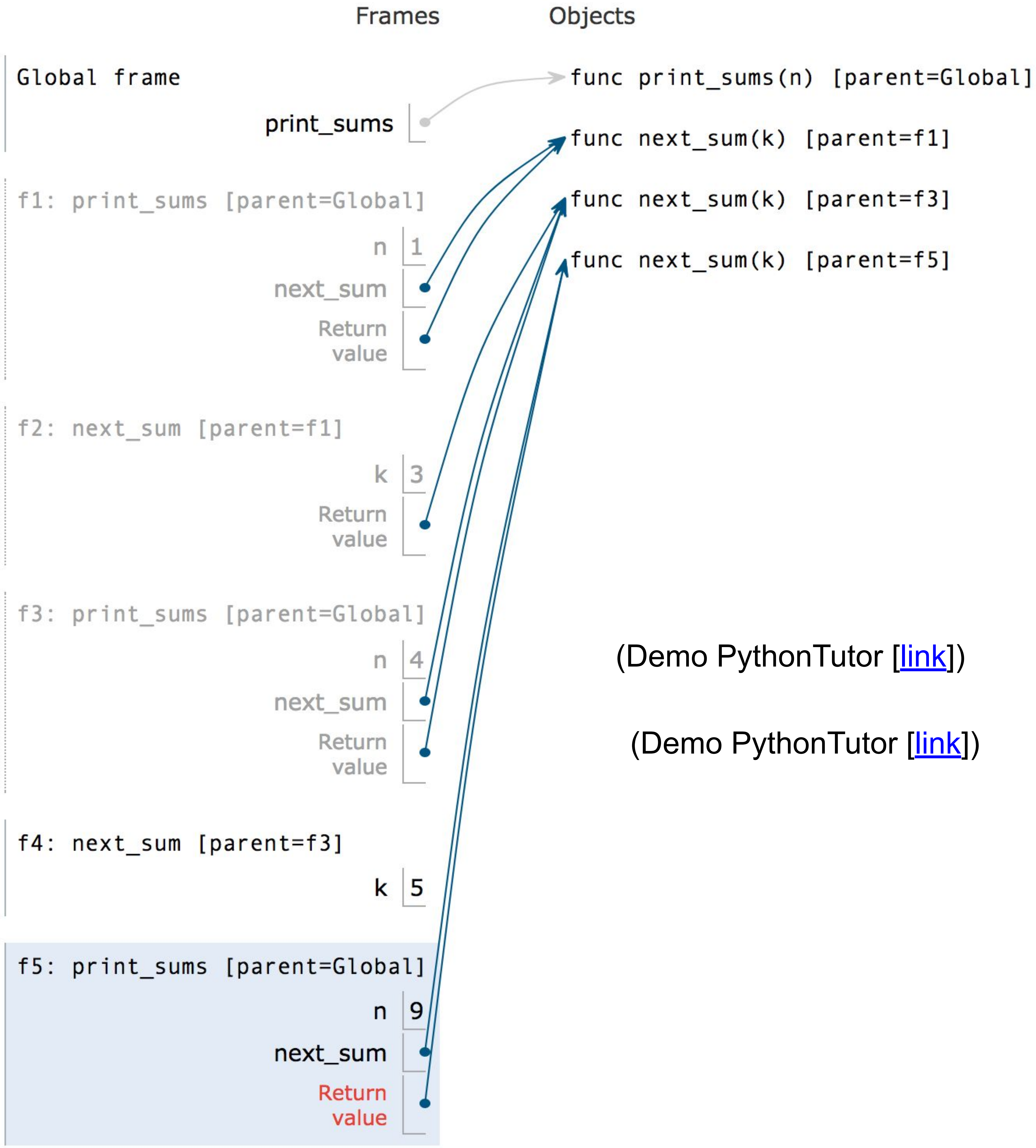
Returning a Function Using Its Own Name

```
def print_sums(n):
    print(n)
    def next_sum(k):
        return print_sums(n+k)
    return next_sum

print_sums(1)(3)(5)
```

print_sums(1)(3)(5) prints:
1
4 (1 + 3)
9 (1 + 3 + 5)

print_sums(3)(4)(5)(6) prints:
3
7 (3 + 4)
12 (3 + 4 + 5)
18 (3 + 4 + 5 + 6)



(Demo PythonTutor [link])

(Demo PythonTutor [link])

http://pythontutor.com/composingprograms.html#code=def%20print_all%28k%29%3A%0A%20%20%20%20print%28k%29%0A%20%20%20%20return%20print_all%0A%20%20%20%20%0Aprint_all%281%29%283%29%285%29&cumulative=true&curInstr=0&mode=display&origin=composingprograms.js&py=3&rawInputLstJSON=%5B%5D

http://pythontutor.com/composingprograms.html#code=def%20print_sums%28n%29%3A%0A%20%20%20%20print%28n%29%0A%20%20%20%20def%20next_sum%28k%29%3A%0A%20%20%20%20%20%20%20%20return%20print_sums%28n%2Bk%29%0A%20%20%20%20return%20next_sum%0A%0Aprint_sums%281%29%283%29%285%29&cumulative=true&curInstr=0&mode=display&origin=composingprograms.js&py=3&rawInputLstJSON=%5B%5D

Example: Add Up Some Numbers (Fall 2016 Midterm 1 Question 5)

Implement `add_up`, which takes a positive integer k. It returns a function that can be called repeatedly k times, one integer argument at a time, and returns the sum of these arguments after k repeated calls.

```
def add_up(k):
    """Add up k numbers after k repeated calls.

    >>> add_up(4)(10)(20)(30)(40)  # Add up 4 numbers: 10 + 20 + 30 + 40
    100
    """
    assert k > 0
    def f(n):
        if k == 1:
            return n
        else:
            return lambda t: add_up(k - 1)(n + t)
    return f
```

add_up(4)(10) returns a one-arg function & needs to remember 3 & 10

add_up(4) returns a one-arg function & needs to remember the 4

Evaluates to a one-arg function that adds k-2 more numbers to n + t

Observation: `f` is a function whose range (output type) changes: sometimes it returns an integer, sometimes it returns a function. Tricky! IMO, it's good practice to try to have your functions always output the same type.

(Demo PythonTutor: [link])

https://pythontutor.com/cp/composingprograms.html#code=def%20add_up%28k%29%3A%0A%20%20%20%20assert%20k%20%3E%200%0A%20%20%20%20def%20f%28n%29%3A%0A%20%20%20%20%20%20%20%20if%20k%20%3D%3D%201%3A%0A%20%20%20%20%20%20%20%20%20%20%20%20return%20n%0A%20%20%20%20%20%20%20%20else%3A%0A%20%20%20%20%20%20%20%20%20%20%20%20return%20lambda%20t%3A%20add_up%28k-1%29%28n%2Bt%29%0A%20%20%20%20return%20f%0A%0Aresult%20%3D%20add_up%284%29%2810%29%2820%29%2830%29%2840%29&cumulative=true&curInstr=0&mode=display&origin=composingprograms.js&py=3&rawInputLstJSON=%5B%5D

Modified add_up()

```python
def add_up_v2(k):
    """Add up k numbers after k repeated calls.

    >>> add_up_v2(4)(10)(20)(30)(40)
    100
    """
    assert k > 0
    def f(n):
        if k == 1:
            return n
        else:
            return lambda t: add_up_v2(k - 1)(t) + n
    return f
```

Question: does this modified implementation work? What Would Python Do?

```
>>> add_up_v2(4)(10)(20)(30)(40)
```

Answer: no it doesn't!

```
Traceback (most recent call last):
  File
"C:\Users\Eric\teaching\data_c88c\lectures\su25\c88c\07.p
y", line 60, in <module>
    print("add_up_v2:", add_up_v2(4)(10)(20)(30)(40))
  File
"C:\Users\Eric\teaching\data_c88c\lectures\su25\c88c\07.p
y", line 57, in <lambda>
    return lambda t: add_up_v2(k - 1)(t) + n
TypeError: unsupported operand type(s) for +: 'function'
and 'int'
```

Question: is there a case where this does "work"?

Answer: yes, when k=1:

```
>>> add_up_v2(1)(42)
42
```

Converting Iteration to Recursion

Discussion Question: Play Twenty-One

Rewrite play as a recursive function without a while statement.

- Do you need to define a new inner function? Why or why not? If so, what are its arguments?
- What is the base case and what is returned for the base case?

```
def play(strategy0, strategy1, goal=21):
    """Play twenty-one and return the winner.

    >>> play(two_strat, two_strat)
    1
    """
    n = 0
    who = 0  # Player 0 goes first
    while n < goal:
        if who == 0:
            n = n + strategy0(n)
            who = 1
        elif who == 1:
            n = n + strategy1(n)
            who = 0
    return who
```

```
def play(strategy0, strategy1, goal=21):
    """Play twenty-one and return the winner.

    >>> play(two_strat, two_strat)
    1
    """
    def f(n, who):
        if n >= goal:
            return who
        if who == 0:
            n = n + strategy0(n)
            who = 1
        elif who == 1:
            n = n + strategy1(n)
            who = 0
        return f(n, who)
    return f(0, 0)
```

Observation: we handled local variables (eg `who`) as additional arguments to the recursive function. One way to pass additional info ("state") to recursion