# CS 301: Recursion

## The Art of Self Reference

Tyler Caraza-Harter

# Goal: use self-reference is a meaningful way

***Hofstadter's Law***: *"It always takes longer than you expect, even when you take into account* ***Hofstadter's Law****."*

(From Gödel, Escher, Bach)

good advice for CS 301 assignments!

https://en.wikipedia.org/wiki/Circular_definition

# Goal: use self-reference is a meaningful way

***Hofstadter's Law**: "It always takes longer than you expect, even when you take into account **Hofstadter's Law**."*

(From Gödel, Escher, Bach)

***mountain**: "a landmass that projects conspicuously above its surroundings and is higher than a **hill**"*

***hill**: "a usually rounded natural elevation of land lower than a **mountain**"*

(Example of **unhelpful** self reference from Merriam-Webster dictionary)

https://en.wikipedia.org/wiki/Circular_definition

# Overview: Learning Objectives

Recursive information

- What is a **recursive definition/structure**?
- Arbitrarily vs. infinitely

Recursive code

- What is **recursive code**?
- Why write recursive code?
- Where do computers keep local variables for recursive calls?
- What happens to programs with **infinite recursion**?

**Read Think Python**

- **Ch 5: “Recursion” through “Infinite Recursion”**
- **Ch 6: “More Recursion” through end**

# Overview: Learning Objectives

Recursive information

- What is a **recursive definition/structure**?
- Arbitrarily vs. infinitely

Recursive code

- What is **recursive code**?
- Why write recursive code?
- Where do computers keep local variables for recursive calls?
- What happens to programs with **infinite recursion**?

# What is Recursion?

**Recursive** definitions contain the term in the body

- Dictionaries, mathematical definitions, etc

A number **x** is a positive even number if:

- **x** is 2
  **OR**
- **x** equals another positive even number plus two

# What is Recursion?

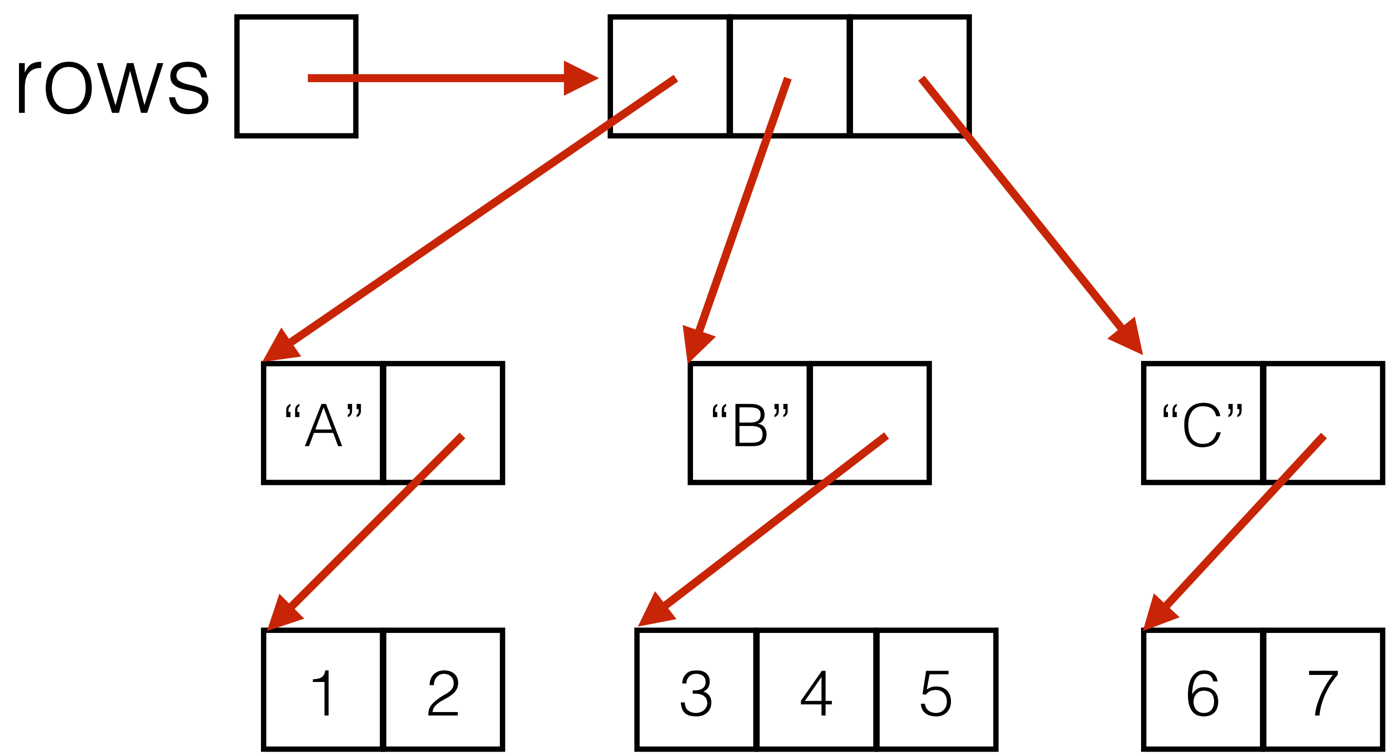
**Recursive** definitions contain the term in the body

- Dictionaries, mathematical definitions, etc

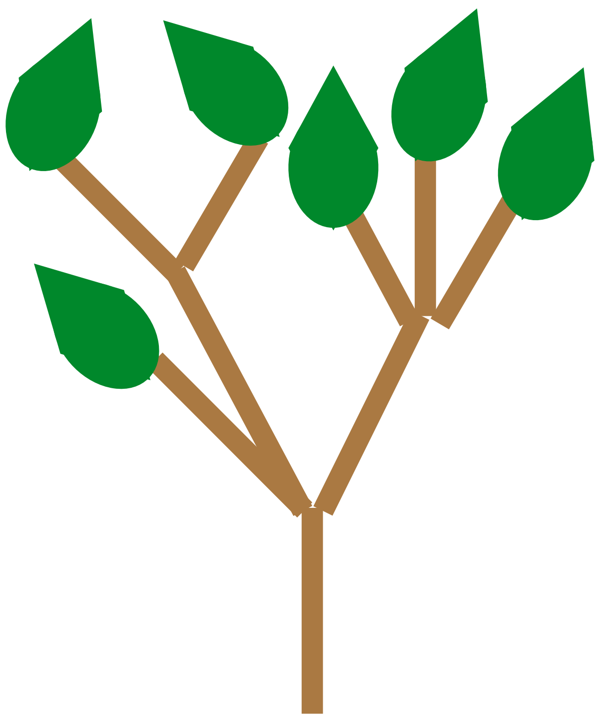
**Recursive** structures may refer to structures of the same type

- data structures or real-world structures

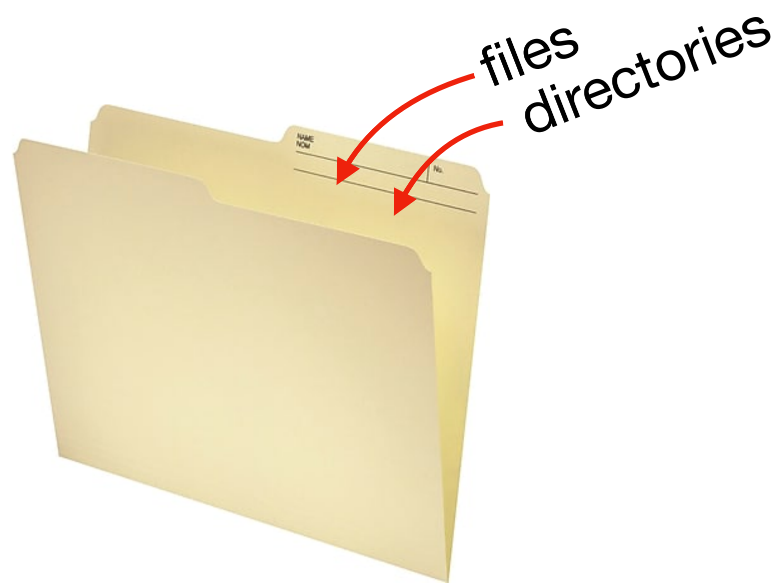
```
rows = [
  ["A",[1,2]],
  ["B",[3,4,5]],
  ["C",[6,7]]
]
```



# Recursive structures are EVERYWHERE!



nature




files

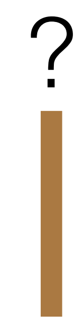
```
{
  “name”: “alice”,
  “grade”: “A”,
  “score”: 96,
  “exams”: {
    “midterm”: {“points”:94,
                “total”:100},
    “final”: {“points”: 98,
              “total”: 100}
  }
}
```

formats

# Example: Trees (Finite Recursion)

**Term:** branch

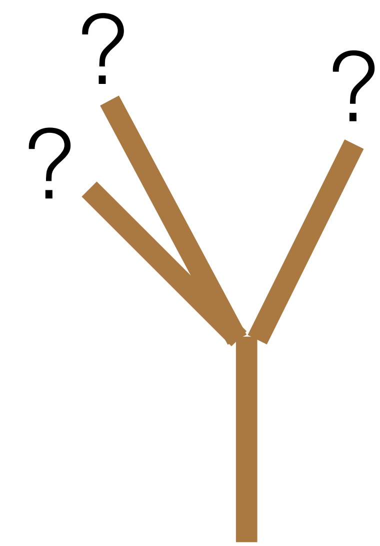
**Def**: wooden stick, with an end splitting into other branches, OR terminating with a leaf



# Example: Trees (Finite Recursion)

**Term:** branch

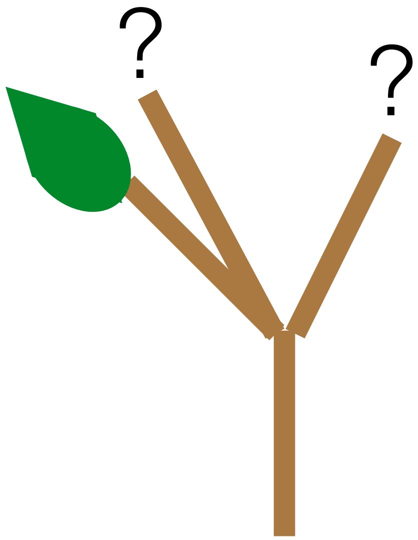
**Def**: wooden stick, with an end splitting into other branches, OR terminating with a leaf



# Example: Trees (Finite Recursion)

**Term:** branch

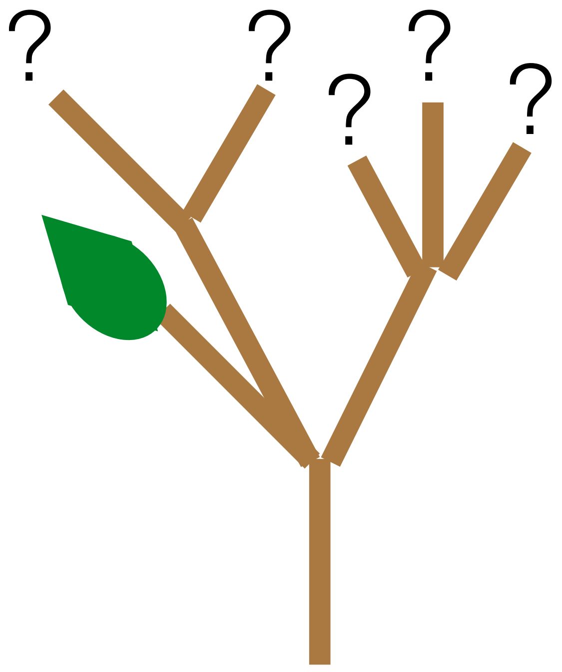
**Def**: wooden stick, with an end splitting into other branches, OR terminating with a leaf



# Example: Trees (Finite Recursion)

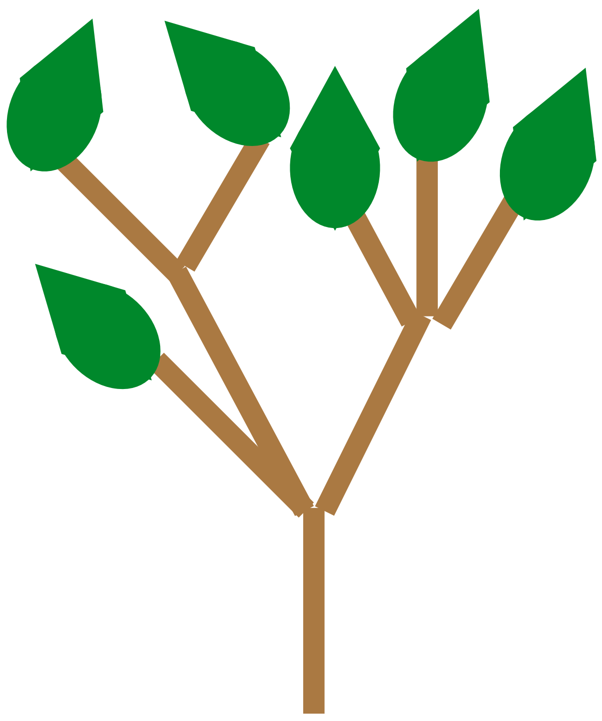
**Term:** branch

**Def**: wooden stick, with an end splitting into other branches, OR terminating with a leaf



# Example: Trees (Finite Recursion)

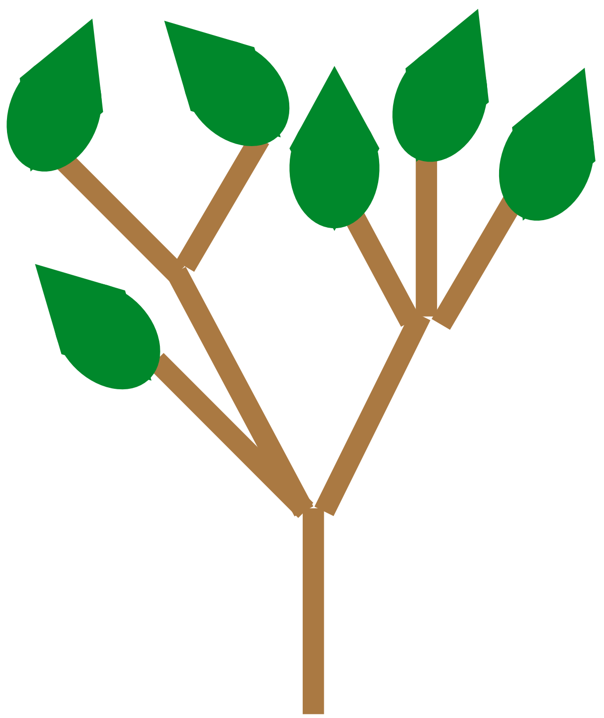
**Term:** branch



**Def**: wooden stick, with an end splitting into other branches, OR terminating with a leaf

# Example: Trees (Finite Recursion)

**Term:** branch



**Def**: wooden stick, with an end splitting into other branches, OR terminating with a leaf

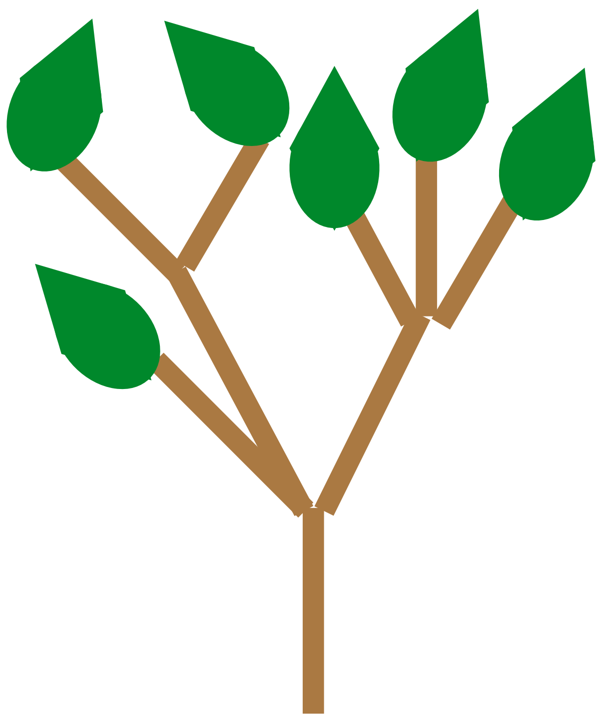
trees are arbitrarily large: **recursive case** allows indefinite growth

arbitrarily != infinitely

# Example: Trees (Finite Recursion)

**Term:** branch

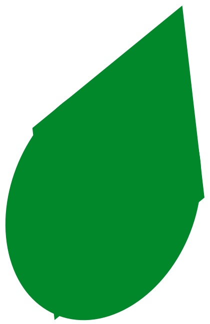
**Def**: wooden stick, with an end
splitting into other branches, OR
terminating with a leaf



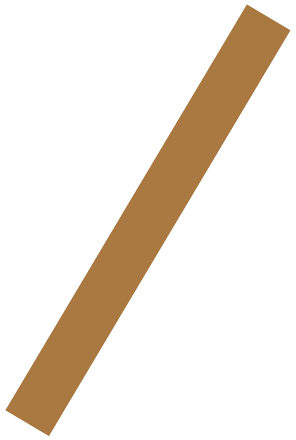
trees are finite:
eventual **base case**
allows completion

trees are arbitrarily large:
**recursive case** allows
indefinite growth

arbitrarily != infinitely



base case (leaf)

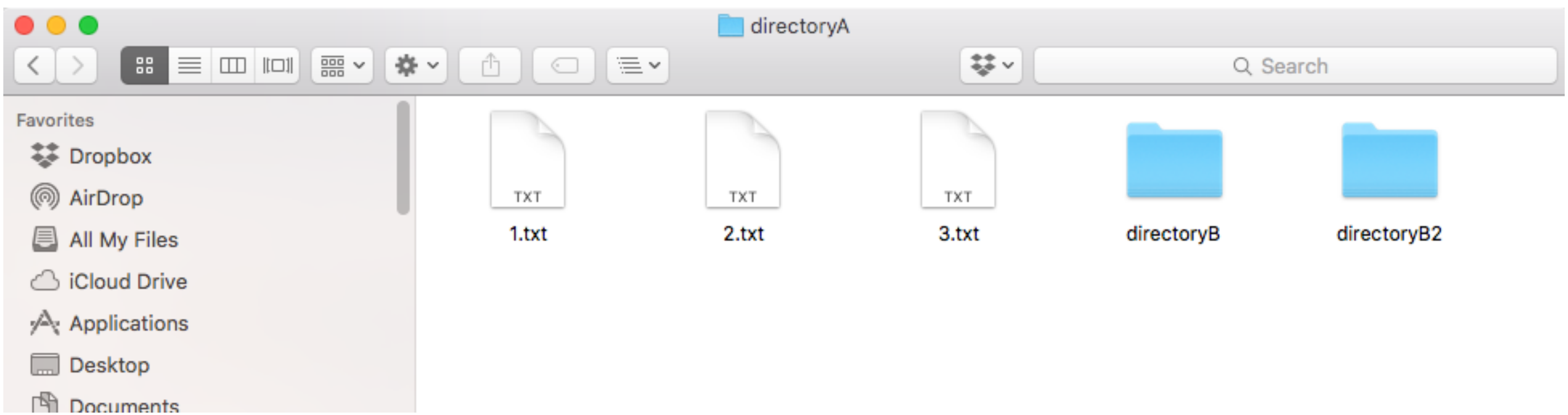


recursive case (branch)

# Example: Directories (aka folders)

**Term:** directory

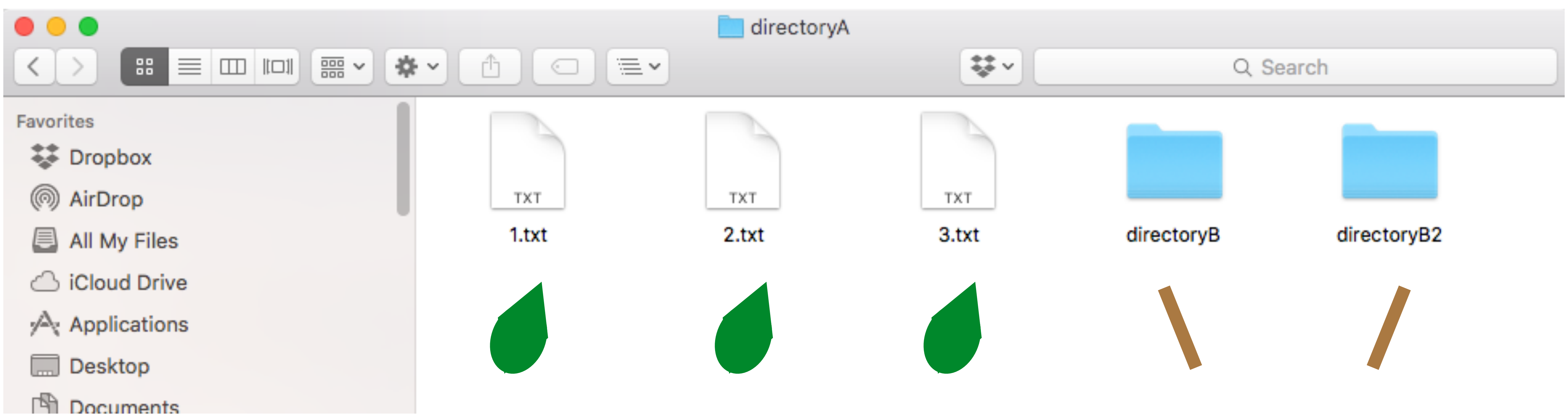
**Def**: a collection of files and directories



# Example: Directories (aka folders)

**Term:** directory

**Def**: a collection of files and directories

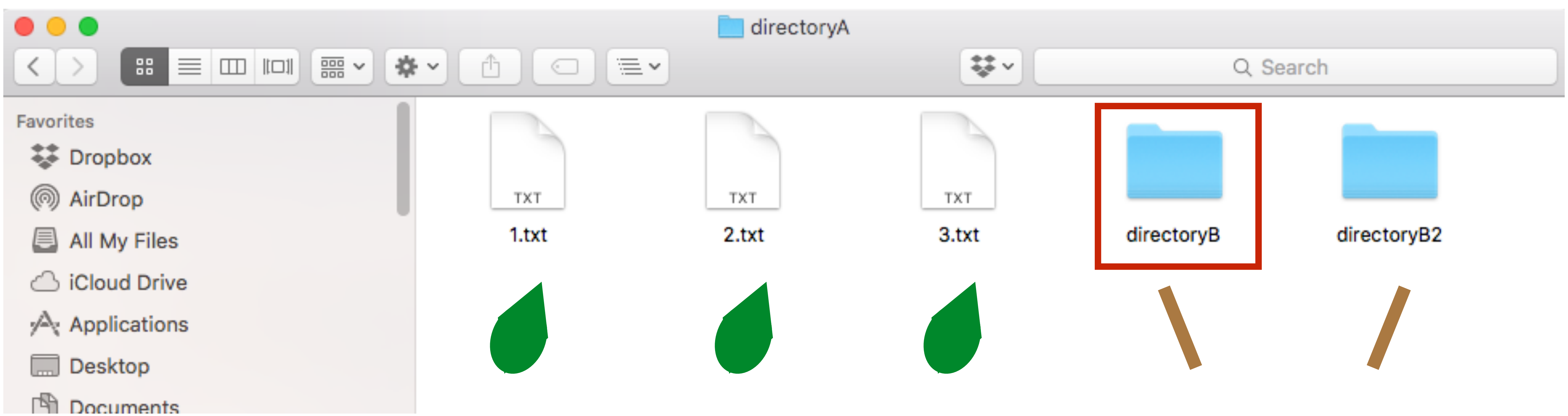



*file system tree*

# Example: Directories (aka folders)

**Term:** directory

**Def**: a collection of files and directories




*file system tree*

# Example: Directories (aka folders)

**Term:** directory

**Def**: a collection of files and directories


*file system tree*

# Example: Directories (aka folders)

**Term:** directory

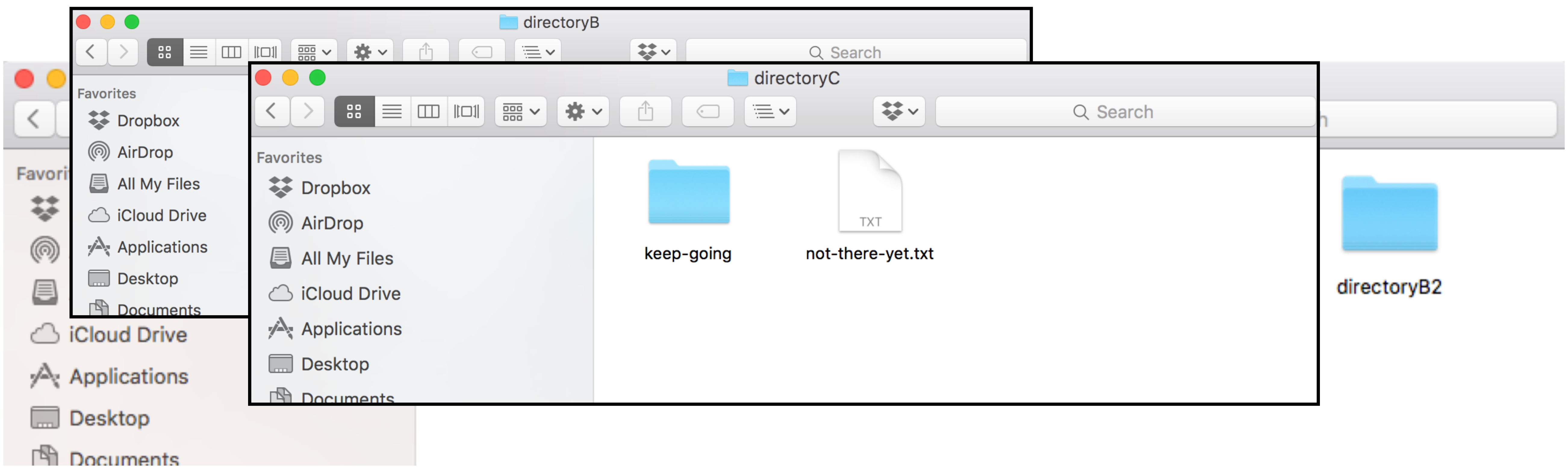
**Def**: a collection of files and directories


*file system tree*

# Example: Directories (aka folders)

**Term:** directory

**Def**: a collection of files and directories

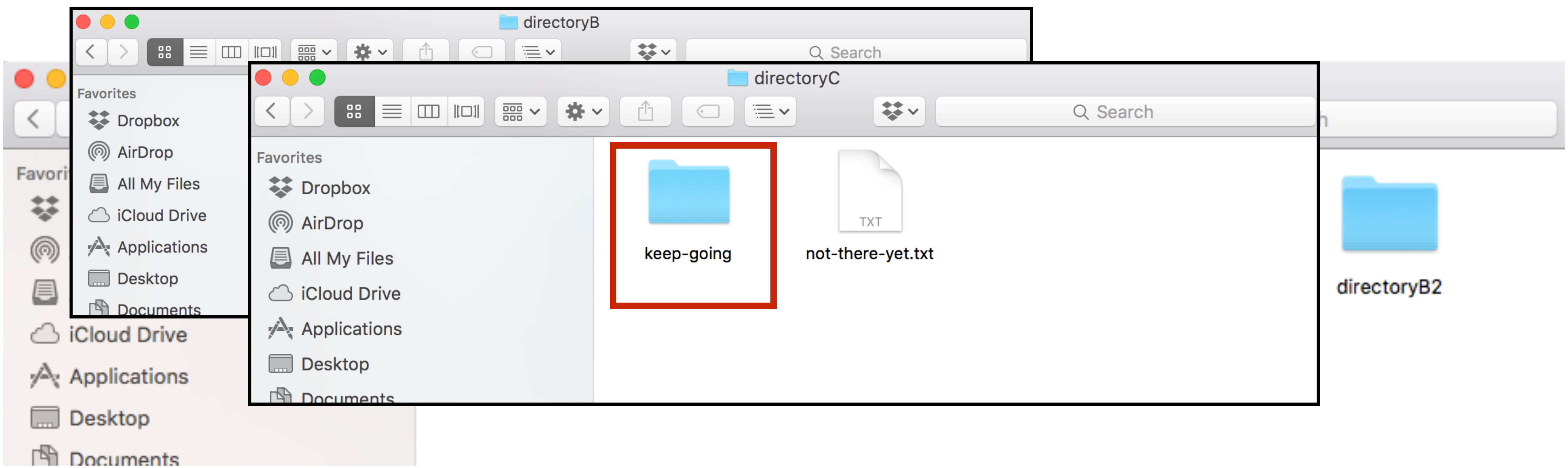



*file system tree*

# Example: Directories (aka folders)

**Term:** directory

**Def**: a collection of files and directories

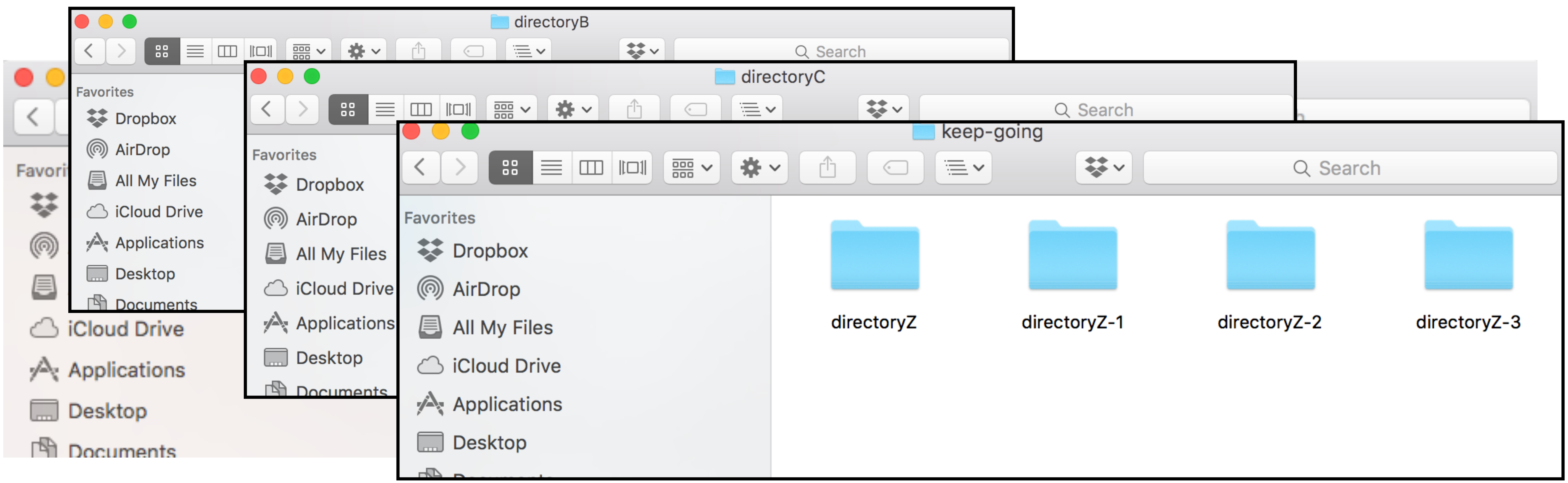



*file system tree*

# Example: Directories (aka folders)

**Term:** directory

**Def**: a collection of files and directories

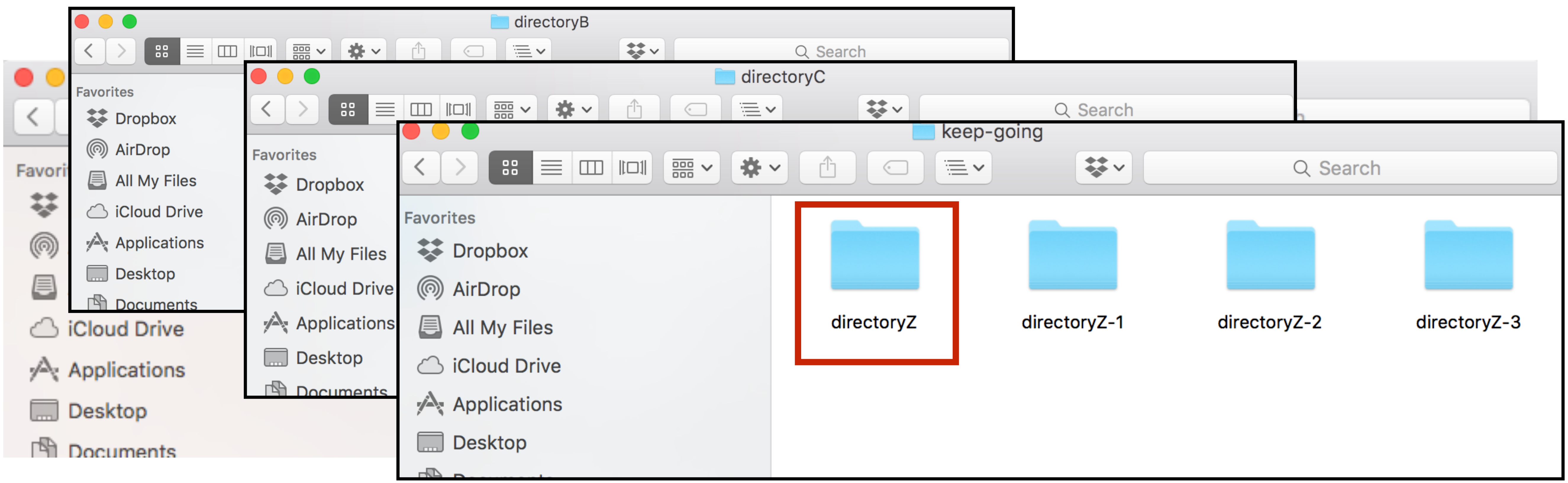



*file system tree*

# Example: Directories (aka folders)

**Term:** directory

**Def**: a collection of files and directories



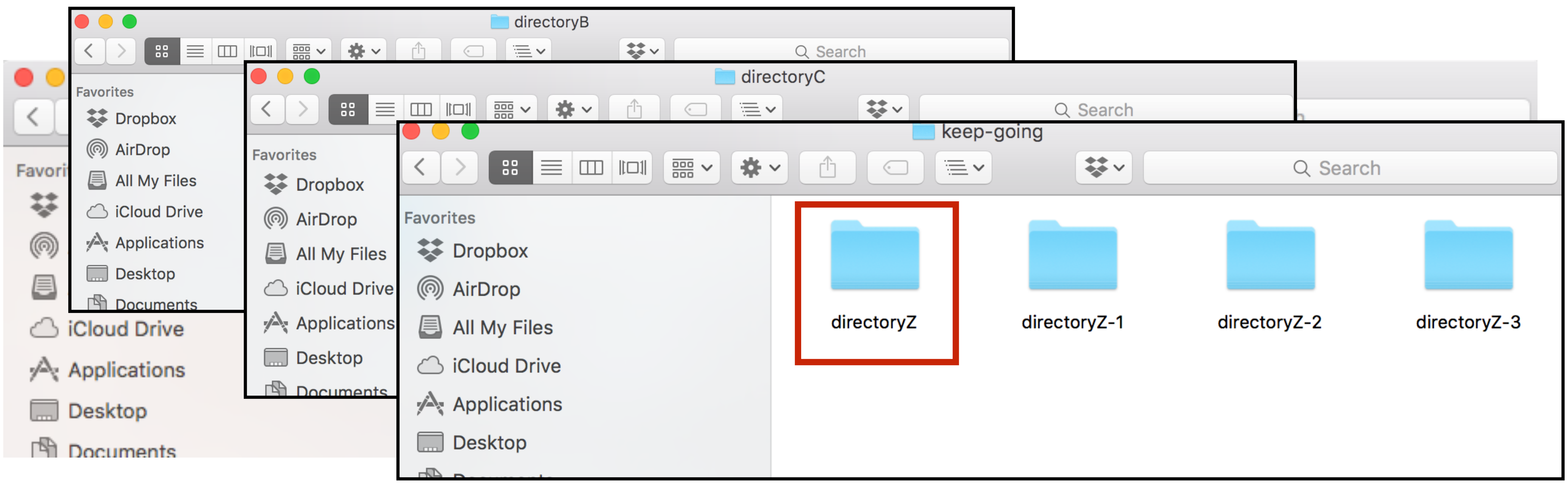

*file system tree*

# Example: Directories (aka folders)

**Term:** directory

*recursive because def contains term*

**Def**: a collection of files and directories




*file system tree*

# Example: JSON Format

**Example JSON Dictionary:**

```
{
 “name”: “alice”,
 “grade”: “A”,
 “score”: 96
}
```

# Example: JSON Format

**Example JSON Dictionary:**

```
{
 “name”: “alice”,
 “grade”: “A”,
 “score”: 96
}
```

**Term:** *json-dict*
**Def:** a set of *json-mapping*'s

# Example: JSON Format

**Example JSON Dictionary:**

```
{
 “name”: “alice”,
 “grade”: “A”,
 “score”: 96
}
```

**Term:** *json-dict*
**Def:** a set of *json-mapping*'s

# Example: JSON Format

**Example JSON Dictionary:**

```
{
 “name”: “alice”,
 “grade”: “A”,
 “score”: 96
}
```

keys values

**Term:** *json-dict*
**Def:** a set of *json-mapping*'s

**Term:** *json-mapping*
**Def:** a *json-string* (KEY) paired with a *json-string* OR *json-number* OR *json-dict* (VALUE)

note: complete JSON is slightly more flexible

# Example: JSON Format

**Example JSON Dictionary:**

```
{
 “name”: “alice”,
 “grade”: “A”,
 “score”: 96
}
```

**Term:** *json-dict*
**Def:** a set of *json-mapping*'s

**Term:** *json-mapping*
**Def:** a *json-string* (KEY) paired with a *json-string* OR *json-number* OR *json-dict* (VALUE)

recursive self reference isn’t always direct!

note: complete JSON is slightly more flexible

# Example: JSON Format

**Example JSON Dictionary:**

```
{
  “name”: “alice”,
  “grade”: “A”,
  “score”: 96,
  “exams”: {
    “midterm”: 94,
    “final”: 98
  }
}
```

**Term:** *json-dict*
**Def:** a set of *json-mapping*'s

**Term:** *json-mapping*
**Def:** a *json-string* (KEY) paired with a *json-string* OR *json-number* OR *json-dict* (VALUE)

note: complete JSON is slightly more flexible

# Example: JSON Format

**Example JSON Dictionary:**

```
{
 “name”: “alice”,
 “grade”: “A”,
 “score”: 96,
 “exams”: {
   “midterm”: {“points”:94,
              “total”:100},
   “final”: {“points”: 98,
            “total”: 100}
 }
}
```

**Term:** *json-dict*
**Def:** a set of *json-mapping*'s

**Term:** *json-mapping*
**Def:** a *json-string* (KEY) paired with a *json-string* OR *json-number* OR *json-dict* (VALUE)

note: complete JSON is slightly more flexible

# Example: JSON Format

**Example JSON Dictionary:**

```
{
  “name”: “alice”,
  “grade”: “A”,
  “score”: 96,
  “exams”: {
    “midterm”: {“points”:94,
                 “total”:100},
    “final”: {“points”: 98,
               “total”: 100}
  }
}
```

**Term:** *json-dict*
**Def:** a set of *json-mapping*'s

**Term:** *json-mapping*
**Def:** a *json-string* (KEY) paired with a *json-string* OR *json-number* OR *json-dict* (VALUE)

note: complete JSON is slightly more flexible

# Overview: Learning Objectives

Recursive information

- What is a **recursive definition/structure**?
- Arbitrarily vs. infinitely

Recursive code

- What is **recursive code**?
- Why write recursive code?
- Where do computers keep local variables for recursive calls?
- What happens to programs with **infinite recursion**?

# Recursive Code

What is it?

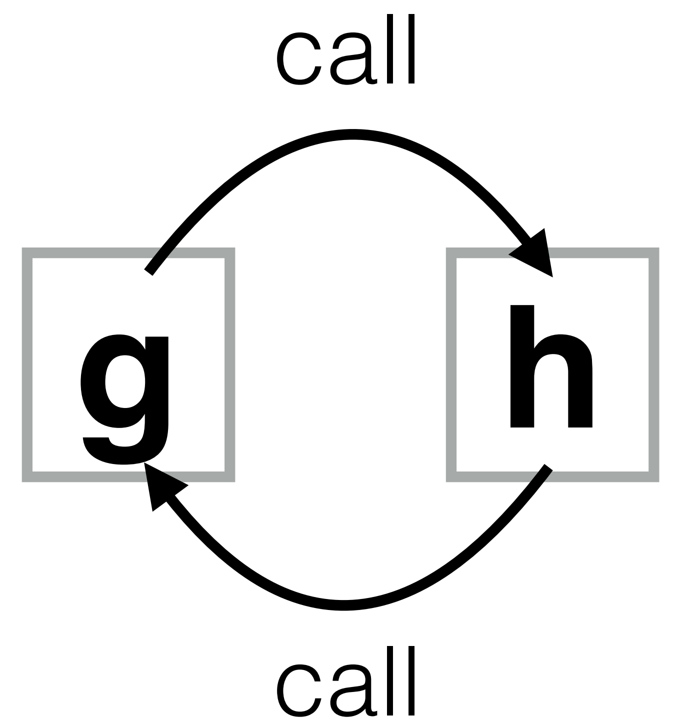
- A function that calls itself (possible indirectly)

# Recursive Code

What is it?

- A function that calls itself (possible indirectly)

```
def f():
    # other code
    f()
    # other code
```



# Recursive Code

What is it?

- A function that calls itself (possible indirectly)

```python
def f():
    # other code
    f()
    # other code
```

```python
def g():
    # other code
    h()
    # other code

def h():
    # other code
    g()
    # other code
```

# Recursive Code

What is it?

- A function that calls itself (possible indirectly)

Motivation: don’t know how big data is before execution

- Need either **iteration** or **recursion**
- In theory, these techniques are equally powerful

# Recursive Code

What is it?

- A function that calls itself (possible indirectly)

Motivation: don’t know how big data is before execution

- Need either **iteration** or **recursion**
- In theory, these techniques are equally powerful

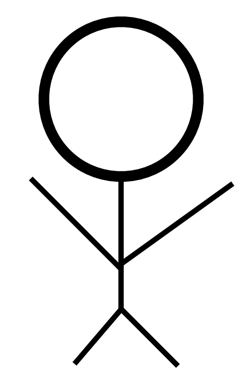
Why recurse? (instead of always iterating)

- in practice, often easier
- recursive code corresponds to recursive data
- reduce a big problem into a smaller problem

# Recursive Students

eager CS 301 students
in the front row

wise teacher

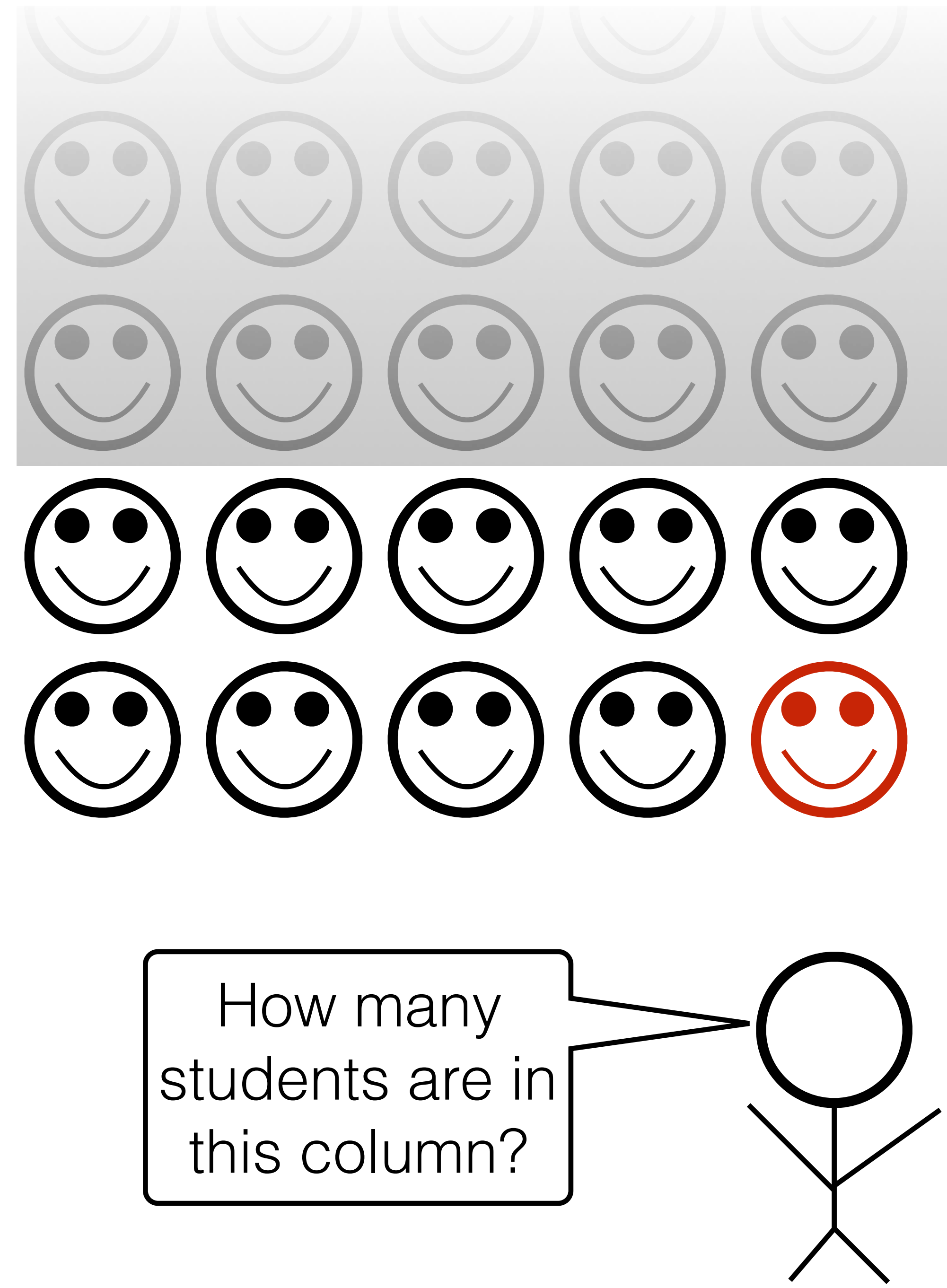


Example from https://courses.cs.washington.edu/courses/cse143/17au/

# Recursive Students

Imagine:

A teacher wants to know how many students are in a column.
**How can front student answer?**

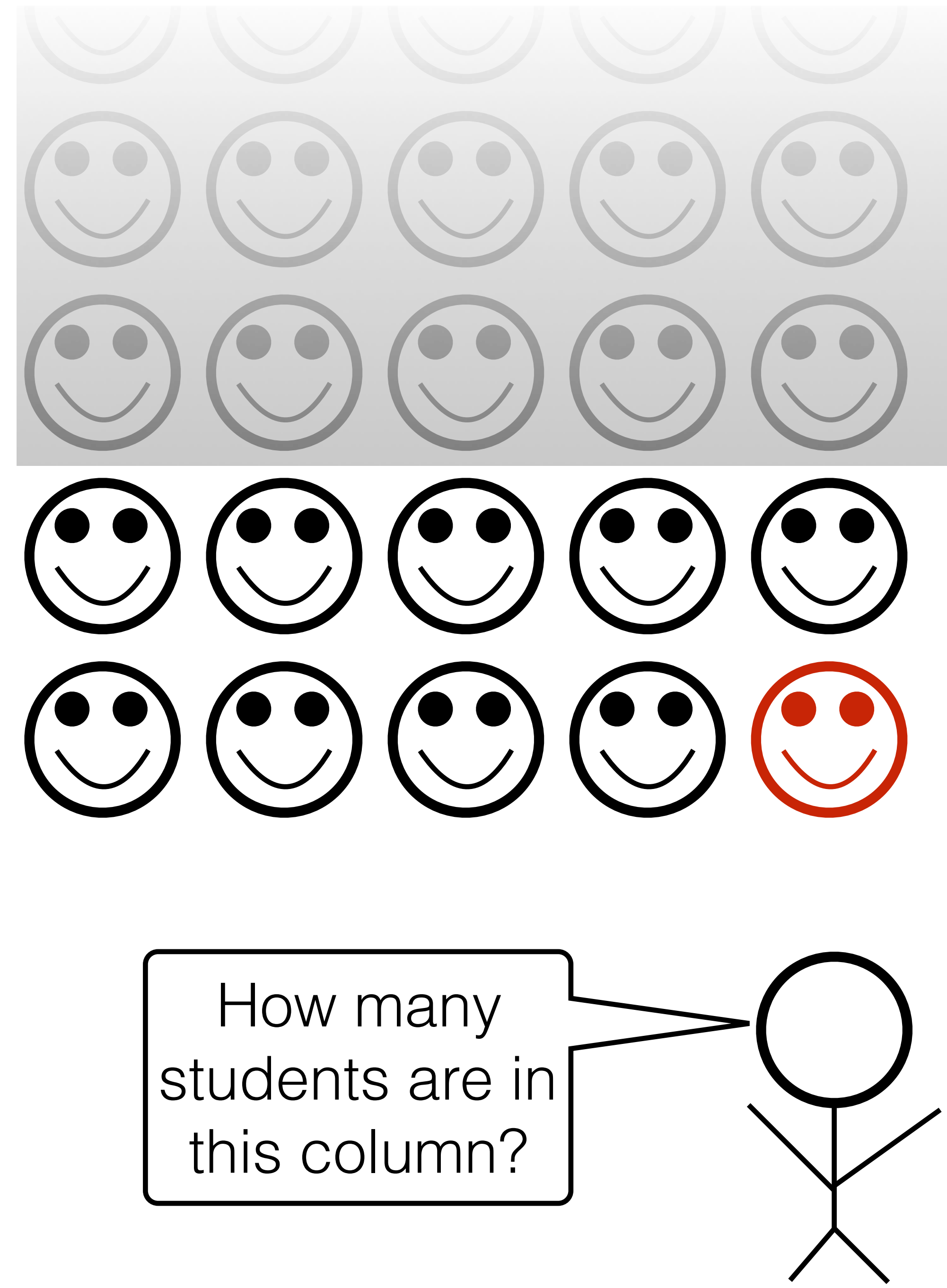



Example from https://courses.cs.washington.edu/courses/cse143/17au/

# Recursive Students

Imagine:

A teacher wants to know how many students are in a column. **How can front student answer?**



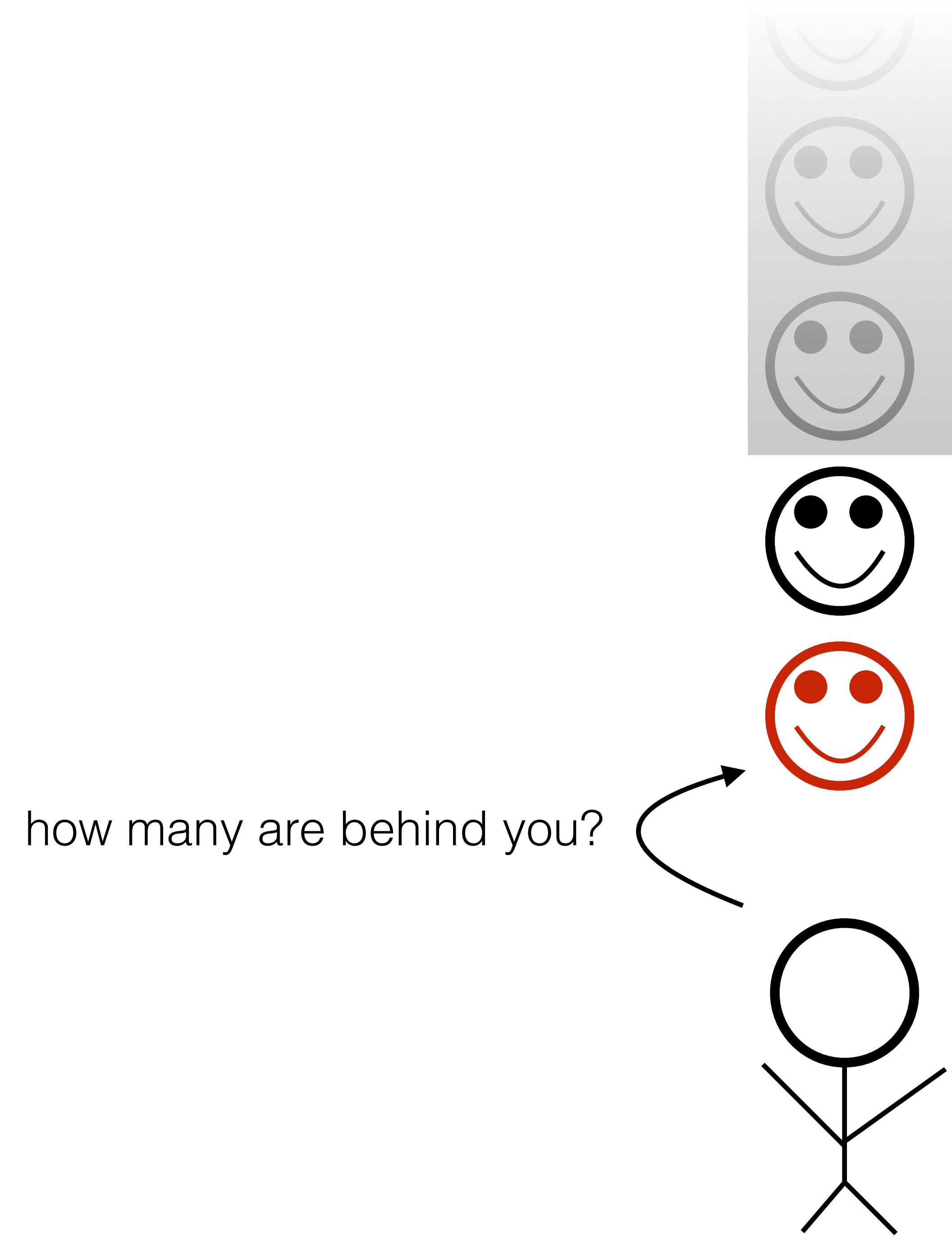

Constraints:

- It is dark, you **can't** see the back
- You **can't** get up to count
- You **may** talk to adjacent students
- Mic is broken (students in back can't hear from front)

Example from https://courses.cs.washington.edu/courses/cse143/17au/

# Recursive Students

Strategy: reframe question as *"how many students are behind you?"*

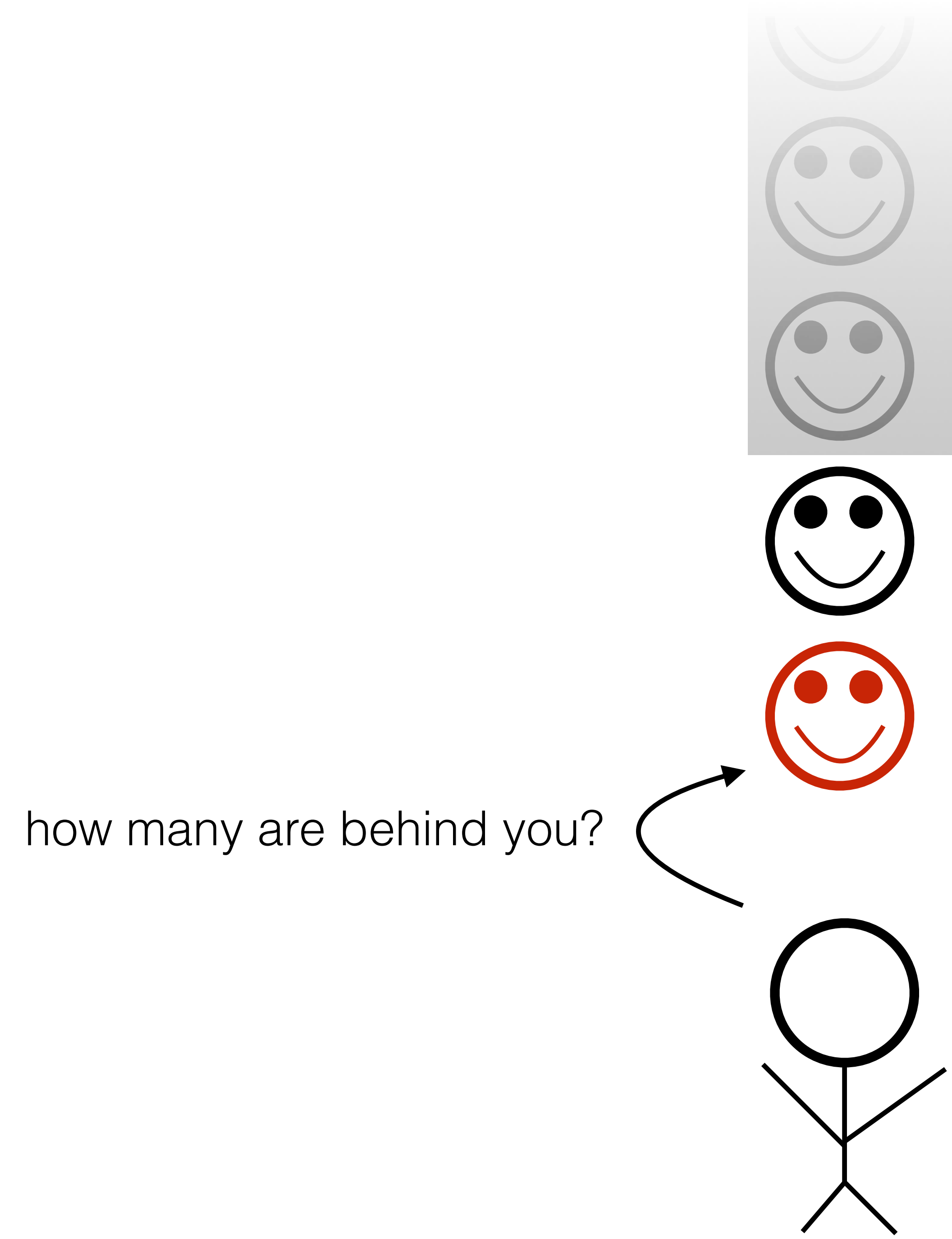



Example from https://courses.cs.washington.edu/courses/cse143/17au/

# Recursive Students

Strategy: reframe question as *"how many students are behind you?"*

*Reframing is the hardest part*

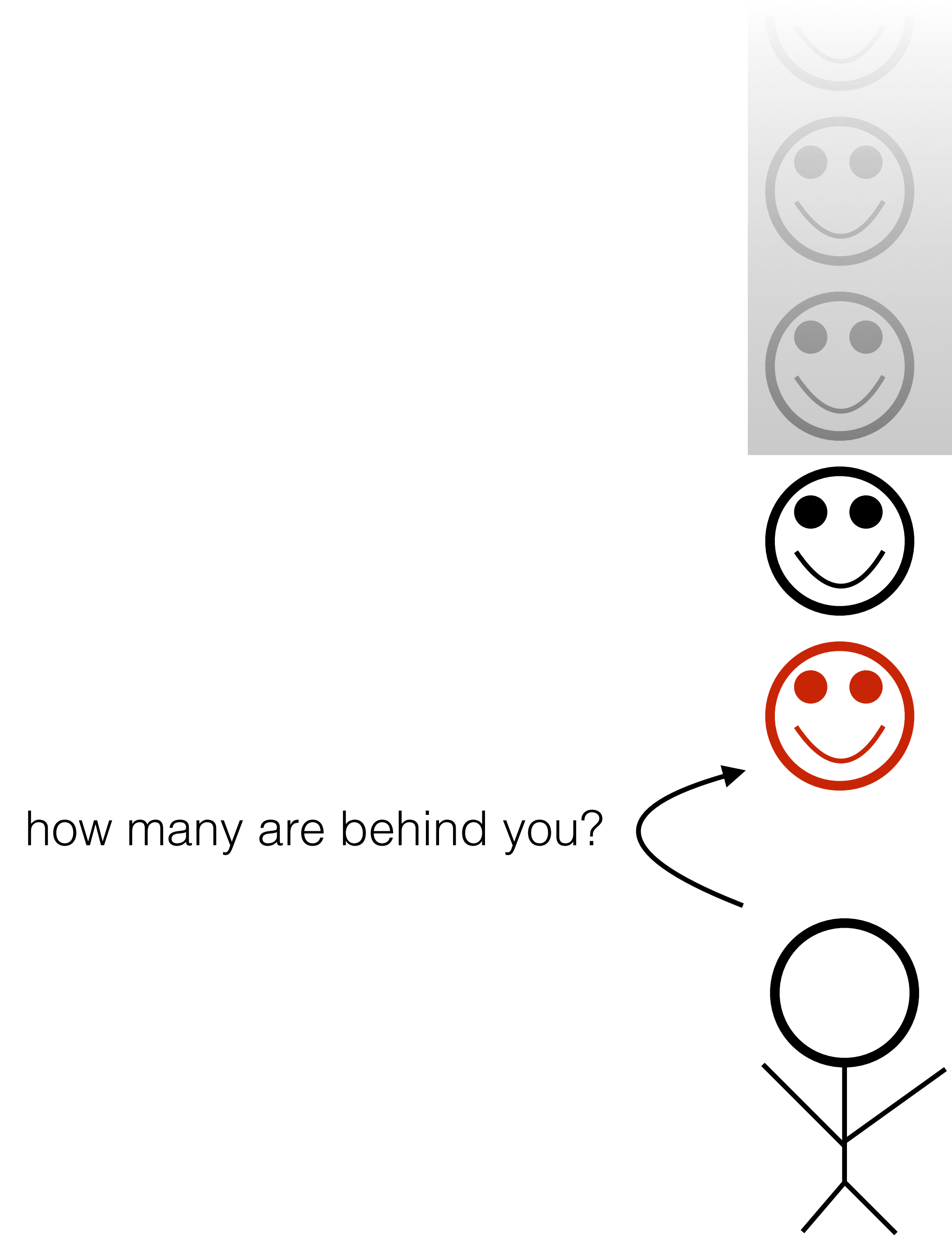



Example from https://courses.cs.washington.edu/courses/cse143/17au/

# Recursive Students

Strategy: reframe question as *"how many students are behind you?"*

Process:
**if** nobody is behind you: **say** 0
**else**: ask them, **say** their answer+1

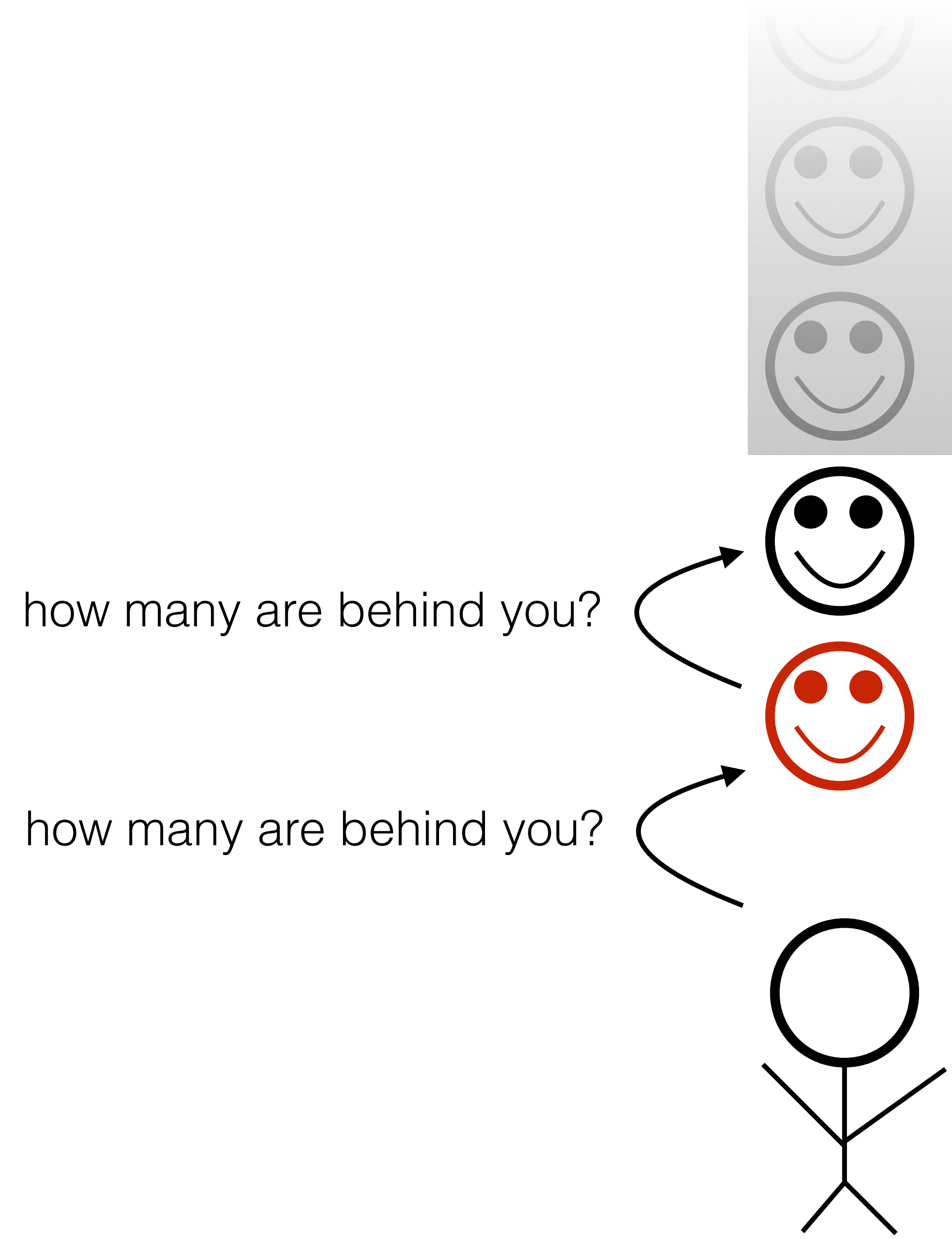



Example from https://courses.cs.washington.edu/courses/cse143/17au/

# Recursive Students

Strategy: reframe question as *"how many students are behind you?"*

Process:
**if** nobody is behind you: **say** 0
**else**: ask them, **say** their answer+1

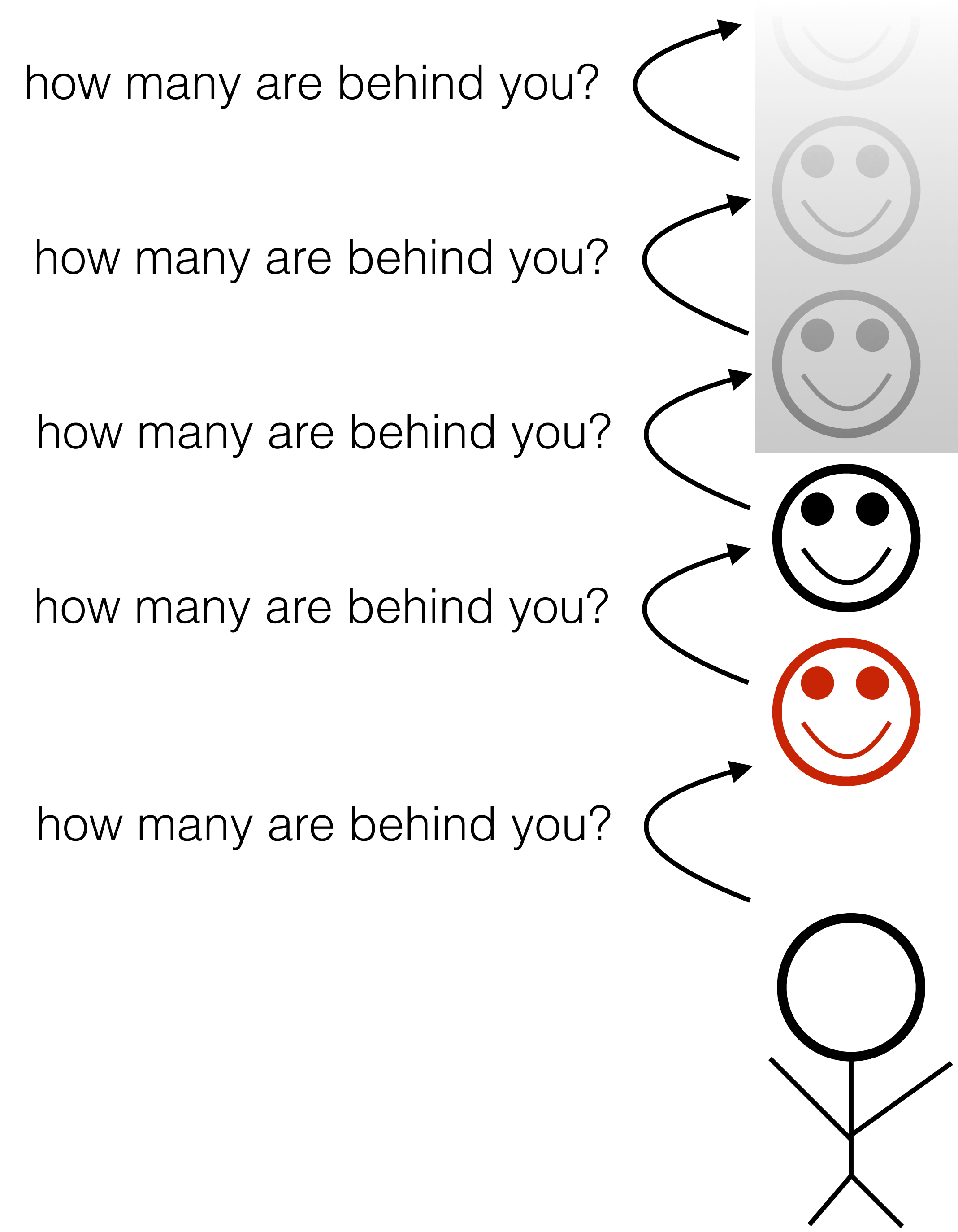



Example from https://courses.cs.washington.edu/courses/cse143/17au/

# Recursive Students

Strategy: reframe question as *"how many students are behind you?"*

Process:
**if** nobody is behind you: **say** 0
**else**: ask them, **say** their answer+1

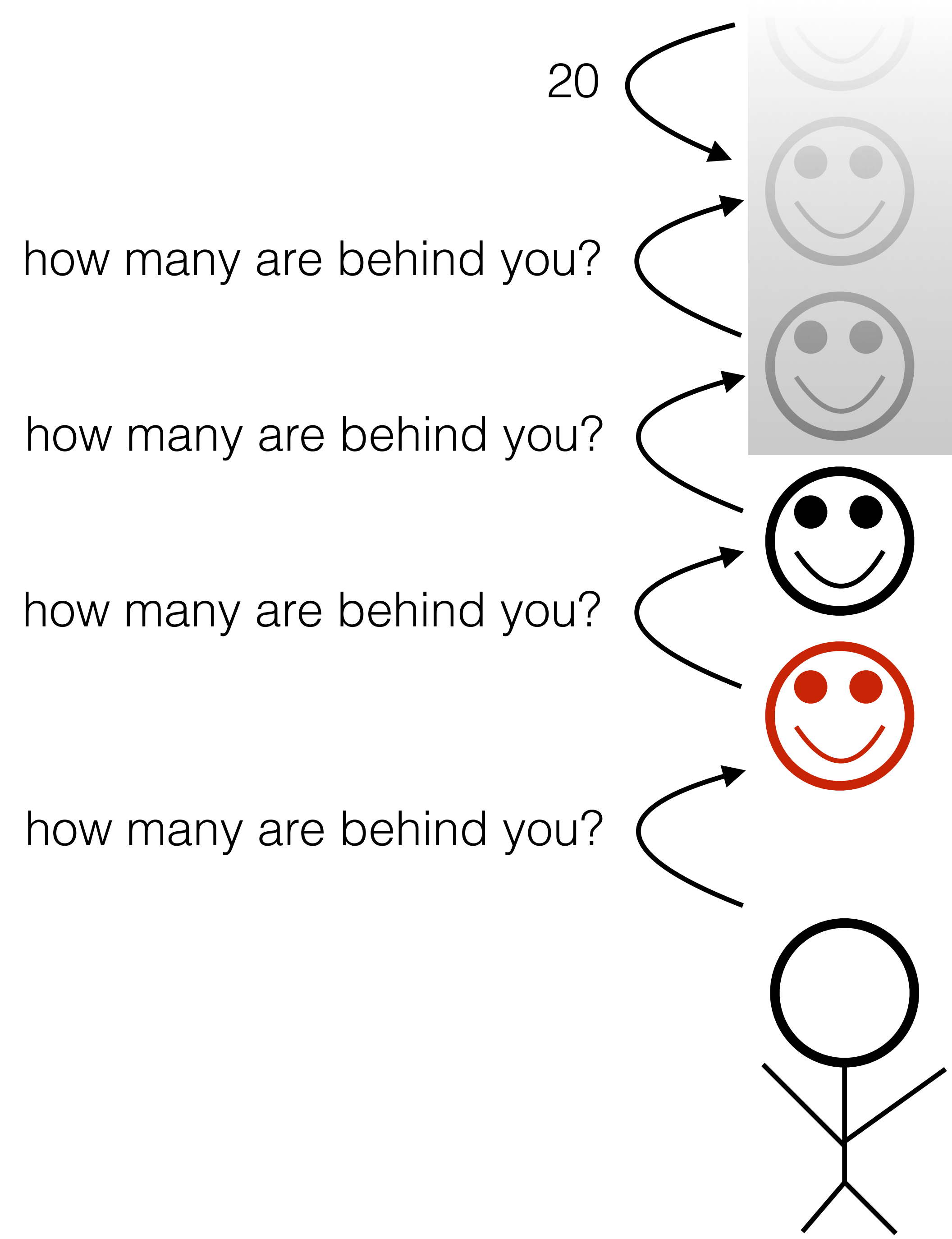



Example from https://courses.cs.washington.edu/courses/cse143/17au/

# Recursive Students

Strategy: reframe question as *"how many students are behind you?"*

Process:
**if** nobody is behind you: **say** 0
**else**: ask them, **say** their answer+1

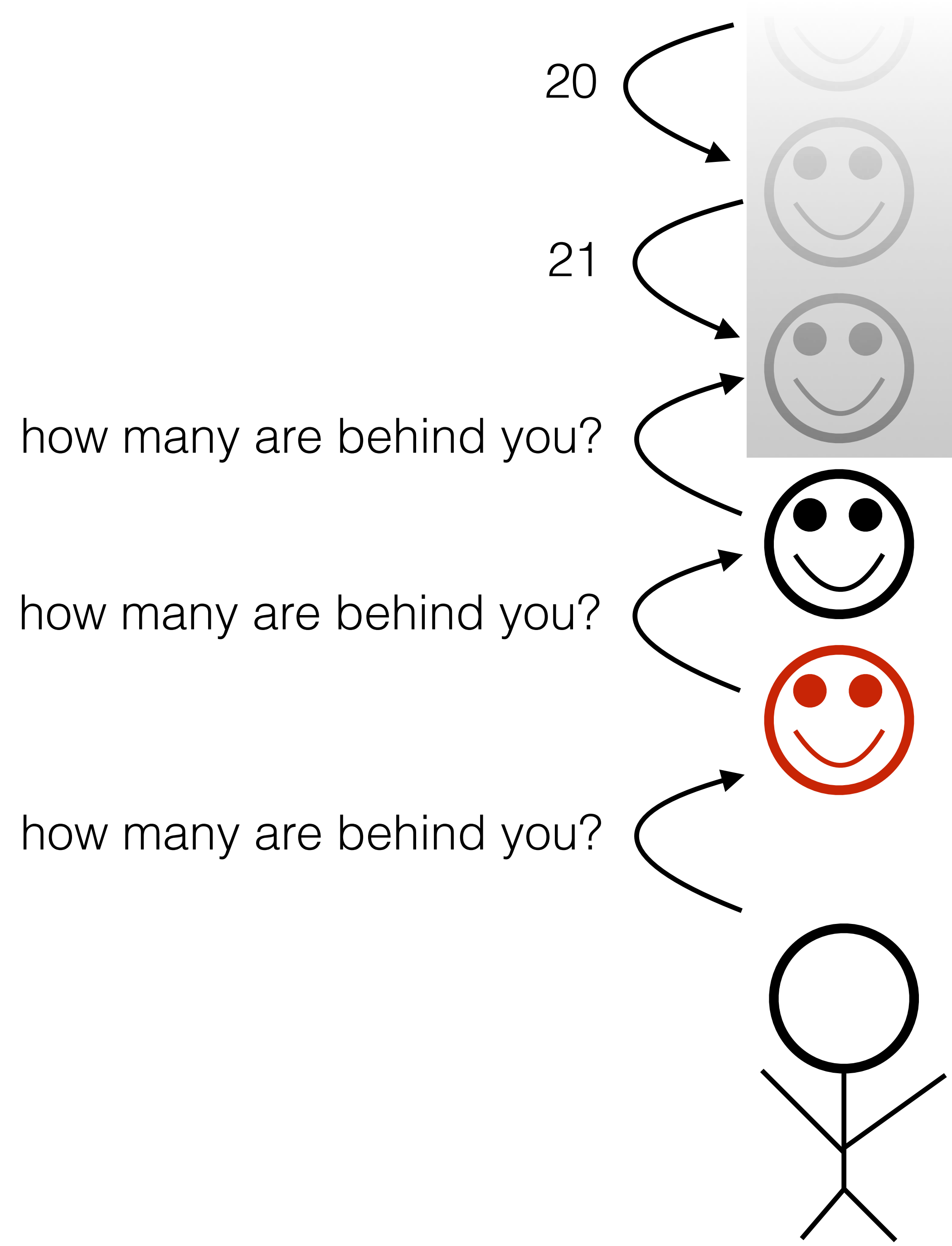



Example from https://courses.cs.washington.edu/courses/cse143/17au/

# Recursive Students

Strategy: reframe question as *"how many students are behind you?"*

Process:
**if** nobody is behind you: **say** 0
**else**: ask them, **say** their answer+1

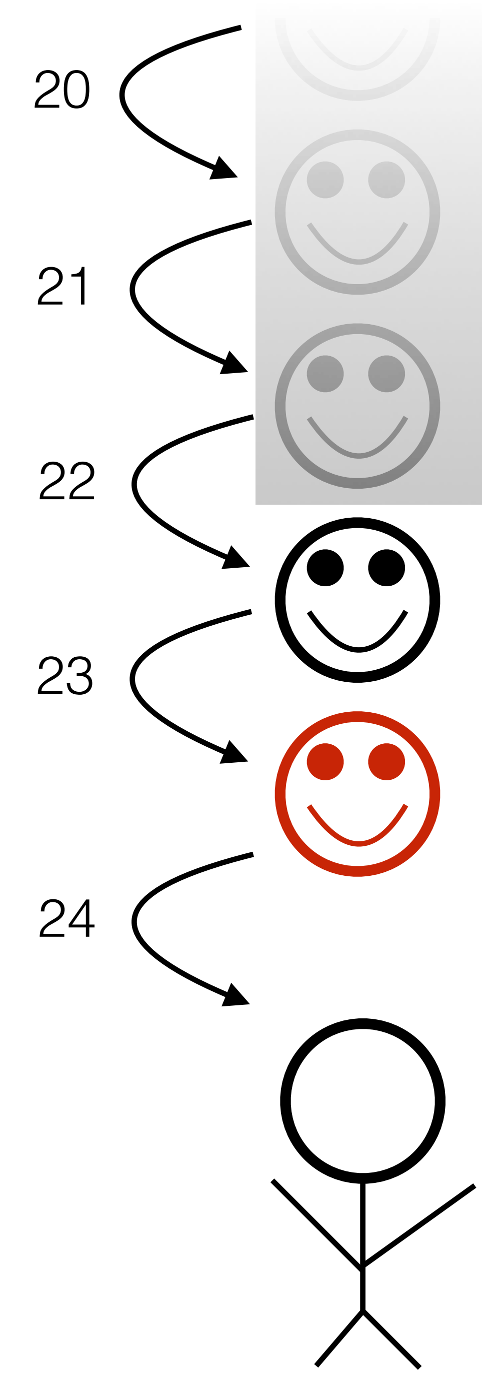



Example from https://courses.cs.washington.edu/courses/cse143/17au/

# Recursive Students

Strategy: reframe question as *"how many students are behind you?"*

Process:
**if** nobody is behind you: **say** 0
**else**: ask them, **say** their answer+1

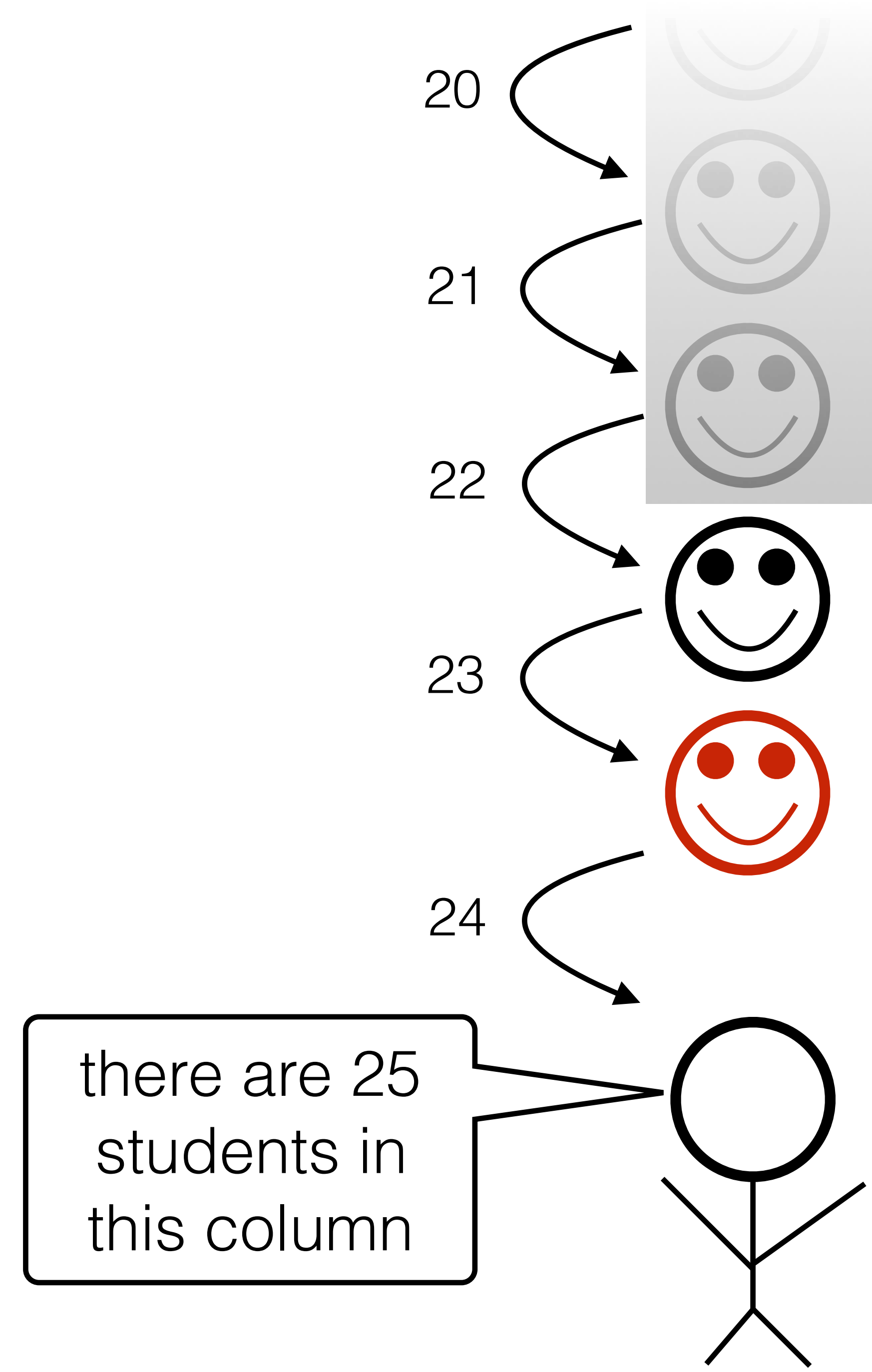



Example from https://courses.cs.washington.edu/courses/cse143/17au/

# Recursive Students

Strategy: reframe question as *"how many students are behind you?"*

Process:
**if** nobody is behind you: **say** 0
**else**: ask them, **say** their answer+1




Example from https://courses.cs.washington.edu/courses/cse143/17au/

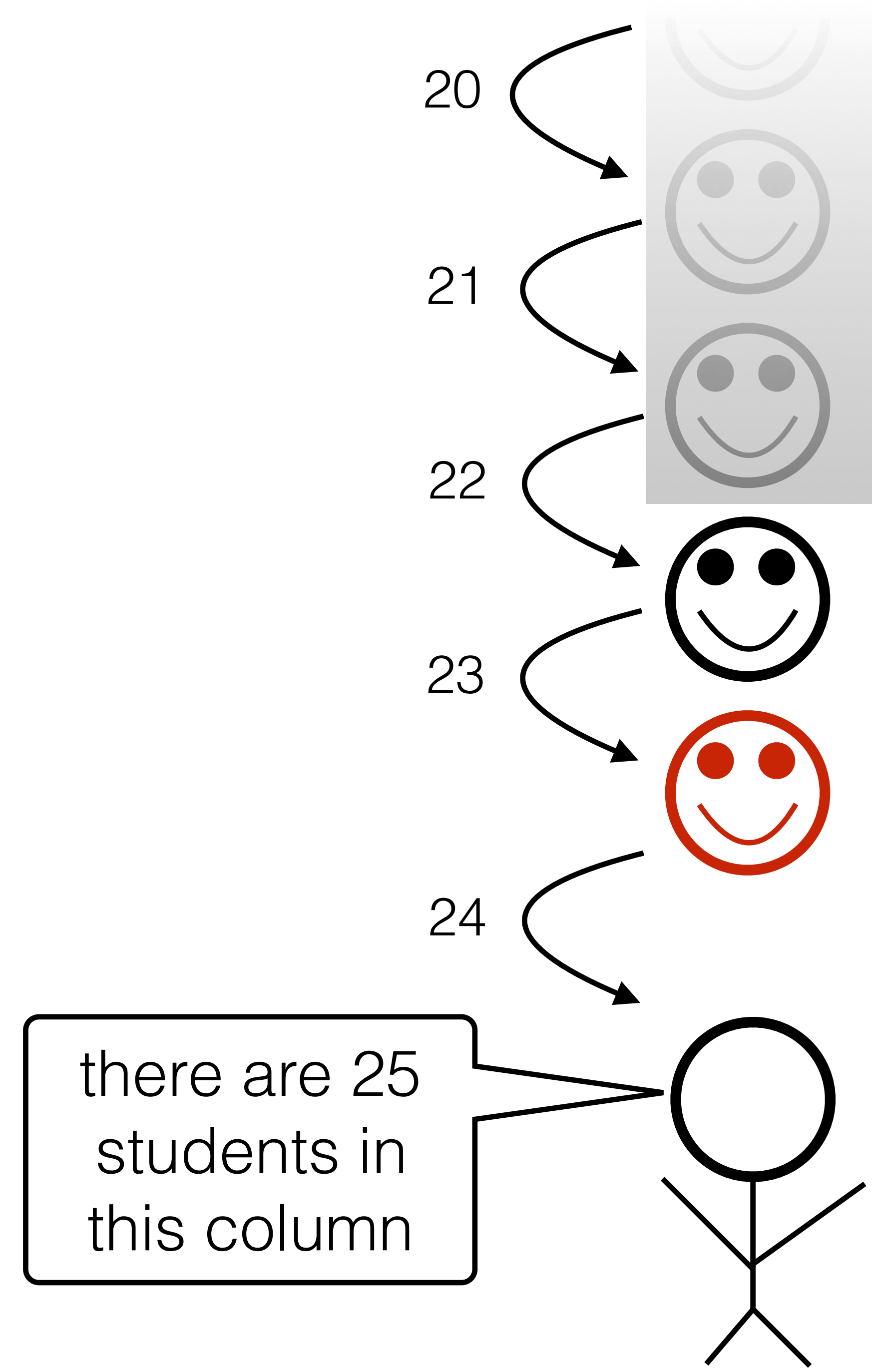
# Recursive Students

Strategy: reframe question as *"how many students are behind you?"*

Process:
**if** nobody is behind you: **say** 0
**else**: ask them, **say** their answer+1

Observations:

- Each student runs the same "code"
- Each student has their own "state"




Example from https://courses.cs.washington.edu/courses/cse143/17au/

# Example: Reframing Factorials

$$N! = 1 \times 2 \times 3 \times \ldots \times (N-2) \times (N-1) \times N$$

# Example: Factorials

**1. Examples:**

```
1! = 1
2! = 1*2 = 2
3! = 1*2*3 = 6
4! = 1*2*3*4 = 24
5! = 1*2*3*4*5 = 120
```

**2. Self Reference:**

**3. Recursive Definition:**

**4. Python Code:**

```
def fact(n):
    pass # TODO
```

Goal: work from examples to get to recursive code

# Example: Factorials

## 1. Examples:

```
1! = 1
2! = 1*2 = 2
3! = 1*2*3 = 6
4! = 1*2*3*4 = 24
5! = 1*2*3*4*5 = 120
```

## 2. Self Reference:

*look for patterns that allow rewrites with self reference*

## 3. Recursive Definition:

## 4. Python Code:

```
def fact(n):
    pass # TODO
```

# Example: Factorials

## 1. Examples:

```
1! = 1
2! = 1*2 = 2
3! = 1*2*3 = 6
4! = 1*2*3*4 = 24
5! = 1*2*3*4*5 = 120
```

## 2. Self Reference:

*look for patterns that allow rewrites with self reference*

## 3. Recursive Definition:

## 4. Python Code:

```
def fact(n):
    pass # TODO
```

# Example: Factorials

## 1. Examples:

```
1! = 1
2! = 1*2 = 2
3! = 1*2*3 = 6
4! = 1*2*3*4 = 24
5! = 1*2*3*4*5 = 120
```

## 2. Self Reference:

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

## 3. Recursive Definition:

## 4. Python Code:

```python
def fact(n):
    pass # TODO
```

# Example: Factorials

## 1. Examples:

```
1! = 1
2! = 1*2 = 2
3! = 1*2*3 = 6
4! = 1*2*3*4 = 24
5! = 1*2*3*4*5 = 120
```

## 2. Self Reference:

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

## 3. Recursive Definition:

## 4. Python Code:

```
def fact(n):
    pass # TODO
```

# Example: Factorials

## 1. Examples:

```
1! = 1
2! = 1*2 = 2
3! = 1*2*3 = 6
4! = 1*2*3*4 = 24
5! = 1*2*3*4*5 = 120
```

## 2. Self Reference:

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

## 3. Recursive Definition:

## 4. Python Code:

```
def fact(n):
    pass # TODO
```

# Example: Factorials

## 1. Examples:

```
1! = 1
2! = 1*2 = 2
3! = 1*2*3 = 6
4! = 1*2*3*4 = 24
5! = 1*2*3*4*5 = 120
```

## 2. Self Reference:

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

## 3. Recursive Definition:

## 4. Python Code:

```
def fact(n):
    pass # TODO
```

# Example: Factorials

## 1. Examples:

```
1! = 1
2! = 1*2 = 2
3! = 1*2*3 = 6
4! = 1*2*3*4 = 24
5! = 1*2*3*4*5 = 120
```

## 2. Self Reference:

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

*don't need a pattern at the start*

## 3. Recursive Definition:

## 4. Python Code:

```python
def fact(n):
    pass # TODO
```

# Example: Factorials

## 1. Examples:

```
1! = 1
2! = 1*2 = 2
3! = 1*2*3 = 6
4! = 1*2*3*4 = 24
5! = 1*2*3*4*5 = 120
```

## 2. Self Reference:

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

## 3. Recursive Definition:

*convert self-referring examples to a recursive definition*

## 4. Python Code:

```python
def fact(n):
    pass # TODO
```

# Example: Factorials

## 1. Examples:

```
1! = 1
2! = 1*2 = 2
3! = 1*2*3 = 6
4! = 1*2*3*4 = 24
5! = 1*2*3*4*5 = 120
```

## 2. Self Reference:

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

## 3. Recursive Definition:

`1!` is `1`

## 4. Python Code:

```
def fact(n):
    pass # TODO
```

# Example: Factorials

## 1. Examples:

```
1! = 1
2! = 1*2 = 2
3! = 1*2*3 = 6
4! = 1*2*3*4 = 24
5! = 1*2*3*4*5 = 120
```

## 2. Self Reference:

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

## 3. Recursive Definition:

`1!` is `1`

`N!` is `????` for N>1

## 4. Python Code:

```
def fact(n):
    pass # TODO
```

# Example: Factorials

### 1. Examples:

```
1! = 1
2! = 1*2 = 2
3! = 1*2*3 = 6
4! = 1*2*3*4 = 24
5! = 1*2*3*4*5 = 120
```

### 2. Self Reference:

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

### 3. Recursive Definition:

`1!` is `1`

`N!` is `(N-1)! * N` for N>1

### 4. Python Code:

```
def fact(n):
    pass # TODO
```

# Example: Factorials

## 1. Examples:

```
1! = 1
2! = 1*2 = 2
3! = 1*2*3 = 6
4! = 1*2*3*4 = 24
5! = 1*2*3*4*5 = 120
```

## 2. Self Reference:

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

## 3. Recursive Definition:

`1!` is `1`

`N!` is `(N-1)! * N` for N>1

## 4. Python Code:

```
def fact(n):
    pass # TODO
```

# Example: Factorials

## 1. Examples:

```
1! = 1
2! = 1*2 = 2
3! = 1*2*3 = 6
4! = 1*2*3*4 = 24
5! = 1*2*3*4*5 = 120
```

## 2. Self Reference:

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

## 3. Recursive Definition:

`1!` is `1`

`N!` is `(N-1)! * N` for N>1

## 4. Python Code:

```
def fact(n):
    if n == 1:
        return 1
```

# Example: Factorials

## 1. Examples:

```
1! = 1
2! = 1*2 = 2
3! = 1*2*3 = 6
4! = 1*2*3*4 = 24
5! = 1*2*3*4*5 = 120
```

## 2. Self Reference:

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

## 3. Recursive Definition:

`1!` is `1`

`N!` is `(N-1)! * N` for N>1

## 4. Python Code:

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

# Example: Factorials

## 1. Examples:

```
1! = 1
2! = 1*2 = 2
3! = 1*2*3 = 6
4! = 1*2*3*4 = 24
5! = 1*2*3*4*5 = 120
```

## 2. Self Reference:

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

## 3. Recursive Definition:

`1!` is `1`

`N!` is `(N-1)! * N` for N>1

## 4. Python Code:

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

# Example: Factorials

## 1. Examples:

```
1! = 1
2! = 1*2 = 2
3! = 1*2*3 = 6
4! = 1*2*3*4 = 24
5! = 1*2*3*4*5 = 120
```

## 2. Self Reference:

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

## 3. Recursive Definition:

`1!` is `1`

`N!` is `(N-1)! * N` for N>1

## 4. Python Code:

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

Let’s “run” it!

# Tracing Factorial

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

**fact(n=4)**

# Tracing Factorial

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

**fact(n=4)**

if n == 1:

# Tracing Factorial

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

**fact(n=4)**

if n == 1:

# Tracing Factorial

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

**fact(n=4)**

if n == 1:

**fact(n=3)**

# Tracing Factorial

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

**fact(n=4)**

if n == 1:

**fact(n=3)**

if n == 1:

# Tracing Factorial

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

**fact(n=4)**

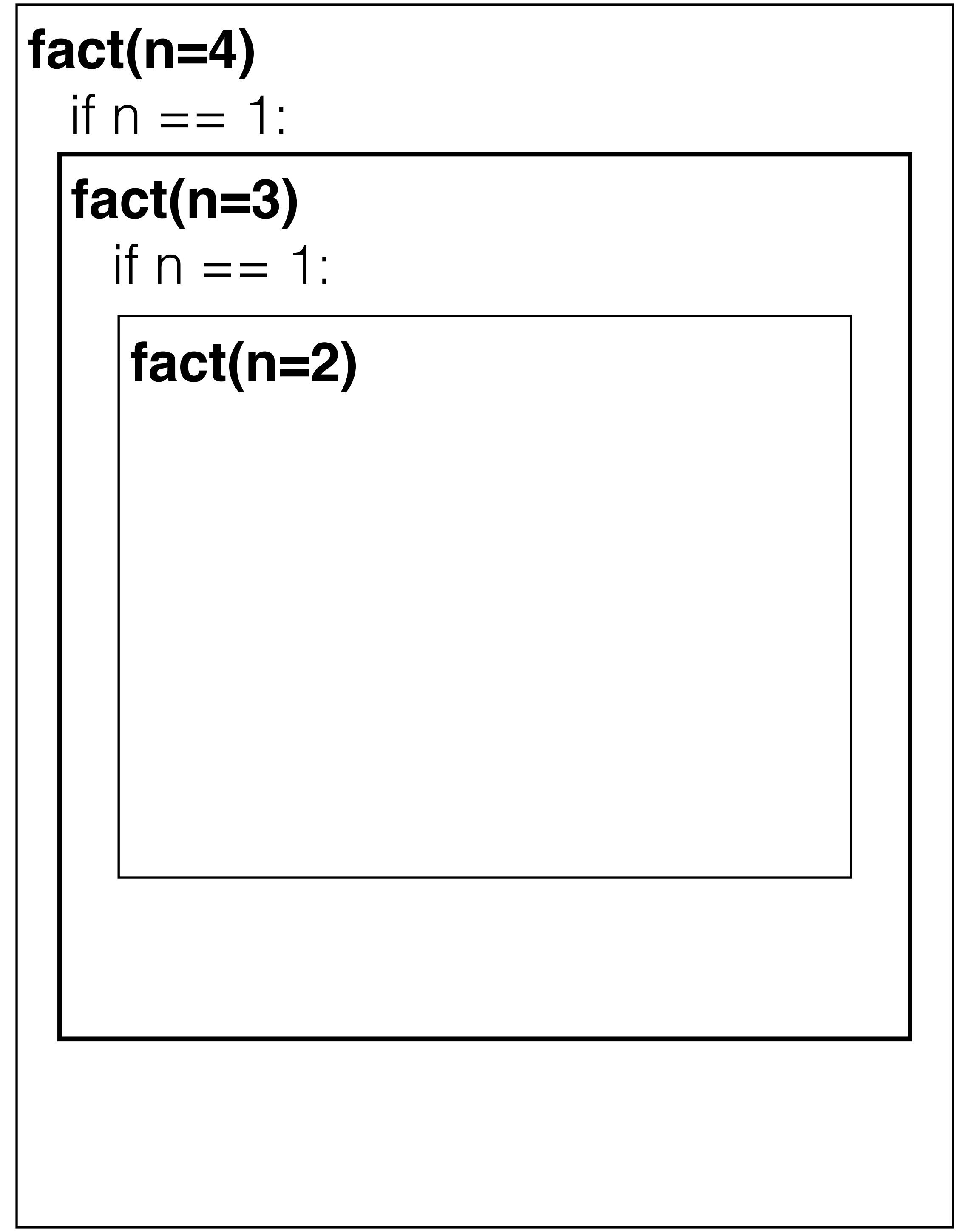
if n == 1:

**fact(n=3)**

if n == 1:

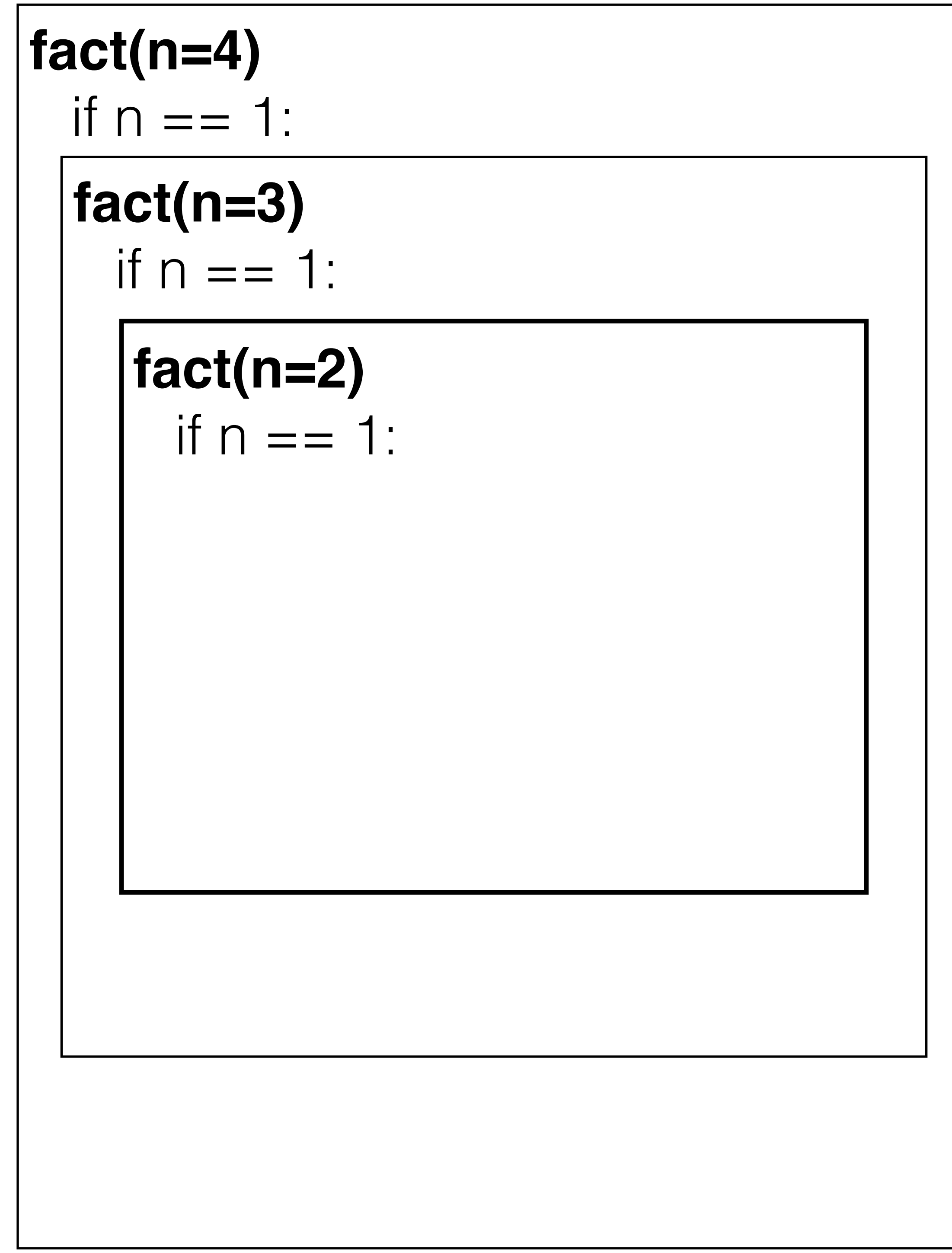
# Tracing Factorial

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



# Tracing Factorial

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



# Tracing Factorial

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

**fact(n=4)**

if n == 1:

**fact(n=3)**

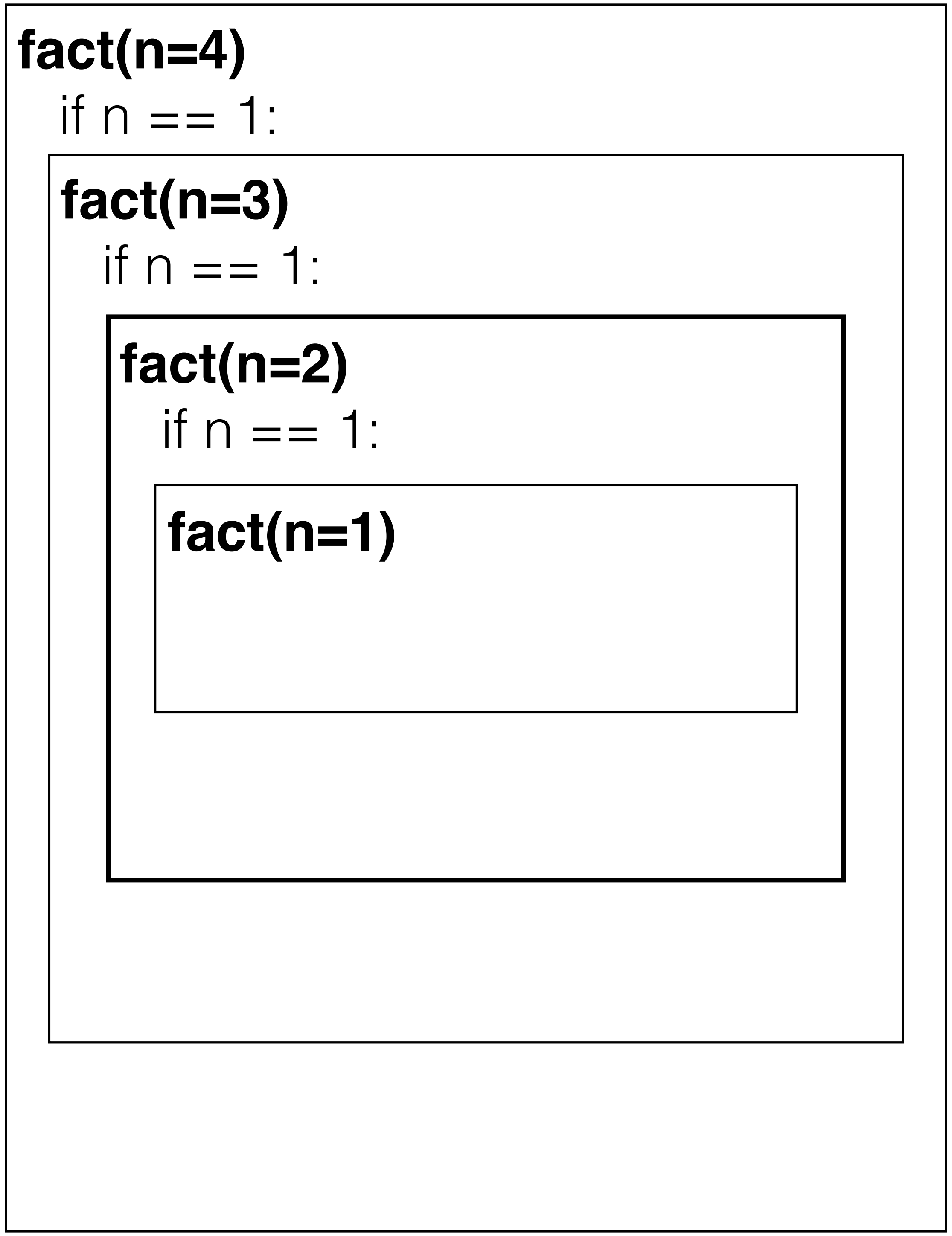
if n == 1:

**fact(n=2)**

if n == 1:

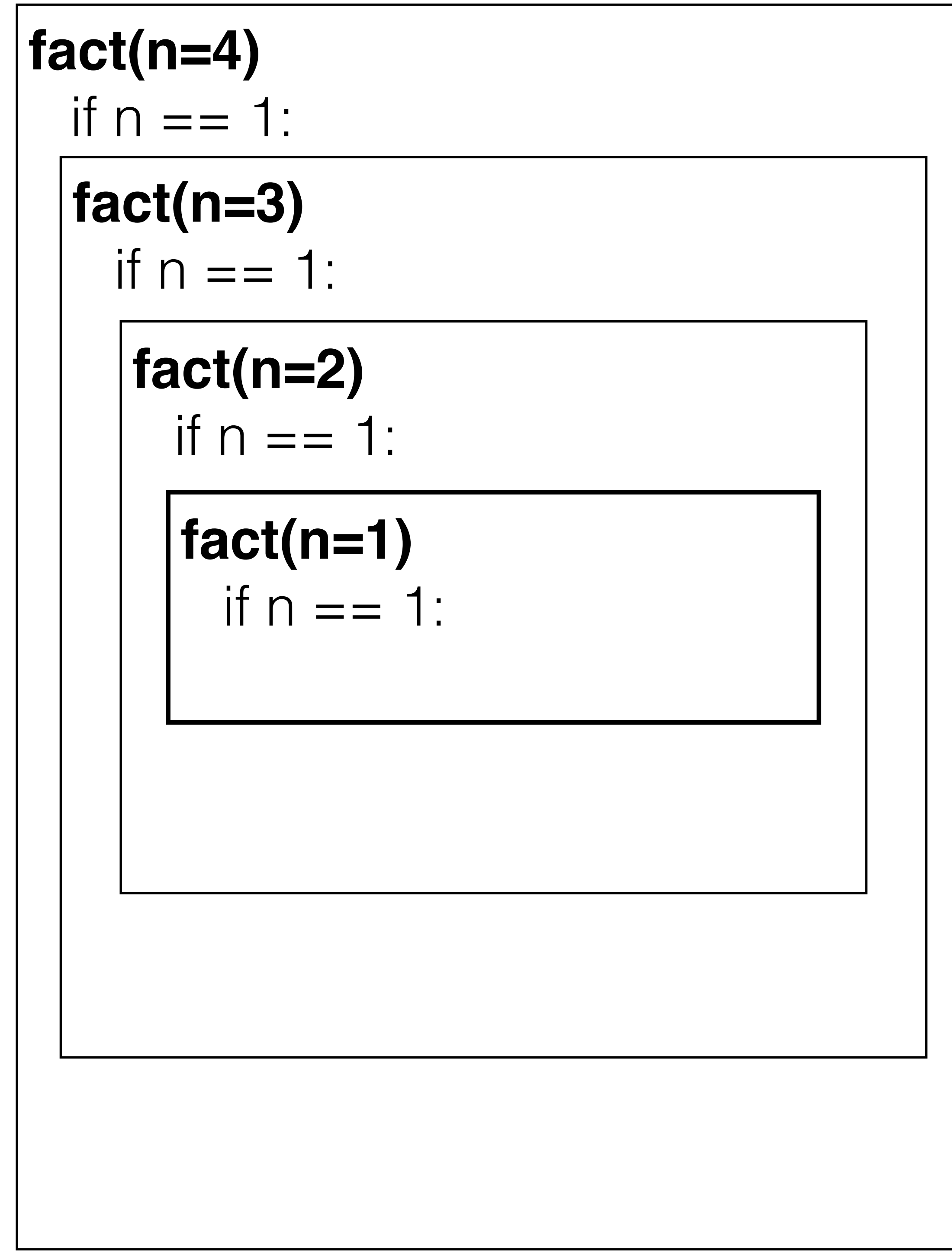
# Tracing Factorial

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



# Tracing Factorial

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



# Tracing Factorial

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

**fact(n=4)**
if n == 1:

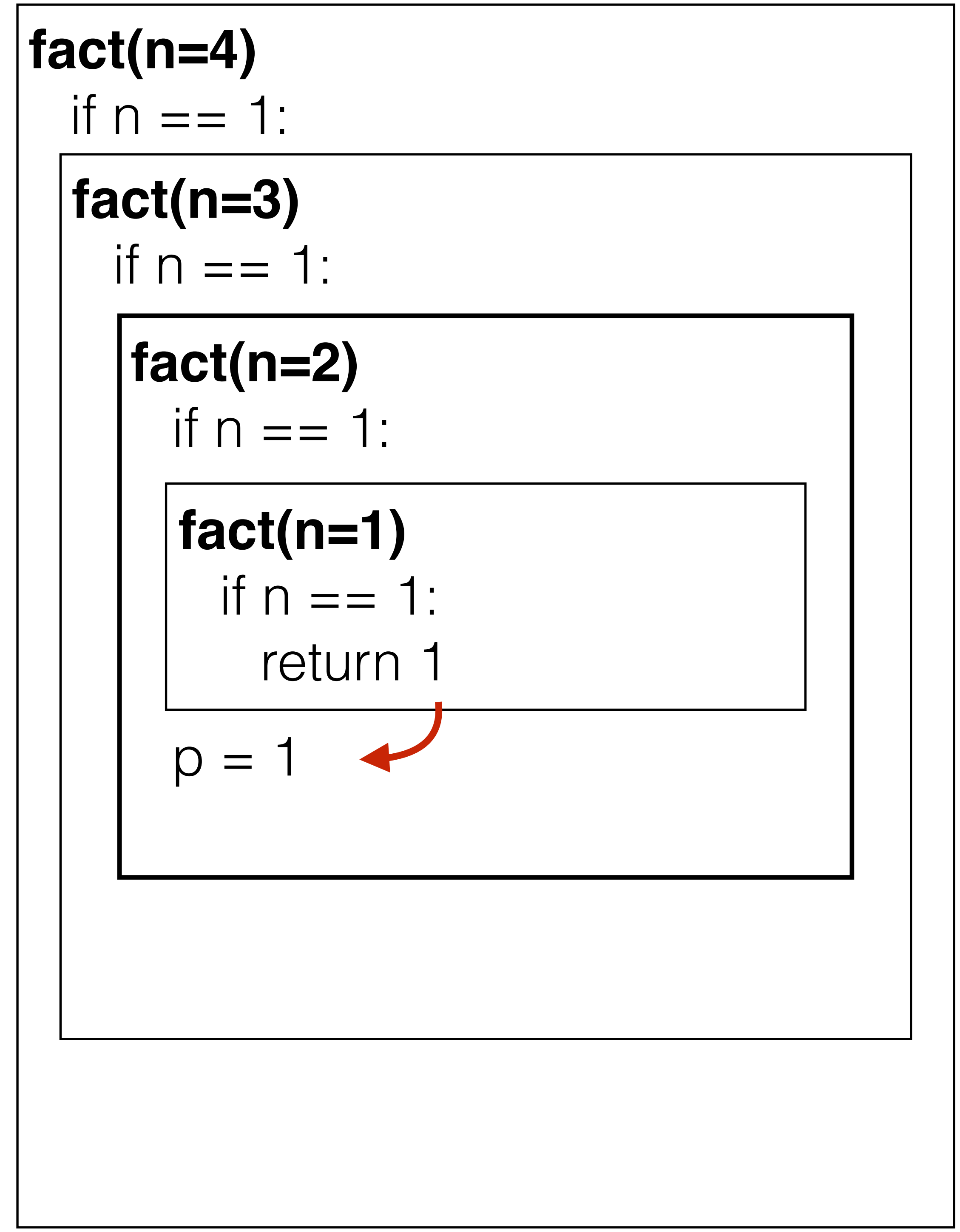
**fact(n=3)**
if n == 1:

**fact(n=2)**
if n == 1:

**fact(n=1)**
if n == 1:
return 1

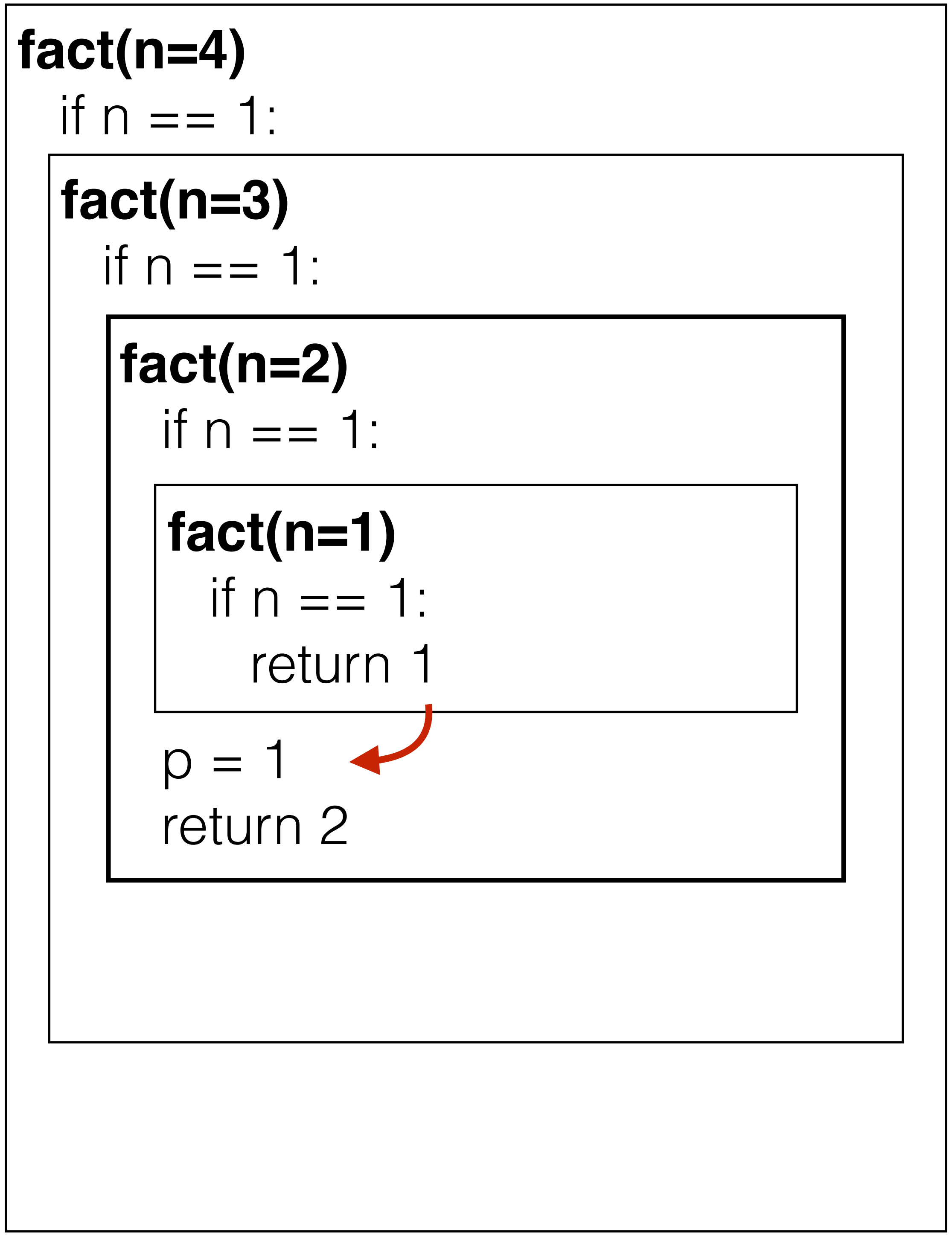
# Tracing Factorial

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



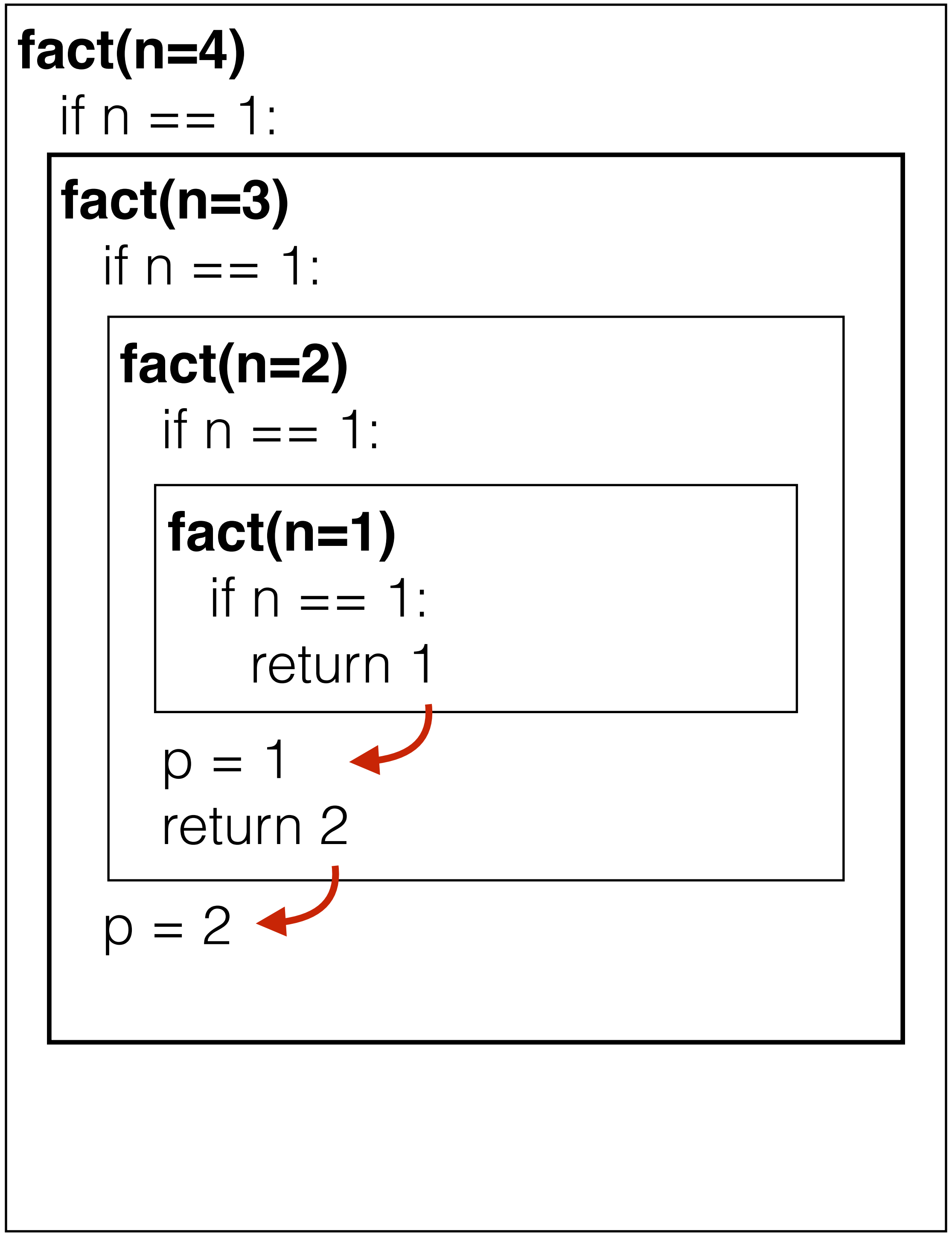
# Tracing Factorial

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



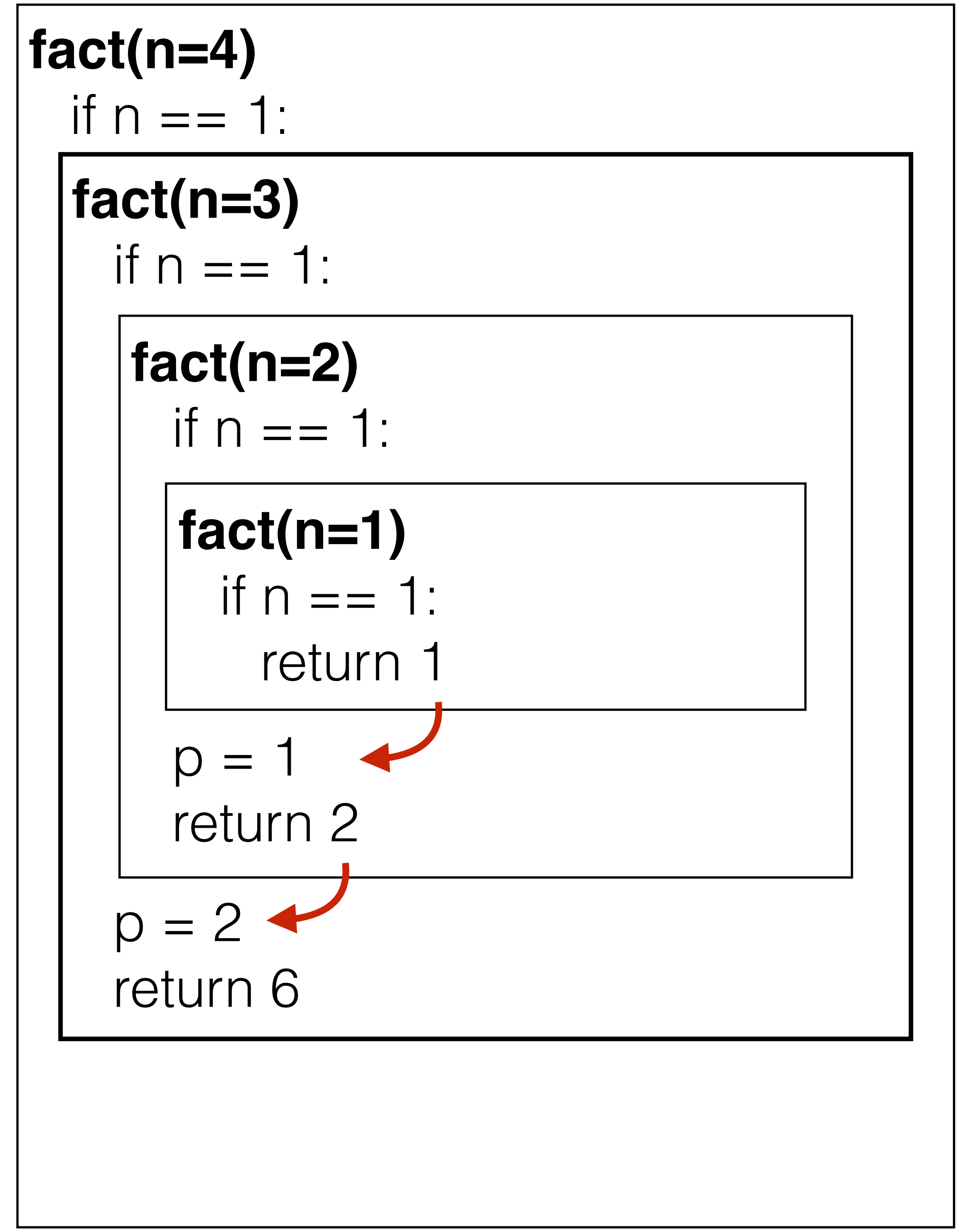
# Tracing Factorial

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



# Tracing Factorial

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



# Tracing Factorial

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

**fact(n=4)**
if n == 1:

**fact(n=3)**
if n == 1:

**fact(n=2)**
if n == 1:

**fact(n=1)**
if n == 1:
return 1

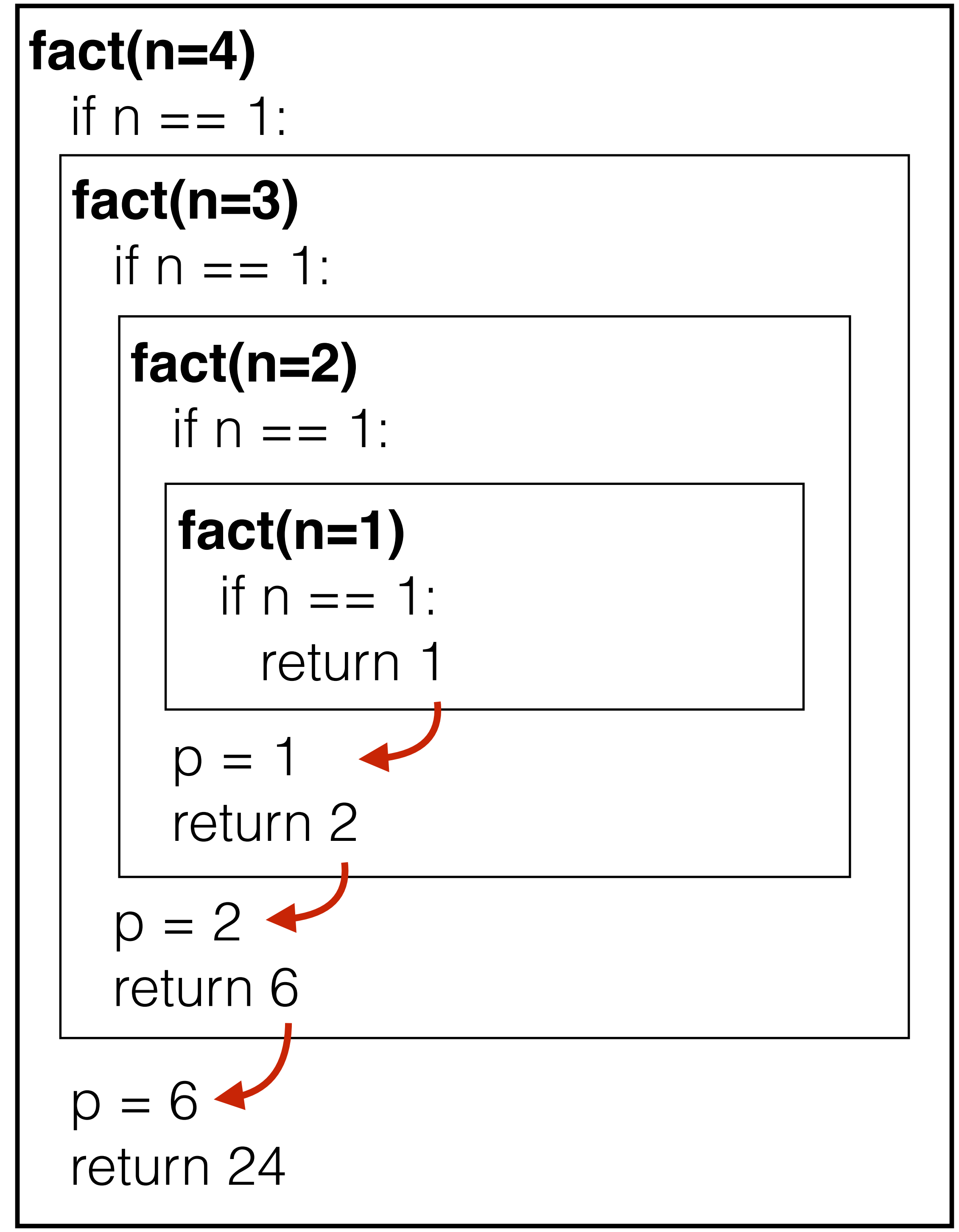
p = 1
return 2

p = 2
return 6

p = 6

# Tracing Factorial

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



# Tracing Factorial

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

```
fact(n=4)
  if n == 1:
    fact(n=3)
      if n == 1:
        fact(n=2)
          if n == 1:
            fact(n=1)
              if n == 1:
                return 1
          p = 1
          return 2
      p = 2
      return 6
  p = 6
  return 24
```

# Tracing Factorial

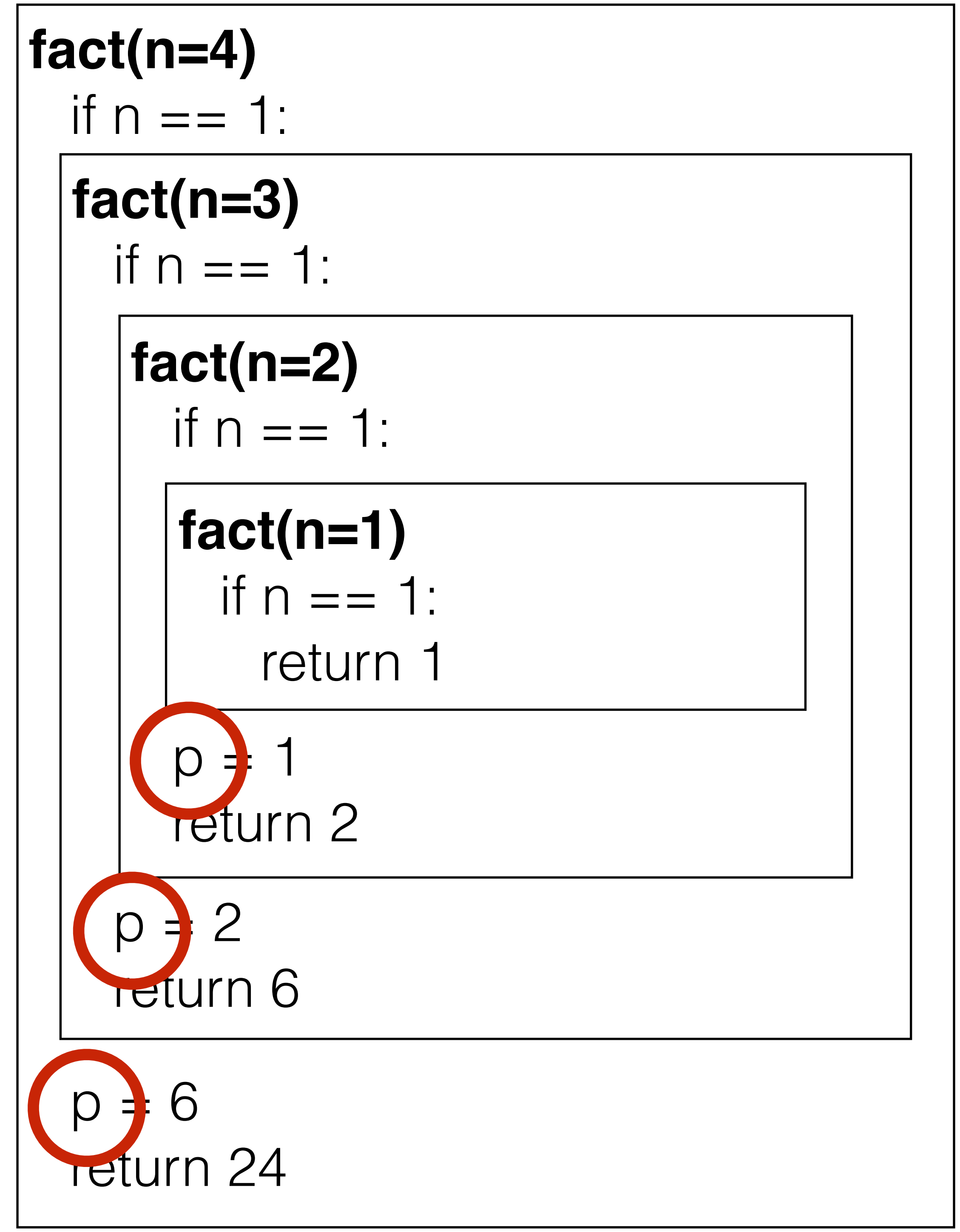
```python
def fact(n):
    if n == 1:
        return 1
    p = fact(n-1)
    return n * p
```

How does Python keep
all the P variables separate?

```
fact(n=4)
  if n == 1:
  fact(n=3)
    if n == 1:
    fact(n=2)
      if n == 1:
      fact(n=1)
        if n == 1:
          return 1
      p = 1
      return 2
    p = 2
    return 6
  p = 6
  return 24
```

# Tracing Factorial

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




How does Python keep
all the P variables separate?

frames to the rescue!

# Deep Dive: Invocation State

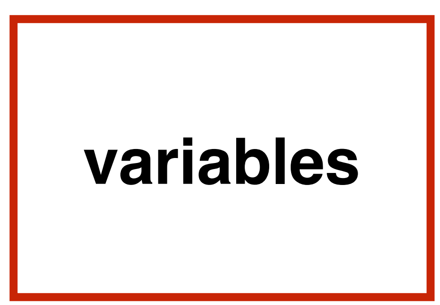
In recursion, each function invocation has its **own state**, but multiple invocations **share code**.

# Deep Dive: Invocation State

In recursion, each function invocation has its **own state**, but multiple invocations **share code**.

Variables for an invocation exist in a ***frame***

frame:

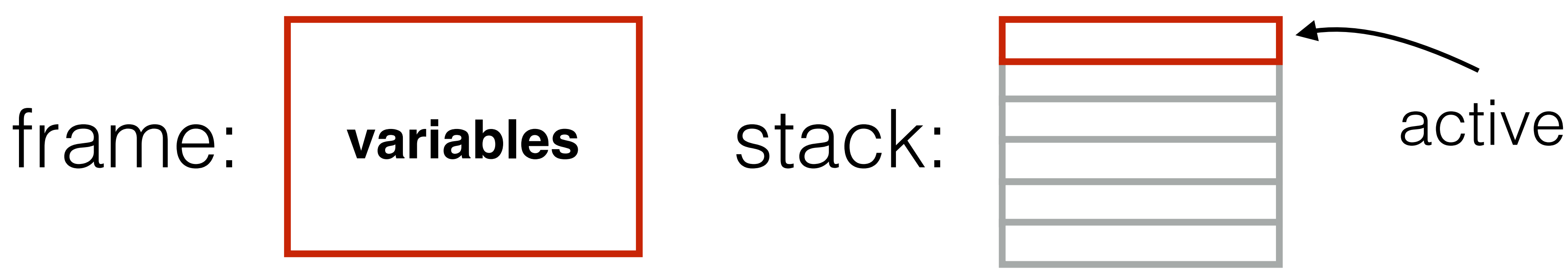


# Deep Dive: Invocation State

In recursion, each function invocation has its **own state**, but multiple invocations **share code**.

Variables for an invocation exist in a ***frame***

- the frames are stored in something called the **runtime stack**

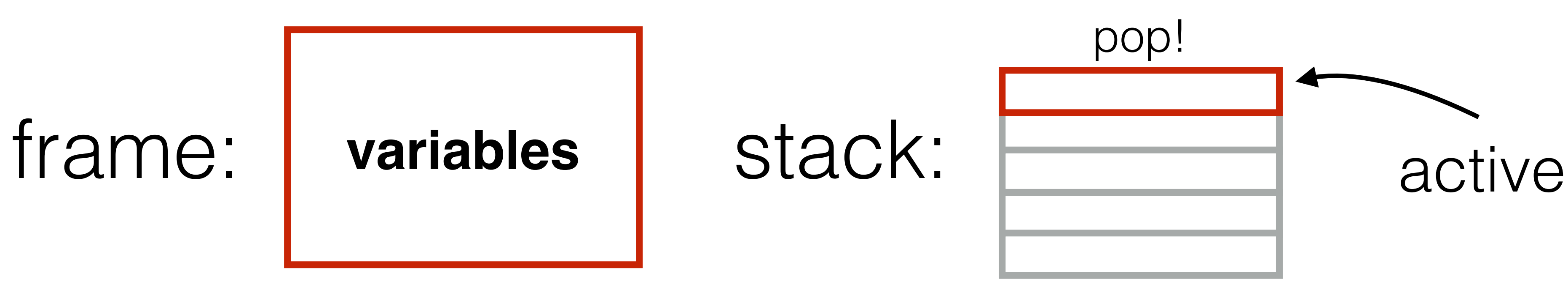


# Deep Dive: Invocation State

In recursion, each function invocation has its **own state**, but multiple invocations **share code**.

Variables for an invocation exist in a ***frame***

- the frames are stored in something called the **runtime stack**
- one invocation is active at a time: its frame is on the top of stack



# Deep Dive: Invocation State

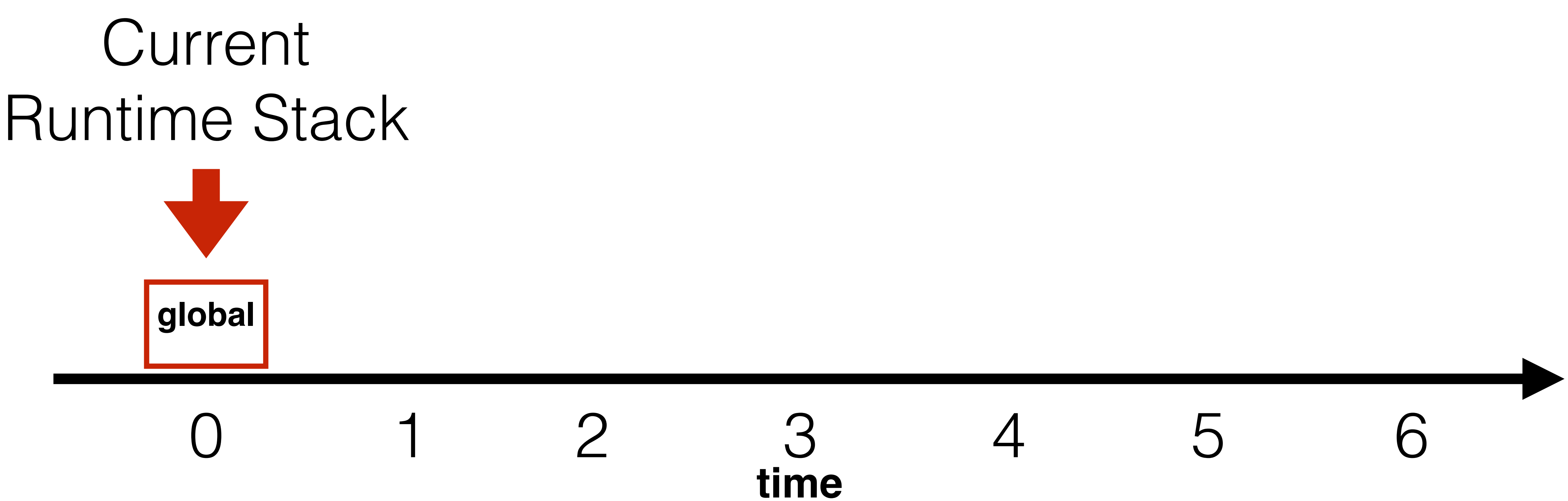
In recursion, each function invocation has its **own state**, but multiple invocations **share code**.

Variables for an invocation exist in a ***frame***

- the frames are stored in something called the **runtime stack**
- one invocation is active at a time: its frame is on the top of stack
- if a function calls itself, there will be multiple frames at the same time for the multiple invocations of the same function

# Deep Dive: Runtime Stack

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

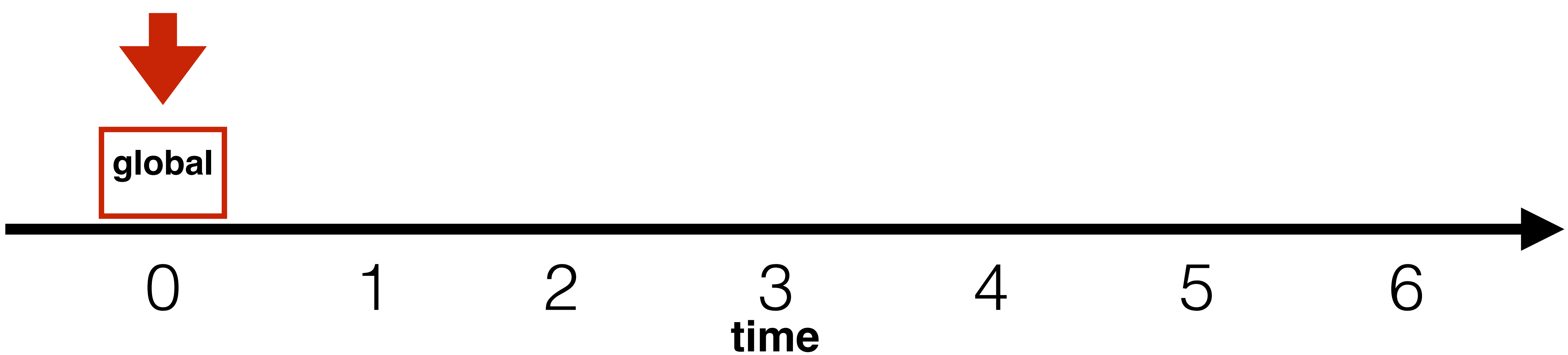


# Deep Dive: Runtime Stack

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

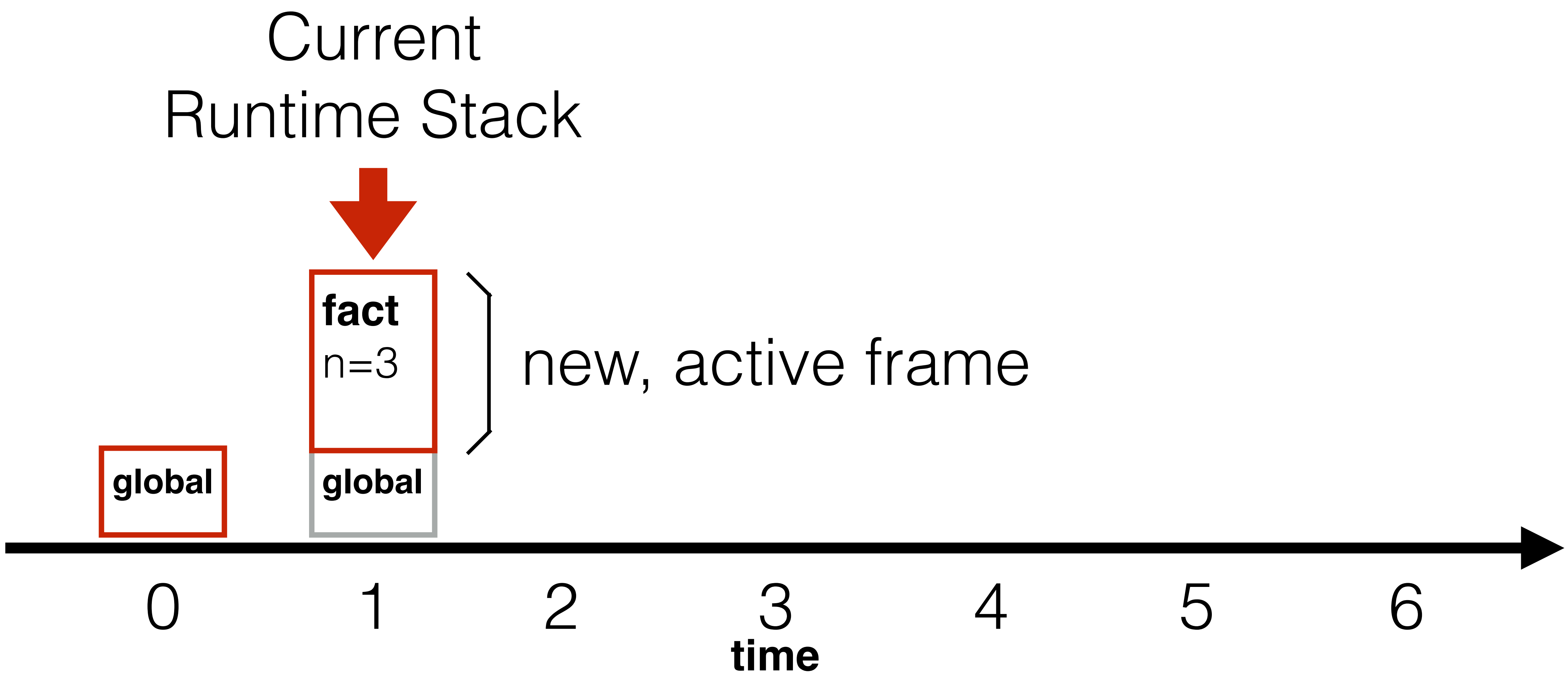
call `fact(3)`

Current Runtime Stack



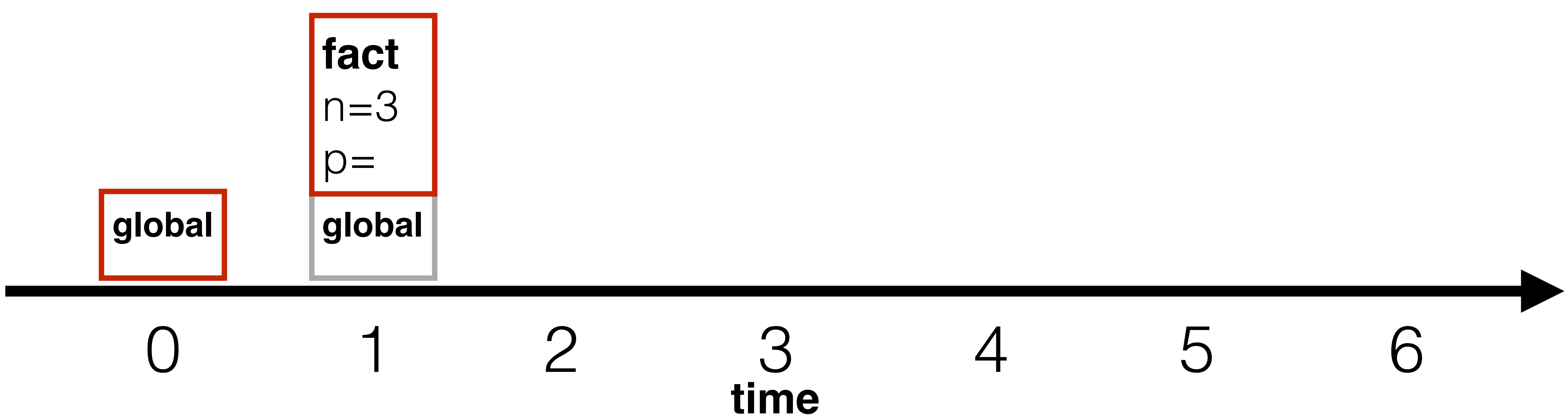
# Deep Dive:
# Runtime Stack

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



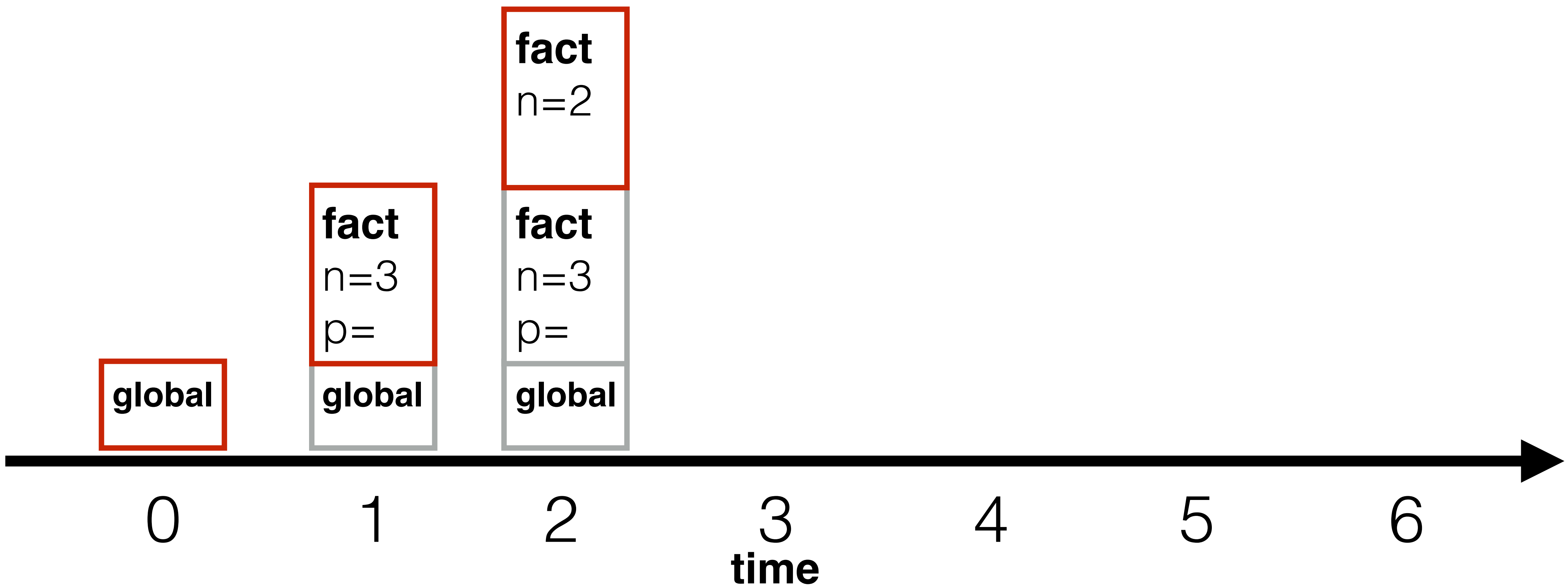
# Deep Dive: Runtime Stack

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



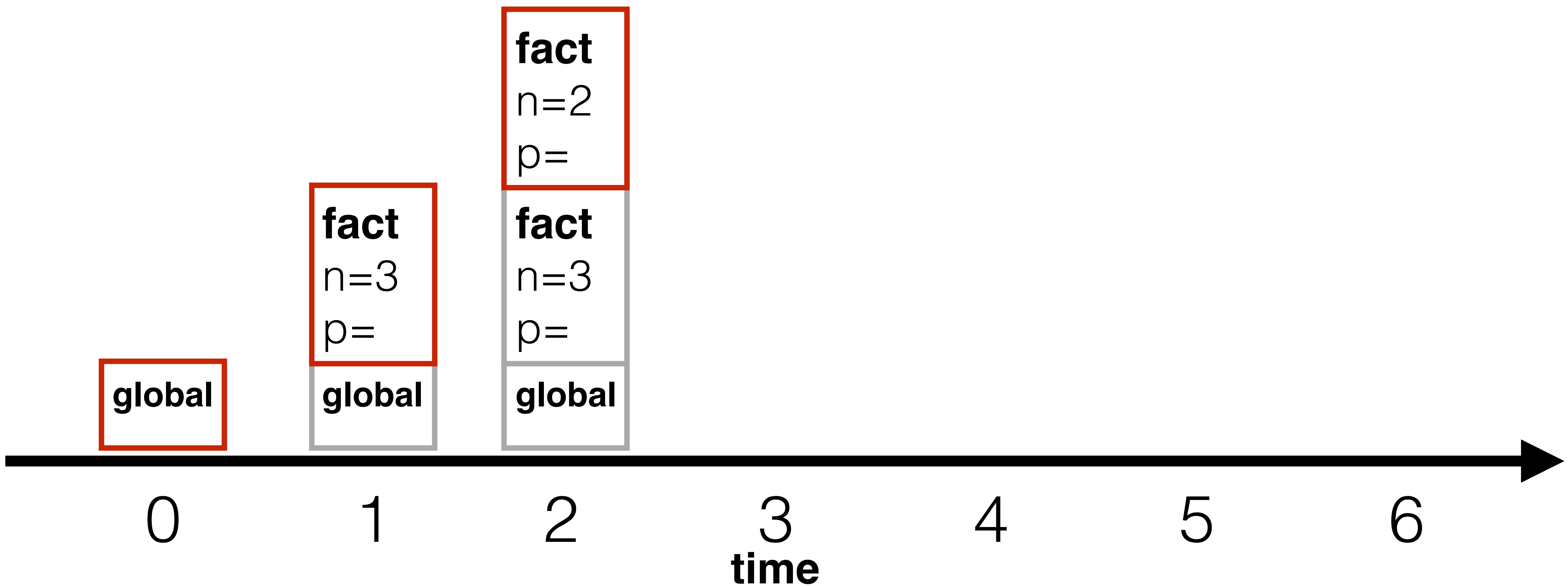
# Deep Dive: Runtime Stack

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



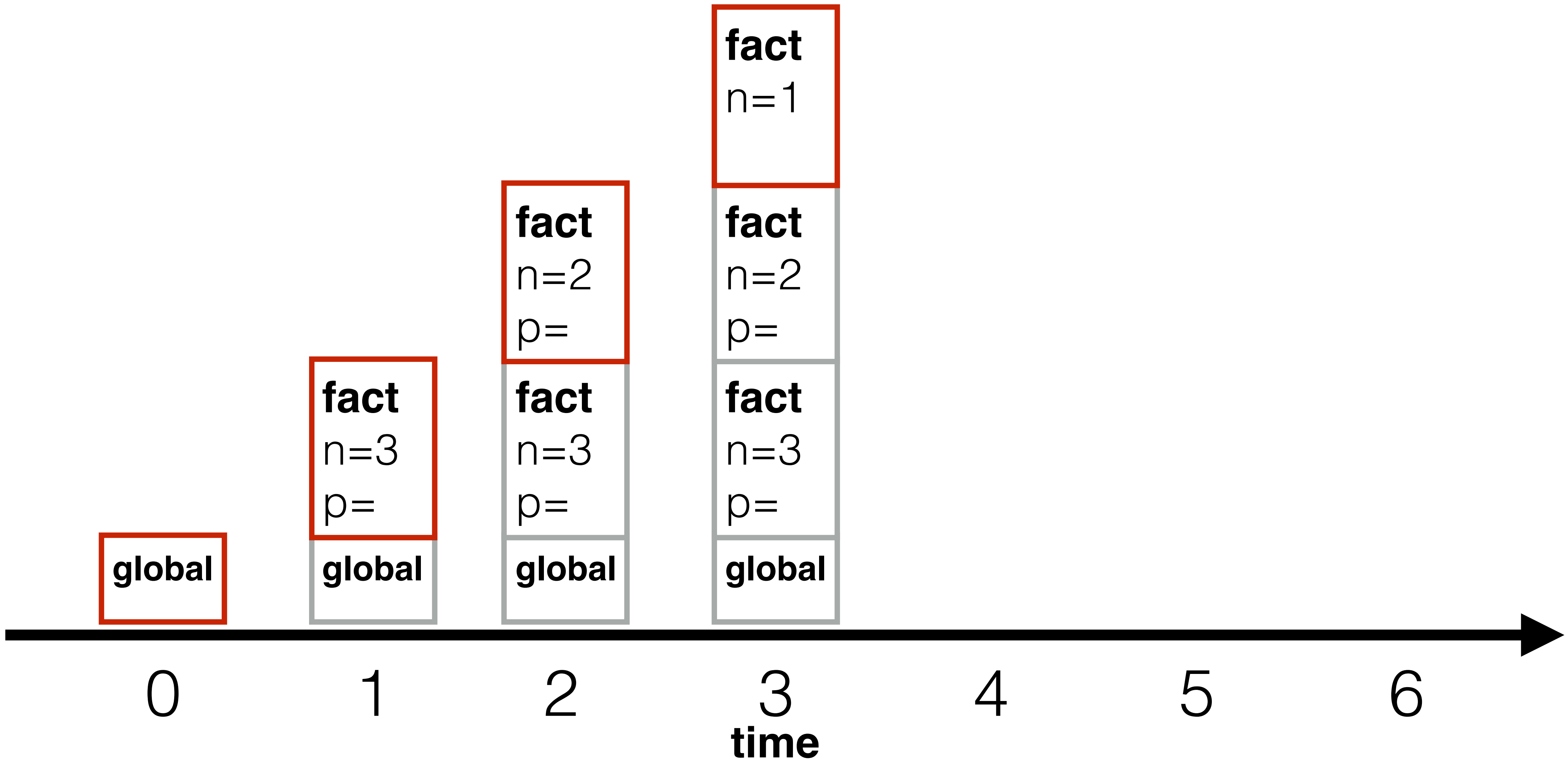
# Deep Dive:
# Runtime Stack

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



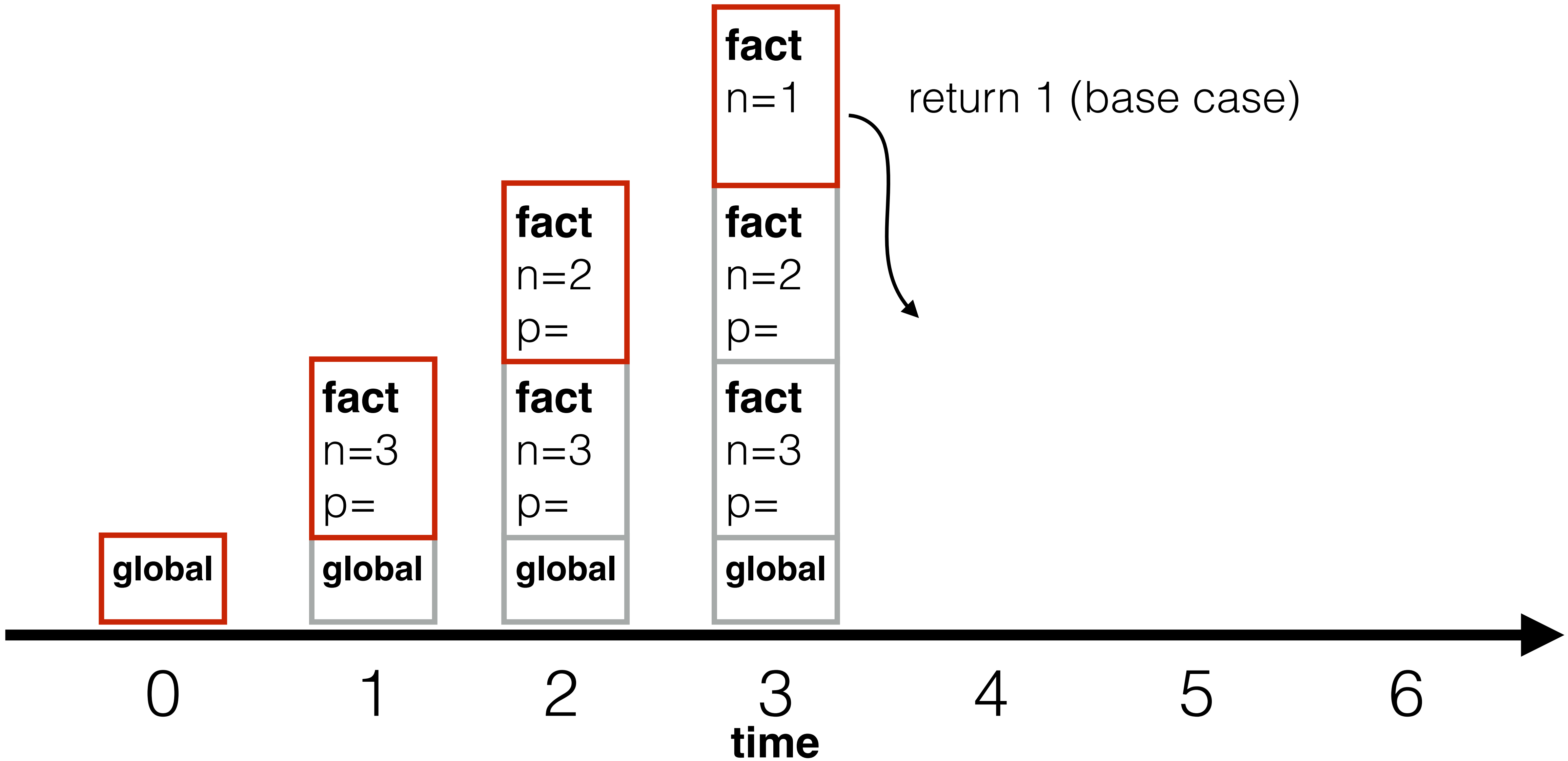
# Deep Dive: Runtime Stack

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



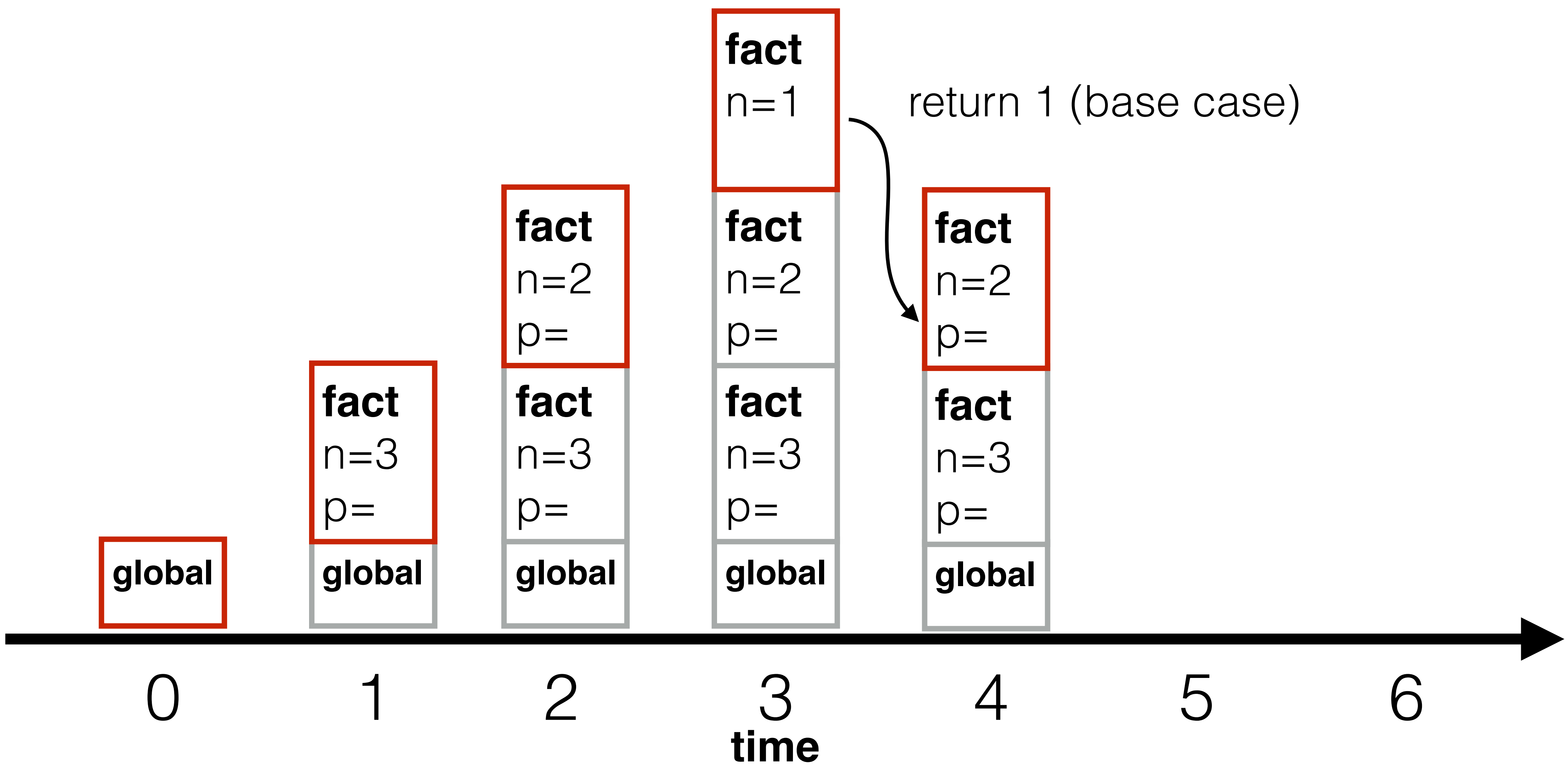
# Deep Dive: Runtime Stack

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



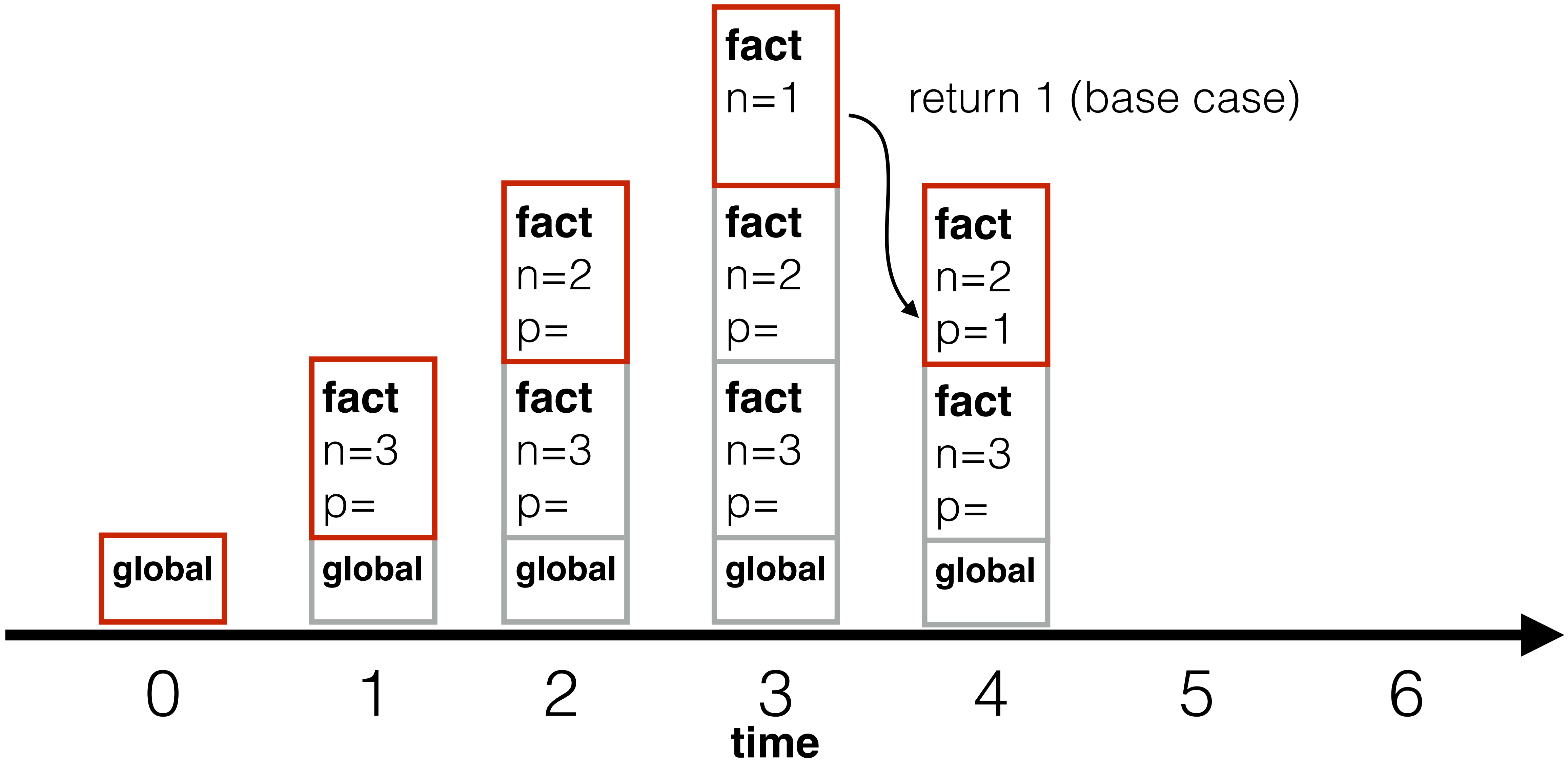
# Deep Dive: Runtime Stack

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



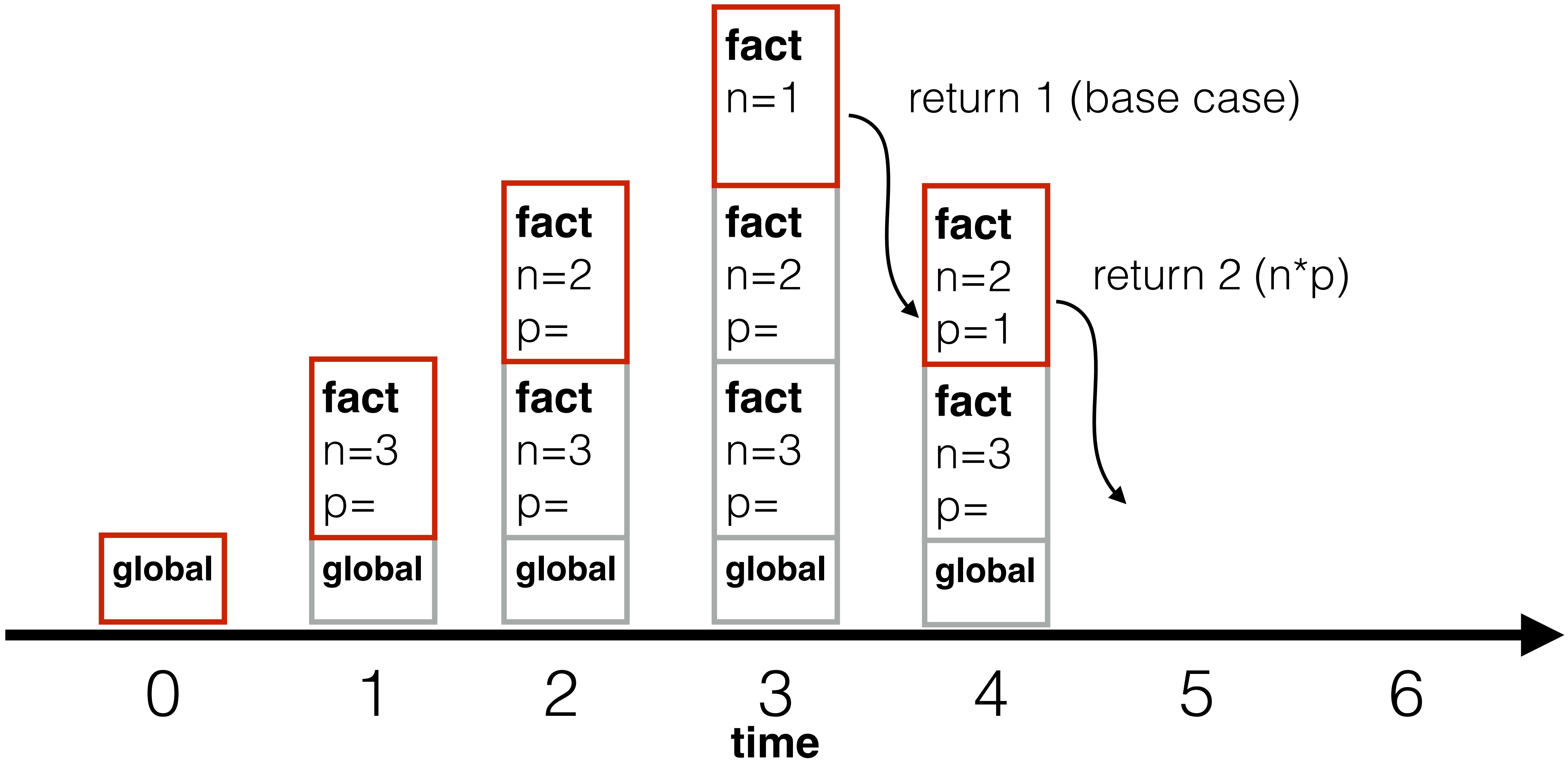
# Deep Dive: Runtime Stack

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



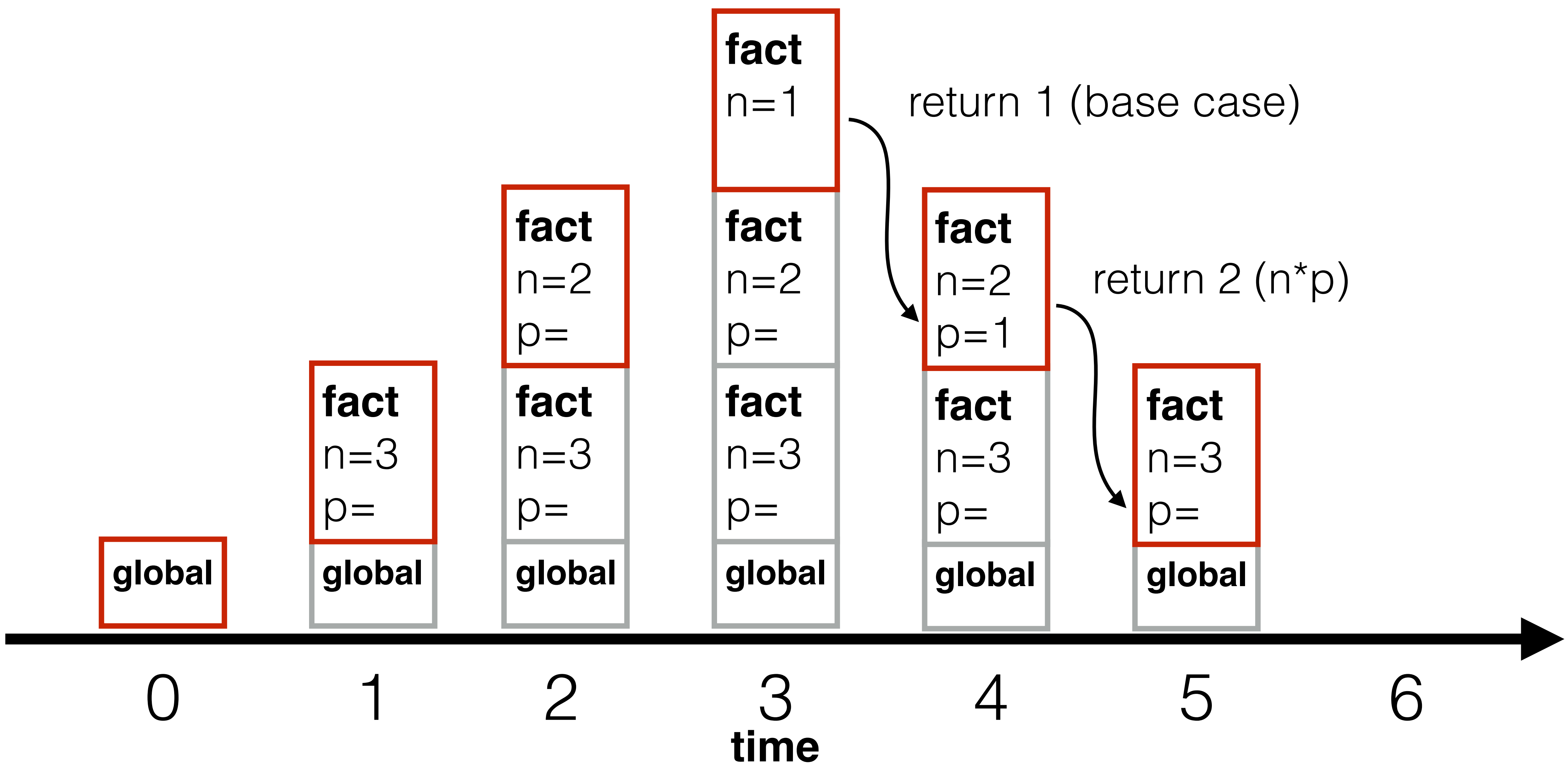
# Deep Dive: Runtime Stack

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



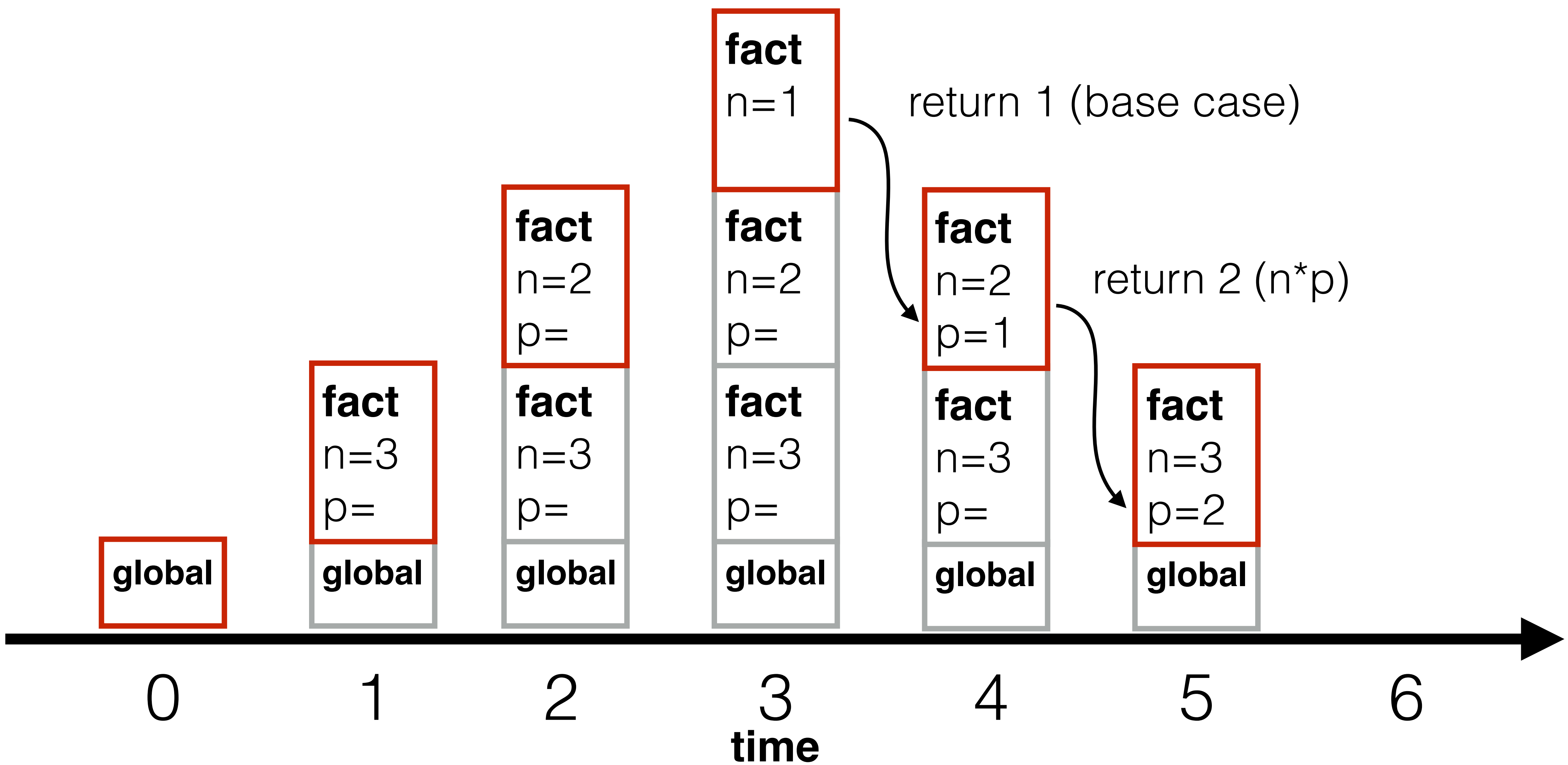
# Deep Dive: Runtime Stack

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



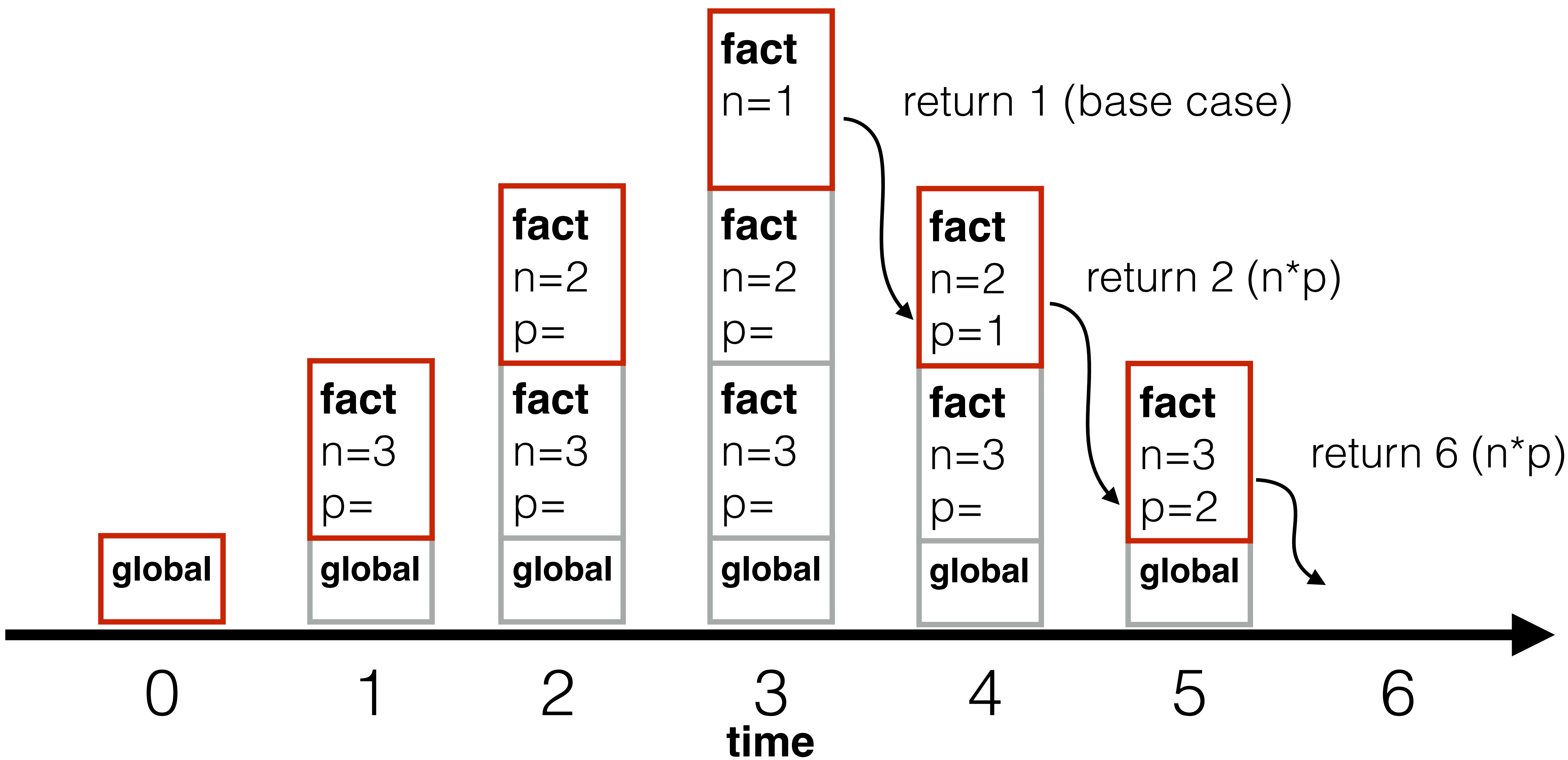
# Deep Dive: Runtime Stack

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



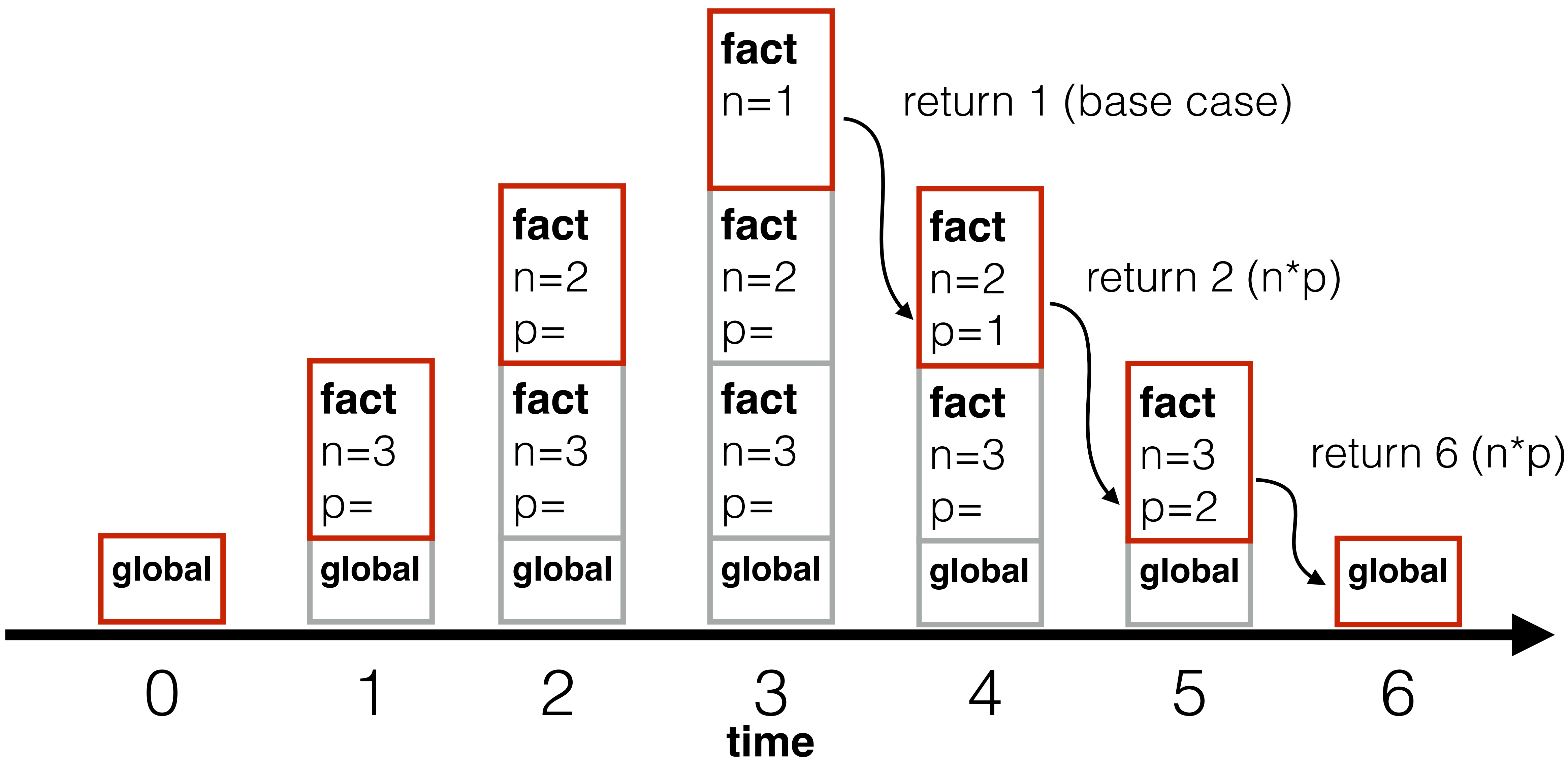
# Deep Dive: Runtime Stack

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



# Deep Dive: Runtime Stack

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



# "Infinite" Recursion Bugs

What happens if:

- 
- 

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

# “Infinite” Recursion Bugs

What happens if:

- we forgot the “n == 1” check?

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

# “Infinite” Recursion Bugs

What happens if:

- we forgot the “n == 1” check?
- factorial is called with a negative number?

-1

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

# “Infinite” Recursion Bugs

What happens if:

- we forgot the “n == 1” check?
- factorial is called with a negative number?

-1

```
def fact(n):
    ~~if n == 1:~~
        ~~return 1~~
    p = fact(n-1)
    return n * p
```

never terminates

# “Infinite” Recursion Bugs

What happens if:

- we forgot the “n == 1” check?
- factorial is called with a negative number?

-1

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

never terminates

**fact**
n=2

**fact**
n=3

**global**

# “Infinite” Recursion Bugs

What happens if:

- we forgot the “n == 1” check?
- factorial is called with a negative number?

-1

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

never terminates

| |
|---|
| **fact** n=-1 |
| **fact** n=0 |
| **fact** n=1 |
| **fact** n=2 |
| **fact** n=3 |
| **global** |

# “Infinite” Recursion Bugs

What happens if:

- we forgot the “n == 1” check?
- factorial is called with a negative number?

-1

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

never terminates

StackOverflowError

...

| |
|---|
| **fact** n=-2 |
| **fact** n=-1 |
| **fact** n=0 |
| **fact** n=1 |
| **fact** n=2 |
| **fact** n=3 |
| **global** |

# Coding Demos

# Demo 1: Pretty Print

Goal: format nested lists of bullet points

**Input**:
- The recursive lists

**Output**:
- Appropriately-tabbed items

**Example**:

```
>>> pretty_print([“A”, [“1”, “2”, “3”,],
                  “B”, [“4”, [“i”, “ii”]]])
*A
  *1
  *2
  *3
*B
  *4
    *i
    *ii
```

# Demo 2: Recursive List Search

Goal: does a given number exist in a recursive structure?

**Input**:

- A number
- A list of numbers and lists (which contain other numbers and lists)

**Output**:

- True if there's a list containing the number, else False

**Example**:

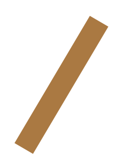
```
>>> contains(3, [1,2,[4,[[3],[8,9]],5,6]])
True
>>> contains(12, [1,2,[4,[[3],[8,9]],5,6]])
False
```

# Conclusion: Review Learning Objectives

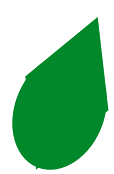
# Learning Objectives: Recursive Information

**What is a recursive definition/structure?**

- Definition contains term
- Structure refers to others of same type
- Example: a dictionary contains dictionaries (which may contain...)



recursive case



base case

# Learning Objectives: Recursive Code

**What is recursive code?**

- Function that sometimes itself (maybe indirectly)

**Why write recursive code?**

- Real-world data/structures are recursive; intuitive for code to reflect data

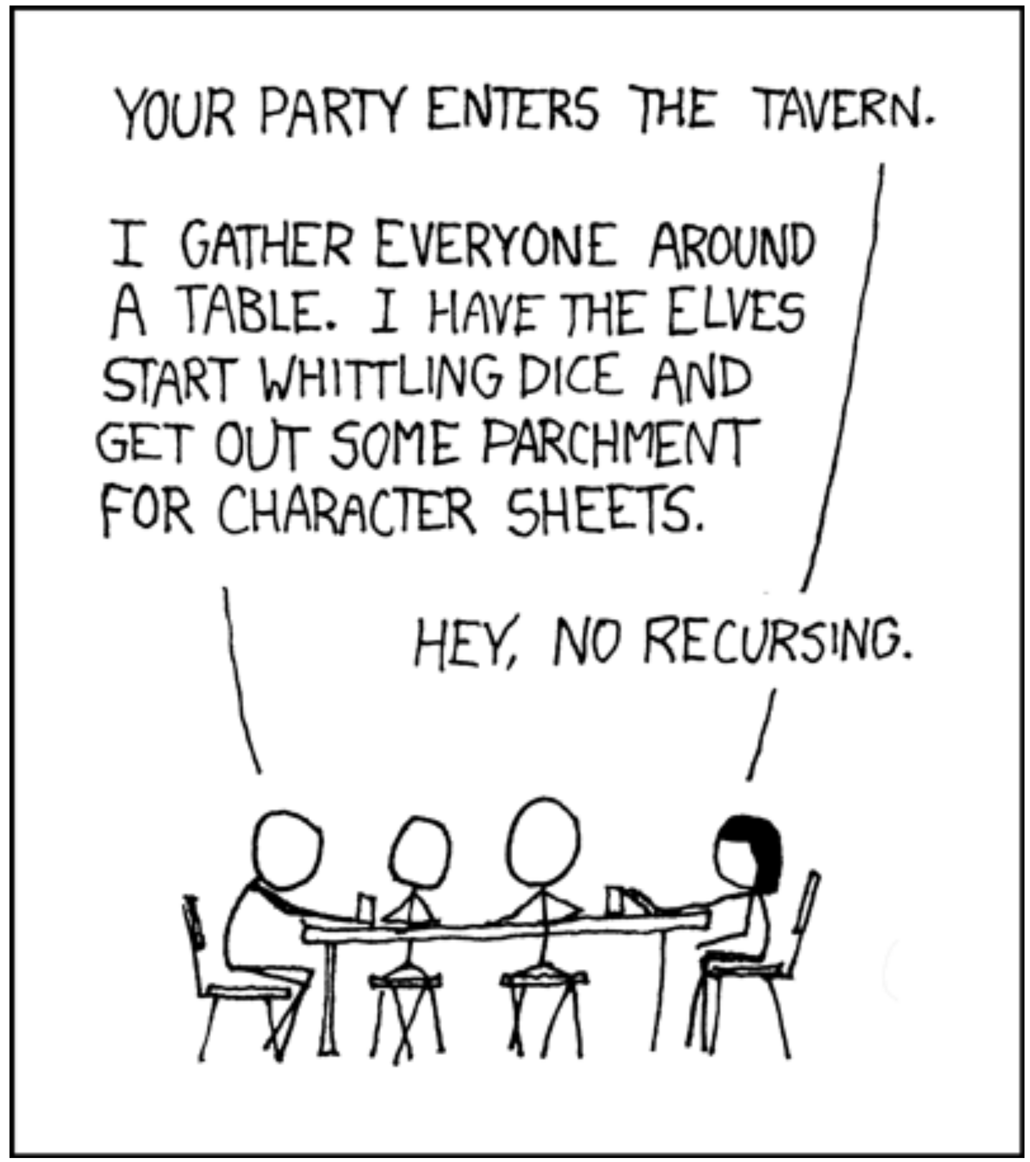
**Where do computers keep local variables for recursive calls?**

- In a section of memory called a “frame”
- Only one function is **active** at a time, so keep frames in a stack

**What happens to programs with infinite recursion?**

- Calls keep pushing more frames
- Exhaust memory, throw **StackOverflowError**

# Questions?




https://xkcd.com/244/