220 / 319: Recursion

The Art of Self Reference

220 / 319: Recursion

The Art of Self Reference

220 / 319: Recursion

The Art of Self Reference

220 / 319: Recursion

The Art of Self Reference

https://en.wikipedia.org/

Department of Computer Sciences
University of Wisconsin-Madison

https://en.wikipedia.org/

Department of Computer Sciences
University of Wisconsin-Madison

https://en.wikipedia.org/

Department of Computer Sciences
University of Wisconsin-Madison

Department of Computer Sciences
University of Wisconsin-Madison

https://en.wikipedia.org/

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 assignments!

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

Learning Objectives

Define recursion and be able to identify

- base case
- recursive case
- infinite recursion

Explain why data structures lists and dicts can be recursively defined

- What is recursive code?

Trace a recursive function

- involving numeric computation
- involving nested data structure

Write a recursive function that processes a nested list

Read *Think Python*

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

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

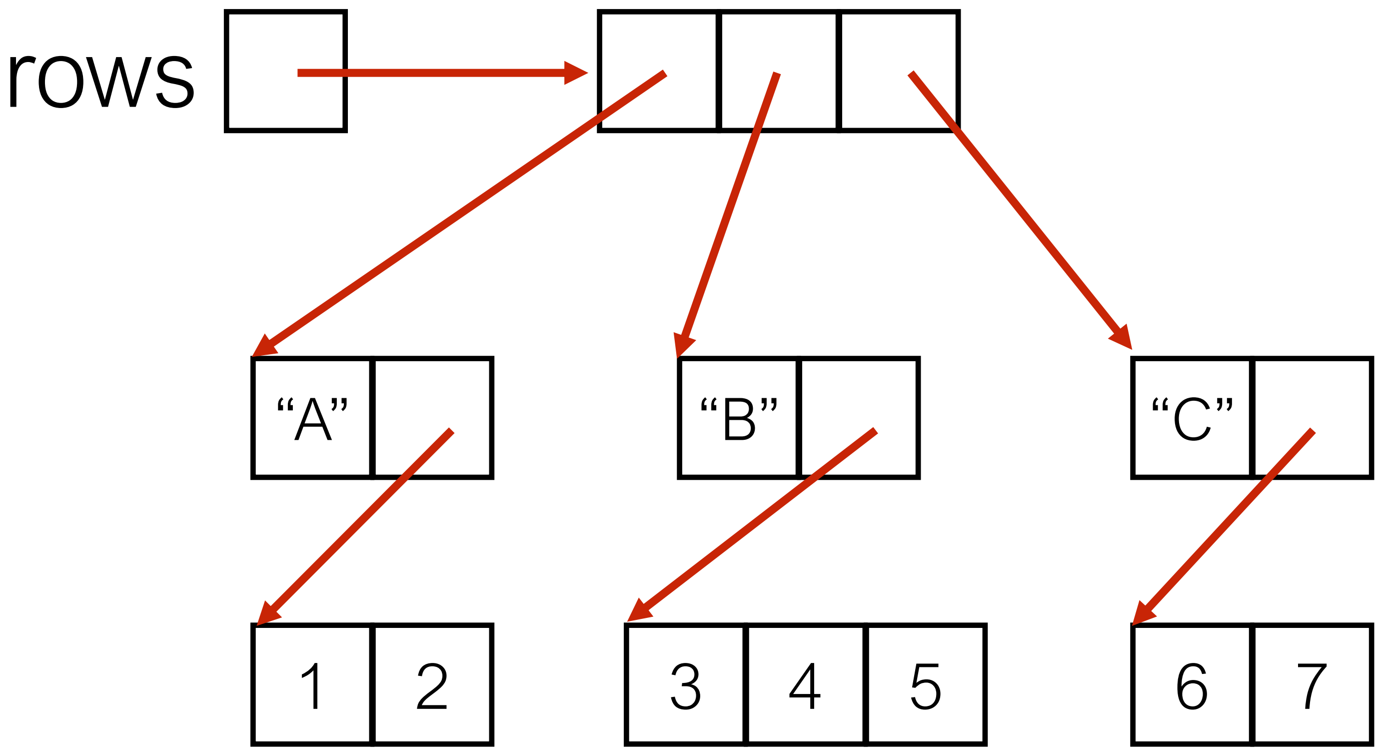
- x equals another positive even number plus two

What is Recursion?

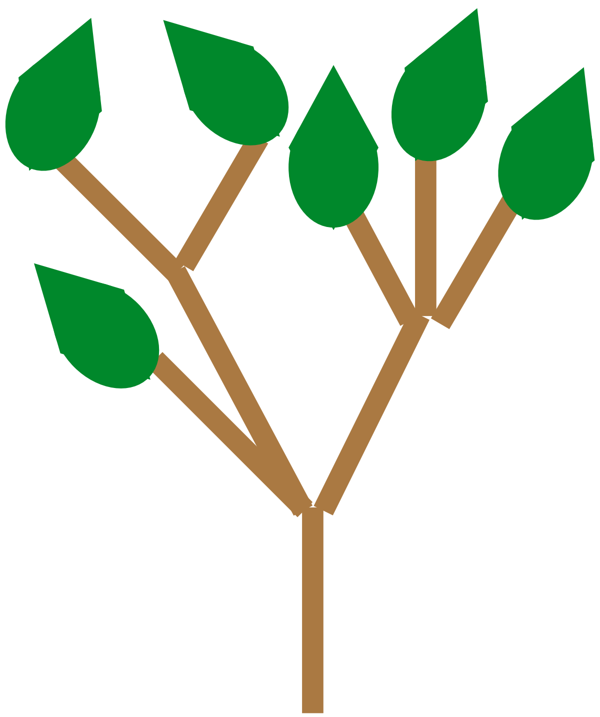
Recursive structures may refer to structures of the same type

- data structures or real-world structures

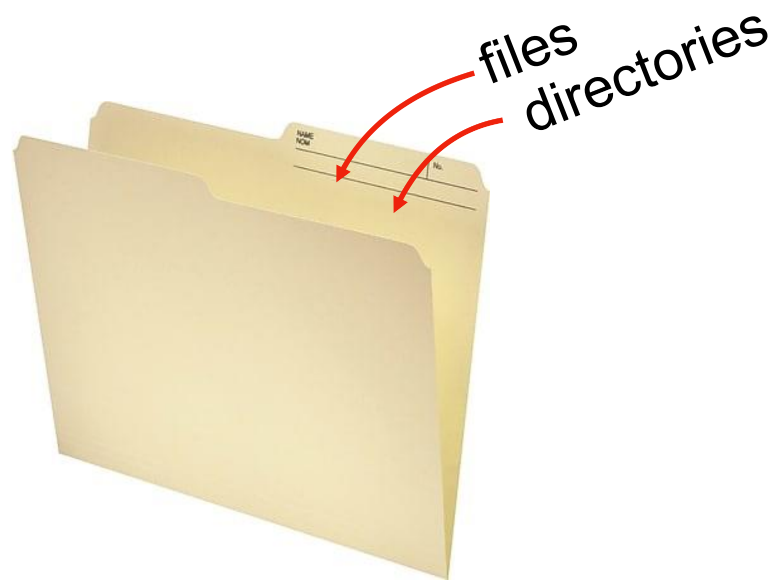
```
rows = [
   ["A", [1, 2]],
   ["B", [3, 4, 5]],
   ["C", [6, 7]]
]
```



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 (Direct Recursion)

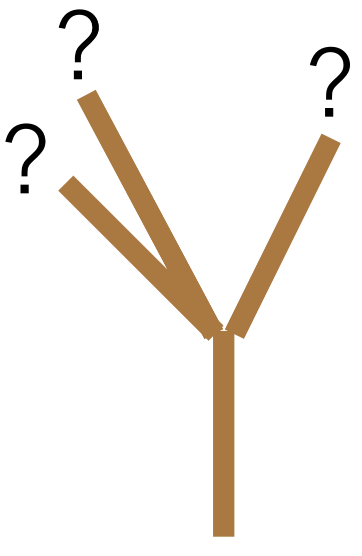
Term: branch

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

Example: Trees (Direct Recursion)

Term: branch

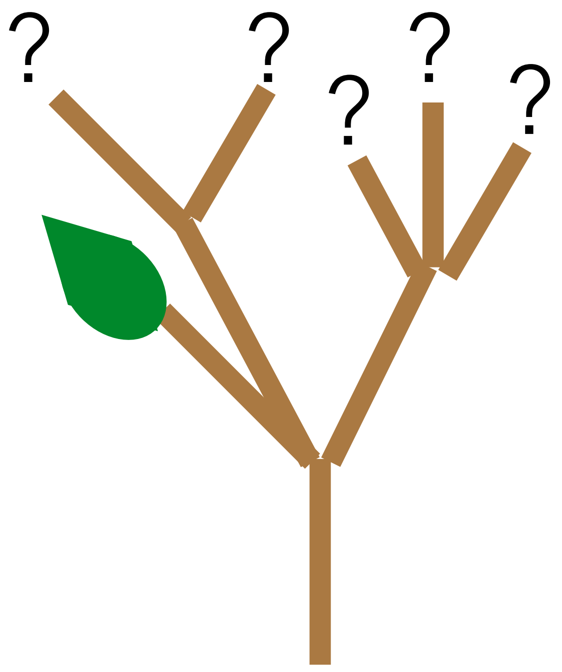
Definition: wooden stick, with an end splitting into other branches, OR terminating with a leaf



Example: Trees (Direct Recursion)

Term: branch

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



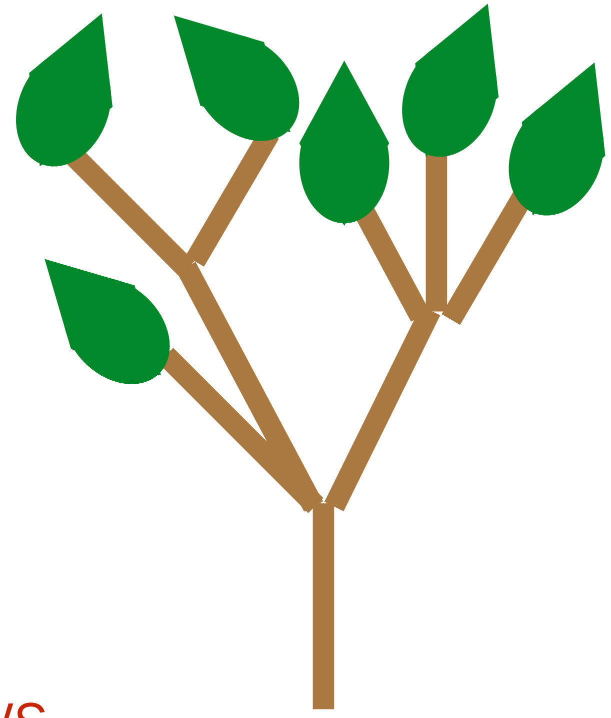
Example: Trees (Direct Recursion)

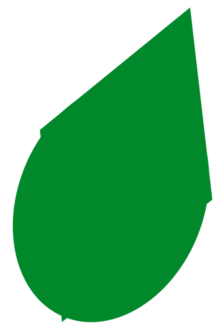
Term: branch

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

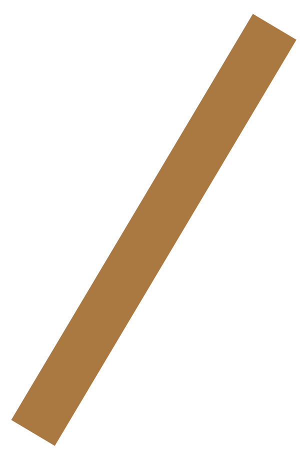
recursive case allows indefinite growth

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





base case (leaf)



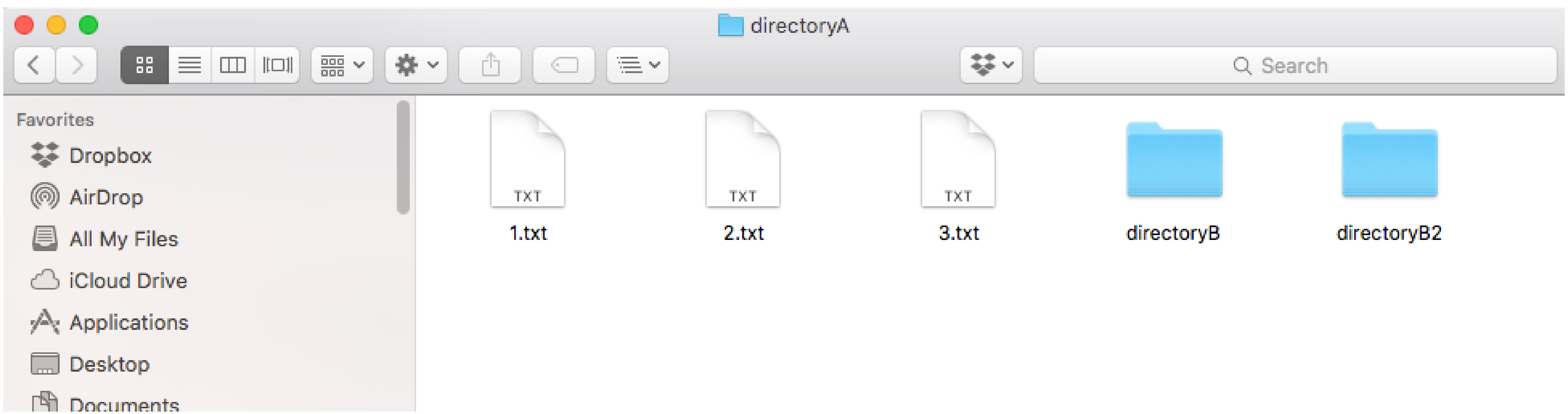
recursive case (branch)

Example: Directories (aka folders)

Term: directory

recursive because def contains term

Definition: a collection of files and directories

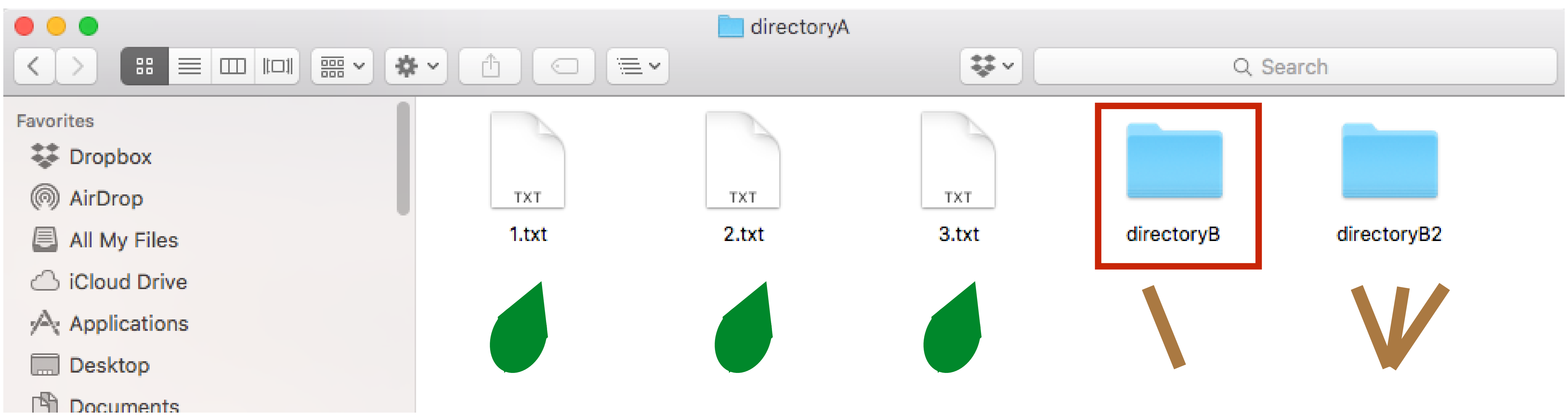


Example: Directories (aka folders)

Term: directory

recursive because def contains term

Definition: a collection of files and directories



file system tree

Example: Directories (aka folders)

Term: directory

recursive because def contains term

Definition: a collection of files and directories

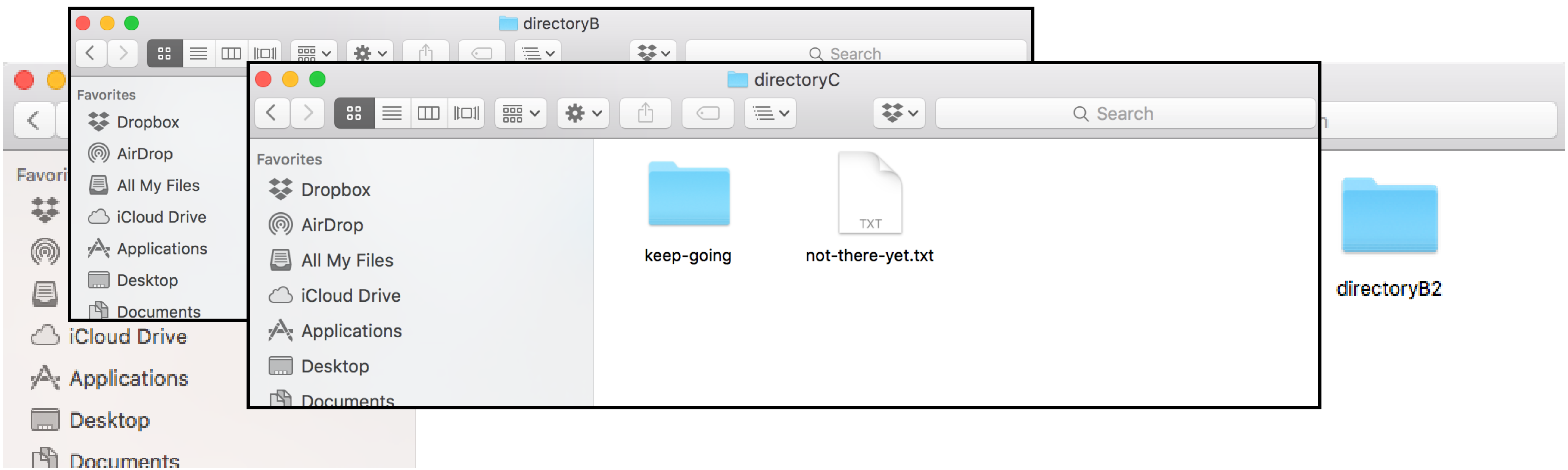
file system tree

Example: Directories (aka folders)

Term: directory

recursive because def contains term

Definition: a collection of files and directories

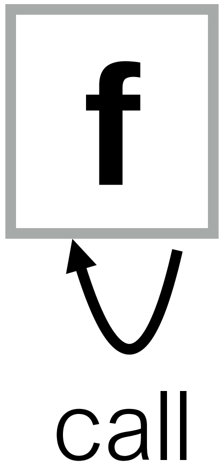


file system tree

Recursive Code

What is it?

- A function that calls itself




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

Recursive Code

What is it?

- A function that calls itself

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

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

Why use recursion?

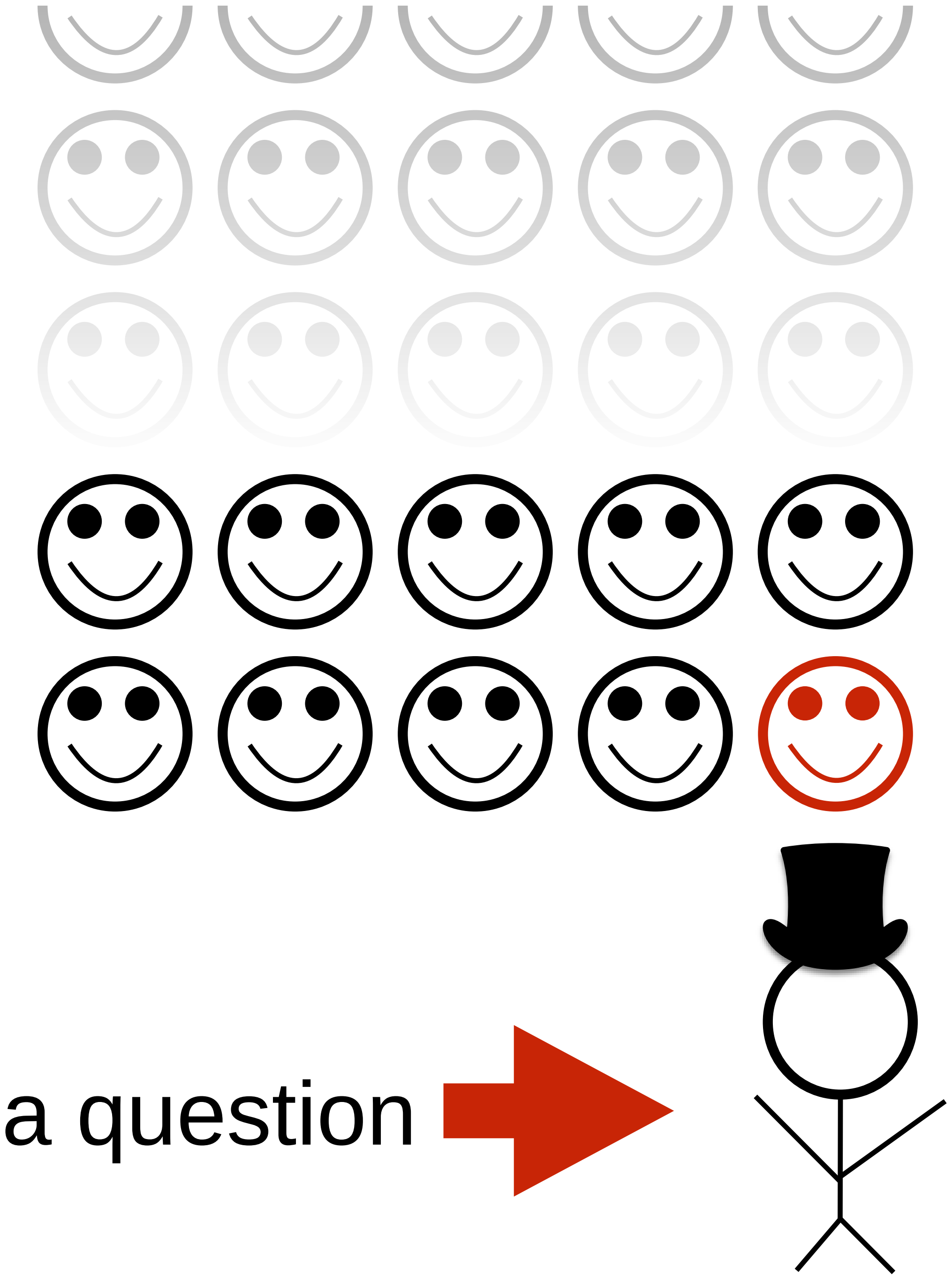
- simple and elegant solution
- recursive code corresponds to recursive data
- reduce a big problem into a smaller problem

https://texastreesurgeons.com/services/tree-removal/

Recursive Student Counting

CS220 students in the front row

Professor with a question



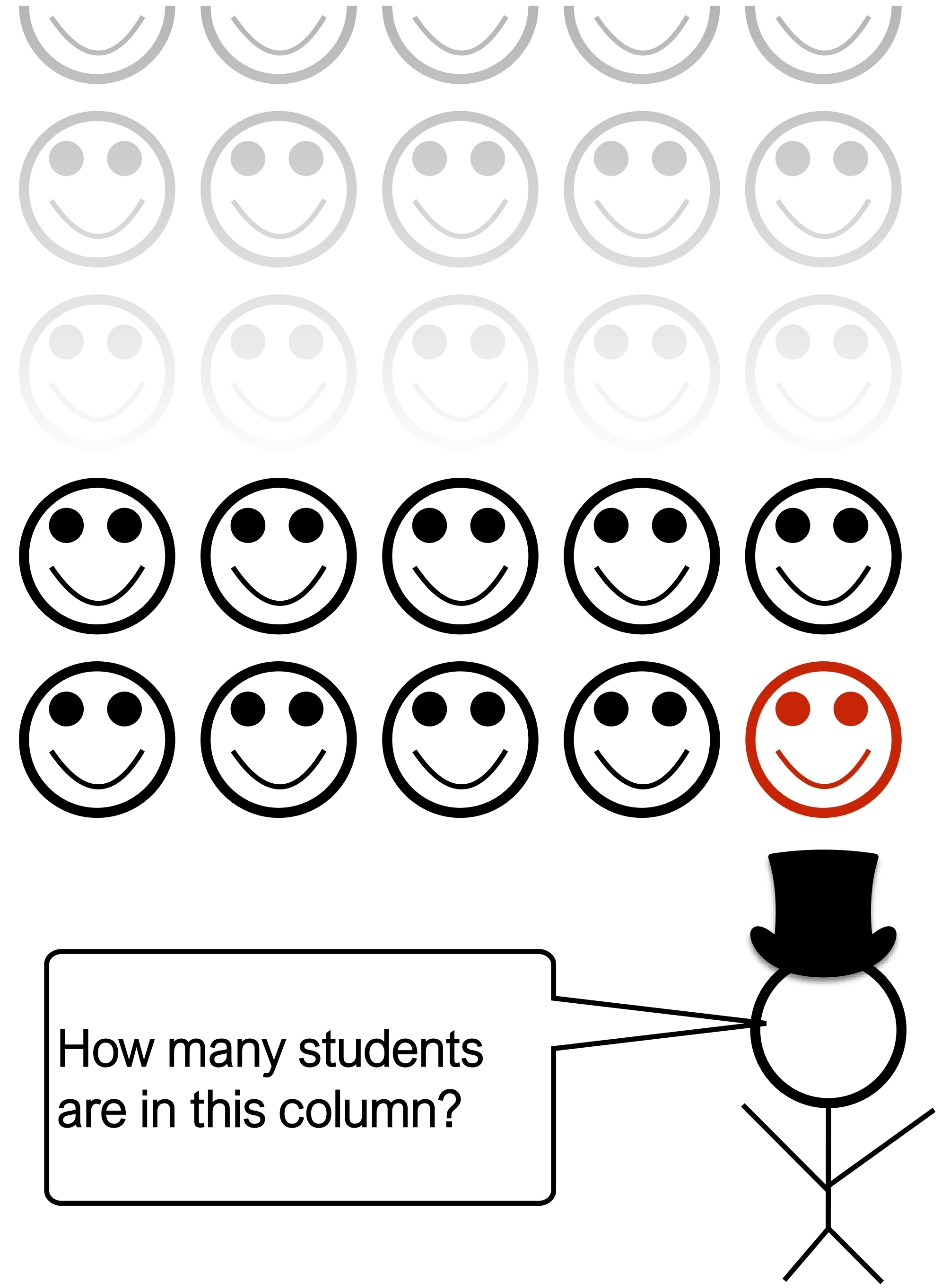
Example from https://courses.cs.washington.edu/courses/cse143/17au/

Recursive Student Counting

Constraints:

- You can only talk to the student behind / in front of you

What should each student ask the person behind them?

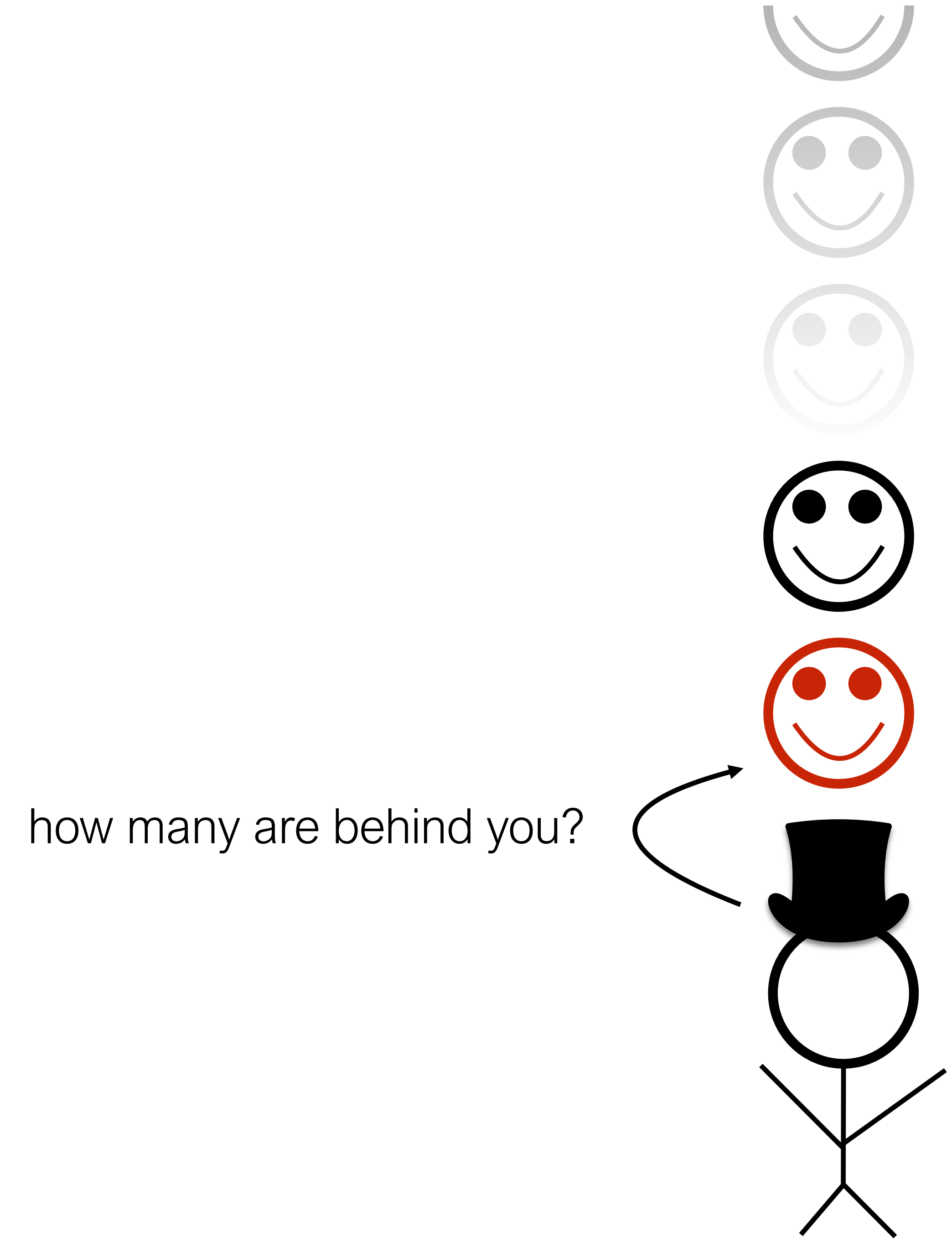


Example from https://courses.cs.washington.edu/courses/cse143/17au/

Recursive Student Counting

Strategy: reframe question as “how many students are behind you?”

Reframing is the hardest part!



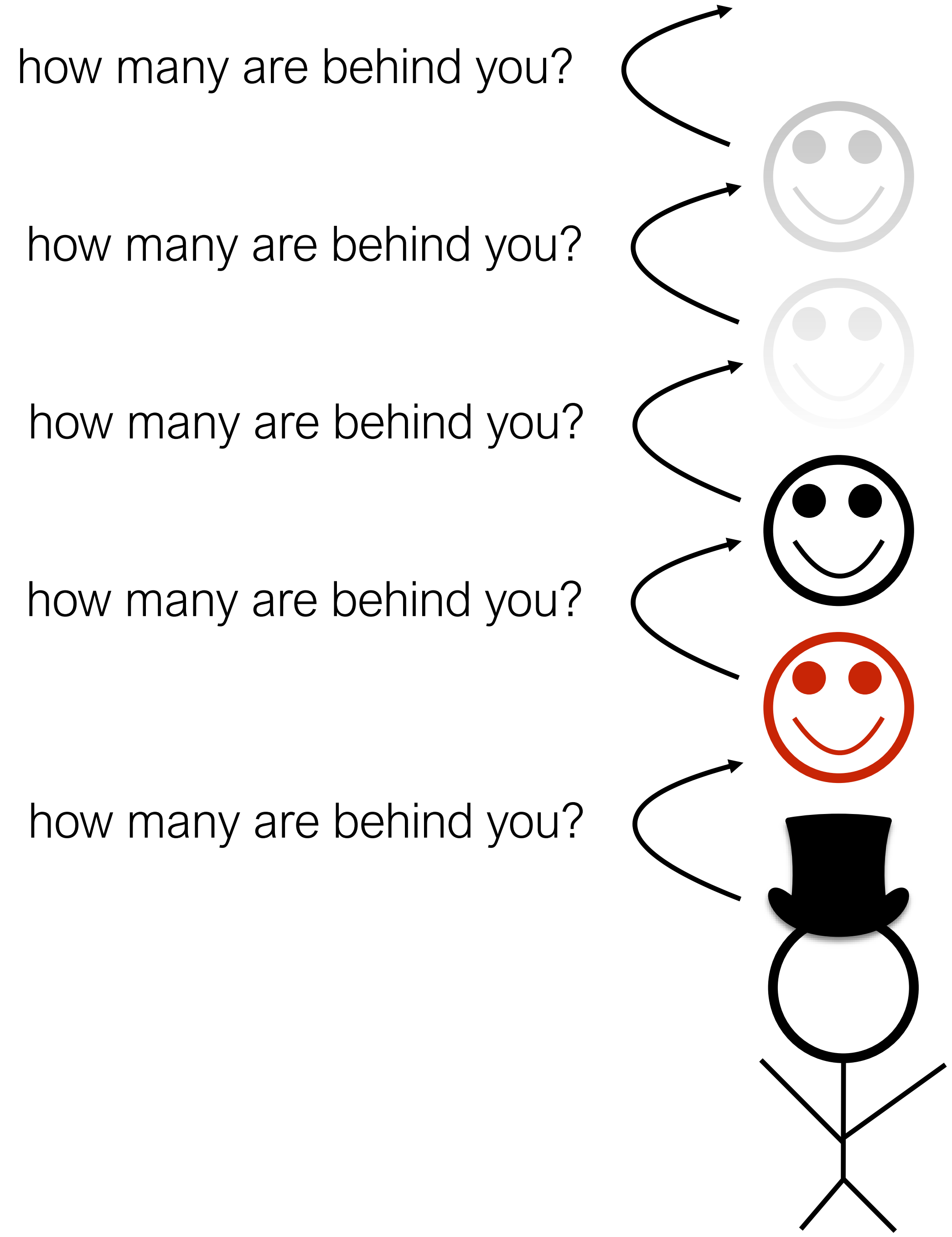
Process:
if nobody is behind you: say 0
else: ask them, say their answer+1

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

Recursive Student Counting

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/

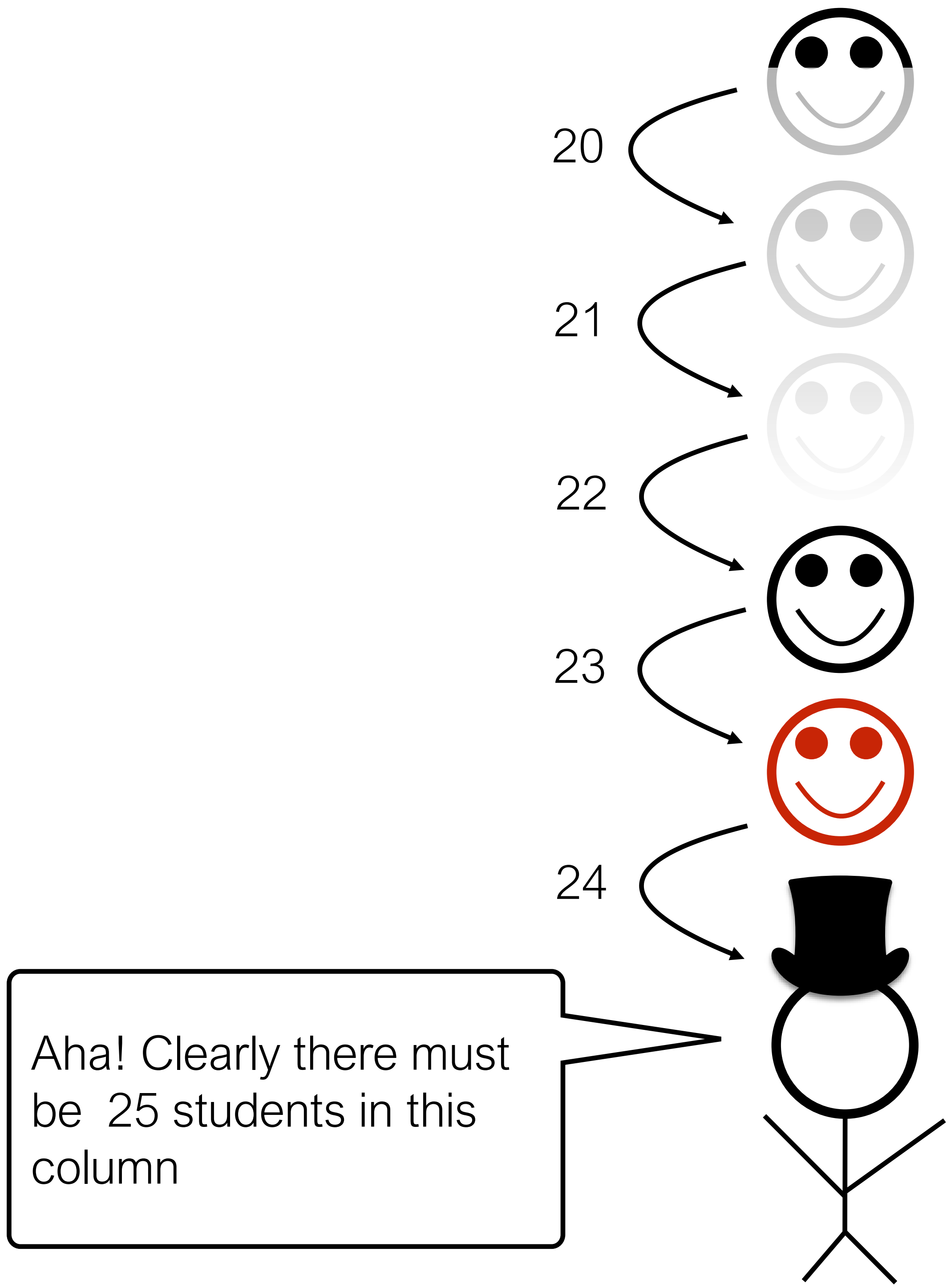
Recursive Student Counting

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/

Practice: 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
```
simplest example

```
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:

```
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:

```
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:

```
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):
    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:

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

Rule 1: Base case should always be defined and be terminal

Rule 2: Recursive case should make progress towards base case

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

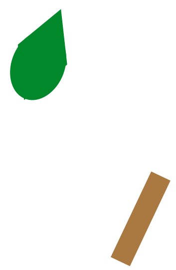
4. Python Code:

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

Let’s “run” it!

Tracing Factorial

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



How does Python keep
all the variables separate?

frames to the rescue!

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

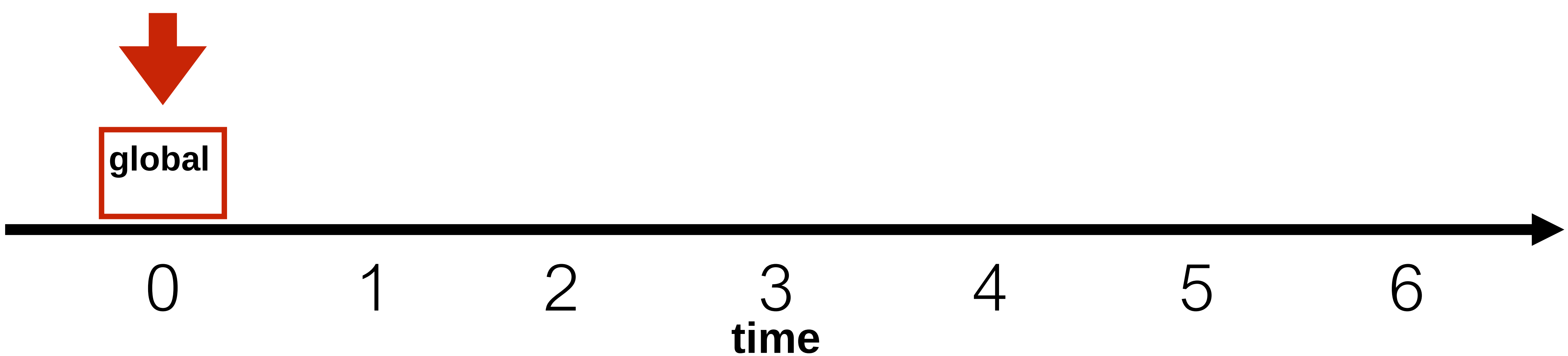
- frames are stored in the stack
- one invocation is active at a time: its frame is on the top of stack
- multiple frames at the same time for the multiple invocations of the same function

Deep Dive: Runtime Stack

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

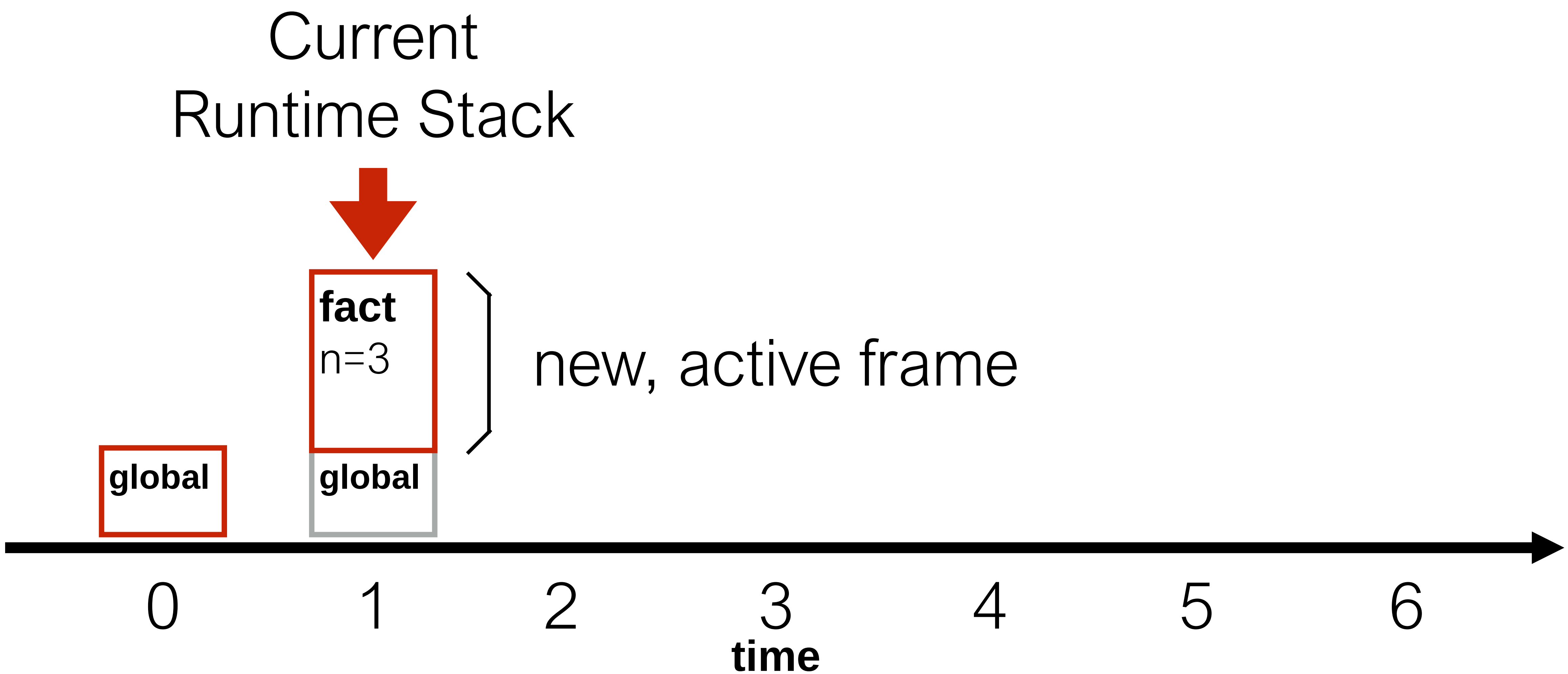
call `fact(3)`

Current Runtime Stack



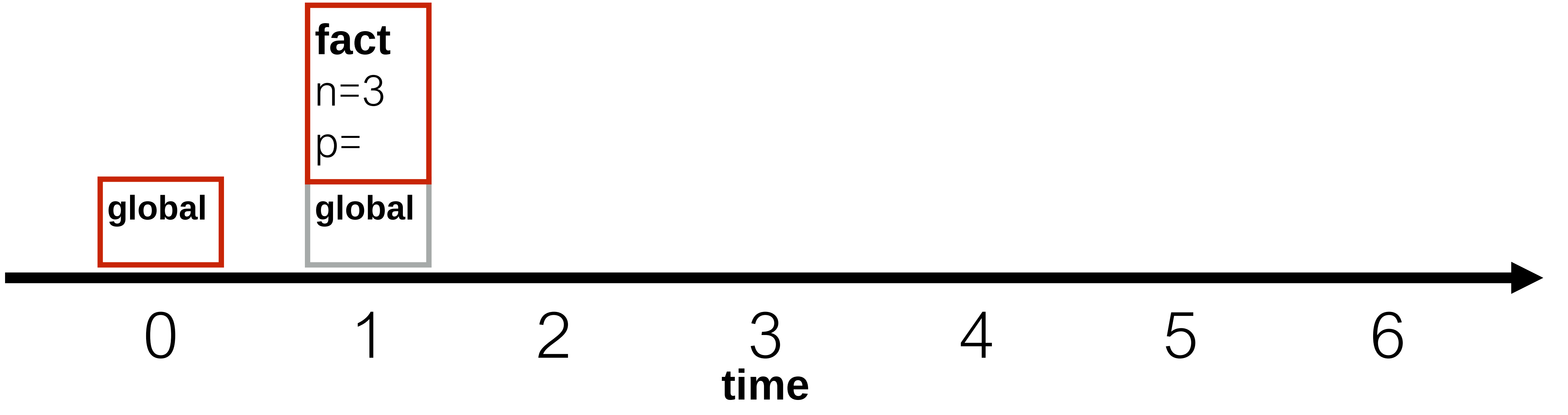
Deep Dive: Runtime Stack

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



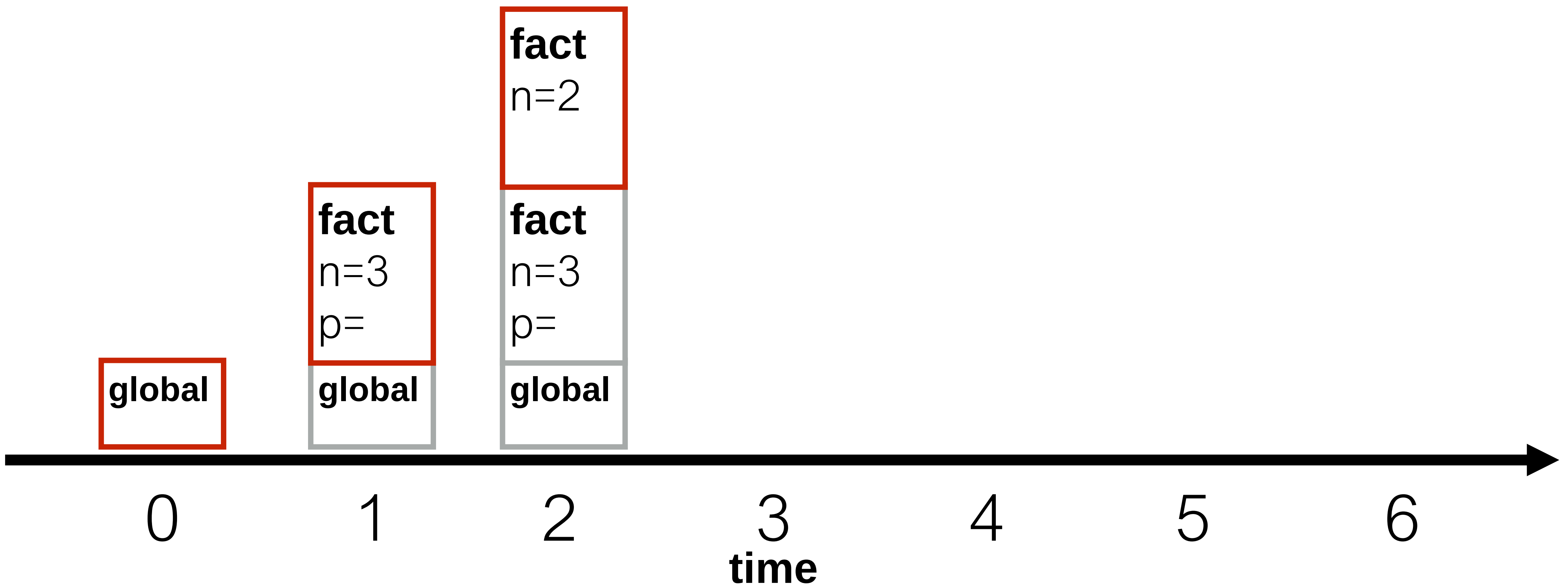
Deep Dive: Runtime Stack

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



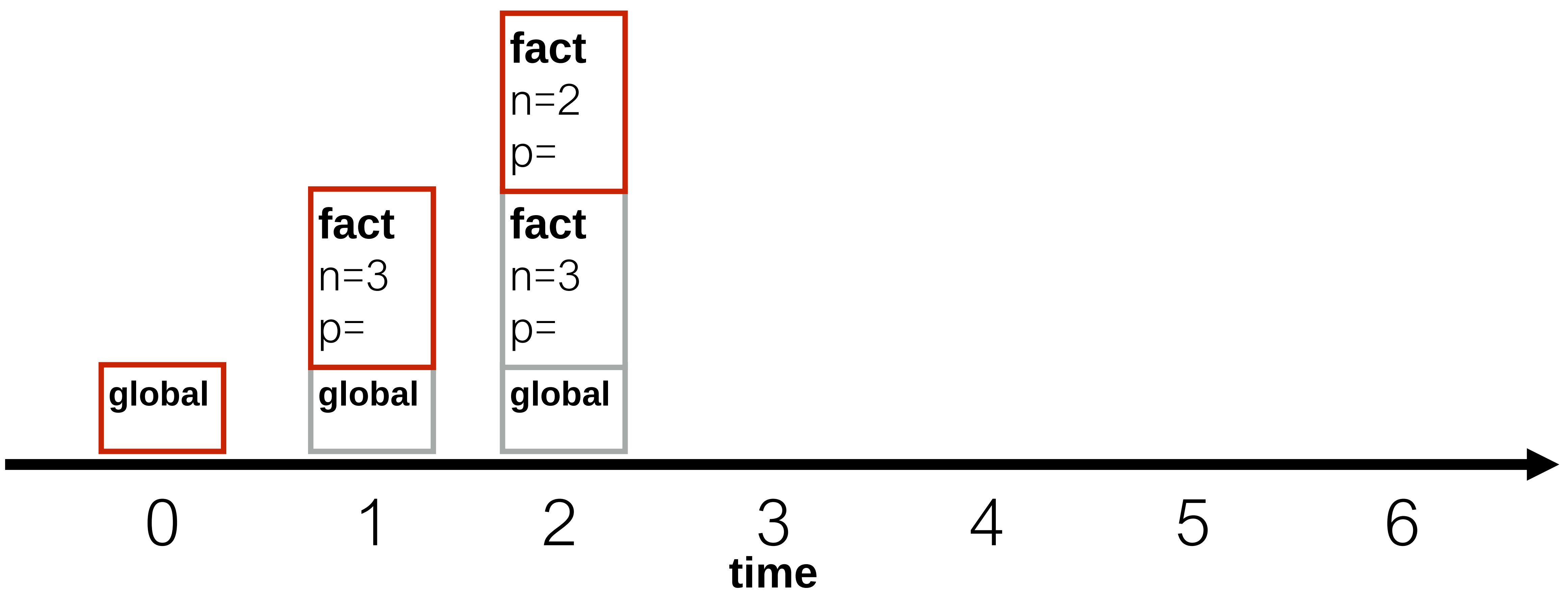
Deep Dive: Runtime Stack

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



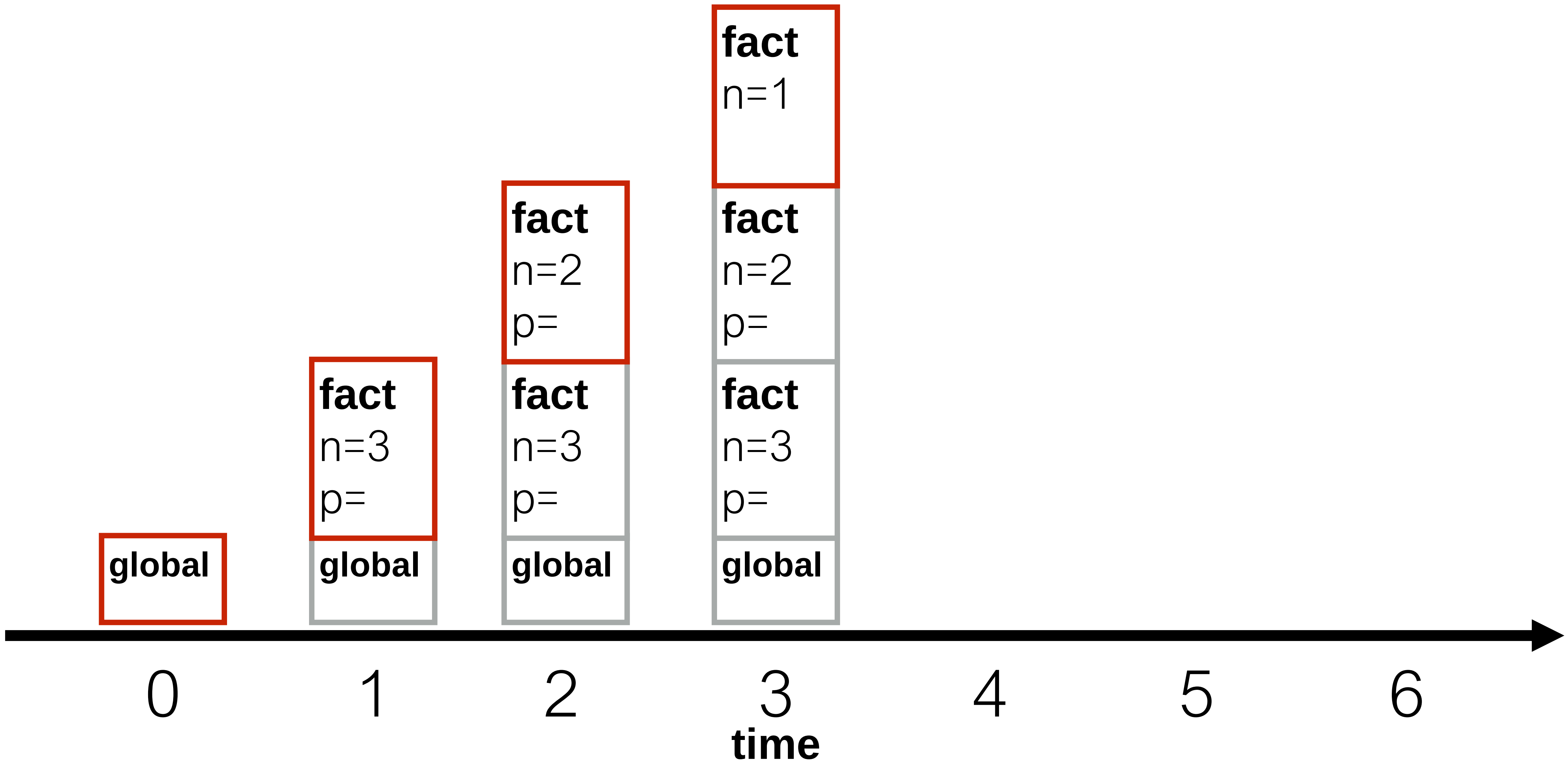
Deep Dive: Runtime Stack

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



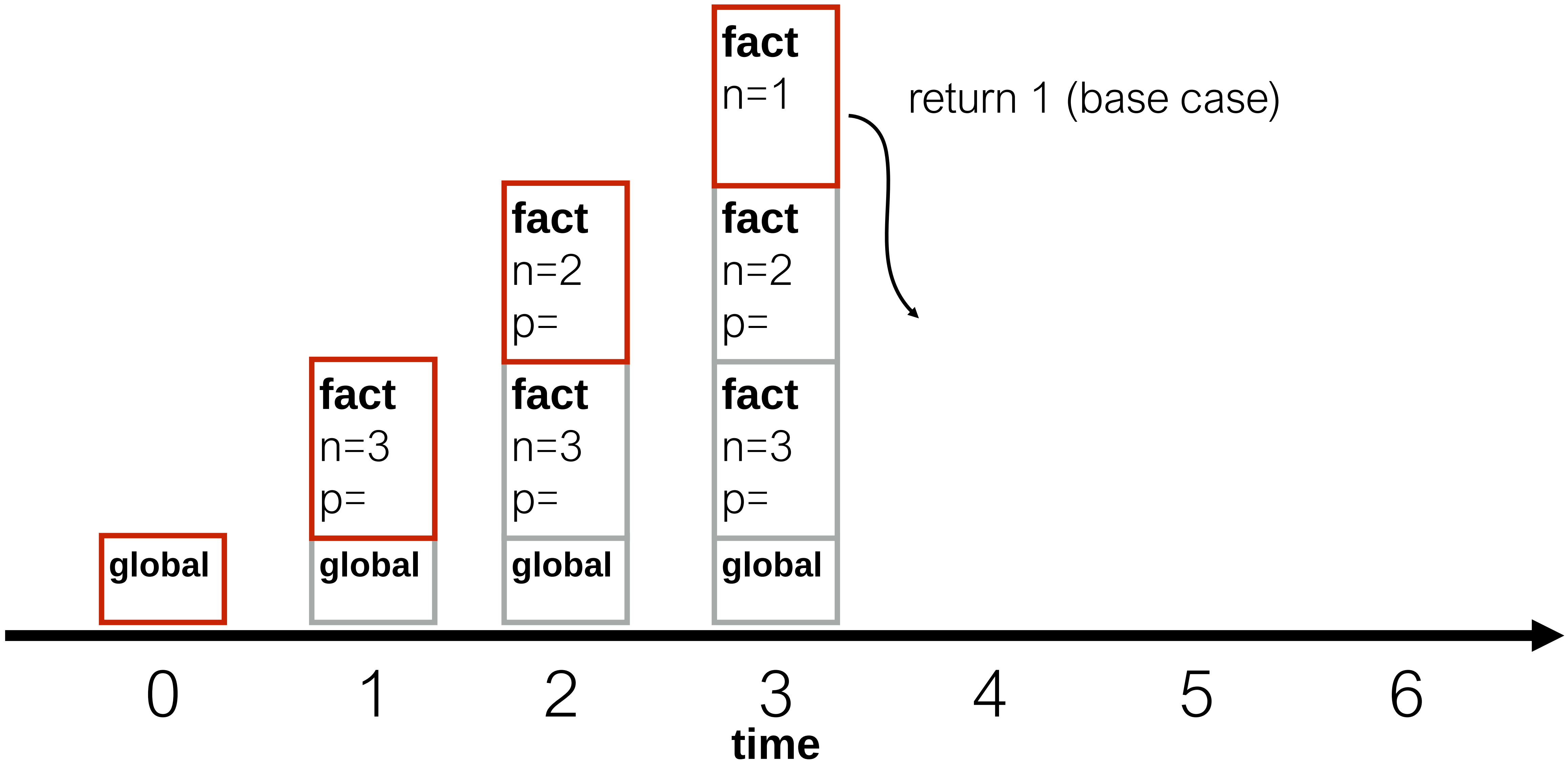
Deep Dive: Runtime Stack

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



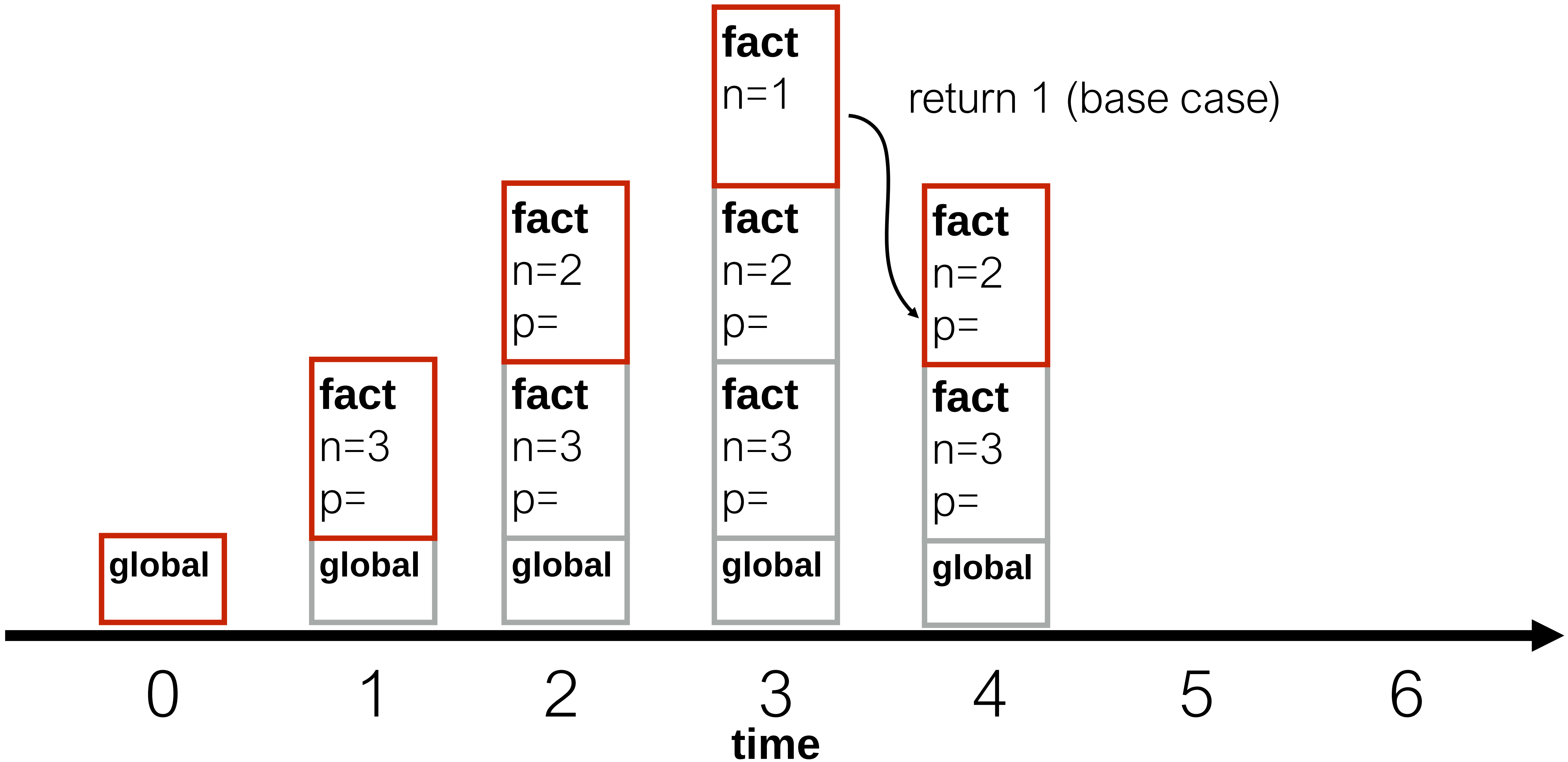
Deep Dive: Runtime Stack

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



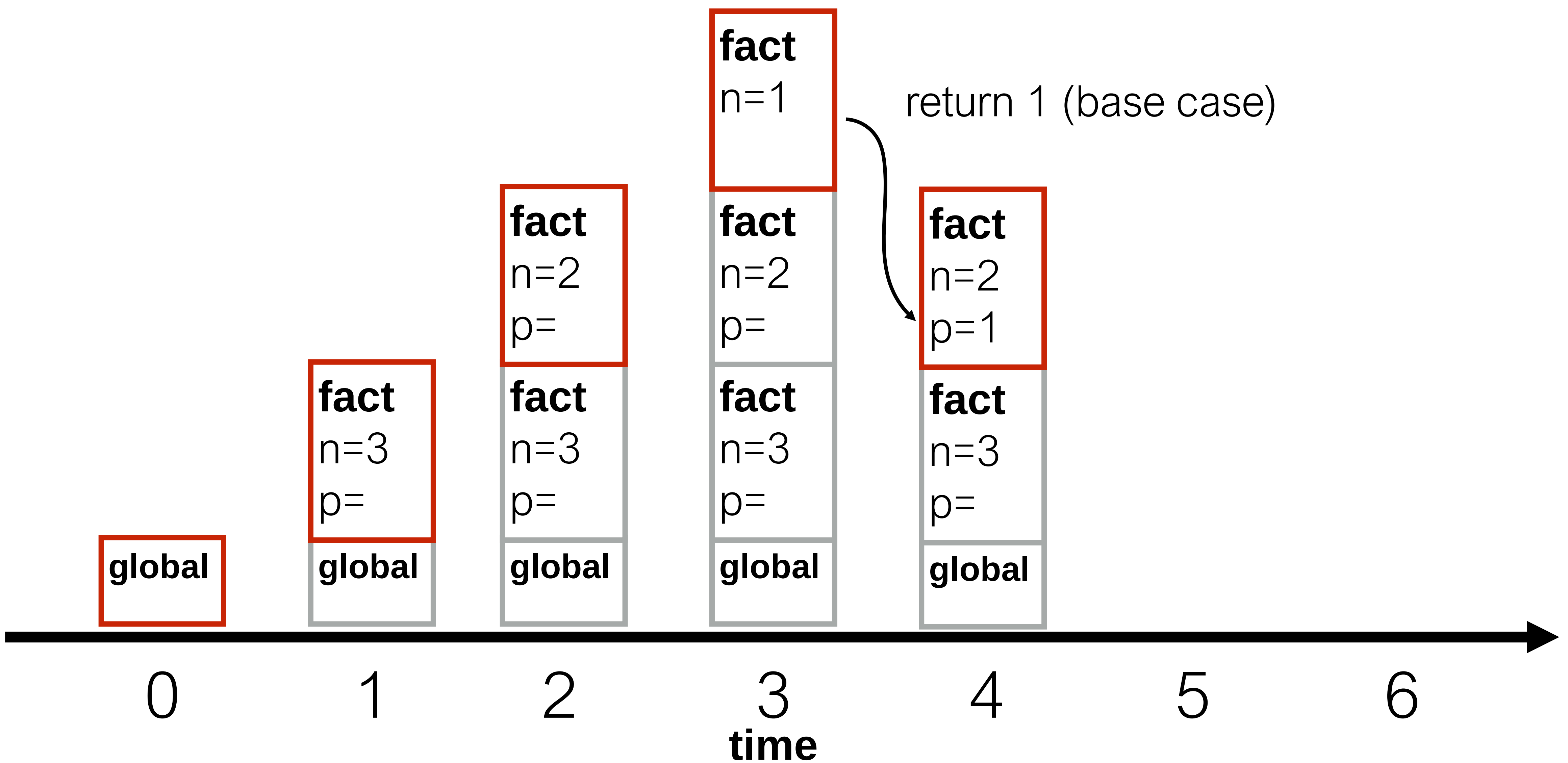
Deep Dive: Runtime Stack

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



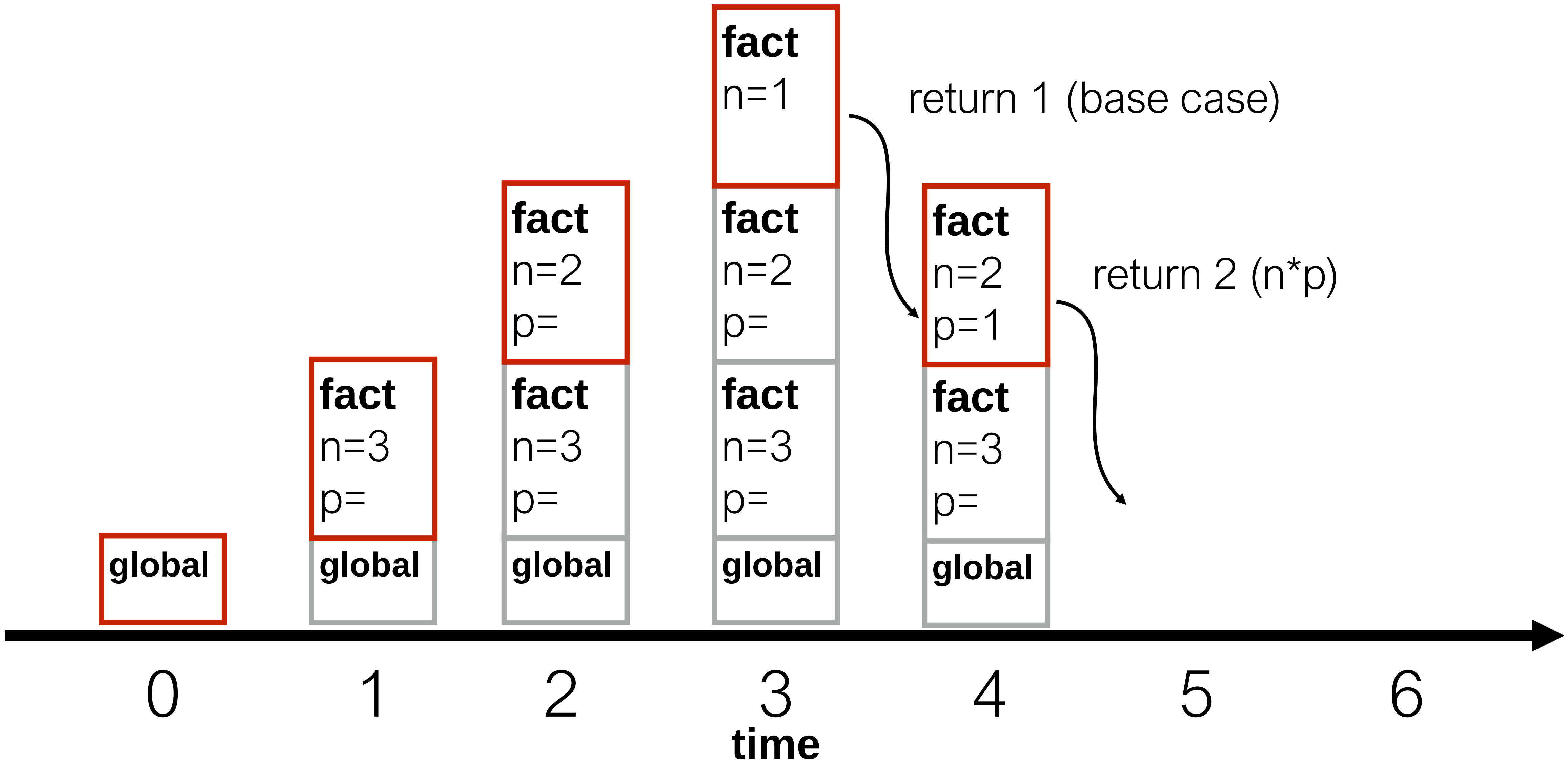
Deep Dive: Runtime Stack

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



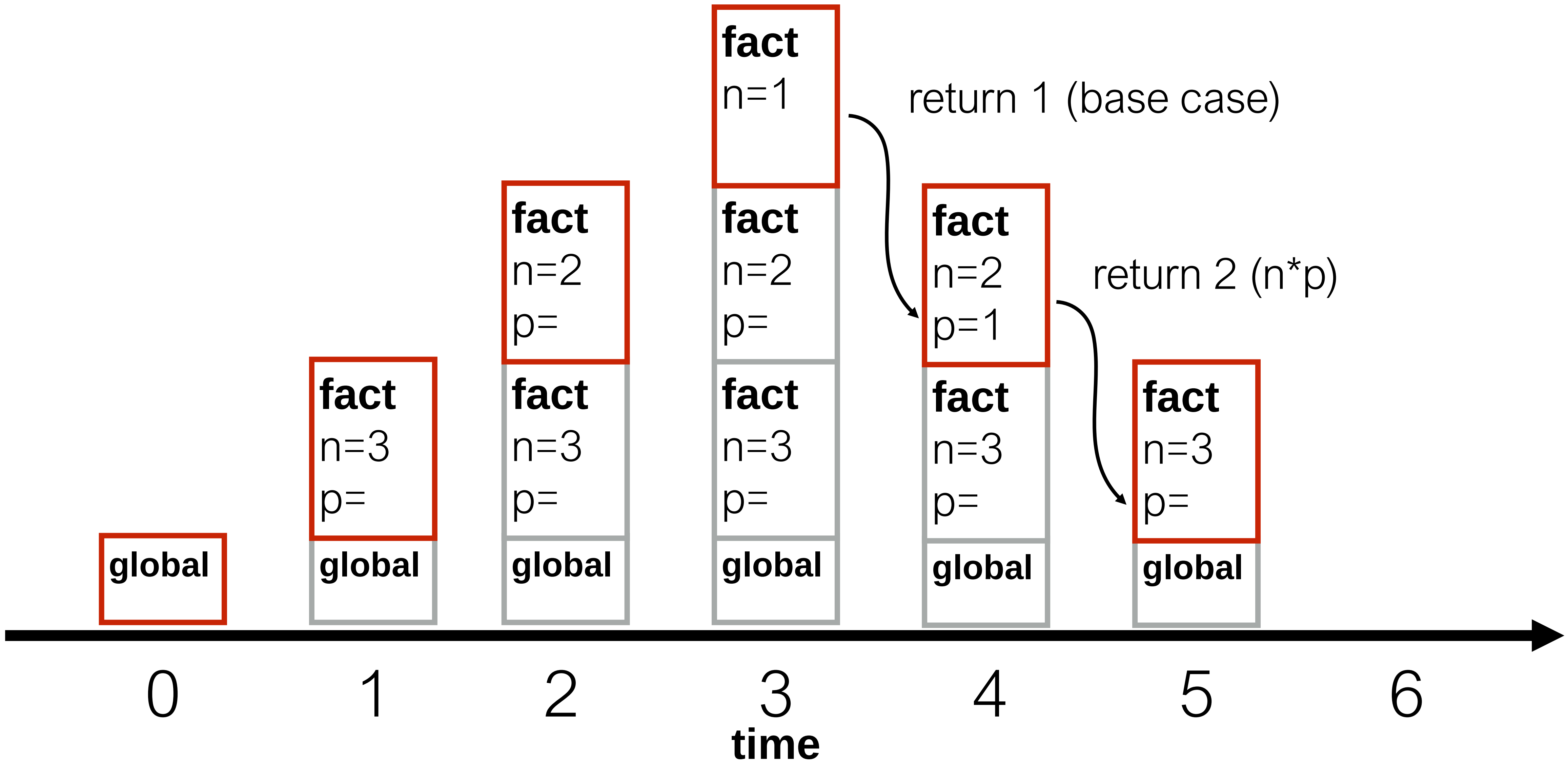
Deep Dive: Runtime Stack

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



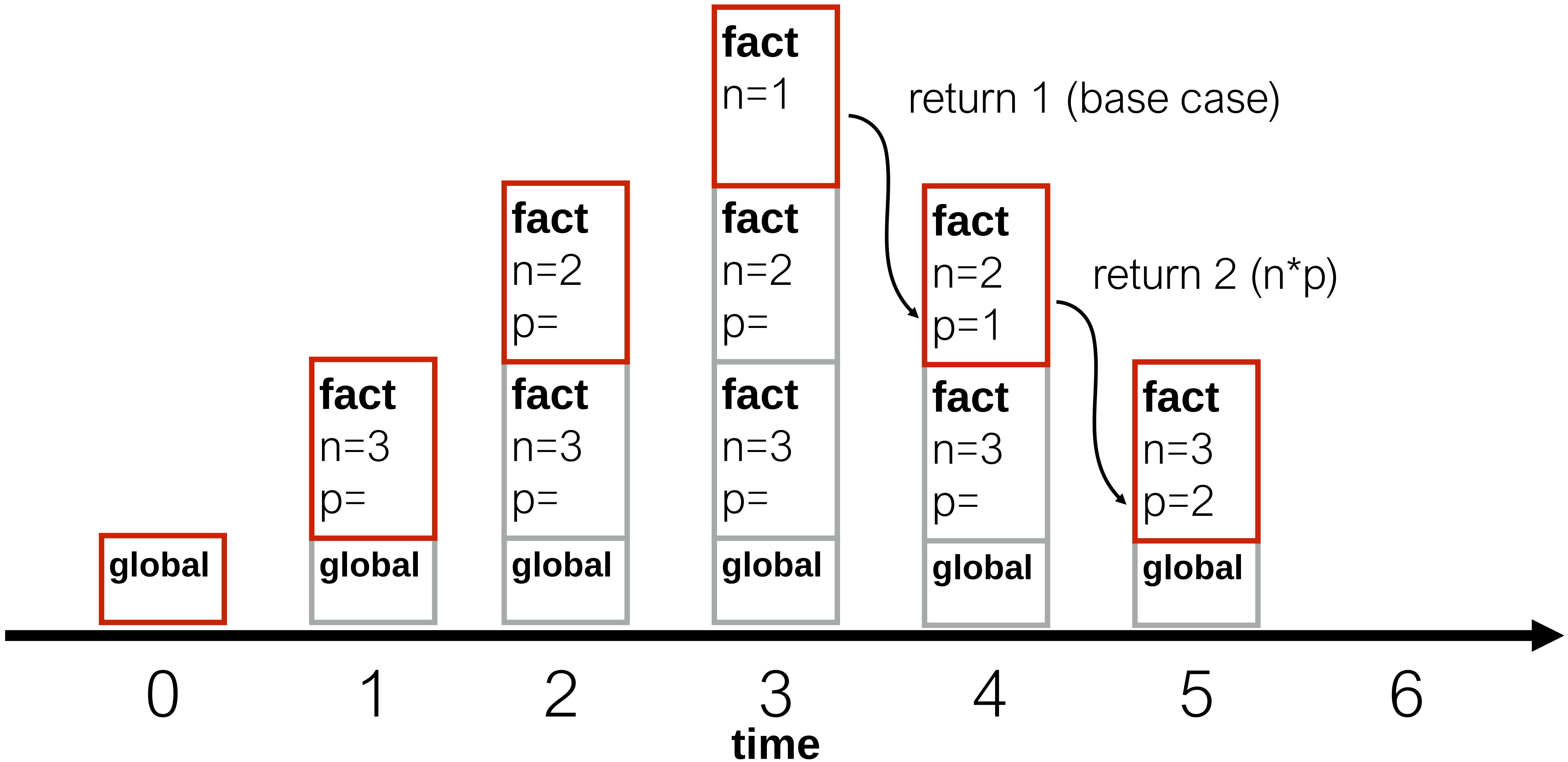
Deep Dive: Runtime Stack

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



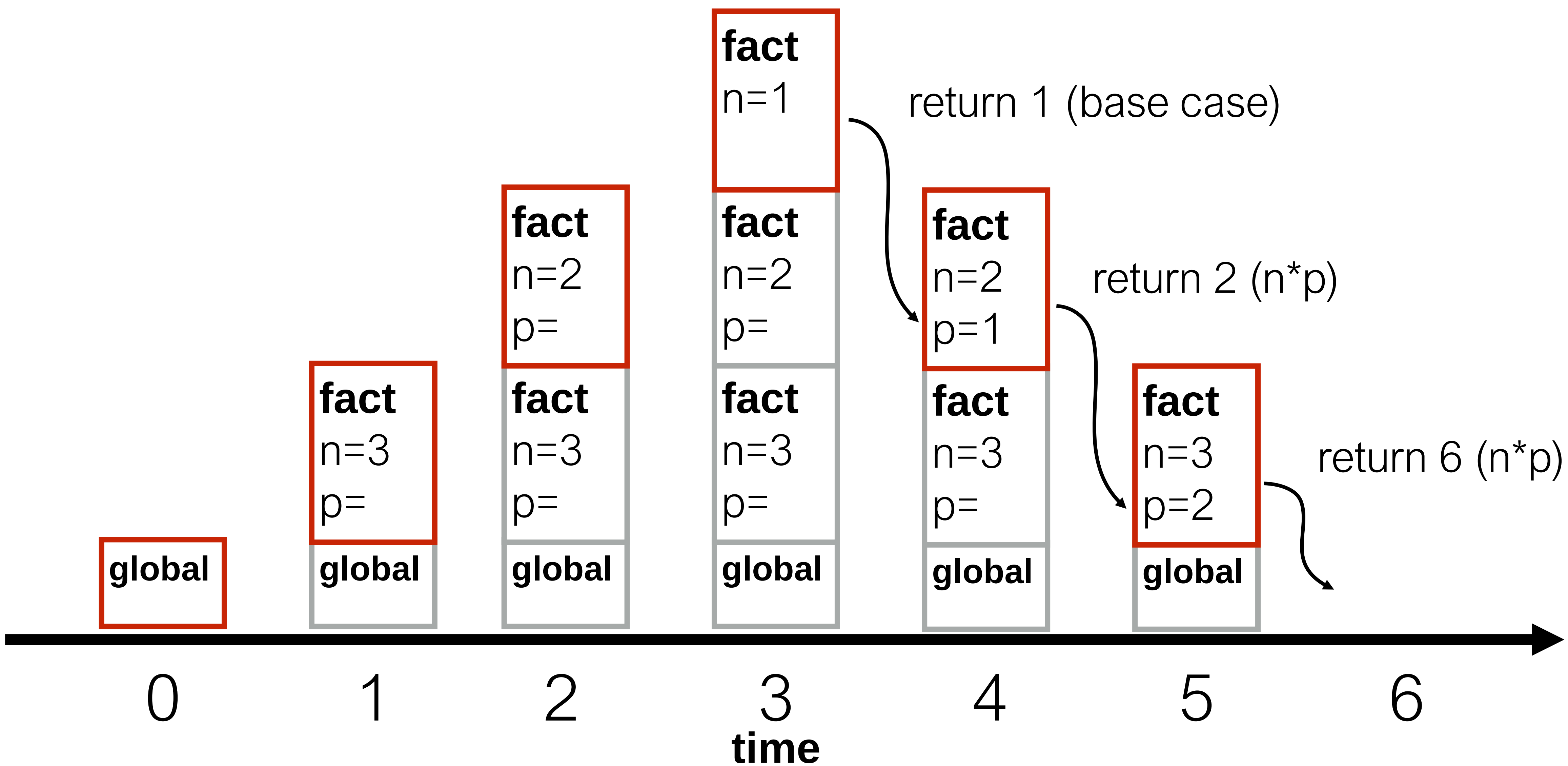
Deep Dive: Runtime Stack

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



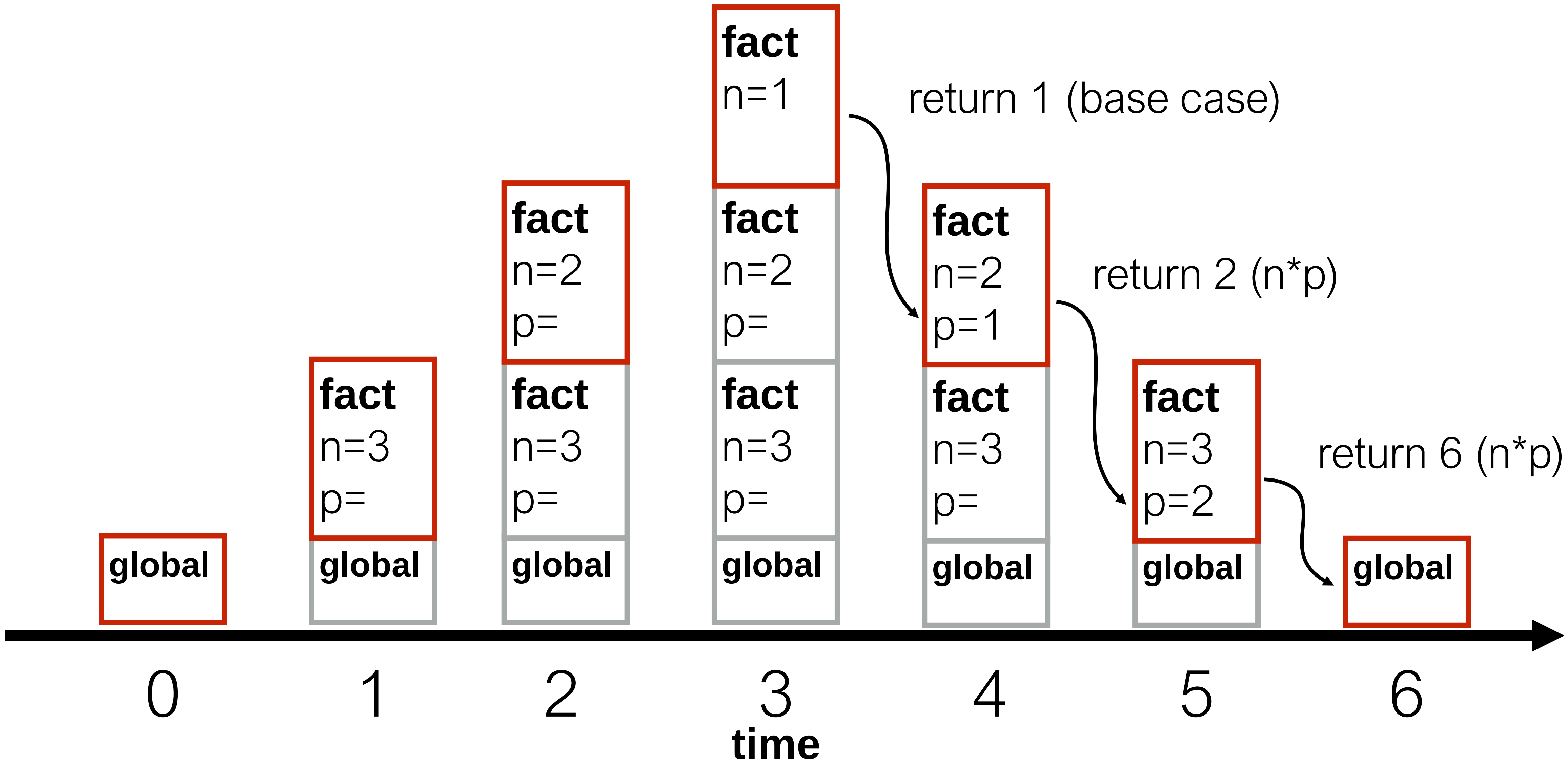
Deep Dive: Runtime Stack

```
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
```



"Infinite" Recursion Bugs

What happens if:

1. 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

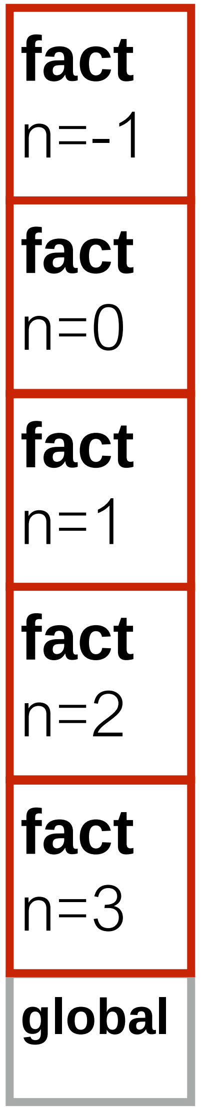
What happens if:

1. factorial is called with a negative number?
2. we forgot the "n == 1" check?

3

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

never terminates



Let’s code

Example: 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
```

Example: 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
```

Practice: Recursive List Search

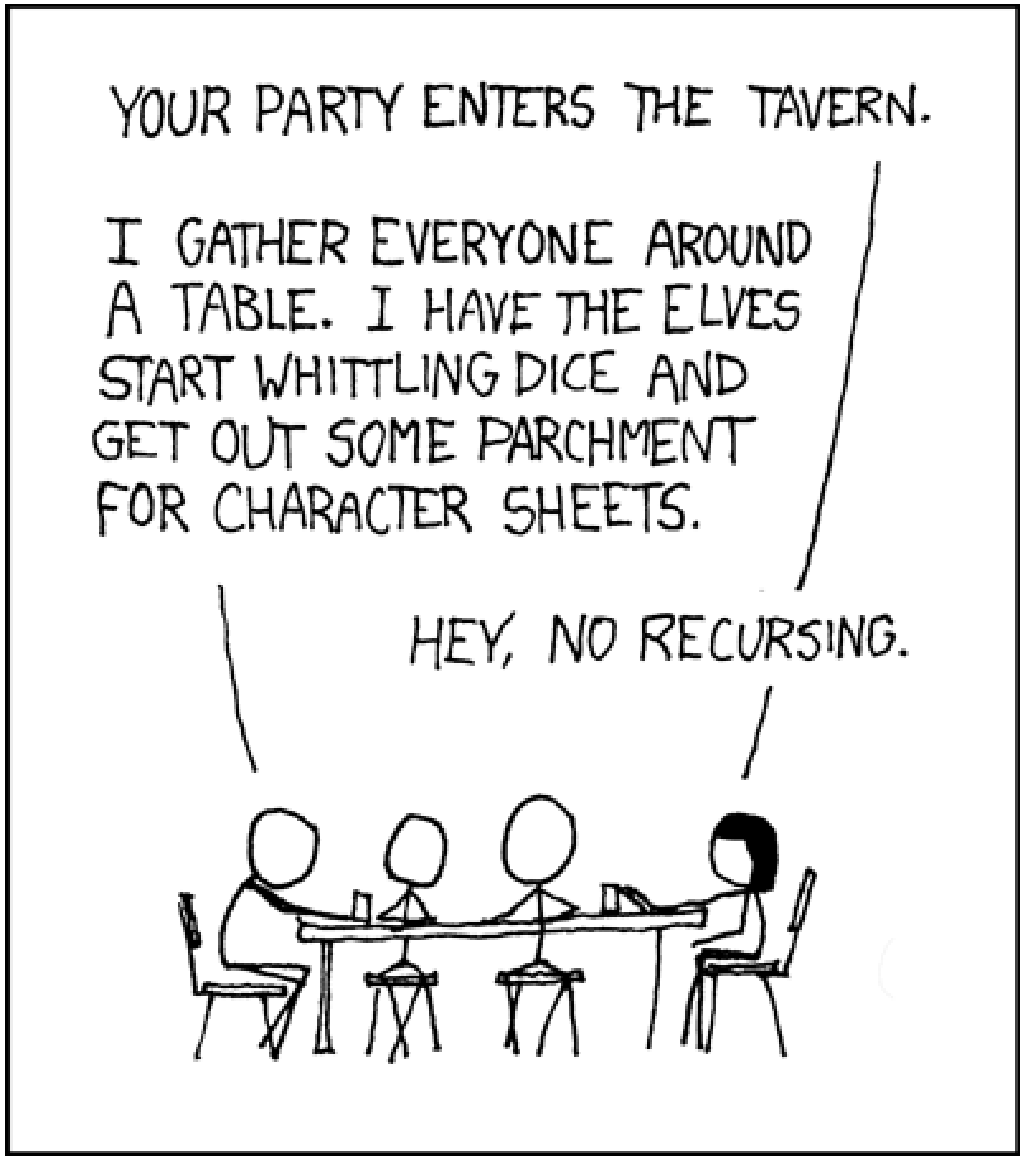
Goal: does a given number exist in a recursive structure?

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
```



https://xkcd.com/244/

"To understand recursion, you need to understand recursion."

(Meena)

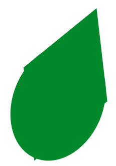
https://hotsigns.net/two-thumbs-up-emoji-247-decal_p_302.html

Summary: 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

Summary: Recursive Code

What is recursive code?

- Function that sometimes itself

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 RecursionError