CSCI 145 · Introduction to Computing · Fall 2026

Recursion with Pending Operations

Lecture 11 — work performed after the recursive call

Count your row

How many people are in your row? One sheet per row:

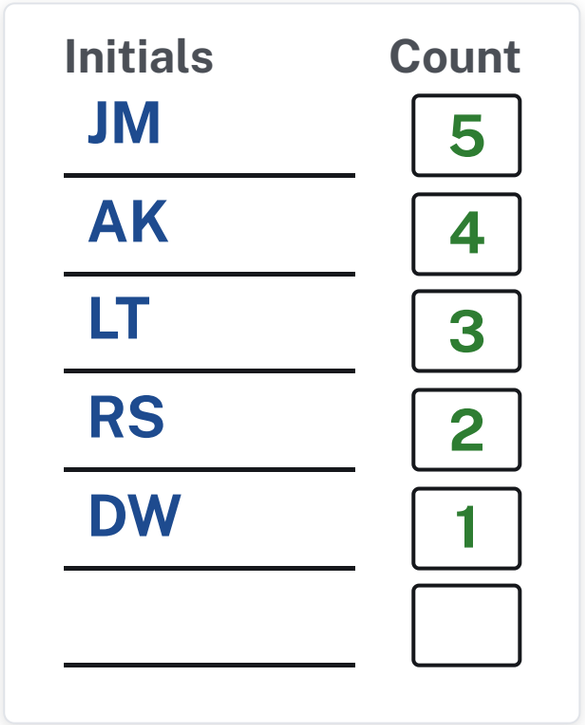
1. The person at the right end of the row starts with the sheet
2. On the way left: write your initials on the next empty line,
   then pass the sheet left
3. No one on your left? Write 1 in your box and pass the sheet back right
4. On the way right: in your box, write the value below plus 1,
   then pass the sheet right
5. Back at the right end: does the top box match the number of initials?

← Passing left is making a call

→ Passing right is returning an answer

The sheet is a call stack

On the way right



- Each person going left is a new call
  Each line is a frame
- An amber box is waiting
  for the answer from below
- No one on the left: the **base case**
  Write 1
- The value below plus 1
  waits for the answer to come back:
  a **pending operation**
- The top box is the answer: 5 people

Outline

- Work that waits
- Pending operations with lists
- The turtle comes home
- Two recursive calls

Part 1 of 4

Work that waits

Predict the output

```
def go_back(n):
    if n == 0:
        print("Stop")
    else:
        print("Go", n)
        go_back(n - 1)
        print("Back", n)

go_back(3)
```

Something new: a line after the recursive call

On paper: what does `go_back(3)` print?

Tracing `go_back(3)`

```
def go_back(n):
    if n == 0:
        print("Stop")
    else:
        print("Go", n)
        go_back(n - 1)
        print("Back", n)

go_back(3)
```

OUTPUT

```
Go 3
Go 2
Go 1
Stop
Back 1
Back 2
Back 3
```

CALL STACK

`go_back`	`n: 3` waiting to print Back 3
`go_back`	`n: 2` waiting to print Back 2
`go_back`	`n: 1` waiting to print Back 1
`go_back`	`n: 0` base case

- An amber frame is waiting
  It still has work to do
- When a call finishes, the frame above picks up where it left off

Pending operations

```
def go_back(n):
    if n == 0:
        print("Stop")
    else:
        print("Go", n)
        go_back(n - 1)
        print("Back", n)
```

Base case | Recursive case | Pending

- A **pending operation** waits until the recursive call returns
- Work before the call happens on the way down
- Work after the call happens on the way back up

Factorial: `4! = 4 * 3 * 2 * 1`

```
def factorial(n):
    """
    If n is 1 or less, returns 1
    Otherwise, returns n * (n - 1) * ... * 1
    """
```

```
>>> factorial(4)
24
>>> factorial(1)
1
```

What is the base case?

If `n <= 1`, return 1

What is the recursive case?

`4! = 4 * 3!`

Return `n * factorial(n - 1)`



pending operation

Tracing `factorial(4)`

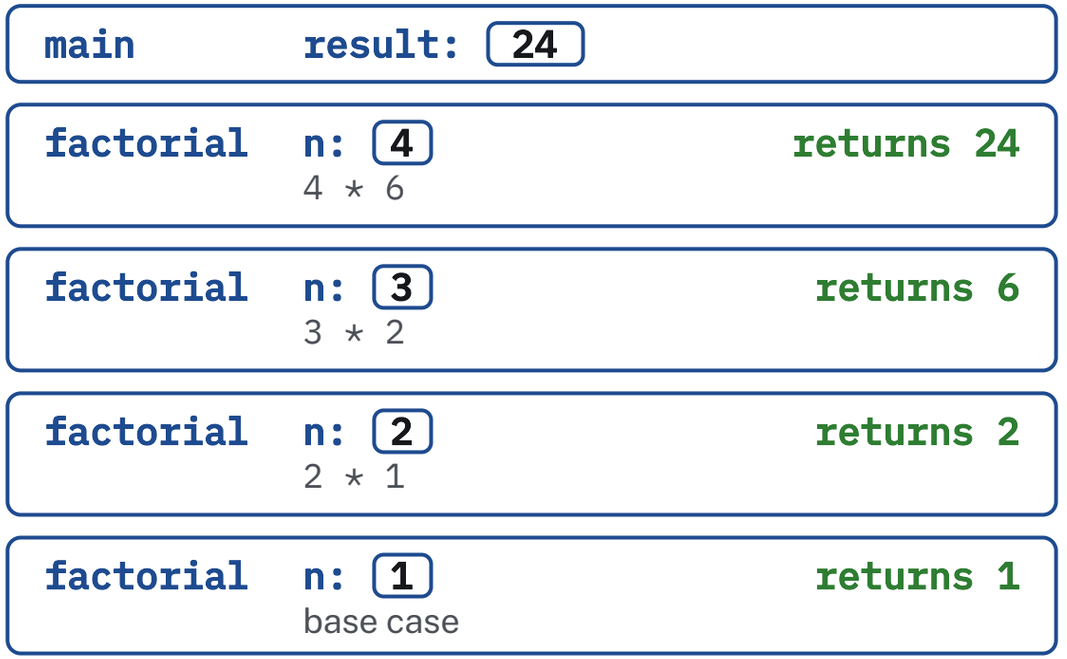
```python
def factorial(n):
    if n <= 1:
        return 1
    else:
        return n * factorial(n - 1)

result = factorial(4)
print(result)
```

OUTPUT

```
24
```

CALL STACK



- The multiplication waits for the smaller factorial
- Multiplying happens on the way back up `2 * 1`, then `3 * 2`, then `4 * 6`

Part 2 of 4

Pending operations with lists

Adding up a list: `list_sum`

```
def list_sum(nums):
    """
    If nums is empty, returns 0
    Otherwise, returns the sum of nums
    """
    if len(nums) == 0:
        return 0
    else:
        return nums[0] + list_sum(nums[1:])
```

- Each `+` waits for the sum of a shorter list
- The empty list adds 0
  Then the additions finish from the inside out

EXPANDING THE CALLS

```
list_sum([4, 7, 2])
= 4 + list_sum([7, 2])
= 4 + (7 + list_sum([2]))
= 4 + (7 + (2 + list_sum([])))
= 4 + (7 + (2 + 0))
= 4 + (7 + 2)
= 4 + 9
= 13
```

What does this print?

```
def mystery(nums):
    if len(nums) == 0:
        return 0
    else:
        return 1 + mystery(nums[1:])

print(mystery([8, 3, 5]))
```

Go to **slido.com** and enter the code: 3674841

Or join directly here

What does this print?

```
def mystery(nums):
    if len(nums) == 0:
        return 0
    else:
        return 1 + mystery(nums[1:])

print(mystery([8, 3, 5]))
```

EXPANDING THE CALLS

```
mystery([8, 3, 5])
= 1 + (1 + (1 + 0))
= 3
```

It prints 3

Each call adds 1 and strips off `nums[0]`
So `mystery` counts the items

Like counting your row

1 for me, plus the count
for the rest of the row

Tail recursion or pending operation?

Tail recursion: the recursive call is the last thing the function does

Pending operation: work waits for the recursive call to return

```
def spiral(length):
    if length > 10:
        t.forward(length)
        t.left(60)
        spiral(length * 0.9)
```

Tail recursion

```
def print_up(n):
    if n > 0:
        print_up(n - 1)
        print(n)
```

Pending: `print(n)`

```
def show(word):
    if word == "":
        print("Done!")
    else:
        print(word)
        show(word[1:])
```

Tail recursion

```
def doubled(nums):
    if len(nums) == 0:
        return []
    else:
        return [nums[0] * 2] + doubled(nums[1:])
```

Pending: `[...] +`

Part 3 of 4

The turtle comes home

Last time's spiral, coming home

```
import turtle as t

def spiral(length):
    if length > 10:
        t.forward(length)
        t.left(60)
        spiral(length * 0.9)
        t.right(60)
        t.backward(length)

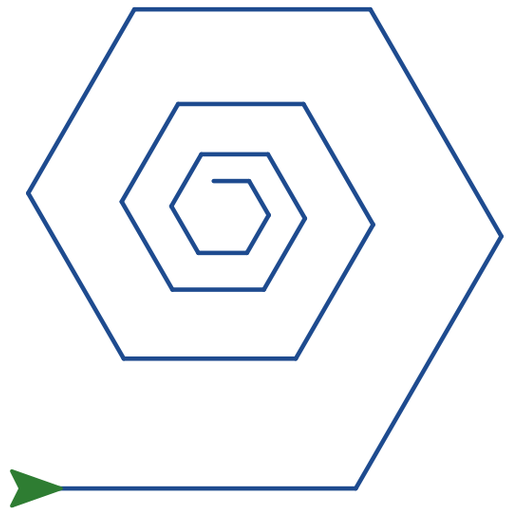
spiral(100)
t.done()
```

New: pending operations

CURRENT CALL

```
back in spiral(100), done
```

TURTLE WINDOW



- Each call undoes its own turn and move after the smaller spiral returns
- The turtle ends where it started facing the same way

A recursive tree

```
def draw_tree(length, levels):
    if levels > 0:
        t.forward(length)
        t.left(45)
        draw_tree(length / 2, levels - 1)
        t.right(90)
        draw_tree(length / 2, levels - 1)
        t.left(45)
        t.backward(length)

draw_tree(100, 2)
```

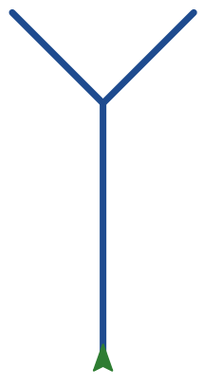
CURRENT CALL

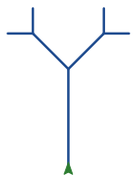
`draw_tree(100, 2) is done`

Two recursive calls

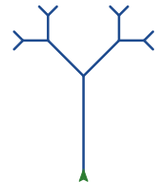
Come home

TURTLE WINDOW





`levels = 3`



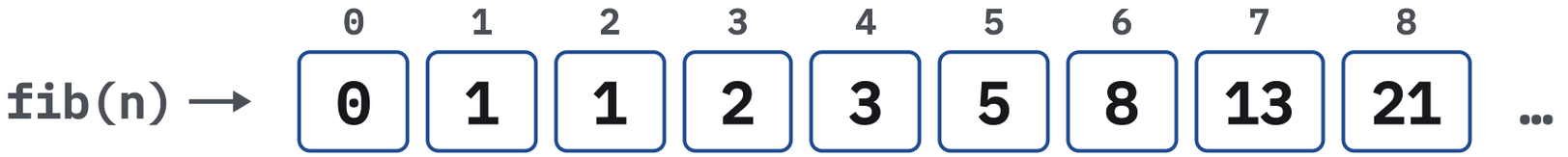
`levels = 4`

Part 4 of 4

Two recursive calls

Fibonacci numbers

Each number is the sum of the two before it



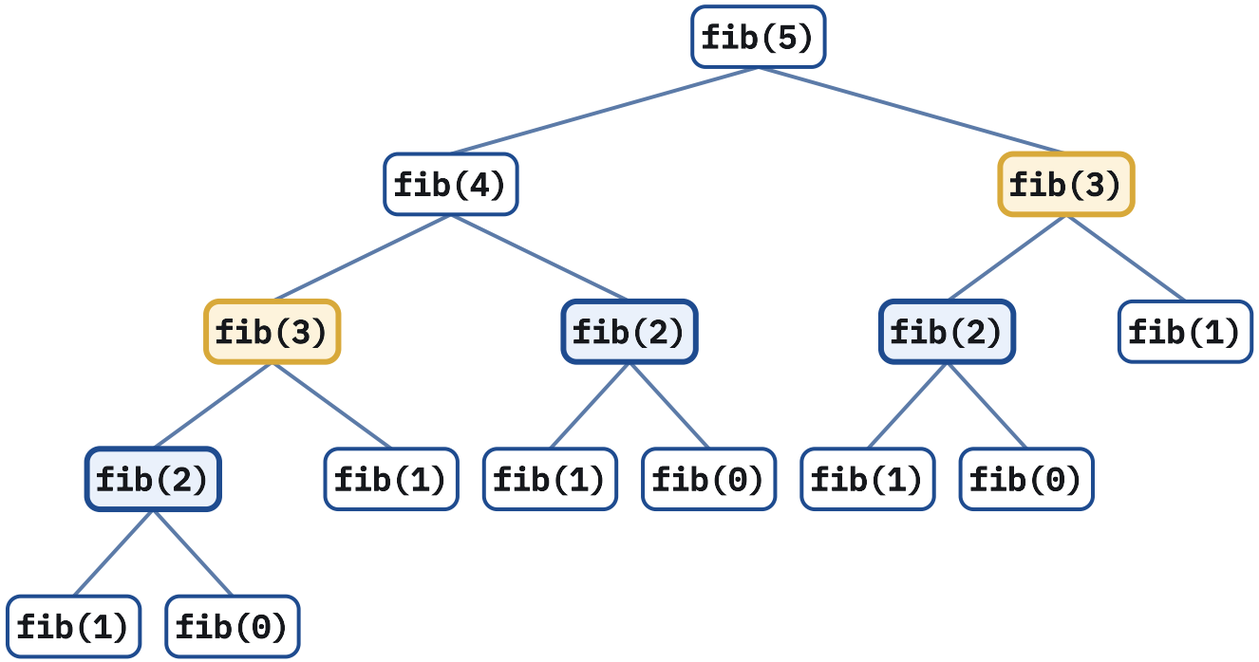

```
def fib(n):
    if n < 2:
        return n
    else:
        return fib(n - 1) + fib(n - 2)
```

Base cases

Recursive case

- `fib(0)` is 0, `fib(1)` is 1
  One `if` covers both
- The `+` waits for two recursive calls

The calls for `fib(5)`

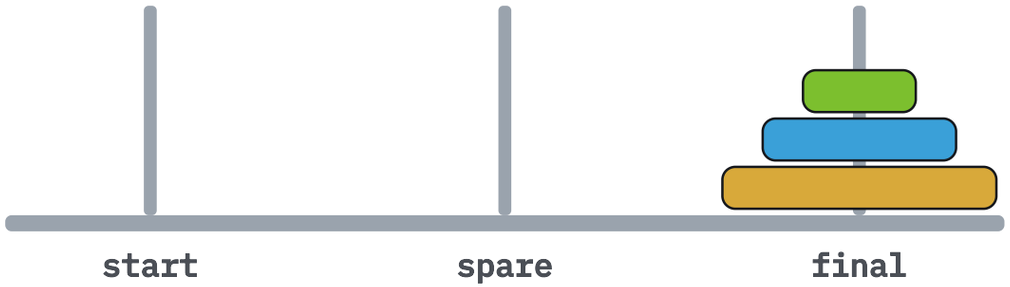


- `fib(3)` is computed twice
- `fib(2)` is computed 3 times
- `fib(30)` makes 2,692,537 calls
  Let's time it in Thonny
- A faster version that passes the last two numbers along:
  `fib_helper` in the demo file

Towers of Hanoi

Move the whole tower from start to final

Move one disk at a time, never a bigger disk on a smaller one



1. Move the top `n - 1` disks from start to spare
2. Move the biggest disk from start to final
3. Move the `n - 1` disks from spare to final

Play it: mathsisfun.com/games/towerofhanoi.html

Towers of Hanoi in Python

```python
def hanoi(n, start, final, spare):
    if n > 0:
        hanoi(n - 1, start, spare, final)
        print("Move disk", n, "from", start, "to", final)
        hanoi(n - 1, spare, final, start)

hanoi(3, "A", "C", "B")
```

OUTPUT

```
Move disk 1 from A to C
Move disk 2 from A to B
Move disk 1 from C to B
Move disk 3 from A to C
Move disk 1 from B to A
Move disk 2 from B to C
Move disk 1 from A to C
```

- Two recursive calls with one move in between
- 3 disks: 7 moves

  `n` disks: $2^n - 1$ moves
- 64 disks at one move per second: about 585 billion years

Activity

1. Download day11.zip.
2. Drag `day11.zip` to your `class` folder and extract it there.
   Mac — double-click it. **Windows** — right-click it and choose "Extract All...".
   Once you see the `day11` folder, delete `day11.zip`.
3. Launch Thonny. From the File menu, open `cs145/class/day11/day11_recursion.py`.
   The code from today's slides is in `day11_demos.py`.
4. For each function, decide on the base case and the recursive case first.
5. Write `product`, `count_evens`, and `reverse`.
6. Play with the turtle: uncomment `turtle_play()` at the bottom and run the file.
   Try other lengths, levels, and angles.
7. If you have time, write `bounce`.

To do

- Review this slide deck and the corresponding notes: Notes 9 and Notes 10
- Exam 2 is Friday 10/23 in lab, covering Lectures 5–9
  See practice materials

Next time

Lecture 12 — tuples and dictionaries