Middlebury

CSCI 146: Intensive Introduction to Computing

Fall 2025

Lecture 12: Recursion

Warmup: practicing with references and mutability from last week.

- **Quiz topic 6.1:** (this Friday) Determine the effect of operations on aliased data structures
- **Exam topic 10:** Implications of the Python memory model

```
a = [3, [4, 5], 6]
b = a[:]
b.append([7, 8])
a[1][0] = 3
```

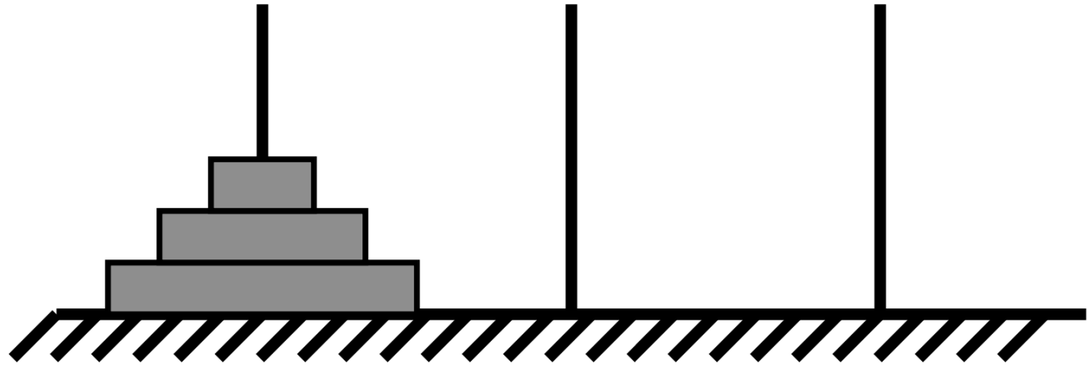
What are the values of `a` and `b` after this code executes?

Solution:

- `a = [3, [3, 5], 6]`,
- `b = [3, [3, 5], 6, [7, 8]]`.

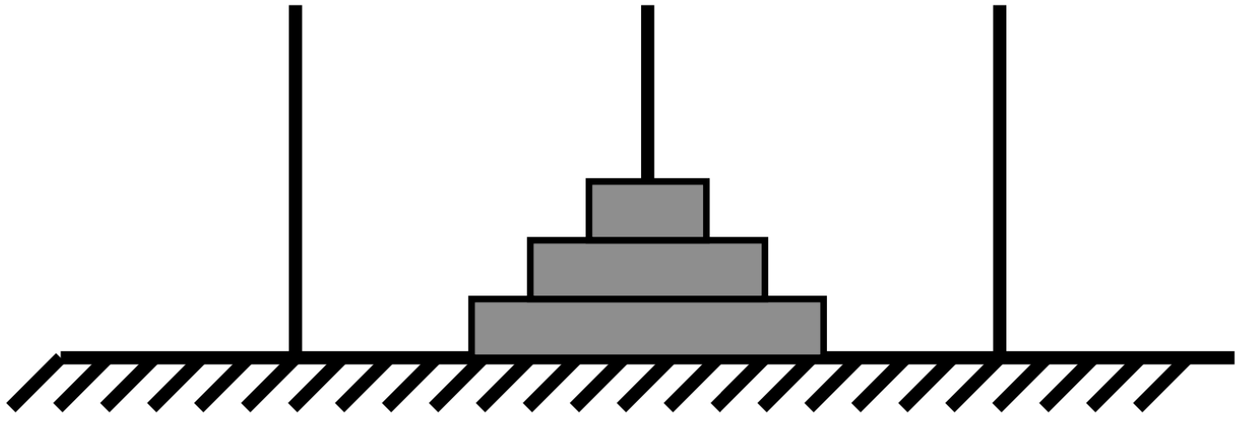
Goals for today

- Define recursion, base case, and recursive case
- Identify the base case and recursive case of a recursively-defined problem
- Write a recursive function to solve a computational problem
- Implement recursion with pending operations



Try the Tower of Hanoi problem: https://www.mathsisfun.com/games/towerofhanoi.html

Solving the Tower of Hanoi problem.



To displace a stack of n disks:

- Displace top $(n-1)$ disks to an empty rod (assuming we know how to do that).
- Display bottom disk to the remaining rod.
- Display top $(n-1)$ disks onto the bottom disk again.

A more mathy example: computing the factorial of some number n.

$$n! = n \cdot (n-1) \cdot (n-2) \cdot (n-3) \ldots 3 \cdot 2 \cdot 1$$

```python
def factorial(n):
    result = 1
    for i in range(2, n + 1):
        result *= i
    return result
```

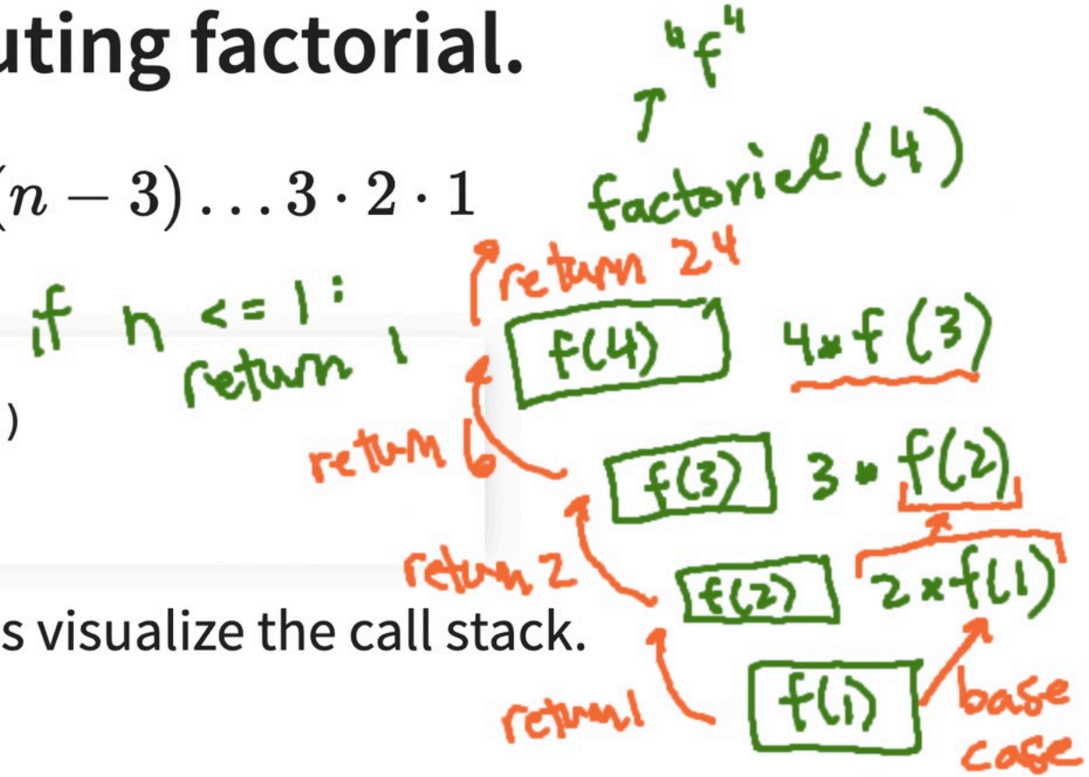
But $n! = n \cdot (n-1)!$.

So if we know how to compute $(n-1)!$, then we can compute $n!$.

A recursive approach to computing factorial.

$$n! = n \cdot (n-1) \cdot (n-2) \cdot (n-3) \ldots 3 \cdot 2 \cdot 1$$

```python
def factorial(n):
    return n * factorial(n - 1)
```



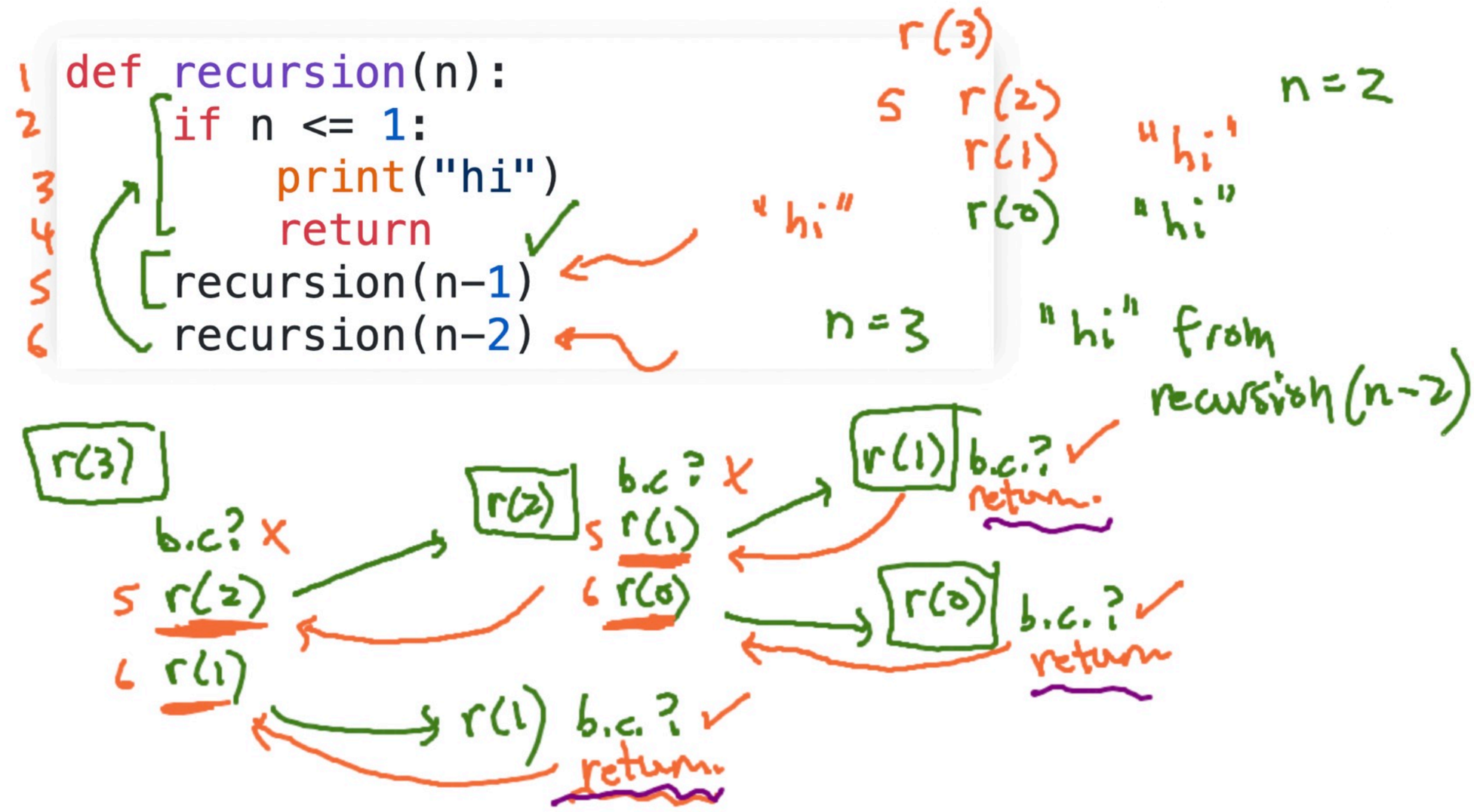
THIS WILL NEVER TERMINATE!!! Let's visualize the call stack.

We need two ingredients for a recursive function:

- **Base case:** problem size we know how to solve (usually the smallest problem size, but not necessarily).
- Treatement of **recursive case** that approaches base case.
  - The problem size might decrease linearly (e.g. by -1) or by some factor (e.g $/2$).

Question 1: How many times will `"hi"` be printed when you invoke `recursion(3)`?

```
1 def recursion(n):
2     if n <= 1:
3         print("hi")
4         return
5     recursion(n-1)
6     recursion(n-2)
```



A. 0

B. 1

C. 2

D. 3

E. 4

Question 2: How many times will `"hi"` be printed when you invoke `recursion(3)`?

```
def recursion(n):
    if n <= 2:
        print("hi")
        return
    recursion(n-1)
    recursion(n-2)
```

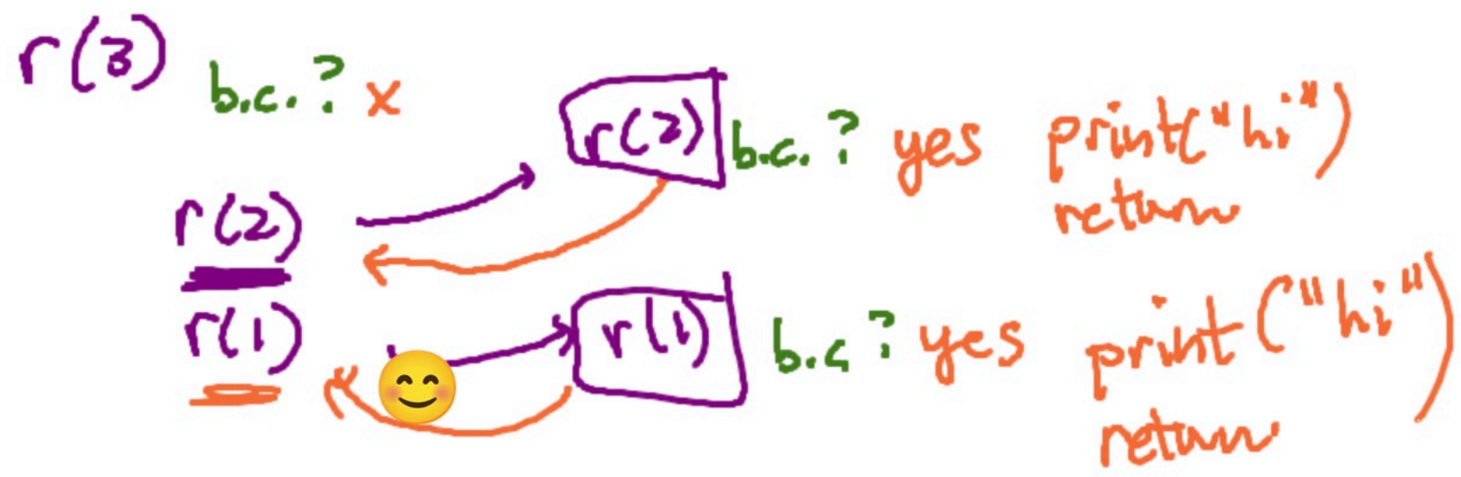
A. 0

B. 1

C. 2

D. 3

E. 4



Tips for writing recursive functions.

- Define the function header, including the parameters
- Define the recursive case
  - Assume your function works as intended, but only on smaller instances of the problem.
  - The recursive problem should get "smaller" (or it will never finish!).
- Define the base case
  - What is the smallest (or simplest) problem? It should have a direct (i.e. non-recursive) solution.
- Put it all together.
  - First, check for the base and return (or do) something specific.
  - If the computation hasn't reached the base case, compute the solution using the recursive definition and return the result.

length = 0

Example: writing a recursive function to reverse a string called `reverse`.(s)

n = 5 "hello"

if you could reverse a length-4 string, how would you use that?

idea 1: reverse first n-1 chars then put last char in front.

idea 2: reverse last n-1 chars then put first char at end.

Possible recursive implementation for `reverse`.

```
def reverse(a_string):
    if len(a_string) == 0:
        return ""
    else:
        return reverse(a_string[1:]) + a_string[0]
```

Pending operations refers to the code that awaits the recursive calls to finish.

What is the output of calling `go_back(3)`?

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

Question 3: which of the following functions will recurse infinitely?

```
# Function 1
def mystery(n):
  if n <= 1:
    return 1
  else:
    return (n-1) * mystery(n)
```

```
# Function 2
def mystery(n):
  if n <= 1:
    return 1
  else:
    return n * mystery(n - 1)
```

```
# Function 3
def mystery(seq):
  if len(seq) == 0:
    return 0
  else:
    return 1 + mystery(seq[:len(seq)])
```

A. Function 1 only

B. Function 3 only

C. Functions 1 and 2

D. Functions 1 and 3

E. All

Question 4: which of the following functions have pending operations?

```python
# Function 1
def mystery(n):
  if n == 0:
    return
  else:
    mystery(n - 1)
    print("Unwinding:", n)
```

```python
# Function 2
def mystery(n):
  if n <= 1:
    return 1
  else:
    return n * mystery(n - 1)
```

```python
# Function 3
def mystery(n, acc):
  if n == 0:
    return acc
  else:
    return mystery(n - 1, acc * n)
```

A. Function 1 only

B. Functions 1 and 2

C. Functions 1 and 3

D. Functions 2 and 3

E. All

Exercise: write a *recursive* palindrome checker.

Here is a loop-based implementation:

```
def is_palindrome_loop(word):
    """
    Determines if a word is a palindrome.
    Args:
      word: word to check (str)
    Returns:
      True if the input word is a palindrome, False otherwise
    """
    for i in range(len(word) // 2):
      if word[i] != word[-i - 1]:
        return False
    return True
```

Examples: racecar, noon, kayak, madam, rotator

When you're done, try to extend it to ignore punctuation and spaces to handle *palindrome phrases*:

1. A Toyota
2. If I had a hi-fi,
3. UFO tofu
4. Never odd or even.
5. A man, a plan, a canal - Panama!

Summary and Reminders

- Always remember to (1) include a base case and (2) make sure your recursive case approaches the base case.
- Recursive `turtle` on Wednesday.
- Programming Assignment 4 final due date on Thursday.
- Use "Regrade Requests" form on the website. See Gradescope comments by clicking on **Code**.