Hack110 Interest Form!

When? Saturday, November 8th from 10 AM - 12 AM (Midnight)

Where? In Sitterson Lower Lobby

Who can join? Anyone in COMP 110! No prior experience required. Bring a partner, find one there, or go solo!

Come for a fun day of coding, workshops and events (also **food and CLE credit will be provided**):

- Choose between web development or game development track
- Go to various **workshops & events** such as: Navigating the CS Major, Resume workshop, ice cream station, and kahoot trivia and MORE!
- Link: Interest Form Here! Or via the QR code
- **Interest form will close Sunday, Sept 21st at 11:59 pm**
  - Fill out this form to get **priority notice** of when we release the sign-up form.

Interest Form!

Positional Arguments & Recursion Practice

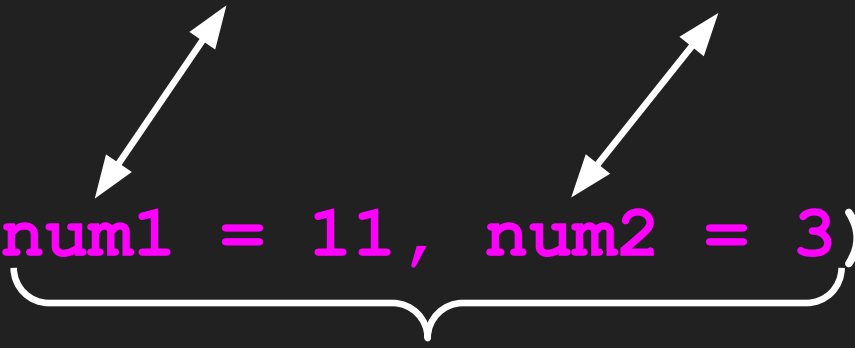
Reminders

- **Quiz 00:** Regrade requests will be open **till 11:59pm tonight!**
  - Please submit a regrade request if you believe your quiz was not graded correctly according to the rubric

Want extra support? We're here and *want* to help!

Recall: Signature vs Call

```
def divide(num1: int, num2: int) -> float:
```



```
divide(num1 = 11, num2 = 3)
```

These are called keyword arguments, since you are assigning values based on the parameter names.

Keyword arguments

```
def divide(num1: int, num2: int) -> float:
```

```
divide(num1 = 11, num2 = 3)
```

Benefit of keyword arguments: order of arguments doesn't matter.

Keyword arguments

```
def divide(num1: int, num2: int) -> float:
```

```
divide(num1 = 11, num2 = 3)
divide(num2 = 3, num1 = 11)
```

Benefit of keyword arguments: order of arguments doesn't matter.

Positional Arguments

```
def divide(num1: int, num2: int) -> float:
```

```
divide(11, 3)
```

For positional arguments, values are assigned based on the order (*position*) of the arguments.

Review: Checklist for developing a recursive function:

Base case:

- ❑ Does the function have a clear base case?
  - ❑ Ensure the base case returns a result directly (without calling the function again).
- ❑ Will the base case *always* be reached?

Recursive case:

- ❑ Ensure the function moves closer to the base case with each recursive call.
- ❑ Combine returned results from recursive calls where necessary.
- ❑ Test the function with edge cases (e.g., empty inputs, smallest and largest valid inputs, etc.). Does the function account for these cases?

`factorial` Algorithm

✱ Reminder: there are multiple ways to write this recursive function!

Create a recursive function called `factorial` that will calculate the product of all positive integers less than or equal to an int, `n`. E.g.,

`factorial(n=5)` would return: 5*4*3*2*1 = **120**

`factorial(n=2)` would return: 2*1 = **2**

`factorial(n=1)` would return: 1 = **1**

`factorial(n=0)` would return: **1**

Conceptually, what will our **base case** be? ✱

when we want the recursion to stop

What will our **recursive case** be? ✱

when we want to recursively call the function again

```
if n == 0 or n == 1:
    return 1
elif n > 1:
    return n * factorial(n-1)
else:
    raise ValueError("Please call w/ non-negative arg.")
```

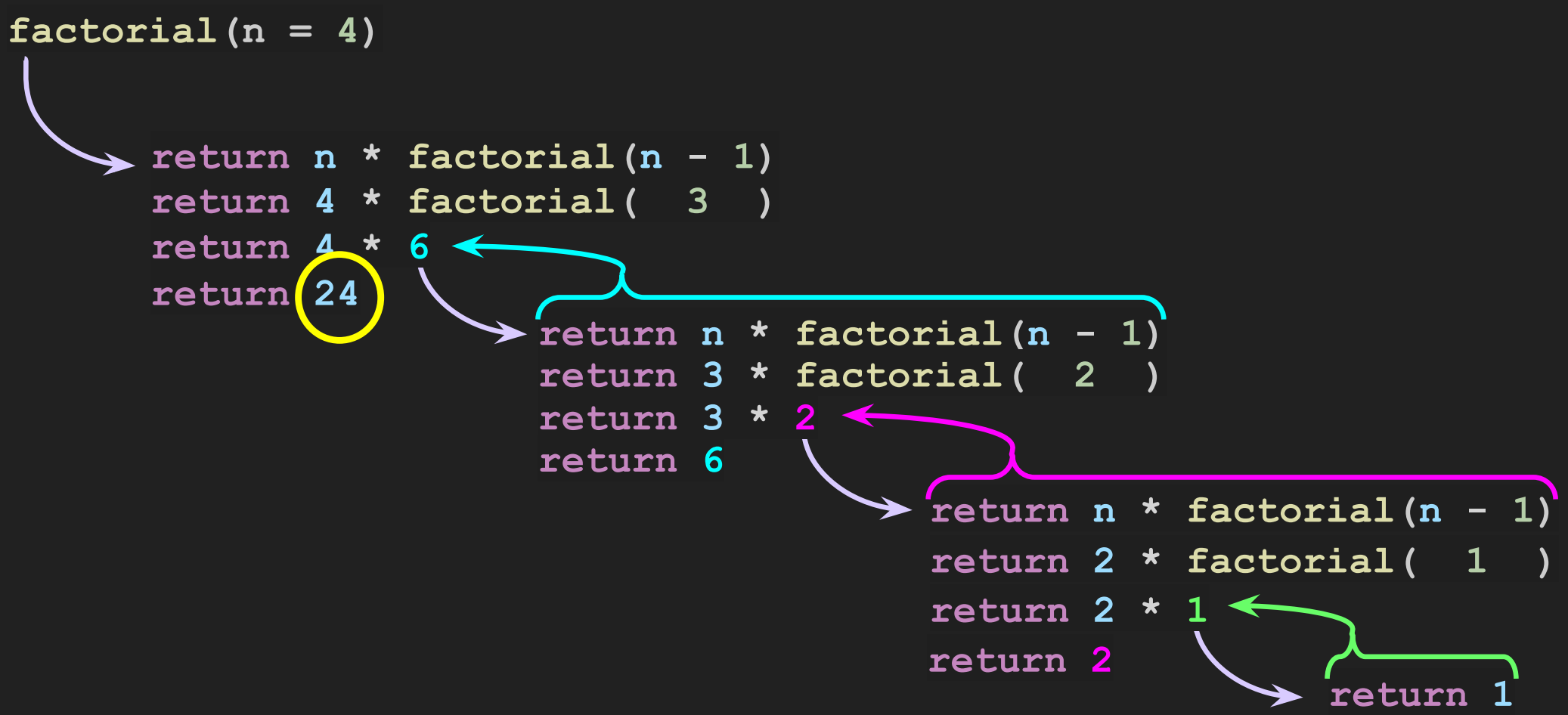
(this is just one way to account for it!)

What is an **edge case** for this function? How could we account for it?

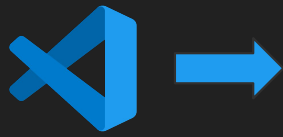
factorial is typically only defined for non-negative values of n, but this function can be called with any int argument. We need to account for this!

Visualizing recursive calls to `factorial`

Visualizing recursive calls to `factorial`



Let’s write the `factorial` function in VS Code!



Memory diagram (without considering the edge case, for space reasons)

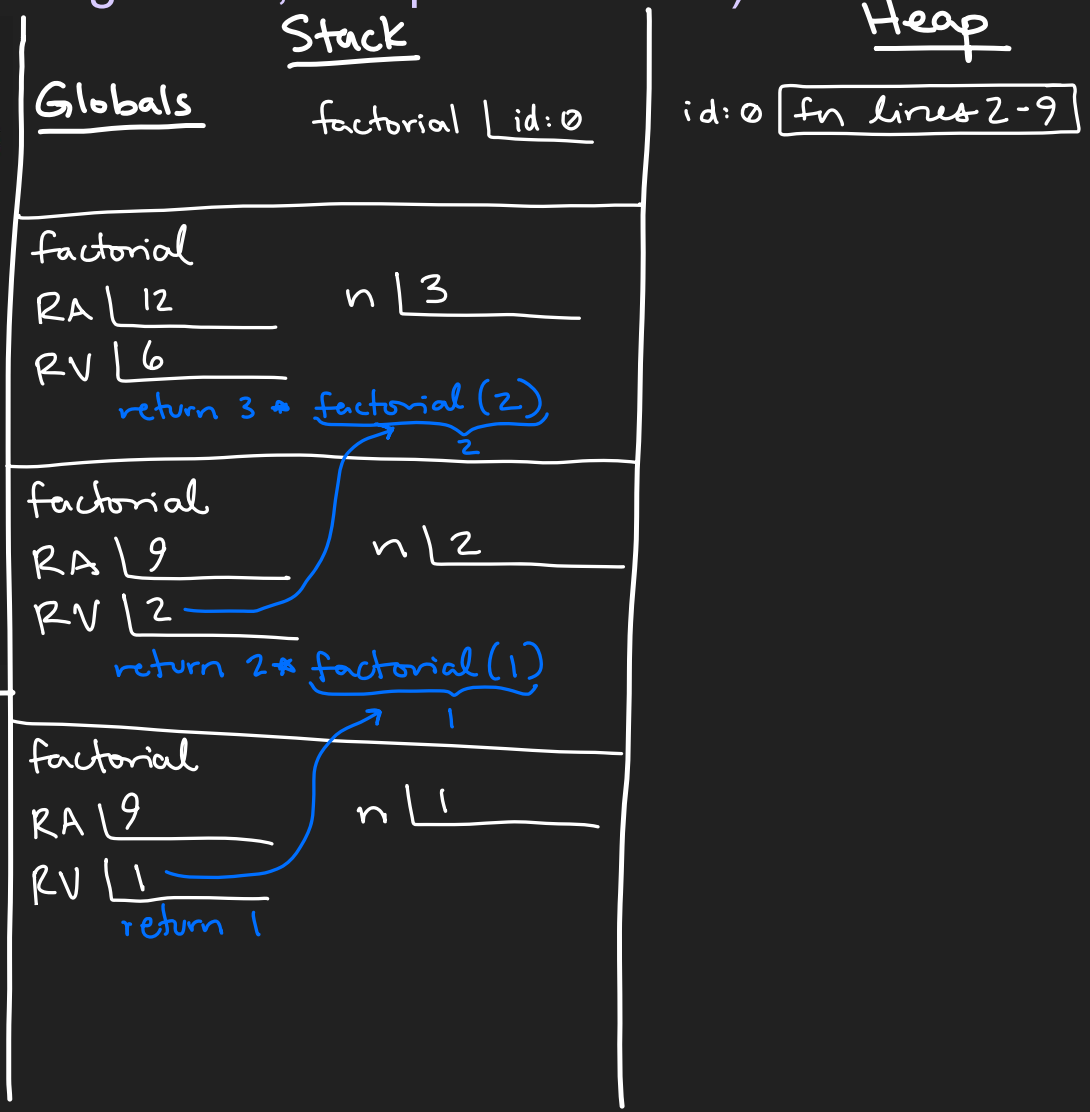
```python
# Factorial
def factorial(n: int) -> int:
    """Calculates factorial of int n."""
    # Base case
    if n == 0 or n == 1:
        return 1
    # Recursive case
    else:
        return n * factorial(n - 1)

# Example usage
print(factorial(3))
```

* notes in blue are just to help you track the return statements!

Output

6



Hand-writing code: An adaptation of `fizzbuzz`

A group of students start counting up from 1, taking turns saying either a number or a phrase.

If their number is divisible by 3, the student says “fizz” rather than the number.

If their number is divisible by 5, they say “buzz” rather than the number.

If their number is divisible by both 3 and 5, they say “fizzbuzz”

Example:

1, 2, fizz, 4, buzz, fizz, 7, 8, fizz, buzz, 11, fizz, 13, 14, fizzbuzz, 16, …

Hand-writing code: An adaptation of `fizzbuzz`

Our function definition should meet the following specifications:

- The function should be named `fizzbuzz`, have one `int` parameter named `n`, and return an `int`
- If `n` is divisible by 3 and not 5, the function should print "fizz"
- If `n` is divisible by 5 and not 3, the function should print "buzz"
- If `n` is divisible by 3 AND 5, the function should print "fizzbuzz"
- If `n` is not divisible by 3 OR 5, the function should print `n`'s literal value
- The function should keep calling itself, increasing the argument by 1 each time, until we finally reach a "fizzbuzz" number, when we'll return `n`
- Explicitly type your parameter and return type.

Solution

```python
def fizzbuzz(n: int) -> int:
    if n % 3 == 0 and n % 5 == 0:  # Base case
        print("fizzbuzz")
        return n
    elif n % 3 == 0:  # If n is divisible by 3 but NOT 5
        print("fizz")
    elif n % 5 == 0:  # If n is divisible by 5 but NOT 3
        print("buzz")
    else:  # If n is not divisible by 3 OR 5
        print(n)
    # If fizzbuzz wasn't reached this time, call function again with n+1
    return fizzbuzz(n=n + 1)
```