Recursion Practice

Reminders

- **Quiz 01 on Friday**
  - Practice problems on the site
  - Want help with a practice problem? Visit Office Hours or Tutoring today!
  - **Hybrid Review Session today** from 6-7pm in Sitterson Hall (SN) 014 and online (link on site)
- **CQ01: Recursion Memory Diagram** due at 11:59pm

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

Review: Is this function recursive?
What was wrong with it?
What kind of error would it raise?

```
1    def icarus(x: int) -> int:
2        """Unbound aspirations!"""
3        print(f"Height: {x}")
4        return icarus(x=x + 1)
5
6
7    print(icarus(x=0))
```

Review: Stack Overflow and Recursion Errors

When a programmer writes a function that calls itself indefinitely (*infinitely*), the **function call stack** will *overflow…*

This leads to a **`Stack Overflow`** or **`Recursion Error`**:

```
RecursionError: maximum recursion depth exceeded while
calling a Python object
```

Review: `safe_icarus`

```
1   def icarus(x: int) -> int:
2       """Unbound aspirations!"""
3       print(f"Height: {x}")
4       return icarus(x=x + 1)
5
6   def safe_icarus(x: int) -> int:
7       """Bound aspirations!"""
8       if x >= 2:
9           return 1
10      else:
11          return 1 + safe_icarus(x=x + 1)
12
13  print(safe_icarus(x=0))
```

We avoid a RecursionError with safe_icarus with:

- A base case
- A recursive case that progresses toward the base case
  - Remember: to avoid a RecursionError, the base case must eventually be reached, no matter the arguments in the original function call

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?

So… when is recursion useful?

Recursion is useful when a problem can be defined in terms of smaller versions of itself.

Hints that recursion may be useful:

- The problem has a natural “stop” (a base case)
  - E.g., “stop when n is 0!”
- The data or process is hierarchical
  - E.g., a family tree, or computer files and folders
- The same operation repeats on smaller inputs
  - E.g, fibonacci, factorial

`factorial` Algorithm

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**

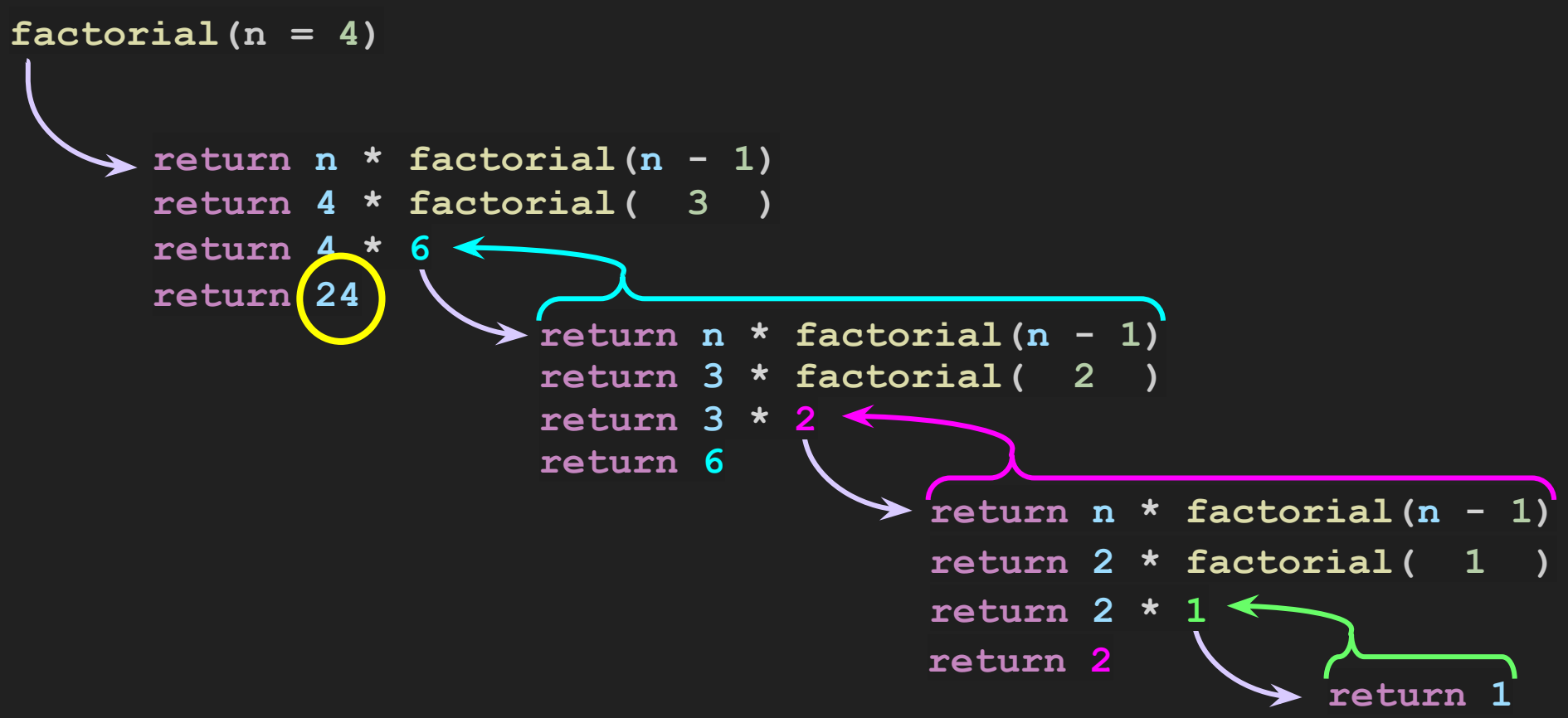
Conceptually, what will our **base case** be?

What will our **recursive case** be?

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

Visualizing recursive calls to `factorial`

Visualizing recursive calls to `factorial`



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

Memory diagram (***to submit for CQ01!***)

```python
def factorial(n: int) -> int:
    """Calculates factorial of int n."""
    if n < 0:  # Edge case
        return -1  # Or, we could raise an error
    elif n == 0 or n == 1:  # Base case
        return 1
    else:  # Recursive case
        return n * factorial(n - 1)


fac_3: int = factorial(3)
print(fac_3)
```

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

```
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)
```