Recursive Structures & Processes

Announcements

RD00 and **LS12 (Recursive Structures)** due today at 11:59pm!

EX05: River Simulation due Monday at 11:59pm!

- Let's review this together!

Re: Quiz 03:

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

Re: Quiz 04:

- Practice quiz on the site
  - Please visit Office Hours if you have questions!
- Virtual Review Session on Thursday at 6pm

Recall these functions: what was the issue with the icarus function?

```
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))
```

The dreaded **`Recursion Error`**!

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
```

Recursive function checklist:

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:

- ❏ Does the function have a recursive case that *progresses toward the base case*?
  - ❏ Does the recursive call have the right arguments? The function should call itself on a simpler or smaller version of the problem.
- ❏ Have you tested your function with multiple cases, including edge cases?

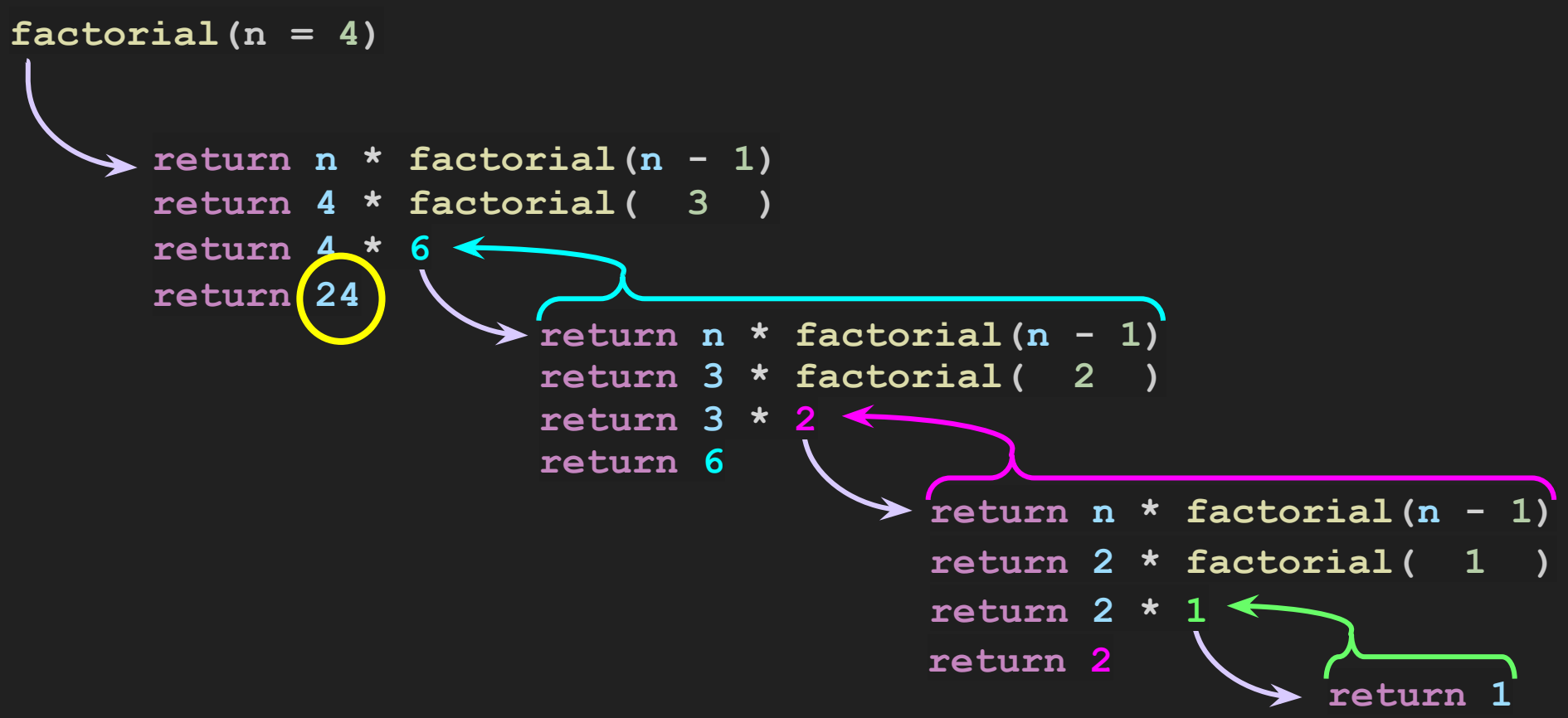
Another example of recursion: factorial!

To calculate the factorial of an int, `n`, we would multiply `n` by `(n-1)`, then `(n-2)`, and so on, until we reach `1`.

For instance, to calculate `5!`, we would do: `5 * 4 * 3 * 2 * 1`, which would evaluate to `120`.

```
def factorial(n: int) -> int:
    # Base case: factorial of 0 or 1 is 1
    if n <= 1:
        return 1
    # Recursive case: n! = n × (n-1)!
    return n * factorial(n - 1)
```

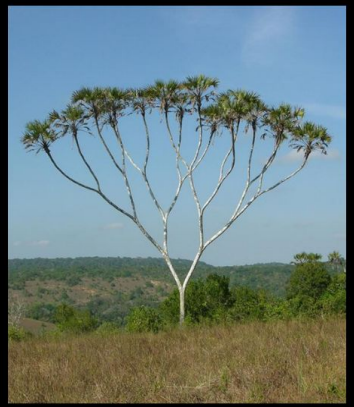
Visualizing recursive calls to `factorial`

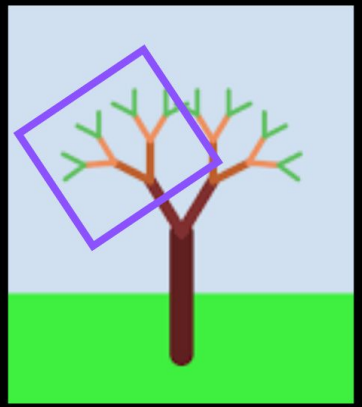


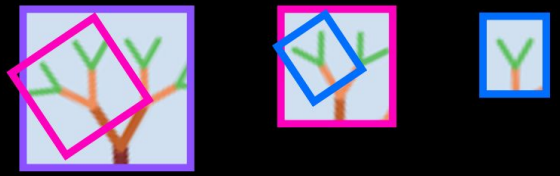
Recursion: defining an operation/object in terms of itself

A real-world phenomenon! Examples:

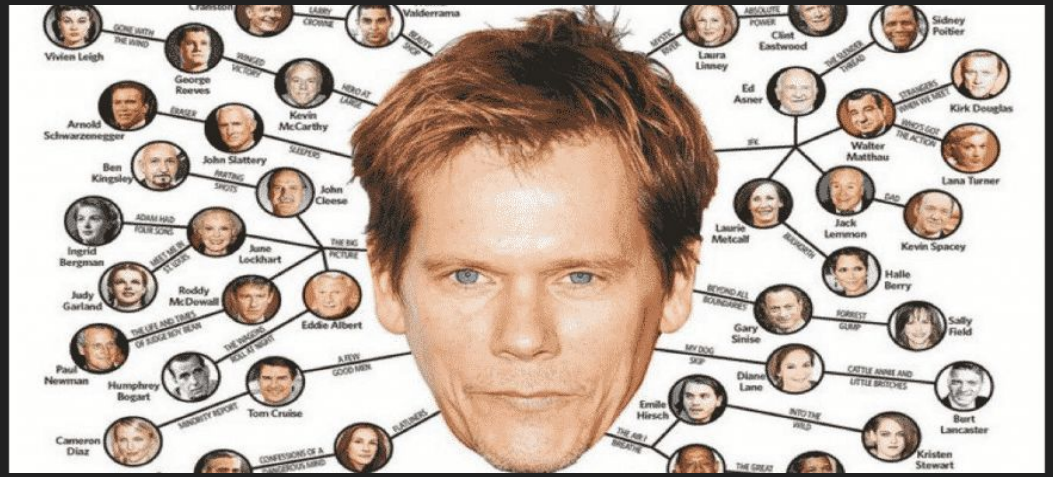
- **You** have **parents**, who have **parents**, who have **parents**, who have **parents**, who…
  … were **early humans**
- A **tree** has **branches**, which have **branches**, which have **branches**, which…
  … have **leaves**





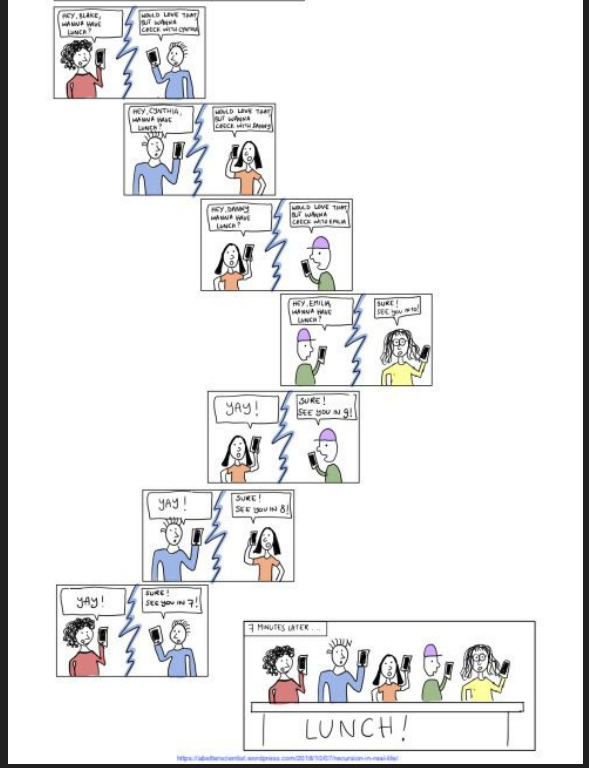


Different recursive structures for different purposes



Six degrees of Kevin Bacon

graph/network



Coordinating plans with individual phone calls

linked list

Anatomy of a Singly-Linked List

Memory diagram

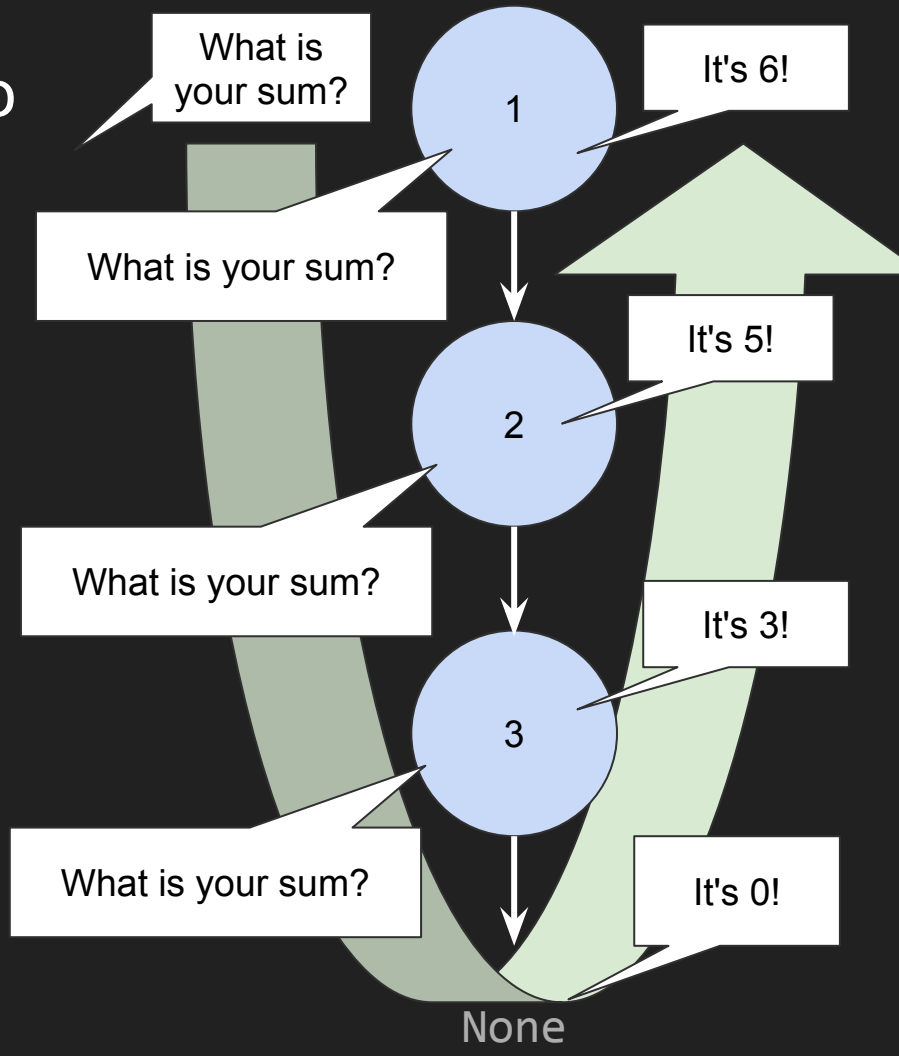
```python
from __future__ import annotations  # Ignore for now!

class Node:
    value: int
    next: Node | None

    def __init__(self, val: int, next: Node | None):
        self.value = val
        self.next = next

# Note: There are no errors!
two: Node = Node(2, None)
one: Node = Node(1, two)
# We'll extend this diagram shortly, leave room
```

A Recursive sum Algorithm Demo

1. When you are asked, "what is your sum?"

2. Ask the ***next*** Node, "what is your sum?"
   Wait patiently for an answer!

3. Once the answer is returned back to you, add ***your value to it***, then turn to the person who asked you and give them this answer.



Let's write a recursive function called `sum`!

```python
from __future__ import annotations  # Ignore for now!

class Node:
    value: int
    next: Node | None

    def __init__(self, val: int, next: Node | None):
        self.value = val
        self.next = next

# Note: There are no errors!
two: Node = Node(2, None)
one: Node = Node(1, two)
# We'll extend this diagram shortly, leave room
```

Write a function called `sum` that adds up the `value`s of all `Node`s in the linked list.

Diagramming the `sum` function call

```python
from __future__ import annotations

class Node:
    value: int
    next: Node | None

    def __init__(self, val: int, next: Node | None):
        self.value = val
        self.next = next

# Note: There are no errors!
two: Node = Node(2, None)
one: Node = Node(1, two)

def sum(head: Node | None) -> int:
    if head is None:
        return 0
    else:
        rest: int = sum(head.next)
        return head.value + rest

print(sum(one))
```

Your turn: Memory Diagram

```python
from __future__ import annotations

class Node:
    """Node in a singly-linked list recursive structure."""
    value: int
    next: Node | None

    def __init__(self, value: int, next: Node | None):
        self.value = value
        self.next = next

    def __str__(self) -> str:
        if self.next is None:
            return f"{self.value} -> None"
        else:
            return f"{self.value} -> {self.next}"

courses: Node = Node(110, Node(210, None))
print(courses)
```

and discuss with a neighbor:

1. What does the __str__ method do?
2. Is this method recursive? How do we know?

Memory Diagram

```python
from __future__ import annotations

class Node:
    """Node in a singly-linked list recursive structure."""
    value: int
    next: Node | None

    def __init__(self, value: int, next: Node | None):
        self.value = value
        self.next = next

    def __str__(self) -> str:
        if self.next is None:
            return f"{self.value} -> None"
        else:
            return f"{self.value} -> {self.next}"

courses: Node = Node(110, Node(210, None))
print(courses)
```

Copy this into VS Code!

```python
from __future__ import annotations

class Node:
    """Node in a singly-linked list recursive structure."""
    value: int
    next: Node | None

    def __init__(self, value: int, next: Node | None):
        self.value = value
        self.next = next

    def __str__(self) -> str:
        if self.next is None:
            return f"{self.value} -> None"
        else:
            return f"{self.value} -> {self.next}"

courses: Node = Node(110, Node(210, Node(211, None)))
print(courses)
```

A Recursive `last` Algorithm Demo

1. When you are asked,
   "What is the value of the last Node?"

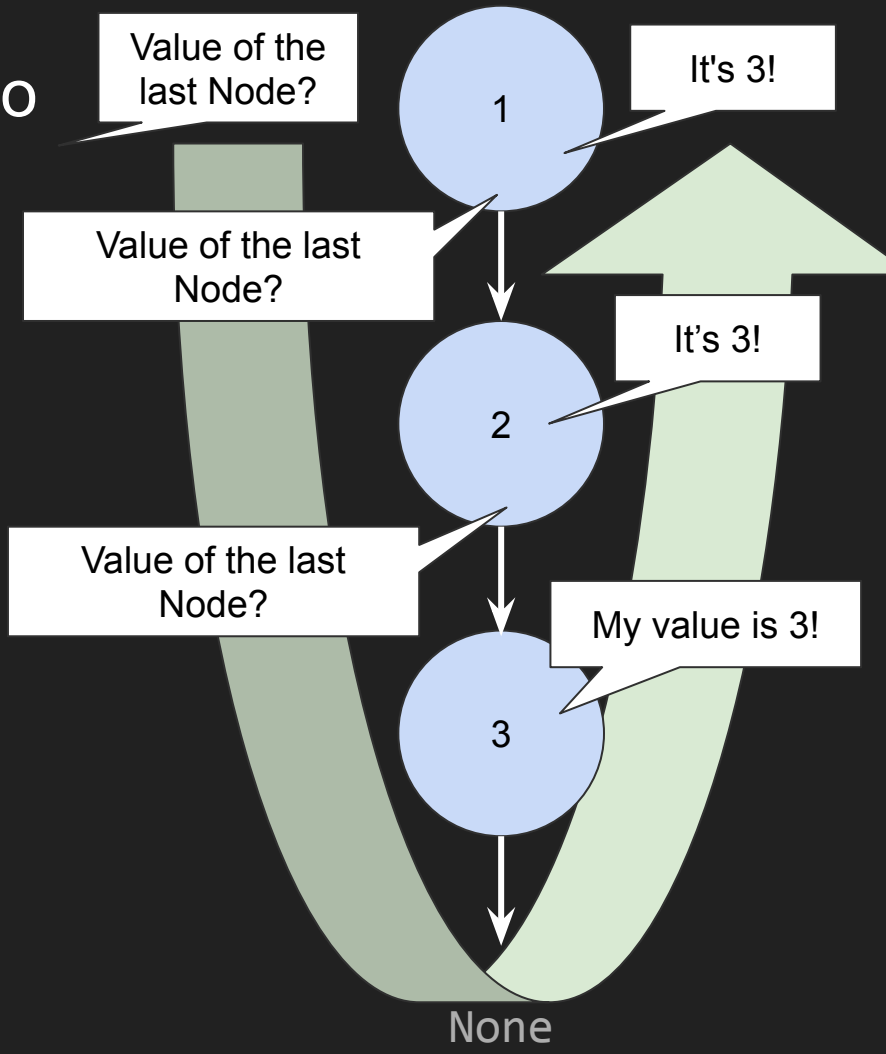
If you're ***not the last Node***:

2. Ask the ***next*** Node,
   "What is the value of the last Node?"
   Wait patiently for an answer!

3. Once the answer is returned back to you, turn to the person who asked you and give them this answer.

If you ***are the last Node***:

2. Tell them, "my value is ____!" and share your value.



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

`recursive_range` Algorithm

Create a recursive function called `recursive_range` that will create a linked list of Nodes with values that increment from a start value up to an end value (exclusive). E.g.,

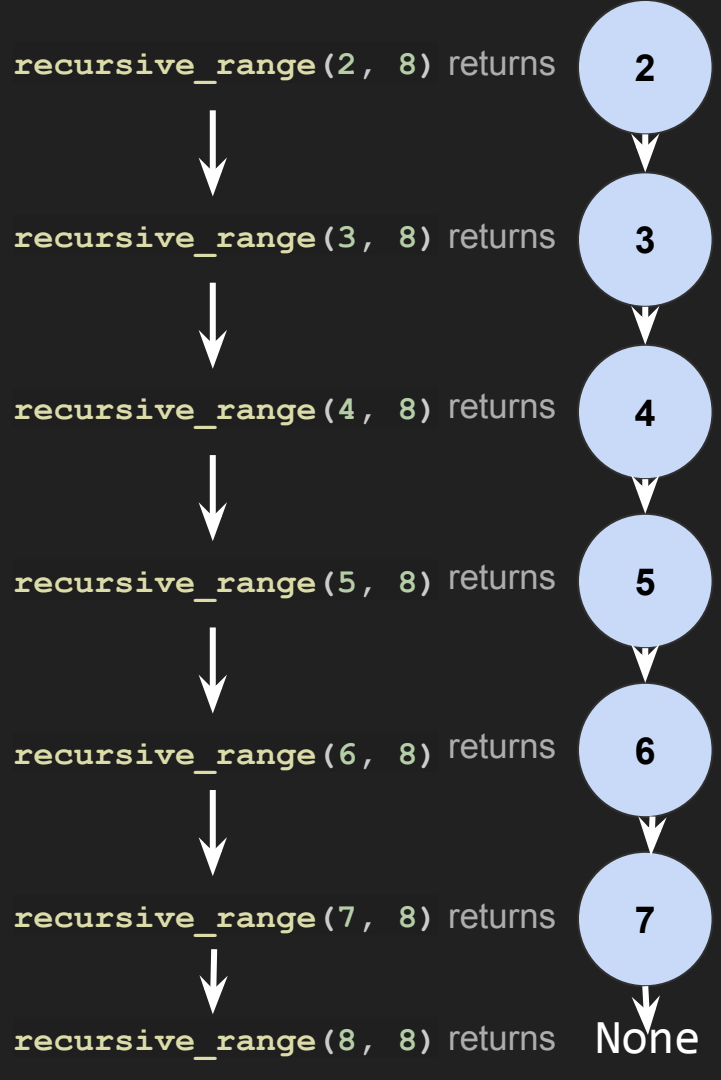
`recursive_range(start=2, end=8)` would return:

2 -> 3 -> 4 -> 5 -> 6 -> 7 -> None

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?



When "building" a new linked list in 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:

- ❏ Determine what the ***first*** value of the new list will be
- ❏ Then "build" the ***rest*** of the list by recursively calling the building function
- ❏ Finally, return a new ***Node(first, rest)***, representing the a new list

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