RECURSION II

REMINDERS

- Be on the lookout for final exam slot assignments coming very soon
- Exam 2 is one week from today in this room at this time
- This semester's practice exam 2 will be posted shortly
  - If you want to take it under "real" conditions, don't look at it until you come to the **TA Review Session** on Sun, Nov 8 from 12-1pm in Towne 100 (here)
  - 1-2pm of the review session will be spent going over the practice exam.
- HW4: Caesar Project component due tonight at 11:59pm

WARM UP: L1

```
def foo(N):
    if N <= 1:
        return 0
    return 1 + foo(N // 2)

def fizz(N):
    if N == 0:
        return 0
    return N % 10 + fizz(N // 10)
```

What do these print?

- `print(foo(3))`
- `print(fizz(1100))`

WRITING WITH RECURSION: AN INVENTORY

- **Recursion** is when a function calls itself to do its work.
- A **recursive call** is what happens when a function calls itself; the input to the recursive call should be a "smaller" version of the problem.
  - e.g. `multiply(3, 4)` can be evaluated as `4 + multiply(2, 4)`
- A **base case** is an input to a recursive function that does not require another recursive call.
  - Multiplication when one of the inputs is 1
  - Searching when the list is empty

WORKED EXAMPLE

Let's work up to a recursive solution to the following *food safety* problem:

> *Given a list of readings from a sensor, determine whether the temperature in a refrigerated case ever reached about 40° F.*

Putting our problem-solving hats on—not focusing on Python in particular—what would we need to do to answer this question?

```
[32.8, 34.9, 33.1, 32.0, 38.8, ..., ..., ..., 34.8, 39.0, 31.1]
```

BIG IDEAS

```
[32.8, 34.9, 33.1, 32.0, 38.8, ..., ..., ..., 34.8, 39.0, 31.1]
```

- Inspect every number in the list, doing so in an order that lets us know when we're done.
  - Randomly picking a number out of the pile? Not that good of an idea.
  - Visiting every element from left to right? Good—no skipping, and we're done when we hit the end of the list
- If we see a too-hot temperature, we're done: *the food is not safe*
- If we see a temperature that is safe, we have to keep going: *one bad reading spoils the bunch.*

WRITING RECURSION: BEING LAZY

Recursion is all about *categorization* of inputs and *delegation* of work to other function calls.

- *categorization*: is this a **base case,** or does someone need to do more work for me?
- *delegation*: if I make a recursive call to get the work done, how do I shape that call and how do I use its results?

THINK!

In which of these cases are we done, and in which do we need to do more work?
(Remember: we read temperatures in order from left to right.)

# READINGS	CURRENT READING NUMBER	CURRENT READING	DONE AFTER THIS READING?
3,123,432	3,123,432	32.4	
3,123,432	14	32.4	
3,123,432	1,344	41.0	
10	10	31.0	
0	0	---	

THINK!

# READINGS	CURRENT READING NUMBER	CURRENT READING	DONE AFTER THIS READING?
3,123,432	3,123,432	32.4	✅
3,123,432	14	32.4	❌
3,123,432	1,344	41.0	✅
10	10	31.0	✅
0	0	---	✅

We can *categorize* our cases without looking at all readings in the list. Just pay attention **progress so far** and **whether the current reading stops us.**

”I’M DONE HERE” ➡️ BASE CASE!

# READINGS	CURRENT READING NUMBER	CURRENT READING	DONE AFTER THIS READING?
3,123,432	3,123,432	32.4	✅
3,123,432	14	32.4	❌
3,123,432	1,344	41.0	✅
10	10	31.0	✅
0	0	---	✅

C1 Write down some "pseudo-Python" conditionals to describe when you've got an answer. *"If I see X, I know I can return Y as an answer.* (There should be two.)

# READINGS	CURRENT READING NUMBER	CURRENT READING	DONE AFTER THIS READING?	RESULT
3,123,432	3,123,432	32.4	✅	safe
3,123,432	14	32.4	❌	👲
3,123,432	1,344	41.0	✅	unsafe
10	10	31.0	✅	safe
0	0	---	✅	safe (trivially)

- *If I see a temperature that's above 40, I know I can return "unsafe" as an answer.*
- *If I see the final temperature in the list, I can return the safety of that reading.*
  - (You would have stopped earlier if anything was already unsafe!)

DELEGATING

# READINGS	CURRENT READING NUMBER	CURRENT READING	DONE AFTER THIS READING?	RESULT
3,123,432	14	32.4	❌	🙇

"I got to this point because 1-13 were all safe. 14 appears to be safe. The food is safe if readings 15-3,123,432 are also all safe."

Think of recursive calls as being extremely lazy individuals working at a factory with infinite employees—if my job was to report about the safety of readings 14 through 3,123,432, then I'll just clear 14 myself and then ask some other person to do 15 through 3,123,432 and report their answer.

DELEGATING

If my "job" was:

```
Verify Safety of Readings (Fridge XYZ, Readings 14–end)
```

1. I assume readings 1-13 were fine
2. If reading 14 isn't fine, I'd just say so
3. If reading 14 is fine, I need someone else to do the rest...

then I'd make another "job" for someone else:

```
Verify Safety of Readings (Fridge XYZ, Readings 15–end)
```

...and just report what they say!

PUTTING IT TOGETHER

```python
def are_readings_safe(temperatures: list[float], idx: int) -> bool:
    if idx >= len(list):
        return True

    current_reading = temperatures[idx]
    if current_reading >= 40.0:
        return False

    return are_readings_safe(temperatures, idx + 1)
```

PRACTICE C2

We want to write the funcion `ping_pong(N, should_ping)` which prints "ping" and then "pong" in alternating order for a total of N prints.

```
def ping_pong(N: int, should_ping: bool):
    # TODO: do something here

```

`ping_pong(3, True)` prints: `ping`, `pong`, `ping`
`ping_pong(3, False)` prints: `pong`, `ping`, `pong`
`ping_pong(2, True)` prints: `ping`, `pong`
`ping_pong(2, False)` prints: `pong`, `ping`
`ping_pong(1, True)` prints: `ping`
`ping_pong(1, False)` prints: `pong`

PRACTICE:

We want to write a function `sum_numbers` that gets the sum of all integers in a list recursively. Answer these questions in **C3**, either in English or with code snippets.

1. What triggers the base case(s)? What should you return in that case?
2. What is the little bit of work done on each step? How do we combine information from this call with the result of the rest of the recursion?
3. What do we tell the function to do recursively? (List slicing may be helpful here.)

We'll take the answers above to build this solution...

```
def sum_numbers(nums_list):
    # TODO
```

PRACTICE L2

We'll take the answers above to build this solution...

```
def sum_numbers(nums_list):
    if nums_list == []:
        return _______

    first = nums_list[0] # the first element of the input list (an int)
    rest = nums_list[1:] # the rest of the input list (a list of ints)
    return ____________________________
```

PRACTICE:

Consider we want to write the function `remove_spaces(word)` that takes in a string and returns the same string without any vowels in it.

So `remove_spaces("Hello")` returns `"Hll"`

Before writing any code, think together then discuss:

- What is the base case? (Why is it the empty string? `""`)
- What is the little bit of work done on each step? How do we combine information from this call with the result of the rest of the recursion?
- What does our recursive call look like?

PRACTICE:

Finish writing this function:

```
def remove_spaces(word):
    # TODO: What do you put here?
```

PRACTICE

Consider we want to write the **recursive** function `find(lst, target)` that returns the position of `target` in the `lst` or `-1` if it's not present. We'll do a *linear search*, inspecting position 0, then 1, then 2, then 3, ... until we find what we're looking for or run out of positions!

Before writing any code:

- What is the base case(s)?
- What is the recursive case?
- What is the work done on each step?
- What additional information needs to be passed in to do a Linear Search/what's the signature of the helper function?

HELPER FUNCTIONS

Linear Search works by repeatedly shrinking the *feasible search area*.

- `idx` is the left-most/lowest index that might still hold the target
- want to keep searching until we find our target or `idx` is big enough that there's nothing left to search in the list!

PRACTICE:

Finish writing this function:

```
def find(lst: list[int], target: int) -> int:
    return find_helper(lst, target, ___________)
def find_helper(lst: list[int], target: int, idx: int) -> int:
    # TODO!
```

PRACTICE:

Consider we want to write the function `find_factors(N)` that returns a set containing all positive factors of the input integer `N`.

A number `x` is a factor of `N` if and only if `N % x == 0`

Before writing any code (L15)

- What is the base case?
- What is the recursive case?
- What is the work done on each step?

HELPER FUNCTIONS

Sometimes to do recrusion we need to remember a bit more information than is provdided to the overall problem.

In this case, what other information do we need for `find_factors(N)` to get a working recursive solution?
Why can't we just recursively call `find_factors(N-1)`?

PRACTICE:

Finish writing this function:

```
def find_factors_helper(current, N):
    # TODO: What do you put here?
    # need to return a set of all factors of N

def find_factors(N):
    return find_factors_helper(1, N)
```

```
def find_factors_helper(current, N):
    if current > N // 2:
        return set()

    if N % current == 0:
        return {current} | find_factors_helper(current + 1, N)
    else:
        return find_factors_helper(current + 1, N)

def find_factors(N):
    return find_factors_helper(1, N)
```