Warmup (L22)

Finish the class `Rational`:

```
import math # math.gcd is greatest common divisor

class Rational:
    """
    Represents a rational number by its (integer) numerator and
    denominator, reduced.
    """
    num: int
    den: int

    def __init__(self, num: int, den: int) -> None:
        gcd = math.gcd(num, den)
        self.num = num // gcd
        self.den = den // gcd

    def __repr__(self) -> str:
        return f"{self.num}/{self.den}"

    def __add__(self, y):
        …
    def __sub__(self, y):
        …
    def __mul__(self, y):
        …
    def __truediv__(self, y):
        …
    def toFloat(self) -> float:
        …
```

Recursion

CS114 M9

When Python gets angry

- Many of you have seen Python get stuck and run forever (or for a while then crash)
- One common mistake that causes this:

```
def silly(x: int) -> int:
    return silly(x-1)
```

- This gets stuck because `silly` keeps calling itself forever, and can never get out

Recursion!

- A function calling itself *is* allowed…
- … but it has to be conditional, or it'll go on forever
- This technique is called *recursion*
- Recursion is no more powerful than looping, but sometimes it's clearer to express something with recursion

Recursion!

- No new syntax for recursion
- This is a new technique, but it’s just function calls as we’ve already seen them

First example: factorial

- Classic example of recursion: factorial
  - E.g., 5! = 5*4*3*2*1

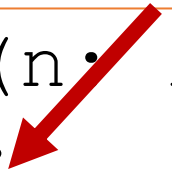
First example: factorial

- Classic example of recursion: factorial
  - E.g., 5! = 5*4*3*2*1

```
def factorial(n: int) -> int:
    if n <= 1:
        return 1
    else:
        return n * factorial(n-1)
```

First example: factorial

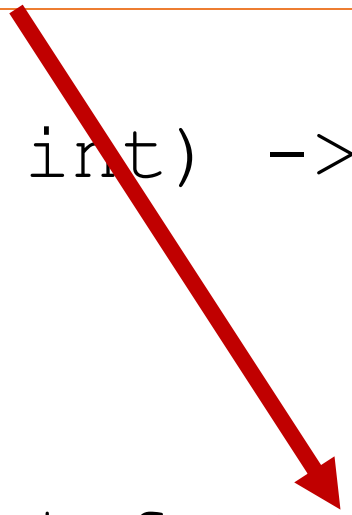
Base case: If n is 1 (or less), `factorial` does not call itself (does not recurse). This is important, because this is how `factorial` stops!



```
def factorial(n: int) -> int:
    if n <= 1:
        return 1
    else:
        return n * factorial(n-1)
```

First example: factorial

Recursive case: We build our definition of factorial(n) for any value of n greater than 1 by calling factorial with a smaller n



```
def factorial(n: int) -> int:
    if n <= 1:
        return 1
    else:
        return n * factorial(n-1)
```

Understanding recursion

- Let's add some prints to see where our code goes

```
def factorial(n: int) -> int:
    print("factorial started ({n})")
    if n <= 1:
        print(f"base case {n}")
        return 1
    else:
        print(f"recursive case {n} started")
        r = n * factorial(n-1)
        print(f"recursive case {n} finished")
        return r
```

Recursion vs. looping

- Recursion is no more powerful than looping! Here's factorial with loops (we developed this in Module 3):

```
def factorial(n: int) -> int:
    r = 1
    for i in range(n, 1, -1):
        r *= i
    return r
```

Recursion concepts

CS114 M9

Recursion in math

- Functions are often described recursively in math, not just programming
- Here's how a mathematician might describe factorial:

$$n! = \begin{cases} 1, & \text{if } n \leq 1 \\ (n-1)! \cdot n, & \text{if } n > 1 \end{cases}$$

Recursion concepts

- For a recursive function to be meaningful (in math or programming), it needs
  - At least one *base case*: Some case(s) where the result does not depend on a recursive call
  - At least one *recursive case*: Some case(s) where the result does depend on a recursive call
  - A correctly implemented *descent condition*: Each call to the recursion gets closer to (or "approaches") a base case

Back to factorial

```
def factorial(n: int) -> int:
    if n <= 1:
        return 1
    else:
        return n * factorial(n-1)
```

- Base case: $n \leq 1$. There is no recursive call when *n* is less than or equal to 1.

Back to factorial

```
def factorial(n: int) -> int:
    if n <= 1:
        return 1
    else:
        return n * factorial(n-1)
```

- Recursive case: $n > 1$. We didn't specifically write "`n>1`" in the code, but the **`else`** case only happens when that's true.

Back to factorial

```
def factorial(n: int) -> int:
    if n <= 1:
        return 1
    return n * factorial(n-1)
```

- The recursive case doesn't need to be part of an **`if`** or **`else`**! All that matters is what actually *happens*.

Back to factorial

```
def factorial(n: int) -> int:
    if n <= 1:
        return 1
    else:
        return n * factorial(n-1)
```

- The way to think about the recursive case is “Imagine we’ve already solved this for *n*-1. What’s the solution for *n* given that?”

Back to factorial

```
def factorial(n: int) -> int:
    if n <= 1:
        return 1
    else:
        return n * factorial(n-1)
```

- Descent condition: `n-1` is smaller than `n`, so will eventually be less than or equal to 1.

In-lecture quiz (L22)

- https://student.cs.uwaterloo.ca/~cs114/quiz/
- Q1: Are there any conditions under which this version of `factorial` would recurse forever?

```
def factorial(n: int) -> int:
    if n == 0 or n == 1:
        return 1
    else:
        return n * factorial(n-1)
```

A. This code crashes
B. Yes (but it doesn't otherwise crash)
C. No (and it doesn't crash either)

In-lecture quiz (L22)

- https://student.cs.uwaterloo.ca/~cs114/quiz/
- Q2: Are there any conditions under which this version of `factorial` would recurse forever?

```
def factorial(n: int) -> int:
    if n > 1:
        return n * factorial(n-1)
    else:
        return 1
```

A. This code crashes
B. Yes (but it doesn't otherwise crash)
C. No (and it doesn't crash either)

Divide and conquer

CS114 M9

Divide and conquer

- Most common reason why recursion is good: *divide and conquer*
- Many problems are done best by dividing the problem (in half or otherwise into subparts), then solving each subproblem
- We'll start with search

in

- How does **`in`** work with a list?
- It's just a loop:

```
def contains(lst: list, needle: typing.Any) -> bool:
    for val in lst:
        if val == needle:
            return True
    return False
```

- With nothing known about the list, this is the best we can do
- What if the list was sorted?

Binary search

- Base case: Empty list doesn't have it
- Descent condition: List gets smaller (approaches empty list)
- Recursive thought process: Look at an item in the middle. If it's what you're looking for, we're done. Otherwise, by comparing it to what you're looking for, you can choose which half the item you're looking for might be in. Assume that the search is already implemented for any smaller list, so for either half.

Binary search

```
def sortedSearch(lst: list, needle: typing.Any) -> bool:
    if len(lst) == 0:
        return False
    midpoint = len(lst) // 2
    if lst[midpoint] == needle:
        return True
    elif lst[midpoint] < needle:
        return sortedSearch(lst[midpoint+1:], needle)
    else: # lst[midpoint] > needle
        return sortedSearch(lst[:midpoint], needle)
```

Understanding binary search

Once again, let’s add some prints to understand what’s happening in our code

Is recursion always about lists?

- We’re about to see a bunch of examples with lists
- Some programming languages are designed for you to use recursion everywhere
- Python is not
- Divide-and-conquer is the usual reason for recursion in Python, which means we need something to divide
- So, recursion is usually for dividable lists

Insertion sort

CS114 M9

Insertion sort

- Keep a list sorted by inserting things in their correct location instead of appending to the end
- Like binary search, but instead of checking if the value is there, *put* it there
- Can’t slice the list, since that makes a new list; we want to insert the value!

Insertion sort

- Descent condition: Segment of list we're considering gets smaller (upperbound minus lowerbound approaches zero)
- Base case: upperbound == lowerbound (insert here)
- Recursive thought process: Look at a middle point. Insert either before or after that middle point. Assume inserting into a smaller range is done.

```
def insertSortedPrime(
    lst: list, val: typing.Any, lb: int, ub: int
) -> int:
    if ub == lb:
        lst.insert(lb, val)
        return lb
    midpoint = (ub - lb) // 2 + lb
    mid = lst[midpoint]
    if val < mid:
        return insertSortedPrime(lst, val, lb, midpoint)
    else:
        return insertSortedPrime(lst, val, midpoint+1, ub)

def insertSorted(lst: list, val: typing.Any) -> int:
    return insertSortedPrime(lst, val, 0, len(lst))
```

Merge sort

CS114 M9

How do you sort?

- How do `lst.sort` and `sorted` actually work?
- There are many algorithms, but we'll show you a classic one: merge sort
- Intuition: It's much easier to merge two lists that are already sorted than it is to sort a list

- This part isn't recursive

```
def merge(a: list, b: list) -> list:
    r = []
    while len(a) > 0 and len(b) > 0:
        if b[0] < a[0]:
            r.append(b[0])
            b.pop(0)
        else:
            r.append(a[0])
            a.pop(0)
    return r + a + b
```

Where's the recursion?

- Descent condition: List becomes smaller (approaches length 1 or empty)
- Base case: An empty list or a list of length 1 is sorted
- Recursive thought process: Assume we can already sort half the list. Sort both halves, then merge the sorted lists.

Merge sort

```
def mergeSort(lst: list) -> list:
    if len(lst) <= 1:
        return lst
    midpoint = len(lst) // 2
    l = mergeSort(lst[:midpoint])
    r = mergeSort(lst[midpoint:])
    return merge(l, r)
```

Module summary

CS114 M9

Module summary

- Recursion:
  - Base case
  - Recursive case
  - Descent condition
- Divide-and-conquer for lists
  - Binary search
  - Insertion sort
  - Merge sort