CS 131 Discussion

Week 2: pitter pattern

matt wang (he/him)
✉️ matt@matthewwang.me
🔗 mattxw.com/131slides
📣 mattxw.com/131feedback

Discussion Agenda

1. feedback from last week
2. warmup
3. hw 1 answers
4. algebraic data types
5. person of the week

~ break ~

6. language of the week
7. hw2 review
8. practice problems
    a. last week
    b. lambdas & closures

snacc of the week

Feedback / Iteration

Thank you for giving feedback!

Some things I'm working on this week:

- more hw review / practice problems (and better depth!)
- timebox person/lang of the week (pacing/timing)
- more (informal) questioning
- making note of 1:1 questions, posting later

Back of mind:

- Online OH
- PDF export for slides / notes – working with Carey
- project walkthrough (incl setup, tooling) - next week!
- ... memes

Other Data

Most excited for ...

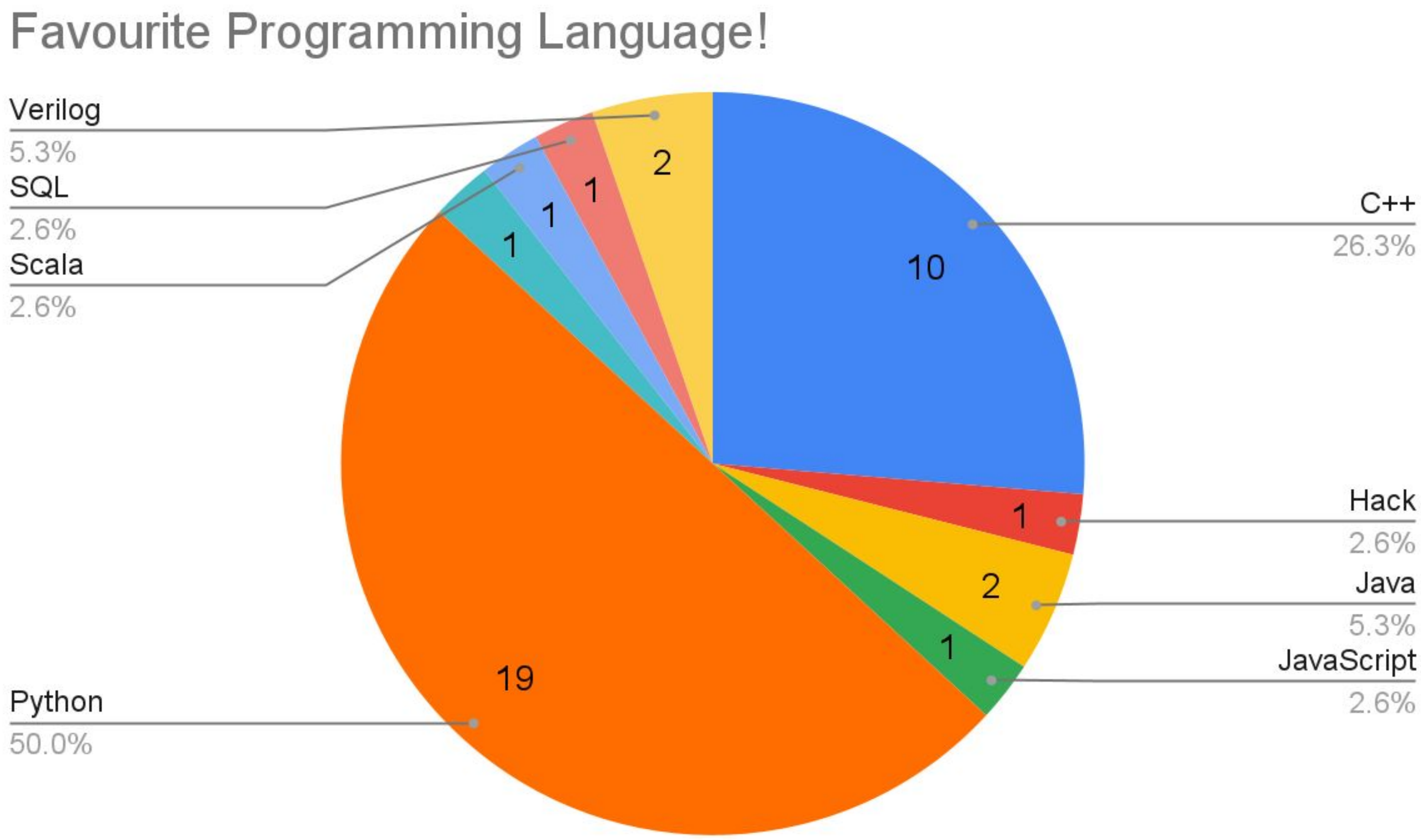
- learning programming languages!
- functional programming!
- **skill of learning a language quickly**
- depth & tradeoffs
- Carey! Matt (<3)!

(some just want to pass - which is reasonable!!)

Most nervous about ...

- **difficulty & workload**
  - note: very different from Eggert 131!!
  - I will do my best <3
- **tests**
  - challenge problems!
- **scary projects**
  - action item: I'll spend significant time breaking the project down!
  - there's no CS 32 P4 :)
- **edge cases**
  - my goal: **build your intuition**

Also: **non-class things** (internships, life, etc.)


Favourite Programming Language!
Verilog
5.3%
SQL
2.6%
Scala
2.6%
C++
26.3%
Hack
2.6%
Java
5.3%
JavaScript
2.6%
Python
50.0%
2
1
1
1
10
1
2
1
19

warmup

how would we reverse a list?

```
reverse' ::
```

First, what’s the type?

how would we reverse a list?

```
reverse' :: [a] -> [a]
```

Next, what’s the trivial case?

how would we reverse a list?

```
reverse' :: [a] -> [a]
reverse' [] = []
```

Okay, and the recursive step?

how would we reverse a list?

```haskell
reverse' :: [a] -> [a]
reverse' [] = []
reverse' (x:xs) = reverse' xs ++ [x]
```

Perfect! What’s the complexity?

how would we reverse a list?

```
reverse' :: [a] -> [a]
reverse' [] = []
reverse' (x:xs) = reverse' xs ++ [x]
```

Oh no … **O(n^2) …** since ++ is O(n)!

how would we reverse a list in O(n) time?

how would we reverse a list in O(n) time?

```
reverse l =  rev l []
  where
    rev []     a = a
    rev (x:xs) a = rev xs (x:a)
```

This is using a **helper function** with an **accumulator!**

how is this done in the source code?

```
rev = foldl (flip (:)) []
```

HUH??????????

how is this done in the source code?

```
rev = foldl (flip (:)) []
```

well, actually ...

- Flip simply takes a function and returns a function that is like our original function, only the first two arguments are flipped. (LYAH)

how is this done in the source code?

```
rev = foldl (flip (:)) []
```

well, actually …

- so, flip (:) is a append-at-end!

how is this done in the source code?

```
rev = foldl (flip (:)) []
```

well, actually …

- so, flip (:) is a append-at-end!
- then, we just foldl
    - because – left to right :)

how is this done in the source code?

```
rev = foldl (flip (:)) []
```

well, actually ...

- so, flip (:) is a append-at-end!
- then, we just foldl
- finally, we partially apply foldl!

last week’s homework!

question 1

Write a Haskell function named `largest` that takes in 2 `String` arguments and returns the longer of the two. If they are the same length, return the first argument.

Example:

`largest “cat” “banana”` should return `“banana”`.

`largest “Carey” “rocks”` should return `“Carey”`.

```haskell
largest :: String -> String -> String
largest first second =
    if length first >= length second
                       then first
                       else second
```

question 2

Barry Snatchenberg is an aspiring Haskell programmer. He wrote a function named `reflect` that takes in an `Integer` and returns that same `Integer`, but he wrote it in a very funny way:

```haskell
reflect :: Integer -> Integer
reflect 0 = 0
reflect num
  | num < 0 = (-1) + reflect num+1
  | num > 0 = 1 + reflect num-1
```

He finds that when he runs his code, it always causes a stack overflow (infinite recursion) for any non-zero argument! What is wrong with Barry's code (i.e. can you fix it so that it works properly)?

```haskell
reflect :: Integer -> Integer
reflect 0 = 0
reflect num
    | num < 0 = (-1) + reflect num+1
    | num > 0 = 1 + reflect num-1
```

```haskell
reflect :: Integer -> Integer
reflect 0 = 0
reflect num
    | num < 0 = (-1) + reflect (num+1)
    | num > 0 = 1 + reflect (num-1)
```

question 3a

Write a Haskell function named `all_factors` that takes in an `Integer` argument and returns a list containing, in ascending order, all factors of that integer. You may assume that the argument is always positive. **Your function's implementation should be a single, one-line list comprehension.**

Example:
`all_factors 1` should return `[1]`.
`all_factors 42` should return `[1, 2, 3, 6, 7, 14, 21, 42]`.

```haskell
all_factors :: Integer -> [Integer]
all_factors num =
  [x | x <- [1..num], num `mod` x == 0]
```

question 3b

A perfect number is defined as a positive integer that is equal to the sum of its proper divisors (where "proper divisors" refers to all of its positive whole number factors, excluding itself). For example, 6 is a perfect number because its proper divisors are 1, 2 and 3 and 1 + 2 + 3 = 6.

Using the `all_factors` function, write a Haskell expression named `perfect_numbers` whose value is a **list comprehension** that generates an infinite list of all perfect numbers (even though it has not been proved yet whether there are infinitely many perfect numbers 😉).

Example:
`take 4 perfect_numbers` should return `[6, 28, 496, 8128]`.

Hint: You may find the init and sum functions useful.

```haskell
all_factors :: Integer -> [Integer]
all_factors num =
  [x | x <- [1..num], num `mod` x == 0]
```

```haskell
perfect_numbers :: [Integer]
perfect_numbers =
  [x | x <- [1..], sum (init (all_factors x)) == x]
```

question 4

Write a pair of Haskell functions named `is_odd` and `is_even` that each take in 1 `Integer` argument and return a `Bool` indicating whether the integer is odd or even respectively. You may assume that the argument is always positive.

You may not use any builtin arithmetic, bitwise or comparison operators (including `mod`, `rem` and `div`). You may only use the addition and subtraction operators (+ and -) and the equality operator (==).

You must implement THREE versions of these functions: (1) with regular if statements, (2) using guards, and (3) using pattern matching.

Example:
`is_even 8` should return `True`.
`is_odd 8` should return `False`.

Hint: The functions can call one another in their implementations. (This is called mutual recursion).

```haskell
-- with if statements
is_odd :: Integer -> Bool
is_odd x =
 if x == 0 then False else is_even (x-1)

is_even :: Integer -> Bool
is_even x =
 if x == 0 then True else is_odd (x-1)
```

```haskell
-- with guards
is_odd :: Integer -> Bool
is_odd x
  | x == 0 = False
  | otherwise = is_even (x-1)

is_even :: Integer -> Bool
is_even x
  | x == 0 = True
  | otherwise = is_odd (x-1)
```

```haskell
-- with pattern matching
is_odd :: Integer -> Bool
is_odd 0 = False
is_odd x = is_even (x-1)

is_even :: Integer -> Bool
is_even 0 = True
is_even x = is_odd (x-1)
```

bigger picture advice

- mutual recursion: it’s helpful if both functions have **the same base case**
- pattern matching, guards, etc. are syntactic sugar* - **use what you like!**

question 5

Write a function named `count_occurrences` that returns the number of ways that all elements of list a_1 appear in list a_2 in the same order (though a_1's items need not necessarily be consecutive in a_2). The empty sequence appears in another sequence of length n in 1 way, even if n is 0.

Examples:
`count_occurrences [10, 20, 40] [10, 50, 40, 20, 50, 40, 30]` should return 1.
`count_occurrences [10, 40, 30] [10, 50, 40, 20, 50, 40, 30]` should return 2.
`count_occurrences [20, 10, 40] [10, 50, 40, 20, 50, 40, 30]` should return `0`.
`count_occurrences [50, 40, 30] [10, 50, 40, 20, 50, 40, 30]` should return 3.
`count_occurrences [] [10, 50, 40, 20, 50, 40, 30]`
should return 1.
`count_occurrences [] []` should return 1.
`count_occurrences [5] []` should return `0`.

```haskell
count_occurrences :: [Integer] -> [Integer] -> Integer
count_occurrences [] _ = 1
count_occurrences _ [] = 0
count_occurrences (x:xs) (y:ys)
  | x == y = count_occurrences xs ys +
other_occurrences
  | otherwise = other_occurrences
  where other_occurrences = count_occurrences (x:xs) ys
```

bigger picture advice / common mistakes

- general recursion
  - your recursive step should be *as small as possible*
  - heuristic: recursive steps should be ~ O(1) for ~polynomial problems
- recursion & lists as inputs
  - the base cases are *almost* always the empty list
  - you typically do not need base cases of the form `[] []`

Common mistakes:

- +1 instead of another recursive call
- (perf) O(n) / O(n^2) recursive steps
- (nit) redundant base cases

~ algebraic data types ~

(matt is switching to Carey's slides – 139 - 154)

person of the week

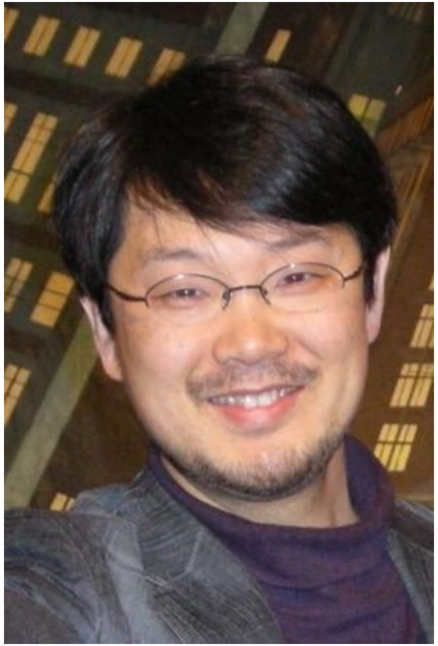
🥁🥁🥁🥁🥁

Yukihiro Matsumoto / “Matz” (1965-)

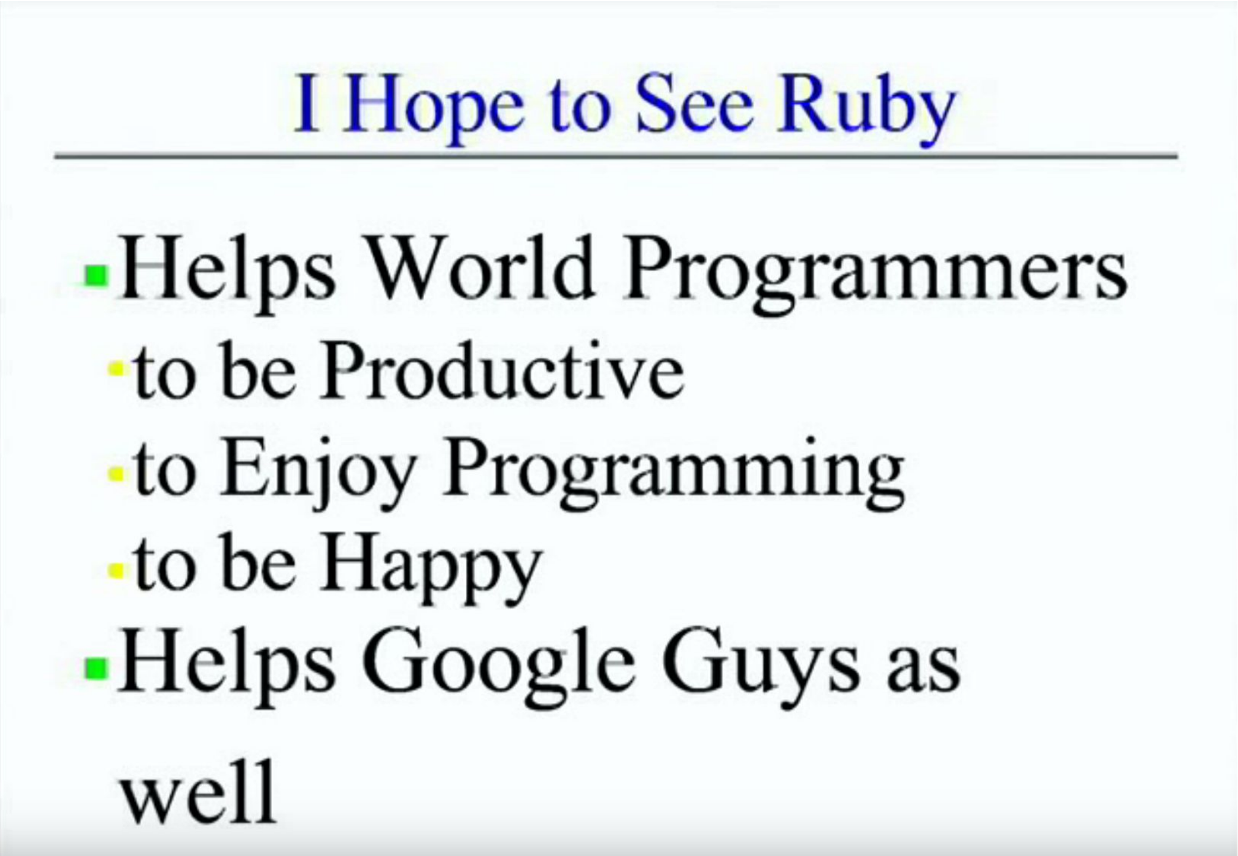
- Born and raised in Japan (Osaka, Tottori), Tsubaka Uni
- Released Ruby v1 in 1995
- Works for Netlab, Heroku, and mostly just on Ruby!

Why Matz?

- Few non-Western major programming language creators
    - This meant that Ruby considered internationalization from the start (and it shows)!
- Has a uniquely positive / “enjoyment”-based view of coding
    - Manifested itself in the language design, community
    - “Matz is nice and so we are nice”



Matz at the 2007 ICPC!
Wikipedia, 2007.



A slide from Matz’s tech talk at Google, YouTube, 2007.

~ break ~

discussion will resume at 11:03

language of the week

🥁🥁🥁🥁🥁

JavaScript

"*Technically it's actually ECMAScript*" smh

A little about JS:

- dynamically typed, ~ strong & weak
- no single standard implementation
  - Chrome versus Firefox, V8, Deno, Node, Bun...
- "JS was built in 10 days" (not *entirely* accurate)
- traces lineage to Scheme (Lisp), Self (Smalltalk), Java

Matt's most-used language (~200k sloc).

Why is JS interesting?

- It powers the web! **You will probably use it in your job.**
- Significant differences in implementation!
- It is arguably the *most mainstream functional language!*

FP in JavaScript (and React)

```
const listItems = products.map(product =>
  <li key={product.id}>
    {product.title}
  </li>
);
```

FP in JavaScript (and React) - **Map and Lambdas!**

```
const listItems = products.map(product =>
  <li key={product.id}>
    {product.title}
  </li>
);
```

FP in JavaScript

```
const dates = [
  '2019/06/01', '2018/06/01', '2019/09/01', '2018/09/01'
].map(v => new Date(v));


const maxDate = dates.reduce(
    (max, d) => d > max ? d : max,
    dates[0]
);
```

FP in JavaScript - Reducers (~ fold)!

```
const dates = [...].map(v => new Date(v));



// reducers don’t always have to sum!

const maxDate = dates.reduce(
    (max, d) => d > max ? d : max, // ternary operator
    dates[0]
);
```

FP in JavaScript (and React)

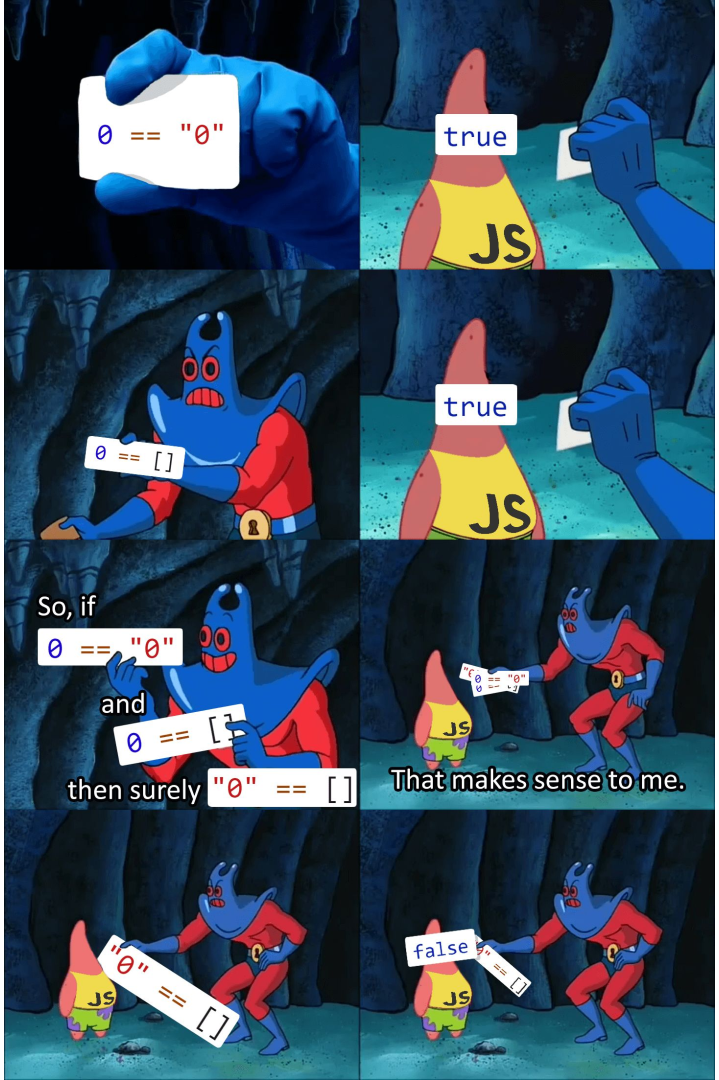
```
function MyButton() {
 const [count, setCount] = useState(0);

 function handleClick() {
   setCount(count + 1);
 }

 return (
   <button onClick={handleClick}>
     Clicked {count} times
   </button>
 );
}
```

FP in JavaScript (and React) - **Closures / Args!**

```
function MyButton() {
 const [count, setCount] = useState(0);

 function handleClick() {
   setCount(count + 1);
 }

 return (
   <button onClick={handleClick}>
     Clicked {count} times
   </button>
 );
}
```

FP in JavaScript (and React) - **Closures / Args!**

```
function MyButton() {
 const [count, setCount] = useState(0);

 function handleClick() {
   setCount(count + 1);
 }

 return (
   <button onClick={handleClick}>
     Clicked {count} times
   </button>
 );
}
```


0 == "0"
true
0 == []
true
So, if
0 == "0"
and
0 == []
then surely "0" == []
That makes sense to me.
"0" == []
false

overview: this week’s HW!

- higher-order functions
  - map, filter, fold
  - lambdas
- currying & partial application
- algebraic data types and “classes”

Week 2 Problems

what is the difference between `foldl` and `foldr`?

… what do y'all think?

foldl is left-associative, e.g.:

$f(\ f(\ f(\ f(\ f(\text{accum}, x_1), x_2), x_3), x_4), x_n)$

foldr is right-associative, e.g.:

$f(x_1,\ f(x_2,\ f(x_3, f(x_4, f(x_n, \text{accum})))))$

e.g., for subtraction:

$(\ ((((\text{accum} - x_1) - x_2) - x_3) - x_4) - x_n)$

e.g., for subtraction:

$(x_1 - (x_2 - (x_3 - (x_4 - (x_n - \text{accum})))))$

map + filter

takes an array, and only returns the squares of the even numbers

```
mf :: [Integer] -> [Integer]
```

map + filter

takes an array, and only returns the squares of the even numbers

```
mf :: [Integer] -> [Integer]
mf l = map
   (\x -> x*x)
   (filter (\x -> (mod x 2) == 0) l)
```

tree sum

traverse this tree ADT, and sum all the nodes

```
data Tree = Empty | Node Integer [Tree]

tsum :: Tree -> Integer
```

tree sum

traverse this tree ADT, and sum all the nodes

```
-- example
tsum (Node 3 [
  (Node 2 [Node 7 []]),
  (Node 5 [Node 4 []])
]) == 21
```

tree sum

traverse this tree ADT, and sum all the nodes

```
data Tree = Empty | Node Integer [Tree]

tsum :: Tree -> Integer
tsum Empty = 0
```

tree sum

traverse this tree ADT, and sum all the nodes

```
data Tree = Empty | Node Integer [Tree]

tsum :: Tree -> Integer
tsum Empty = 0
tsum (Node val []) = val
```

tree sum

traverse this tree ADT, and sum all the nodes

```
data Tree = Empty | Node Integer [Tree]

tsum :: Tree -> Integer
tsum Empty = 0
tsum (Node val []) = val
tsum (Node val lst) = val +
                      sum (map tsum lst)
```

Week 1 Problems

warmup: naive fibonacci, but ... wrong?

```
fib :: (Int n) => n -> n
fib 1 = 1
fib n = fib(n-1) + fib(n-2)
```

this breaks ... but how, why, and how do we fix it?

warmup: naive fibonacci, but ... wrong?

```
fib :: (Int n) => n -> n
fib 1 = 1
fib n = fib(n-1) + fib(n-2)
```

hint: what happens with fib 2?

warmup: naive fibonacci, but … wrong?

```
fib :: (Int n) => n -> n
fib 1 = 1
fib 2 = 1
fib n = fib(n-1) + fib(n-2)
```

all good!

(last week) problem: do not comprehend

```
f :: Int -> Int -> [Int]
f n d = ...

-- equal to list comprehension
[x | x <- [1..n], (mod x d) == 0]

-- BUT… no comprehensions!!
```

solution – single function

```
f n d =
  if n == 0
    then []
    else f (n-1) d ++
      if (mod n d) == 0
        then [n]
        else []
```

solution – multiple functions

```
g lst d = if null lst
  then []
  else if (mod (head lst) d) == 0
    then (head lst) : g (tail lst) d
    else g (tail lst) d

f n d = g [1..n] d
```

solution – w/ tail recursion

```
g lst accum d = if null lst
  then accum
  else g (tail lst) (accum ++ (
    if (mod (head lst) d) == 0
      then [(head lst)] else []
  )) d
f n d = g [1..n] [] d
```

challenge problem: stack overflow

```
fib :: (Int n) => n -> n
fib 1 = 1
fib 2 = 1
fib n = fib(n-1) + fib(n-2)
```

I lied – not all good, and in particular, not performant.

Why? How would we implement this?

(hint: CS 33)

~ post-discussion survey ~

always appreciate the feedback!

- was review / problems helpful?
- was pacing / extra content good?

see you next week <3

https://forms.gle/33gPkKDfajrrQrZ88

appendix

challenge problem solution(s)

challenge problem: stack overflow

```
fib :: (Int n) => n -> n
fib 1 = 1
fib 2 = 1
fib n = fib(n-1) + fib(n-2)
```

Two core problems:

- fib is not memoized/cached - exponential blowup
- recursive call stack explodes / lack of tail recursion

challenge problem: stack overflow

```
fib :: (Int n) => n -> n
fib 1 = 1
fib 2 = 1
fib n = fib(n-1) + fib(n-2)
```

Resolving memoization:

use a helper accumulator / HashSet (out of scope for now).

challenge problem: stack overflow

```
fib :: (Int n) => n -> n
fib 1 = 1
fib 2 = 1
fib n = fib(n-1) + fib(n-2)
```

Resolving tail recursion:

Tail recursion optimization (ghc) + rewrite to be tail recursive!

one fib implementation

```
fibAux n result previous
| n == 0 = result
| otherwise =
    fibAux (n − 1) ( result + previous ) result

fib n
| n == 0 = 0
| otherwise = fibAux n 1 0
```