61A FINAL REVIEW

Fall 2015

Topics to be covered:

- SQL
- Tail recursion
- Interpreters
- Streams
- Orders of Growth
- Iterators and Generators
- Recursion
- Miscellaneous tips
- Where to go from here

But first… some motivation

- One of my favorite quotes: “*Studying doesn’t suck nearly as much as failing.*”

- All you can do is your best :)
- But your best doesn’t start tomorrow, or even in 25 minutes… it starts now
- Stay focused, guys! If I’m boring, leave and go be focused wherever you can

Knowledge first, then practise

That is all.

SQL: The Rundown

- SQL is a **query language for databases**.
  - Fact: there is information stored in databases. In this class, we learned how to **retrieve** and **add** to that information using SQL.
- In SQL, we store information as **records** in **tables**.
  - An example of a single **record**:

Latitude	Longitude	Name
38	122	Berkeley

An example of a **table** containing three **records**:

Latitude	Longitude	Name
38	122	Berkeley
42	71	Cambridge
45	93	Minneapolis

SQL, *continued*

- Your main focus should be on information **retrieval**. For that, we'll write SQL queries. These queries all follow a similar structure:

  - `select [column_name(s)] from [table_name(s)] where [condition(s)] group by [column_name(s)] having [condition(s)] order by [column_name(s)]`

- Note that everything is optional except for `select` and `from`.
- You can also make and use temporary tables by using a `with` clause. We'll get to that in a moment.

SQL, *continued*

Here's a quick breakdown of that query structure:

- `select [column_name(s)]`: "we want *these* columns as output..."
- `from [table_name(s)]`: "...from *these* tables..."
- `where [condition(s)]`: "...but only the ones that fulfill *these* conditions."
- `group by [column_name(s)]`: "Okay, now aggregate the results into groups. Every record within a group will have the same value in *these* columns..."
- `having [condition(s)]`: "...but wait! Only give us groups that satisfy *these* conditions."
- `order by [column_name(s)]`: "And to finish it off, order the results in ascending order of *this* expression."

SQL, *continued*

And of course we can also create intermediate tables via the `with` clause. This is best illustrated through example:

```
with
     professors(name, age, dept, rating) as (
          select “DeNero”, 26, “CS”, 9.8 union
          select “Hug”, 25, “CS”, 9.7 union
          select “Hilfinger”, 70, “pain”, 9.6
     )
select name, age from professors where dept = “CS”;
```

What does this output? (This should not be very hard.)

SQL, *continued*

Here's a *real* example (with recursion).

>>> Problem statement:
Select all **odd** Fibonacci numbers less than 200, ordered from high to low.

How would you go about this?

SQL, *continued*

Here's a *real* example (with recursion).

>>> Problem statement:
Select all **odd** Fibonacci numbers less than 200, ordered from high to low.

How would you go about this?

Well, let's break it up. The first thing we need to do is figure out how to make Fibonacci numbers in SQL. Sounds like recursion to me!

SQL, *continued*

>>> Problem statement:
Select all **odd** Fibonacci numbers less than 200, ordered from high to low.

First, let's just generate some Fibonacci numbers. We know that we only want odd numbers, so we'll start with 1:

```
with fibonacci(num, prev) as (
    select 1, 0 union
    select num + prev, num from fibonacci where num + prev < 200
) select num from fibonacci;
```

SQL, *continued*

>>> Problem statement:
Select all **odd** Fibonacci numbers less than 200, ordered from high to low.

Seems pretty good, right? We've got all of the Fibonacci numbers less than 200, starting with 1. Now let's make sure we only keep the odd numbers:

```
with fibonacci(num, prev) as (
    select 1, 0 union
    select num + prev, num from fibonacci where num + prev < 200
) select num from fibonacci where num % 2 != 0;
```

SQL, *continued*

>>> Problem statement:
Select all **odd** Fibonacci numbers less than 200, ordered from high to low.

The last thing to do is order them from high to low. You've probably seen this before:

```
with fibonacci(num, prev) as (
    select 1, 0 union
    select num + prev, num from fibonacci where num + prev < 200
) select num from fibonacci where num % 2 != 0 order by -num;
```

...aaand that should do it.

SQL: A Problem for You

We have the following schema:

```
Songs (name, artist, album, year_released)
```

Select the **names** of all artists who have released more than 2 albums in a single year. (*Assume for the sake of the question that we live in a world where artists actually release more than 2 albums per year.*)

SQL: A Problem for You

Our schema:

```
Songs (name, artist, album, year_released)
```

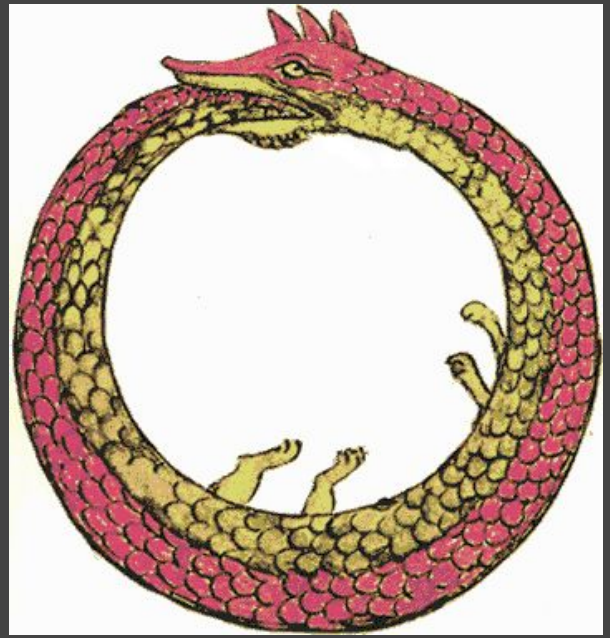
Our objective:

To select the name of any artist who's released more than two albums in one year.

Our solution:

```
with Artists(name, album, year_released, num_albums) as (
     select artist, album, year_released, 1 from Songs union
     select artist, a.album, a.year_released, num_albums + 1
          from Songs as s, Artists as a
          where s.year_released = a.year_released and a.album > s.album
) select name from Artists where num_albums > 2 group by name;
```

Tail recursion



Tail recursion

- A technique that allows us to execute **recursive** programs in **constant** [O(1)] space. That means **no stack overflows**!

Tail recursion, *continued*

Basically, we make the recursive call the absolute **last** computation to happen in the function body. Then, once we've made that recursive call we don't care about **any** data in the frame that made the call.

So we can just throw that frame out. And every other frame that isn't the last one.

Constant space!

Tail recursion, *continued*

As a simple example, let's compute the sum of the first **n** Fibonacci numbers. Here's a solution that's not optimized for tail recursion:

```
def fib_sum(n):
    """Returns the sum of the first n Fibonacci numbers."""
    def helper(n, prev, curr):
        if n <= 1:
            return prev
        else:
            return prev + helper(n - 1, curr, prev + curr)
    return helper(n, 0, 1)
```

Tail recursion, *continued*

Computing the sum of the first **n** Fibonacci numbers:
here's a solution that *is* optimized for tail recursion –

```
def fib_sum(n):
    """Returns the sum of the first n Fibonacci numbers."""
    def helper(n, prev, curr, total):
        if n <= 1:
            return total + prev
        else:
            return helper(n - 1, curr, prev + curr, total + prev)
    return helper(n, 0, 1, 0)
```

Tail recursion: someone'd better call Houston

Summer 2014 Final:

Write `waldo-tail`, a *tail-recursive* version of the `wheres-waldo` function you saw on Midterm 2. As a reminder, `waldo-tail` is a Scheme procedure that takes in a Scheme list and outputs the index of `waldo` if the symbol `waldo` exists in the list. Otherwise, it outputs the symbol `nowhere`.

```
STk> (waldo-tail '(moe larry waldo curly))
2
STk> (waldo-tail '(1 2))
nowhere

(define (waldo-tail lst)

  (define (helper ____________________________________)

    (cond ((_________________________________________) 'nowhere) ; Base Case

          (_____________________________________________________) ; Base Case

          (else _____________________________________________)))

  (helper _____________________________________________))
```

Tail recursion [Summer 2014 #4]

Solution:

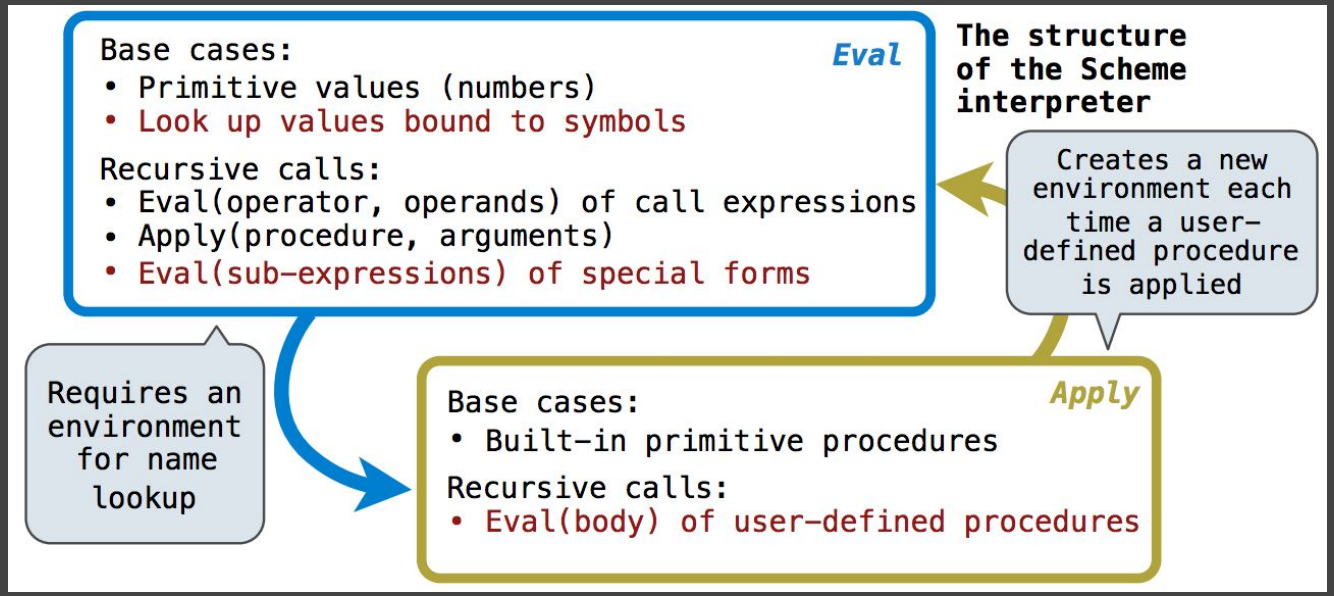
Write `waldo-tail`, a *tail-recursive* version of the `wheres-waldo` function you saw on Midterm 2. As a reminder, `waldo-tail` is a Scheme procedure that takes in a Scheme list and outputs the index of `waldo` if the symbol `waldo` exists in the list. Otherwise, it outputs the symbol `nowhere`.

```
STk> (waldo-tail '(moe larry waldo curly))
2
STk> (waldo-tail '(1 2))
nowhere

(define (waldo-tail lst)
  (define (helper lst index)
    (cond ((null? lst) 'nowhere)
          ((equal? 'waldo (car lst)) index)
          (else (helper (cdr lst) (+ index 1)))))
  (helper lst 0))
```

Interpreters

Interpreters **read in** and **evaluate / execute** your code. Hopefully, you got a sense of how these work via the Scheme project.



Interpreting Interpreters

(4 pt) Assume the following definition has been loaded into the Scheme interpreter from Project 4:

```
(define (sum-of-squares x y z)
  (+ (* x x) (* y y) (* z z)))
```

Given the following Scheme expressions, circle the correct number of calls to `scheme_eval` and `scheme_apply`:

```
(+ 5 (* 3 7 3))
```

`scheme_eval`	3	4	5	6	7	8	9
`scheme_apply`	1	2	3	4	5	6	

```
(sum-of-squares 3 4 5)
```

`scheme_eval`	4	5	8	10	14	19	24	25
`scheme_apply`	1	2	3	4	5	6		

Solutions!

```
(+ 5 (* 3 7 3))
```

Calls to scheme_eval [8]:

```
(+ 5 (* 3 7 3))
+
5
(* 3 7 3)
*
3
7
3
```

Calls to scheme_apply [2]:

```
(* 3 7 3)
(+ 5 (* 3 7 3))
```

Solutions!, pt. 2

```
(sum-of-squares 3 4 5)
```

Calls to scheme_eval [19]:

```
(sum-of-squares 3 4 5)
sum-of-squares
3
4
5
(+ (* x x) (* y y) (* z z))
+
(* x x)
*
x
x
(* y y)
*
y
y
(* z z)
*
z
z
```

Calls to scheme_apply [5]:

```
(sum-of-squares 3 4 5)
(+ (* x x) (* y y) (* z z))
(* x x)
(* y y)
(* z z)
```

Streams

Streams

are lazily computed linked lists.

- In other words, you only actually compute the elements of a stream when they’re requested.
- Streams can be defined in terms of themselves if you declare what the 1st element is, and you can define a way of computing the rest of the stream.

With that said, let’s move on to an example...

Wait! Just kidding

Before the example, here's a quick review of `cons-stream` and `cdr-stream`:

- `cons-stream` creates a stream. Elements will not be computed until someone asks for them.
- `cdr-stream` evaluates and returns the "rest" of a stream – one element forward – but stops there.

Streams: An Example

Let’s define a stream for Fibonacci numbers. (All of them!) However, we’ll require that you use `add_streams`, which adds the corresponding elements of two streams:

```
(define (add-streams s t)
  (cons-stream (+ (car s) (car t))
               (add-streams (cdr-stream s)
                            (cdr-stream t))))
```

Template:

```
(define (fib-stream)
    ‘YOUR-CODE-HERE
)
```

Streams, *continued*

Solution:

```
(define (fib-stream)
    (cons-stream 0
                 (cons-stream 1
                              (add-streams (fib-stream)
                                           (cdr-stream (fib-stream))
                              )
                 )
    )
)
```

Orders of Growth

- A way to measure time/space efficiency.
- Look at how resource usage scales as the problem size gets **really big**.

- If I add one unit to the problem size, does the amount of time consumed...
  - stay the same? Most likely O(1).
  - increase by some constant factor? Most likely O(n).
  - double? triple? multiply at all? Most likely $O(p^n)$.
- If I multiply the problem size by some factor, does the amt. of time used…
  - increase by some additive factor? Most likely O(log(n)).
  - scale by the square of the factor? Most likely $O(n^2)$.

Orders of Growth, *continued*

Look out for tricks. They like those.

Examples:

- nested `for`-loops that always terminate after a constant amount of time.
- seemingly exponential-time functions (i.e. every call spawns two recursive calls) where every time the problem size is cut in half.

Orders of Growth, *continued*

Sample problem:

```
def fizzle(n):
    if n <= 0:
        return n
    elif n % 23 == 0:
        return n
    return fizzle(n - 1)
```

What is the time complexity of this function?

Orders of Growth: Problem #2

```
def carpe_diem(n):
    if n <= 1:
        return n
    return carpe_diem(n - 1) + carpe_diem(n - 2)

def yolo(n):
    if n <= 1:
        return 5
    sum = 0
    for i in range(n):
        sum += carpe_diem(n)
    return sum + yolo(n - 1)
```

What is the order of growth in n of the runtime of `yolo`, where n is its input?

Problem #2 Solution: $O(n^2 \cdot 2^n)$

```
def carpe_diem(n):
    if n <= 1:
        return n
    return carpe_diem(n - 1) + carpe_diem(n - 2)

def yolo(n):
    if n <= 1:
        return 5
    sum = 0
    for i in range(n):
        sum += carpe_diem(n)
    return sum + yolo(n - 1)
```

What is the order of growth in n of the runtime of `yolo`, where n is its input?

Iterators and Generators

- **Iterator:** tracks progress through sequential data (has a `__next__` method)
- **Iterable:** represents the data itself (has an `__iter__` method)
- **Generator:** a special iterator created by calling a generator function
- **Generator function:** a function with `yield` statements. Generator functions are iterable! You get a new generator every time you invoke the function.

Let’s get to it...

The following are the first six rows of *Pascal's triangle*:

```
     1
    1 1
   1 2 1
  1 3 3 1
 1 4 6 4 1
1 5 10 10 5 1
```

The first and last element in each row is 1, and each of the other elements is equal to the sum of the element above it and to the left and the element above it and to the right. For example, the third element in the last row is $4 + 6 = 10$, since 4 and 6 are the elements above it and to the left and right.

```
def pascal_gen(n):
    """Return an iterator over all the elements in the nth row of
    Pascal's triangle.

    >>> list(pascal_gen(5))
    [1, 5, 10, 10, 5, 1]
    """

    if ____________________________________________________________:

        last = 0

        for num in ____________________________________________:

            yield ____________________________________________

            ____________________________________________________

    yield 1
```

pascal_gen

```
def pascal_gen(n):
    if n != 0:
        last = 0
        for num in pascal_gen(n - 1)
            yield last + num
            last = num
    yield 1
```

Recursion

Remember: recursion is just fancy iteration. In a convenient form.
Approach: How to solve a recursive problem.

- Identify the base case(s). Ask yourself: what is the **simplest** or **most basic** case?
- Figure out how you can work toward that base case in order to solve the problem.
- Put that into code.

Programming Questions

Approach: How to solve **any** programming problem.

- Figure out how you'd do it IRL, or just by thinking about it. (Do not worry about putting it into code. Just figure out how you'd do it. Make sure you really understand the problem.)
- Consider trying to visualize it, by writing stuff down or drawing things.
- Work at simple examples.
- Translate your thoughts into code. If you can do it *without* code, then you can do it *with* code too.

A recursive programming question

A binary tree is *balanced* if for every node, the depth of its left subtree differs by at most 1 from the depth of its right subtree. Fill in the definition of the `is_balanced` function below to determine whether or not a binary tree is balanced. You may assume that the `depth` function works correctly for this part.

```
def is_balanced(tree):
    """Determine whether or not a binary tree is balanced.

    >>> t = Tree(6, Tree(2, Tree(1)), Tree(7))
    >>> is_balanced(t)
    True
    >>> t.left.right = Tree(4, Tree(3), Tree(5))
    >>> is_balanced(t)
    False
    """
```

Solving the problem

- We have a `depth` function that will give us the depth of the left and right subtrees.
- We want to make sure that these values don't differ by more than 1.
- This condition is simply
  - `(-1 <= depth(tree.left) - depth(tree.right) <= 1)`
  - ...or, if you prefer, `(abs(depth(tree.left) - depth(tree.right)) <= 1)`
- We also want to make sure that the subtrees themselves are balanced, since "every node" must be balanced.
  - Recursive call!
  - What's the base case? **No tree.**

The solution

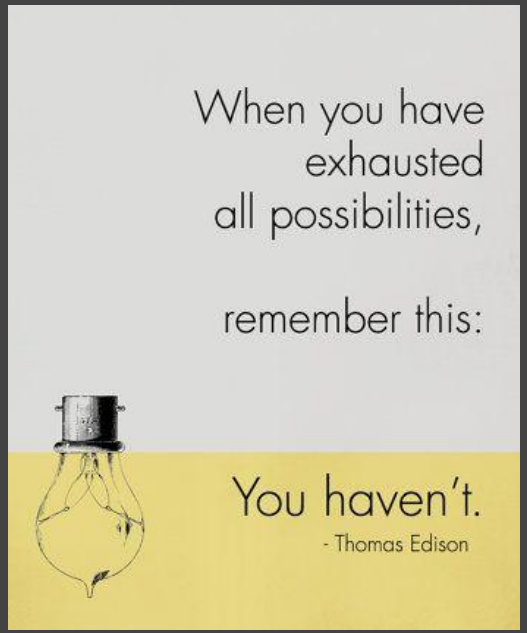
```
def is_balanced(tree):
    """Determine whether or not a binary tree is balanced.

    >>> t = Tree(6, Tree(2, Tree(1)), Tree(7))
    >>> is_balanced(t)
    True
    >>> t.left.right = Tree(4, Tree(3), Tree(5))
    >>> is_balanced(t)
    False
    """
    if not tree:
        return True
    return (is_balanced(tree.left) and is_balanced(tree.right) and
            -1 <= depth(tree.left) - depth(tree.right) <= 1)
```

Miscellaneous tips (just the standard stuff)

- Focus first on the questions you feel the best about.
- Don’t panic... or think you can’t do something… or shut down when it comes to any one problem. You can figure it out.
    - Just start writing things. Write what you know; think about the question. Key word: **think**. Lightbulbs *do* appear over people’s heads, didn’t you know?

- Make sure you enjoy your break after the final is over!



Where to go from here

- You have **23 hours** until your final begins.

- Unless you have significant holes in your conceptual knowledge, you'll definitely want to get sleep tonight. Preferably 7-8 hours!
- You'll also want to eat (well).
- And, you know, take a bit of time to relax.

- 23 hours is a lot of time… if you spend it wisely ;)

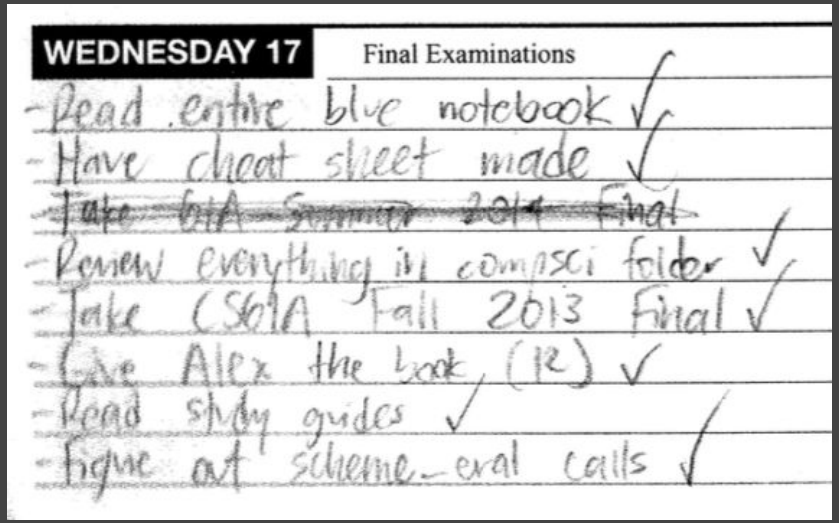
Where to go, *continued*

If you don't feel confident about concepts:

- Read through notes. Review lecture slides. Use all the resources at your disposal (friends, videos, Google, the textbook) to **get** confident.

If you **do** feel sufficiently confident:

- The best thing to do is to practice. Take past finals under exam settings, or go through any discussion worksheets you haven't done.



Good luck! :)

I believe in you.