SQL AND FINAL REVIEW

COMPUTER SCIENCE MENTORS CS 61A

April 29 to May 1, 2019

1 Creating Tables, Querying Data

Examine the table, `mentors`, depicted below.

Name	Food	Color	Editor	Language
Jade	Thai	Purple	Notepad++	Java
Evan	Pie	Green	Sublime	Java
Jack	Sushi	Orange	Emacs	Ruby
Kevin	Tacos	Blue	Vim	Python
Jemmy	Ramen	Green	Vim	Python

1. Create a new table `mentors` that contains all the information above. (You only have to write out the first two rows.)

Solution:
```
create table mentors as
  select 'Catherine' as name, 'Thai' as food, 'Purple' as
     color, 'Notepad++' as editor, 'Java' as language union
  select 'Lindsay', 'Pie', 'Green', 'Sublime', 'Java' union
  select 'Brandon', 'Sushi', 'Orange', 'Emacs', 'Ruby'
     union
  select 'Shreya', 'Tacos', 'Blue', 'Vim', 'Python' union
  select 'Keon', 'Ramen', 'Green', 'Vim', 'Python';
```

2. Write a query that lists all the mentors along with their favorite food if their favorite color is green.

```
Lindsay|Pie
Keon|Ramen
```

Solution:

```
select name, food
  from mentors
  where color = 'Green';

-- With aliasing
select m.name, m.food
  from mentors as m
  where m.color = 'Green';
```

3. Write a query that lists the food and the color of every person whose favorite language is *not* Python.

```
Thai|Purple
Pie|Green
Sushi|Orange
```

Solution:

```
select food, color
  from mentors
  where language != 'Python';

-- With aliasing
select m.food, m.color
  from mentors as m
  where m.language <> 'Python';
```

4. Write a query that lists all the pairs of mentors who like the same language. (How can we make sure to remove duplicates?)

```
Catherine|Lindsay
Keon|Shreya
```

Solution:

```
select m1.name, m2.name
    from mentors as m1, mentors as m2
    where m1.language = m2.language and m1.name < m2.name;
```

2 Aggregation

CS 61A wants to start a fish hatchery, and we need your help to analyze the data we've collected for the fish populations! Running a hatchery is expensive – we'd like to make some money on the side by selling some seafood (only older fish of course) to make delicious sushi.

The table `fish` contains a subset of the data that has been collected. The SQL column names are listed in brackets.
Table name: `fish`*

Species [species]	**Population [pop]**	**Breeding Rate [rate]**	**$/piece [price]**	**# of pieces per fish [pieces]**
Salmon	500	3.3	4	30
Eel	100	1.3	4	15
Yellowtail	700	2.0	3	30
Tuna	600	1.1	3	20

*(This was made with fake data, do not actually sell fish at these rates)

Hint: The aggregate functions `MAX`, `MIN`, `COUNT`, and `SUM` return the maximum, minimum, number, and sum of the values in a column. The `GROUP BY` clause of a select statement is used to partition rows into groups.

1. Write a query to find the three most populated fish species.

Solution:
```
select species from fish order by -pop LIMIT 3;
```

2. Write a query to find the total number of fish in the ocean. Additionally, include the number of species we summed. Your output should have the number of species and the total population.

Solution:
```
select COUNT(species), SUM(pop) from fish;
```

3. Profit is good, but more profit is better. Write a query to select the species that yields the most number of pieces for each price. Your output should include the species, price, and pieces.

Solution:
```
select species, price, MAX(pieces) from fish GROUP BY
  price;
```

The table `competitor` contains the competitor's `price` for each `species`.

Species [species]	**$/piece [price]**
Salmon	2
Eel	3.4
Yellowtail	3.2
Tuna	2.6

4. Business is good, but a bunch of competition has sprung up! Through some cunning corporate espionage, we have determined one such competitor's selling prices.

   Write a query that returns, for each species, the difference between our hatchery's revenue versus the competitor's revenue for one whole fish.

   This is because we make 30 pieces at $4 a piece for $120, whereas the competitor will make 30 pieces at $2 a piece for $60. Therefore, the difference is 60.

Solution:

```
select fish.species, (fish.price - competitor.price) *
   pieces
    from fish, competitor
    where fish.species = competitor.species;
```

FINAL REVIEW

3 Environment Diagrams

1. Draw the environment diagram that results from running the following code.

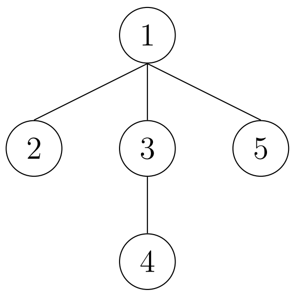
```
def f(f):
    def h(x, y):
        z = 4
        return lambda z:  (x + y) * z

    def g(y):
        nonlocal g, h
        g = lambda y: y[:4]
        h = lambda x, y: lambda f: f(x + y)
        return y[3] + y[5:8]

    return h(g("sarcasm!"), g("why?"))

f = f("61a")(2)
```

Solution: https://tinyurl.com/env-prog

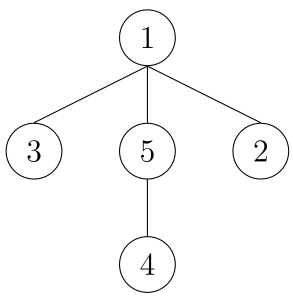
4 Recursive Data Structures

1. Implement `rotate`, which takes in a tree and rotates the labels at each level of the tree by one to the left destructively. This rotation should be modular (That is, the leftmost label at a level will become the rightmost label after running rotate). You do NOT need to rotate across different branches.

   For example, given the following tree,`t`

   calling rotate on `t` should mutate it to give us

   Fill in your implementation on the next page.

```
def rotate(t):
    """
    >>> t1 = Tree(1, [Tree(2), Tree(3, [Tree(4)]), Tree(5)])
    >>> rotate(t1)
    >>> t1
    Tree(1, [Tree(3), Tree(5, [Tree(4)]), Tree(2)])
    >>> t2 = Tree(1, [Tree(2, [Tree(3), Tree(4)]),
                      Tree(5, [Tree(6)])])
    >>> rotate(t2)
    >>> t2
    Tree(1, [Tree(5, [Tree(4), Tree(3)]),
                     Tree(2, [Tree(6)])])
    """

    branch_labels = ______________________________

    n = len(t.branches)

    for ______________________________________:

        branch = ________________________________

        _________________________________________

        _________________________________________
```

Solution:

```
def rotate(t):
    branch_labels = [b.label for b in t.branches]
    n = len(t.branches)
    for i in range(n):
        branch = t.branches(i)
        branch.label = branch_labels[(i + 1) \% n]
        rotate(branch)
```

2. Fill in the implementation of `shuffle`.

```
def shuffle(lnk):
    """Swaps each pair of items in a linked list.

    >>> shuffle(Link(1, Link(2, Link(3, Link(4)))))
    Link(2, Link(1, Link(4, Link(3))))
    >>> shuffle(Link('s', Link('c', Link(1, Link(6,
       Link('a'))))))
    Link('c', Link('s', Link(6, Link(1, Link('a')))))
    """
```

Solution:

```
    if lnk is Link.empty or lnk.rest is Link.empty:
        return lnk
    front = lnk.rest
    lnk.rest = shuffle(front.rest)
    front.rest = lnk
    return front
```

5 Recursion

1. Imagine we have a game where there are multiple cards with numbers laid out in a straight line. Each turn, a player can take a card from the very left, or the very right. We want to write a function, `game`, that determines whether or not it is possible for a sequence of valid moves to result in both players getting an equal score once all cards are used.
We represent the cards as a list. Say `lst = [0, 2, 2, 4]`. In this case, our function should return True. The current player (`curr`) can pick the last card (4) on their first turn. After that, the two players switch. On the next turn, the new current player (previously the opponent) can pick 2. Then the original player picks 0, and the other player picks 2.
`Game` takes in the scores of both players (initially 0) and a list representing the cards.

```
def game(cards):
    """
    >>> game([1,2,3]) #1, then 3, then 2
    True
    >>> game([1, 1, 1])
    False
    >>> game([])
    True
    """
```

Solution:

```
def game_helper(curr, opp, lst):
   if not lst:
      return curr == opp
   else:
      return game(opp, curr + lst[0], lst[1:]) or
         game(opp, curr + lst[-1], lst[:-1])
```

6 Scheme

1. Write a Scheme function `insert` that creates a new list that would result from inserting an item into an existing list at the given index. Assume that the given index is between 0 and the length of the original list, inclusive.

   Challenge: Write this as a tail recursive function. Assume append is tail recursive.

   ```
   (define (insert lst item index)



   )
   ```

   Solution:

   ```
   (define (insert lst item index)
     (if (= index 0)
       (cons item lst)
       (cons (car lst)
             (insert (cdr lst)
                     item
                     (- index 1))))
   )
   ```

```
; Tail recursive
(define (insert-tail lst item index)








)
```

Solution:

```
; Tail recursive
(define (insert-tail lst item index)
  (define (helper lst index so-far)
    (if (or (null? lst) (= index 0))
      (append so-far (cons item lst))
      (helper (cdr lst) (- index 1)
              (append so-far (list (car lst))))
    )
  )
  (helper lst index nil)
)
```

7 Iterators, Generators, and Streams

1. Implement all_ways_gen, which takes in a list `lst` and integer `n` and returns a generator which yields all possible ways to add together non-consecutive elements of lst to sum up to n. You can assume all elements of lst are positive.

```
def all_ways_gen(lst, n):
    """
    >>> g = all_ways_gen([1, 6, 4, 7, 2, 3], 7)
    >>> sorted(g)
    [[1, 4, 2], [4, 3], [7]]
    >>> g2 = all_ways_gen([1], 2)
    >>> list(g2)
    []
    """
    if ________________________:
        ___________________________
    elif ______________________:
        return
    else:
        first_el = lst[0]
        _______________________________________
        for s in _____________________________:
            yield __________________________
```

Solution:

```
def all_ways_gen(lst, n):
    if n == 0:
        yield []
    elif not lst:
        return
    else:
        first_el = lst[0]
        yield from all_ways_gen(lst[1:], n)
        for s in all_ways_gen(lst[2:], n - first_el):
            yield [first_el] + s
```

2. You and your CS 61A friends are cons. You cdr'd just studied for the final, but instead you scheme to drive away across a stream in a car during dead week. Of course, you would like a variety of food to eat on your road trip.

   Write an infinite stream that takes in a list of foods and loops back to the first food in the list when the list is exhausted.

   Solution:
   ```
   (define (food-stream foods)
        (cons-stream (car foods)
             (food-stream (append (cdr foods)
                  (list (car foods))))))
   ```

   We discover that some of our food is stale! Every other food that we go through is stale, so put it into a new stale food stream. Assume `is-stale` starts off as `#f`.

   Solution:
   ```
   (define (stale-stream foods is-stale)
     (cond ((null? foods) nil)
           (is-stale
             (cons-stream (car foods)
                          (stale-stream (cdr foods) #f)))
           (else (stale-stream (cdr foods) #t))))
   ```

CS61A Spring 2019: Catherine Cang and Alex Stennet, with
Albert Xu, Albert Yu, Alex Yang, Allan Yu, Amy Hung, Chris Niu, Danelle Nachum, Evan Corriere, Jack Coyle, Jade Singh, Jason Chang, Jemmy Zhou, Kevin Wang, Lucas Pan, Mary Xu, Michelle Cheung, Olivia Shiah, Patricia Ouyang, Richard Roggenkemper, Samy Cherfaoui, Shreya Sahoo, Varun Jhunjhunwalla