Review Potpourri and SQL Solutions

Computer Science Mentors

November 16, 2020 - November 19, 2020

1 SQL

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) 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

1. 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.

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

For the following two questions, you have access to two tables.

Grades, which contains three columns: `day`, **`class`**, and `score`. Each row represents the `score` you got on a midterm for some **`class`** that you took on some `day`.

Outfits, which contains two columns: `day` and `color`. Each row represents the `color` of the shirt you wore on some `day`. Assume you have a row for each possible day.

Table name: `grades`

Day	Class	Score
10/31	Music 70	88
9/20	Math 1A	72

Table name: `outfits`

Day	Color
11/5	Blue
9/13	Red
10/31	Orange

1. Instead of actually studying for your finals, you decide it would be the best use of your time to determine what your ”lucky shirt” is. Suppose you’re pretty happy with your exam scores this semester, so you define your lucky shirt as the shirt you wore to the most exams.

   Write a query that will output the color of your lucky shirt and how many times you wore it.

```
select color, count(g.day) as cnt
    from outfits as o, grades as g
    where o.day = g.day
    group by color
    order by cnt desc
    limit 1;
```

2 Scheme

1. Fill in `skip-list`, which takes in a potentially nested list `lst` and a single-argument filter function `filter-fn` that returns a boolean when called, and goes through each element in order. It returns a new list that contains all elements that return true when passed into `filter-fn`. The returned list is *not nested*.

```
;Doctests
scm> (skip-list '(1 (3)) even?)
()
scm> (skip-list '(1 (2 (3 4) 5) 6 (7) 8 9) odd?)
(1 3 5 7 9)

(define (skip-list lst filter-fn)
    (define (helper lst lst-so-far next)
        (cond
            ((null? lst)

                (if (null? ____________________)

                    ____________________________________________

                    ____________________________________________)
            )

            ((pair? ___________________________)

                (______________________________________________))
            ((filter-fn (car lst))

                __________________________________________________)
            (else

                __________________________________________________)
        )
    )

    (helper ________________________________________________)
)
```

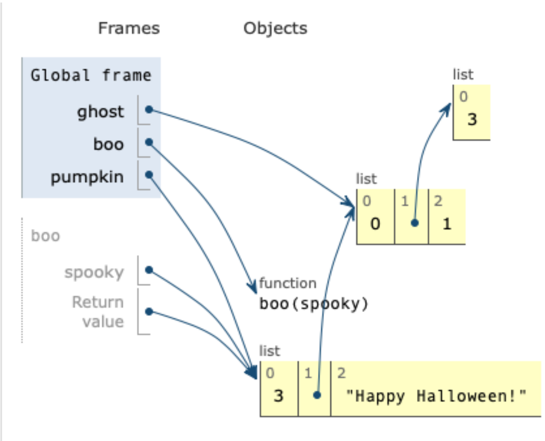
```
(define (skip-list lst filter-fn)
    (define (helper lst lst-so-far next)
```

```
            (cond
                ((null? lst) (if (null? next)
                    lst-so-far
                    (helper (car next) lst-so-far (cdr next)))
                )
                ((pair? (car lst))
                    (helper (car lst)
                        lst-so-far
                        (cons (cdr lst) next)))
                ((filter-fn (car lst))
                    (helper (cdr lst) (append lst-so-far (list (
                       car lst))) next))
                (else (helper (cdr lst) lst-so-far next))
            )
        )
        (helper lst nil nil)
    )
```

3 Environment Diagrams

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

```
ghost = [1, 0,[3], 1]
def boo(spooky):
  ghost.append(spooky.append(ghost))
  spooky = spooky[ghost[2][1][1]]
  ghost[:].extend([spooky])
  spooky = [spooky] + [ghost[spooky - 1].pop()]
  ghost.remove(ghost.remove(1))
  spooky += ["Happy Halloween!"]
  return spooky
pumpkin = boo(ghost[2])
```



PythonTutor Link

4 Recursion

1. Suppose the **position** of the rightmost digit of a number is defined as 1. The position of a digit increases from right to left.

   (a) Fill in the `positionizer` function below. It takes in a non-negative integer `n` and returns a non-negative integer. For each digit `d` of `n`, `positionizer` either changes `d` to be the remainder of `d` divided by its position, or leaves `d` the same if it is equal to its position.

```
def positionizer(n):
    """
    >>> positionizer(12)
    10
    >>> positionizer(23)
    20
    >>> positionizer(12345)
    12300
    """
    def helper(n, pos):

        if ______________________________________:

            return _________________________________

        rest = __________________________________
        if n % 10 == pos:

            return rest + _________________________
        else:

            return rest + _________________________

    return helper(______________, ______________)
```

```
def positionizer(n):
  def helper(n, pos):
    if n == 0:
      return 0
    rest = helper(n // 10, pos + 1) * 10
    if n % 10 == pos:
      return rest + n % 10
    else:
      return rest + (n % 10) % pos
  return helper(n, 1)
```

(b) Now fill in the `max_positionizer` function below. Given a list of digits, it returns the `k`-digit number with the highest value when positionized. The digits must stay in the same order. (ex: for `k = 2` and `lst = [1,2,3]`, you can't form 21). You may use `positionizer` in your solution.

Hint: Use the helper to create a list of all possible k digit numbers formed from `lst`.

```
def max_positionizer(k, lst):
    """
    >>> max_positionizer(2, [1, 2, 3]) # positionized version
                                       # of 12, 13, 23 are
                                       # 10, 10, 20
    23
    >>> max_positionizer(3, [2, 5, 3, 1])
    251
    """
    def make_nums(k,lst):

        if ________________________________:

            return ________________________________

        elif ______________________________:
            return []

        a =  [______________________________ \

                    for rest in ___________________________]

        b = ________________________________________
        return a + b

    return _______(make_nums(____, ____), ____________)
```

```
def max_positionizer(k, lst):
  def make_nums(k,lst):
    if k == 0: # Note that the check for k must come first
               # (what should be returned if k == 0 and
               # len(lst) == 0?)
      return [0]
    elif len(lst) == 0:
      return []
    a =  [lst[0] * 10**(k - 1) + rest \
            for rest in make_nums(k - 1, lst[1:])]
    b = make_nums(k, lst[1:])
    return a + b
  return max(make_nums(k, lst), key=positionizer)
```

5 OOP

1. What would Python display? The questions continue on the next page.

```
class Food:
    def __init__(self, name, spoiled = False):
        self.name = name
        self.num_days = 0
        self.spoiled = spoiled

    def can_eat(self):
        self.num_days += 1
        if self.num_days >= 3:
            self.spoiled = True
            print("Oh no! Your food is spoiled!")
        return not self.spoiled

    def mix_food(self, other_food):
        self.num_days = self.num_days + other_food.num_days
        self.name += " " + other_food.name
        self.spoiled = self.spoiled and other_food.spoiled

class Salad(Food):
    def __init__(self, ingredients):
        super().__init__("salad", False)
        self.ingredients = ingredients

    def add_ingredients(self, ingredient):
        self.ingredients.append(ingredient)
        print(ingredient.name + " has been added")

    def mix_ingredients(self):
        for ingredient in self.ingredients:
            self.mix_food(ingredient)
        print("Your salad has been mixed.")

lettuce = Food("lettuce")
tomatoes = Food("tomatoes")
chicken = Food("chicken")
ingredients = [lettuce, tomatoes]
my_salad = Salad(ingredients)
```

See visualizations for solutions: https://docs.google.com/presentation/d/1t1yE9DuT8a2ij_QszLOxzUu6-unN46PY1SA_Q48fLz4/edit?usp=sharing

```
>>> lettuce.can_eat()
```

True

```
>>> my_salad.can_eat()
```

True

```
>>> my_salad.mix_ingredients()
```

Your salad has been mixed.

```
>>> my_salad.name
```

"salad lettuce tomatoes"

6 Trees

1. The total weight of a tree is defined as the sum of the labels of all its nodes. A tree is defined to be `equally_weighted` if the total weight of each of its branches are equal. A leaf is assumed to be equally weighted.

   Complete the following functions for `equally_weighted` and `num_eq_weight`. You may use the below definition of the `total_weight` function in your answer.

```
def total_weight(t):
    """
    Return the total weight of a tree, i.e. the sum of all its
       labels.
    >>>total_weight(Tree(1, [Tree(2), Tree(3,[Tree(4)])]))
    10
    """
    weight =  t.label + sum([total_weight(branch) for branch in
       t.branches])
    return weight
```

(a)

```
def equally_weighted(t):
    """
    Return whether a tree is equally weighted.
    >>>equally_weighted(Tree(1))
    True
    >>>equally_weighted(Tree(1,[Tree(2), Tree(1, [Tree(1)])])
       )
    True
    >>>equally_weighted(Tree(0, [Tree(3), Tree(2, [Tree(3)])
       ]))
    False
    """
    ______________________ = [________________________]

    for _______________ in ____________________________:

        if ____________________________:

            return ______________________________________

    return __________________________________________
```

```
def equally_weighted(t):
  all_weights = [total_weight(b) for b in t.branches]
  for weight in all_weights[1:]:
    if weight != all_weights[0]:
      return False
  return True
```

(b) **Note:** You are allowed to use `equally_weighted` in this part.

```
def num_eq_weight(t):
    """
    Return the number of equally weighted subtrees of t. Note
        that t is considered a subtree of itself.
    >>> num_eq_weight(Tree(1, [Tree(4), Tree(3,[Tree(1)])]))
    4
    >>> num_eq_weight(Tree(1, [Tree(9),
                                Tree(1, [Tree(4),
                                         Tree(3,[Tree(1)])])])
                                            )
    6
    >>> num_eq_weight(Tree(1, [Tree(8, [Tree(1)]),
                                Tree(1, [Tree(4),
                                         Tree(3,[Tree(1)])])])
                                            )
    7
    """
    val = ______________________________________________

    if ________________________________________:

        return ____________________________________________
    else:
        return val
```

```
def num_eq_weight(t):
    val = sum([num_eq_weight(b) for b in t.branches])
    if equally_weighted(t):
        return 1 + val
    else:
        return val
```

7 Generators

1. Define a **non-decreasing path** as a path from the root where each node's label is greater than or equal to the previous node along the path. A **subpath** is a path between nodes X and Y, where Y must be a descendent of X (ex: Y is a branch of a branch of X).

   (a) Write a generator function `root_to_leaf` that takes in a tree `t` and yields all non-decreasing paths from the root to a leaf node, in any order. Assume that `t` has at least one node.

```
def root_to_leaf(t):
    """
    >>> t1 = Tree(3, [Tree(5), Tree(4)])
    >>> list(root_to_leaf(t1))
    [[3, 5], [3, 4]]
    >>> t2 = Tree(5, [Tree(2, [Tree(7), Tree(8)]), Tree(5,
       [Tree(6)])])
    [[5, 5, 6]]
    """

    if ______________________:

        ____________________________

    for ____________________________:

        if _____________________________:

            for _____________________________:

                __________________________________
```

```
def root_to_leaf(t):
    if t.is_leaf():
        yield [t.label]
    for b in t.branches:
        if t.label <= b.label:
            for path in root_to_leaf(b):
                yield [t.label] + path
```

The easiest way to approach this is to notice the two blocks of code that are provided: first an if statement, probably referring to a base case, and a for loop, which will probably be the recursive case. From the doctests, we can see that giving the function a tree that just has one node, or in other words `is_leaf()`, returns a list containing just that node.

In our recursive case we want to do two things. First, we want to check if the next branch value really is non-decreasing. Then, if it is, we want to append the result of calling `root_to_leaf` on the branch to the value of our current tree to create a complete path. So we recurse through each of the branches in `t` (**`for`** `b` **`in`** `t.branches`), then check if it is nondecreasing (`t.label <= b.label`), then yield our tree's label appended to the recursive call (the last two lines).

(b) Write a generator function `subpaths` that takes in a tree `t` and yields all non-decreasing subpaths that end with a leaf node, in any order. You may use the `root_to_leaf` function above, and assume again that `t` has at least one node.

```
def subpaths(t):

    yield from ________________________

    for b in t.branches:

        ______________________________
```

```
def subpaths(t):
    yield from root_to_leaf(t)
    for b in t.branches:
        yield from subpaths(b)
```

We can split this problem into two steps – yielding all subpaths for the current tree that we have, then yielding all subpaths for all other trees within this tree. It is important to realize that each node in the tree is merely a subtree of the original tree to solve this problem.

To yield all non-decreasing subpaths for our current tree (that is all non-decreasing subpaths that start at our current node and end at the leaf nodes), we can just yield from our previous function, `root_to_leaf`, called on that node. For the rest of the subpaths, we want to recursively call `subpaths` on all our child nodes. This will give us all paths that end on the leaf nodes (because `root_to_leaf` ends on the leaf nodes) that start from any child on this tree. It is important to realize that the base case in this situation is implicit. If a leaf node is passed in and reaches the for loop, the for loop finds no items in `t.branches`, and will just terminate without calling the clause inside.