CS 61A Summer 2022

More SQL, Final Review

Discussion 12: August 4, 2022 Solutions

SQL

> You can refer back to Discussion 11 for an overview of SQL, including joins and aggregation.

(Adapted from Fall 2019) The scoring table has three columns, a player column of strings, a points column of integers, and a quarter column of integers. The players table has two columns, a name column of strings and a team column of strings. Complete the SQL statements below so that they would compute the correct result even if the rows in these tables were different than those shown.

Important: You may write anything in the blanks including keywords such as WHERE or ORDER BY. Use the following tables for the questions below:

```
CREATE TABLE scoring AS
    SELECT "Donald Stewart" AS player, 7 AS points, 1 AS quarter
    UNION
    SELECT "Christopher Brown Jr.", 7, 1 UNION
    SELECT "Ryan Sanborn", 3, 2 UNION
    SELECT "Greg Thomas", 3, 2 UNION
    SELECT "Cameron Scarlett", 7, 3 UNION
    SELECT "Nikko Remigio", 7, 4 UNION
    SELECT "Ryan Sanborn", 3, 4 UNION
    SELECT "Chase Garbers", 7, 4;

CREATE TABLE players AS
    SELECT "Ryan Sanborn" AS name, "Stanford" AS team UNION
    SELECT "Donald Stewart", "Stanford" UNION
    SELECT "Cameron Scarlett", "Stanford" UNION
    SELECT "Christopher Brown Jr.", "Cal" UNION
    SELECT "Greg Thomas", "Cal" UNION
    SELECT "Nikko Remigio", "Cal" UNION
    SELECT "Chase Garbers", "Cal";
```

Q1: Big Quarters

Write a SQL statement to select a one-column table of quarters in which more than 10 total points were scored.

Solution: SELECT quarter FROM scoring GROUP BY quarter HAVING SUM(points) > 10;

Q2: Score

Write a SQL statement to select a two-column table where the first column is the team name and the second column is the total points scored by that team. Assume that no two players have the same name.

Solution: SELECT team, SUM(points) FROM scoring, players WHERE player=name GROUP BY team;

Final Review

The following worksheet is final review! It covers various topics that have been seen throughout the semester.

Your TA will not be able to get to all of the problems on this worksheet so feel free to work through the remaining problems on your own. Bring any questions you have to office hours or post them on piazza.

Good luck on the final and congratulations on making it to the last discussion of CS61A!

Recursion

Q3: Maximum Subsequence

A subsequence of a number is a series of (not necessarily contiguous) digits of the number. For example, 12345 has subsequences that include 123, 234, 124, 245, etc. Your task is to get the maximum subsequence below a certain length.

```python
def max_subseq(n, t):
    """
    Return the maximum subsequence of length at most t that can be
    found in the given number n.
    For example, for n = 2012 and t = 2, we have that the
    subsequences are
        2
        0
        1
        2
        20
        21
        22
        01
        02
        12
    and of these, the maxumum number is 22, so our answer is 22.

    >>> max_subseq(2012, 2)
    22
    >>> max_subseq(20125, 3)
    225
    >>> max_subseq(20125, 5)
    20125
    >>> max_subseq(20125, 6) # note that 20125 == 020125
    20125
    >>> max_subseq(12345, 3)
    345
    >>> max_subseq(12345, 0) # 0 is of length 0
    0
    >>> max_subseq(12345, 1)
    5
    >>> from construct_check import check
    >>> # ban usage of str
    >>> check(LAB_SOURCE_FILE, 'max_subseq', ['Str', 'Slice'])
    True
    """
    if n == 0 or t == 0:
        return 0
    with_last = max_subseq(n // 10, t - 1) * 10 + n % 10
    without_last = max_subseq(n // 10, t)
    return max(with_last, without_last)
```

Mutation

Q4: Reverse

Write a function that reverses the given list. Be sure to mutate the original list. This is practice, so don't use the built-in `reverse` function!

```python
def reverse(lst):
    """Reverses lst using mutation.

    >>> original_list = [5, -1, 29, 0]
    >>> reverse(original_list)
    >>> original_list
    [0, 29, -1, 5]
    >>> odd_list = [42, 72, -8]
    >>> reverse(odd_list)
    >>> odd_list
    [-8, 72, 42]
    """
    # iterative solution
    midpoint = len(lst) // 2
    last = len(lst) - 1
    for i in range(midpoint):
        lst[i], lst[last - i] = lst[last - i], lst[i]
```

Efficiency

Q5: The First Order...of Growth

What is the efficiency of `rey`?

```
def rey(finn):
    poe = 0
    while finn >= 2:
        poe += finn
        finn = finn / 2
    return
```

Logarithmic, because our while loop iterates at most log(`finn`) times, due to `finn` being halved in every iteration. This is commonly known as $\Theta(\log(\texttt{finn}))$ runtime. Another way of looking at this if you duplicate the input, we only add a single iteration to the time, which also indicates logarithmic.

What is the efficiency of `mod_7`?

```
def mod_7(n):
    if n % 7 == 0:
        return 0
    else:
        return 1 + mod_7(n - 1)
```

Constant, since at worst it will require 6 recursive calls to reach the base case. This is commonly known as a $\Theta(1)$ runtime.

Trees

Q6: Level Mutation

As a reminder, the depth of a node is how far away the node is from the root. We define this as the number of edges between the root to the node. As there are no edges between the root and itself, the root has depth 0.

Given a tree `t` and a list of one-argument functions `funcs`, write a function that will mutate the labels of `t` using the function from `funcs` at the corresponding depth. For example, the label at the root node (with a depth of 0) will be mutated using the function at index 0, or `funcs[0]`. Assume all of the functions in `funcs` will be able to take in a label value and return a valid label value.

If `t` is a leaf and there are more than 1 functions in `funcs`, all of the remaining functions should be applied in order to the label of `t`. (See the doctests for an example.) If `funcs` is empty, the tree should remain unmodified.

```
def level_mutation(t, funcs):
    """Mutates t using the functions in the list funcs.

    >>> t = Tree(1, [Tree(2, [Tree(3)])])
    >>> funcs = [lambda x: x + 1, lambda y: y * 5, lambda z: z ** 2]
    >>> level_mutation(t, funcs)
    >>> t                              # funcs[0] was applied to
    the label 1, funcs[1] to the label 2, etc.
    Tree(2, [Tree(10, [Tree(9)])])
    >>> t2 = Tree(1, [Tree(2), Tree(3, [Tree(4)])])
    >>> level_mutation(t2, funcs)
    >>> t2                             # (2 * 5) ** 2 = 100
    Tree(2, [Tree(100), Tree(15, [Tree(16)])])
    >>> t3 = Tree(1, [Tree(2)])
    >>> level_mutation(t3, funcs)
    >>> t3
    Tree(2, [Tree(100)])
    """
    if not funcs:
        return
    t.label = funcs[0](t.label)
    remaining = funcs[1:]
    if t.is_leaf() and remaining:
        for f in remaining:
            t.label = f(t.label)
    for b in t.branches:
        level_mutation(b, remaining)
```

As an extra challenge, try implementing `level_mutation_link`, which has the same behavior as `level_mutation` except it will take in a Linked List of functions `funcs` (rather than a Python list of functions).

```
def level_mutation_link(t, funcs):
    if funcs is Link.empty:
        return
    t.label = funcs.first(t.label)
    remaining = funcs.rest
    if t.is_leaf() and remaining is not Link.empty:
        while remaining is not Link.empty:
            t.label = remaining.first(t.label)
            remaining = remaining.rest
    for b in t.branches:
        level_mutation(b, remaining)
```

Linked Lists

Q7: Multiply Links

Write a function that takes in a Python list of linked lists and multiplies them element-wise. It should return a new linked list.

If not all of the `Link` objects are of equal length, return a linked list whose length is that of the shortest linked list given. You may assume the `Link` objects are shallow linked lists, and that `lst_of_lnks` contains at least one linked list.

```
def multiply_lnks(lst_of_lnks):
    """
    >>> a = Link(2, Link(3, Link(5)))
    >>> b = Link(6, Link(4, Link(2)))
    >>> c = Link(4, Link(1, Link(0, Link(2))))
    >>> p = multiply_lnks([a, b, c])
    >>> p.first
    48
    >>> p.rest.first
    12
    >>> p.rest.rest.rest is Link.empty
    True
    """
    # Implementation Note: you might not need all lines in this
    skeleton code
    product = 1
    for lnk in lst_of_lnks:
        if lnk is Link.empty:
            return Link.empty
        product *= lnk.first
    lst_of_lnks_rests = [lnk.rest for lnk in lst_of_lnks]
    return Link(product, multiply_lnks(lst_of_lnks_rests))
```

For our base case, if we detect that any of the lists in the list of `Links` is empty, we can return the empty linked list as we're not going to multiply anything.

Otherwise, we compute the product of all the firsts in our list of `Links`. Then, the subproblem we use here is the rest of all the linked lists in our list of Links. Remember that the result of calling `multiply_lnks` will be a linked list! We'll use the product we've built so far as the first item in the returned `Link`, and then the result of the recursive call as the rest of that `Link`.

Next, we have the iterative solution:

```
def multiply_lnks(lst_of_lnks):
    """
    >>> a = Link(2, Link(3, Link(5)))
    >>> b = Link(6, Link(4, Link(2)))
    >>> c = Link(4, Link(1, Link(0, Link(2))))
    >>> p = multiply_lnks([a, b, c])
    >>> p.first
    48
    >>> p.rest.first
    12
    >>> p.rest.rest.rest is Link.empty
    True
    """
    # Alternate iterative approach
    import operator
    from functools import reduce
    def prod(factors):
         return reduce(operator.mul, factors, 1)

    head = Link.empty
    tail = head
    while Link.empty not in lst_of_lnks:
        all_prod = prod([l.first for l in lst_of_lnks])
        if head is Link.empty:
            head = Link(all_prod)
            tail = head
        else:
            tail.rest = Link(all_prod)
            tail = tail.rest
        lst_of_lnks = [l.rest for l in lst_of_lnks]
    return head
```

The iterative solution is a bit more involved than the recursive solution. Instead of building the list **backwards** as in the recursive solution (because of the order that the recursive calls result in, the last item in our list will be finished first), we'll build the resulting linked list as we go along.

We use`head` and `tail` to track the front and end of the new linked list we're creating. Our stopping condition for the loop is if any of the `Link`s in our list of `Link`s runs out of items.

Finally, there's some special handling for the first item. We need to update both head and tail in that case. Otherwise, we just append to the end of our list using tail, and update tail.

Scheme

Q8: List Insert

Write a Scheme function that, when given an element, a list, and an index, inserts the element into the list at that index. You can assume that the index is in bounds for the list.

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

(expect (insert 2 '(1 7 9) 2) (1 7 2 9))
(expect (insert 'a '(b c) 0) (a b c))
```

Q9: Group by Non-Decreasing

Define a function `nondecreaselist`, which takes in a scheme list of numbers and outputs a list of lists, which overall has the same numbers in the same order, but grouped into lists that are non-decreasing.

For example, if the input is a stream containing elements

```
(1 2 3 4 1 2 3 4 1 1 1 2 1 1 0 4 3 2 1)
```

the output should contain elements

```
((1 2 3 4) (1 2 3 4) (1 1 1 2) (1 1) (0 4) (3) (2) (1))
```

Note: The skeleton code is just a suggestion; feel free to use your own structure if you prefer.

```
(define (nondecreaselist s)

    (if (null? s)
        nil
        (let ((rest (nondecreaselist (cdr s)) ))
            (if (or (null? (cdr s)) (> (car s) (car (cdr s))))
                (cons (list (car s)) rest)
                (cons (cons (car s) (car rest)) (cdr rest))
            )
        )
    )
)

(expect (nondecreaselist '(1 2 3 1 2 3)) ((1 2 3) (1 2 3)))

(expect (nondecreaselist '(1 2 3 4 1 2 3 4 1 1 1 2 1 1 0 4 3 2 1))
        ((1 2 3 4) (1 2 3 4) (1 1 1 2) (1 1) (0 4) (3) (2) (1)))
```

Regex

Q10: Greetings

Let's say hello to our fellow bears! We've received messages from our new friends at Berkeley, and we want to determine whether or not these messages are *greetings*. In this problem, there are two types of greetings - salutations and valedictions. The first are messages that start with "hi", "hello", or "hey", where the first letter of these words can be either capitalized or lowercase. The second are messages that end with the word "bye" (capitalized or lowercase), followed by either an exclamation point, a period, or no punctuation. Write a regular expression that determines whether a given message is a greeting.

```
import re

def greetings(message):
    """
    Returns whether a string is a greeting. Greetings begin with
    either Hi, Hello, or
    Hey (first letter either capitalized or lowercase), and/or end
    with Bye (first letter
    either capitalized or lowercase) optionally followed by an
    exclamation point or period.

    >>> greetings("Hi! Let's talk about our favorite submissions to
    the Scheme Art Contest")
    True
    >>> greetings("Hey I love Taco Bell")
    True
    >>> greetings("I'm going to watch the sun set from the top of
    the Campanile! Bye!")
    True
    >>> greetings("Bye Bye Birdie is one of my favorite musicals.")
    False
    >>> greetings("High in the hills of Berkeley lived a legendary
    creature. His name was Oski")
    False
    >>> greetings('Hi!')
    True
    >>> greetings("bye")
    True
    """
    return bool(re.search(r"(^([Hh](ey|i|ello)\b))|(\b[bB]ye[!\.]?$)
    ", message))
```