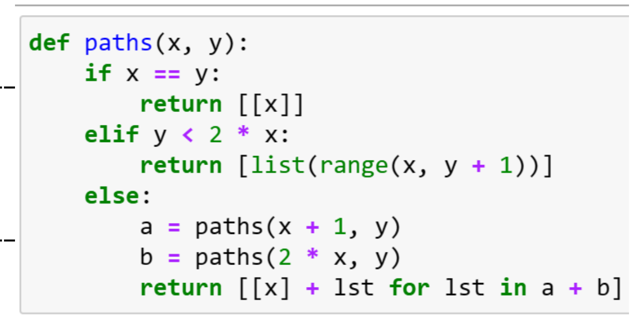
CS 61A Summer 2020 | Final Review
Discussion 14: August 11, 2020

1 Recursion

1.1 (Adapted from Fall 2013) Fill in the blanks in the implementation of `paths`, which takes as input two positive integers `x` and `y`. It returns a list of paths, where each path is a list containing steps to reach `y` from `x` by repeated incrementing or doubling. For instance, we can reach 9 from 3 by incrementing to 4, doubling to 8, then incrementing again to 9, so one path is `[3, 4, 8, 9]`

```
def paths(x, y):
    """Return a list of ways to reach y from x by repeated
    incrementing or doubling.
    >>> paths(3, 5)
    [[3, 4, 5]]
    >>> sorted(paths(3, 6))
    [[3, 4, 5, 6], [3, 6]]
    >>> sorted(paths(3, 9))
    [[3, 4, 5, 6, 7, 8, 9], [3, 4, 8, 9], [3, 6, 7, 8, 9]]
    >>> paths(3, 3) # No calls is a valid path
    [[3]]
    """
    if ___________________________:

        return ____________________

    elif _________________________:

        return ____________________

    else:

        a = __________________________________________________

        b = __________________________________________________

        return _______________________________________________
```



```
def paths(x, y):
    if x == y:
        return [[x]]
    elif y < 2 * x:
        return [list(range(x, y + 1))]
    else:
        a = paths(x + 1, y)
        b = paths(2 * x, y)
        return [[x] + lst for lst in a + b]
```

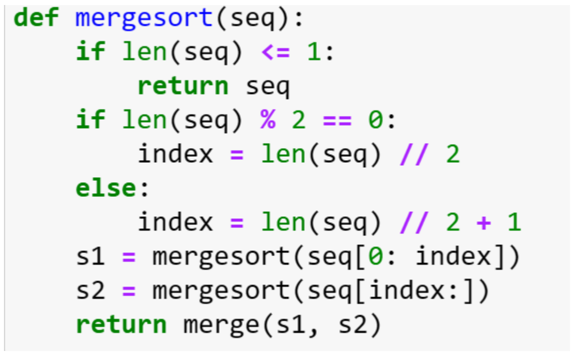
1.2 We will now write one of the faster sorting algorithms commonly used, known as *merge sort*. Merge sort works like this:

1. If there is only one (or zero) item(s) in the sequence, it is already sorted!
2. If there are more than one item, then we can split the sequence in half, sort each half recursively, then merge the results, using the merge procedure described below. The result will be a sorted sequence.

Using the algorithm described, write a function mergesort(seq) that takes an unsorted sequence and sorts it.

Recall the merge procedure is as follows:

```
def merge(s1, s2):
    """ Merges two sorted lists """
    if len(s1) == 0:
        return s2
    elif len(s2) == 0:
        return s1
    elif s1[0] < s2[0]:
        return [s1[0]] + merge(s1[1:], s2)
    else:
        return [s2[0]] + merge(s1, s2[1:])
```



```
def mergesort(seq):
    if len(seq) <= 1:
        return seq
    if len(seq) % 2 == 0:
        index = len(seq) // 2
    else:
        index = len(seq) // 2 + 1
    s1 = mergesort(seq[0: index])
    s2 = mergesort(seq[index:])
    return merge(s1, s2)
```

```
def mergesort(seq):
```

2 Trees

2.1 Implement long_paths, which returns a list of all *paths* in a tree with length at least n. A path in a tree is a linked list of node values that starts with the root and ends at a leaf. Each subsequent element must be from a child of the previous value's node. The *length* of a path is the number of edges in the path (i.e. one less than the number of nodes in the path). Paths are listed in order from left to right. See the doctests for some examples.

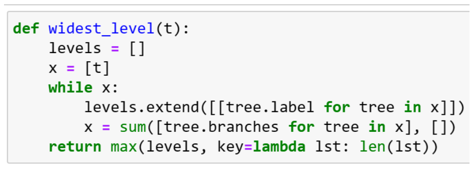
```
def long_paths(tree, n):
    """Return a list of all paths in tree with length at least n.

    >>> t = Tree(3, [Tree(4), Tree(4), Tree(5)])
    >>> left = Tree(1, [Tree(2), t])
    >>> mid = Tree(6, [Tree(7, [Tree(8)]), Tree(9)])
    >>> right = Tree(11, [Tree(12, [Tree(13, [Tree(14)])])])
    >>> whole = Tree(0, [left, Tree(13), mid, right])
    >>> for path in long_paths(whole, 2):
    ...     print(path)
    ...
    <0 1 2>
    <0 1 3 4>
    <0 1 3 4>
    <0 1 3 5>
    <0 6 7 8>
    <0 6 9>
    <0 11 12 13 14>
    >>> for path in long_paths(whole, 3):
    ...     print(path)
    ...
    <0 1 3 4>
    <0 1 3 4>
    <0 1 3 5>
    <0 6 7 8>
    <0 11 12 13 14>
    >>> long_paths(whole, 4)
    [Link(0, Link(11, Link(12, Link(13, Link(14)))))]
    """
```

```
def long_paths(tree, n):
    if tree.is_leaf() and n <= 0:
        return [Link(tree.label)]
    elif tree.is_leaf() and n > 0:
        return []
    ans = []
    for b in tree.branches:
        ans = ans + long_paths(b, n - 1)
    return [Link(tree.label, rest) for rest in ans]
```

2.2 Write a function that takes a `Tree` object and returns the elements at the depth with the most elements.

In this problem, you may find it helpful to use the second optional argument to `sum`, which provides a starting value. All items in the sequence to be summed will be concatenated to the starting value. By default, start will default to 0, which allows you to sum a sequence of numbers. We provide an example of sum starting with a list, which allows you to concatenate items in a list.

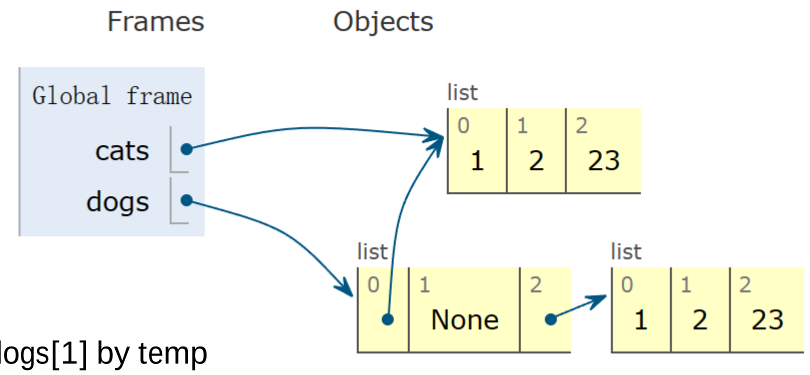
```
def widest_level(t):
    """
    >>> sum([[1], [2]], [])
    [1, 2]
    >>> t = Tree(3, [Tree(1, [Tree(1), Tree(5)]),
    ...              Tree(4, [Tree(9, [Tree(2)])])])
    >>> widest_level(t)
    [1, 5, 9]
    """
    levels = []
    x = [t]

    while __________________________________________________:

        _____________________________________________________

        __________ = sum(_______________________________, [])

    return max(levels, key=_________________________________)
```



```
def widest_level(t):
    levels = []
    x = [t]
    while x:
        levels.extend([[tree.label for tree in x]])
        x = sum([tree.branches for tree in x], [])
    return max(levels, key=lambda lst: len(lst))
```

3 Mutation

3.1 For each row below, fill in the blanks in the output displayed by the interactive Python interpreter when the expression is evaluated. Expressions are evaluated in order, and expressions may affect later expressions.

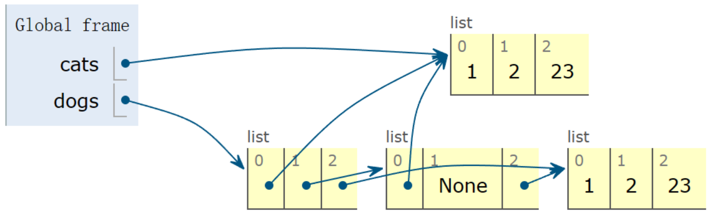


```
>>> cats = [1, 2]
>>> dogs = [cats, cats.append(23), list(cats)]
>>> cats
```

[1, 2, 23]

```
>>> dogs[1] = list(dogs)
>>> dogs[1]
```

firstly, copy dogs as temp; secondly, replace dogs[1] by temp
[[1, 2, 23], None, [1, 2, 23]]



```
>>> dogs[0].append(2)
>>> cats
```

[1, 2, 23, 2]

```
>>> cats[1::2]
```

[2, 2]

```
>>> cats[:3]
```

[1, 2, 23]

```
>>> dogs[2].extend([list(cats).pop(0), 3])
>>> dogs[3]
```

[[2, 23, 2], 3] Error
notice it's "dogs[2].extend", not "dogs.extend"!

```
>>> dogs
```

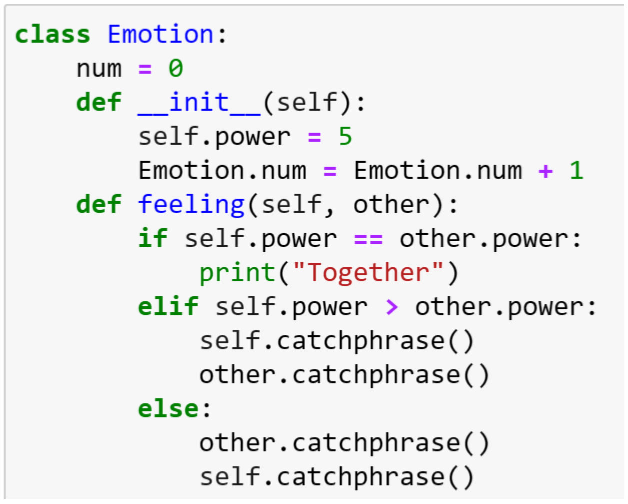
[[1, 2, 23, 2], [[1, 2, 23, 2], None, [1, 2, 23, [2, 3, 23, 2], 3]], [1, 2, 23, [2, 3, 23, 2], 3]]
[2, 23, 2] -> 1 (pop changes the original list and returns the poped value)

4 OOP

4.1 Fill in the classes `Emotion`, `Joy`, and `Sadness` below so that you get the following output from the Python interpreter.

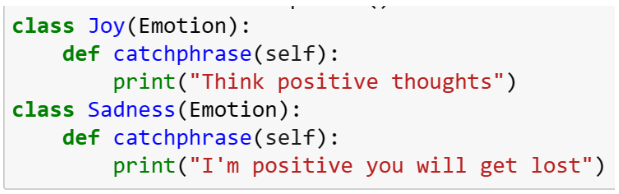
```
>>> Emotion.num
0
>>> joy = Joy()
>>> sadness = Sadness()
>>> Emotion.num # number of Emotion instances created
2
>>> joy.power
5
>>> joy.catchphrase() # Print Joy's catchphrase
Think positive thoughts
>>> sadness.catchphrase() #Print Sad's catchphrase
I'm positive you will get lost
>>> sadness.power
5
>>> joy.feeling(sadness) # When both Emotions have same power value, print "Together"
Together
>>> sadness.feeling(joy)
Together
>>> joy.power = 7
>>> joy.feeling(sadness) # Print the catchphrase of the more powerful feeling before the less
    powerful feeling
Think positive thoughts
I'm positive you will get lost
>>> sadness.feeling(joy)
Think positive thoughts
I'm positive you will get lost
```

```
class Emotion(_______):



    def __init__(self):





    def feeling(self, other):
```



```
class Emotion:
    num = 0
    def __init__(self):
        self.power = 5
        Emotion.num = Emotion.num + 1
    def feeling(self, other):
        if self.power == other.power:
            print("Together")
        elif self.power > other.power:
            self.catchphrase()
            other.catchphrase()
        else:
            other.catchphrase()
            self.catchphrase()
```

```
class Joy(_______):

    def catchphrase(self):
```



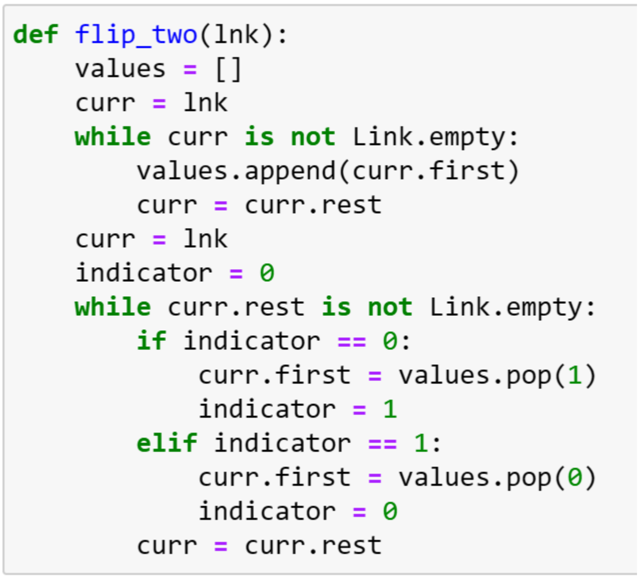
```
class Joy(Emotion):
    def catchphrase(self):
        print("Think positive thoughts")
class Sadness(Emotion):
    def catchphrase(self):
        print("I'm positive you will get lost")
```

```
class Sadness(_______):

    def catchphrase(self):
```

5 Mutable Linked Lists and Trees

5.1 Write a recursive function `flip_two` that takes as input a linked list `lnk` and mutates `lnk` so that every pair is flipped.

```
def flip_two(lnk):
    """
    >>> one_lnk = Link(1)
    >>> flip_two(one_lnk)
    >>> one_lnk
    Link(1)
    >>> lnk = Link(1, Link(2, Link(3, Link(4, Link(5)))))
    >>> flip_two(lnk)
    >>> lnk
    Link(2, Link(1, Link(4, Link(3, Link(5)))))
    """
```

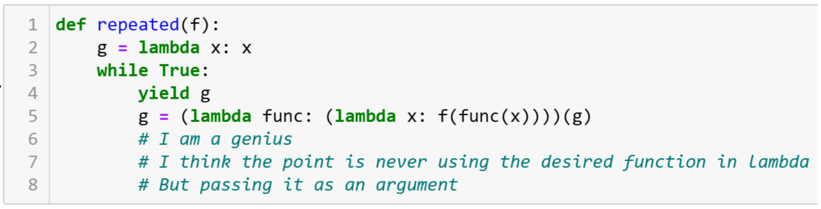


```
def flip_two(lnk):
    values = []
    curr = lnk
    while curr is not Link.empty:
        values.append(curr.first)
        curr = curr.rest
    curr = lnk
    indicator = 0
    while curr.rest is not Link.empty:
        if indicator == 0:
            curr.first = values.pop(1)
            indicator = 1
        elif indicator == 1:
            curr.first = values.pop(0)
            indicator = 0
        curr = curr.rest
```

6 Generators

6.1 Write a generator function that yields functions that are repeated applications of a one-argument function `f`. The first function yielded should apply `f` 0 times (the identity function), the second function yielded should apply `f` once, etc.

```
def repeated(f):
    """
    >>> double = lambda x: 2 * x
    >>> funcs = repeated(double)
    >>> identity = next(funcs)
    >>> double = next(funcs)
    >>> quad = next(funcs)
    >>> oct = next(funcs)
    >>> quad(1)
    4
    >>> oct(1)
    8
    >>> [g(1) for _, g in
    ...  zip(range(5), repeated(lambda x: 2 * x))]
    [1, 2, 4, 8, 16]
    """

    g = ________________

    while True:

        ________________________________________________________________________

        ________________________________________________________________________
```



```
def repeated(f):
    g = lambda x: x
    while True:
        yield g
        g = (lambda func: (lambda x: f(func(x))))(g)
        # I am a genius
        # I think the point is never using the desired function in lambda
        # But passing it as an argument
```

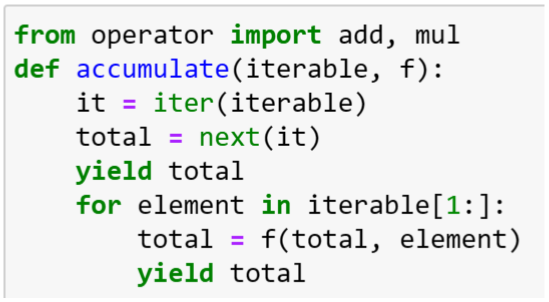
6.2 Ben Bitdiddle proposes the following alternate solution. Does it work?

```
def ben_repeated(f):
    g = lambda x: x
    while True:
        yield g
        g = lambda x: f(g(x))
```

it cannot work as expected

6.3 Implement `accumulate`, which takes in an `iterable` and a function `f` and yields each accumulated value from applying `f` to the running total and the next element.

```
from operator import add, mul

def accumulate(iterable, f):
    """
    >>> list(accumulate([1, 2, 3, 4, 5], add))
    [1, 3, 6, 10, 15]
    >>> list(accumulate([1, 2, 3, 4, 5], mul))
    [1, 2, 6, 24, 120]
    """
    it = iter(iterable)

    ____________________________________________________________________________

    ____________________________________________________________________________

    for ________________________________________________________________________:

        ________________________________________________________________________

        ________________________________________________________________________
```



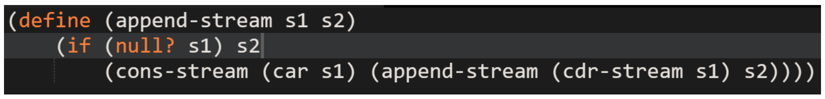
```
from operator import add, mul
def accumulate(iterable, f):
    it = iter(iterable)
    total = next(it)
    yield total
    for element in iterable[1:]:
        total = f(total, element)
        yield total
```

7 Scheme

7.1 Implement the `append-stream` procedure, which takes in two streams and returns a stream with the two streams concatenated. (Note that if the first stream is infinite, the result will not contain any elements from the second stream.)

```
scm> (define s (cons-stream 1 (cons-stream 2 nil)))
s
scm> (define a (append-stream s s))
a
scm> (car a)
1
scm> (car (cdr-stream a))
2
scm> (car (cdr-stream (cdr-stream a)))
1
scm> (car (cdr-stream (cdr-stream (cdr-stream a))))
2
```

```
(define (append-stream s1 s2)
```

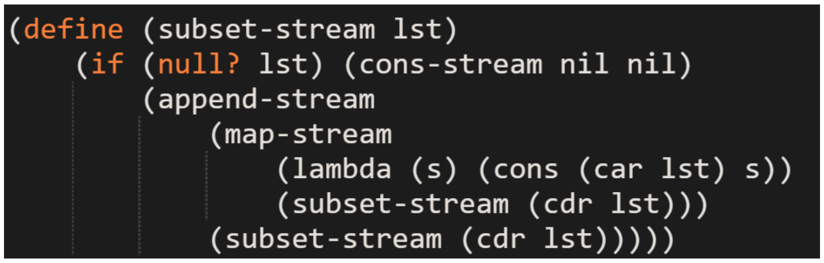


```
(define (append-stream s1 s2)
    (if (null? s1) s2
        (cons-stream (car s1) (append-stream (cdr-stream s1) s2))))
```

7.2 Now implement `subset-stream`, which takes in a normal Scheme list and returns a stream with every possible subset of that Scheme list.

It might help to use `map-stream`, which we've defined for you:

```
(define (map-stream f s)
  (if (null? s)
    nil
    (cons-stream (f (car s)) (map-stream f (cdr-stream s)))))

scm> (define a (subset-stream '(1 2 3 4 5)))
a
scm> (car a)
(1 2 3 4 5)
scm> (car (cdr-stream a))
(1 2 3 4)
scm> (car (cdr-stream (cdr-stream a)))
(1 2 3 5)
scm> (car (cdr-stream (cdr-stream (cdr-stream a))))
(1 2 3)

(define (subset-stream lst)
```



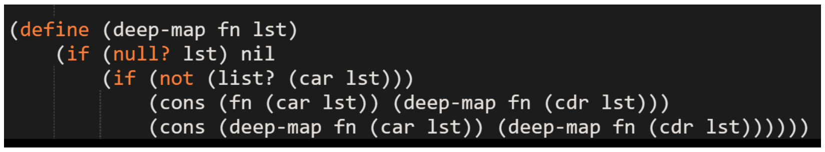
```
(define (subset-stream lst)
    (if (null? lst) (cons-stream nil nil)
        (append-stream
            (map-stream
                (lambda (s) (cons (car lst) s))
                (subset-stream (cdr lst)))
            (subset-stream (cdr lst)))))
```

7.3 Write a function that takes a procedure and applies to every element in a given nested list.

The result should be a nested list with the same structure as the input list, but with each element replaced by the result of applying the procedure to that element.

Use the built-in `list?` procedure to detect whether a value is a list.

```
(define (deep-map fn lst)
```

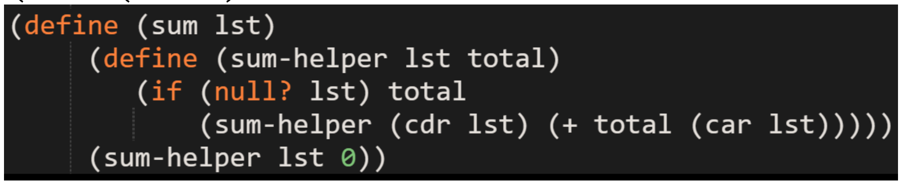


```
(define (deep-map fn lst)
    (if (null? lst) nil
        (if (not (list? (car lst)))
            (cons (fn (car lst)) (deep-map fn (cdr lst)))
            (cons (deep-map fn (car lst)) (deep-map fn (cdr lst))))))
```

```
scm> (deep-map (lambda (x) (* x x)) '(1 2 3))
(1 4 9)
scm> (deep-map (lambda (x) (* x x)) '(1 ((4) 5) 9))
(1 ((16) 25) 81)
```

7.4 Write a tail recursive function that takes in a Scheme list and returns the numerical sum of all values in the list. You can assume that the list contains only numbers (no nested lists).

```
(define (sum lst)
```



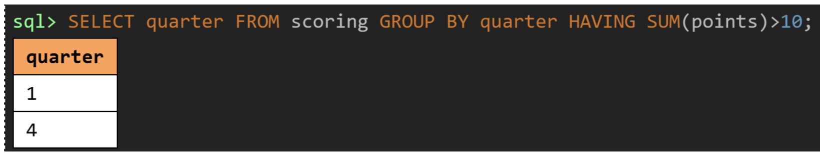
```
(define (sum lst)
     (define (sum-helper lst total)
        (if (null? lst) total
            (sum-helper (cdr lst) (+ total (car lst)))))
     (sum-helper lst 0))
```

8 SQL

(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 table called `courses` 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";
```

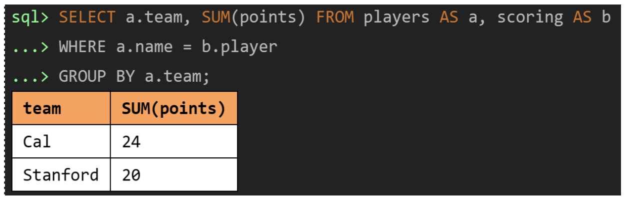
8.1 Write a SQL statement to select a one-column table of quarters in which more than 10 total points were scored.



```
sql> SELECT quarter FROM scoring GROUP BY quarter HAVING SUM(points)>10;
```

quarter
1
4

8.2 Write a SQL statement to select a two-column table of the points scored by each team. Assume that no two players have the same name.



```
sql> SELECT a.team, SUM(points) FROM players AS a, scoring AS b
...> WHERE a.name = b.player
...> GROUP BY a.team;
```

team	SUM(points)
Cal	24
Stanford	20

Note: This worksheet is a problem bank—most TAs will not cover all the problems in discussion section.