CS 370 Introduction to Teaching Computer Science
Fall 2018 Christopher Hunn, Sequoia Eyzaguirre, & Bryan Tong

Station 1

Name:

Section:

At this station, you will design questions that target the higher levels of Bloom's taxonomy while relying on the lower levels for support. On the following pages, you'll find 4 challenging problems from past CS 61A midterms and final exams. For each problem, complete the two tasks below. Don't feel compelled to go in any particular order, and please complete *both* tasks for each question before moving on.

1. **Task 1**
   Identify where the problem falls on Bloom's pyramid. Remember, the categories are *Remember*, *Understand*, *Apply*, *Analyze*, and *Create*. Discuss this in depth with your teammates! Multiple answers may be correct.

2. **Task 2**
   Come up with a set of questions that you could use to lead a student through the problem, using the Socratic method. Assume no skeleton code is provided. Remember to go high-level first, and then expand on the details. Also think about how you or your team would approach solving the problem.

1. MULTIADDER

Implement `multiadder`, which takes a positive integer `n` and returns an order *n* numeric function that sums an argument sequence of length *n*.

```
def multiadder(n):
    """Return a function that takes n arguments, one at a time, and adds them.
    >>> f = multiadder(3)
    >>> f(5)(6)(7)              # 6 + 7
    18
    >>> multiadder(1)(5)        # 5
    5
    >>> multiadder(2)(5)(6)     # 5 + 6
    11
    >>> multiadder(4)(5)(6)(7)(8) # 5 + 6 + 7 + 8
    26
    """
    if _______________________________________________________________:
        return ________________________________________________________
    else:
        return ________________________________________________________

def multiadder(n):
    if n == 1:
        return lambda x: x
    else:
        return lambda a: lambda b: multiadder(n-1)(a+b)
```

2. PILE

A *pile* for a tree *t* with no repeated leaf labels is a dictionary in which the label for each leaf of *t* is a key, and its value is the path from that leaf to the root. Each path from a node to the root is either an empty tuple, if the node is the root, or a two-element tuple containing the label of the node's parent and the rest of the path. Implement `pile`, which takes a tree constructed using a tree constructed with the `tree` data abstraction. It returns a *pile* for that tree. You may use the `tree`, `label`, `branches`, and `is_leaf` functional abstractions.

```
def pile(t):
    """Return a dict that contains every path from a leaf to the root of t.
    >>> pile(tree(5, [tree(3, [tree(1), tree(2)]), tree(6, [tree(7)])]))
    {1: (3, 5, ()), 2: (3, 5, ()), 7: (6, (5, ()))}
    """
    p = {}
    def gather(__________________________, __________________________):
        if is_leaf(u):
            ___________________________________________________________
        for b in branches(u):
            ___________________________________________________________
    ___________________________________________________________________
    return p
```

```
def pile(t):
    """Return a dict that contains every path from a leaf to the root of t.
    >>> pile(tree(5, [tree(3, [tree(1), tree(2)]), tree(6, [tree(7)])]))
    {1: (3, 5, ()), 2: (3, 5, ()), 7: (6, (5, ()))}
    """
    p = {}
    def gather(u, parent):
        if is_leaf(u):
            p[label(u)] = parent
        for b in branches(u):
            gather(b, (label(u), parent))
    gather(t, ())
    return p
```

3. RE-SET

Implement the `fset` function, which returns two functions that together represent a set. Both the `add` and `has` functions return whether a value is already in the set. The `add` function also adds its argument value to the set. You may assign to only one name in the assignment statement. You may not use any built-in container, such as a set or dictionary or list.

```
def fset():
    """Return two functions that together represent a set.
    >>> add, has = fset()
    >>> [add(1), add(3)]               # Neither 1 nor 3 were already added.
    [False, False]
    >>> [has(k) for k in range(5)]
    [False, True, False, True, False]
    >>> [add(3), add(2)]              # 3 was already in the set; 2 is added.
    [True, False]
    >>> [has(k) for k in range(5)]
    [False, True, True, True, False]
    """
    items = lambda x: ________________________________________________
    def add(y):
        ____________________________________________________________
        f = items
        _________ = ___________________________________________________
        return _______________________________________________________
    return add, _______________________________________________________

def fset():
    items = lambda x: False
    def add(y):
        nonlocal items
        f = items
        items = lambda x: x == y or f(x)
        return f(y)
    return add, lambda y: items(y)
```