Q1: WWPD: Representation

class A:
    def __init__(self, x):
        self.x = x

    def __repr__(self):
        return self.x

    def __str__(self):
        return self.x * 2

class B:
    def __init__(self):
        print('boo!')
        self.a = []

    def add_a(self, a):
        self.a.append(a)

    def __repr__(self):
        print(len(self.a))
        ret = ''
        for a in self.a:
            ret += str(a)
        return ret

Given the above class definitions, what will the following lines output?

```python
>>> A('one')
one

>>> print(A('one'))
oneone

>>> repr(A('two'))
```

Linked Lists, Trees

'two'

```python
>>> b = B()
```

`boo!`

```python
>>> b.add_a(A('a'))
>>> b.add_a(A('b'))
>>> b
```

2

aabb

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

There are many different implementations of sequences in Python. Today, we’ll explore the linked list implementation.

A linked list is either an empty linked list, or a Link object containing a first value and the rest of the linked list.

To check if a linked list is an empty linked list, compare it against the class attribute Link.empty:

```python
if link is Link.empty:
    print('This linked list is empty!')
else:
    print('This linked list is not empty!')
```

You can find an implementation of the Link class below:

```python
class Link:
    """A linked list."""
    empty = ()

    def __init__(self, first, rest=empty):
        assert rest is Link.empty or isinstance(rest, Link)
        self.first = first
        self.rest = rest

    def __repr__(self):
        if self.rest:
            rest_repr = ', ' + repr(self.rest)
        else:
            rest_repr = ''
        return 'Link(' + repr(self.first) + rest_repr + ')

    def __str__(self):
        string = '<'
        while self.rest is not Link.empty:
            string += str(self.first) + ' '
            self = self.rest
        return string + str(self.first) + '>'
```

Note: This worksheet is a problem bank—most TAs will not cover all the problems in discussion section.
Q2: The Hy-rules of Linked Lists

In this question, we are given the following Linked List:

```
 ganondorf = Link('zelda', Link('link', Link('sheik', Link.empty)))
```

What expression would give us the value 'sheik' from this Linked List?

```
 ganondorf.rest.rest.first
```

What is the value of `ganondorf.rest.first`?

'link'

What would be the value of `str(ganondorf)`?

'<zelda link sheik>'

What expression would mutate this linked list to `<zelda ganondorf sheik>`?

```
 ganondorf.rest.first = 'ganondorf'
```

Q3: Sum Nums

Write a function that takes in a linked list and returns the sum of all its elements.
You may assume all elements in `s` are integers. Try to implement this recursively!

```
def sum_nums(s):
    """
    >>> a = Link(1, Link(6, Link(7)))
    >>> sum_nums(a)
    14
    """
    if s == Link.empty:
        return 0
    return s.first + sum_nums(s.rest)
```
Q4: 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_lns contains at least one linked list.

def multiply_lns(lst_of_lns):
    ""
    >>> 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_lns([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_lns:
        if lnk is Link.empty:
            return Link.empty
        product *= lnk.first
    lst_of_lns_rests = [lnk.rest for lnk in lst_of_lns]
    return Link(product, multiply_lns(lst_of_lns_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_lns 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:

*Note: This worksheet is a problem bank—most TAs will not cover all the problems in discussion section.*
```python
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 Links in our list of Links 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.

Note: This worksheet is a problem bank—most TAs will not cover all the problems in discussion section.
Q5: Flip Two

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

```python
def flip_two(s):
    """
    >>> one_lnk = Link(1)
    >>> flip_two(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)))))
    """
    # Recursive solution:
    if s is Link.empty or s.rest is Link.empty:
        return
    s.first, s.rest.first = s.rest.first, s.first
    flip_two(s.rest.rest)

    # For an extra challenge, try writing out an iterative approach as well below!
    return # separating recursive and iterative implementations

    # Iterative approach
    while s is not Link.empty and s.rest is not Link.empty:
        s.first, s.rest.first = s.rest.first, s.first
        s = s.rest.rest
```

If there’s only a single item (or no item) to flip, then we’re done.

Otherwise, we swap the contents of the first and second items in the list. Since we’ve handled the first two items, we then need to recurse on

Although the question explicitly asks for a recursive solution, there is also a fairly similar iterative solution (see python solution).

We will advance `s` until we see there are no more items or there is only one more Link object to process. Processing each Link involves swapping the contents of the first and second items in the list (same as the recursive solution).
Trees

We define a tree to be a recursive data abstraction that has a label (the value stored in the root of the tree) and branches (a list of trees directly underneath the root). Previously, we implemented the tree abstraction using Python lists. Let’s look at another implementation using objects instead:

```python
class Tree:
    def __init__(self, label, branches=[]):
        for b in branches:
            assert isinstance(b, Tree)
        self.label = label
        self.branches = branches

    def is_leaf(self):
        return not self.branches
```

With this implementation, we can mutate a tree using attribute assignment, which wasn’t possible in the previous implementation using lists. That’s why we sometimes call these objects “mutable trees.”

```python
>>> t = Tree(3, [Tree(4), Tree(5)])
>>> t.label = 5
>>> t.label
5
```
Q6: Make Even

Define a function `make_even` which takes in a tree `t` whose values are integers, and mutates the tree such that all the odd integers are increased by 1 and all the even integers remain the same.

```python
def make_even(t):
    """
    >>> t = Tree(1, [Tree(2, [Tree(3)]), Tree(4), Tree(5)])
    >>> make_even(t)
    >>> t.label
    2
    >>> t.branches[0].branches[0].label
    4
    """
    if t.label % 2 != 0:
        t.label += 1
    for branch in t.branches:
        make_even(branch)
    return

# Alternate Solution

t.label += t.label % 2
for branch in t.branches:
    make_even(branch)
return
```

Note: This worksheet is a problem bank—most TAs will not cover all the problems in discussion section.
Q7: Leaves

Write a function `leaves` that returns a list of all the label values of the leaf nodes of a Tree.

```python
def leaves(t):
    """Returns a list of all the labels of the leaf nodes of the Tree t."

    >>> leaves(Tree(1))
    [1]
    >>> leaves(Tree(1, [Tree(2, [Tree(3)]), Tree(4)]))
    [3, 4]
    """
    if t.is_leaf():
        return [t.label]
    all_leaves = []
    for b in t.branches:
        all_leaves += leaves(b)
    return all_leaves
```

*Note: This worksheet is a problem bank—most TAs will not cover all the problems in discussion section.*
Q8: Find Paths

**Hint:** This question is similar to `find_path` on Discussion 05.

Define the procedure `find_paths` that, given a Tree `t` and an `entry`, returns a list of lists containing the nodes along each path from the root of `t` to `entry`. You may return the paths in any order.

For instance, for the following tree `tree_ex`, `find_paths` should behave as specified in the function doctests.

```python
def find_paths(t, entry):
    """
    >>> tree_ex = Tree(2, [Tree(7, [Tree(3), Tree(6, [Tree(5), Tree(11)])]), Tree(1, [Tree(5)])])
    >>> find_paths(tree_ex, 5)
    [[2, 7, 6, 5], [2, 1, 5]]
    >>> find_paths(tree_ex, 12)
    []
    """
    paths = []
    if t.label == entry:
        paths.append([t.label])
    for b in t.branches:
        for path in find_paths(b, entry):
            paths.append([t.label] + path)
    return paths
```

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