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 []
    elif x == y:
        return [[x]]
    else:
```

```
a = paths(x + 1, y)
b = paths(x * 2, y)
return [[x] + subpath for subpath 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 from earlier in the notes. The result will be a sorted sequence.

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

```python
def mergesort(seq):

    if len(seq) <= 1:
        return seq
    else:
        return merge(mergesort(seq[:len(seq)//2]),
            mergesort(seq[len(seq)//2:]))
```

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.

```python
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)))))]
    """

    paths = []
    if n <= 0 and tree.is_leaf():
        paths.append(Link(tree.label))
    for b in tree.branches:
        for path in long_paths(b, n - 1):
            paths.append(Link(tree.label, path))
    return paths
```

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]
```

[[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]
```

Index Error

```
>>> dogs
```

[[1, 2, 23, 2], [[1, 2, 23, 2], None, [1, 2, 23, 1, 3]], [1, 2, 23, 1, 3]]

4   Mutable Linked Lists and Trees

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

```python
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)))))
    """
```

Recursive solution:

```python
    if lnk is Link.empty or lnk.rest is Link.empty:
        return
    lnk.first, lnk.rest.first = lnk.rest.first, lnk.first
    flip_two(lnk.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:

```python
    while lnk is not Link.empty and lnk.rest is not Link.empty:
        lnk.first, lnk.rest.first = lnk.rest.first, lnk.first
        lnk = lnk.rest.rest
```

We will advance `lnk` 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).

Notice that the code is remarkably similar to the recursive implementation of `flip_two`.

Video walkthrough

5 Generators

5.1 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 g: lambda x: f(g(x)))(g)
```

Video walkthrough

5.2 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))
```

This solution does not work. The value g changes with each iteration so the bodies of the lambdas yielded change as well.

5.3   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.

```python
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 _____:


        _____


        _____


    total = next(it)
    yield total
    for element in it:
        total = f(total, element)
        yield total
```

6   Streams

6.1   Write a function `merge` that takes 2 sorted streams `s1` and `s2`, and returns a new sorted stream which contains all the elements from `s1` and `s2`.
Assume that both `s1` and `s2` have infinite length.

```
(define (merge s1 s2)



    (if _____



        _____



        _____))
```

```
(define (merge s1 s2)
    (if (< (car s1) (car s2))
        (cons-stream (car s1) (merge (cdr-stream s1) s2))
        (cons-stream (car s2) (merge s1 (cdr-stream s2)))))
```

Video walkthrough

6.2   (Adapted from Fall 2014) Implement `cycle` which returns a stream repeating the digits 1, 3, 0, 2, and 4, forever. Write `cons-stream` only once in your solution!
Hint: (3+2) % 5 == 0.

```
(define (cycle start)



    _____)
```

```
(define (cycle start)
    (cons-stream start (cycle (modulo (+ start 2) 5))))
```

Video walkthrough

7   Macros

7.1 Using macros, let's make a new special form, when, that has the following structure:

```
(when <condition>
      (<expr1> <expr2> <expr3> ...))
```

If the condition is not false (a truthy expression), all the subsequent operands are evaluated in order and the value of the last expression is returned. Otherwise, the entire when expression evaluates to okay.

```
scm> (when (= 1 0) ((/ 1 0) 'error))
okay
scm> (when (= 1 1) ((print 6) (print 1) 'a))
6
1
a
```

(a) Fill in the skeleton below to implement this without using quasiquotes.

```
(define-macro (when condition exprs)

   (list 'if_____))
```

```
(define-macro (when condition exprs)
       (list 'if condition (cons 'begin exprs) ''okay))
```

(b) Now, implement the macro using quasiquotes.

```
(define-macro (when condition exprs)

   `(if _____))
```

```
(define-macro (when condition exprs)
   `(if ,condition ,(cons 'begin exprs) 'okay))
```

7.2 Write a macro called zero-cond that takes in a list of clauses, where each clause is a two-element list containing two expressions, a predicate and a corresponding result expression. All predicates evaluate to a number. The macro should return the value of the expression corresponding to the first true predicate, *treating 0 as a false value*.

```
scm> (zero-cond
            ((0 'result1)
             ((- 1 1) 'result2)
             ((* 1 1) 'result3)
             (2 'result4)))
result3
```

```
(define-macro (zero-cond clauses)
    (cons 'cond
        (map              _____

                   _____

                   _____)))
```

```
(define-macro (zero-cond clauses)
    (cons 'cond
        (map (lambda (clause)
               (cons `(not (= 0 ,(car clause))) (cdr clause)))
             clauses)))
```

Video walkthrough

8   SQL

Questions

Our tables:

dogs: **Name   Age   Phrase, DEFAULT="woof"**

8.1   What would SQL display? **Keep track of the contents of the table after every statement below.** Write `Error` if you think a statement would cause an error.

```
sqlite> SELECT * FROM dogs;
Fido|1|woof
Sparky|2|woof
Lassie|2|I'll save you!
Floofy|3|Much doge

sqlite> INSERT INTO dogs(age, name) VALUES ("Rover", 3);
sqlite> SELECT * FROM dogs;


Fido|1|woof
Sparky|2|woof
Lassie|2|I'll save you!
Floofy|3|Much doge
3|Rover|woof

sqlite> UPDATE dogs SET name=age, age=name WHERE name=3;
sqlite> SELECT * FROM dogs;


Fido|1|woof
Sparky|2|woof
Lassie|2|I'll save you!
Floofy|3|Much doge
Rover|3|woof

sqlite> UPDATE dogs SET phrase="Hi there!" WHERE name LIKE "F%";
sqlite> SELECT * FROM dogs;


Fido|1|Hi there!
Sparky|2|woof
Lassie|2|I'll save you!
Floofy|3|Hi there!
Rover|3|woof

sqlite> DELETE FROM dogs WHERE age < 3;
sqlite> SELECT * FROM dogs;
```

Floofy|3|Hi there!
Rover|3|woof

sqlite> INSERT INTO dogs VALUES ("Spot", 2), ("Buster", 4);

Error: table dogs has 3 columns but 2 values were supplied

sqlite> INSERT INTO dogs(name, phrase) VALUES ("Spot", "bark"), ("Buster", "barkbark");
sqlite> SELECT * FROM dogs;

Floofy|3|Hi there!
Rover|3|woof
Spot||bark
Buster||barkbark

sqlite> INSERT INTO dogs(name, age) SELECT name, phrase from dogs where age = 3;
sqlite> DELETE FROM dogs WHERE phrase != "woof";
sqlite> SELECT * FROM dogs;

Rover|3|woof
Floofy|Hi there!|woof
Rover|woof|woof