For each of the following problems, assume linked lists are defined as follows:

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

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

```python
if link is Link.empty:
    print('This linked list is empty!')
```
1. What will Python output? Draw box-and-pointer diagrams to help determine this.

```python
>>> a = Link(1, Link(2, Link(3)))
>>> a.first

>>> a.first = 5
>>> a.first

>>> a.rest.first

>>> a.rest.rest.rest.first

>>> a.rest.rest.rest = a
>>> a.rest.rest.rest.rest.first
```
2. Write a function `skip`, which takes in a `Link` and returns a new `Link` with every other element skipped.

```python
def skip(lst):
    """
    >>> a = Link(1, Link(2, Link(3, Link(4))))
    >>> a
    Link(1, Link(2, Link(3, Link(4))))
    >>> b = skip(a)
    >>> b
    Link(1, Link(3))
    >>> a
    Link(1, Link(2, Link(3, Link(4)))) # Original is unchanged
    """
    if ____________________________________________:
        ____________________________________________:
    elif ____________________________________________:
        ____________________________________________:
    """
```

3. Now write function `skip` by mutating the original list, instead of returning a new list. Do NOT call the `Link` constructor.

```python
def skip(lst):
    """
    >>> a = Link(1, Link(2, Link(3, Link(4))))
    >>> b = skip(a)
    >>> b
    None
    >>> a
    Link(1, Link(3))
    """
```
4. Write a function `reverse`, which takes in a `Link` and returns a new `Link` that has the order of the contents reversed.  
   Hint: You may want to use a helper function if you’re solving this recursively.

   ```python
   def reverse(lst):
       ""
       >>> a = Link(1, Link(2, Link(3)))
       >>> b = reverse(a)
       >>> b
       Link(3, Link(2, Link(1)))
       >>> a
       Link(1, Link(2, Link(3)))
       ""
   ```

---

**Midterm Review**

For each of the following problems, assume the `Tree` class is defined as follows:

```python
class Tree:
    def __init__(self, label, branches=[]):
        self.label = label
        self.branches = branches

    def is_leaf(self):
        return not self.branches
```
1. Write a function that returns true only if there exists a path from root to leaf that contains at least \(n\) instances of \(\text{elem}\) in a tree \(t\).

```python
def contains_n(elem, n, t):
    """
    >>> t1 = Tree(1, [Tree(1, [Tree(2)])])
    >>> contains(1, 2, t1)
    True
    >>> contains(2, 2, t1)
    False
    >>> contains(2, 1, t1)
    True
    >>> t2 = Tree(1, [Tree(2), Tree(1, [Tree(1), Tree(2)])])
    >>> contains(1, 3, t2)
    True
    >>> contains(2, 2, t2) # Not on a path
    False
    ""
    if n == 0:
        return True
    elif _____________________________________________:
        return ______________________________________
    elif t.label == elem:
        return ______________________________________
    else:
        return ______________________________________
```
2. Define the function `factor_tree` which returns a factor tree. Recall that in a factor tree, multiplying the leaves together is the prime factorization of the root, \( n \). See below for an example of a factor tree for \( n = 20 \).

```
def factor_tree(n):
    for i in ____________________:
        if ________________________:
            return Tree(_____, _____________________________)
```

\[
\begin{array}{c}
20 \\
/ \  \\
2  10 \\
/  \\
2  5
\end{array}
\]
3. Draw the environment diagram that results from running the following code. If the code errors, draw the environment diagram up to the point that the error occurs.

```python
earth = [0]
earth.append([earth])

def wind(fire, groove):
    fire[1][0][0] = groove
def fire():
    nonlocal fire
    fire = lambda fantasy: earth.pop(1).extend(fantasy)
    return fire(groove)
return fire()

sep = earth[1]
wind(earth, [earth[0]] + [earth.append(0)])
```