Question 7 Village (adapted from Su20 MT)

The `village` operation takes

- a function `apple` that maps an integer to a tree where every label is an integer.
- a tree `t` whose labels are all integers

...and applies `apple` to every label in `t`.

To recombine this tree of trees into a single tree, simply copy all its branches to each of the leaves of the new tree.

For example, if we have

```
apple = lambda x: Tree(x, [Tree(x + 1), Tree(x + 2)])
```

and

```
t =         10
          /    \
        20      30
```

We should get the following output:

```
village(apple, t)
  =                   10
                   /       \
                 /            \
               11               12
             /    \           /    \
           20      30       20      30
          / \     /  \     /  \    /  \
        21  22  31    32  21  22  31  32
```

```
def village(apple, t):
    """Takes
        - a function `apple` that maps an integer to a tree where every
          label is an integer.
        - a tree `t` whose labels are all integers
          ...and applies `apple` to every label in `t`.
    >>> t = Tree(10, [Tree(20), Tree(30)])
    >>> apple = lambda x: Tree(x, [Tree(x + 1), Tree(x + 2)])
    >>> print_tree(village(apple, t))
    10
      11
        20
          21
          22
        30
          31
          32
      12
        20
          21
          22
        30
          31
          32
    """
    def graft(t, bs):
        """
        Grafts the given branches `bs` onto each leaf
        of the given tree `t`, returning a new tree.
        """
        if ___(c)___:
            return ___(d)___
        new_branches = ___(e)___
        return __________(f)__________
    base_t = ___(a)___
    bs = ___(b)___
    return graft(base_t, bs)
```

(a) Fill in blank (a)

- ◯ `t`
- ◯ `graft(t, t.branches)`
- ◯ `apple(t)`
- ◯ `apple(t.label)`

(b) Fill in blank (b)

- ◯ `t.branches`
- ◯ `[apple(b.label) for b in t.branches]`
- ◯ `[village(apple, b) for b in t.branches]`
- ◯ `[graft(b, b.branches) for b in t.branches]`

(c) Fill in blank (c)

(d) Fill in blank (d)

(e) Fill in blank (e)

(f) Fill in blank (f)

- ◯ `Tree(t.label, new_branches)`
- ◯ `graft(base_t, bs)`
- ◯ `Tree(apple(t.label), new_branches)`
- ◯ `Tree(t.label, [apple(b.label) for b in t.branches])`

7. (9.0 points) Hills

(a) (9.0 points) Hill

Implement the generator function `hill` which takes in a positive integer `n` and returns a generator that yields every subsequence of `n` where each digit is exactly 1 away from its adjacent digits. The order in which numbers are yielded does not matter. Assume all digits in the number are unique.

```
def hill(n):
    """
    Accepts a positive integer N, and returns a generator that
    yields every subsequence of N where each digit is exactly 1
    away from its adjacent digits.

    >>> sorted(list(hill(354)))
    [3, 4, 5, 34, 54]
    >>> sorted(list(hill(246))) # individual digits are hills themselves
    [2, 4, 6]
    >>> sorted(list(hill(32451)))
    [1, 2, 3, 4, 5, 21, 32, 34, 45, 321, 345]
    """
    _________
       (a)
    if n >= 10:
        _________:
           (b)
            _________
               (c)
            if _________ == 1:
                  (d)
                _________
                   (e)
```

i. (2.0 pt) Fill in blank (a).

ii. (1.5 pt) Fill in blank (b).

- ◯ `for x in hill(n - 1)`
- ◯ `for x in hill(n // 10)`
- ◯ `for x in range(n)`
- ◯ `for x in range(n + 1)`
- ◯ `while True`
- ◯ `while n > 0`
- ◯ `if n > 0`
- ◯ `if n % 10`

iii. (2.0 pt) Fill in blank (c).

iv. (1.5 pt) Fill in blank (d).

- ◯ `x // pow(10, n)`
- ◯ `n // x`
- ◯ `abs(x - n % 10)`
- ◯ `abs(x // 10 - n % 10)`
- ◯ `abs(x % 10 - n % 10)`
- ◯ `abs(n)`
- ◯ `x % 10 - n // 10`

v. (2.0 pt) Fill in blank (e).

4. (15.0 points) Linked List Comprehension

In Python, we can use list comprehensions to quickly generate lists from other lists—list comprehensions include an expression that is applied to each element in the list, and can optionally include an `if` clause.

```
>>> a = [1, 2, 3, 4, 5]
>>> [i ** 2 for i in a if i % 2 == 0]
[4, 16]
```

We often refer to the expression as a "mapping expression", and the `if` clause as a "filter clause". `i ** 2` is the mapping expression in the above example, and `i % 2 == 0` is the filter clause.

In this problem, we will write a function that works similarly for linked lists:

```
>>> b = Link(1, Link(2, Link(3, Link(4, Link(5)))))
>>> link_comp(b, lambda x: x ** 2, lambda x: x % 2 == 0)
Link(4, Link(16))
```

You will implement this function both recursively and iteratively.

(a) (7.0 points) Recursive Version

Fill in the definition of the below function `link_comp_recur` to execute a "list comprehension" over a linked list `lnk`, using the functions `map_func` and `filter_func`.

`map_func` is a function that takes in one argument and returns one value, and `filter_func` is a one-argument function that will always return a boolean value.

`link_comp_recur` should return a *new* linked list that is identical to `lnk`, but only keeping each `Link` for whom calling `filter_func` on the `first` of that `Link` returns `True`. Additionally, the `first` of each `Link` in your new list should be equal to the `first` of the corresponding `Link` in `lnk` with `map_func` applied.

Note: `filter_func` is always applied *before* `map_func`—we check whether a link should be filtered before applying our mapping function to it.

```
def link_comp_recur(lnk, map_func, filter_func):
    """
    >>> lnk = Link(1, Link(2, Link(3, Link(4, Link(5)))))
    >>> print(lnk)
    <1 2 3 4 5>
    >>> add_one = lambda x: x + 1
    >>> is_even = lambda x: x % 2 == 0
    print(link_comp_recur(lnk, add_one, is_even))
    <3 5>
    >>> square = lambda x: x ** 2
    >>> greater_than_2 = lambda x: x > 2
    print(link_comp_recur(lnk, square, greater_than_2))
    <9 16 25>
    """
    if __________:
          (a)
        return Link.empty
    new_rest = __________
                   (b)
    if __________:
          (c)
        return __________
                   (d)
    else:
        return new_rest
```

i. (1.0 pt) Fill in blank (a)

ii. (2.0 pt) Fill in blank (b)

iii. (2.0 pt) Fill in blank (c)

iv. (2.0 pt) Fill in blank (d)

(b) (7.0 points) Iterative Version

Next, fill in the behavior of `link_comp_iter`. `link_comp_iter` should behave exactly the same as `link_comp_recur`, but internally it is implemented using iteration rather than recursion. For simplicity, you can assume that `lnk` will never be equal to `Link.empty`, and that we will never filter out *every* value of the list.

```
def link_comp_iter(lnk, map_func, filter_func):
    """
    >>> lnk = Link(1, Link(2, Link(3, Link(4, Link(5)))))
    >>> print(lnk)
    <1 2 3 4 5>
    >>> add_one = lambda x: x + 1
    >>> is_even = lambda x: x % 2 == 0
    print(link_comp_iter(lnk, add_one, is_even))
    <3 5>
    >>> square = lambda x: x ** 2
    >>> greater_than_2 = lambda x: x > 2
    print(link_comp_iter(lnk, square, greater_than_2))
    <9 16 25>
    """
    while __________:
            (a)
        lnk = lnk.rest
    front = Link(map_func(lnk.first))
    end = front
    lnk = lnk.rest
    while lnk is not Link.empty:
        if __________:
              (b)
            end.rest = __________
                           (c)

            __________
                (d)

        __________
           (e)
    return front
```

i. (1.0 pt) Fill in blank (a)

ii. (2.0 pt) Fill in blank (b)

iii. (1.0 pt) Fill in blank (c)

iv. **(2.0 pt)** Fill in blank (d)

- ◯ `lnk = lnk.rest`
- ◯ `lnk.first = map_func(lnk.first)`
- ◯ `end = end.rest`
- ◯ `end, lnk = end.rest, lnk.rest`
- ◯ `end.first = map_func(end.first)`
- ◯ `end, lnk = lnk, end`

v. **(1.0 pt)** Fill in blank (e)

(c) **(1.0 points)** **Efficiency**

i. **(1.0 pt)** Which of the following describes the efficiency of `link_comp_recur` and `link_comp_iter`, relative to the length of the linked list `lnk`? Efficiency is the same for both functions.

- ◯ Constant
- ◯ Logarithmic
- ◯ Linear
- ◯ Quadratic
- ◯ Exponential