Question 7 Village (adapted from Su'20 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).

 (Fa'22 MT2)

5. (7.0 points) Aim for 100

The function `count_subsets` takes as input a list of positive integers `s`. It returns the number of lists that sum to 100 and contain a subset of the elements of `s` in order.

```
def count_subsets(s):
    """
    >>> count_subsets([25, 50, 75, 100, 125, 150])  # [25, 75], [100]
    2
    >>> count_subsets([25, 50, 25, 75])  # [25, 75] (first 25), [25, 75] (second 25), [25, 50, 25]
    3
    >>> count_subsets(list(range(1,10000)))
    444793
    """
```

(a) (5.0 points)

Complete the following implementation of `count_subsets`.

```
def count_subsets(s):
    def helper(sum_so_far, index):
        if ___(a)___:
            if ___(b)___:
                return 1
            return 0
        return ___(c)___ + ___(d)___
    return helper(0,0)
```

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

- ◯ `index == s`
- ◯ `index == len(s)`
- ◯ `sum_so_far == 100`
- ◯ `sum_so_far != 100`

ii. (1.0 pt) Fill in blank (b).

- ◯ `index == s`
- ◯ `index == len(s)`
- ◯ `sum_so_far == 100`
- ◯ `sum_so_far != 100`

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

- ◯ `count_subsets(s[index:])`
- ◯ `count_subsets(s[1:])`
- ◯ `helper(sum_so_far, index)`
- ◯ `helper(sum_so_far, index + 1)`

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

(b) (2.0 points)

i. (1.0 pt) What is the order of growth of the time required to evaluate `count_subsets`, in terms of the length of `s`?

◯ Exponential

◯ Quadratic

◯ Linear

◯ Logarithmic

◯ Constant

ii. (1.0 pt) We decide to rewrite `count_subsets`, following a different approach:

```
def count_subsets(s):
    values = [1]+[0]*100
    for i in s:
        for j in reversed(range(100-i+1)):
            values[j+i]+=values[j]
    return values[100]
```

What is the order of growth of the time required to evaluate the new version of `count_subsets`, in terms of the length of `s`?

◯ Exponential

◯ Quadratic

◯ Linear

◯ Logarithmic

◯ Constant