Homework 5 Solutions **hw05.zip (hw05.zip)**

Solution Files

You can find the solutions in hw05.py (hw05.py).

Required Questions

Q1: Infinite Hailstone

Write a generator function that yields the elements of the hailstone sequence starting at number `n`. After reaching the end of the hailstone sequence, the generator should yield the value 1 indefinitely.

Here is a quick reminder of how the hailstone sequence is defined:

1. Pick a positive integer `n` as the start.
2. If `n` is even, divide it by 2.
3. If `n` is odd, multiply it by 3 and add 1.
4. Continue this process until `n` is 1.

Try to write this generator function recursively. If you are stuck, you can first try writing it iteratively and then seeing how you can turn that implementation into a recursive one.

> **Hint:** Since `hailstone` returns a generator, you can `yield from` a call to `hailstone`!

```
def hailstone(n):
    """
    Yields the elements of the hailstone sequence starting at n.
    At the end of the sequence, yield 1 infinitely.

    >>> hail_gen = hailstone(10)
    >>> [next(hail_gen) for _ in range(10)]
    [10, 5, 16, 8, 4, 2, 1, 1, 1, 1]
    >>> next(hail_gen)
    1
    """
    yield n
    if n == 1:
        yield from hailstone(n)
    elif n % 2 == 0:
        yield from hailstone(n // 2)
    else:
        yield from hailstone(n * 3 + 1)
```

Use Ok to test your code:

```
python3 ok -q hailstone
```

Q2: Merge

Definition: An *infinite iterator* is a iterator that never stops providing values when `next` is called. For example, `ones()` evaluates to an infinite iterator:

```
def ones():
    while True:
        yield 1
```

Write a generator function `merge(a, b)` that takes two infinite iterators, `a` and `b`, as inputs. Both iterators yield elements in strictly increasing order with no duplicates. The generator should produce all unique elements from both input iterators in increasing order, ensuring no duplicates.

> **Note:** The input iterators do not contain duplicates within themselves, but they may have common elements between them.

```
def merge(a, b):
    """
    Return a generator that has all of the elements of generators a and b,
    in increasing order, without duplicates.

    >>> def sequence(start, step):
    ...     while True:
    ...         yield start
    ...         start += step
    >>> a = sequence(2, 3) # 2, 5, 8, 11, 14, ...
    >>> b = sequence(3, 2) # 3, 5, 7, 9, 11, 13, 15, ...
    >>> result = merge(a, b) # 2, 3, 5, 7, 8, 9, 11, 13, 14, 15
    >>> [next(result) for _ in range(10)]
    [2, 3, 5, 7, 8, 9, 11, 13, 14, 15]
    """
    a_val, b_val = next(a), next(b)
    while True:
        if a_val == b_val:
            yield a_val
            a_val, b_val = next(a), next(b)
        elif a_val < b_val:
            yield a_val
            a_val = next(a)
        else:
            yield b_val
            b_val = next(b)
```

Use Ok to test your code:

```
python3 ok -q merge
```

Q3: Stair Ways

Imagine that you want to go up a staircase that has `n` steps, where `n` is a positive integer. You can take either **one** or **two steps** each time you move.

Write a generator function `stair_ways` that yields all the different ways you can climb the staircase.

Each "way" of climbing a staircase can be represented by a list of 1s and 2s, where each number indicates whether you take one step or two steps at a time.

For example, for a staircase with 3 steps, there are three ways to climb it:

- You can take one step each time: `[1, 1, 1]`.
- You can take two steps then one step: `[2, 1]`.
- You can take one step then two steps: `[1, 2]`. .

Therefore, `stair_ways(3)` should yield `[1, 1, 1]`, `[2, 1]`, and `[1, 2]`. These can be yielded in any order.

```
def stair_ways(n):
    """
    Yield all the ways to climb a set of n stairs taking
    1 or 2 steps at a time.

    >>> list(stair_ways(0))
    [[]]
    >>> s_w = stair_ways(4)
    >>> sorted([next(s_w) for _ in range(5)])
    [[1, 1, 1, 1], [1, 1, 2], [1, 2, 1], [2, 1, 1], [2, 2]]
    >>> list(s_w) # Ensure you're not yielding extra
    []
    """
    if n == 0:
        yield []
    elif n == 1:
        yield [1]
    else:
        for way in stair_ways(n - 1):
            yield [1] + way
        for way in stair_ways(n - 2):
            yield [2] + way
```

Use Ok to test your code:

```
python3 ok -q stair_ways
```

Q4: Yield Paths

Write a generator function `yield_paths` that takes a tree `t` and a target `value`. It yields each path from the root of `t` to any node with the label `value`.

Each path should be returned as a list of labels from the root to the matching node. The paths can be yielded in any order.

Hint: If you are having trouble getting started, think about how you would approach this problem if it was not a generator function. What would the recursive steps look like?

Hint: Remember, you can iterate over generator objects because they are a type of iterator!

```
def yield_paths(t, value):
    """
    Yields all possible paths from the root of t to a node with the label
    value as a list.

    >>> t1 = tree(1, [tree(2, [tree(3), tree(4, [tree(6)]), tree(5)]), tree(5)])
    >>> print_tree(t1)
    1
      2
        3
        4
          6
        5
      5
    >>> next(yield_paths(t1, 6))
    [1, 2, 4, 6]
    >>> path_to_5 = yield_paths(t1, 5)
    >>> sorted(list(path_to_5))
    [[1, 2, 5], [1, 5]]

    >>> t2 = tree(0, [tree(2, [t1])])
    >>> print_tree(t2)
    0
      2
        1
          2
            3
            4
              6
            5
          5
    >>> path_to_2 = yield_paths(t2, 2)
    >>> sorted(list(path_to_2))
    [[0, 2], [0, 2, 1, 2]]
    """
    if label(t) == value:
        yield [value]
    for b in branches(t):
        for path in yield_paths(b, value):
            yield [label(t)] + path
```

Use Ok to test your code:

```
python3 ok -q yield_paths
```

If our current label is equal to `value`, we've found a path from the root to a node containing `value` containing only our current label, so we should yield that. From there, we'll see if there are any paths starting from one of our branches that ends at a node containing `value`. If we find these "partial paths" we can simply add our current label to the beinning of a path to obtain a path starting from the root.

In order to do this, we'll create a generator for each of the branches which yields these "partial paths". By calling `yield_paths` on each of the branches, we'll create exactly this generator! Then, since a generator is also an iterable, we can iterate over the paths in this generator and yield the result of concatenating it with our current label.

Check Your Score Locally

You can locally check your score on each question of this assignment by running

```
python3 ok --score
```

This does NOT submit the assignment! When you are satisfied with your score, submit the assignment to Gradescope to receive credit for it.

Submit Assignment

Submit this assignment by uploading any files you've edited **to the appropriate Gradescope assignment.** Lab 00 (../../lab/lab00/#submit-with-gradescope) has detailed instructions.

Exam Practice

Homework assignments will also contain prior exam questions for you to try. These questions have no submission component; feel free to attempt them if you'd like some practice!

1. Summer 2016 Final Q8: Zhen-erators Produce Power (https://inst.eecs.berkeley.edu//~cs61a/su16/assets/pdfs/61a-su16-final.pdf#page=13)
2. Spring 2018 Final Q4(a): Apply Yourself (https://inst.eecs.berkeley.edu/~cs61a/sp18/assets/pdfs/61a-sp18-final.pdf#page=5)