Homework 5 Solutions hw05.zip (hw05.zip)

Solution Files

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

Required Questions

Q1: Merge

Write a generator function `merge` that takes in two infinite generators `a` and `b` that are in increasing order without duplicates and returns a generator that has all the elements of both generators, in increasing order, without duplicates.

```
def merge(a, b):
    """
    >>> 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]
    """
    first_a, first_b = next(a), next(b)
    while True:
        if first_a == first_b:
            yield first_a
            first_a, first_b = next(a), next(b)
        elif first_a < first_b:
            yield first_a
            first_a = next(a)
        else:
            yield first_b
            first_b = next(b)
```

Use Ok to test your code:

```
python3 ok -q merge
```

Q2: Generate Permutations

Given a sequence of unique elements, a *permutation* of the sequence is a list containing the elements of the sequence in some arbitrary order. For example, `[2, 1, 3]`, `[1, 3, 2]`, and `[3, 2, 1]` are some of the permutations of the sequence `[1, 2, 3]`.

Implement `gen_perms`, a generator function that takes in a sequence `seq` and returns a generator that yields all permutations of `seq`. For this question, assume that `seq` will not be empty.

Permutations may be yielded in any order. Note that the doctests test whether you are yielding all possible permutations, but not in any particular order. The built-in `sorted` function takes in an iterable object and returns a list containing the elements of the iterable in non-decreasing order.

> *Hint:* If you had the permutations of all the elements in `seq` not including the first element, how could you use that to generate the permutations of the full `seq`?

> *Hint:* Remember, it's possible to loop over generator objects because generators are iterators!

```
def gen_perms(seq):
    """Generates all permutations of the given sequence. Each permutation is a
    list of the elements in SEQ in a different order. The permutations may be
    yielded in any order.

    >>> perms = gen_perms([100])
    >>> type(perms)
    <class 'generator'>
    >>> next(perms)
    [100]
    >>> try: #this piece of code prints "No more permutations!" if calling next would
    ...     next(perms)
    ... except StopIteration:
    ...     print('No more permutations!')
    No more permutations!
    >>> sorted(gen_perms([1, 2, 3])) # Returns a sorted list containing elements of th
    [[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]
    >>> sorted(gen_perms((10, 20, 30)))
    [[10, 20, 30], [10, 30, 20], [20, 10, 30], [20, 30, 10], [30, 10, 20], [30, 20, 10
    >>> sorted(gen_perms("ab"))
    [['a', 'b'], ['b', 'a']]
    """
    if not seq:
        yield []
    else:
        for perm in gen_perms(seq[1:]):
            for i in range(len(seq)):
                yield perm[:i] + [seq[0]] + perm[i:]
```

Use Ok to test your code:

```
python3 ok -q gen_perms
```

Q3: Yield Paths

Define a generator function `yield_paths` which takes in a tree `t`, a value `value`, and returns a generator object which yields each path from the root of `t` to a node that has label `value`.

Each path should be represented as a list of the labels along that path in the tree. You may yield the paths in any order.

We have provided a skeleton for you. You do not need to use this skeleton, but if your implementation diverges significantly from it, you might want to think about how you can get it to fit the skeleton.

```
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
```

> *Hint:* If you're having trouble getting started, think about how you'd approach this problem if it wasn't a generator function. What would your recursive calls be? With a generator function, what happens if you make a "recursive call" within its body?

> *Hint:* Remember, it's possible to loop over generator objects because generators are iterators!

> Note: Remember that this problem should **yield items** -- do not return a list!

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.

Submit

Make sure to submit this assignment by running:

```
python3 ok --submit
```

Optional Questions

Q4: Infinite Hailstone

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

Here's 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're 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
```

Q5: Remainder Generator

Like functions, generators can also be *higher-order*. For this problem, we will be writing `remainders_generator`, which yields a series of generator objects.

`remainders_generator` takes in an integer `m`, and yields `m` different generators. The first generator is a generator of multiples of `m`, i.e. numbers where the remainder is 0. The second is a generator of natural numbers with remainder 1 when divided by `m`. The last generator yields natural numbers with remainder `m - 1` when divided by `m`.

> *Hint*: To create a generator of infinite natural numbers, you can call the `naturals` function that's provided in the starter code.

> *Hint*: Consider defining an inner generator function. Each yielded generator varies only in that the elements of each generator have a particular remainder when divided by `m`. What does that tell you about the argument(s) that the inner function should take in?

```python
def remainders_generator(m):
    """
    Yields m generators. The ith yielded generator yields natural numbers whose
    remainder is i when divided by m.

    >>> import types
    >>> [isinstance(gen, types.GeneratorType) for gen in remainders_generator(5)]
    [True, True, True, True, True]
    >>> remainders_four = remainders_generator(4)
    >>> for i in range(4):
    ...     print("First 3 natural numbers with remainder {0} when divided by 4:".form
    ...     gen = next(remainders_four)
    ...     for _ in range(3):
    ...         print(next(gen))
    First 3 natural numbers with remainder 0 when divided by 4:
    4
    8
    12
    First 3 natural numbers with remainder 1 when divided by 4:
    1
    5
    9
    First 3 natural numbers with remainder 2 when divided by 4:
    2
    6
    10
    First 3 natural numbers with remainder 3 when divided by 4:
    3
    7
    11
    """
    def gen(i):
        for e in naturals():
            if e % m == i:
                yield e
    for i in range(m):
        yield gen(i)
```

Note that if you have implemented this correctly, each of the generators yielded by `remainder_generator` will be *infinite* - you can keep calling `next` on them forever without running into a `StopIteration` exception.

Use Ok to test your code:

```
python3 ok -q remainders_generator
```

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 2018 Final Q7a,b: Streams and Jennyrators (https://inst.eecs.berkeley.edu/~cs61a/su18/assets/pdfs/61a-su18-final.pdf#page=9)
2. Spring 2019 Final Q1: Iterators are inevitable (https://cs61a.org/exam/sp19/final/61a-sp19-final.pdf#page=2)
3. Spring 2021 MT2 Q8: The Tree of L-I-F-E (https://cs61a.org/exam/sp21/mt2/61a-sp21-mt2.pdf#page=18)
4. Summer 2016 Final Q8: Zhen-erators Produce Power (https://inst.eecs.berkeley.edu//~cs61a/su16/assets/pdfs/61a-su16-final.pdf#page=13)
5. Spring 2018 Final Q4a: Apply Yourself (https://inst.eecs.berkeley.edu/~cs61a/sp18/assets/pdfs/61a-sp18-final.pdf#page=5)