Lab 4 Solutions **lab04.zip (lab04.zip)**

Solution Files

Topics

Consult this section if you need a refresher on the material for this lab. It's okay to skip directly to the questions and refer back here should you get stuck.

Required Questions

Recursion

Q1: WWPD: Journey to the Center of the Earth

Use Ok to test your knowledge with the following "What Would Python Display?" questions:

```
python3 ok -q drill-wwpd -u
```

Important: For all WWPD questions, type `Function` if you believe the answer is `<function...>`, `Error` if it errors, and `Nothing` if nothing is displayed.

```
>>> def crust():
...     print("70km")
...     def mantle():
...         print("2900km")
...         def core():
...             print("5300km")
...             return mantle()
...         return core
...     return mantle
>>> drill = crust
>>> drill = drill()
______

>>> drill = drill()
______

>>> drill = drill()
______

>>> drill()
______
```

Q2: Summation

Write a recursive implementation of `summation`, which takes a positive integer `n` and a function `term`. It applies `term` to every number from `1` to `n` including `n` and returns the sum.

Important: Use recursion; the tests will fail if you use any loops (for, while).

```
def summation(n, term):
    """Return the sum of numbers 1 through n (including n) with term applied to each numbe
    Implement using recursion!

    >>> summation(5, lambda x: x * x * x) # 1^3 + 2^3 + 3^3 + 4^3 + 5^3
    225
    >>> summation(9, lambda x: x + 1) # 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10
    54
    >>> summation(5, lambda x: 2**x) # 2^1 + 2^2 + 2^3 + 2^4 + 2^5
    62
    >>> # Do not use while/for loops!
    >>> from construct_check import check
    >>> # ban iteration
    >>> check(HW_SOURCE_FILE, 'summation',
    ...       ['While', 'For'])
    True
    """
    assert n >= 1
    if n == 1:
        return term(n)
    else:
        return term(n) + summation(n - 1, term)
    # Base case: only one item to sum, so we return that item.
    # Recursive call: returns the result of summing the numbers up to n-1 using
    # term. All that's missing is term applied to the current value n.
```

Use Ok to test your code:

```
python3 ok -q summation
```

Tree Recursion

Q3: Pascal's Triangle

Here's a part of the Pascal's trangle:

```
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
```

Every number in Pascal's triangle is defined as the sum of the item above it and the item above and to the left of it. Use `0` if the item does not exist.

Define the procedure `pascal(row, column)` which takes a row and a column, and finds the value of the item at that position in Pascal's triangle. Rows and columns are zero-indexed; that is, the first row is row 0 instead of 1 and the first column is column 0 instead of column 1.

For example, the item at row 2, column 1 in Pascal's triangle is 2.

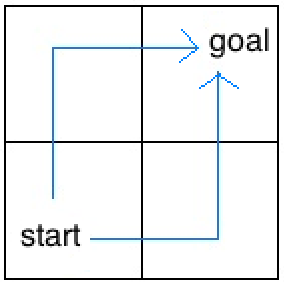
```
def pascal(row, column):
    """Returns the value of the item in Pascal's Triangle
    whose position is specified by row and column.
    >>> pascal(0, 0)
    1
    >>> pascal(0, 5)    # Empty entry; outside of Pascal's Triangle
    0
    >>> pascal(3, 2)    # Row 3 (1 3 3 1), Column 2
    3
    >>> pascal(4, 2)     # Row 4 (1 4 6 4 1), Column 2
    6
    """
    if column == 0:
        return 1
    elif row == 0:
        return 0
    else:
        return pascal(row - 1, column) + pascal(row - 1, column - 1)
```

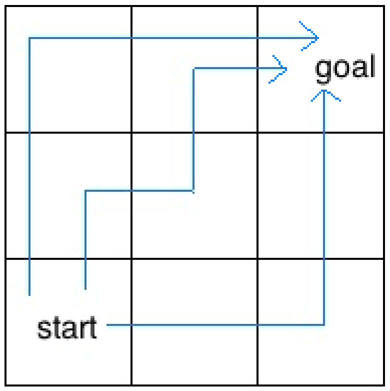
Use Ok to test your code:

```
python3 ok -q pascal
```

Q4: Insect Combinatorics

Consider an insect in an *M* by *N* grid. The insect starts at the bottom left corner, *(0, 0)*, and wants to end up at the top right corner, *(M-1, N-1)*. The insect is only capable of moving right or up. Write a function `paths` that takes a grid length and width and returns the number of different paths the insect can take from the start to the goal. (There is a closed-form solution (https://en.wikipedia.org/wiki/Closed-form_expression) to this problem, but try to answer it procedurally using recursion.)





For example, the 2 by 2 grid has a total of two ways for the insect to move from the start to the goal. For the 3 by 3 grid, the insect has 6 diferent paths (only 3 are shown above).

Hint: What happens if we hit the top or rightmost edge?

```
def paths(m, n):
    """Return the number of paths from one corner of an
    M by N grid to the opposite corner.

    >>> paths(2, 2)
    2
    >>> paths(5, 7)
    210
    >>> paths(117, 1)
    1
    >>> paths(1, 157)
    1
    """
    if m == 1 or n == 1:
        return 1
    return paths(m - 1, n) + paths(m, n - 1)
```

Use Ok to test your code:

```
python3 ok -q paths
```

List Comprehensions

Q5: Couple

Implement the function `couple`, which takes in two lists and returns a list that contains lists with i-th elements of two sequences coupled together. You can assume the lengths of two sequences are the same. Try using a list comprehension.

> *Hint*: You may find the built in range (https://www.w3schools.com/python/ref_func_range.asp) function helpful.

```
def couple(s, t):
    """Return a list of two-element lists in which the i-th element is [s[i], t[i]].

    >>> a = [1, 2, 3]
    >>> b = [4, 5, 6]
    >>> couple(a, b)
    [[1, 4], [2, 5], [3, 6]]
    >>> c = ['c', 6]
    >>> d = ['s', '1']
    >>> couple(c, d)
    [['c', 's'], [6, '1']]
    """
    assert len(s) == len(t)
    return [[s[i], t[i]] for i in range(0, len(s))]
```

Use Ok to test your code:

```
python3 ok -q couple
```

Submit

Make sure to submit this assignment by running:

```
python3 ok --submit
```

Optional Questions

Q6: Coordinates

Implement a function `coords` that takes a function `fn`, a sequence `seq`, and a `lower` and `upper` bound on the output of the function. `coords` then returns a list of coordinate pairs (lists) such that:

- Each (x, y) pair is represented as `[x, fn(x)]`
- The x-coordinates are elements in the sequence
- The result contains only pairs whose y-coordinate is within the upper and lower bounds (inclusive)

See the doctest for examples.

> *Note*: your answer can only be *one line long*. You should make use of list comprehensions!

```
def coords(fn, seq, lower, upper):
    """
    >>> seq = [-4, -2, 0, 1, 3]
    >>> fn = lambda x: x**2
    >>> coords(fn, seq, 1, 9)
    [[-2, 4], [1, 1], [3, 9]]
    """
    return [[x, fn(x)] for x in seq if lower <= fn(x) <= upper]
```

Use Ok to test your code:

```
python3 ok -q coords
```

> Reflect: What are the drawbacks to the one-line answer, in terms of using computer resources?

Q7: Riffle Shuffle

A common way of shuffling cards is known as the riffle shuffle (https://www.youtube.com/watch?v=JmbVNyIiD54). The shuffle produces a new configuration of cards in which the top card is followed by the middle card, then by the second card, then the card after the middle, and so forth.

Write a list comprehension that riffle shuffles a sequence of items. You can assume the sequence contains an even number of items.

Hint: There are two ways you can write this as a single list comprension: 1) You may find the expression `k%2`, which evaluates to 0 on even numbers and 1 on odd numbers, to be alternatively access the beginning and middle of the deck. 2) You can utilize an if expression in your comprehension for the odd and even numbers respectively.

```
def riffle(deck):
    """Produces a single, perfect riffle shuffle of DECK, consisting of
    DECK[0], DECK[M], DECK[1], DECK[M+1], ... where M is position of the
    second half of the deck.  Assume that len(DECK) is even.
    >>> riffle([3, 4, 5, 6])
    [3, 5, 4, 6]
    >>> riffle(range(20))
    [0, 10, 1, 11, 2, 12, 3, 13, 4, 14, 5, 15, 6, 16, 7, 17, 8, 18, 9, 19]
    """
    return [deck[(i % 2) * len(deck) // 2 + i // 2] for i in range(len(deck))]

#ALTERNATE SOLUTION
def riffle(deck):
    return [deck[i // 2] if i % 2 == 0 else deck[len(deck) // 2 + i // 2] for i in range(]
```

Use Ok to test your code:

```
python3 ok -q riffle
```