RECURSION AND MIDTERM REVIEW

COMPUTER SCIENCE MENTORS CS 88

March 8th to March 12th

1 Recursion

A recursive function is a function that is defined in terms of itself. One might want to use recursion when a problem is more easily expressed in terms of its sub-problems.

Here is an example of using recursion to sum all the numbers from 1 to n, assuming n is a positive integer.

```
def sum_to_n(n):
    if n == 1:
        return 1
    else:
        return n + sum_to_n(n-1)
```

The base case is usually the the simplest case that your function handles (in this case, where the input is 1) since the problem cannot be further divided into smaller sub problems.

In the recursive case, we call our function on a smaller version of the input, namely on an input size of `n` - 1, because we now want to find the sum of numbers until `n` - 1, and then simply add our current number, `n`, to the overall sum.

You may be thinking, why can't I use iteration for this? If you did, you're right! Iteration can indeed be used to write this specific function as well, and often times, functions can be expressed both iteratively and recursively. It may just be that sometimes, the recursive approach is more intuitive or simpler, and vice versa.

1. Find everything wrong with the following function. How can we fix each issue?

```
def factorial(n):
    return n * factorial(n)
```

Solution: There is no base case and the recursive call is made on the same `n`.

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
```

2. Complete the definition for `num_digits`, which takes in a number `n` and returns the number of digits it has.

```
def num_digits(n):
    """Takes in an positive integer and returns the number of
    digits.

    >>> num_digits(0)
    1
    >>> num_digits(1)
    1
    >>> num_digits(7)
    1
    >>> num_digits(1093)
    4
    """
```

Solution:

```
    if n < 10:
        return 1
    else:
        return 1 + num_digits(n // 10)
```

3. Write a function that takes two numbers **m** and **n** and returns their product. Assume the inputs are positive integers. Use recursion, not mul, *, or iteration!

```
def multiply(m, n):
    """
    >>> multiply(3, 5)
    15
    """
```

Solution:

```
    if n == 1:
        return m
    else:
        return m + multiply(m, n - 1)
```

4. Implement `sum_some`, which takes a non-negative integer n and a function p. It returns the sum of all the digits d for which p returns a true value when given d as an argument. Assume that the function p takes a single digit d (from 0 to 9) and returns either True or False.

```
def even(x):
    return x % 2 == 0

def big(x):
    return x > 5

def sum_some(n, p):
    """
    >>> sum_some(124567, even) # Sum even digits: 2 + 4 + 6
    12
    >>> sum_some(124567, big) # Sum big digits: 6 + 7
    13
    """
```

Solution:

```
    # Iterative solution
    total = 0
    while n:
        if p(n % 10):
            total += n % 10
        n = n // 10
    return total
```

Solution:

```
    # Recursive
    if n == 0:
        return 0
    else:
        x = n % 10
        if p(x):
            return x + sum_some(n // 10, p)
        return sum_some(n // 10, p)
```

2 Midterm Review

5. (a) Given a list `lst`, and an index `i`, return whether or not `num` appears at index `i` or onwards in the given `lst`.

```
def contains_num_after_i(lst, num, i):
    """
    >>> contains_num_after_i([1, 11, 3, 4, 5, 6, 7, 8], 11,
        3)
    False
    >>> contains_num_after_i([1, 2, 11, 4, 5, 6, 7, 8], 11,
        3)
    False
    >>> contains_num_after_i([1, 2, 3, 11, 5, 6, 7, 8], 11,
        3)
    True
    >>> contains_num_after_i([1, 11, 3, 4, 5, 6, 7, 11],
       11, 5)
    True
    """
```

Solution:

```
    for j in range(i, len(lst)):
        if lst[j] == num:
            return True
    return False
```

(b) Return whether or not there are duplicates in the given `lst`. Hint: Call the function above!

```
def duplicates(lst):
    """
    >>> duplicates([1, 11, 3, 4, 5, 6, 7, 8])
    False
    >>> duplicates([1, 2, 11, 4, 5, 6, 7, 8])
    False
    >>> duplicates([1, 2, 3, 11, 5, 6, 7, 3, 8])
    True
    >>> duplicates([1, 11, 4, 4, 9, 5, 6, 7, 11])
    True
    """
```

Solution:

```
    for i in range(len(lst)):
        if contains_num_after_i(lst, lst[i], i+1):
            return True
    return False
```

6. Write a function, `whole_sum`, which takes in an integer, `n`. It returns another function which takes in an integer, and returns True if the digits of that integer sum to `n` and `False` otherwise.

```
def whole_sum(n):
    """
    >>> whole_sum(21)(777)
    True
    >>> whole_sum(142)(10010101010)
    False
    """
    def check(x):

        ____________________

        while _______________:

            last = ____________________

            ____________________________

            ____________________________

        return ____________________

    return ____________________
```

Solution:

```
def whole_sum(n):
    def check(x):
        total = 0
        while x > 0:
            last = x % 10
            x = x // 10
            total += last
        return total == n
    return check
```

7. Fill in the blanks (*without using any numbers in the first blank*) such that the entire expression evaluates to `9`.

Solution:
```
(lambda x: lambda y: lambda: y(x))(3)(lambda z: z*z)()
```

8. Draw the box-and-pointer diagram.

```
>>> violet = [7, 77, 17]
>>> violet.append([violet.pop(1)])

>>> dash = violet * 2
>>> jack = dash[3:5]
>>> jackjack = jack.extend(jack)

>>> helen = list(violet)
>>> helen += [jackjack]
>>> helen[2].append(violet)
```

Solution: https://goo.gl/EAmZBW

9. Given some list `lst`, possibly a deep list, mutate `lst` to have the accumulated sum of all elements so far in the list. If there is a nested list, mutate it to similarly reflect the accumulated sum of all elements so far in the nested list. Return the total sum of `lst`.

   Hint: The **`isinstance`** function returns True for **`isinstance`**`(l, list)` if `l` is a list and False otherwise.

```
def accumulate(lst):
    """
    >>> l = [1, 5, 13, 4]
    >>> accumulate(l)
    23
    >>> l
    [1, 6, 19, 23]
    >>> deep_l = [3, 7, [2, 5, 6], 9]
    >>> accumulate(deep_l)
    32
    >>> deep_l
    [3, 10, [2, 7, 13], 32]
    """
    ____________________________________________
    for ____________________________________________:
        ____________________________________________
        if isinstance(____________________, list):
            inside = ____________________________
            ____________________________________
        else:
            ____________________________________
            ____________________________________
    ____________________________________________
```

Solution:

```
    sum_so_far = 0
    for i in range(len(lst)):
        item = lst[i]
        if isinstance(item, list):
            inside = accumulate(item)
            sum_so_far += inside
        else:
            sum_so_far += item
            lst[i] = sum_so_far
    return sum_so_far
```

In lecture, we discussed the `rational` data type, which represents fractions with the following methods:

- `rational(n, d)` - constructs a rational number with numerator `n`, denominator `d`
- `numer(x)` - returns the numerator of rational number `x`
- `denom(x)` - returns the denominator of rational number `x`

We also presented the following methods that perform operations with rational numbers:

- `add_rationals(x, y)`
- `mul_rationals(x, y)`
- `rationals_are_equal(x, y)`

There is a lot we can do with just these simple methods.

10. Implement the following rational number methods.

```
def inverse_rational(x):
    """Returns the inverse of the given non-zero rational
    number
    """
```

Solution:

```
    return rational(denom(x), numer(x))
```

```
def div_rationals(x, y):
    """Returns x / y for given rational x and non-zero
    rational y
    """
```

Solution:

```
    return mul_rationals(x, inverse_rational(y))
```