1 Recursion

Concept of Recursive Thinking

A recursive function is a function that calls itself in its body, either directly or indirectly. Recursive functions have three important components:

Base case(s), the simplest possible form of the problem you're trying to solve.
Recursive case(s), where the function calls itself with a simpler argument as part of the computation.
Using the recursive calls to solve the full problem.

Let's look at the canonical example, factorial:

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

We know by its definition that 0! is 1. So we choose n == 0 as our base case. The recursive step also follows from the definition of factorial, i.e., n! = n * (n-1)!.

The next few questions in lab will have you writing recursive functions. Here are some general tips:

- Consider how you can solve the current problem using the solution to a simpler version of the problem. Remember to trust the recursion: assume that your solution to the simpler problem works correctly without worrying about how.

- Think about what the answer would be in the simplest possible case(s). These will be your base cases - the stopping points for your recursive calls. Make sure to consider the possibility that you're missing base cases (this is a common way recursive solutions fail).

- It may help to write the iterative version first.

Practice Problem: AB+C

Implement ab_plus_c, a function that takes arguments a, b, and c and computes a * b + c. You can assume a and b are both non-negative integers. However, you can't use the * operator. Use recursion!

```
def ab_plus_c(a, b, c):
    """Computes a * b + c.

    >>> ab_plus_c(2, 4, 3)  # 2 * 4 + 3
    11
    >>> ab_plus_c(0, 3, 2)  # 0 * 3 + 2
    2
    >>> ab_plus_c(3, 0, 2)  # 3 * 0 + 2
    2
    """
    "*** YOUR CODE HERE ***"
```

Practice Problem: GCD

The greatest common divisor of two positive integers a and b is the largest integer which evenly divides both numbers (with no remainder). Euclid, a Greek mathematician in 300 B.C., realized that the greatest common divisor of a and b is one of the following:

the smaller value if it evenly divides the larger value, or
the greatest common divisor of the smaller value and the remainder of the larger value divided by the smaller value

In other words, if a is greater than b and a is not divisible by b, then

```
 gcd(a, b) = gcd(b, a % b)
```

Write the gcd function recursively using Euclid's algorithm.

```
def gcd(a, b):
    """Returns the greatest common divisor of a and b.
    Should be implemented using recursion.

    >>> gcd(34, 19)
    1
    >>> gcd(39, 91)
    13
    >>> gcd(20, 30)
    10
    >>> gcd(40, 40)
    40
    """
    "*** YOUR CODE HERE ***"
```

Practice Problem: Hailstone

For the hailstone function from homework 1, you pick a positive integer n as the start. If n is even, divide it by 2. If n is odd, multiply it by 3 and add 1. Repeat this process until n is 1. Write a recursive version of hailstone that prints out the values of the sequence and returns the number of steps.

```
def hailstone(n):
    """Print out the hailstone sequence starting at n, and return the
    number of elements in the sequence.

    >>> a = hailstone(10)
    10
    5
    16
    8
    4
    2
    1
    >>> a
    7
    """
    "*** YOUR CODE HERE ***"
```

2 Tree Recursion

The Difference between Recursion and Tree Recursion

In order to understand Tree Recursion, we must understand Recursion first.

Recursion is the idea of using a function as you are defining them. A recursive definition consists of two things:

1. Base Case
2. Recursive Step

A recursive step is where the nth value is defined in terms of the $(n-1)$th value, while a base case defines a limiting condition that would terminate the function call. Here's an example:

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

Try drawing a diagram for factorial(5).

Example	Answer
f(x)	
|	
f(x-1)	
|	
f(x-2)	
|	
f(x-3)	
|	
...	

In Tree Recursion, the computational process of a recursion could be expressed in a form of tree. Here's an example:

```
def fib(x):
    if x <= 1:
       return x
    else:
       return fib(x-2) + fib(x-1)
```

Try drawing a diagram for fib(4).

Practice Problem: Count Change

Once the machines take over, the denomination of every coin will be a power of two: 1cent, 2cent, 4cent, 8cent, 16cent, etc. There will be no limit to how much a coin can be worth.

A set of coins makes change for n if the sum of the values of the coins is n. For example, the following sets make change for 7:

7 1-cent coins
5 1-cent, 1 2-cent coins
3 1-cent, 2 2-cent coins
3 1-cent, 1 4-cent coins
1 1-cent, 3 2-cent coins
1 1-cent, 1 2-cent, 1 4-cent coins

Thus, there are 6 ways to make change for 7. Write a function count_change that takes a positive integer n and returns the number of ways to make change for n using these coins of the future:

```
def biggest_change(amount):
   """Return the biggest change that is less than or equal to the amount
   >>> biggest_change(5)
   4
   >>> biggest_change(10)
   8
   >>> biggest_change(100)
   64
   >>> biggest_change(1000)
   512
   if amount == 0:
      return 0
   n = 0
   while 2**(n+1) <= amount:
      n += 1
   return 2**n

def count_change(amount):
   """Return the number of ways to make change for amount.
    >>> count_change(7)
    6
    >>> count_change(10)
    14
    >>> count_change(20)
    60
    >>> count_change(100)
    9828
    """
    "*** YOUR CODE HERE ***"
```

Practice Problem: Your Father's Parenthesis

Suppose we have a sequence of quantities that we want to multiply together, but can only multiply two at a time. We can express the various ways of doing so by counting the number of different ways to parenthesize the sequence. For example, here are the possibilities for products of 1, 2, 3, 4 and 5 elements:

Product	a	ab	abc	$abcd$	$abcde$
Count	1	1	2	5	14
Parenthesizations	a	ab	$a(bc)$ $(ab)c$	$a(b(cd))$ $a((bc)d)$ $(ab)(cd)$ $(a(bc))d$ $((ab)c)d$	$a(b(c(de)))$ $(ab)(c(de))$ $(a((bc)d))e$ $a(b((cd)e))$ $(ab)((cd)e)$ $((ab)(cd))e$ $a((bc)(de))$ $(a(bc))(de)$ $((a(bc))d)e$ $a((b(cd))e)$ $((ab)c)(de)$ $(((ab)c)d)e$ $a(((bc)d)e)$ $(a(b(cd)))e$

Assume, as in the table above, that we don?t want to reorder elements.

Define a function count_groupings that takes a positive integer n and returns the number of ways of parenthesizing the product of n numbers. (You might not need to use all lines.)

```
def count_groupings(n):
   """ For N >= 1 , the number of distinct parenthesizations
   of a product of N items.
   >>> count_groupings(1)
   1
   >>> count_groupings(2)
   1
   >>> count_groupings(3)
   2
   >>> count_groupings(4)
   5
   >>> count_groupings(5)
   14
   """
   if n == 1:

      return ________________________________________________________

   _________________________________________________________________

   i =  ____________________________________________________________

   while  ____________________________________________________:

      ______________________________________________________________

      i += 1

   return  _________________________________________________________
```

3 Citation and Sources

Example questions and explanations are brought from the following source:

1. Hilfinger Paul, DeNero John, “CS 61A: Structure and Interpretation of Computer Programs.” CS 61A Fall 2017, cs61a.org.

©hyunjaemoon