

1 More Recursion

Questions

- 1.1 In discussion 1, we implemented the function `is_prime`, which takes in a positive integer and returns whether or not that integer is prime, iteratively.

Now, let's implement it recursively! As a reminder, an integer is considered prime if it has exactly two unique factors: 1 and itself.

```
def is_prime(n):
    """
    >>> is_prime(7)
    True
    >>> is_prime(10)
    False
    >>> is_prime(1)
    False
    """
    def prime_helper(_____):

        if _____:

            _____

        elif _____:

            _____

        else:

            _____

    return _____
```

```
def prime_helper(index):  
    if index == n:  
        return True  
    elif n % index == 0 or n == 1:  
        return False  
    else:  
        return prime_helper(index + 1)  
return prime_helper(2)
```

- 1.2 Define a function `make_fn_repeater` which takes in a one-argument function `f` and an integer `x`. It should return another function which takes in one argument, another integer. This function returns the result of applying `f` to `x` this number of times.

Make sure to use recursion in your solution.

```
def make_func_repeater(f, x):
    """
    >>> incr_1 = make_func_repeater(lambda x: x + 1, 1)
    >>> incr_1(2) #same as f(f(x))
    3
    >>> incr_1(5)
    6
    """
```

```
def repeat(_____):

    if _____:

        return _____

    else:

        return _____

    return _____
```

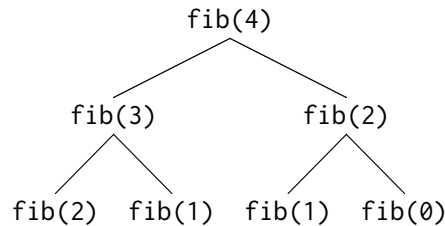
```
def repeat(i):
    if i == 0:
        return x
    else:
        return f(repeat(i - 1))
return repeat
```

2 Tree Recursion

Consider a function that requires more than one recursive call. A simple example is the recursive `fibonacci` function:

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)
```

This type of recursion is called **tree recursion**, because it makes more than one recursive call in its recursive case. If we draw out the recursive calls, we see the recursive calls in the shape of an upside-down tree:



We could, in theory, use loops to write the same procedure. However, problems that are naturally solved using tree recursive procedures are generally difficult to write iteratively. It is sometimes the case that a tree recursive problem also involves iteration: for example, you might use a while loop to add together multiple recursive calls.

As a general rule of thumb, whenever you need to try multiple possibilities at the same time, you should consider using tree recursion.

How to diagram Tree Recursion

Questions

- 2.1 I want to go up a flight of stairs that has n steps. I can either take 1 or 2 steps each time. How many different ways can I go up this flight of stairs? Write a function `count_stair_ways` that solves this problem for me. Assume n is positive.

Before we start, what's the base case for this question? What is the simplest input?

When there is only 1 step, there is only one way to go up the stair. When there are two steps, we can go up in two ways: take a two-step, or take 2 one-steps.

What do `count_stair_ways(n - 1)` and `count_stair_ways(n - 2)` represent?

`count_stair_ways(n - 1)` represents the number of different ways to go up the first $n - 1$ stairs. `count_stair_ways(n - 2)` represents the number of different ways to go up the first $n - 2$ stairs. Our base cases will take care of the remaining 1 or 2 steps.

Use those two recursive calls to write the recursive case:

```
def count_stair_ways(n):

    if n == 1:
        return 1
    elif n == 2:
        return 2
    return count_stair_ways(n-1) + count_stair_ways(n-2)
```

[Video walkthrough \(Leap of Faith\)](#)

[Video Walkthrough \(Diagramming Trees\)](#)

- 2.2 Consider a special version of the `count_stairways` problem, where instead of taking 1 or 2 steps, we are able to take **up to and including** k steps at a time.

Write a function `count_k` that figures out the number of paths for this scenario. Assume n and k are positive.

```
def count_k(n, k):
    """
    >>> count_k(3, 3) # 3, 2 + 1, 1 + 2, 1 + 1 + 1
    4
    >>> count_k(4, 4)
    8
    >>> count_k(10, 3)
    274
    >>> count_k(300, 1) # Only one step at a time
    1
    """

    if n == 0:
        return 1
    elif n < 0:
        return 0
    else:
        total = 0
```

```
i = 1
while i <= k:
    total += count_k(n - i, k)
    i += 1
return total
```

[Video Walkthrough](#)

2.3 Here's a part of the Pascal's triangle:

Column:	0	1	2	3	4	...
Row 0:	1					
Row 1:	1	1				
Row 2:	1	2	1			
Row 3:	1	3	3	1		
Row 4:	1	4	6	4	1	
...						

Every number in Pascal's triangle is defined as the sum of the item above it and the item that is directly to the upper left of it, use 0 if the entry is empty. Define the procedure `pascal(row, column)` which takes a row and a column, and finds the value at that position in the triangle.

def `pascal(row, column):`

```

    if column == 0:
        return 1
    elif row == 0:
        return 0
    else:
        return pascal(row - 1, column) + \
            pascal(row - 1, column - 1)

```

Background: Pascal's triangle is a useful recursive definition that tells us the coefficients in the expansion of the polynomial $(x + a)^n$. Each element in the triangle has a coordinate, given by the row it is on and its position in the row (which you could call its column).