# RECURSION **6**

## DATA C88C

October 5, 2022

## 1 Recursion

#### 1.1 Introduction

A **recursive function** is a function that is defined in terms of itself. A good example is the factorial function. Consider this example:

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

Although we haven't finished defining factorial, we are still able to call it in the second return statement since the function body is not evaluated until the function is called.

Note that when n is 0 or 1, we just return 1. This is known as the **base case**, and it prevents the function from infinitely recursing. Otherwise, we would continue calling factorial (n - 1) forever on smaller and smaller values of n. This would not be the desired behavior since factorial is not defined for negative numbers.

Now we can compute factorial (2) in terms of factorial (1), and factorial (3) in terms of factorial (2), and factorial (4) – well, you get the idea.

#### 1.2 Writing Recursive Functions

There are three common steps in a recursive definition:

- 1. **Figure out your base case**: The base case is usually the simplest input possible to the function. For example, factorial(0) is 1 by definition. You can also think of a base case as a stopping condition for the recursion. If you can't figure this out right away, move on to the recursive case and try to figure out the point at which we can't reduce the problem any further.
- 2. Make a recursive call with a simpler argument: Simplify your problem, and assume that a recursive call for this new problem will simply work. We call this the "recursive leap of faith". For factorial, we reduce the problem by calling factorial(n-1).
- 3. Use your recursive call to solve the full problem: Remember that we are assuming the recursive call works. With the result of the recursive call, how can you solve the original problem you were asked? For factorial, we just multiply (n 1)! by n.

#### 1.3 Correctness

One way to go understand recursion is to separate out two things: *internal correctness* and not running forever (known as *halting*).

A recursive function is *internally correct* if it is always does the right thing assuming that every recursive call does the right thing. For example, the same factorial function from above but with no base case is internally correct, but does not halt.

A recursive function is correct if and only if it is both internally correct and halts; but you can check each property separately. The "recursive leap of faith" is temporarily placing yourself in a mindset where you only check internal correctness.

# 1.4 Questions

1. Write a recursive function that takes in an integer n and prints out a countdown from n to 1.

**Problem 1a** First, think about a base case for the countdown function. What is the simplest input the problem could be given?

**Problem 1b** After you've thought of a base case, think about a recursive call with a smaller argument that approches the base case. What happens if you call countdown (n - 1)?

**Problem 1c** Then, put the base case and the recursive call together, and think about where a print statement would be needed.

```
def countdown(n):
"""
>>> countdown(3)
3
2
1
"""
```

**Problem 1d** How can we change countdown to count up instead without modifying a lot of the code?

Page 3

2. Write a function expt (base, power), which implements the exponent function. For example, expt (3, 2) returns 9, and expt (2, 3) returns 8. Assume power is always a non-negative integer. Use recursion, not pow! def expt (base, power):

3. Write a recursive function that takes a number n and returns the sum of every other digit, starting from the rightmost digit. Assume n is non-negative.

You might find the operators // and % useful.

```
def sum_every_other_digit(n):
"""
>>> sum_every_other_digit(7)
7
>>> sum_every_other_digit(30)
0
>>> sum_every_other_digit(228)
10
>>> sum_every_other_digit(123456)
12
>>> sum_every_other_digit(1234567) # 1 + 3 + 5 + 7
16
"""
```

**DISCUSSION 6: RECURSION** 

4. Remember map? Given a list of elements seq and a function fn, we want to return a list with the function applied to each element. Let's write it recursively! def map(fn, seq):

Extra Challenge: Try to write a mutation version! (i.e. Don't return a new list. Instead, apply map by altering the original list.)

DISCUSSION 6: RECURSION

5. Below is the iterative version of is\_prime, which returns True if positive integer n is a prime number and False otherwise:

```
def is_prime (n):
if n == 1:
    return False
k = 2
while k < n:
    if n % k == 0:
        return False
    k += 1
return True</pre>
```

Implement the recursive is\_prime function. Do not use a while loop, use recursion. 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 ______:
    elif ______:
    return ______
```