Recursion

Week 13

Recursive definition

- In mathematics, a **recursive definition** of a function defines values of the function in terms of the values of the same function for inputs of smaller size.
- "Going back" needs to also stop at some points, so there needs to be some initial values.
- Factorial $n! = 1 \times 2 \times \cdots \times (n-1) \times n$ of a number n can be defined recursively:

$$n! = n \times (n-1)! \quad \text{for } n \geq 1.$$

- Initial value $0! = 1.$

How to compute 5!

$$\overset{120}{5!} = 5 \times \overset{24}{4!}$$

$$4! = 4 \times \overset{6}{3!}$$

$$3! = 3 \times \overset{2}{2!}$$

$$2! = 2 \times \overset{1}{1!}$$

$$1! = 1 \times \overset{1}{0!}$$

$$0! = 1$$

Initial value

Recursive algorithm

- Because we can call functions, it is possible that a function can call also **itself**.
- An algorithm which **calls itself** with *smaller inputs* is called a **recursive algorithm.**
- Each instance of a call contains own values of variables (including parameters).
- Each instance **returns** the result for the *current input* after doing some operations on the returned value of smaller inputs.
- The recursion needs to **stop** somewhere, we cannot go smaller and smaller cases infinitely!

Recursive algorithm

- It resembles recursive definitions of mathematical functions.
- Recursive functions are often relatively short.
- If you can write a recursive mathematical definition of a *concept*, you can also write a recursive algorithm for computing values of that concept.
- Iteration (looping) in *functional languages* (like Scala) is usually accomplished via recursion.

Recursive algorithm

If we are going to solve something recursively, we need to define two things:

1. What is the **base case**?

- For what inputs can we automatically just spit out the answer without having to do any work?

2. What is the **recursive case**?

- How would the answer to a **"smaller" problem of the same kind** help get the answer to the original problem?

Computing the factorial recursively

```
def factorial(n):
    if n == 0:                    # base case
        return 1
    else:                         # recursive case
        return n * factorial(n-1)
```

How it works

- **Call**: `factorial(5)` is called
  Recursive case: Since 5 is not 0, **return** `5 * factorial(4).`
- **Call:** `factorial(4)` is called.
  Recursive Case: Since 4 is not 0, **return** `4 * factorial(3)`
- **Call:** `factorial(3)` is called.
  Recursive Case: Since 3 is not 0, **return** `3 * factorial(2)`
- **Call:** `factorial(2)` is called.
  Recursive Case: Since 2 is not 0, **return** `2 * factorial(1)`
- **Call:** `factorial(1)` is called.
  Recursive Case: Since 1 is not 0, **return** `1 * factorial(0)`
- **Call:** `factorial(0)` is called.
  Base Case: Since 0 equals 0, **return** `1`

Now we start "unwind" calls which are waiting for their values.

How it works

- `factorial(1)` returns `1 * factorial(0) = 1 * 1 = 1`
- `factorial(2)` returns `2 * factorial(1) = 2 * 1 = 2`
- `factorial(3)` returns `3 * factorial(2) = 3 * 2 = 6`
- `factorial(4)` returns `4 * factorial(3) = 4 * 6 = 24`
- `factorial(5)` returns `5 * factorial(4) = 5 * 24`

`= 120`

Reversing a string

```
def reverse_string(s):
    if len(s) <= 1:                      # base case
        return s
    else:                                # recursive case
        return reverse_string(s[1:]) + s[0]
```

Greatest common divisor

```
def gcd(a, b):
    if b == 0:                          # base case
        return a
    else:                               # recursive case
        return gcd(b, a % b)
```

- GCD of two numbers `a` and `b` remains the same if we replace the larger number `a` with its remainder when divided by the smaller number `b`.
- This allows us to reduce the problem to smaller and smaller numbers until we reach the **base** case where one of the numbers becomes zero.