# Computer Science 1 — CSci 1100
# Lab 12 — Recursion
# Fall Semester 2019

## Lab Overview

This is the last lab of the semester! It only has two checkpoints. You will explore the basic mechanisms of recursion and the design of simple recursive functions.

As discussed during class, recursion is most often used as a formal way of modeling algorithms and data structures and less often used as a practical programming technique. There are some problems, however, that are almost impossible to solve using a non-recursive algorithm. Many of the search algorithms for tree-like data structures — which you will see if you take Data Structures — fit into this category. Unfortunately, we will not see any algorithms here that fit this category. Still, building an understanding of recursion is an important step in your development as a programmer and as a computer scientist.

## Checkpoint 1

Attempts to define the formal foundations of mathematics at the start of the 20th century depended heavily on the mathematical notion of recursion. While ultimately completing this formalization was shown to be impossible, the theory that was developed contributed heavily to the formal basis of computer science. The idea is to start with primitive, axiomatic definitions and build from there.

We will build basic arithmetic operations starting with just one value — 0 — and three operations:

1. adding 1 to a value (the *successor* operation),
2. subtracting 1 from a value (the *predecessor* operation), and
3. testing a value to see if it is 0.

In this case, we can think of all values as successor of 0: the value 1 is the successor of 0, the value 2 is the successor of 1, etc.

Using this logic, we can represent addition, `m+n`, as adding 1 to `m` as many times as we can subtract 1 from `n` (until we reach 0). We can write this recursively as follows:

```
def add(m,n):
    if n == 0:
        return m
    else:
        return add(m,n-1) + 1
```

To make sure you understand this, show the sequence of calls made by

```
print(add(5,3))
```

You may add print statements to show the results.

Building on this idea, now write a recursive function to multiply two non-negative integers using only the `add` function we've just defined, together with +1, -1 and the equality with 0 test. Call this function `mult`. Demonstrate the result by multiplying 8 and 3. As a recursive function `mult(8,3)`, `mult` can call itself of course.

Now, define the integer power function, power$(x, n) = x^n$, in terms of the `mult` function you just wrote, together with +1, -1, and equality. Demonstrate the result by computing `power(6,3)`. Now try `power(6,8)`. Can you explain the error?

**To complete Checkpoint 1:** Show:

1. the calls from `add(5,3)`,
2. a working version of `mult`, and
3. a working version of `power`.

## Checkpoint 2

**Please come to lab for the last checkpoint.**