# Recursion

This lecture will focus on

- recursion
    - base cases
    - simple recursion

## Your future in CS

I used to include this on my slides, but since these slides have changed - going to just leave it up here for every notebook. I get a lot of questions about more programming courses, the concentrations, and minors in computer science. Here is a brief reminder.

CS 164 – Next Course In Sequence, also consider CS 220 (math and stats especially)

- CO Jobs Report 2021 – 77% of *all* new jobs in Colorado require programming
- 60% of all STEM jobs requires *advanced* (200-300 level)
- 31% of all Bachelor of Arts degree titled jobs also required coding skills
- 2016 Report found on average jobs that require coding skills paid $22,000 more

- Concentrations in CS:
    - Computer science has a number of concentrations.

        - General concentration is the most flexible, and even allows students to double major or minor pretty easily.
        - Software Engineering
        - Computing Systems
        - Human Centered Computing
        - Networks and Security
        - Artificial Intelligence
        - Computer Science Education.
    - Minors:
        - Minor in Computer Science - choose your own adventure minor
        - Minor in Machine Learning - popular with stats/math, and engineering
        - Minor in Bioinformatics - Biology + Computer Science

## Warmup Activity - Loops

- Write a function that takes in a parameter that will be a whole number
- Calculate the **factorial** of that number
    - Reminder: factorial for:
      $$f(5) = 5 * 4 * 3 * 2 * 1 = 120$$
      $$f(4) = 4 * 3 * 2 * 1 = 24$$
- Use a **loop** to calculate the factorial.

In [ ]:
```
def loop_factorial(whole):
    answer = 1
    ## your loop here, you may want to use a while loop, but you can use for with the
    while(whole > 1):
        answer *= whole
        whole -= 1
    #for i in range(whole, 1, -1):  ## have this here for the range version
    #    answer *= i
    return answer

print("Testing loop factorial", loop_factorial(5))
print("Testling loop factorial", loop_factorial(4))
```

```
Testing loop factorial 120
Testling loop factorial 24
```

# Recursion

- Fancy Word...
    - Often people over complicate it!
- Means creating a loop, by calling a function over and over!
    - Yes, Recursion is another way to loop!
    - Essentially, **Divide**-**Conquer**-**Glue** at the algorithm level
- Advantages of Recursion
    - Complex iteration can be broken into smaller functions
    - Creation of sequences or values across sequences can be done with recursion

- Try to think in **two** steps only
    - Base case (exit condition)
    - Repetition / Recursive condition

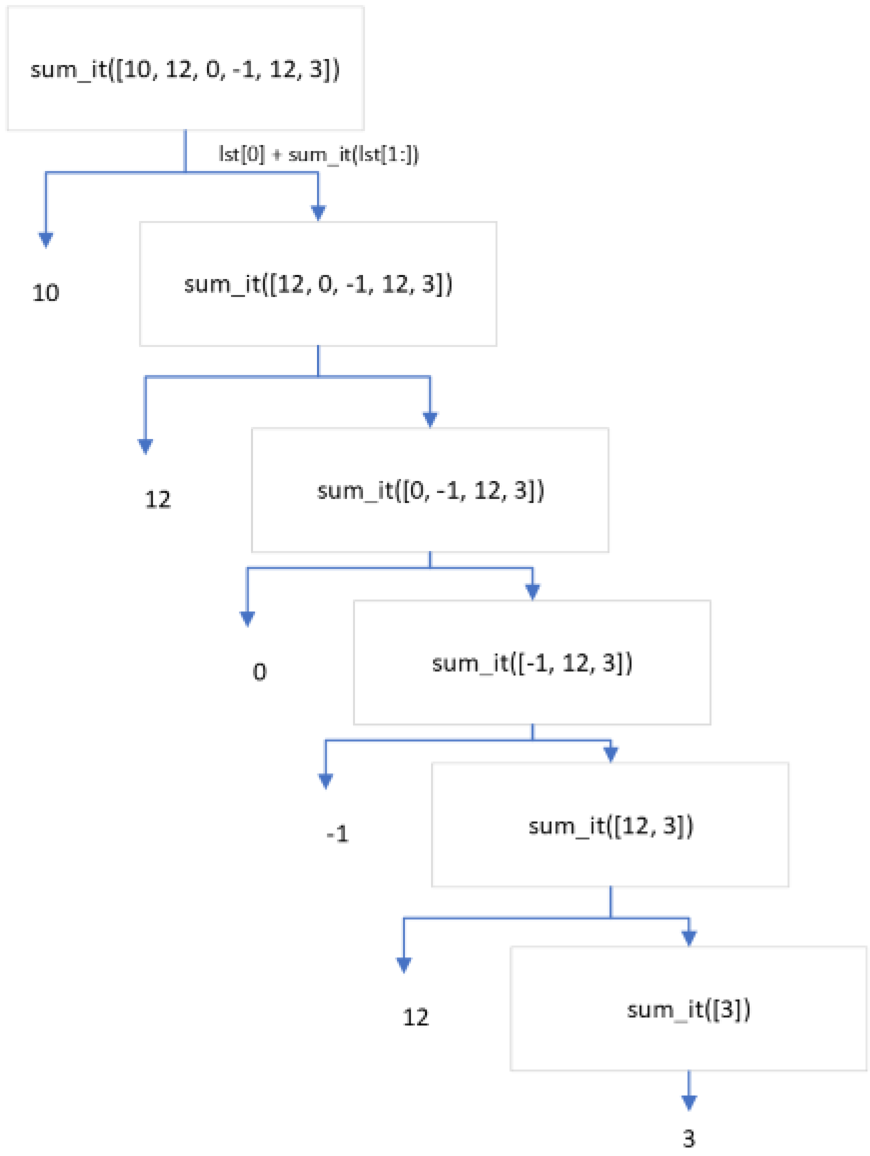
In [ ]:

```
def sum_it(lst):
    # base case
    if len(lst) < 1: return 0
    # recursive condition
    return lst[0] + sum_it(lst[1:])  ### NOTICE: calling this same function!

my_list = [10, 12, 0, -1, 12, 3]
print(sum_it(my_list))
```

```
36
```

## What is happening?

- Break it down into steps
    - In the base case
        - if the list less than 1 (empty) - return 0
        - Could also be said `if len(lst) <= 1: return lst[0]`
    - In the recursive condition
        - take the first spot
        - add the other values to it!



## In Class Activity: Return to Factorial

- Go ahead and write the factorial
  - but now use recursion
- what is your **base** case?
  - often similar to when you want to *stop* looping
- What is your recursive condition?
- Often it is easier to break it down into simpler cases and work it out on paper
  - factorial(1)
  - factorial(2)
  - factorial(3)
  - Then when you have those few situations figured out, it can expand

In [ ]:

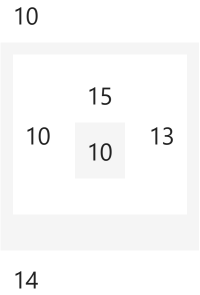
```
def factorial(whole):
    ## base case
    if whole < 2: return 1
    ## recursive call
    return whole * factorial(whole - 1)


print("Testing Recursive Factorial", factorial(1))
print("Testing Recursive Factorial", factorial(2))
print("Testing Recursive Factorial", factorial(3))
print("Testing Recursive Factorial", factorial(4))
print("Testing Recursive Factorial", factorial(5))
print("Testing Recursive Factorial", factorial(25))
```

```
Testing Recursive Factorial 1
Testing Recursive Factorial 2
Testing Recursive Factorial 6
Testing Recursive Factorial 24
Testing Recursive Factorial 120
Testing Recursive Factorial 15511210043330985984000000
```

## Why Recursion?

- These examples can use loops!
- Correct, but as we look at more 'complex' ways to store data, loops fall apart

Assume the following list structure



or in python

```
lst = [10, [10, [15, 10], 13], 14]
```

- If this was fixed, you would need at least 3 nested loops, but then every time a list appears, you need another loop! When does it stop?
- Tree structures (such as those used in Router Tables) can look like this (sort of)!

Recursion to the rescue!

In [ ]:

```
def sum_it(lst):
    if len(lst) < 1:return 0
    if type(lst[0]) is list:
        return sum_it(lst[0]) + sum_it(lst[1:])
    return lst[0] + sum_it(lst[1:])


lst = [10, [10, [15, 10], 13], 14]
print(sum_it(lst))
```

```
72
```

## Tail Recursion

- Tail Recursion
- Another way to look at recursion
- Use a list to build the 'result' as the method is called
- Helps with memory, and commonly used to build lists!

**Code Along** Goes over this example

In [ ]:

```python
def reverse_list_tail(values, reverse):
    # base first!
    if len(values) < 1: return reverse # this contains the final list
    return  reverse_list_tail(values[:-1], reverse + [values[-1]])

def reverse_list_tail_start(values):
    return  reverse_list_tail(values, []) # will learn a way to make default values ir


names = ["Princess Zelda", "Ganon", "Link", "Epona", "Impa"]
reversed = reverse_list_tail_start(names)

print(reversed)
```

```
['Impa', 'Epona', 'Link', 'Ganon', 'Princess Zelda']
```

# Recursion Overview

- Keep it simple!
    - Often folks over think it
- Always find the base case first
- Then write for the N+1 case
    - Wait! That is induction!
    - Correct, math majors recursion and induction two sides of the same coin.
    - **CS 220** goes into more in depth
- Used heavily in data science
    - And Artificial Intelligence
- But not always good to use!
    - There is a cost
    - Most notable, speed / memory concerns (to be explored later)