Recursion == Happiness
### xkcd, Fixing Problems

What are you working on?

Trying to fix the problems I created when I tried to fix the problems I created when I tried to fix the problems I created when...

### xkcd, Dependencies

<table>
<thead>
<tr>
<th>DEPARTMENT</th>
<th>COURSE</th>
<th>DESCRIPTION</th>
<th>PREREQS</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMPUTER SCIENCE</td>
<td>CPSC 432</td>
<td>INTERMEDIATE COMPILER DESIGN, WITH A FOCUS ON DEPENDENCY RESOLUTION.</td>
<td>CPSC 432</td>
</tr>
</tbody>
</table>
Uh dad, I think you forgot the base case!
Bourton-on-the-Water, England
Level 1
Level 2
Level 4
Homeworks

• Go over grader comments
  – Don’t repeat the same mistakes

• Comment your code
  – Better to comment too much than too little
  – Easier for graders will tell you that comments are not needed for a specific line or block of code
Learning Goals

- Review recursion (and prove that recursion works)
- Explain the use-it-or-lose-it strategy for recursion
Math Induction = CS Recursion

Math

inductive definition

0! = 1

n! = n \times (n-1)!

Python (Functional)

recursive function

```python
# recursive factorial
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```
"Tracing" Recursive Factorial

factorial(3)
↓↑6
return 3 * factorial(2)
↓↑2
return 2 * factorial(1)
↓↑1
return 1 * factorial(0)
↓↑1
return 1

# recursive factorial
def factorial(n):
    if n == 0:
        return 1
    else:
        return n*factorial(n-1)
Proving (by induction) that the recursive factorial function works

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

Claim: The recursive factorial function correctly computes $n!$ for any integer $n \geq 0$.

Proof: By induction on $n$

Basis: For the case $n=0$, we have that $n! = 1$ by definition and `factorial(1)` returns 1.

Induction: For the general case $n > 0$ ...

Induction hypothesis: We assume that `factorial(n-1)` correctly returns $(n-1)!$.

Induction step: We wish to show that `factorial(n)` correctly computes $n!$.
• By the way that we have written our function, `factorial(n)` returns $n \times \text{factorial}(n-1)$.
• But, by the induction hypothesis, `factorial(n-1)` returns $(n-1)!$.
• Therefore, `factorial(n)` returns $n \times \text{factorial}(n-1) = n(n-1)! = n!$. 
>>> sum([10, 2, 30])
42

def sum(my_list):
    """Return sum of the numbers in my_list""
    if my_list == []: return 0
    else: return my_list[0] + sum(my_list[1:])

There are no extra variables here, as in ... counter = 0
Variables Verboten?

```python
def sum(my_list):
    """Return sum of the numbers in my_list""
    if my_list == []: return 0
    else:
        sum_of_rest = sum(my_list[1:])
        new_sum = my_list[0] + sum_of_rest
    return new_sum
```

This kind of use of a temporary variable is fine!
def sum(my_list):
    """Return sum of the numbers in my_list"""
    counter = 0
    if my_list == []: return 0
    else:
        counter = counter + my_list[0]
        sum(my_list[1:]):
    return counter

But this is NOT going to work!
The Moral of the Story...

def sum(my_list):
    """Return sum of the numbers in my_list""
    if my_list == []: return 0
    else:
        sum_of_rest = sum(my_list[1:])
        new_sum = my_list[0] + sum_of_rest
    return new_sum

Variables that you are just using to temporarily store intermediate results are fine to use!
def mystery(my_list):
    """bad docstring :(""
    if my_list == []: return 0
    elif len(my_list) == 1: return my_list[0]
    else:
        split = len(my_list) // 2
        return mystery(my_list[:split]) + mystery(my_list[split:])
Learning Goals

• Review recursion (and prove that recursion works)

• Explain the use-it-or-lose-it strategy for recursion
Prof. Lai’s Spam Find!

I claim that this can of Spam weighs 12 kilos!
The Scale Problem

Gold Spam nugget is claimed to weight 12 kilos.
Weights: [2, 3, 4, 7, 10, 42]

>> subset(12, [2, 3, 4, 7, 10, 42])
True

>>> subset(8, [2, 3, 4, 7, 10, 42])
False

>>> subset(15, [2, 3, 4, 7, 10, 42])
???
The Protein Problem

```python
>>> subset(4, [1, 2, 3, 5])
True or False?
```

We will not answer that question, but we will answer subset questions with smaller targets and/or shorter lists!
Writing `subset`

```python
def subset(target, L):
    if target == 0: return ???
```

Fill this in (in your notes)!
Writing `subset`

```python
def subset(target, L):
    if target == 0: return True
    elif L == ???: return ???
```

Two inputs usually means two base cases!

Fill this in (in your notes)!
Writing `subset`

```python
>>> subset(4, [1, 2, 3, 5])
True

def subset(target, L):
    if target == 0: return True
    elif L == []: return False
    elif L[0] > target: return ???
    Fill this in (in your notes)!
```

Two inputs usually means two base cases!
>>> subset(4, [1, 2, 3, 5])
True

def subset(target, L):
    if target == 0: return True
    elif L == []: return False
    elif L[0] > target: return subset(target, L[1:])
    else:
        it = L[0]
        use_it =
        lose_it =
        return ________________________________

Fill this in (in your notes)!
>>> subset(4, [1, 2, 3, 5])
True

def subset(target, L):
    if target == 0: return True
    elif L == []: return False
    elif L[0] > target: return subset(target, L[1:])
    else:
        it = L[0]
        use_it = subset(target - it, L[1:])
        lose_it =
        return _____________________________

Fill this in (in your notes)!
>>> subset(4, [1, 2, 3, 5])
True

def subset(target, L):
    if target == 0: return True
    elif L == []: return False
    elif L[0] > target: return subset(target, L[1:])
    else:
        it = L[0]
        use_it = subset(target - it, L[1:])
        lose_it = subset(target, L[1:])
        return ________________________

Fill this in (in your notes)!
```python
def subset(target, L):
    if target == 0: return True
    elif L == []: return False
    elif L[0] > target: return subset(target, L[1:])
    else:
        it = L[0]
        use_it = subset(target - it, L[1:])
        lose_it = subset(target, L[1:])
        return use_it or lose_it
```
def subset(target, L):
    if target == 0: return True
    elif L == []: return False
    elif L[0] > target: return subset(target, L[1:])
    else:
        it = L[0]
        use_it = subset(target - it, L[1:])
        lose_it = subset(target, L[1:])
        return use_it or lose_it
def subset(target, L):
    if target == 0: return True
    elif L == []: return False
    elif L[0] > target: return subset(target, L[1:])
    else:
        it = L[0]
        use_it = subset(target - it, L[1:])
        lose_it = subset(target, L[1:])
        return use_it or lose_it

def subset(target, L):
    if target == 0: return True
    elif L == []: return False
    elif L[0] > target: return subset(target, L[1:])
    else:
        it = L[0]
        use_it = subset(target - it, L[1:])
        lose_it = subset(target, L[1:])
        return use_it or lose_it
Change

In this problem, we are allowed to use a coin denomination as many times as we want!

But in Shmorbodia…

“greedy” approach does not work!
Change

>>> change(42, [25, 21, 1])
2

def change(amount, coins):

Fill this in (in your notes)!
>>> change(42, [25, 21, 1])
2

def change(amount, coins):
    if amount == 0: return ???
Change

```python
def change(amount, coins):
    if amount == 0: return 0
    elif coins == []: return ???
```
Change

```python
>>> change(42, [25, 21, 1])
2

def change(amount, coins):
    if amount == 0: return 0
    elif coins == []: return float('inf')
    else:
        it = coins[0]
```

We are making change!

Fill this in (in your notes)!
```python
def change(amount, coins):
    if amount == 0: return 0
    elif coins == []: return float('inf')
    else:
        it = coins[0]
        if it > amount:
            return ____________________
        else:
            use_it = ____________________
            lose_it = ____________________
            return ____________________
```
>>> change(42, [25, 21, 1])
2

def change(amount, coins):
    if amount == 0: return 0
    elif coins == []: return float('inf')
    else:
        it = coins[0]
        if it > amount:

            return change(amount, coins[1:])
    else:
        use_it = 1 + change(amount-it, coins)

    lose_it = change(amount, coins[1:])

    return min(use_it, lose_it)
Change

```python
>>> change(42, [25, 21, 1])
2

def change(amount, coins):
    if amount == 0: return 0
    elif coins == []: return float('inf')
    else:
        it = coins[0]
        if it > amount:
            return change(amount, coins[1:])
        else:
            use_it = 1 + change(amount-it, coins)
            lose_it = change(amount, coins[1:])
            return min(use_it, lose_it)

Fill this in (in your notes)!
```
```python
def change(amount, coins):
    if amount == 0: return 0
    elif coins == []: return float('inf')
    else:
        it = coins[0]
        if it > amount:
            return change(amount, coins[1:])
        else:
            use_it = 1 + change(amount-it, coins)
            lose_it = change(amount, coins[1:])
            return min(use_it, lose_it)
```

```
def change(amount, coins):
    if amount == 0: return 0
    elif coins == []: return float('inf')
    else:
        it = coins[0]
        if it > amount:
            return change(amount, coins[1:])
        else:
            use_it = 1 + change(amount-it, coins)
            lose_it = change(amount, coins[1:])
            return min(use_it, lose_it)
```
Comparing DNA with Longest Common Subsequence (LCS)

AGGACAT
ATTACGAT

>>> LCS("AGGACAT", "ATTACGAT")
5

EGS = "GTACGTCGATAACTG"
WGS = "TGATCGTCATAACGT"

Sclitz gene

Groovy Groody, profdude!

LCS("spam", "pims")
Try writing LCS

```python
>>> LCS("AGGACAT", "ATTACGAT")
5

def LCS(string1, string2):
    if string1 == "" or string2 == "": return 0
    elif string1[0] == string2[0]:
        return 1 + LCS(string1[1:], string2[1:])
    else:
        return LCS(string1[1:], string2[1:])

LCS("spam", "pam") -> LCS("pam", "am") -> LCS("am", "m") -> LCS("a", "") -> 0
```

That's not what I want =(
Try writing LCS

```python
>>> LCS("AGGACAT", "ATTACGAT")
5

def LCS(string1, string2):
    if string1 == "" or string2 == "": return 0
    elif string1[0] == string2[0]:
        return 1 + LCS(string1[1:], string2[1:])
    else:
        option1 =
        option2 =
        return max(option1, option2)
```

Try this in your notes!
def LCS(string1, string2):
    if string1 == "" or string2 == "": return 0
    elif string1[0] == string2[0]:
        return 1 + LCS(string1[1:], string2[1:])
    else:
        option1 = LCS(string1, string2[1:])
        option2 = LCS(string1[1:], string2)
        return max(option1, option2)
Edit Distance (a sneak preview of things to come!)

>>> ED("ATTATCG", "ACATTC")
4
ATTAT-CG
A-CATTC-

>>> ED("spam", "scramble")
5
sp_am___
s_scramble

spam ->
scam ->
scram ->
scramb -> scrambl -> scramble
Turtle Meets Recursion!

```python
import turtle

def mystery(leg_length, num_legs):
    if num_legs == 0: return
    else:
        turtle.forward(leg_length)
        turtle.right(90)
        mystery(leg_length + 10, num_legs - 1)
    return
```

Demo
Recursion and the Fractal Geometry of Nature
Need a hand with recursion?
The Koch Snowflake Fractal:

Turtle + Fractals = :^)

The Koch Snowflake Fractal:

level 0  level 1  level 2  level 3  level 4  level 5

level 0

level 1

1/3 1/3 1/3

1/3 1/3

Hey, look, what I can do!
import turtle

def koch_side(length, level):
    if level == 0:
        turtle.forward(length)
    else:
        koch_side(length/3, level-1)
turtle.left(60)
koch_side(length/3, level-1)
turtle.right(120)
koch_side(length/3, level-1)
turtle.left(60)
koch_side(length/3, level-1)

def koch_flake(length, level):
    for i in range(3):
        koch_side(length, level)
turtle.right(120)
The Sierpinski Triangle!

Snazzy!