Module 17

Recursion
Motivation for Video

• This series is **not** about a control structure

• **Recursion:** a *programming technique*
  - Uses techniques you know in an usual way
  - Duplicates the iteration of for and while
  - Exists because it is often more efficient

• It is a very **advanced** topic
  - You will study this all four years of a CS program
  - We are not expecting you to master this
  - We just want you to understand the foundations
Recursive Definition

• A definition defined in terms of itself

• **Example:** PIP
  - Tool for installing Python packages
  - **PIP** stands for “**PIP** Installs Packages”

• Sounds like a circular definition
  - The example above is
  - But need not be in right circumstances
Example: Factorial

- Non-recursive definition (n an int $\geq 0$):
  $n! = n \times (n-1) \times \ldots \times 2 \times 1$
  $0! = 1$

- Refactor top formula as:
  $n! = n \times (n-1) \times \ldots \times 2 \times 1$

- Recursive definition:
  $n! = n \times (n-1)!$ for $n > 0$
  $0! = 1$
Example: Fibonacci

- Sequence of numbers: 1, 1, 2, 3, 5, 8, 13, ...
  \[ a_0 \ a_1 \ a_2 \ a_3 \ a_4 \ a_5 \ a_6 \]
  - Refer to element at position \( n \) as \( a_n \)
  - Get the next element by adding previous two

- Recursive definition:
  - \( a_n = a_{n-1} + a_{n-2} \) \hspace{1cm} \text{Recursive Case}
  - \( a_0 = 1 \) \hspace{1cm} \text{Base Case}
  - \( a_1 = 1 \) \hspace{1cm} \text{(another) Base Case}
Example: Fibonacci

- Sequence of numbers: 1, 1, 2, 3, 5, 8, 13, ...

- Refer to element at position \( n \) as \( a_n \)

- Get the next element by adding previous two

- Recursive definition:
  \[
  a_n = a_{n-1} + a_{n-2}
  \]

- Base Case
  - \( a_0 = 1 \)
  - \( a_1 = 1 \)

- Recursive Case

While recursion may be *weird* it is well-defined and not circular

(another) Base Case
Recursive Functions

• A function that calls itself
  ▪ Inside of body there is a call to itself
  ▪ Very natural for recursive math defs

• Recall: Factorial
  ▪ $n! = n \times (n-1)!$  \textbf{Recursive Case}
  ▪ $0! = 1$  \textbf{Base Case}
def factorial(n):
    """Returns: factorial of n.
    Pre: n ≥ 0 an int"
    if n == 0:
        return 1
    return n*factorial(n-1)
def factorial(n):
    """Returns: factorial of n.
    Pre: n ≥ 0 an int"""
    if n == 0:
        return 1
    return n*factorial(n-1)

y = factorial(4)
def fibonacci(n):
    """Returns: Fibonacci $a_n$
    Precondition: n ≥ 0 an int"""
    if n <= 1:
        return 1
    return (fibonacci(n-1) + fibonacci(n-2))
Fibonacci: # of Frames vs. # of Calls

- Fibonacci is very inefficient.
  - $\text{fib}(n)$ has a stack that is always $\leq n$
  - But $\text{fib}(n)$ makes a lot of redundant calls

Path to end = the call stack
Fibonacci: # of Frames vs. # of Calls

- Fibonacci is very inefficient.
  - \( \text{fib}(n) \) has a stack that is always \( \leq n \)
  - But \( \text{fib}(n) \) makes a lot of redundant calls

Recursion is not the best way, but it is the easiest way
Recursion vs Iteration

- **Recursion** is *provably equivalent* to **iteration**
  - Iteration includes **for-loop** and **while-loop** (later)
  - Anything can do in one, can do in the other
- But some things are easier with recursion
  - And some things are easier with iteration
- Will **not** teach you when to choose recursion
  - This is a topic for more advanced courses
- But we will cover one popular use case
Recursion is best for Divide and Conquer

**Goal:** Solve problem P on a piece of data

- **data**
  - string or tuple (something slicable)
Recursion is best for Divide and Conquer

**Goal**: Solve problem P on a piece of data

**Idea**: Split data into two parts and solve problem

Combine Answer!
Divide and Conquer Example

Count the number of 'e's in a string:

```
penne
```

Two 'e's

```
pe
```

One 'e'

```
nne
```

One 'e'
Divide and Conquer Example

Count the number of 'e's in a string:

```
penne
```

Often more than one way to break up:

- Zero 'e's
- Two 'e's
Divide and Conquer Example

Remove all spaces from a string:

```
a  b  c
```

```
a  b  c
```

```
a  +  b  c
```

```
a  b  c
```
Divide and Conquer Example

Remove all spaces from a string:

Will see how to implement next
Three Steps for Divide and Conquer

1. Decide what to do on “small” data
   - Some data cannot be broken up
   - Have to compute this answer directly

2. Decide how to break up your data
   - Both “halves” should be smaller than whole
   - Often no wrong way to do this (next lecture)

3. Decide how to combine your answers
   - Assume the smaller answers are correct
   - Combining them should give bigger answer
def num_es(s):
    """Returns: # of 'e's in s""
    # 1. Handle small data
    if s == '':
        return 0
    elif len(s) == 1:
        return 1 if s[0] == 'e' else 0
    # 2. Break into two parts
    left = num_es(s[0])
    right = num_es(s[1:])
    # 3. Combine the result
    return left+right

"""Short-cut"" for
if s[0] == 'e':
    return 1
else:
    return 0
def num_es(s):
    """Returns: # of 'e's in s""
    # 1. Handle small data
    if s == ":
        return 0
    elif len(s) == 1:
        return 1 if s[0] == 'e' else 0
    # 2. Break into two parts
    left = num_es(s[0])
    right = num_es(s[1:])
    # 3. Combine the result
    return left+right
def deblank(s):
    """Returns: s but with its blanks removed""

1. Decide what to do on “small” data

   - If it is the empty string, nothing to do
     if s == "":
         return s

   - If it is a single character, delete it if a blank
     if s == ' ':
         # There is a space here
         return "" # Empty string
     else:
         return s
def deblank(s):
    """Returns: s but with its blanks removed""

2. Decide how to break it up

    left = deblank(s[0])    # A string with no blanks
    right = deblank(s[1:])  # A string with no blanks

3. Decide how to combine the answer

    return left + right     # String concatenation
Putting it All Together

```python
def deblank(s):
    """Returns: s w/o blanks"""
    if s == '':
        return s
    elif len(s) == 1:
        return '' if s[0] == ' ' else s
    left = deblank(s[0])
    right = deblank(s[1:])
    return left + right
```

Handle small data

Break up the data

Combine answers
def deblank(s):
    """Returns: s w/o blanks"""
    if s == '':
        return s
    elif len(s) == 1:
        return '' if s[0] == ' ' else s
    left = deblank(s[0])
    right = deblank(s[1:])
    return left+right
Following the Recursion

deblank

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
</table>
Following the Recursion

debblank

\[
\begin{array}{ccc}
\text{a} & \text{b} & \text{c} \\
\end{array}
\]

debblank

\[
\begin{array}{ccc}
\text{a} & \text{b} & \text{c} \\
\end{array}
\]
Following the Recursion

deblank

\[
\begin{array}{ccc}
  & a & b & c \\
\end{array}
\]

deblank

\[
\begin{array}{ccc}
  & a & b & c \\
\end{array}
\]

deblank

\[
\begin{array}{cc}
a & b & c \\
\end{array}
\]
Following the Recursion

deblank


a


b


b


c
deblank


b


c
deblank


b
deblank


c


b
deblank


b
deblank


c


b
deblank


b
deblank


c
Following the Recursion

defblank

\[
\begin{array}{c|c|c|c}
& a & b & c \\
\hline
\text{deblank} & a & b & c \\
\hline
a & b & c \\
\hline
\end{array}
\]

defblank

\[
\begin{array}{c|c|c|c}
& b & c \\
\hline
deblank & b & c \\
\hline
\end{array}
\]

defblank

\[
\begin{array}{c|c|c|c}
& c \\
\hline
deblank & c \\
\hline
\end{array}
\]
Following the Recursion

debblank

\[
\begin{array}{ccc}
  & a & b & c \\
  & a & b & c \\
 a & & b & c \\
 & & b & c \\
 b & & & c \\
\end{array}
\]
Following the Recursion

deblank

\[
\begin{array}{ccc}
\text{a} & \text{b} & \text{c} \\
\text{a} & \text{b} & \text{c} \\
\text{a} & \text{b} & \text{c} \\
\text{a} & \text{b} & \text{c} \\
\end{array}
\]
Following the Recursion

deblank

[Diagram showing a recursive process with boxes labeled a, b, c and a final c]
Following the Recursion

deblank

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>b</th>
<th>c</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>c</th>
</tr>
</thead>
</table>

✗
Following the Recursion

deblank

\[ \begin{array}{ccc} 
\text{a} & \text{b} & \text{c} \\
\end{array} \]

deblank

\[ \begin{array}{ccc} 
\text{a} & \text{b} & \text{c} \\
\end{array} \]

deblank

\[ \begin{array}{ccc} 
\text{b} & \text{c} \\
\end{array} \]

deblank

\[ \begin{array}{ccc} 
\text{b} & \text{c} \\
\end{array} \]

deblank

\[ \begin{array}{ccc} 
\text{c} \\
\end{array} \]
Following the Recursion

deblank

a

b
c

da

b
c

✗

b
c

✗

c

✗
Following the Recursion

debblank

a b c

debblank

a b c

a b c

b c

b c

b c

c

c
Following the Recursion

deblank

\[
\begin{array}{ccc}
\text{a} & \text{b} & \text{c} \\
\text{x} & \text{a} & \text{b} & \text{c} \\
\text{a} & \text{x} & \text{b} & \text{c} \\
\text{b} & \text{x} & \text{b} & \text{c} \\
\text{c} & \text{x} & \text{c} & \text{c} \\
\end{array}
\]
Following the Recursion

debl ank  a  b  c  ➔  a  b  c

✗

debl ank  a  b  c  ➔  a  b  c

✗

debl ank  a  b  c  ➔  b  c

✗

debl ank  a  b  c  ➔  b  c

✗

debl ank  a  b  c  ➔  c

✗

debl ank  a  b  c  ➔  c
An Observation

- Divide & Conquer works in phases
  - Starts by splitting the data
  - Gets smaller and smaller
  - Until it reaches the base case
- Only then does it give an answer
  - Gives answer on the small parts
- Then glues all of them back together
  - Glues as the call frames are erased
Recursion vs For-Loop

- Think about our for-loop functions
  - For-loop extract one element at a time
  - Accumulator gathers the return value
- When we have a recursive function
  - The recursive step breaks into single elements
  - The return value IS the accumulator
  - The final step combines the return values
- Divide-and-conquer same as loop+accumulator
Breaking Up Recursion

• D&C requires that we *divide* the data
  ▪ Often does not matter how divide
  ▪ So far, we just pulled off one element
  ▪ **Example:** 'penne' to 'p' and 'enne'

• Can we always do this?
  ▪ It depends on the *combination step*
  ▪ Want to divide to make combination easy
How to Break Up a Recursive Function?

```python
def commafy(s):
    """Returns: string with commas every 3 digits
    e.g. commafy('5341267') = '5,341,267'
    Precondition: s represents a non-negative int"
```

**Approach 1**

```
5 341267
```
def commafy(s):
    """Returns: string with commas every 3 digits
    e.g. commafy('5341267') = '5,341,267'
    Precondition: s represents a non-negative int""

Approach 1

5  341267  
  ↓
commafy

341,267
def commafy(s):
    """Returns: string with commas every 3 digits
e.g. commafy('5341267') = '5,341,267'
Precondition: s represents a non-negative int"""

Approach 1

```
5 341267
```

```
5 341,267
```
How to Break Up a Recursive Function?

```python
def commafy(s):
    '''
    Returns: string with commas every 3 digits
    e.g. commafy('5341267') = '5,341,267'
    Precondition: s represents a non-negative int'''
```

**Approach 1**

5 341267

↓

```
5 , 341,267
```

Always? When?
def commafy(s):
    """Returns: string with commas every 3 digits
    e.g. commafy('5341267') = '5,341,267'
    Precondition: s represents a non-negative int""

Approach 1

5  341267

Approach 2

5341  267

Always? When?
def commafy(s):
    """Returns: string with commas every 3 digits
e.g. commafy('5341267') = '5,341,267'
    Precondition: s represents a non-negative int"""

Approach 1

Approach 2

Always? When?
How to Break Up a Recursive Function?

```python
def commafy(s):
    """Returns: string with commas every 3 digits
    e.g. commafy('5341267') = '5,341,267'
    Precondition: s represents a non-negative int"""
```

### Approach 1

1. 5
2. 341267
3. commafy
4. 5, , 341,267
5. Always? When?

### Approach 2

1. 5341
2. 267
3. commafy
4. 5,341
5. 267
def commafy(s):
    """Returns: string with commas every 3 digits
e.g. commafy('5341267') = '5,341,267'
Precondition: s represents a non-negative int""

Approach 1

<table>
<thead>
<tr>
<th>5</th>
<th>341267</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>,</td>
</tr>
</tbody>
</table>

Always? When?

Approach 2

<table>
<thead>
<tr>
<th>5341</th>
<th>267</th>
</tr>
</thead>
<tbody>
<tr>
<td>5,341</td>
<td>,</td>
</tr>
</tbody>
</table>

Always!
How to Break Up a Recursive Function?

def commafy(s):
    """Returns: string with commas every 3 digits
    e.g. commafy('5341267') = '5,341,267'
    Precondition: s represents a non-negative int""

    # 1. Handle small data.
    if len(s) <= 3:
        return s

    # 2. Break into two parts
    left = commafy(s[:-3])
    right = s[-3:]  # Small part on RIGHT

    # 3. Combine the result
    return left + ',' + right
More Reasons to be Careful

- Does division only affect code complexity?
  - Does it matter if we are “good” at coding?
  - What if also affects performance?
- Think about the number of recursive calls
  - Each call generates a call frame
  - Have to execute steps in definition (again)
  - So more calls == slower performance
- Want to reduce number of recursive calls
def exp(b, c):
    """Returns: b^c
    Precondition: b a float, c ≥ 0 an int"""

Approach 1

\[12^{256} = 12 \times (12^{255})\]

\[b^c = b \times (b^{c-1})\]

Approach 2

\[12^{256} = (12^{128}) \times (12^{128})\]

\[b^c = (b\times b)^{c/2} \text{ if } c \text{ even}\]
Raising a Number to an Exponent

**Approach 1**

```python
def exp(b, c):
    """Returns: b^c
    Precond: b a float, c ≥ 0 an int""
    # b^0 is 1
    if c == 0:
        return 1
    # b^c = b(b^{c-1})
    left = b
    right = exp(b, c-1)
    return left*right
```

**Approach 2**

```python
def exp(b, c):
    """Returns: b^c
    Precond: b a float, c ≥ 0 an int""
    # b^0 is 1
    if c == 0:
        return 1
    # c > 0
    if c % 2 == 0:
        return exp(b*b, c//2)
    return b*exp(b*b, (c-1)//2)
```
Raising a Number to an Exponent

### Approach 1

```python
def exp(b, c):
    
    """Returns: \( b^c \)
    Precond: b a float, c \(\geq 0 \) an int""
    # \( b^0 \) is 1
    if c == 0:
        return 1
    # \( b^c = b(b^{c-1}) \)
    left = b
    right = exp(b,c-1)
    return left*right
```

### Approach 2

```python
def exp(b, c):
    
    """Returns: \( b^c \)
    Precond: b a float, c \(\geq 0 \) an int""
    # \( b^0 \) is 1
    if c == 0:
        return 1
    # \( c > 0 \)
    if c % 2 == 0:
        return exp(b*b,c//2)
    return b*exp(b*b,(c-1)//2)
```
def exp(b, c):
    """Returns: \( b^c \)
    Precond: \( b \) a float, \( c \geq 0 \) an int""
    # \( b^0 \) is 1
    if c == 0:
        return 1

    # \( c > 0 \)
    if c % 2 == 0:
        return exp(b*b,c//2)
    return b*exp(b*b,(c-1)//2)

<table>
<thead>
<tr>
<th>c</th>
<th># of calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>16</td>
<td>5</td>
</tr>
<tr>
<td>32</td>
<td>6</td>
</tr>
</tbody>
</table>

\( 32768 \) is 215
\( b^{32768} \) needs only 215 calls!
Recursion and Objects

- **Class Person** (`person.py`)
  - Objects have 3 attributes
    - **name**: String
    - **mom**: Person (or None)
    - **dad**: Person (or None)
- Represents the “family tree”
  - Goes as far back as known
  - Attributes `mom` and `dad` are `None` if not known
- **Constructor**: `Person(n, m, d)`
  - Or `Person(n)` if no `mom`, `dad`
Recursion and Objects

def num_ancestors(p):
    """Returns: num of known ancestors
    Pre: p is a Person"
    # 1. Handle small data.
    # No mom or dad (no ancestors)

    # 2. Break into two parts
    # Has mom or dad
    # Count ancestors of each one
    # (plus mom, dad themselves)

    # 3. Combine the result
Recursion and Objects

def num_ancestors(p):
    """Returns: num of known ancestors
    Pre: p is a Person"""
    # 1. Handle small data.
    if p.mom == None and p.dad == None:
        return 0
    # 2. Break into two parts
    moms = 0
    if not p.mom == None:
        moms = 1+num_ancestors(p.mom)
    dads = 0
    if not p.dad == None:
        dads = 1+num_ancestors(p.dad)
    # 3. Combine the result
    return moms+dads
Is All Recursion Divide and Conquer?

- Divide and conquer implies two halves “equal”
  - Performing the same check on each half
  - With some optimization for small halves
- Sometimes we are given a recursive definition
  - Math formula to compute that is recursive
  - String definition to check that is recursive
  - Picture to draw that is recursive
  - Example: $n! = n \times (n-1)!$
- In that case, we are just implementing definition
Example: Palindromes

- String with $\geq 2$ characters is a palindrome if:
  - its first and last characters are equal, and
  - the rest of the characters form a palindrome

- Example:
  - AMANAPLANACANALPANAMA

- Function to Implement:
  
  ```python
def ispalindrome(s):
    """Returns: True if s is a palindrome"""
  ```
Example: Palindromes

- String with \( \geq 2 \) characters is a palindrome if:
  - its first and last characters are equal, and
  - the rest of the characters form a palindrome

```python
def ispalindrome(s):
    """Returns: True if s is a palindrome"""
    if len(s) < 2:
        return True
    ends = s[0] == s[-1]
    middle = ispalindrome(s[1:-1])
    return ends and middle
```

Recursive case
Base case
Recursive Definition
Example: Palindromes

- String with $\geq 2$ characters is a palindrome if:
  - its first and last characters are equal, and
  - the rest of the characters form a palindrome.

```python
def ispalindrome(s):
    """Returns: True if s is a palindrome""
    if len(s) < 2:
        return True
    ends = s[0] == s[-1]
    middle = ispalindrome(s[1:-1])
    return ends and middle
```

But what if we want to deviate?
def ispalindrome2(s):
    """Returns: True if s is a palindrome
Case of characters is ignored."""
    if len(s) < 2:
        return True
    # Halves not the same; not divide and conquer
    ends = equals_ignore_case(s[0], s[-1])
    middle = ispalindrome(s[1:-1])
    return ends and middle
Recursive Functions and Helpers

```python
def ispalindrome2(s):
    """Returns: True if s is a palindrome
    Case of characters is ignored."
    if len(s) < 2:
        return True
    ends = equals_ignore_case(s[0], s[-1])
    middle = ispalindrome(s[1:-1])
    return ends and middle
```

def ispalindrome2(s):
    """Returns: True if s is a palindrome
    Case of characters is ignored."""
    if len(s) < 2:
        return True
    # Halves not the same; not divide and conquer
    ends = equals_ignore_case(s[0], s[-1])
    middle = ispalindrome(s[1:-1])
    return ends and middle

def equals_ignore_case(a, b):
    """Returns: True if a and b are same ignoring case""
    return a.upper() == b.upper()
def ispalindrom3(s):
    """Returns: True if s is a palindrome
    Case of characters and non-letters ignored."""
    return ispalindrom2(depunct(s))

def depunct(s):
    """Returns: s with non-letters removed""
    if s == "":
        return s
    # Combine left and right
    if s[0] in string.letters:
        return s[0]+depunct(s[1:])
    # Ignore left if it is not a letter
    return depunct(s[1:])

Use helper functions!
- Sometimes the helper is a recursive function
- Allows you break up problem in smaller parts
“Turtle” Graphics: Assignment A4

Turn

Move

Draw Line

Change Color
Example: Space Filling Curves

**Challenge**

- Draw a curve that
  - Starts in the left corner
  - Ends in the right corner
  - Touches every grid point
  - Does not touch or cross itself anywhere

- Useful for analysis of 2-dimensional data
Hilbert’s Space Filling Curve

Hilbert(1):

Hilbert(2):

Hilbert(n):

$$2^n$$

$$H(n-1)$$

down

$$H(n-1)$$

left

$$H(n-1)$$

right
Hilbert’s Space Filling Curve

Basic Idea

• Given a box
• Draw $2^n \times 2^n$ grid in box
• Trace the curve
• As $n$ goes to $\infty$, curve fills box