Recursion II
Outline

- **Recursion**
  - A method calling itself
    - A new way of thinking about a problem
    - A powerful programming paradigm

- **Examples:**
  - Last time:
    - Factorial, binary search, H-tree, Fibonacci
  - Today:
    - Brownian Motion
    - Sorting
def mystery(n):
    print(n)
    if n <= 0:
        return       #Pop
    mystery(n - 1);  #Push
    # Return here
    mystery(n - 2);  #Push
    # Return here
    #Pop

if __name__ == "__main__":
    mystery(3)       #Push
    # Return here
    #Pop
def mystery(n):
    print(n)
    if n <= 0:
        return       #Pop
    mystery(n - 1);  #Push
    # Return here
    mystery(n - 2);  #Push
    # Return here
    #Pop

if __name__ == "__main__":
    mystery(3)       #Push
    # Return here
    #Pop
Recursion Walkthrough

```python
def mystery(n):
    print(n)
    if n <= 0:
        #Pop
        return
    mystery(n - 1);  #Push
    # Return here
    mystery(n - 2);  #Push
    # Return here
    #Pop

if __name__ == "__main__":
    mystery(3)       #Push
    # Return here
    #Pop
```

**Call Stack**

<table>
<thead>
<tr>
<th>Call Stack</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>main</td>
<td></td>
</tr>
<tr>
<td>mystery(3)</td>
<td>3</td>
</tr>
<tr>
<td>mystery(3-1)</td>
<td>2</td>
</tr>
<tr>
<td>mystery(n - 2)</td>
<td></td>
</tr>
<tr>
<td>mystery(n - 1)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Recursion Walkthrough

def mystery(n):
    print(n)
    if n <= 0:
        return       #Pop
    mystery(n - 1);  #Push
    # Return here
    mystery(n - 2);  #Push
    # Return here
    #Pop

if __name__ == "__main__":
    mystery(3)       #Push
    # Return here
    #Pop

Call Stack     n

mystery(2-1)   2
mystery(3-1)   3
mystery(3)     3
main            2
Recursion Walkthrough

```python
def mystery(n):
    print(n)
    if n <= 0:
        return       #Pop
    mystery(n - 1);  #Push
        # Return here
    mystery(n - 2);  #Push
        # Return here
    #Pop

if __name__ == "__main__":
    mystery(3)       #Push
        # Return here
    #Pop
```

<table>
<thead>
<tr>
<th>Call Stack</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td>mystery(1-1)</td>
<td>1</td>
</tr>
<tr>
<td>mystery(2-1)</td>
<td>2</td>
</tr>
<tr>
<td>mystery(3-1)</td>
<td>3</td>
</tr>
<tr>
<td>mystery(3)</td>
<td></td>
</tr>
<tr>
<td>main</td>
<td></td>
</tr>
</tbody>
</table>

3 2 1
def mystery(n):
    print(n)
    if n <= 0:
        return  # Pop
    mystery(n - 1);  # Push
    # Return here
    mystery(n - 2);  # Push
    # Return here
    # Pop

if __name__ == "__main__":
    mystery(3)  # Push
    # Return here
    # Pop
def mystery(n):
    print(n)
    if n <= 0:
        #Pop
        return
    mystery(n - 1);  #Push
    # Return here
    mystery(n - 2);  #Push
    # Return here
    #Pop

if __name__ == "__main__":
    mystery(3)       #Push
    # Return here
    #Pop

Call Stack | n
-----------|------
            | 3
            | 2
            | 1
            | 0
main
def mystery(n):
    print(n)
    if n <= 0:
        return  # Pop
    mystery(n - 1);  # Push
    # Return here
    mystery(n - 2);  # Push
    # Return here
    # Pop

if __name__ == '__main__':
    mystery(3)  # Push
    # Return here
    # Pop

Call Stack | n
------------|------
mystery(1-2) | 1
mystery(2-1) | 2
mystery(3-1) | 3
mystery(3) |
main |
def mystery(n):
    print(n)
    if n <= 0:
        return  # Pop
    mystery(n - 1);  # Push
    # Return here
    mystery(n - 2);  # Push
    # Return here
    # Pop

if __name__ == "__main__":
    mystery(3)  # Push
    # Return here
    # Pop
def mystery(n):
    print(n)
    if n <= 0:
        return  # Pop
    mystery(n - 1);  # Push
    # Return here
    mystery(n - 2);  # Push
    # Return here
    # Pop

if __name__ == "__main__":
    mystery(3);  # Push
    # Return here
    # Pop

Call Stack | n
---|---
mystery(3-1) | 3
mystery(3) | 2
main | 3
3 2 1 0 -1
```python
def mystery(n):
    print(n)
    if n <= 0:
        return    # Pop
    mystery(n - 1);    # Push
    # Return here
    mystery(n - 2);    # Push
    # Return here
    # Pop

if __name__ == "__main__":
    mystery(3)    # Push
    # Return here
    # Pop
```

### Call Stack

<table>
<thead>
<tr>
<th>Function</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>main</td>
<td></td>
</tr>
<tr>
<td>mystery(3)</td>
<td>3</td>
</tr>
<tr>
<td>mystery(3-1)</td>
<td>3</td>
</tr>
<tr>
<td>mystery(2-2)</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>
def mystery(n):
    print(n)
    if n <= 0:
        return  # Pop
    mystery(n - 1);  # Push
    # Return here
    mystery(n - 2);  # Push
    # Return here
    # Pop

if __name__ == "__main__":
    mystery(3)  # Push
    # Return here
    # Pop

Call Stack | n
----------|------
          | 3
          | 2
          | 1
          | 0
          | -1
          | 0

main
def mystery(n):
    print(n)
    if n <= 0:
        return  # Pop
    mystery(n - 1);  # Push
    # Return here
    mystery(n - 2);  # Push
    # Return here
    # Pop

if __name__ == "__main__":
    mystery(3)  # Push
    # Return here
    # Pop

    Call Stack | n
    |-----------|-----
    | 3         | mystery(3)
    | main      |
def mystery(n):
    print(n)
    if n <= 0:
        return  # Pop
    mystery(n - 1);  # Push
    # Return here
    mystery(n - 2);  # Push
    # Return here
    # Pop

if __name__ == "__main__":
    mystery(3)  # Push
    # Return here
    # Pop

<table>
<thead>
<tr>
<th>Call Stack</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>mystery(3)</td>
<td>3</td>
</tr>
<tr>
<td>mystery(3-2)</td>
<td>1</td>
</tr>
<tr>
<td>main</td>
<td></td>
</tr>
</tbody>
</table>
def mystery(n):
    print(n)
    if n <= 0:
        return  # Pop
    mystery(n - 1);  # Push
    # Return here
    mystery(n - 2);  # Push
    # Return here
    # Pop

if __name__ == '__main__':
    mystery(3)  # Push
    # Return here
    # Pop

<table>
<thead>
<tr>
<th>Call Stack</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td>mystery(1-1)</td>
<td>1</td>
</tr>
<tr>
<td>mystery(3-2)</td>
<td>3</td>
</tr>
<tr>
<td>mystery(3)</td>
<td></td>
</tr>
<tr>
<td>main</td>
<td></td>
</tr>
</tbody>
</table>
Recursion Walkthrough

def mystery(n):
    print(n)
    if n <= 0:
        return  # Pop
    mystery(n - 1);  # Push
    # Return here
    mystery(n - 2);  # Push
    # Return here
    # Pop

if __name__ == "__main__":
    mystery(3)  # Push
    # Return here
    # Pop

Call Stack | n
------------|--------
            | 3
mystery(3) | 2
mystery(3) | 1
mystery(3-2)| 0
main        | -1
            | 0
Recursion Walkthrough

def mystery(n):
    print(n)
    if n <= 0:
        return  # Pop
    mystery(n - 1);  # Push
    # Return here
    mystery(n - 2);  # Push
    # Return here
    # Pop

if __name__ == "__main__":
    mystery(3)  # Push
    # Return here
    # Pop
def mystery(n):
    print(n)
    if n <= 0:
        return  # Pop
    mystery(n - 1);  # Push
    # Return here
    mystery(n - 2);  # Push
    # Return here
    # Pop

if __name__ == '__main__':
    mystery(3)  # Push
    # Return here
    # Pop

Call Stack | n
-----------|---
            | 3
mystery(3-2) | 3
mystery(3)  | 1
main
def mystery(n):
    print(n)
    if n <= 0:
        return       # Pop
    mystery(n - 1);  # Push
    # Return here
    mystery(n - 2);  # Push
    # Return here
    # Pop

if __name__ == "__main__":
    mystery(3)       # Push
    # Return here
    # Pop

Call Stack n

main

3
mystery(3)

-1 0 1 2 3
Recursion Walkthrough

```python
def mystery(n):
    print(n)
    if n <= 0:
        return  # Pop
    mystery(n - 1);  # Push
    # Return here
    mystery(n - 2);  # Push
    # Return here
    # Pop

if __name__ == "__main__":
    mystery(3)  # Push  # Return here  # Pop
```

**Call Stack**

<table>
<thead>
<tr>
<th>Call Stack</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
```
Brownian Motion

- Models many natural and artificial phenomenon
  - Motion of pollen grains in water
  - Price of stocks
  - Rugged shapes of mountains and clouds
Simulating Brownian Motion

- **Midpoint displacement method:**
  - Track interval \((x_0, y_0)\) to \((x_1, y_1)\)
  - Choose \(\delta\) displacement randomly from Gaussian
  - Divide in half, \(x_m = (x_0 + x_1)/2\) and \(y_m = (y_0 + y_1)/2 + \delta\)
  - Recur on the left and right intervals
Recursive Midpoint Displacement Algorithm

```python
def curve(x0, y0, x1, y1, var):
    # Base case: stop if interval is sufficiently small
    if x1 - x0 < .005:
        StdDraw.line(x0, y0, x1, y1)
        StdDraw.show(10)
        return

    xm = (x0 + x1) / 2.0
    ym = (y0 + y1) / 2.0

    # Randomly displace the y coordinate of the midpoint
    ym += random.gauss(0, math.sqrt(var))

    curve(x0, y0, xm, ym, var / 2.0)
    curve(xm, ym, x1, y1, var / 2.0)
```

---

**Diagram Notes:**
- The diagram shows a line segment from \((x_0, y_0)\) to \((x_1, y_1)\) with random displacement \(\delta\) of the midpoint \((x_m, y_m)\). The midpoint is calculated as \(x_m = (x_0 + x_1) / 2.0\) and \(y_m = (y_0 + y_1) / 2.0\). The y-coordinate is then randomly displaced by \(\delta\) using \(\text{random.gauss}(0, \sqrt{\text{var}})\).
- The algorithm terminates if the interval is sufficiently small (less than .005).
- The reduction step is recursively applied to the segments from the midpoint to the new endpoints.

---

**Code Snippet:**
- The `curve` function is defined to recursively draw the line segment with midpoint displacement.
- The base case is when the interval is less than .005, the line is drawn, and the function returns.
- In the reduction step, the midpoint is calculated, its y-coordinate is randomly displaced, and the function is called recursively on the new segments.
Plasma Cloud

- **Same idea, but in 2D**
  - Each corner of square has some color value
  - Divide into four sub-squares
  - New corners: avg of original corners, or all 4 + random
  - Recur on four sub-squares
Brownian Landscape
Divide and Conquer

- **Divide and conquer paradigm**
  - Break big problem into small sub-problems
  - Solve sub-problems recursively
  - Combine results

- **Used to solve many important problems**
  - Sorting things, mergesort: $O(N \log N)$
  - Parsing programming languages
  - Discrete FFT, signal processing
  - Multiplying large numbers
  - Traversing multiply linked structures (stay tuned)

"Divide et impera. Vendi, vidi, vici."

- Julius Caesar
Divide and Conquer: Sorting

Goal: Sort by number, ignore suit, aces high

Approach
1) Split in half (or as close as possible)
2) Give each half to somebody to sort
3) Take two halves and merge together
**Approach**

1) Split in half (or as close as possible)
2) Give each half to somebody to sort
3) Take two halves and merge together

---

**Merging**

Take card from whichever pile has lowest card
**Approach**

1) Split in half (or as close as possible)
2) Give each half to somebody to sort
3) Take two halves and merge together

**Sorted pile #1**

**Sorted pile #2**
**Approach**

1) Split in half (or as close as possible)
2) Give each half to somebody to sort
3) Take two halves and merge together
**Approach**

1) Split in half (or as close as possible)
2) Give each half to somebody to sort
3) Take two halves and merge together
**Approach**
1) Split in half (or as close as possible)
2) Give each half to somebody to sort
3) Take two halves and merge together
**Approach**

1) Split in half (or as close as possible)
2) Give each half to somebody to sort
3) Take two halves and merge together

**Sorted pile #1**

**Sorted pile #2**
**Approach**

1) Split in half (or as close as possible)
2) Give each half to somebody to sort
3) Take two halves and merge together

Sorted pile #1

Sorted pile #2
**Approach**
1) Split in half (or as close as possible)
2) Give each half to somebody to sort
3) Take two halves and merge together

Sorted pile #1

Sorted pile #2
**Approach**

1) Split in half (or as close as possible)
2) Give each half to somebody to sort
3) Take two halves and merge together

**Sorted pile #1**

**Sorted pile #2**
Approach
1) Split in half (or as close as possible)
2) Give each half to somebody to sort
3) Take two halves and merge together

Sorted pile #1

Sorted pile #2

How many operations to do the merge?
Linear in the number of cards, $O(N)$

But how did pile 1 and 2 get sorted?

Recursively of course!
Split each pile into two halves, give to different people to sort.
How many split levels?
$O(\log_2 N)$

How many merge levels?
$O(\log_2 N)$

Operations per level?
$O(N)$

Total operations?
$O(N \log_2 N)$
Summary

- **Recursion**
  - A method calling itself:
    - Sometimes just once, e.g. binary search
    - Sometimes twice, e.g. mergesort
    - Sometimes multiple times, e.g. H-tree
  - All good recursion must come to an end:
    - Base case that does NOT call itself recursively
  - A powerful tool in computer science:
    - Allows elegant and easy to understand algorithms
    - (Once you get your head around it)