Recursion
Outline

- **Recursion**
  - A method calling itself
  - All good recursion must come to an end
  - A powerful tool in computer science
    - Allows writing elegant and easy to understand algorithms
  - A new way of thinking about a problem
    - Divide and conquer
  - A powerful programming paradigm
  - Related to mathematical induction

- **Example applications**
  - Factorial
  - Binary search
  - Pretty graphics
  - Sorting things
Mathematical Induction

- **Prove a statement involving an integer N**
  - **Base case:** Prove it for small N (usually 0 or 1)
  - **Induction step:**
    - Assume true for size N-1
    - Prove it is true for size N

- **Example:**
  - Prove \( T(N) = 1 + 2 + 3 + \ldots + N = \frac{N(N + 1)}{2} \) for all N
  - **Base case:** \( T(1) = 1 = \frac{1(1 + 1)}{2} \)
  - **Induction step:**
    - Assume true for size \( N - 1 \): \( 1 + 2 + \ldots + N-1 = T(N - 1) = \frac{(N - 1)(N)}{2} \)
    - \( T(N) = 1 + 2 + 3 + \ldots + N-1 + N \)
    - \( = \frac{(N - 1)(N)}{2} + N \)
    - \( = \frac{(N - 1)(N)}{2} + 2N / 2 \)
    - \( = \frac{(N - 1 + 2)(N)}{2} \)
    - \( = \frac{(N + 1)(N)}{2} \)
Hello Recursion

- **Goal:** Compute factorial $N! = 1 \times 2 \times 3 \ldots \times N$
  - **Base case:** $0! = 1$
  - **Induction step:**
    - Assume we know $(N-1)!$
    - Use $(N-1)!$ to find $N!$

```python
import sys
def fact(N):
    if N == 0:
        return 1
    return fact(N - 1) * N

if __name__ == "__main__":
    N = int(sys.argv[1])
    print(str(N) + "! = " + str(fact(N)))
```

<table>
<thead>
<tr>
<th>$N!$</th>
<th>Calculation</th>
</tr>
</thead>
<tbody>
<tr>
<td>4!</td>
<td>$4 \times 3!$</td>
</tr>
<tr>
<td>3!</td>
<td>$3 \times 2!$</td>
</tr>
<tr>
<td>2!</td>
<td>$2 \times 1!$</td>
</tr>
<tr>
<td>1!</td>
<td>$1 \times 0!$</td>
</tr>
<tr>
<td>0!</td>
<td>1</td>
</tr>
</tbody>
</table>

4! = 4 \times 3 \times 2 \times 1 = 24
def fact(N):
    print("start, fact " + str(N))
    if N == 0:
        print("end base, fact " + str(N))
        return 1
    step = fact(N - 1)
    print("end, fact " + str(N))
    return step * N

start, fact 4
start, fact 3
start, fact 2
start, fact 1
start, fact 0
end base, fact 0
end, fact 1
end, fact 2
end, fact 3
end, fact 4
4! = 24

5 levels of fact()
Recursion vs. Iteration

• **Recursive algorithms also have an iterative version**

```python
def fact(N):
    if N == 0:
        return 1
    return fact(N - 1) * N
```

```python
def fact(N):
    result = 1
    for i in range(1, N+1):
        result *= i
    return result
```

Recursive algorithm                      Iterative algorithm

• **Reasons to use recursion:**
  - Code is more compact and easier to understand
  - Easier to reason about correctness

• **Reasons **not** to use recursion:**
  - If you end up recalculating things repeatedly (stay tuned)
  - Processor with very little memory (e.g. 8051 = 128 bytes)
A Useful Example of Recursion

- **Binary search**
  - Given an array of $N$ sorted numbers
  - Find out if some number $t$ is in the list
  - Do it faster than going linearly through the list, i.e. $O(N)$

- **Basic idea:**
  - Like playing higher/lower number guessing:

<table>
<thead>
<tr>
<th>Me</th>
<th>You</th>
</tr>
</thead>
<tbody>
<tr>
<td>I'm thinking of a number between 1 and 100.</td>
<td>50</td>
</tr>
<tr>
<td>Higher</td>
<td>75</td>
</tr>
<tr>
<td>Lower</td>
<td>63</td>
</tr>
<tr>
<td>Higher</td>
<td>69</td>
</tr>
<tr>
<td>Higher</td>
<td>72</td>
</tr>
<tr>
<td>You got it</td>
<td>Wow I'm super smart!</td>
</tr>
</tbody>
</table>
**Binary Search**

- **Binary search algorithm**
  - Find midpoint of sorted array
  - Is that element the one you're looking for?
    - If yes, you found it!
  - If target is < midpoint, search lower half
  - If target is > midpoint, search upper half

- **Example:** Is 5 in this sorted array?

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>8</th>
<th>9</th>
<th>14</th>
<th>14</th>
<th>50</th>
<th>88</th>
<th>89</th>
</tr>
</thead>
</table>

- target (value) = 5
- low (index) = 0
- high (index) = 10
- midpoint (index) = (0 + 10) / 2 = 5
Binary Search

- Binary search algorithm
  - Find midpoint of sorted array
  - Is that element the one you're looking for?
    - If yes, you found it!
  - If target is < midpoint, search lower half
  - If target is > midpoint, search upper half

- Example: Is 5 in the sorted array?

| 1 | 2 | 2 | 5 | 8 | 9 | 14 | 14 | 50 | 88 | 89 |

- target (value) = 5
- low (index) = 0
- high (index) = 4
- midpoint (index) = (0 + 4) / 2 = 2
Binary Search

- **Binary search algorithm**
  - Find midpoint of sorted array
  - Is that element the one you're looking for?
    - If yes, you found it!
  - If target is < midpoint, search lower half
  - If target is > midpoint, search upper half

- **Example: Is 5 in the sorted array?**

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>2</th>
<th>5</th>
<th>8</th>
<th>9</th>
<th>14</th>
<th>14</th>
<th>50</th>
<th>88</th>
<th>89</th>
</tr>
</thead>
</table>

target (value) = 5
low (index) = 3
high (index) = 4
midpoint (index) = (3 + 4) / 2 = 3
Binary Search

- **Binary search algorithm**
  - Find midpoint of sorted array
  - Is that element the one you're looking for?
    - If yes, you found it!
  - If target is < midpoint, search lower half
  - If target is > midpoint, search upper half

- **Example:** Is 5 in the sorted array?

```
1   2   2   5   8   9  14  14  50  88  89
```

YES. Element at new midpoint is target!
def binarySearch(target, low, high, d):
    mid = int((low + high) / 2)
    print("low", low, "high", high, "mid", mid)

    if d[mid] == target:
        return True

    if high < low:
        return False

    if d[mid] < target:
        return binarySearch(target, mid + 1, high, d)
    else:
        return binarySearch(target, low, mid - 1, d)

if __name__ == "__main__":
    d = [1, 2, 2, 5, 8, 9, 14, 14, 50, 88, 89]
    target = int(sys.argv[1])
    print("found " + str(target) + "? " + str(binarySearch(target, 0, len(d)-1, d)))
Things to Avoid

- Missing base case
  ```python
def fact(N):
    return fact(N - 1) * N
  ```

- No convergence guarantee
  ```python
def badIdea(N):
    if N == 1:
      return 1.0
    return badIdea(1 + N/2) + 1.0/N
  ```

- Both result in infinite recursion = stack overflow

% python FactBad.py 5
Traceback (most recent call last):
  File "FactBad.py", line 20, in <module>
    print(str(N) + "! = " + str(fact(N)))
  File "FactBad.py", line 15, in fact
    return fact(N - 1) * N
  File "FactBad.py", line 15, in fact
    return fact(N - 1) * N
  File "FactBad.py", line 15, in fact
    return fact(N - 1) * N
  [Previous line repeated 996 more times]
RecursionError: maximum recursion depth exceeded
Sometimes We Don't Know...

- **Collatz sequence**
  - If $N = 1$, stop
  - If $N$ is even, divide by 2
  - If $N$ is odd, multiply by 3 and add 1
  - e.g. $24 \rightarrow 12 \rightarrow 6 \rightarrow 3 \rightarrow 10 \rightarrow 5 \rightarrow 16 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1$
  - No one knows if this terminates for all $N$!

```python
def collatz(N):
    print(N)
    if N == 1:
        return
    elif N % 2 == 0:
        collatz(int(N / 2))
    else:
        collatz(3 * N + 1)
```
THE COLLATZ CONJECTURE STATES THAT IF YOU PICK A NUMBER, AND IF IT'S EVEN DIVIDE IT BY TWO AND IF IT'S ODD MULTIPLY IT BY THREE AND ADD ONE, AND YOU REPEAT THIS PROCEDURE LONG ENOUGH, EVENTUALLY YOUR FRIENDS WILL STOP CALLING TO SEE IF YOU WANT TO HANG OUT.

http://xkcd.com/710/
H-tree

- H-tree of order N
  - Draw an H
  - Recursively draw 4 H-trees
    - One at each "tip" of the original H, half the size
    - Stop once N = 0
Fibonacci Numbers

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, ...

\[
\begin{align*}
F_0 &= 0 \\
F_1 &= 1 \\
F_n &= F_{n-1} + F_{n-2}
\end{align*}
\]

Fibonacci numbers. A natural fit for recursion?

def fib(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    return fib(n - 1) + fib(n - 2)

Yellow Chamomile head showing the arrangement in 21 (blue) and 13 (aqua) spirals.
Trouble in Recursion City...

<table>
<thead>
<tr>
<th>N</th>
<th>time, fib(N)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.000</td>
</tr>
<tr>
<td>20</td>
<td>0.002</td>
</tr>
<tr>
<td>30</td>
<td>0.011</td>
</tr>
<tr>
<td>40</td>
<td>0.661</td>
</tr>
<tr>
<td>41</td>
<td>1.080</td>
</tr>
<tr>
<td>42</td>
<td>1.748</td>
</tr>
<tr>
<td>43</td>
<td>2.814</td>
</tr>
<tr>
<td>44</td>
<td>4.531</td>
</tr>
<tr>
<td>45</td>
<td>7.371</td>
</tr>
<tr>
<td>46</td>
<td>11.860</td>
</tr>
<tr>
<td>47</td>
<td>19.295</td>
</tr>
<tr>
<td>48</td>
<td>31.319</td>
</tr>
<tr>
<td>49</td>
<td>50.668</td>
</tr>
<tr>
<td>50</td>
<td>81.542</td>
</tr>
</tbody>
</table>

Bad news bears: Order of growth = Exponential!
Efficient Fibonacci Version

- Remember last two numbers
  - Use $F_{n-2}$ and $F_{n-1}$ to get $F_n$

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377
Efficient Fibonacci Version

- Remember last two numbers
  - Use $F_{n-2}$ and $F_{n-1}$ to get $F_n$
Efficient Fibonacci Version

- **Remember last two numbers**
  - Use $F_{n-2}$ and $F_{n-1}$ to get $F_n$

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377
Efficient Fibonacci Version

- **Remember last two numbers**
  - Use $F_{n-2}$ and $F_{n-1}$ to get $F_n$

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377

```python
def fibFast(n):
    n2 = 0
    n1 = 1
    if n == 0:
        return 0
    for i in range(1, n):
        n0 = n1 + n2
        n2 = n1
        n1 = n0
    return n1
```

<table>
<thead>
<tr>
<th>N</th>
<th>time, fib(N)</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>0.001</td>
</tr>
<tr>
<td>100</td>
<td>0.001</td>
</tr>
<tr>
<td>200</td>
<td>0.001</td>
</tr>
<tr>
<td>400</td>
<td>0.001</td>
</tr>
<tr>
<td>10,000,000</td>
<td>0.010</td>
</tr>
<tr>
<td>20,000,000</td>
<td>0.016</td>
</tr>
<tr>
<td>40,000,000</td>
<td>0.028</td>
</tr>
<tr>
<td>80,000,000</td>
<td>0.051</td>
</tr>
<tr>
<td>160,000,000</td>
<td>0.096</td>
</tr>
</tbody>
</table>
Summary

- **Recursion**
  - A method calling itself
  - All good recursion must come to an end
  - A powerful tool in computer science
    - Allows writing *elegant and easy to understand* algorithms
  - A new way of thinking about a problem
    - Divide and conquer
  - A powerful programming paradigm
  - Related to mathematical induction

- **Example applications**
  - Factorial
  - Binary search
  - Pretty graphics
  - Sorting things
Here is a recursive definition for exponentiation. Write a recursive method to implement this definition. The test main is provided for you:

import sys

def fastExp(a, n):
    # Your code goes here...

    if __name__ == '__main__':
        a = int(sys.argv[1])
        n = int(sys.argv[2]);
        print(a, " raised to the ", n, " is: ", fastExp(a, n))

- Open Moodle, go to CSCI 136, Section 11
- Open the dropbox for today – Activity 4: Recursion
- Drag and drop your program file to the Moodle dropbox
- You get: 1 point if you turn in something, 2 points if you turn in something that is correct.