What is this?

```python
def mystery(n):
    if (n == 1):
        print '*'
    else:
        print ' ',
        mystery(n - 1)
```

Recursion

- **recursion**: The definition of an operation in terms of itself.
  - Solving a problem using recursion depends on solving smaller occurrences of the same problem.

- **recursive programming**: Writing methods that call themselves
  - An equally powerful substitute for iteration (loops)
  - Particularly well-suited to solving certain types of problems

Why learn recursion?

- A different way of thinking about problems
- Can solve some problems better than iteration
- Leads to elegant, simple, concise code (when used well)
- Some programming languages ("functional" languages such as Scheme, ML, and Haskell) use recursion exclusively (no loops)

Exercise

- **Exercise**: How many students are directly behind you?
  - We all have poor vision, and can only see the people right next to us. So you can't just look back and count.
  - But you are allowed to ask questions of the person behind you.
  - How can we solve this problem? (recursively)
The idea

- Recursion is all about breaking a big problem into smaller occurrences of that same problem.
  - Each person can solve a small part of the problem.
    - What is a small version of the problem that would be easy to answer?
    - What information from a neighbor might help you?

Recursive algorithm

- Number of people behind me:
  - If there is someone behind me, ask him/her how many people are behind him/her.
    - When they respond with a value N, then I will answer N + 1.
  - If there is nobody behind me, I will answer 0.

Cases

- Every recursive algorithm has at least 2 cases:
  - base case: A simple instance that can be answered directly.
  - recursive case: A more complex instance of the problem that cannot be directly answered, but can instead be described in terms of smaller instances.
  - Can have more than one base or recursive case, but all have at least one of each.
  - A crucial part of recursive programming is identifying these cases.

Base and Recursive Cases: Example

```python
def print_stars(n):
    if (n == 1):
        # base case - print one star and end line
        print '*'
    else:
        # recursive case; print one more star
        print '*',
        print_stars(n - 1)
```

Recursion Zen

- An even simpler, base case is n=0:

```python
def print_stars(n):
    if (n == 0):
        # base case - end the line of input
        print
    else:
        # recursive case; print one more star
        print '*',
        print_stars(n - 1)
```

- Recursion Zen: The art of identifying the best set of cases for a recursive algorithm and expressing them elegantly.

Everything recursive can be done non-recursively

```python
# Prints a line containing a given number of stars.
print_stars(n):
    for i in range(n):
        print '*',
    print
```

or even simpler:

```python
print_stars(n):
    print '*' * n
```
Recursive power example

- Write a function that computes $x^n$.
  \[ x^n = x \times x \times x \times \ldots \times x \text{ (n times)} \]

- An iterative solution:

```python
def pow(x, n):
    product = 1
    for i in range(n):
        product = product * x
    return product
```

Exercise solution

```python
def pow(x, n):
    """Returns x^n""
    if n == 0:
        # base case: any number to 0th power is 1
        return 1
    else:
        # recursive case: x^n = x * x^(n-1)
        return x * pow(x, n-1)
```

How recursion works

- Each call sets up a new instance of all the parameters and the local variables.
- When the function completes, control returns to the function that invoked it (which might be another invocation of the same function).

```
pow(4, 3) = 4 * pow(4, 2)
= 4 * 4 * pow(4, 1)
= 4 * 4 * 4 * pow(4, 0)
= 4 * 4 * 4 * 1
= 64
```

Infinite recursion

- A function with a missing or badly written base case can cause infinite recursion.

```python
def pow(x, n):
    return x * pow(x, n - 1)  # Oops! Forgot base case
```

```
pow(4, 3) = 4 * pow(4, 2)
= 4 * 4 * pow(4, 1)
= 4 * 4 * 4 * pow(4, 0)
= 4 * 4 * 4 * 4 * pow(4, -1)
= 4 * 4 * 4 * 4 * 4 * pow(4, -2)
= ... crashes
```

An optimization

- Notice the following mathematical property:
  \[ 3^{12} = 531441 = (3^2)^6 \]
  \[ = ((3^2)^3)^2 \]

- When does this "trick" work?
- How can we incorporate this optimization into our pow method?
- What is the benefit of this trick?

Coding it

```python
def pow(x, n):
    if n == 0:
        # base case: any number to 0th power is 1
        return 1
    elif n % 2 == 0:
        # recursive case 1: x^n = (x^2)^(n/2)
        return pow(x * x, n / 2)
    else:
        # recursive case 2: x^n = x * x^(n-1)
        return x * pow(x, n - 1)
```
Activation records

- **activation record**: memory that Python allocates to store information about each running method
  - return point ("RP"), argument values, local variable values
  - Python stacks up the records as methods are called; a method's activation record exists until it returns

```
<table>
<thead>
<tr>
<th>x</th>
<th>n</th>
<th>pow(x, n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0</td>
<td>pow(4, 0)</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>pow(4, 1)</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>pow(4, 2)</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>pow(4, 3)</td>
</tr>
</tbody>
</table>
```

Recursion - examples

- Problem: given a string as input, write it backward
  - Base case?
  - Recursion

Fractals – the Koch curve

Binary search - recursively

```python
def binary_search(dictionary, word):
    if dictionary has one page:  # base case
        scan the page for word
    else:  # recursive case
        open the dictionary to a point near the middle
determine which half of the dictionary contains word
        if (word is in first half of the dictionary):
            binary_search(first half of dictionary, word)
        else:
            binary_search(second half of dictionary, word)
```

Binary search

```python
# Returns the index of an occurrence of the given value in the given array, or -1 if not found.
def binary_search(a, value):
    return binary_search(a, value, 0, len(a) - 1)

# Recursive helper to implement search.
def binary_search_helper(a, value, left, right):
    if (left > right):
        return -1  # not found
    else:
        mid = (first + last) / 2
        if (a[mid] == value):
            return mid  # found it!
        elif (a[mid] < value):
            # middle element too small; search right half
            return binary_search_helper(a, value, mid+1, right)
        else:
            # middle element too large; search left half
            return binary_search_helper(a, value, left, mid-1)
```