Recursion

Introduction

- Recursion is a way of defining functions self-referentially.
- Examples:
  - Droste effect;
  - (Parenthetical comments (especially comments within comments), etc.);
  - A chain of phone callers on hold;
  - Inductive Proofs.

Example: Factorial

- Write a function that, given n, computes n!
  \[ n! = 1 \times 2 \times \ldots \times (n-1) \times n \]
- Example:
  \[ 5! = 1 \times 2 \times 3 \times 4 \times 5 = 120 \]
- Assumptions:
  - Receive n, a non-negative integer.
  - Return n!, a double (to avoid integer overflow).

Preliminary Analysis

- This can be viewed as a counting problem, so we could solve it iteratively:

```
def factorial(n):
    result = 1
    for i in range(1, n+1):
        result *= i
    return result
```

An Alternate Analysis

- Consider the following alternate analysis:
  \[ n! = 1 \times 2 \times \ldots \times (n-1) \times n \]
  \[ (n-1)! = 1 \times 2 \times \ldots \times (n-1) \]
  \[ n! = (n-1)! \times n \]
- Historically, this is how the factorial function was defined.

The Mechanics of Recursion

Design recursive functions using a three-step process:

1. Identify a base case - an instance of the problem whose solution is trivial.
   E.g., The factorial function has two base cases:
   - if \( n = 0 \): \( n! = 1 \)
   - if \( n = 1 \): \( n! = 1 \)
2. Identify an induction step - a means of solving the non-trivial instances of the problem using one or more “smaller” instances of the problem. E.g.,
\[ n! = (n-1)! \times n \]

Mechanics (cont.)

3. Form an algorithm that includes the base case and induction step, and ensures that each inductive step moves toward the base case.

_factorial:_
- Receive n.
- If \( n > 1 \) then
  - Return \( \text{factorial}(n-1) \times n \).
- Else
  - Return 1.

Implementation

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

Example: Towers of Hanoi

- Move disks from a source pin to a target pin, using the third pin as an auxiliary.
- Rules:
  - move only one disk at a time;
  - never put a larger disk onto a smaller one.

Design

- Today’s problem is to write a program that generates the instructions for the priests to follow in moving the disks.
- While quite difficult to solve iteratively, this problem has a simple and elegant recursive solution.

Design (cont.)

- Base case:
- Inductive case:
Algorithm
We can combine these steps into the following algorithm:
0. Receive \( n, \) src, dest, aux.
1. If \( n > 1 \):
   a. move \((n-1, \text{src}, \text{aux}, \text{dest})\);
   b. move \((1, \text{src}, \text{dest}, \text{aux})\);
   c. move \((n-1, \text{aux}, \text{dest}, \text{src})\);
Else
   Display “Move the top disk from “ + src + “ to “ + dest.

Implementation
```python
def tower_instructions(n, src, dest, aux):
    if n > 1:
        tower_instructions(n-1, src, aux, dest)
        tower_instructions(1, src, dest, aux)
        tower_instructions(n-1, aux, dest, src)
    else:
        print("Move the top disk from " + src + " to " + dest)

print('Welcome to towers!')
disks = int(input('Please enter the number of disks:'))
tower_instructions(disks, 'A', 'B', 'C')
```

Algorithm Analysis
How many “moves” does it take to solve this problem as a function of \( n \), the number of disks to be moved.

<table>
<thead>
<tr>
<th>( n )</th>
<th># of moves required</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>15</td>
</tr>
<tr>
<td>5</td>
<td>31</td>
</tr>
<tr>
<td>...</td>
<td>( 2^{i-1} )</td>
</tr>
<tr>
<td>64</td>
<td>( 2^{64-1} )</td>
</tr>
</tbody>
</table>

Analysis (cont.)
Given a “super-printer” that can generate and print 1,048,576 \( (2^{20}) \) instructions/second, how long would it take to print \( 2^{64} - 1 \) instructions?

\[
2^{64}/2^{20} = 2^{44} \text{ seconds}
\]

\[
= 2^{44}/2^4 = 2^{40} \text{ minutes}
\]

\[
= 2^{40}/2^2 = 2^{38} \text{ hours}
\]

\[
= 2^{38}/2^2 = 2^{34} \text{ days}
\]

\[
= 2^{34}/2^2 = 2^{32} \text{ years}
\]

\[
= 2^{32}/2^2 = 2^{30} \text{ centuries}
\]

\[
= 2^{30}/2^2 = 2^{28} \approx 128 \text{ millennia}
\]

Graphical Examples
- Sierpinski triangles
- Tree fractals
- Koch’s snowflake