Recursion

APRIL 15TH, 2015
What is *recursion*?

- In most programming languages, a function can call itself.
- A function that calls itself is called a *recursive* function.
- Why might this be useful?
- Many computational problems can be solved by:
  - solving smaller versions of the problem
  - combining the solutions (of the smaller problems) to obtain the solution of the bigger problem.
- Recursion is a natural way of implementing this problem-solving approach.
Recursion Outside Computer Science

• Search for “recursion” on Google and you will be asked the question “Did you mean recursion?”

• Jokes aside, recursion plays an important role in natural languages, fractals, etc.
Our First Example

- Write a function to compute the factorial of a given positive integer \( n \).

- A simple *iterative* solution:

```python
def factorial(n):
    if n == 0 or n == 1:
        return 1
    mult = 1
    for i in range(2, n+1):
        mult = mult*i
    return mult
```
The Recursive Approach

- Note that
  \[ n! = (n-1)! \times n \]

- Thus to compute the factorial of \( n \), we just have to compute the factorial of \( n-1 \) and then multiply the answer by \( n \).

- Example: \( 4! = 3! \times 3 \)
Recursive Function for Factorial (Version 1)

def factorial(n):
    return factorial(n-1) * n

- To compute 4!, we first compute 3!
- To compute 3!, we first compute 2!
- To compute 2!, we first compute 1!
- To compute 1!, we first compute 0!
- To compute 0!, we first compute -1!
- ...

- You’ll notice that this goes on forever – we have infinite loop!
So we need a base case...

- To complete a recursive function, it is not enough to specify how the problem is solved in terms of smaller problems.
- It is also necessary to enumerate the smallest problem(s) and specify how to solve these.
- This is called the *base case* of the recursion.
- Without a base case, we will have *infinite descent*!
- The base case for the factorial function could be \( n = 0 \). In this case, we know the answer is 1.
def factorial(n):
    # Base Case
    if n == 0:
        return 1

    # Recursive Case
    return factorial(n-1) * n

Notes:
- Look, no loops (while- or for-)! Recursive function calls can often replace looping behavior.
- Recursive functions are typically organized into one or more recursive cases and one or more base cases.
The Fibonacci sequence is 1, 1, 2, 3, 5, 8, 13, 21, ...

More precisely, $F(1) = 1$, $F(2) = 1$, and

For any $n > 2$, $F(n) = F(n-1) + F(n-2)$.

Notice that the definition of this sequence itself is recursive and so computing the $n$th Fibonacci number can be easily solved by a recursive function.
def fibonacci(n):
    # Base cases
    if n == 1 or n == 2:
        return 1
    else:
        # Recursive case
        return fibonacci(n-1) + fibonacci(n-2)
Partial Recursion Tree for fibonacci(8)
The inefficiency of recursive fibo

• It is fairly easy to implement a non-recursive function that computes the $n^{\text{th}}$ Fibonacci number.

• Comparing the efficiency of the recursive and non-recursive versions is quite instructive.

• Here are the running times to compute the 40$^{\text{th}}$ Fibonacci number:
  
<table>
<thead>
<tr>
<th></th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slow fibo</td>
<td>112.93942213058472</td>
</tr>
<tr>
<td>Fast fibo</td>
<td>2.5033950805664062e-05</td>
</tr>
</tbody>
</table>
Inefficiency of the Recursive Function

- Sometimes recursive functions can be extremely inefficient. The \textit{fibonacci} function is an example.

- The (partial) recursion tree for \texttt{fibonacci(8)} hints at why this might be: the same problem is being solved many times.

- The function has no recollection of having solved the problem earlier!
Efficiency of fibonacci(n): Exponential Growth

The plot shows the running time (in seconds) of fibo(n) for n = 20, 21,…, 30.
Example: Efficiently computing the power function

- The fibonacci example showed that sometimes recursion can be extremely inefficient compared.

- However, sometimes a recursive-approach can lead to tremendous gains in efficiency.

- We will see several important examples of this.

- **Problem:** Given a real number $a$ and a nonnegative integer $n$, compute $a^n$. 