CS 142
Recursion

Rhodes College

Announcements

Reminder:

- Program 3 due 2/18 by 11:55pm

Factorial Function

Write a function that computes the factorial of a number using a loop (for or while loop is fine).

Examples:

```
factorial(5) returns 120          #(1*2*3*4*5)
factorial(8) returns 40320        #(1*2*3*4*5*6*7*8)
```


Factorial Function

- A different view of the same problem
- `fact(1) = 1`
- `fact(2) = 1 * 2`
- `fact(3) = 1 * 2 * 3`
- `fact(4) = 1 * 2 * 3 * 4`
- `fact(5) = 1 * 2 * 3 * 4 * 5`
- `Notice that each product involves computing the entire product on the row above.`

Factorial Function

- fact(1) = 1
- fact(2) = 1 * 2
- fact(3) = 1 * 2 * 3
- fact(4) = 1 * 2 * 3 * 4
- fact(5) = 1 * 2 * 3 * 4 * 5

Factorial Function

- fact(1) = 1
- fact(2) = 1 * 2
- fact(3) = 1 * 2 * 3
- fact(4) = 1 * 2 * 3 * 4
- fact(5) = 1 * 2 * 3 * 4 * 5

Factorial Function

- fact(1) = 1
- fact(2) = 1 * 2
- fact(3) = 1 * 2 * 3
- fact(4) = 1 * 2 * 3 * 4
- fact(5) = 1 * 2 * 3 * 4 * 5

Factorial Function

- fact(1) = 1
- fact(2) = 1 * 2
- fact(3) = 1 * 2 * 3
- fact(4) = 1 * 2 * 3 * 4
- fact(5) = 1 * 2 * 3 * 4 * 5
- Let's reformulate the definition of a factorial to take advantage of this relationship.

Recursion

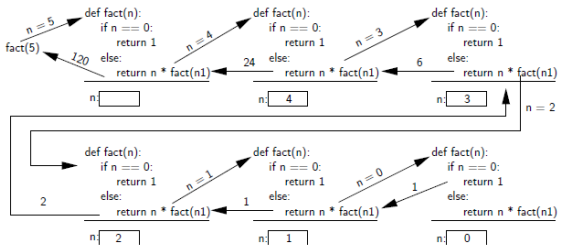
- A ***recursive function*** is a function that calls itself.
- Recursive functions are used to solve problems where the ***solution to a problem involves solving a smaller version of the same problem.***

Recursive Functions

- 2 parts:
  - Base Case: How to solve the smallest version of the problem that we care about.
  - Recursive Case: How to reduce the bigger version of the problem to a smaller version.
    - In order to work, the recursive case (when applied over and over) must eventually reduce every size of the problem down to the base case.
- What are these 2 parts for factorial?

How does recursion work?

- Recursion works (in all modern programming languages) because:
  - All variables are local
  - The stack makes new memory for local variables every time a function is called
  - Let's look at the stack when we call fact(5).



Why is recursion useful?

- Any loop (for/while) can be replaced with a recursive function that does the same thing.
  - Some languages don't include loops!
- Because we started with Python and are moving onto C++, we naturally see things in terms of loops.
- Some problems have a "naturally" recursive solution that is hard to solve with a loop.
- Other problems have solutions that work equally well recursively or with loops (iteratively).

How to "get" recursion

An "instance" of a problem is a single occurrence of that problem.

- Forget all loops
- To find the base case:
  - "What's the smallest version of this problem I would ever care about solving?"
- To find the recursive case:
  - "If I have an instance of the problem, how can I phrase how to solve the problem in terms of a smaller instance?"

Trust the recursion

- Base case is usually easy (When do I stop?)
- In recursive case:
  - Break the problem into two parts (not necessarily the same size):
    - A part I can solve "now".
    - The answer from a smaller instance of the problem.
  - *Assume the recursive call does the right thing.*
  - Figure out how to combine the two parts.

3 Laws of Recursion

- A recursive algorithm must have a **base case**.
- A recursive algorithm must change its state and move toward the base case.
- A recursive algorithm must call itself, recursively.

Another Example

Write a recursive function to compute the sum of a list of numbers.

Iterative Solution:

```
def listsum(numList):
    theSum = 0
    for i in numList:
        theSum = theSum + i
    return theSum
```

We could rewrite the list as a fully parenthesized expression
`((((1+3)+5)+7)+9)`

We can also parenthesize the expression the other way around,
`(1+(3+(5+(7+9))))`

Notice that the innermost set of parentheses, `(7+9)`, is a problem that we can solve without a loop or any special constructs.

Sum of a List of Numbers

```
total= (1+(3+(5+(7+9))))
total= (1+(3+(5+16)))
total= (1+(3+21))
total= (1+24)
total= 25
```

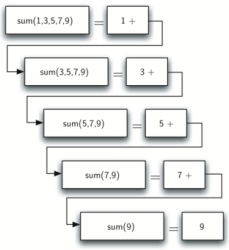
Let's restate the sum problem in terms of Python lists.

The sum of the list `numList` is the sum of the first element of the list (`numList[0]`), and the sum of the numbers in the rest of the list (`numList[1:])`.

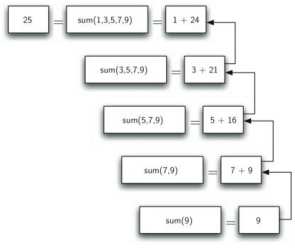
`listSum(numList)=first(numList)+listSum(rest(numList))`

So, what's our base case?

Sum of a List of Numbers



Series of Recursive Calls Adding a List of Numbers



Series of Recursive Returns from Adding a List of Numbers

Practice

- Write a recursive function to compute a^b (a raised to the b power) where a and b are both positive integers.
- To solve this recursively, solve the recursive case and the base case.
- Assume there are no built-in functions that will do this operation.
- Examples:
  - `power_to(4, 3) returns 64`
  - `power_to(3, 5) returns 243`