Tree Recursion

Announcements

Recursive Factorial

factorial (!)

if n == 0
n! = 1

if n > 0
n! = n x (n-1) x (n-2) x ... x 1

```
def factorial(n):          factorial(5)
    fact = 1
    i = 1
    while i <= n:          1   = 1*1
        fact *= i          2   = 2*1!
        i += 1             6   = 3*2!
    return fact            24  = 4*3!
                           120 = 5*4!
```

factorial (!)

if n == 0 — *base case*
n! = 1

if n > 0 — *recursive case*
n! = n x (n-1)!

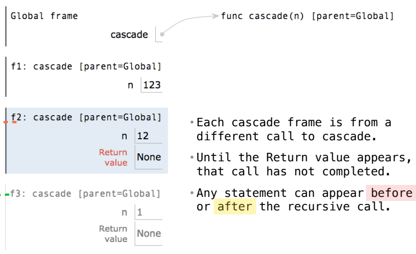
```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)


factorial(3)                     3 * factorial(2)
                                     2 * factorial(1)
                                         1 * factorial(0)
```

Order of Recursive Calls

The Cascade Function

(Demo)

```
def cascade(n):
    if n < 10:
        print(n)
    else:
        print(n)
        cascade(n//10)
        print(n)

cascade(123)
```



Program output:

```
123
12
1
12
```

- Each cascade frame is from a different call to cascade.
- Until the Return value appears, that call has not completed.
- Any statement can appear before or after the recursive call.

Two Definitions of Cascade

(Demo)

```
def cascade(n):
    if n < 10:
        print(n)
    else:
        print(n)
        cascade(n//10)
        print(n)
```

```
def cascade(n):
    print(n)
    if n >= 10:
        cascade(n//10)
        print(n)
```

- If two implementations are equally clear, then shorter is usually better
- In this case, the longer implementation is more clear (at least to me)
- When learning to write recursive functions, put the base cases first
- Both are recursive functions, even though only the first has typical structure

Example: Inverse Cascade

Inverse Cascade

Write a function that prints an inverse cascade:

```
1
12
123
1234
123
12
1
```

```
def inverse_cascade(n):
    grow(n)
    print(n)
    shrink(n)

def f_then_g(f, g, n):
    if n:
        f(n)
        g(n)

grow =   lambda n: f_then_g(                    )
shrink = lambda n: f_then_g(                    )
```

Tree Recursion

Tree Recursion

Tree-shaped processes arise whenever executing the body of a recursive function makes more than one recursive call

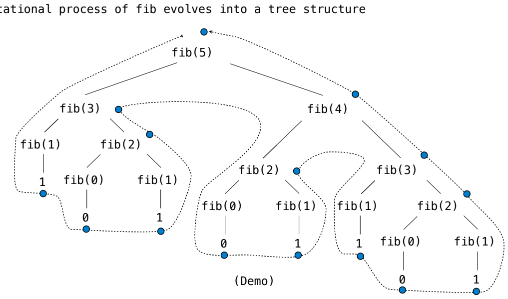
n:	0,	1,	2,	3,	4,	5,	6,	7,	8,	... ,	35
fib(n):	0,	1,	1,	2,	3,	5,	8,	13,	21,	... ,	9,227,465

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

http://en.wikipedia.org/wiki/File:Fibonacci.jpg

A Tree-Recursive Process

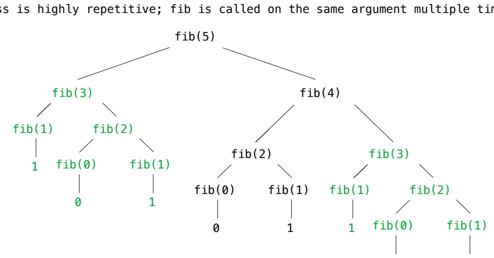
The computational process of fib evolves into a tree structure



(Demo)

Repetition in Tree-Recursive Computation

This process is highly repetitive; fib is called on the same argument multiple times

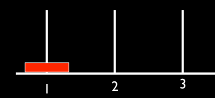


(We will speed up this computation dramatically in a few weeks by remembering results)

Example: Towers of Hanoi

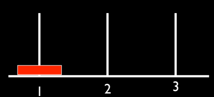
Towers of Hanoi

n = 1: move disk from post 1 to post 2



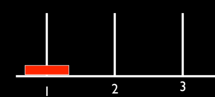
Towers of Hanoi

n = 1: move disk from post 1 to post 2



Towers of Hanoi

n = 1: move disk from post 1 to post 2

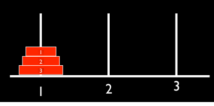



```
def move_disk(disk_number, from_peg, to_peg):
    print("Move disk " + str(disk_number) + " from peg " \
        + str(from_peg) + " to peg " + str(to_peg) + ".")


def solve_hanoi(n, start_peg, end_peg):
    if n == 1:
        move_disk(n, start_peg, end_peg)
    else:
```

```
def move_disk(disk_number, from_peg, to_peg):
    print("Move disk " + str(disk_number) + " from peg " \
        + str(from_peg) + " to peg " + str(to_peg) + ".")


def solve_hanoi(n, start_peg, end_peg):
    if n == 1:
        move_disk(n, start_peg, end_peg)
    else:
        spare_peg = 6 - start_peg - end_peg
        solve_hanoi(n - 1, start_peg, spare_peg)
        move_disk(n, start_peg, end_peg)
        solve_hanoi(n - 1, spare_peg, end_peg)
```

```
def solve_hanoi(n, start_peg, end_peg):
    if n == 1:
        move_disk(n, start_peg, end_peg)
    else:
        spare_peg = 6 - start_peg - end_peg
        solve_hanoi(n - 1, start_peg, spare_peg)
        move_disk(n, start_peg, end_peg)
        solve_hanoi(n - 1, spare_peg, end_peg)
```

```
hanoi(3,1,2)
```



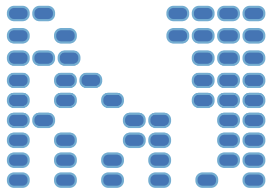
Example: Counting Partitions

Counting Partitions

The number of partitions of a positive integer n, using parts up to size m, is the number of ways in which n can be expressed as the sum of positive integer parts up to m in increasing order.

count_partitions(6, 4)

2 + 4 = 6
1 + 1 + 4 = 6
3 + 3 = 6
1 + 2 + 3 = 6
1 + 1 + 1 + 3 = 6
2 + 2 + 2 = 6
1 + 1 + 2 + 2 = 6
1 + 1 + 1 + 1 + 2 = 6
1 + 1 + 1 + 1 + 1 + 1 = 6

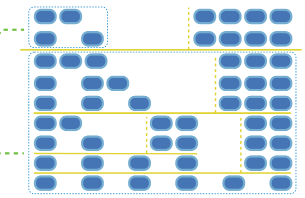


Counting Partitions

The number of partitions of a positive integer n, using parts up to size m, is the number of ways in which n can be expressed as the sum of positive integer parts up to m in non-decreasing order.

count_partitions(6, 4)

- Recursive decomposition: finding simpler instances of the problem.
- Explore two possibilities:
  - Use at least one 4
  - Don't use any 4
- Solve two simpler problems:
  - count_partitions(2, 4)
  - count_partitions(6, 3)
- Tree recursion often involves exploring different choices.



Counting Partitions

The number of partitions of a positive integer n, using parts up to size m, is the number of ways in which n can be expressed as the sum of positive integer parts up to m in increasing order.

- Recursive decomposition: finding simpler instances of the problem.
- Explore two possibilities:
  - Use at least one 4
  - Don't use any 4
- Solve two simpler problems:
  - count_partitions(2, 4)
  - count_partitions(6, 3)
- Tree recursion often involves exploring different choices.

```
def count_partitions(n, m):
    if n == 0:
        return 1
    elif n < 0:
        return 0
    elif m == 0:
        return 0
    else:
        with_m = count_partitions(n-m, m)
        without_m = count_partitions(n, m-1)
        return with_m + without_m
```

(Demo)