

Lecture9 tree recursion

recall

factorial

```
def factorial(n):
    fact = 1
    i = 1
    while i <= n:
        fact *= i # fact = fact * i
        i += 1    # i = i + 1
    retrun fact
```

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

Orders of Recursive Calls

Cascade Function

```
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)
```

Inverse Cascade

```
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(grow, print ,n // 10)
shrink = lambda n: f_then_g(print, f_then_g(print, shrink , n // 10)
```

well, i have to say that's surprising, cause this function solved this problem elegantly by utilizing the orders of the functions.

i have to say that this is a helpful way of thinking

Tree Recursion

Tree Recursion



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

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)
```



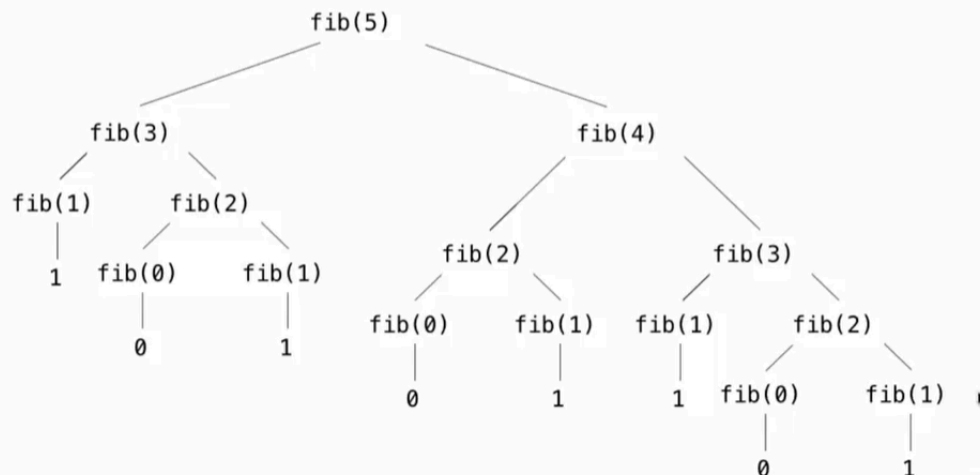
<http://en.wikipedia.org/wiki/File:Fibonacci.jpg>

10

A Tree-Recursive Process



The computational process of fib evolves into a tree structure

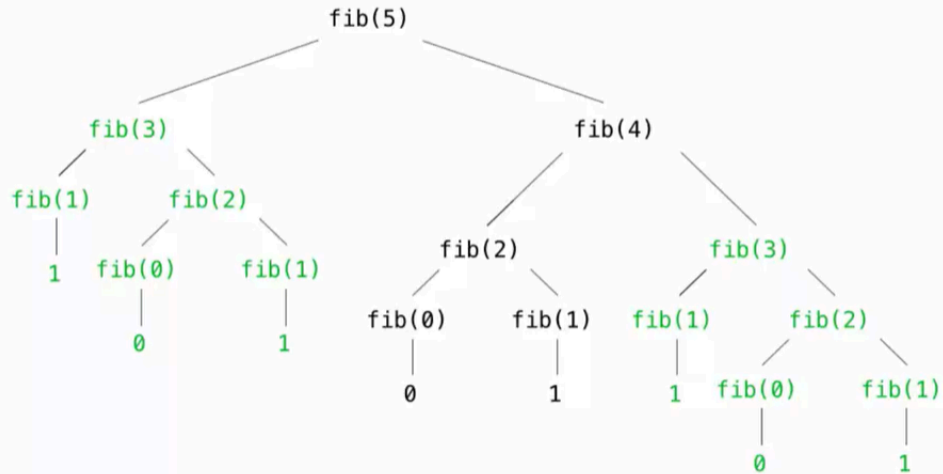


11

Repetition in Tree-Recursive Computation



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



Hanoi Tower

```
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)
```

The video shows a man in a dark suit and light blue shirt pointing towards the code on the left. The code defines two functions: move_disk, which prints the move of a disk from one peg to another, and solve_hanoi, which recursively solves the Hanoi Tower problem by moving n-1 disks to a spare peg, moving the nth disk to the target peg, and then moving the n-1 disks from the spare peg to the target peg.

Example: Counting Partitions

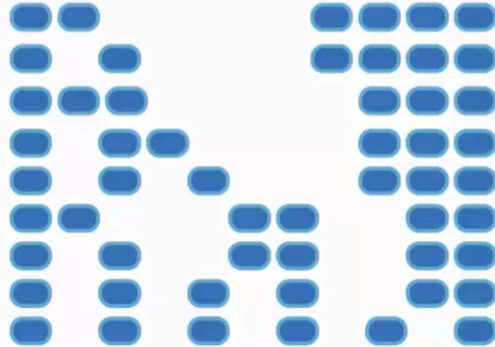
a good example for tree recursion.

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)`

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



14

strategies to deal with such problems

- 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

reflect on recursion homework

Q2. pingpong

```
def pingpong(n):  
    """Return the nth element of the ping-pong sequence.  
    >>> pingpong(8)  
    8  
    >>> pingpong(10)  
    6  
    >>> pingpong(15)  
    1  
    >>> pingpong(21)  
    -1
```

```

>>> pingpong(22)
-2
>>> pingpong(30)
-2
>>> pingpong(68)
0
>>> pingpong(69)
-1
>>> pingpong(80)
0
>>> pingpong(81)
1
>>> pingpong(82)
0
>>> pingpong(100)
-6
>>> from construct_check import check
>>> # ban assignment statements
>>> check(HW_SOURCE_FILE, 'pingpong', ['Assign', 'AugAssign'])
True
"""
"""
"""*** YOUR CODE HERE ***"""
def helper(val, dir, idx):
    if idx > n: return 0
    if num_eights(idx - 1) > 0 or (idx - 1) % 8 == 0: return helper(val, -
dir, idx + 1) - dir
    return helper(val, dir, idx + 1) + dir
return helper(1, -1, 1)

```

why recalling on this

we know that recursion use parameters as the record of the status, however, what should we do if we only have a single parameter but several status to be passed?

the answer is, use a helper function, which is naturally a great state machine

moreover, this recursive function, infact, this is logically equivalent to iteration; the difference is purely in form.

another good example from hw02

```

def count_coins(total):
    """Return the number of ways to make change for total using coins of value
    of 1, 5, 10, 25.
    >>> count_coins(15)
    6
    >>> count_coins(10)
    4

```

```

>>> count_coins(20)
9
>>> count_coins(100) # How many ways to make change for a dollar?
242
>>> from construct_check import check
>>> # ban iteration
>>> check(HW_SOURCE_FILE, 'count_coins', ['While', 'For'])

True
"""
"*** YOUR CODE HERE ***"
def helper(total, coin):
    if total <= 0: return 0
    elif total == coin: return 1
    elif coin == 25: return helper(total - coin, coin)
    return helper(total - coin, coin) + helper(total,
next_largest_coin(coin)
    return helper(total, 1)

```

```

def make_anonymous_factorial():
    return (lambda n:
        (lambda f: f(f))
        (lambda f: lambda n: 1 if n == 1 else mul(n, f(f)(sub(n, 1))))
        (n)
    )

```

this one is impressive too, though cs61a don't require us to handle this but it is still helpful for us to understand what happen's here

the reason why i didn't solve this at the first time is that i was struggling about how to call a function in its body, but in stead of considering creating a function that calls itself. a solution to this might be thinking of things step by step, and consider them with their functions.