CS61A Lecture 10

Thursday, September 19th, 2019

Examples today, especially the last one of the day, will be examples that are only doable with recursion.

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

With the above example, if we call `cascade` on 123, it will print:

```
123
12
1
12
123
```

`cascade` is first called on 123, prints 123, then this calls `cascade` on 12, prints 12, then calls `cascade` on 1. We then go back to the previous call, which prints 12 again, then 123 again.

> **What if cascade returned instead of printed?**
>
> ```
> def cascade(n):
>     if n < 10:
>         print(n)
>     else:
>         print(n)
>         cascade(n//10)
>         return n
> ```
>
> This would only print:
>
> ```
> 1
> 2
> 3
> 4
> 5
> ```

We don't have to write the above example in such an obviously recursive way:

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

Which one is better? Well, this second implementation is shorter yes, but the first one is more obvious, and as such, is better for readability.

Inverse Cascade

Write a function that prints an inverse cascade:

```
1
12
123
12
1
```

Template given:

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

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

Write the codes for grow and shrink:

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

> **Recursion vs. Self-Reference**
> Recursion involves calling the function on some smaller version on the same problem. Self-Reference just calls the function defined within itself.

Tree Recursion

Tree recursion occurs when a function calls itself twice.

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

We can do this with our friend Fibonacci:

```
def fib(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
```

```
    else:
        return fib(n-2)+fib(n-1)
```

The only numbers this function ever returns are 1s and 0s. You can imagine it's quite slow! Fibonacci numbers get pretty big: the 35th is 9 million+.

You can trace the order in which things operate:

```
fib(5)
```

first calls `fib(3)`, then `fib(1)`, which returns 1. `fib(3)` before calling `fib(2)`, which calls `fib(0)` and `fib(1)`.

> **Recursion depth error**
> This does not stop you from computing very large recursion problems. It only stops you from computing problems that never end. It is possible to increase the maximum number before Python errors (the default is 1000), but you don't need to know it.

> **ucb.py**
> Every time there is a project, you get a file `ucb.py`, which includes a `@trace` decorator. This `@trace` decorator is nicer in the sense that it indents stuff, so that you can see an even nicer breakdown of what stuff gets returned where.

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 the positive integer parts up to `m` in increasing order.

```
count_partitions(6,4)
```

1. 2+4=6
2. 1+1+4=6
3. 3+3=6
4. 1+2+3=6
5. 1+1+1+3=6
6. 2+2+2=6
7. 1+1+2+2=6
8. 1+1+1+1+2=6
9. 1+1+1+1+1+1=6

The above should return 9, because there are 9 ways to split 6 into partitions up to size 4.

How can we do this?

We need to figure out how to split up the list, so we can count the number of rows in one group, and then the other group, ensuring there is no overlap between them also while making sure there are no missing scenarios.

Recursive decomposition is finding simpler instances of the same problem.

In the example case, explore two possibilities:

- Use at least one 4.
- Use no 4s.

In this case, our group of 9 is divided into 2 and 7. In this case, we are really solving two simpler problems:

- count_partitions(2,4)
- count_partitions(6,3)

How do we know this works? We make a recurisve leap of faith and assume that the two smaller functions really do what they're supposed to do.

Tree recursion often involves exploring different choices.

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

Base cases don't always come just from what's the simplest scenario. Base cases come from trying to figure out where your recursive calls will lead to. In the above case, we first write the `else` clause, then try to figure out the base case from there.

```
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
```

> **Why don't we use assert statements on `n<0` ?**
> If we use assert statements, we need to make doubly sure our recursive calls never reach this situation. This is easier than it seems, for example, `count_partitions(6,4)` calls `count_partitions(2,4)` , which itself calls `count_partitions(-2,4)` .
>
> Simply writing them into the if statements makes your life a little easier because you just need to handle the bogus cases manually.

You **must** think about this in terms of abstraction. If you were to call `count_partitions(5,3)` and draw an environment diagram, there are 234 separate steps! There's no way you'd be able to do this by hand, especially in an exam.