CS61A Lecture 9

Wednesday, September 18th, 2019

Recursion

A recursive function is a function whose body calls the function itself, either directly or indirectly, where the implication is that executing the body of a recursive function may require applying that function.

Digit Sums

How does this problem show up in computing?

Let's say we want to sum the digits in the number 2019. Why would it be useful? For example, a number is divisible by 9 when the sum of its digits is divisible by 9. Credit cards also use this property for typo detection. The last digit of a card is actually just a checksum, so algorithms can tell if you mistyped your credit card number without having to check against a database.

The Problem within the Problem

Let's take the number 6. The sum of its digits its 6. Likewise, for any one-digit positive number, it is the number itself. The sum of the digits of 2019 is 9 plus the sum of the digits of 201, which is the sum of the digits of 20 and 1, and so on and so forth.

We can define a recursive function as such:

```
def split(n):
    return n//10, n□

def sum_digits(n):
    if n<10:
        return n
    else:
        all_but_last, last = split(n)
        return sum_digits(all_but_last)+last
```

An if statement is almost always somewhere within its body. This is because if you called a function within itself all the time, then it would just be an infinite loop.

Conditional statements check for base cases. A base case is an argument to the function that you can pass in without having recursive calls. In the above, the base case is `n<10`.

The rest is what we call recursive cases. Why do recursive calls work within the body of a function? The reason is when you define a function, it isn't executed yet. Functions are executed only when they are called, so by the time they are called within the function body, it already has a name and a body to be executed.

Recursion in Environment Diagrams

Let’s see how recursion works with a new problem, the factorial of n.

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

This recursive definition, when fact is called on 3, means that 4 separate frames are opened up, 1 each for `n=3`, `n=2`, `n=1` and `n=0`. The return value is first left blank for `n=3` until `=n=1` until the code finds the base case, gets a return value, and then progressively multiplies that return value with `n` to get the final value of `fact(3)`.

We must make a path to the base case in every recursive function. What distinguishes a base case from a non-base case? It’s unique to every problem, and it’s up to you to properly define base cases.

Sometimes, the order of the base cases may matter.

Iteration vs. Recursion

Iteration is a special case of recursion. For example, factorial with iteration:

```
def fact_iter(n):
    total, k = 1, 1
    while k <= n:
        total, k = totak*k, k+1
    return total
```

and factorial using recursion:

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

In Python, recursion runs a little slower than iteration, but it would not in most programming languages.

Verifying Recursive Functions

How do you know any function you write works? Using environment diagrams.

But it might be a little simpler. Instead, we can use a concept similar to the mathematical concept of induction, but here we will just call it the recursive leap of faith.

- Use the following steps:

1. Verify the base case.
2. Treat `fact` as a functional abstraction!
3. Assume `fact(n-1)` is correct.
4. Verify that `fact(n)` is correct.

- Basically assume your function works for simpler calls of the same function, and then asking how you can build on that to work on more complicated cases.

Mutual Recursion

An example would be a function `f` that calls `g`, and a function `g` that calls `f`.

The Luhn Algorithm

- Used to verify credit card numbers.
- It's quite long to explain, so just check Wikipedia.
- Let's implement:

```
def luhn_sum(n):
    if n>10:
        return n
    else:
        all_but_last, last = split(n)
        return luhn_sum_double(all_but_last) + last

def luhn_sum_double(n);
    all_but_last, last = split(n)
    luhn_digit = sum_digits(2*last)
    if n < 10:
        return luhn_digit
    else:
        return luhn_sum(all_but_last)+luhn_digit
```

- We have a function `luhn_sum` calling `luhn_sum_double`, and a function `luhn_sum_double` calling `luhn_sum`. This works for the same reason basic recursion works.

Converting Iteration to Recursion

More formulaic: Iteration is a special case of recursion.

In iteration, we use assignment statements. In recursion, we do the assignment by calling the function recursively.

Look for the names in iteration and use them as the argument names for the recursive call, and look for the while loop and convert it to the base case. With these basic steps, you should have done a good chunk of the conversion work!