Recursion

15-110: Principles of Computing

Giselle Reis

Sometimes when solving a problem we come across situations where we perform some simple operations and end up with the same problem again, although a bit smaller. We have already seen problems like this: take palindromes, for example. Palindromes are strings that read the same way back and forth. To find out if a string is a palindrome, we need to check if the first and last characters are the same, and then check if the inner part is also a palindrome. This second part consists on solving *the same problem again*, but on a smaller string.

This solution can be translated straightforwardly into code:

```
def isPalindrome(s):
    return s[0] == s[-1] and isPalindrome(s[1:-1])
```

Notice how the definition of the function calls itself. This is one of the fingerprints of recursion! We say that this `isPalindrome` function is recursive.

If we trace this code for a simple input (e.g, `abba`), we will find ourselves in a never ending loop. This is because the slice `[1:-1]` of an empty string is the empty string, and the function will call itself over and over again. To prevent that, we usually need a base case for the recursion: a case for the input that will not result on another recursive call.

Once we reach the empty string, the palindrome function can stop. Since the empty string is a palindrome, we can return `True` in this case:

```
def isPalindrome(s):
    if s == '':
      return True
    else:
      return s[0] == s[-1] and isPalindrome(s[1:-1])
```

Recursion is a very powerful programming technique[1]. Generally recursive solutions are concise and elegant, and many times very natural. This is an important tool to have in your problem solving skills.

[1] So powerful that we could get rid of all loops and we would still be able to solve the same problems using only recursion!

Rabbits

Remember the rabbit breeding problem?

You have decided to get a pair of rabbits as pets. Since you want them to reproduce, you bought one female and one male rabbit. It turns out that this is a very special kind of rabbit that reproduces with amazing regularity. Their reproduction follows the rules:

1. A rabbit takes one month to reach the adult age.
2. A pair of adult rabbits reproduces once a month (conveniently) generating another pair of rabbits.
3. The rabbits are immortal. (Trust us. We've tried.)

Your job was to write a function that computes the number of rabbits on a given month n. Let's put our recursive hats on and try to think of this problem again. The rabbit pairs that we have in one month are all the pairs that we had on the previous month, because they are immortal. Additionally, all pairs that we had up to two months ago have reproduced. Putting this information together, we have that, at month n, the number of pairs of rabbits can be obtained by the following equation:

$$R(n) = R(n-1) + R(n-2)$$

But how do we stop? Well, on the first and second months we have one pair of rabbits, so these are the base case. The complete solution thus is:

$$\begin{aligned} R(1) &= 1 \\ R(2) &= 1 \\ R(n) &= R(n-1) + R(n-2) \end{aligned}$$

Translating this to python gives us the following recursive function:

```
def rabbits(n):
    if n == 1 or n == 2:
        return 1
    else:
        return rabbits(n-1) + rabbits(n-2)
```

Towers of Hanoi

The game *towers of Hanoi* consists on moving disks one at a time from one tower to another, using only one extra spare tower and never placing a bigger disk on top of a smaller one. We want to write a program for solving this problem for us. Before we do that, let's spend more time looking at the problem and the solutions we can find for different number of disks.

After playing with the towers for a while, we realize that the pattern is:

1. move the upper disks to the spare tower
2. move the biggest disk to the target tower
3. move the disks on the spare tower to the target one

Since we cannot move many disks at once, steps 1 and 3 will actually be composed of many mini-steps. But our recursive solution does not need to care about it! It will simply recurse on a different number of disks with different origin and target towers. The base case is when we have only one disk, when we simply move if from the origin to the target.

```
def hanoi(n, origin, target, spare):
    if n == 1:
        print("Move disk from", origin, "to", target)
    else:
        hanoi(n-1, origin, spare, target)
        print("Move disk from", origin, "to", target)
        hanoi(n-1, spare, target, origin)
```

Recursion in nature

Recursion happens in nature in the form of *fractals*. Fractals are patterns that repeat themselves as you zoom in or out. Take for example a snowflake, a fern leaf, or a Romanesco broccoli:

In fact, the following fern leaf was generated using a recursive function that computes new coordinates x, y as a function of the previous coordinates x, y. Note how there are mini-leaves inside the leaf. This goes on and on in the picture, until you get to the smallest component: one pixel.