Recursion

Recursion is the most important technique for designing algorithms and data structures. In general, recursion means to define something in terms of itself. For instance, we can define a (rooted) *tree* as follows: a tree consists of a root node, which is connected to the roots of zero or more trees. The power of recursion here lies in the possibility of defining an infinite set of objects by a finite statement.

In computer science, recursion means a function that calls itself (directly, or indirectly). When you first meet recursion, it may seem like a form of magic. Once you get used to it, it becomes a powerful and actually quite natural technique for solving problems.

Humans solve difficult problems by dividing them into easier subproblems, and then solve these subproblems. A good manager, for instance, takes a task, splits them up into smaller tasks, and asks his team members to handle these tasks. In programming, this means we have a main function that takes the problem, splits it up into smaller problems, and calls a subroutine for each of the subproblems.

Sometimes it turns out that the "easier subproblems" are actually the same problem we originally started with, but on a smaller or easier input. In this case, there is no separate function for the subproblem, and instead our main function calls itself *recursively* to solve the subproblem.

In our manager analogy, the manager delegates the smaller subproblems to *himself.* If you find this self-reference confusing, its helpful to imagine that someone else—some team member—is going to solve the simpler problems, just as in the non-recursive case.

Jeff Erickson invented the name *Recursion Fairy* for this "someone else." To quote him:

> "Your only task is to simplify the original problem, or to solve it directly when simplification is either unnecessary or impossible; the Recursion Fairy will magically take care of all the simpler subproblems for you, using Methods That Are None Of Your Business So Butt Out. Mathematically sophisticated readers might recognize the Recursion Fairy by its more formal name, the Induction Hypothesis."

We will loosely define recursion like this:

- If the problem is small or simple enough, solve it directly (*base case*).
- Otherwise, divide the problem into one or more simpler instances of the same problem, solve these recursively, and combine the solutions.

Printing a number in any base

Let's start with a simple example: How do you find the representation of the number 83790 in base 8? More generally, how do you find the base-b representation of a given number n?

It is easy to find the *last digit* of this representation: it is $n \bmod b$ (`n % b` in Python). The remaining digits are then the base-b representation of $\lfloor n/b \rfloor$ (that is, `n // b` in Python).

The following method implements this recursive strategy:

```
DIGITS = "0123456789abcdef"

def print_rec(n, base):
  if n >= base:
    print_rec(n // base, base)
  digit = n % base
  print(DIGITS[digit], end="")
```

Note that there is a base case: If the number is less than b, we do not make a recursive call and actually print the number n directly.

A common mistake students make when designing a recursive function is to try to mentally simulate how the program is executed. Instead, you must *trust the recursion fairy.* You should reason about your method by assuming that the recursive call works correctly, and then argue that the method itself will do the right thing.

In other words, your *only* task is to *simplify* the original problem, or to solve it directly when simplification is either unnecessary or impossible. The *Recursion Fairy* will magically take care of the simpler subproblems.

In mathematics, recursion appears all the time, in definitions such as the definition of trees above (or, for instance, in the following definition of the natural numbers $\mathbb{N}$: 0 *is a natural number; if* n *is a natural number, then* $n+1$ *is a natural number*).

Mathematical induction is a form of recursion, and indeed, when we want to prove that a recursive method works correctly, we use a proof by induction. For example, to prove that our method `print_rec` is correct, we would use induction on k, where k is the number of digits of n when written in base b (in other words, $k = \lceil \log_b(n+1) \rceil$.

- Base case $k = 1$: If n has only one digit, then $0 \leqslant n < b$, and so we are in the base case of `print_rec`, which prints one digit correctly.
- Inductive step $k > 1$: We assume that `print_rec` works correctly for numbers with less than k digits. Consider now a number n with k digits in base b. Then $\lfloor n/b \rfloor$ has $k-1$ digits. By the inductive assumption, this means that the recursive call `print_rec(n // b)` correctly prints $\lfloor n/b \rfloor$. The method then prints the last digit, and therefore the number n has been printed correctly.

In a recursive algorithm, there must be no infinite sequence of reductions to simpler and simpler subproblems. Eventually, the recursive reductions must stop with an elementary base case that can be solved by some other method; otherwise, the recursive algorithm will never terminate.

This means that, first, we have to remember to include a base case that requires no recursive call. Second, the recursive calls must in some sense be "easier" then the original call, so that progress is made and the base case is eventually reached. In other words, the recursion fairy is your magic helper, but she will not do all your work for you by herself. If you hand her the full version of the problem without making it smaller or simpler first, she will refuse to work her magic.

Towers of Hanoi

We again quote directly from Jeff Erickson's lecture notes on recursion:[1] The Towers of Hanoi puzzle was first published by the mathematician François Éduoard Anatole Lucas in 1883, under the pseudonym "N. Claus (de Siam)" (an anagram of "Lucas dAmiens"). The following year, Henri de Parville described the puzzle with the following remarkable story:

> In the great temple at Benares beneath the dome which marks the centre of the world, rests a brass plate in which are fixed three diamond needles, each a cubit high and as thick as the body of a bee. On one of these needles, at the creation, God placed sixty-four discs of pure gold, the largest disc resting on the brass plate, and the others getting smaller and smaller up to the top one. This is the Tower of Bramah. Day and night unceasingly the priests transfer the discs from one diamond needle to another according to the fixed and immutable laws of Bramah, which require that the priest on duty must not move more than one disc at a time and that he must place this disc on a needle so that there is no smaller disc below it. When the sixty-four discs shall have been thus transferred from the needle on which at the creation God placed them to one of the other needles, tower, temple, and Brahmins alike will crumble into dust, and with a thunderclap the world will vanish.

Of course, being good computer scientists, we read this story and immediately substitute n for the hardwired constant sixty-four. How can we move a tower of n disks from one needle to another, using a third needles as an occasional placeholder, never placing any disk on top of a smaller disk? The trick to solving this puzzle is to think recursively. Instead of trying to solve the entire puzzle all at once, let's concentrate on moving just the largest disk. We can't move it at the beginning, because all the other disks are covering it; we have to move those $n-1$ disks to the third needle before we can move the nth disk. And then after we move the nth disk, we have to move those $n-1$ disks back on top of it. So now all we have to figure out is how to...

[1] `http://web.engr.illinois.edu/~jeffe/teaching/algorithms/`

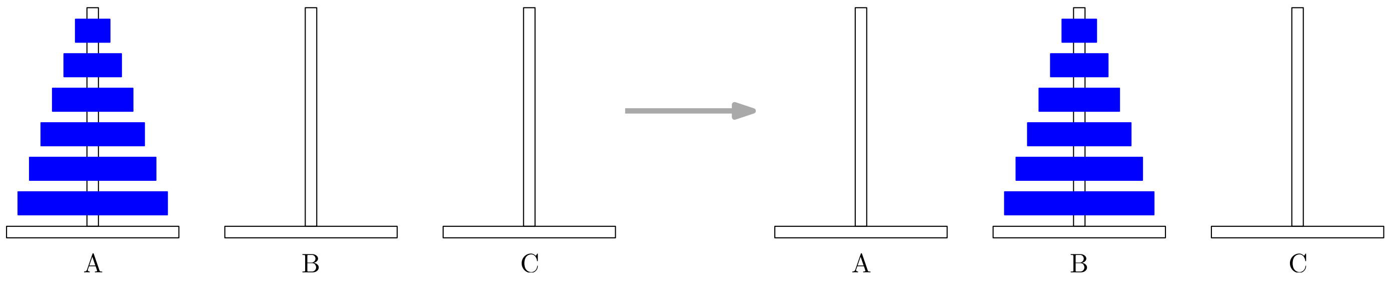

Figure 1: Towers of Hanoi

STOP!! That's it! We're done! We've successfully solved the n-disk Tower of Hanoi problem by solving two instances of the $(n-1)$-disk Tower of Hanoi problem, which we can gleefully hand off to the Recursion Fairy (or, to carry the original story further, to the junior monks at the temple).

Our algorithm does make one subtle but important assumption: *There is a largest disk.* In other words, our recursive algorithm works for any $n \geqslant 1$, but it breaks down when $n = 0$. We must handle that base case directly. Fortunately, the monks at Benares are quite adept at moving zero disks from one needle to another in no time at all, and we arrive at the following code:

```
def solveHanoi(n, source, destination, spare):
  if n == 1:
    print("Move disc 1 from %s to %s" % (source, destination))
  else:
    solveHanoi(n-1, source, spare, destination)
    print("Move disc %d from %s to %s" % (n, source, destination))
    solveHanoi(n-1, spare, destination, source)


solveHanoi(n, 'A', 'B', 'C')
```

It is tempting to think about how all those smaller disks get moved—in other words, what happens when the recursion is unfolded—but it's not necessary. In fact, for more complicated problems, unfolding the recursive calls is merely *distracting*. Our only task is to reduce the problem to one or more simpler instances, or to solve the problem directly if such a reduction is impossible. Our algorithm is trivially correct when $n = 0$. For any $n \geqslant 1$, the Recursion Fairy correctly moves the top $n-1$ disks, so our algorithm is clearly correct.

How many moves are needed to solve the problem for n disks? We can quickly check the answer for $n = 0, 1, 2, 3$, and obtain $0, 1, 3, 7$ moves. We therefore guess that the number of moves for n disks must be $2^n - 1$, and prove this claim by mathematical induction:

- Base case ($n = 0$ disks): The number of moves is $0 = 2^0 - 1$, and so the claim is true.
- The inductive step: Let $n > 0$, and assume (*Induction hypothesis*) that the number of moves for $n-1$ disks is $2^{n-1} - 1$. To move n disks, we first have to move the $n-1$ top disks from `A` to `C`. By the induction hypothesis, this takes $2^{n-1} - 1$ moves. Then we have to move the largest disk (1 move), and finally we have to move the $n-1$ top disks from `C` to `B`. This takes again $2^{n-1} - 1$ moves by the induction hypothesis. So the total number of moves is

$$(2^{n-1} - 1) + 1 + (2^{n-1} - 1) = 2^n - 1,$$

  and we have proven the claim for n disks.

Fibonacci numbers

The Fibonacci numbers F_n are defined as follows:

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_n &= F_{n-1} + F_{n-2} \qquad \text{for } n > 1 \end{aligned}$$

This immediately gives us a recursive method for computing Fibonacci numbers:

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

It turns out that this method becomes very slow when $n \approx 35$. To see why, we draw a tree that shows all the calls and recursive calls to the function `fib`—see Fig. 2. As we can see in the tree, computing `fib(40)`

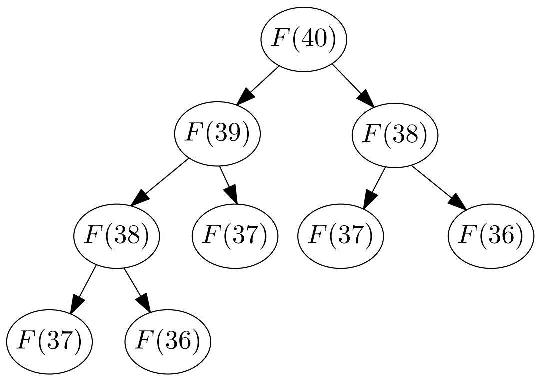

Figure 2: Recursive calls for computing `fib(40)`

means that we compute F_{38} two times, F_{37} three times, F_{36} five times—and so on, so that F_{41-k} is computed F_k times. Since the Fibonacci numbers grow exponentially, this procedure quickly becomes too slow. The problem in this example is that the recursive routine performs *redundant* calculations, that is, calculations that have already been done before. This example shows that recursion—blindly applied—is not always appropriate.

The solution is to either use an array to store all previous Fibonacci numbers (so no recursion is necessary), or to return both F_n and F_{n-1} from the function `fib(n)` (see example code *fibonacci2.py* and *fibonacci3.py*).