Recursion

To understand recursion

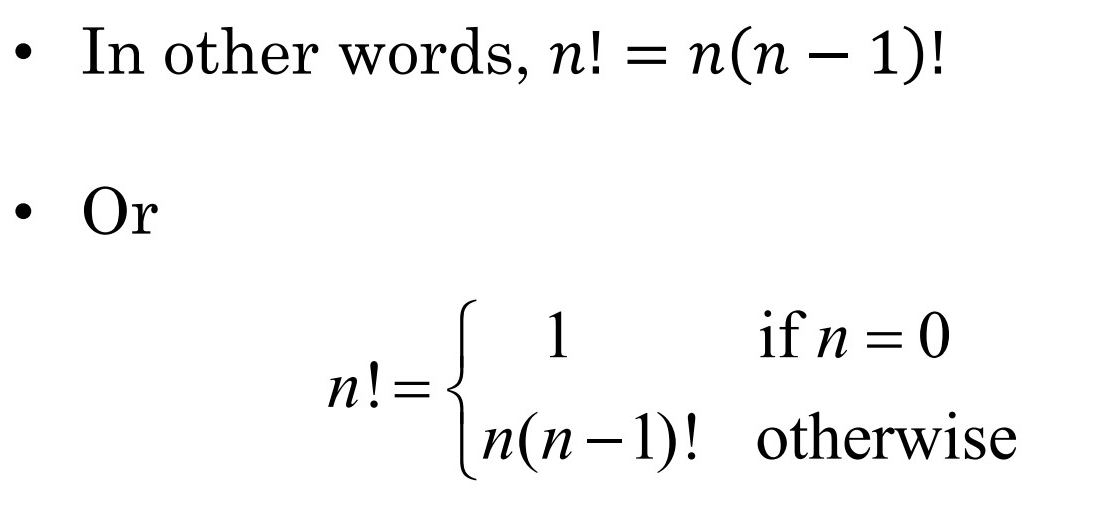
To understand when to use recursion

Recursion vs iteration

Recursive definitions

A description of something that refers to itself is called recursive definition

It is used in the factorial



The definition says that 0! = 1, while the factorial of any other number is that number times the factorial of one less than that number.

0! = 1 is **the base case of the recursion ( a very small version of the original problem that can be solved without recursion becomes the base case)**

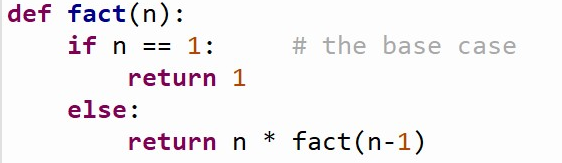
When the base case is encountered, we get a closed expression that can be directly computed.

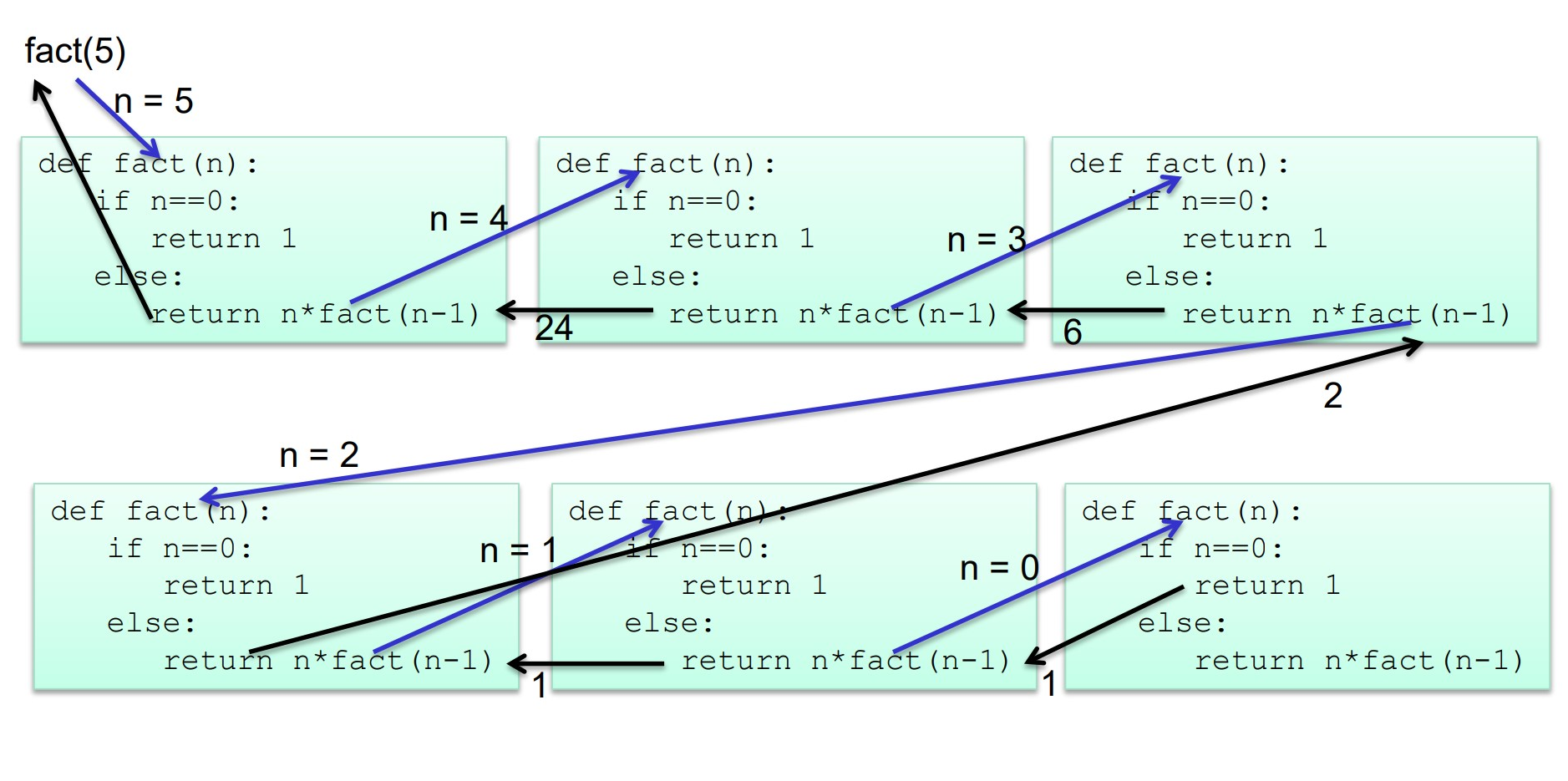
All good recursive definitions have these two key characteristics:

1. There are one or more bases case for which no recursion applied.
2. All chains of recursion eventually end up at one of the base cases

Put differently, each iteration must drive the computation toward a base case.

Recursive functions







先走到最后然后再往回走

(the first n and the second n, …) they are independent, whenever this function is called, it would create a new n

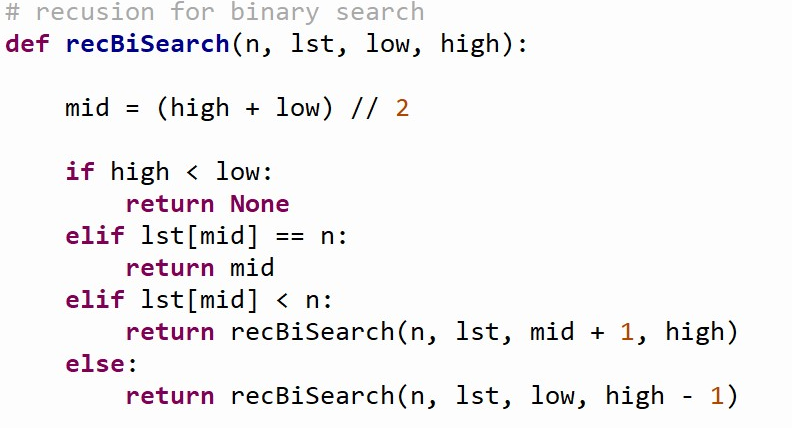
The function is waiting for other functions return the result to it. Once it returns the small version result, it would be deleted by the system.

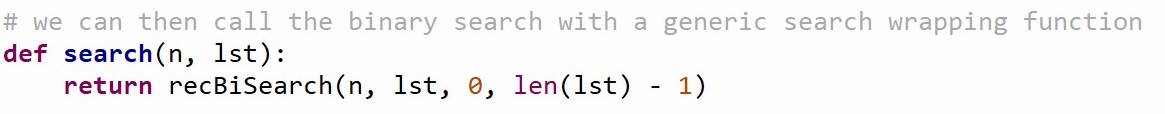
Example: binary search

There are two base cases (to stop recursion / searching)

1. When the target value is found
2. When we have run out of places to look

You need to consider all of them.

****

****

Recursion vs. iteration

In fact, anything that can be done with a loop can be done with a simple recursive function (but some algorithms hard to set up with iteration)

And some programming languages use recursion exclusively.

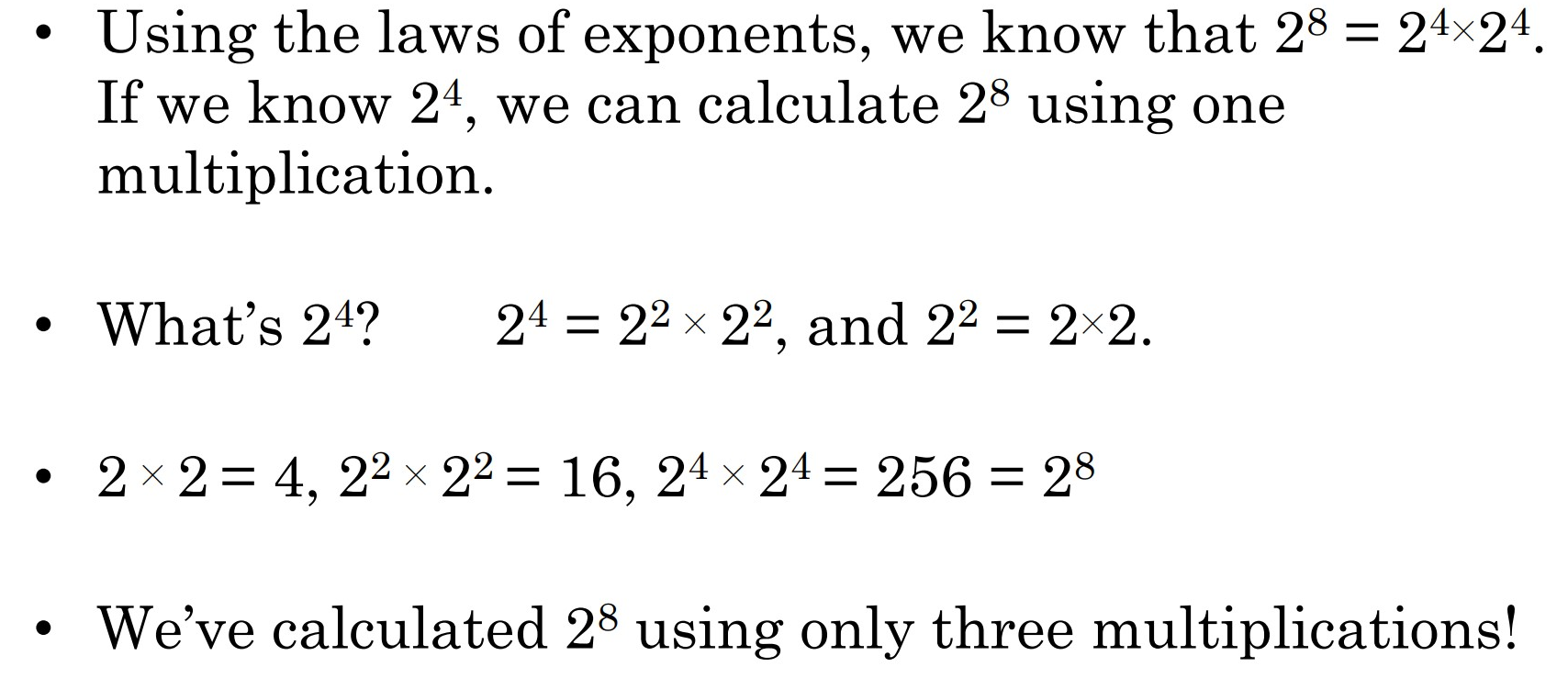
Some problems that are simple to solve with recursion are quite difficult to solve with loops

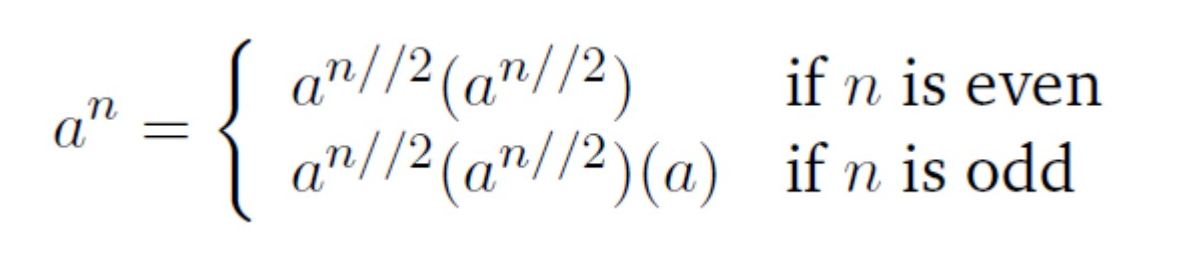
Example 1 to prove their difference: fast exponentiation

Where recursion is far much better than looping

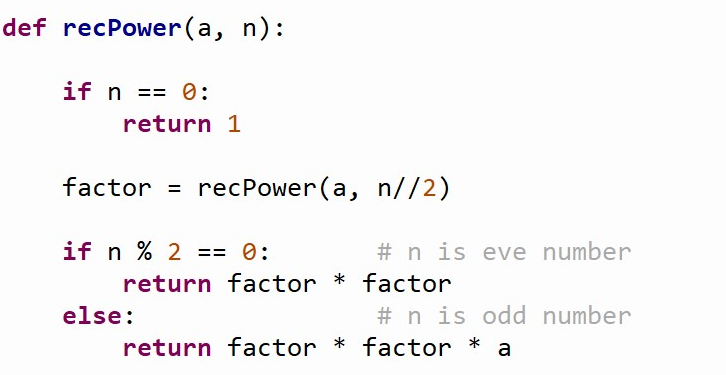
One way to compute for an integer n is to multiply a by itself n times

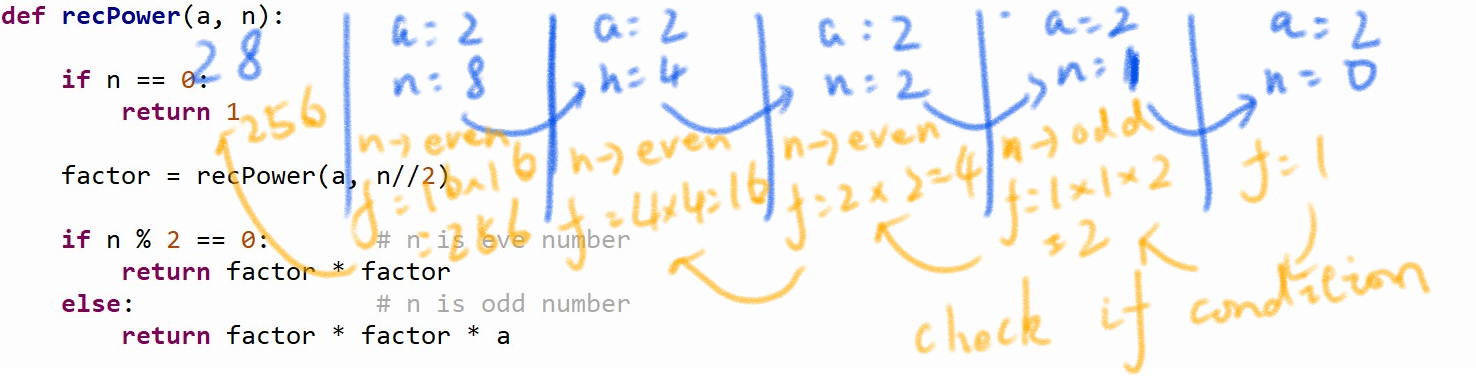
We can also solve this problem suing recursion and divide & conquer approach





If we keep using smaller and smaller values for n, n will eventually be equal to 0 (1//2 = 0), and a\*\*0 for nay value except a = 0

****

****

**Here, a temporary variable called factor is introduced so that we don’t need to calculate a\*\*(n//2) more than once, simply for efficiency.**

**The reason why I don’t use n == 1 to stop it is when n == 1, 1 % 2 = 1, so there is factor \* factor \* a, a is the most important thing.**

**Without a, you just times 1 again ang again and never get the right answer.**

**Recursion is better in this time**

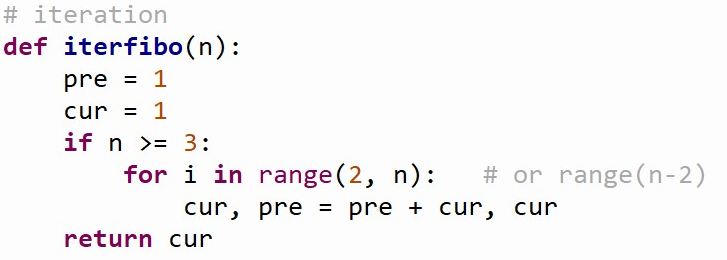
Example 2

This time iteration is better than the recursion

The Fibonacci sequence (1,1,2,3,5,8…)

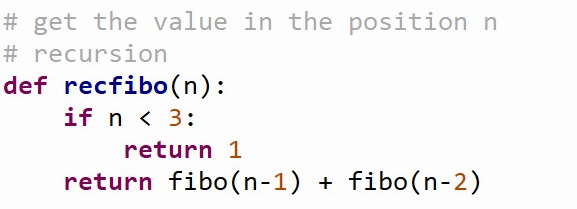
Successive number are calculated by finding the sum of the previous two numbers

Iteration

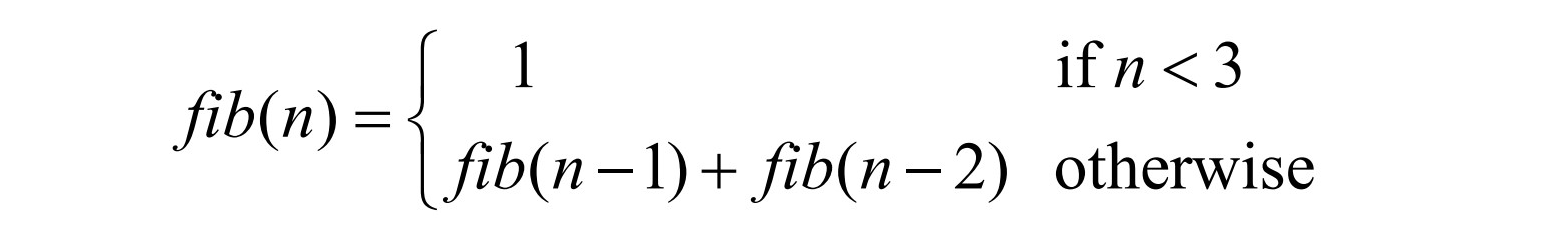


The loop executes only n-2 times since the first two values have already been provided as a starting point.

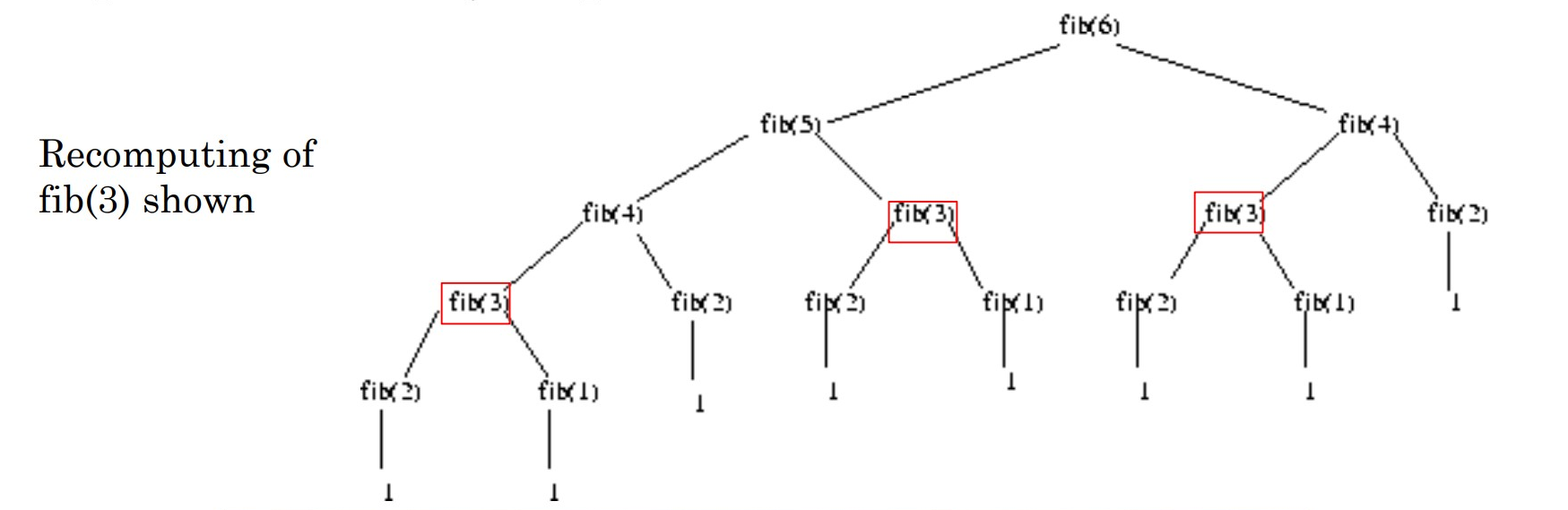
Recursion



The recursive def:



For this time, iteration is better



There are just some repeated calculations

Conclusion

Sometimes recursion provides a good solution because it is more elegant or efficient than a looping version

At other times, when both algorithms are quite similar, the edge goes to the looping solution on the basis of speed and simplicity of programming.

Avoid the recursive solution if it is terribly inefficient, unless you can’t come up with an iterative solution.