Recursion:

- Functions that call themselves

- Works very well for situations where a function does a calculation for ***n*** that directly depends on the result of the function on ***n-1.***
  - **f(n)** = results from operation(s) on the value of **f(n-1)**

- **Factorial** is a classic example.
  - I tell you the factorial of 6 is 720: **f(6) = 720.**
  - I ask what is the factorial of 7: **f(7) ?**
    - **f(7) = 7 * f(6) = 7 * 720 = 5,040**
    - **f(n) = n * f(n-1)**

Compounding interest is another classic example.

- I tell you that I have a bank account that has a monthly interest rate of **0.003**. After 6 months I have **$1,018.14** in the account.
- I ask what will I have after 7 months: **f(7) ?**
  - **f(7) = 1,018.14 + (1,018.14 * 0.003)**
  - **f(7) = f(6) + (f(6) * 0.003)**
  - **f(n) = f(n-1) + (f(n-1) * rate)**

Sometimes in math, a recursive formula is defined in terms of f(n+1).

Verhulst formula for predicting next year's population from this year's:

$$p(n+1) = (1+g-h) * p(n) - g*p(n)^2 / M$$

- The population next year is based on the population this year.

p(n) is the population at time n

g is the growth rate

h is the loss rate

M cap or carrying capacity

Recursion doesn't have to be mathematical.

- **COUNTING**: Countdown from n to 1 then output "Blastoff!"

```
countdown(n)
if  n == 0 output "Blastoff!"
else  output n and call countdown( n-1 )
```

- **SORTING**: I tell you that I have an array in which first 6 elements are sorted and that they are the smallest 6 elements in the array.
- I ask what will make the first 7 elements sorted?
  - **f(7) = f(6) + the smallest of the remaining elements**

Recursion doesn't have to be mathematical. Another "famous" example is **binary search**. **Remember phone books?**

- Assume I have a sorted array and I'm searching for a target element **targ.**
- **SEARCH (from lowIDX, to highIDX):**
  - Look at the middle (**median**) element between

    lowIDX and highIDX
  - If median element = = target then found ! END.
  - If **targ < median,**

    SEARCH (from index lowIDX, medianIDX-1)
    // bottom half
  - **else**

    SEARCH (from medianIDX+1, highindex)
    // top half

Recursion doesn't have to be on just n-1.

The butterfly population grows in my garden very rapidly (there are happy butterflies in my neighborhood!). Every week the population is the same as last week's population plus the population the week before.

- I tell you that it's the 7th week of spring. The population in my garden last week was 8 butterflies, and the week before there were 5 butterflies
- I ask how many butterflies are in the garden now?
  - f(7) = f(6) + f(5)
  - f(7) = 8 + 5
  - f(n) = f(n-1) + f(n-2)

This is called the fibonacci series or sequence

Recursion – base case:

- Functions that call themselves
- Works very well for situations where a function does a calculation for ***n*** that directly depends on the result of the function on ***n-1.***
  - That is: **f(n)** = some operation(s) on the value of **f(n-1)**
- **But**, how do we "know" the value of **f(n-1)?**
- **Factorial** is a classic example.
  - What is **f(5)** ?
    - **f(5) = 5 * f(4) What is f(4)?**
      - **f(4) = 4 * f(3) What is f(3)?**
        - **f(3) = 3 * f(2) What is f(2)?**
          - **f(2) = 2 * f(1) What is f(1)?**
          - **We, human programmer/mathematician, stipulate:**

            f(1) == 1. This is the "base case".
            NO RECURSION AT THIS STEP.

I know f(0) == 1 but making f(1) the base case makes is easier to trace.

Recursion – building back up from the base case

- What is f(5) ?
  - f(5) = 5 * f(4) What is f(4)?
    - f(4) = 4 * f(3) What is f(3)?
      - f(3) = 3 * f(2) What is f(2)?
        - f(2) = 2 * f(1) What is f(1)?
        - f(1) == 1. This is the "base case".
        - f(2) = 2 * 1 , f(1) = 1
      - f(3) = 3 * 2, f(2) = 2
    - f(4) = 4 * 6, f(3) = 6
  - f(5) = 5 * 24, f(4) = 24
  - f(5) = 120

Recursion – base case examples

- For the factorial function:
  - What would be the effect if we stipulated the base case is f(3) == 6?

- What would be a reasonable base case:
  - for the compounding interest function?
  - for the fibonacci sequence function?

- Trickier: What would be a reasonable base case:
  - for binary search?
  - for the selection sort?

To Sum up: A Recursion definition contains:

- **Base cases**
  - Condition or conditions where the function does not recursively call itself.
  - Their must be a least one base case
  - every possible chain of recursive calls must eventually reach a base case

- **Recursive calls**
  - Function calls made within a function, to that same function
  - Each recursive call should be defined so that it makes progress towards a base case.

Recursion vs. Iteration (looping)

- There are similarities between iteration (looping) and recursion.
- In fact, anything that can be done with a loop can be done with a simple recursive function! Some programming languages use recursion exclusively.
- Some problems that are simple to solve with recursion are quite difficult to solve with loops.

Recursion vs. Iteration

There are similarities between iteration (looping) and recursion.

- In fact, anything that can be done with a loop can be done with a simple recursive function! And visa versa. Some programming languages use recursion (almost) exclusively.
- Some problems that are simple to solve with recursion are quite difficult to solve with loops.

Recursion vs. Iteration: Efficiency

- In the factorial and binary search problems, the looping and recursive solutions use roughly the same algorithms, and their efficiency is nearly the same.
- There is a recursive algorithm to compute x^n, which is more efficient than an intuitive looping solution. The difference between them is like the difference between a linear and binary search.
- Usually, however iteration is quicker. And is preferred for elegance rather efficiency.

Recursion can be very inefficient even if elegant

- Calculating the n^{th} Fibonacci number is very elegant recursively:
- int fib(n)
  if (n == 0 or n == 1) return n
  else return fib(n-1) + fib(n-2)

The iterative one is a little more, well, clumsy:

in loopfib(n):
curr = 1; prev = 1;
for i == 0; i++ up to i == n-1
next = curr + prev; prev = curr; curr = next;
return curr;

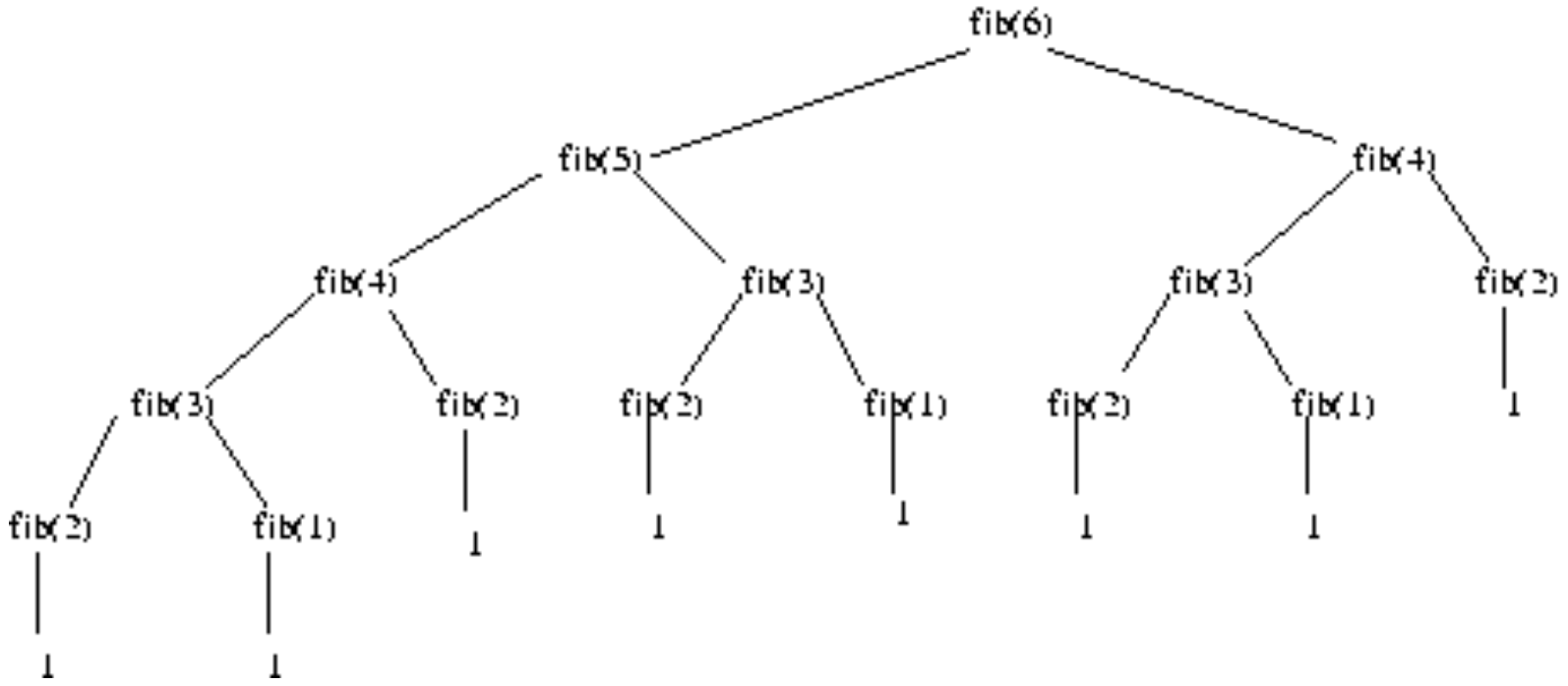
So go with the recursive algorithm. **Why not?**

Recursion vs. Iteration

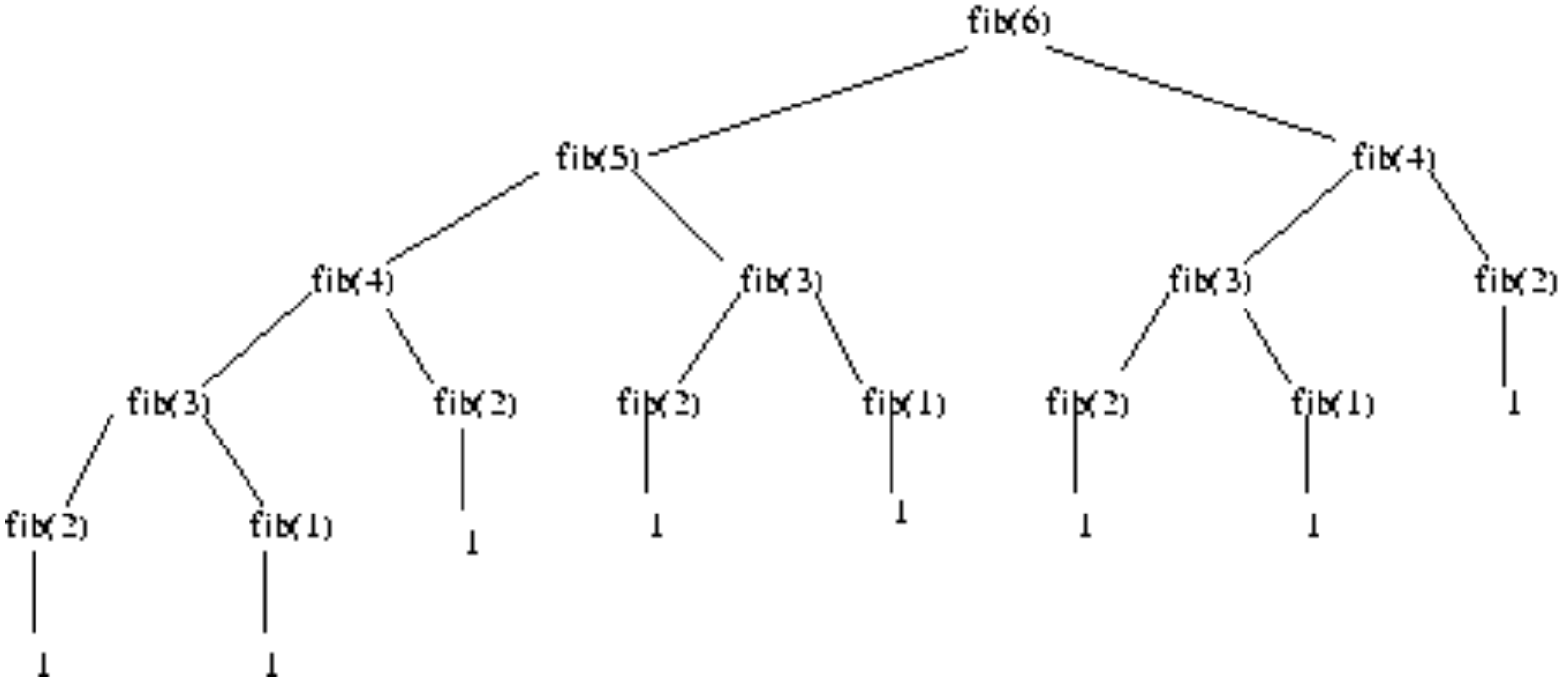
- The recursive Fibonacci algorithm obeys the rules that we've set out.
- The recursion is always based on smaller values.
- There is a non-recursive base case.
- So, this function will work great, won't it? – Sort of…

Recursion vs. Iteration

- The recursive solution is extremely inefficient, since it performs many duplicate calculations!



Recursion vs. Iteration



- To calculate fib(6), fib(4)is calculated twice, fib(3)is calculated three times, fib(2)is calculated four times… For large numbers, this adds up!

Recursion vs. Iteration

- Recursion is another tool in your problem-solving toolbox.
- 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.
- Avoid the recursive solution if it is terribly inefficient, unless you can't come up with an iterative solution (which sometimes happens!)

Summary: Recursion vs. Iteration

- Iteration can be used in place of recursion
- An iterative algorithm uses loops.
- A recursive algorithm uses function calls.
- Recursive solutions are often less efficient, in time and memory space, than iterative solutions.
- There's overhead for all those function calls.
- However, recursion can simplify the solution of a problem, often resulting in shorter source code that is more easily understood.

When to use recursion

- When there is no other easy-to-conceive way to solve a problem.
- If any other approach would become so convoluted and complex that it would be error-prone and difficult to understand, then use recursion.
- If the recursive version produces code that is much shorter and simpler, then use recursion.
- When the depth of recursive calls is "shallow."
- So, there won't be a lot of function calls.
- The recursive version does about the same amount of work as the non-recursive version.

The end