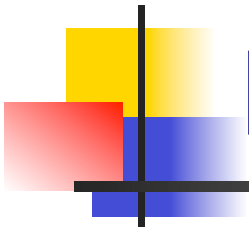
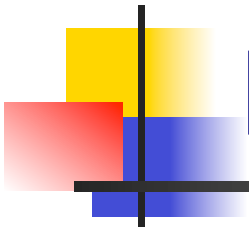
# Recursion



# Recursion

Recursion has a meaning in mathematics, computing, philosophy, and language.

Generally speaking, the idea behind recursion is that a given element or idea is based upon a prior element or idea, which in turn is based upon a prior element or idea, and so forth. In mathematics and computing, recursive functions are those which relate all elements of a set by a step algorithm and base cases.



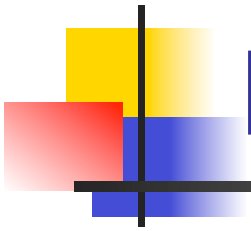
# Recursion

Base cases are like axioms. They define values for the algorithm which are accepted rather than calculated.

The step algorithm defines a series of actions that determine what the next element of a set is based on a given element.

An example of a recursive definition for natural numbers would look like:

1 is in N
If n is in N, then n+1 is also in N.

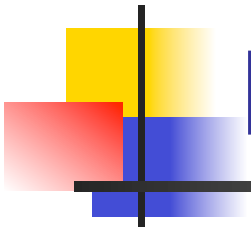


# Recursion

From that definition we can find all the elements in the set of natural numbers by starting at 1 and then running the recursive step over and over.

Functions define set mappings so they can also have a recursive definition that looks very similar to the one for natural numbers. You need all the base cases and the recursive step. The recursive function for factorial is:

F(0) = 1
F(n) = n * F(n-1)

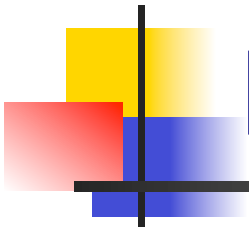


# Recursion

Let's translate this into C.

First we'll define how F(n) looks as a C function. Since factorial is only recursively defined for integers, we'll keep things as int.

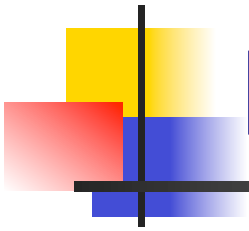
```
int factorial(int n)
{
}
```



# Recursion

Next, we need to check to see if the function is called with a base case. If so, we simply return the value dictated by the recursive definition. In our case, F(0) = 1, so factorial(0) should return 1.

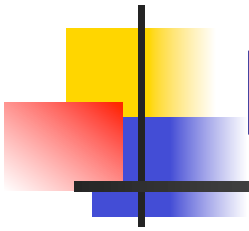
```
int factorial(int n)
{
  if (n == 0) { return 1; }
}
```



# Recursion

We only have the one base case, so now we need to add the recursive step algorithm. It is F(n) = n * F(n-1). So we just return n * F(n-1).

```
int factorial (int n)
{
  if (n == 0) { return 1; }
  else { return n * factorial(n-1); }
}
```

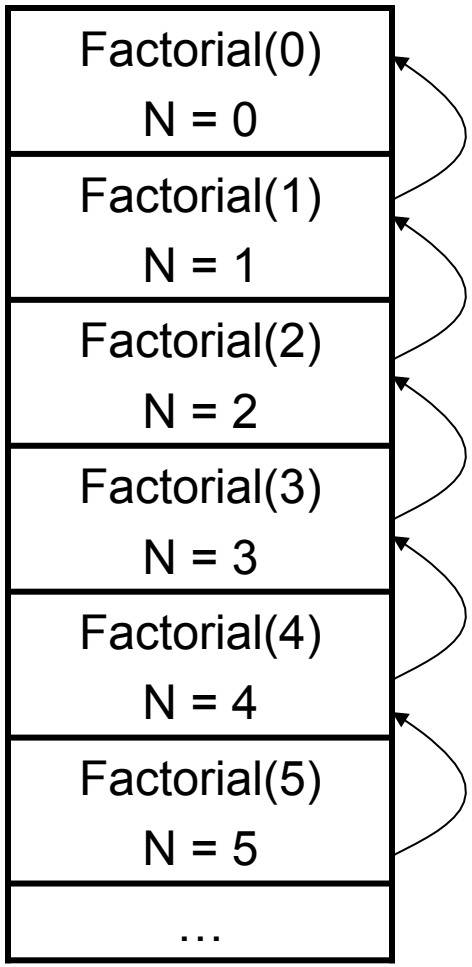


# Recursion

That's it. That probably seemed pretty easy. Is it easy to understand how the code works? Most people tend to find recursive code easier to read once they understand the process. So why don't we use it?

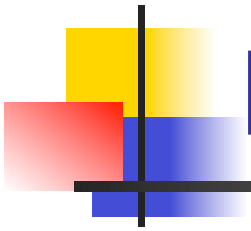
The problem stems from how compilers and computers handle functions. Most compilers handle each function call with a special process. They break out of the main program and create a new space in memory (called the stack) for the function. Data is copied into the stack that tells the computer where the function came from, along with any parameters.

# Recursion




Thus, a recursive function will create as many copies of itself on the stack as dictated by the step algorithm (ie factorial(5) would create six copies of itself on the stack). This takes up more memory on the stack than a single function call.

Moreover, each call requires setup and tear down time to create the stack. So recursive functions are actually slower and more memory intensive than regular iterative functions.



# Recursion

In fact, any recursive function can be defined as an iterative loop, so there is never a need to write recursive functions. (You all wrote loop versions of factorial for example).

All that said, some languages, like Scheme, handle function calls differently and can be more efficient, or as efficient, handling recursive functions as regular functions. In those languages, most programmers do use recursive functions. However, most of these languages are not used for application or scientific programming.