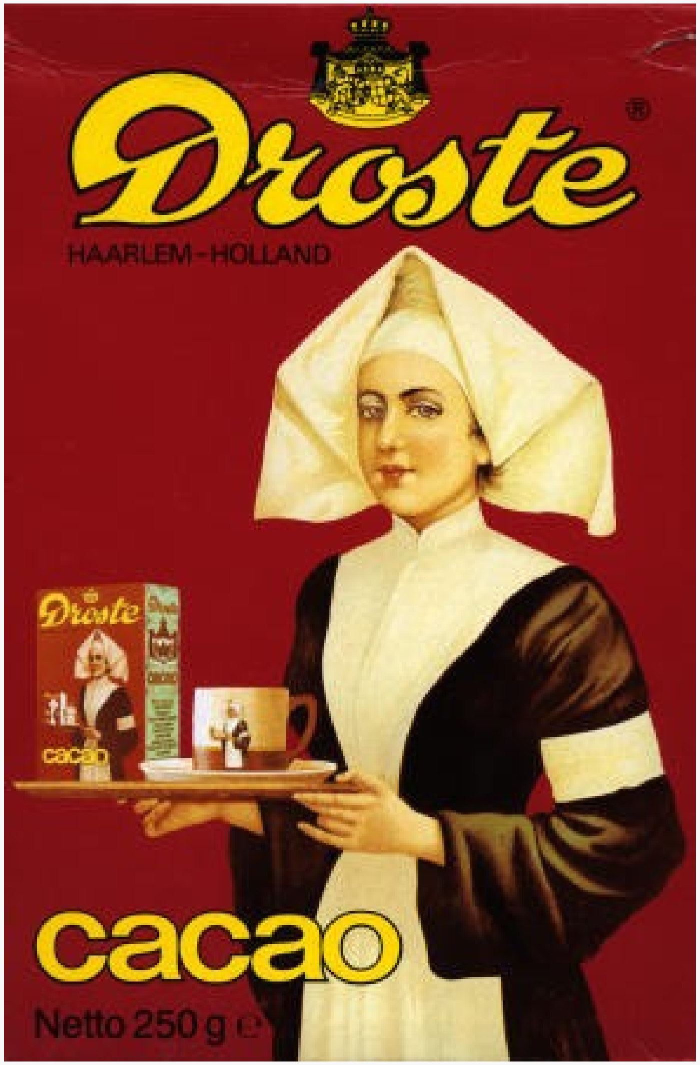
# Recursion




Around the year 1900 the illustration of the "nurse" appeared on Droste's cocoa tins.

This is most probably invented by the commercial artist Jan (Johannes) Musset, who had been inspired by a pastel of the Swiss painter Jean Etienne Liotard, *La serveuse de chocolat*, also known as *La belle chocolatière*.

The illustration indicated the wholesome effect of chocolate milk and became inextricably bound with the name Droste.

- Wikipedia Commons

# Recursively-defined Functions

**recursion** a method of defining functions in which the function being defined is applied within its own definition

$$factorial(n) = \begin{cases} 1 & n = 0 \\ n \cdot factorial(n-1) & n > 0 \end{cases}$$

$$fibonacci(n) = \begin{cases} 1 & n = 0,1 \\ fibonacci(n-1) + fibonacci(n-2) & n > 1 \end{cases}$$

```
uint64_t Fibonacci(uint64_t n) {
   if ( n < 2 )
      return 1;
   return Fibonacci(n – 1) + Fibonacci(n – 2);
}
```

**Reality check:**

**Does your recursive fn return a value?**

**If yes, are the calls to it all in assignment or return statements?**

**If (yes and) no, what are you thinking??**

# Execution Trace



```
uint64_t Fibonacci(uint64_t n) {
   if ( n < 2 )
      return 1;
   return Fibonacci(n – 1) + Fibonacci(n – 2);
}
```

```
Fibonacci(5)
  == Fibonacci(4) + Fibonacci(3)
  == Fibonaccci(3) + Fibonacci(2) + Fibonacci(2) + Fibonacci(1)
  == Fibonacci(2) + Fibonacci(1) + Fibonacci(1) + Fibonacci(1) +
                    Fibonacci(1) + Fibonacci(1) + 1
  == Fibonacci(1) + Fibonacci(1) + 1 + 1 + 1 + 1 + 1 + 1
  == 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1
  == 8
```

**Fibonacci(5) leads to 15 function calls… 14 of them recursive.**

# Printing Large Integers



Very large integers are (somewhat) easier to read if they are not simply printed as a sequence of digits:

`12345678901234567890` vs `12,345,678,901,234,567,890`

How can we do this efficiently? The basic difficulty is that printing proceeds from left to right, and the number of digits that should precede the left-most comma depends on the total number of digits in the number.

Here's an idea; let N be the integer to be printed, then:

if N has no more than 3 digits, just print it normally

otherwise

print all but the last 3 digits

print a comma followed by the last 3 digits

# Printing Large Integers

The preceding analysis leads directly to a recursive solution:

```
void printWithCommas(uint64_t N) {

   if ( N < 1000 )
      printf("%d", N);
   else {
      printWithCommas( N / 1000 );
      printf(",%#03d", N % 1000);
   }
}
```

```
printWithCommas( 1234567890123456789 0 )
printWithCommas( 1234567890123456 7 )          defer 890
printWithCommas( 1234567890123 4 )             defer 567
printWithCommas( 1234567890 1 )                defer 234
printWithCommas( 12345678 )                    defer 901
printWithCommas( 12345 )                       defer 678
printWithCommas( 12 )                          defer 345
                                               print 12
```

# GCD

If x and y are non-negative integers so that x >= y and y >= 0, and not both are 0, the *greatest common divisor* (GCD) of x and y is the largest integer z that divides both x and y.

So: GCD(12, 9) = 3 GCD(36, 28) = 4

GCD(18, 18) = 18 GCD(12, 0) = 12

That is:

$$GCD(x, y) = \begin{cases} x & y = 0 \\ GCD(y, x\%y) & x \geq y \text{ and } y > 0 \end{cases}$$

```
uint64_t GCD(uint64_t x, uint64_t y) {

   if ( y == 0 ) return x;

   return GCD(y, x % y);
}
```

$$z \mid x \text{ and } z \mid y \Rightarrow z \mid (x\%y)$$

$$z \mid y \text{ and } z \mid (x\%y) \Rightarrow z \mid x$$

# Recursion vs Iteration

```
uint64_t Fibonacci(uint64_t n) {
   if ( n < 2 )
      return 1;
   return Fibonacci(n – 1) + Fibonacci(n – 2);
}
```

```
uint64_t iFibonacci(uint64_t n) {

   if ( n < 2 ) return 1;

   uint64_t FiboNMinusTwo = 1;
   uint64_t FiboNMinusOne = 1;
   uint64_t FiboN;

   for (uint64_t i = 2; i <= n; i++) {
      FiboN = FiboNMinusOne + FiboNMinusTwo;

      FiboNMinusTwo = FiboNMinusOne;
      FiboNMinusOne = FiboN;
   }

   return FiboN;
}
```

# Recursion vs Iteration

```
uint64_t GCD(uint64_t x, uint64_t y) {

           if ( y == 0 ) return x;

           return GCD(y, x % y);
}
```

```
uint64_t iGCD(uint64_t x, uint64_t y) {

   while ( y != 0 ) {
      uint64_t Remainder = x % y;
      x = y;
      y = Remainder;
   }
   return x;
}
```

# Tail vs non-Tail Recursion

Tail-recursive algorithms are ones that end in a recursive call (the only recursive call) and do not leave any deferred operations:

```
uint64_t GCD(uint64_t x, uint64_t y) {

           if ( y == 0 ) return x;

           return GCD(y, x % y);
}
```

```
uint64_t Fibonacci(uint64_t n) {
   if ( n < 2 )
      return 1;
   return Fibonacci(n – 1) + Fibonacci(n – 2);
}
```

**deferred until the second call returns; sometimes called augmenting recursive**

**How many function calls would result from Fibonacci(5)?**

# A Fibonacci Refinement



Here's an alternative recursive Fibonacci calculator:

```
uint64_t Fibonacci(unsigned int N, uint64_t FiboNMinusTwo,
                                   uint64_t FiboNMinusOne) {

   if (N < 2) {                                   // base cases
      return FiboNMinusTwo;
   } else if (N == 2) {                           // don't recurse, trivial
      return FiboNMinusOne + FiboNMinusTwo;
   } else {
      return Fibonacci(N - 1, FiboNMinusOne,
                              FiboNMinusOne + FiboNMinusTwo);
   }
}
```

**is this tail-recursive?**

**How many function calls would result from Fibonacci(5,1,1)?**

# Execution Trace

```
uint64_t Fibonacci(unsigned int N, uint64_t FiboNMinusTwo,
                                   uint64_t FiboNMinusOne) {

   if (N < 2) {                               // base cases
      return FiboNMinusTwo;
   } else if (N == 2) {                       // don't recurse, trivial
      return FiboNMinusOne + FiboNMinusTwo;
   } else {
      return Fibonacci(N - 1, FiboNMinusOne,
                              FiboNMinusOne + FiboNMinusTwo);
   }
}
```

```
Fibonacci(5, 1, 1)
  == Fibonacci(4, 1, 2)
  == Fibonacci(3, 2, 3)
  == Fibonacci(2, 3, 5)
  == 8
```

**How much difference does this make?**

**On my laptop, computing Fibonacci(10) 10,000,000 times using the original recursive version took 20.3130 seconds.**

**This version took 1.0630 seconds.**

**The iterative version took 0.9350 seconds.**

# Performance: Recursion vs Iteration

Recursive algorithms can always be implemented using iteration instead (although some sort of auxiliary structure like a stack may be required).

A recursive implementation is often shorter and more obvious.

An recursive implementation adds cost due to the number of function calls that are required and the related activity on the run-time stack.

An iterative implementation may, therefore, be faster… but not always.

An iterative implementation may require more memory from the heap (dynamic allocations) and a recursive implementation may require more memory on the stack… this has implications if you're targeting a limited environment, like an embedded system.