Prof. James G. Looby
Hudson Valley Community College

What do we know?

Well as humans dealing with bytes, we typically convert decimal to binary by inspection and that we write out the powers of two and look for the highest common denominator. As an example let's convert 0d13 to binary (recall we preface the number by indicating the numbering system so binary numbers are preceded with 0b and hexadecimal numbers are proceeded with 0x).

We write out the positional values noting that I also provided the powers of two   below.

$$2^3 \qquad 2^2 \qquad 2^1 \qquad 2^0$$
$$8\text{'s} \qquad 4\text{'s} \qquad 2\text{'s} \qquad 1\text{'s}$$

So to convert 0d13 to binary, moving left to right,
I would determine that there is one 8 in 13 so
     I record a 1
     I would subtract 8 from 13 leaving a remainder of 5,
In the remainder 5 there is one 4 so
     I record a 1 and I would subtract 4 from 5 leaving a remainder of 1
There are no 2s in 1 so I would record a 0
There is one 1 in 1 so I would record a 1
Result ➔ 0b1101

Ok, but this is not efficient for a computer. Consider if we needed to determine the binary value for 0d123,464,231,378,223. To do this using our inspection algorithm above we would need to determine what is the largest power of two and this is wasteful so let's look at a more efficient algorithm.
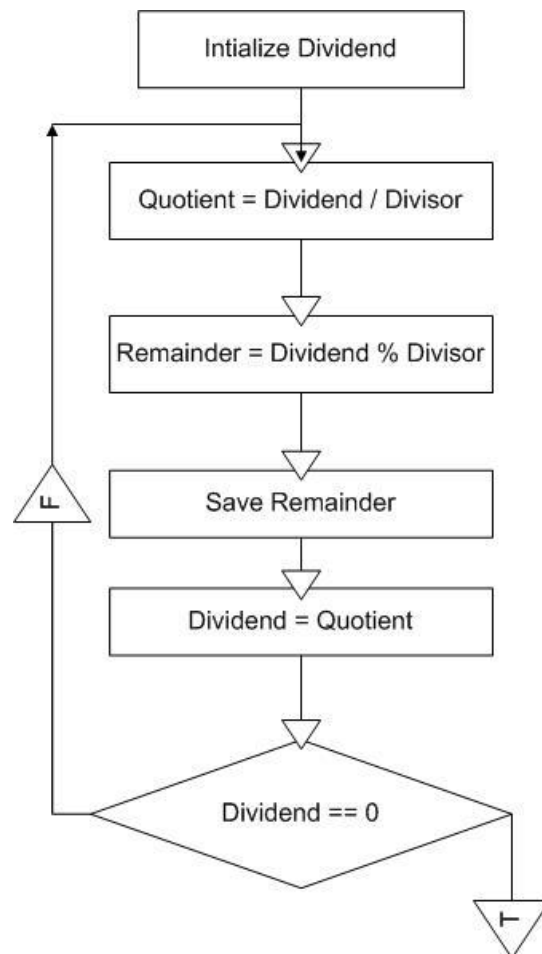
Decimal to Binary Programming

Algorithm:

The general procedure is to repeatedly divide the Decimal Number by two and record the remainder moving right to left.

Terminology:

Quotient   Remainder

Divisor /Dividend

Sequential Algorithm in Pseudocode:

1. Divide Decimal Number (Dividend) by 2. (i.e. Quotient = Dividend/Divisor)
2. Save/Record the remainder
3. Quotient becomes the new Dividend (i.e. Dividend = Quotient)
4. Goto 1.

Now for our 0d13 example we have:

1st pass/iteration

       Dividend is 13
       Divisor is 2
       $13/2 \Rightarrow 6$ r 1 (6 is quotient, remainder is 1)

2nd pass

       Dividend is now 6 (last quotient)
       Divisor remains 2
       $6/2 \Rightarrow 3$ r 0

3rd pass

       Dividend is now 3 (last quotient)
       Divisor remains 2
       $3/2 \Rightarrow 1$ r 1

4th pass

       Dividend is now 1
       Divisor remains 2
       $1/2 \Rightarrow 0$ r 1
       Dividend becomes 0 and loop terminates

Now if we work through this and assess this a process, we see that we determine the least significant bit first (one's place in 13/2) but we need to print out the result left to right beginning with the most significant bit. Now of course we could use a data structure like an array to store the bits, keep track of our array index and then print out the result from the last or highest array position down to the lowest. This would be just as an even more efficient than our recursive solution but where's the fun in that and we need to learn recursion.

Recursion

In normal procedural languages, one can go about defining functions and procedures, and 'calling' these from the 'parent' functions. I hope you already know that.   Some languages also provide the ability of a function to call **itself**. This is called Recursion.

Factorial

Factorial is a mathematical term. Factorial of a number, say n, is equal to the product of all integers from 1 to n. Factorial of n is denoted by **n! = 1x2x3...x n.**
Eg: `10! = 1x2x3x4x5x6x7x8x9x10`

The simplest program to calculate factorial of a number is a loop with a product variable. Instead, it is possible to give a *recursive* definition for Factorial as follows:

```
1) If n=1, then Factorial of n = 1
2) Otherwise, Factorial of n = product of n and
                               Factorial of (n-1)
```

Check it out for yourself; it works. The following code fragment (in C) depicts Recursion at work.

```
int Factorial(int n)
{
    if (n==1)
        return 1;
    else
        return Factorial(n-1) * n;
```

The important thing to remember when creating a recursive function is to give an '*end-condition*'. We don't want the function to keep calling itself forever, now, do we? Somehow, it should know when to stop. There are many ways of doing this. One of the simplest is by means of an 'if condition' statement, as above. In the above example, the recursion stops when n reaches 1. In each instance of the function, the value of n keeps decreasing. So it ultimately reaches 1 and ends. Of course, the above function will run infinitely if the initial value of n is less than 1. So the function is not perfect. The `n==1` condition should be changed to `n<=1`.

Imagination is a very hard thing. Imagination of Recursion is all the more tricky. Think of clones. Say you have a machine to make clones of yourself, and (for lack of a better pass-time) decide to find the factorial of a number, say 10, using your clones. So, being smart, this is what you do:

First, there's only *You*. Let's call you *You-1*. You have the number 10 in your pocket. Being smart, you know that all you need to find the factorial of 10 (`10*9*...*2*1`) is to somehow obtain the value of 9 factorial (`9*8*..*2*1`), and then just multiply it with 10. So that's what you do. You turn on your machine and out pops a clone! You give the clone *You-2* strict instructions to find the factorial of 9 and make it quick! Your job is done for a while, so you (*You-1*) stretch on your sofa sipping on your lemonade. Meanwhile...

You-2 is (you guessed it) just as smart as you! He tucks his number (9) into his pocket, turns on the machine, and out pops a clone (*You-3*). The new clone is given the job of 8-factorial, which it proceeds to do while (unbeknownst to you) *You-2* is sipping on his own glass of lemonade on his own sofa. And so the story goes on until finally one fine day...

Out pops *You-10* who is given strict instructions (by *You-9*) to get the factorial of 1. Now, *You-10*, being just as smart as any of the other you's, knows very well that the factorial of 1 is... 1. So he says to *You-9* (who was just about to doze off on his sofa), "Here's your factorial of 1." *You-9* snatches the result from his subordinate *You-10*, takes out his plasma gun, and zaps *You-10* out of existence. He scribbles on a piece of paper, calculating the product of the value he got from *You-10* with the number in his pocket, 2. "Heh, heh, heh" he thinks, and goes to his boss, *You-8*, saying,"Here's your factorial of 2..." *...blah...blah...* and finally *You-2* wakes you up from your slumber, and says to you, "Here's your factorial of 9" You zap him off, multiply by the 10 in your pocket, and There You Have It !! Now, wasn't that simple?

Here, '*You*' were the function. The 'clones' are merely new instances of the same function. They all think and act alike. At one point, there are 10 You's (which occupies a lot of memory space). As soon as an instance returns a value and finishes its job, it is zapped off from memory.

Recursion can get much, much trickier than this but it is a good mental exercise and beginning.

Recursive Decimal to Binary in Java

//Recursive Java Method printBinary

```java
public static String printBinary (int dividend)
{
        int quotient, remainder;
        quotient = dividend/2;                          //1
        remainder = dividend%2;                         //2
        if (quotient > 0)                               //3
            return(printBinary (quotient) + remainder);  //4
        else return "1";                                //5
}
```