3. Algebra

Solving Equations ax + b = cx + d
Algebra is like one big number guessing game.

"I'm thinking of a number. If you multiply it by 2 and add 5, you get 21."	$2x + 5 = 21$

For a long time in Algebra class, there are hundreds of equations to solve that can be put in the form a*x* + b = c*x* + d. Sometimes there's no x term on one side, meaning the coefficient is zero.

$$3x - 5 = 22$$
$$4x - 12 = 2x - 9$$
$$\frac{1}{2}x + \frac{2}{3} = \frac{1}{5}x + \frac{7}{8}$$

Using a little Algebra, you can solve the general form of the equation and that will help you solve all equations of that form.

$$ax + b = cx + d$$
$$ax - cx = d - b$$
$$x(a - c) = d - b$$
$$x = \frac{d - b}{a - c}$$

Now we can write a function to take the coefficients of the equation you want to solve and return that value of x:

```
def equation(a,b,c,d):
    '''Returns the solution of an equation
    of the form ax + b = cx + d'''
    return (d - b)/(a - c)
```

See how I translated the algebra solution to the output of the function? To solve our first equation 3x - 5 = 22, enter the coefficients, with c being 0:

```
>>> equation(3,-5,0,22)
9.0
```

That's true: 3(9) - 5 = 22. How about the second equation? (4x - 12 = 2x - 9)

```
>>> equation(4,-12,2,-9)
1.5
```

Yes, 4(1.5) - 12 = -6 and 2(1.5) - 9 = -6.

To solve the equation

$$2x - 8 = 4x + 5$$

Enter this:

```
>>> equation(2,-8,4,5)
-6.5
```

Using the Pythagorean Theorem

Here's a very useful program to solve for the third side of a right triangle using the Pythagorean Theorem:

```
from math import sqrt

def pythag(leg1,leg2):
    '''solves for hypotenuse'''
    return sqrt(leg1**2 + leg2**2)

def pythag2(leg1,hyp):
    '''solves for other leg
    of right triangle'''
    return sqrt(hyp**2 - leg1**2)
```

```
>>> pythag(5,12)
13.0
>>> pythag2(24,25)
7.0
```

Solving Higher-Degree Equations

Algebra is where Python starts to really get useful. At some point in Algebra, the book starts using "f(x) =" instead of "y =". Mathematical functions are like machines where you put a number in (the input), the machine does something to it, then returns a number (the output).

We've already seen how easy this is in Python. We've made functions that take parameters like "length" and "number" like:

```
def square(length):
    for i in range(4):
        fd(length)
        rt(90)
```

and

```
def equation(a,b,c,d):
    return (d - b)/(a - c)
```

We already know how to take in input and return output. So when an Algebra book gives us a problem like this one:

Let $f(x) = x^2 + 5x + 6$. Find $f(2)$ and $f(10)$.

We can define the function f(x) like any Python function, input numbers and get the output instantaneously:

```
def f(x):
    return x**2 + 5*x + 6

>>> f(2)
20
>>> f(10)
156
```

Program: Factoring Polynomials

Some algebra textbooks make you factor a lot of polynomials. Here's how to make a computer do it by brute force.

You know the expression $ax^2 + bx + c$ (if it's factorable) must split up into the form

$$(\boldsymbol{dx} + \boldsymbol{e})(\boldsymbol{fx} + \boldsymbol{g})$$

d and **f** are factors of **a** and **e** and **g** are factors of **c**. All you have to do is go through all the factors of **a** and all the factors of **c**, and see if d * g + e * f equals b. An annoying job for a human but the computer doesn't mind all that repetition. We can't just import our factors function for this one, because we need to include the negative factors. Here's the updated function:

```
def factors(n):
    '''returns a list of the positive and
    negative factors of a number'''
    factorList = []
    n = abs(n) #absolute value of the number
    for i in range(1,n+1): #all the numbers from 1 to n
        if n % i == 0: #if n is divisible by i
```

```
            factorList.append(i) #i is a factor
            factorList.append(-i) # so is -i
    return factorList
```

Now factoring the polynomial is just a matter of plugging in the factors of a and c and seeing which combination adds up to b.

```
def factorPoly(a,b,c):
    '''factors a polynomial in the form
    ax**2 + b*x + c'''
    afactors = factors(a) #Get the factors of a
    cfactors = factors(c)   # and c
    for afactor in afactors: #try all the factors of a
        for cfactor in cfactors: #and c
            #see which combination adds up to b
            if afactor * c/cfactor + a/afactor * cfactor == b:
                print('(',afactor,"x +",cfactor,")(",
                      int(a/afactor),"x +",int(c/cfactor),")")
                return
```

Here's how to factor the polynomial $6x^2 - x - 15$:

```
>>> factorPoly(6,-1,-15)
( 2 x + 3 )( 3 x + -5 )
```

Program: Quadratic Formula

But factoring polynomials is of limited value because not all quadratics are factorable. There's a better way to solve equations involving an x^2 term: the quadratic formula. It'll give you solutions that are whole numbers or decimals, real numbers or imaginary!

```
def quad(a,b,c):
    '''Returns the solutions of an equation
    of the form a*x**2 + b*x + c = 0'''
    x1 = (-b + sqrt(b**2 - 4*a*c))/(2*a)
    x2 = (-b - sqrt(b**2 - 4*a*c))/(2*a)
    return x1,x2
```

If you need to solve the quadratic equation $x^2 + 3x - 10 = 0$, you would enter 1 for a, 3 for b and -10 for c this way:

```
>>> quad(1,3,-10)
(2.0, -5.0)
```

That means the values for x that make the equation true are 2 and -5. To check them, you plug in those values for x: $2^2 + 3(2) - 10 = 0$ and $(-5)^2 + 3(-5) - 10 = 0$

With a little more coding the quadratic formula can give you solutions whether they're whole numbers, fractions, decimals, irrationals, even imaginary numbers. It all depends on whether the "discriminant," $b^2 - 4ac$, is positive or negative.

```
from math import sqrt

def quad(a,b,c):
    '''returns solutions of equations of the
    form a*x**2 + b*x + c = 0'''
    discriminant = b**2 - 4*a*c
    if discriminant >= 0: #real solutions
        x1 = (-b + sqrt(discriminant))/(2*a) #first solution
        x2 = (-b - sqrt(discriminant))/(2*a) #second solution
        return x1, x2
    else: #imaginary solutions
        real_part = -b/(2*a)
        imaginary_part = sqrt(-discriminant)/(2*a)
        print(real_part,"+",str(imaginary_part)+"i")
        print(real_part,"-",str(imaginary_part)+"i")
```

Now it will solve the quadratic even if there are only imaginary solutions.

```
>>> quad(1,3,10)
-1.5 + 2.7838821814150108i
-1.5 - 2.7838821814150108i
```

Higher order solutions (for equations with x^3 or x^4 and so on) require other methods.

Brute Force

You could just plug in every number you can think of. Actually, this is pretty easy for a computer. To solve the equation $6x^3 + 31x^2 + 3x - 10 = 0$, for example, you could just have the computer plug in every number between -100 and 100 for x and if it equals zero, print it out:

```
def plug():
    for x in range(-100,100):
        if 6*x**3 + 31*x**2 + 3*x - 10 == 0:
```

```
            print("One solution is", x)
    print("Done plugging.")
```

```
>>> plug()
One solution is -5
Done plugging.
```

So the other solutions to that equation are not integers. You could change the program to plug in decimals, too.

```
def plug():
    x = -100
    while x <= 100:
        if 6*x**3 + 31*x**2 + 3*x - 10 == 0:
            print("One solution is", x)
        x += 0.5 #Will go up in half-unit steps
    print("Done plugging.")
```

```
>>> plug()
One solution is -5.0
One solution is 0.5
Done plugging.
```

But is there a better way? There is a name for a method of solving equations by showing every possible input and output of a function within a certain range and it's called **graphing**.

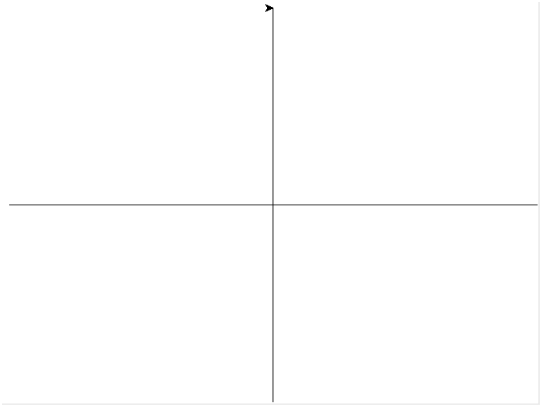
Major Math Tool: Create Your Own Grapher

What better tool is there for exploring functions than a graph? There are many graphing calculators and programs out there (some are free!) but many math teachers don't allow students to use graphers. What if the student made the grapher tool? We can use the `turtle` module to create graphs of lines and curves. First you have to define how to draw a grid.

```
from turtle import *
def setup():
    speed(0)         #Sets turtle speed to fastest
    setworldcoordinates(-5,-5,5,5) #lower left and upper right corners
    setpos(0,0)      #sets turtle's position to (0,0)
    clear()          #Clears its trails
    setheading(0)    #Faces the right of the screen
    pd()             #Puts its pen down to draw
```

```
    color("black")  #Sets its color to black
    for i in range(4):  #"Do this four times"
        setpos(0,0)     #Centers itself again
        fd(5)           #Go forward 5 steps
        rt(90)          #turn right 90 degrees
    pu()            #Sets its pen in up position
```

If you type “setup()” in the shell you’ll see this:



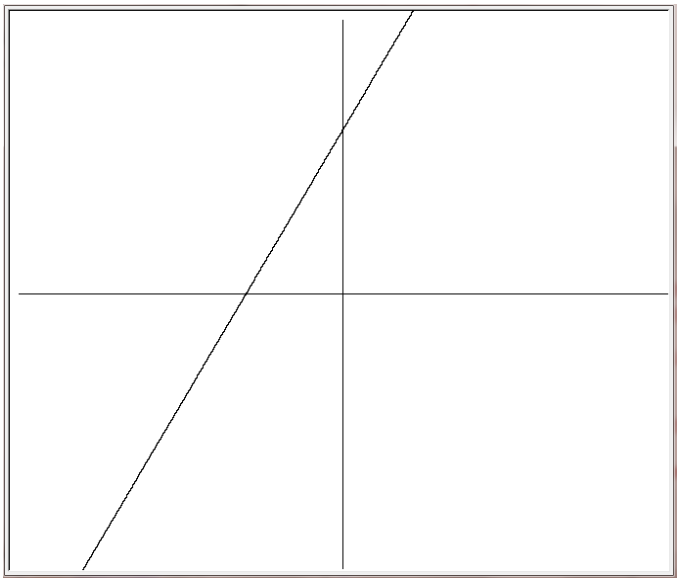
Now you have to define the function for the turtle to draw.

```
def f(x):           #This defines the function f(x)
    return 2*x + 3  #The line y = 2x + 3

def graph(f):    #The function f(x) has to be defined
    speed(0)    #Sets speed to the fastest
    color("black")  #Sets its color
    setpos(-6,f(-6))    #Sets its position to the left edge
    pd()                #pen down
    setheading(0)       #Face right
    x = xcor()  #set a variable, x, to the x-coordinate of the turtle
    while xcor() <= 6:  #Do this until the x-coordinate is more than 6
        x += 0.01       #make x go up a tiny bit
        goto(x,f(x))    #Go to the next point on the graph
```

You’ve defined the functions, but you haven’t told the program to run those functions yet. You can type commands in the shell or add this to the end of the program:

```
#Execute these functions automatically on "run"
setup()
graph(f)
```

Your output should be a line:



The great thing about having this tool is it will come in handy for all your math subjects from Algebra to Differential Equations!

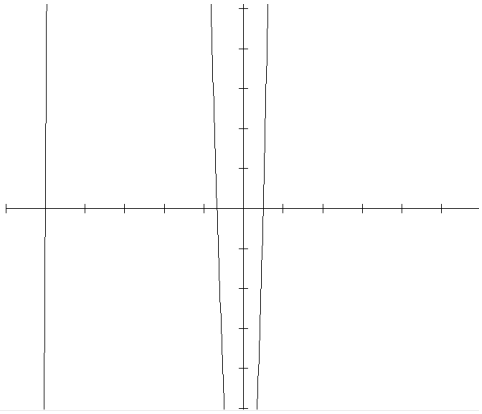
If you add ticks to the axes, then you can graph a function and maybe the solution will be easy to see. Here's the setup program for a grapher with ticks:

```
#Grapher with tick marks at whole numbers
from turtle import *

def mark(): #to make a tick mark on an axis
    rt(90)  #turn right 90 degrees
    fd(0.1) #go forward a little bit
    bk(0.2) #go back twice as far. The tick is drawn.
    fd(0.1) #go forward to the axis
    lt(90)  #turn left to continue drawing the axis

def setup():
    speed(0) #Sets turtle speed to fastest
    setworldcoordinates(-6,-5,6,5) #lower left and upper right corners
    setpos(0,0)     #sets turtle's position to (0,0)
    clear()         #Clears its trails
    setheading(0)   #Faces the right of the screen
    pd()            #Puts its pen down to draw
    color("black")  #Sets its color to black
    for i in range(4):  #"Do this four times"
```

```
        setpos(0,0)     #Centers itself again
        for i in range(6): #Make 6 tick marks
            fd(1)
            mark()
        rt(90)
    pu()            #Sets its pen in up position
```

Run this and you'll get a pretty graph.

We'll use our grapher to solve our equation $6x^3 + 31x^2 + 3x - 10 = 0$. First, graph the function $f(x) = 6x^3 + 31x^2 + 3x - 10$.



Change the "def f(x)" line of code:

```
def f(x):#This defines the function f(x)
    return 6*x**3 + 31*x**2 + 3*x - 10
```

and run it. The graph is on the left. We knew about the solutions at x = -5 and at x = 0.5 and now we can tell there's a solution between -1 and 0.

Rational Roots

We can narrow down all the possible rational roots (whole numbers or fractions) by dividing all the factors of the constant term (in this case, -10) by all the factors of the coefficient of the highest degree of x (in this case, 6). This is the list, positives and negatives:

$\pm 1, \pm 2, \pm 5, \pm 10, \pm ½, \pm ⅓, \pm ⅙, \pm ⅔, \pm 5/3, \pm ⅚, \pm$ 10/3

It used to be a whole lot of work to plug in a couple of dozen numbers, but that was before computers. Now we can just write a program to generate all those roots and plug them in for us. First we need the `factors` function (which we saved in our ***arithmetic.py*** file), which returns a list of all the factors of a number:

```
from arithmetic import factors
```

We can use this function to generate all the possible rational roots of our equation. Here's the function:

```
def rationalRoots(a,b):
```

```
    '''Returns all the possible rational roots of
    a polynomial with first coefficient a and
    constant b'''
    roots = [] #create a list to store the roots
    numerators = factors(b) # generate a list of factors of b
    denominators = factors(a) # generate a list of factors of a
    for numerator in numerators: #loop through the numerator list
        for denominator in denominators: #and the denominator list
            roots.append(numerator / denominator)
            roots.append(-numerator / denominator)
    return roots
```

Now if I put in 6 for a and 10 for b:

```
>>> rationalRoots(6,10)
[1.0, -1.0, 0.5, -0.5, 0.3333333333333333, -0.3333333333333333,
0.16666666666666666, -0.16666666666666666, 2.0, -2.0, 1.0, -1.0,
0.6666666666666666, -0.6666666666666666, 0.3333333333333333, -
0.3333333333333333, 5.0, -5.0, 2.5, -2.5, 1.6666666666666667, -
1.6666666666666667, 0.8333333333333334, -0.8333333333333334, 10.0, -
10.0, 5.0, -5.0, 3.3333333333333335, -3.3333333333333335,
1.6666666666666667, -1.6666666666666667]
```

We already have a plug function, so I can just change it a little to plug in all those values.

```
def plug2(factorlist):
    for x in factorlist:
        if 6*x**3 + 31*x**2 + 3*x - 10 == 0:
            print("One solution is", x)
    print("Done plugging.")
```

Execute the plug function on the factor list:

```
>>> plug2(rationalRoots(6,10))
One solution is 0.5
One solution is -0.666666666666
One solution is -5.0
One solution is -5.0
Done plugging.
```

The root x = -5 came up twice (-5/1 and -10/2). **Now we know our three solutions: x = -5, x = -**

⅔ and x = ½.

Now we can use the tools we just created to easily generate all the rational solutions to **any** polynomial equation. For example, if we wanted to solve the equation

$$48x^5 - 44x^4 - 884x^3 + 321x^2 + 3143x + 980 = 0.$$

What a monstrosity! But the worst part is typing it all out:

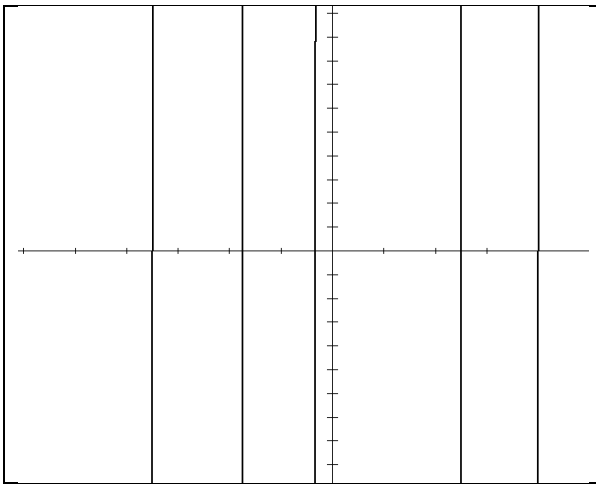
```
def h(x): #define the function so we can use "graph(h)"
        return 48*x**5 - 44*x**4 - 884*x**3 + 321*x**2 + 3143*x +980

def plug3():
    '''plugs every item in a list of factors into h(x)'''
    for x in rationalRoots(48,980):
        if h(x) == 0:
            print("One solution is", x)
    print("Done plugging.")
```

Now the program does all the work instantly:

```
>>> plug3()
One solution is 4.0
One solution is 2.5
One solution is -3.5
One solution is -1.75
One solution is 2.5
One solution is -3.5
One solution is -1.75
One solution is 2.5
One solution is -3.5
One solution is -1.75
Done plugging.
```

There's some repetition here; there are four rational solutions above.
x = -3.5, -1.75, 2.5 and 4.

Here's the graph:



It looks like there's another root between -1 and 0. This method fails if there's a repeating decimal, like -0.33333… because if you plug that decimal into the equation you don't get zero. You get something tiny like a trillionth, but technically it's not equal to zero so our function threw it out. But there is a way to use the roots we know to get more roots.

Synthetic Division

Synthetic division is a way to do long division on polynomials. It's time consuming to do by hand, but it's easy to write a program to make the computer do it. Because we know x = 4 is a root of $48x^5 - 44x^4 - 884x^3 + 321x^2 + 3143x + 980$, we know we can write it as

$$(x - 4)(???) = 48x^5 - 44x^4 - 884x^3 + 321x^2 + 3143x + 980$$

"(???)" is a fourth-degree polynomial. We can divide the polynomial $48x^5 - 44x^4 - 884x^3 + 321x^2 + 3143x + 980$ by x - 4 to find out what it is.

```
def synthDiv(divisor,dividend):
    '''divides a polynomial by a constant and returns a lower-degree
    polynomial. Enter divisor as a constant: (x - 3) is 3
    Enter dividend as a list of coefficients:
    x**2 – 5*x + 6 becomes [1,-5,6]'''
    quotient = [] #empty list for coefficients of quotient
    row2 = [0]    #start the second row
    for i in range(len(dividend)):
        quotient.append(dividend[i]+row2[i]) #add the ith column
        row2.append(divisor*quotient[i]) #put the new number in row 2
    print(quotient)
```

And here's how to enter it:

```
>>> synthDiv(4,[48,-44,-884,321,3143,980])
[48, 148, -292, -847, -245, 0]
```

Those are the coefficients of the new polynomial. It's
$48x^4 + 148x^3 - 292x^2 - 847x - 245$. The last number is the remainder of the division, which is 0.

So we've factored

$48x^5 - 44x^4 - 884x^3 + 321x^2 + 3143x + 980$

into

$(x - 4)(48x^4 + 148x^3 - 292x^2 - 847x - 245)$

We can keep doing this with all the other factors we know, like 2.5. Copying and pasting the factor list:

```
>>> synthDiv(2.5,[48, 148, -292, -847, -245])
[48, 268.0, 378.0, 98.0, 0.0]
```

Now we've factored our polynomial further into

$(x - 4)(x - 2.5)(48x^3 + 268x^2 + 378x + 98)$

Using the factor x = -3.5:

```
>>> synthDiv(-3.5,[48, 268.0, 378.0, 98.0])
[48, 100.0, 28.0, 0.0]
```

Now our factored polynomial becomes:
$(x - 4)(x - 2.5)(x + 3.5)(48x^2 + 100x + 28)$

We can put the coefficients in the x^2 expression into the Quadratic Formula to find our last two roots:

```
>>> quad(48,100,28)
-0.333333333333 -1.75
```

We already knew x = -1.75 is a solution, and now we know the last one is x = `-0.33333333`.

Synthetic Division can help us find all the rational roots of a polynomial, no matter how ugly it is! It can even help us find irrational roots, because we used the quadratic formula to find the last two roots.

Use this program to find all the solutions to the equation

$$8x^6 - 206x^5 + 1325x^4 + 1273x^3 - 14980x^2 - 17825x - 4875 = 0$$

Hint: four solutions are rational, and two are irrational.

Exploring Prime Numbers

Program: IsPrime()

Here's a good exploration for learning to use loops and conditionals. It tests whether a number is prime. You have to divide by 2, then 3, and so on up to which number? The number minus one? Half the number?

Remember the modulo or "mod" operator. Its symbol is the percent sign (%). The remainder when you divide 10 by 3 is:

```
>>> 10 % 3
1
```

This allows us to check whether a number is divisible by another one. Let's create a "divisible" function:

```
def divisible(a,b):
    '''Returns True if a is divisible by b'''
    return a % b == 0
```

We can use this function inside an "isPrime" function. To test whether 61 is prime, we just divide 61 by every number less than 61:

```
def isPrime(n):
    for i in range(2,n): #every number from 2 to n - 1
        if divisible(n,i): #if n is divisible by i
            return False    #n is not Prime (and stop)
    return True     #if it hasn't stopped, n is Prime
```

Here's how you check:

```
>>> isPrime(61)
True
```

But what if it isn't? You need to factor the number if it isn't Prime:

```
def isPrime2(n):
    '''Returns "True" if n is Prime'''
    for i in range(2,n):
        if divisible(n,i):
            print (n,"=",i,"x",n/i)
            return
    return True
```

```
>>> isPrime2(161)
161 = 7 x 23.0
```

How many numbers do we really have to divide by? For example, we check the number 101. The number we divide by (i) starts off smaller than n/i, but somewhere it gets bigger. Can you tell where?

```
n       i       n/i
101      1      101.0
101      2      50.5
101      3      33.6666666667
101      4      25.25
101      5      20.2
101      6      16.8333333333
101      7      14.4285714286
101      8      12.625
101      9      11.2222222222
101      10      10.1
101      11      9.18181818182
101      12      8.41666666667
101      13      7.76923076923
```

It's between 10 and 11. This means there's no way there could be a factor of 101 bigger than 11 because it would have to be multiplied by a number smaller than 11. **And we've checked all those numbers already.** What's this magic point in between 10 and 11? **The square root of 101.**
Change your code to:

```
from math import sqrt
def isPrime3(n):
    m = sqrt(n)
    for i in range(2,int(m) + 1): #range has to be integers
        if divisible(n,i):
            print(n,'=',i,'x',n/i)
            return
    return True
```

Can you modify this code to print out every prime number up to "n"?

Here's how to print out a list of n primes. We'll use the original "isPrime" function and create a

"primeList" function:

```
def primeList(n):
    prime_list = []
    num = 2
    while len(prime_list) < n:
        if isPrime(num):
            prime_list.append(num)
        num += 1
    print(prime_list)
```

To get a list of 10 Primes:

```
>>> primeList(10)
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

Binary Numbers

How can we convert a decimal number to binary?

27	**11011**
decimal	binary

The place values in decimal are ones or 10^0, tens or 10^1, hundreds (10^2) and so on. First we need to find out the largest power of 2 we need to work with. In the case of 27, the we check to see the lowest exponent of 2 that's smaller than 27. 2^5 is bigger so 2^4 (16) is the highest power of 2 we're dealing with. Here's how you check for that in Python.

First loop through the powers of two until the power of two is bigger than the decimal number. Then take away 1 from the exponent:

```
number = 27
exponent = 0
binary_number = 0
while number >= 2**exponent:
    exponent += 1
exponent -= 1
```

"exponent" is now 4, so that's 5 times we have to check powers of 2. If "number" is bigger, we take away that power of 2 and add that power of 10 so there's a 1 in that column.

if 27 is bigger than 2^4, take away 16 (number is now 11) and add 10^4
if 11 is bigger than 2^3, take away 8 (number is now 3) and add 10^3
and so on.

```
for i in range(exponent + 1):
      if number - 2**exponent > -1:
          binary_number += 10**exponent
          number -= 2**exponent
      exponent -= 1
```

And when the loop is done, print out the binary number. Here's the whole code:
Program: Binary converter

```
def binary(number):
    '''Converts decimal number to binary'''
    exponent = 0    #We're dealing with exponents of 2
    binary_number = 0            #The binary form of the number
    while number >= 2**exponent: #Finds the lowest power of 2
        exponent += 1            #the number is less than
    exponent -= 1
    for i in range(exponent + 1):
        if number - 2**exponent > -1: #If number contains power of 2
            binary_number += 10**exponent #Add that power of 10
            number -= 2**exponent  #Take away that power of 2 from
                                   #number
        exponent -= 1              #Next lower exponent
    return binary_number
```

What's the number 30 in binary?

```
>>> binary(30)
11110
```

Yes, because 16(1) + 8(1) + 4(1) + 2(1) = 30

The tools so far

We've made quite a few useful tools in this chapter!

```
equation
pythag
factors
factorPoly
graph
plug
rationalRoots
synthDiv
isPrime
binary
```

Save them to a file called **algebra.py** and we'll be able to easily import them for future use.

Algebra Exercises (Answers on page 141)

In problems #1 - 8, solve the equation for x.

1. $3x - 5 = 34$.
2. $7x + 25 = 2x - 20$
3. $x^2 - 13x + 40 = 0$
4. $5x^2 - 25x - 29 = 0$
5. $12x^3 + 68x^2 - 115x - 21 = 0$
6. $105x^4 + 326x^3 - 369x^2 + 34x + 24 = 0$
7. $315x^6 - 709x^5 + 1870x^4 - 1473x^3 - 5743x^2 + 3976x = 588$
8. $378x^7 - 4737x^6 + 5380x^5 + 6548x^4 - 6439x^3 - 4190x^2 + 1256x + 704 = 0$
9. What is the (base 10) number 44 in binary?

Is 1,000,001 (a million and 1) a prime number? If not, factor it.