In [1]:
```
# Styling notebook
from IPython.core.display import HTML
def css_styling():
    styles = open("./styles/custom.css", "r").read()
    return HTML(styles)
css_styling()
```

Out[1]:

Modular operations addition, multiplication

Addition and multiplication mod m are theoretically easy, but practically a little tricky. The main complication come from the fact that we intend to use modular arithmetic to limit the danger of overflows. The result of an operation "mod m" is known to be between 0 and $m - 1$, but before the mod is taken an overflow *in the arguments* can actually have occured.

In [2]:
```
# modular addition/multiplication (trivial, but a little dangerous - why??)
def addModNotGood(a,b,m) :
    return (a + b)%m

def multModNotGood(a,b,m) :
    return (a*b)%m

print(multModNotGood(34,25,11))
```

```
3
```

Apparently the expressions $a + b$ and $a \times b$ can overflow before the remainder mod m is taken. So we use the idea of replacing every operand o with another value that's congruent to o. Well, $o \mod m$ is probably the safest choice here!

In [3]:
```
# modular addition/multiplication (better)
def addMod(a,b,m) :
    return (a%m + b%m)%m

def multMod(a,b,m) :
    return ((a%m)*(b%m))%m
```

In [4]:
```
multMod(34,25,11)
```

Out[4]:
```
3
```

All good? No!

Our **multMod** has a weird little problem: It's perfectly possible that the sub-expression $(a \mod m) \times (b \mod m)$ overflows if m is in the area of $1/2 \times$ maximal Integer. If that happens, we really need to make sure that we break up this function into one that *a)* is safe(r) and *b)* is still fast.

Here's one that uses addition (considered safe enough) and limited multiplication (by 2) paying for that with a slightly slower runtime:

In [5]:
```
def multModSafe(a,b,m) :
    res = 0
    a = a % m
    while b > 0 :
        # If b is odd, add a to result
        if b%2 == 1 :
            res = (res + a) % m

        # In any case: double a
        a = (a + a) % m

        # Compensate by halving b
        b = b//2
    return res % m
```

In [6]:
```
multModSafe(34,25,11)
```

Out[6]: 3

Invariance under congruence mod m

One of the most important properties of any modular operation is the fact that whenever the modulus is fixed (say m) any operand can be replaced by one that's congruent mod m without changing the result. So, whenever $a \equiv c \pmod{m}$ and $b \equiv d \pmod{m}$:

- $(a + b) \mod m = (c + d) \mod m$
- $(a \times b) \mod m = (c \times d) \mod m$
- $a^b \mod m = c^d \mod m$

In [7]:
```
# Demonstrate we can replace any operand with one that's congruent mod m (a multi
# Here just for multiplication!

print(multModSafe(34,36,11))   # 25 + 11
print(multModSafe(23,25,11))   # 34 - 11
print(multModSafe(23,36,11))   # both
print(multModSafe(-10,135,11)) # 34 - 44, 36 + 99
```

```
3
3
3
3
```

Exponentiation mod m

Watch it! This is probably the most important operation we need. The problem is to compute

$$a^n \mod m$$

quickly and safely.

Now for safety we use the main idea (as always): $a^n \mod m = (a \mod m)^n \mod m$.

1. Recursively (to demonstrate the principle)
2. Principal iterative version with exponential speed-up

3. How it's actually used

In [8]:
```
# Just for demonstration - there are better ways to compute that
def expModRec(a,n,m) :
    if n == 0 : return 1
    else      : return (a%m)*expModRec(a,n-1,m) % m
```

In [9]:
```
expModRec(123,456,987)
```

Out[9]: 267

Fast (?) algorithm for exponentiation mod m - $a^b \mod p$

The calls to "multModSafe" are necessary to avoid the potential overflows in these statements:

1. `res = (res*a)%p`
2. `a = (a*a)%p`

Fast if the standard exponentiation algorithm doesn't use binary speedup

In [10]:
```
def fastExpModManual(a,b,p) :
    res = 1;
    a = a % p;

    if a == 0 : return 0;

    while b > 0 :
        # If b is odd, multiply a with result
        if b % 2 == 1 :
            res = multModSafe(res,a,p); # res = (res*a) mod p
        # b must be even now
        b = b // 2;
        a = multModSafe(a,a,p);         # a = (a*a) mod p
    return res;
```

In [11]:
```
fastExpModManual(123,456,987)
```

Out[11]: 267

The Python pow method has binary speed-up

That's what we're going to use

In [12]:
```
def fastExpMod(x, e, m) :   # Just to make code "portable"
    return pow(x,e,m)       # Well, this thing uses binary templates and is A LOT
```

In [13]:
```
fastExpMod(123,456,987)
```

Out[13]: 267

Inverting multiplication (lengthy)

Inverting multiplication (lengthy)

Ok, we started out trying to find something that can replace floating-point arithmetic without all the problems. We're trying to find out if and when we can **divide** in modular arithmetic. That's made a little easier if we look at a special case of division first, namely at $1/a \mod m$.

Solve: $a \times_m x = 1$, calling x the *multiplicative inverse of a mod m* written (little sloppy) a^{-1} or $1/a$.

Two questions arise, of course:

- Does this exist?
- If yes, how can we compute it?

Fact 1: The multiplicative inverse of *a* mod *m* exists only if GCD(a,m) = 1, i.e. they are "relatively prime"

Now that sounds a little weird, but it's actually the case. There's a deeper reason for this which we will find out in a while. First, let's look at two examples:

- a = 2, m = 4, i.e. not relatively prime
  There are just four possible values and we can try them all out: 2×0 = 0, 2×1 = 2, 2×2 = 0, 2×3 = 2 (all mod 4). Apparently none leads to a product of 1.
- a = 3, m = 4, i.e. relatively prime
  3×0 = 0, 3×1 = 3, 3×2 = 2, 3×3 = 1, and we see that 3 is its own inverse!

Fact 2: If a solution to $a \times_m x = 1$ exists, then all numbers congruent x mod *m* are also solutions! The above example shows that we either have no or solutions or infinitely many:

- a = 3, m = 4: …, -9, -5, -1, 3, 7, 11, 15, …
- Example: 3×15 = 45 = 44 + 1 = 11×4 + 1 etc.

Ok, how do we find one solution assuming it exists? Here's where *Bézout's identity* comes back in.

So, given *a* and *m* there are integers *s* and *t* such that $as + mt = gcd(a, m)$. Now since *a* and *m* are relatively prime (else no inverse!) we have:

$$
\begin{aligned}
& as + mt = gcd(a, m) = 1 \\
\rightarrow \quad & as = 1 - mt \\
\rightarrow \quad & as \equiv 1 \pmod{m} \\
\rightarrow \quad & a \times_m s = 1 \\
\rightarrow \quad & s \text{ is a modular inverse of } a
\end{aligned}
$$

We just need to use the *Extended Euclidian Algorithm* and take one final remainder mod *m* to make sure the result is really in the range of remainders mod *m*.

In [14]:
```
from Numbers import xgcd

# Solves (a*x) = 1 mod m for x
def modInverse(a,m) :
    _, s, _ = xgcd(a,m)
    return s%m

modInverse(3,4)
```

Out[14]: 3

Now you can use the modular inverse if it exists, to divide mod a number that's relative prime to the divisor:

In [15]:
```
# Compute (a/b) mod p for p relative prime to be (usually p is a prime)
# We're not testing that
# USE AT YOUR OWN RISK!
def divMod(a,b,p) :
    return multMod(a,modInverse(b,p),p)
```

In [16]:
```
divMod(12,11,17)
```

Out[16]:
```
15
```

In [17]:
```
# Check
multModSafe(11,15,17)
```

Out[17]:
```
12
```

Discrete Logarithm: Solve $y = a^b$ mod p for b

It's named in analogy to the (well-known?) continuous logarithm defined to be the solution x to $a^x = b$. This discrete analogon is one of the interesting "trap door" functions that protect specific cryptographic methods - here specifically the *Diffie-Helman Key Exchange*.

Now, that's tricky! There is at the moment no efficient algorithm to do that, so here we "brute force" it, by trying exponentiation $y = a^b \mod p$ with b = 0, 1, 2, 3, ... until it solves

Only good for small p, of course!

In [18]:
```
def dLog(a,p,y) :
    b = 0
    while fastExpMod(a,b,p) != y : # brute force
        b += 1
    return b
```

In [23]:
```
y = 17
print(y)
print(dLog(23,19,y))
print(fastExpMod(23,5,19))
```

```
17
5
17
```