In [2]:
```
# Styling notebook
from IPython.core.display import HTML
def css_styling():
    styles = open("./styles/custom.css", "r").read()
    return HTML(styles)
css_styling()
```

Out[2]:

Integer Factorization

This means: Given an integer n, find another integer that divides n. If this other integer is a prime number, we refer to this as *prime factorization.*

Now this problem is acknowledged to be difficult, it's actually one part of a "trap door" scheme (Multiplication: Easy, Factorization: Difficult).

So this is just a simplified demo

The "primeFactor" function runs through a list of "primes" until it finds a divisor or doesn't (it actually checks for divisibility by each member of this list - so we just assume they're all primes, generated by a sieve or similar algorithm)

In [3]:
```
import Primes
from math import sqrt

# Extract the first element of "primes" that divides n
# If that IS a list of primes and it finds a divisor,
# then it finds a prime factor of n - BUT ONLY THEN!
def primeFactor(n,primes) :
    lim = (int)(sqrt(n))
    index = 0
    while primes[index] <= lim :        # brute force trial division by primes
            if n%primes[index] == 0 :
                return primes[index]    # got one!
            index += 1
    return n                            # looks prime - but maybe we rean out of

primesList = Primes.sieve(250000)
primeFactor(1234567,primesList)
```

Out[3]: 127

Generate two random primes and recover the smaller one via "primeFactor" - that'll give us the other, too

Compare the notes about "randomness" in the definition of "**randomPrime**" - it's really not that easy in practice!

In [4]:
```
p = Primes.randomPrime(primesList)
q = Primes.randomPrime(primesList)
n = p*q
print("p =",p,"q =",q,"n =",n)
print("One factor of n =",primeFactor(n,primesList))
```

```
p = 138821 q = 157489 n = 21862780469
One factor of n = 138821
```

The Fundamental Theorem of Arithmetic

Every integer n has a *unique* prime factorization

$$n = p_1^{k_1} \times p_2^{k_2} \times \ldots \times p_r^{k_r},$$

where the p_i are *distinct* prime numbers and the k_i are exponents that count how often this prime appears in the factorization.

Example: $106470 = 2^1 \times 3^2 \times 5^1 \times 7^1 \times 13^2$

In [5]:
```
# Compute all prime factors (even multiples!) and return a list of them
def allFactors(n,primes) :
    result = []
    while n != 1 :
        p = primeFactor(n,primes) # Find a prime factor
        if p == 1 : break         # None found --> Done!

        # Divide by p as long as possible
        count = 1
        n = n//p                  # Should work once at least!
        while n % p == 0 :        # Try over and increment the count
            count += 1            # each time it divides n
            n = n//p              # This count is the exponent

        result.append([p,count])  # [p,k] stands for p^k

    return result
```

In [6]:
```
k = 2*3*3*5*7*13*13
print(k)
allFactors(k,primesList)
```

```
106470
```

Out[6]:
```
[[2, 1], [3, 2], [5, 1], [7, 1], [13, 2]]
```

Specialized stuff (only really needed if you're curious)

Euler's **totient function** $\phi(n)$ is a central part of some cryptographic systems, most prominently RSA. It counts the positive integers less than a given integer n that are relatively prime to n.

Examples:

- $\phi(7) = 6$, since $1, 2, 3, 4, 5, 6$ are all relatively prime to 7.

- $\phi(8) = 4$, since $1, 3, 5, 7$ are relatively prime to 8.

It's possible to understand the basics of RSA without the whole theory of ϕ, so you can skip this part, but it's quite interesting to think about how to compute that.

To compute it, there are a couple of important properties (the first one is not terribly difficult to see, the other two are a little tricky):

1. $\phi(p) = p - 1$, if p is a prime number (because every number $<\ p$ has no factor in common with p)
2. $\phi(pq) = \phi(p)\phi(q)$, if p and q are relatively prime.
3. $\phi(p^k) = p^{k-1}(p-1)$, if p is a prime number.

Together with the Fundamental Theorem it's enough to comput $\phi(n)$ for every n :

If $n = p_1^{k_1} \times p_2^{k_2} \times \ldots \times p_r^{k_r}$, then
$\phi(n) =$
$= \ \phi(p_1^{k_1} \times p_2^{k_2} \times \ldots \times p_r^{k_r})$ *Just plug in the prime factors*
$= \ \phi(p_1^{k_1}) \times \phi(p_2^{k_2}) \times \ldots \times \phi(p_r^{k_r})$ *Because of property 2. and the fact that powers of distinct primes are relatively prime*
$= \ p_1^{k_1-1}(p_1 - 1) \times p_2^{k_2-1}(p_2 - 1) \times \ldots \times p_r^{k_r-1}(p_r - 1)$ *Property 3*

In [9]:
```
def eulerPhi(n,primesList) :
    product = 1
    factors = allFactors(n,primesList) # Get prime factors!
    # Each one is a pair [p, e],
    # so we multiply up phi(p^e) = p^(e-1)(p-1) over all pairs
    for pair in factors :
        p = pair[0]
        e = pair[1]
        product *= pow(p,e-1)*(p-1)
    return product

eulerPhi(8,primesList)
# eulerPhi(79437491597,primesList)
```

Out[9]: 4

Since 79437491597 is the product of two primes (because it's been generated like that) we can just factor it into p and q and compute $(p-1)(q-1)$.

That's exactly what the RSA key generation algorithm does!

In [ ]:
```
n = 79437491597
p = primeFactor(n,primesList)
q = n//p
phi = (p-1)*(q-1)
print("n =",n,"p =",p,"q =",q,"phi =",phi)
```