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]:

Primality Testing

This section sketches an interesting idea of an algorithm that takes an integer as input and returns **True** if it's a prime number or **False** if it isn't (meaning, it's composite).

Of course there are again *brute-force* methods to do that, by checking if the input p is even (which makes it composite, assuming it's greater than 2) and else dividing it by every odd number up to $\sqrt{p}$. If any division leaves a zero remainder, p is composite.

This is nice for small inputs, but as soon as we get to larger numbers it will become very tedious.

Are there better ways? Yep!

Fermat's Little Theorem and the Miller-Rabin Primality Test

Fermat's Little Theorem: If p is prime, then $a^{p-1} \equiv 1 \pmod{p}$, where $1 \leq a < p$.

(Proof: https://primes.utm.edu/notes/proofs/FermatsLittleTheorem.html (https://primes.utm.edu/notes/proofs/FermatsLittleTheorem.html))

It might not look like that, but we can use it to develop a pretty efficient primality tester - we just need to "cheat" a little.

Why cheat? Well, note that the theorem assumes that p is prime, so it's not immediately clear how we can use it to deduce primality!

This first version of the so-called *Miller-Rabin Primality Test* is done by using the converse of this condition - which we all know (?) is not correct. But let's try it anyway and see what goes wrong!

All we do is to randomly pick a value for a and see what happens with $a^{p-1} \bmod p$. Maybe it's 1, maybe it isn't.

In [2]:
```
import random
from Modular import fastExpMod

def miller_rabin_1(p) :
    a = random.randrange(2,p - 1)
    return fastExpMod(a,p-1,p) == 1
```

267

In [3]:
```
# If p is a prime, then the test correctly always returns True
# because that's exactly what the Little Fermat says
#
miller_rabin_1(342833) # That's a prime numbeer!
```

Out[3]: True

In [4]:
```
# The tricky part is a composite input value.
# In this case the test can fail, if it stumbles on a value of a
# which gives a 1. Interestingly enough that doesn't happen too often!
#
# 175598789 = 5437 * 32297 (thus composite)
#
print(miller_rabin_1(175598789)) # Generally says False
```

False

Test Failures

Let's have a closer look at why this test can fail. It fails in case of a composite input whenever the algorithm picks a value of a such that $a^{p-1} \equiv 1 \pmod{p}$. This can of course happen, since primality of p is sufficient, but not necessary.

In other words, p might be composite, but an unlucky choice of a might suggest it's prime.

- A value of a such that $a^{p-1} \not\equiv 1 \pmod{p}$ for a composite p is called a *Fermat Witness*.
- A value of a such that $a^{p-1} \equiv 1 \pmod{p}$ for a composite p is called a *Fermat Liar*.

In [5]:
```
# Now if the algorithm had picked a = 157706252,
# the test would have answered True
# because the right side of the theorem would be satisfied:
#
a = 157706252 # Fermat Liar
p = 175598789 # Composite!
print(fastExpMod(a,p-1,p))
```

1

Repetition Legitimizes (Adam Neely)

We don't really know a lot about "Fermat Liars", but we can assume that the test fails for composite numbers in a certain ratio of cases: $r = \frac{n_{\text{fail}}}{n_{\text{total}}} < 1$.

Since this version of the Miller-Rabin Test is not reliably correct, we can use a classical trick, namely to run it a couple of times. We know:

- If the test returns **False**, the number p is certainly composite.
- If the test returns **True**, it's not clear if it's prime or composite.

Assume now that just one of the many executions of this test return "False". That would mean that the number actually **is** composite (else we wouldn't have a **False** at all) and all the **True** results are caused by liars. Unlikely and certainly weird, but not impossible.

Assume now that all executions of the test return **True**.

If p actually **is** composite, then *all* results are generated by liars. Also, not impossible.

Assuming the failure rate r from above, that would mean that after n executions which all (falsely) returned **True** the ratio of total failures would now be r^n which is getting smaller when n increases.

So, a large number of runs which all return **True** makes it more and more unlikely that they're all liars.

We would need a deeper analysis of the problem of Fermat Liars in order to get to a better estimate about the reliability of the modified test, so let's just state the main result:

The probability that the Miller-Rabin Test fails after n executions is $\leq 1/2^n$.

In summary, this makes the quality of the test a matter of *probability*, not *strict provability*.

In [6]:
```
def miller_rabin(p,n) :
    # Execute the basic test n times or until it returns False
    result = miller_rabin_1(p)
    count = 1
    while count < n and result == True :
        result = miller_rabin_1(p)
        count += 1
    # Assert: count >= n OR result == False
    return result
```

In [7]:
```
print(miller_rabin(342833,100))        # Prime
print(miller_rabin(175598789,100))     # Composite
```

```
True
False
```

A last word

Miller-Rabin provides a useful example of a *probabilistic algorithm*, that is an algorithm whose measure of correctness is based on probabilities, rather than logical reasoning. For a long time it was unclear if there is a *non-probabilistic, correct* and *efficient* primality testing algorithm. Specifically, the notion of "efficient" needs to be clarified before even thinking about that.

This ended in 2002 with the publication of the Agrawal–Kayal–Saxena primality test (https://en.wikipedia.org/wiki/AKS_primality_test), short AKS primality test. It showed that primality testing can be done efficiently - unfortunately it's a little too complicated to be included here.