In [36]:
```
# Styling notebook
from IPython.core.display import HTML
def css_styling():
    styles = open("./styles/custom.css", "r").read()
    return HTML(styles)
css_styling()
```

Out[36]:

Number Theory Motivation: Limited size for standard number representations

From any introductory programming course it’s well-known that the two standard number representations – for fixed and for floating point numbers – are based on the allocation of a fixed, finite amount of storage for a number. That’s usually measured in bits, such 16 bit, 32 bit, 64 bit and so on.

We know that fixed point numbers, generally called “Integers” (here with a capital “I” to distinguish them from the mathematical “integers” – the numbers in $\mathbb{Z}$) can overflow – either by abruptly changing the sign if they are “signed” or switching from a large value to 0 if they are “unsigned”.

Floating point (“Floats”) have a limited mantissa space and limited exponent space. That results in overflow behavior and additionally rounding problems if the mantissa space isn’t sufficient.

Both aspects of these representations are theoretically and practically problematic – especially floating-point numbers show behavior that doesn’t comply with mathematical principles at all:

1. Even rational numbers are represented in this format, meaning that as soon as a number has a “long” or infinite decimal part, it cannot be represented accurately
2. The “normal” arithmetic rules don’t apply to Floats, at least not ideally. So, numerical stability is a severe issue

See the following examples:

In [37]:
```
aThird = 1/3
print(format(aThird,".30f"))
1/7/7/7/7/7/7/7/7*7*7*7*7*7*7*7*7
```

```
0.333333333333333314829616256247
```

Out[37]: 0.9999999999999998

The first output shows the usual “degradation” of the mantissa due to truncation. The second example shows that this degradation renders normal arithmetic laws to be approximate at best.

The standard “solution” is in general to turn to longer formats and hope for the best.

Yes, there is an Integer "division"

But it's actually not a real division, since it doesn't work with multiplication as it should: $\frac{a}{b} \times b = a$
So, we would expect: $\frac{5}{3} \times 3 = 5$
But the Integer "division" gives us: $\frac{5}{3} \times 3 = 3$

In [38]:
```
print(5//3)
print((5//3)*3)
```

```
1
3
```

Summary 1
So, we're looking for a way to represent numbers that obeys the standard rules of arithmetic. We know Floats don't work, and Integers are too restricted.

Yes, you can add, subtract and multiply integers (ignoring overflows for the moment), these operations are

- commutative: $m + n = n + m$ and $m \times n = n \times m$
- associative: $(m + n) + o = m + (n + o)$ and $(m \times n) \times o = m \times (n \times o)$
- distributive: $m \times (n + o) = m \times n + m \times o$

Also, there are two distinguished numbers 0 and 1:

- $0 \neq 1$
- $n + 0 = n$ and $n \times 1 = n$ (the *neutral elements* for addition and multiplication)

Furthermore, the *additive inverse* exists which means that the equation $m + x = 0$ has a solution in the integers $(-m)$

Unfortunately, the *multiplicate inverse* doesn't exist, since the equation $m \times x = 1$ in general doesn't have an integer solution $(1/m)$

Note that the multiplicative inverse exists in the rational numbers $\mathbb{Q}$, but that's not much help, because we want to avoid them.

Since division is the "weak spot" of integers, let's see how far division can bring us.

Elementary integer algorithms 1: Division

Assuming that we know how addition, subtraction and multiplication work in the integers, let's have a look how this integer "division" works and find out if it's really that bad.

In general for integers a and b the expression a/b has two results (and you know them already)

$a/b = q$ with remainder r
q: quotient
r: remainder $(0 \leq r < b)$

Together: $a = q \times b + r, 0 \leq r < b$

Assuming a and b both being non-negative (unsigned), the following algorithm does that:

In [39]:
```
# unsigned integer division algorithm
# a,b (a >= 0, b > 0) --> (q,r), a = q*b + r and 0 <= r < b
def intDivU(a,b) :
    assert a >= 0 and b > 0, "illegal args to unsigned division"
    q = 0
    r = a
    while r >= b :
        q += 1
        r -= b
    return q,r
```

In [40]:
```
intDivU(5,3)
```

Out[40]:
```
(1, 2)
```

Of course, in most programming languages q is the result of the integer division and r is the remainder or **mod** function. So, don't ever implement this **intDivU** function!

We write the remainder of a under division by b as $a \mod b$. In most programming languages the percent operator is used: a%b.

In [4]:
```
5//3, 5%3
```

Out[4]:
```
(1, 2)
```

We can now define what it means to say "a divides b": it means the remainder of b when divided by a is 0:

In [51]:
```
def divides(a,b) :
    return b%a == 0

divides(5,36)
```

Out[51]:
```
False
```

Two numbers a and b are called "congruent mod m" $a \equiv b \pmod{m}$ (Don't worry about the term, this "mod" will appear more often from now on) iff

- m divides $(a - b)$ or
- a mod m = b mod m (a and b have the same remainder under m)

Important: It's pretty easy to see if two numbers are congruent given m : They need to be a multiple of m apart, that's all!

In [3]:
```
# congruence mod m
def congMod(a,b,m) :
    return (a-b)%m == 0

congMod(5,38,11) # 38 - 5 = 33 which is a multiple of 11
```

Out[3]:
```
True
```

The **greatest common divisor (GCD)** of two integers, which are not both zero, is the largest positive integer that divides both integers.

The Euclidean algorithm is based on the principle that the greatest common divisor of two numbers does not change if the larger number is replaced by its difference with the smaller number.

In [43]:
```
# Euclidean algorithm
# Don't use this if you don't have to.
#   It's built into Python
def gcd(a,b) :
    if a == 0 : return b
    if b == 0 : return a
    return gcd(b%a, a)
```

In [54]:
```
gcd(12345, 67890)
```

Out[54]:
```
15
```

Extended Euclidean Algorithm (needed a little later)

An extension to the Euclidean algorithm that computes, in addition to the greatest common divisor (gcd) of a and b, also the coefficients of *Bézout's identity*, which are integers s and t such that

$$as + bt = gcd(a, b)$$

Example: $a = 12, b = 30, \gcd(12, 30) = 6, s = -2, t = 1$
Since $12 \times (-2) + 30 \times 1 = 6$

The **Extended Euclidian Algorithm** produces these coefficients together with the GCD: [https://www.mauriciopoppe.com/notes/mathematics/number-theory/extended-euclidean-algorithm/ (https://www.mauriciopoppe.com/notes/mathematics/number-theory/extended-euclidean-algorithm/)](https://www.mauriciopoppe.com/notes/mathematics/number-theory/extended-euclidean-algorithm/)

In [45]:
```
# Extended Euclidean Algorithm
# Returns gcd(a,b), s and t (Bezout coefficients )
def xgcd(a,b) :
    prevx, x = 1, 0; prevy, y = 0, 1
    while b:
        q = a//b
        x, prevx = prevx - q*x, x
        y, prevy = prevy - q*y, y
        a, b = b, a % b
    return a, prevx, prevy
```

In [55]:
```
xgcd(12,30)
```

Out[55]:
```
(6, -2, 1)
```