In [3]:
```
# Styling notebook
from IPython.core.display import HTML
def css_styling():
    styles = open("./styles/custom.css", "r").read()
    return HTML(styles)
css_styling()
```

Out[3]:

Prime numbers

Prime numbers are some of the most "elementary" integers in the sense that any integer is somehow composed of primes. They also play a decisive role in today's cryptography - generally since they facilitate "complete arithmetic" by allowing all four arithmetic operations (addition, subtraction, multiplication and division) when used creatively.

There are a lot of questions as far as prime numbers are concerned:

- Are the infinitely many? (Yes)
- How can I find out if a number is prime? (It's complicated)
- How can I generate prime numbers? (Also complicated, but there is a simple way that just takes a lot of time)

Testing if a number is prime is not overly important for the rest of this module, so it's going to be skipped for the moment. In case you're curious please refer to the "Miller-Rabin" section.

We start with the last problem which has a solution dating back to around 200 B.C.

Prime Generation with the "Sieve of Eratosthenes"

A relatively simple iterative process, that starts with a list of integers and successively crosses out all composite numbers up to a given value n :

1. Start with $k = 2$. In the next steps find the next k that is not crossed out (that's a prime)
2. If $k > \sqrt{n}$, we're done - nothing more to cross out
3. Cross out all multiples of k up to the limit n.
4. Find the next k that is not crossed out (that's a prime) and continue with that.

Standard implementation over a boolean array (**True** means "it's a prime"). Initialize to all **True** and set any multiple of k to **False**. What remains **True** to the end are the primes.

In [4]:
```
def sieve(n) :
    sieve = [True for i in range(n + 1)]
    k = 2
    while (k * k <= n):
        # Find next prime - mark all multiples as not prime
        if (sieve[k]) :
            for i in range(k*k, n + 1, k):
                sieve[i] = False
        k += 1

    # Collect/return primes off the sieve
    primes = []
    for p in range(2, n+1):
        if sieve[p] :
            primes.append(p)
    return primes
```

In [32]:
```
print(sieve(2000))
```

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 7
3, 79, 83, 89, 97, 101, 103, 107, 109, 113, 127, 131, 137, 139, 149, 151, 157,
163, 167, 173, 179, 181, 191, 193, 197, 199, 211, 223, 227, 229, 233, 239, 241,
251, 257, 263, 269, 271, 277, 281, 283, 293, 307, 311, 313, 317, 331, 337, 347,
349, 353, 359, 367, 373, 379, 383, 389, 397, 401, 409, 419, 421, 431, 433, 439,
443, 449, 457, 461, 463, 467, 479, 487, 491, 499, 503, 509, 521, 523, 541, 547,
557, 563, 569, 571, 577, 587, 593, 599, 601, 607, 613, 617, 619, 631, 641, 643,
647, 653, 659, 661, 673, 677, 683, 691, 701, 709, 719, 727, 733, 739, 743, 751,
757, 761, 769, 773, 787, 797, 809, 811, 821, 823, 827, 829, 839, 853, 857, 859,
863, 877, 881, 883, 887, 907, 911, 919, 929, 937, 941, 947, 953, 967, 971, 977,
983, 991, 997, 1009, 1013, 1019, 1021, 1031, 1033, 1039, 1049, 1051, 1061, 106
3, 1069, 1087, 1091, 1093, 1097, 1103, 1109, 1117, 1123, 1129, 1151, 1153, 116
3, 1171, 1181, 1187, 1193, 1201, 1213, 1217, 1223, 1229, 1231, 1237, 1249, 125
9, 1277, 1279, 1283, 1289, 1291, 1297, 1301, 1303, 1307, 1319, 1321, 1327, 136
1, 1367, 1373, 1381, 1399, 1409, 1423, 1427, 1429, 1433, 1439, 1447, 1451, 145
3, 1459, 1471, 1481, 1483, 1487, 1489, 1493, 1499, 1511, 1523, 1531, 1543, 154
9, 1553, 1559, 1567, 1571, 1579, 1583, 1597, 1601, 1607, 1609, 1613, 1619, 162
1, 1627, 1637, 1657, 1663, 1667, 1669, 1693, 1697, 1699, 1709, 1721, 1723, 173
3, 1741, 1747, 1753, 1759, 1777, 1783, 1787, 1789, 1801, 1811, 1823, 1831, 184
7, 1861, 1867, 1871, 1873, 1877, 1879, 1889, 1901, 1907, 1913, 1931, 1933, 194
9, 1951, 1973, 1979, 1987, 1993, 1997, 1999]
```

Pick a random prime out of the list

Not quite as easy as it looks, since we need to pick them in a particular way to make brute force attempts difficult. They should be away from the "sides" of the interval so they cannot be found by exhaustive search from either side - but they shouldn't be too close together.

Also, the random number generators in normal programming languages are (at least theoretically) predictable - we don't want that. Cryptographically acceptable random numbers are tough to generate!!

In [6]:
```
primesList = sieve(250000)
```

In [7]:
```
import random

def randomPrime(primes, start = 1) :
    index = random.randrange(start,len(primes)) - 1
    return primes[index + start - 1]
```

In [29]:
```
randomPrime(primesList)
```

Out[29]:
```
16481
```