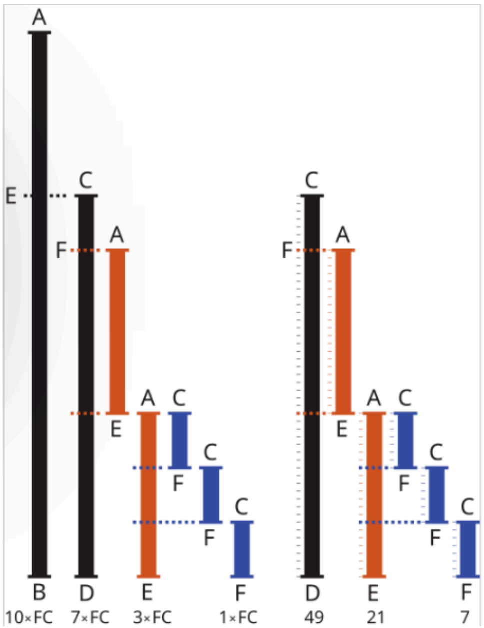
Week 2: Math Behind Cryptography

Euclid's algorithm



CODE :

```python
def findGCD(a, b):
    if a == 0:
        return b
    return findGCD(b % a, a)
```

Extended GCD

Let a and b be positive integers.
The extended Euclidean algorithm is an efficient way to find integers u,v such that

$$a \cdot u + b \cdot v = \gcd(a,b)$$

Mathematical Working for finding gcd, x, y:

$ax + by = gcd(a, b)$

$gcd(a, b) = gcd(b\%a, a)$

$gcd(b\%a, a) = (b\%a)x1 + ay1$

$ax + by = (b\%a)x1 + ay1$

$ax + by = (b - [b/a] * a)x1 + ay1$

$ax + by = a(y1 - [b/a] * x1) + bx1$

Comparing LHS and RHS,

$x = y1 - \lfloor b/a \rfloor \lfloor b/a \rfloor * x1$

$y = x1$

CODE :

```python
def extended_gcd(a, b):
    if b == 0:
        return a, 1, 0  # gcd, x, y
    else:
        gcd, x1, y1 = extended_gcd(b, a % b)
        x = y1
        y = x1 - (a // b) * y1
        return gcd, x, y
```

Modular Arithmetic

```python
def modulo(a,b):
    return a % b
```

Fermat's Little Theorem

states that if p is a prime number, then for any integer a, the number $a^p - a$ is an integer multiple of p.

i.e, $a^p \equiv a \pmod{p}$

CODE :

```python
def mod_inverse(a, m):
    gcd, x, _ = extended_gcd(a, m)
    if gcd != 1:
        return None  # Inverse doesn't exist if a and m are not coprime
    else:
        return x % m  # Ensure the result is positive
```

Why it works?

For two numbers `a` and `m`, the modular inverse of `a` modulo `m` is a number `x` such that:

$$a \cdot x \equiv 1 \bmod m$$

This means: $a \cdot x - 1$ is divisible by m

Or: $a \cdot x + m \cdot y = 1$

(for some integer `y`)

This equation is exactly what the Extended Euclidean Algorithm solves.

1. `extended_gcd(a, m)` gives you `x` and `y` such that:
   $a \cdot x + m \cdot y = \gcd(a, m)$
2. If `gcd = 1`, then `x` is the **modular inverse** of `a` modulo `m`.
3. But `x` can be **negative**, so we return `x % m` to make it positive.

Quadratic Residue

If a and m are coprime integers, then a is called a quadratic residue modulo m if the congruence

$x^2 \equiv a \pmod{n}$ has a solution.

Likewise, if it has no solution, then it is called a quadratic non-residue modulo m.

CODE :

```python
def is_quadratic_residue(a, n):
    for x in range(n):
        if (x * x) % n == a % n:
            return x
    return 0
```

Quadratic Residue * Quadratic Residue = Quadratic Residue
Quadratic Residue * Quadratic Non-residue = Quadratic Non-residue
Quadratic Non-residue * Quadratic Non-residue = Quadratic Residue

Legendre Symbol

The *Legendre Symbol* gives an efficient way to determine whether an integer is a quadratic residue modulo an odd prime p.

Legendre's Symbol: $(a/p) \equiv a^{(p-1)/2} \bmod p$ obeys:

$$(a/p) = 1 \text{ if } a \text{ is a quadratic residue and } a \not\equiv 0 \mod p$$
$$(a/p) = -1 \text{ if } a \text{ is a quadratic non-residue } \mod p$$
$$(a/p) = 0 \text{ if } a \equiv 0 \mod p$$

Shank Tonelli's Algorithm

Shank Tonelli's algorithm works for all types of inputs.
Algorithm steps to find modular square root using shank Tonelli's algorithm :

1) Calculate $n^{(p-1)/2}$ (mod p), it must be 1 or p-1, if it is p-1, then modular square root is not possible.
2) Then after write p-1 as $(s * 2^e)$ for some integer s and e, where s must be an odd number and both s and e should be positive.
3) Then find a number q such that $q^{(p-1)/2}$ (mod p) = -1
4) Initialize variable x, b, g and r by following values

```
x = n ^ ((s + 1) / 2 (first guess of square root)
b = n ^ s
g = q ^ s
r = e   (exponent e will decrease after each updation)
```

5) Now loop until m > 0 and update value of x, which will be our final answer.

```
Find least integer m such that b^(2^m) = 1(mod p)  and  0 <= m <= r - 1
If m = 0, then we found correct answer and return x as result
Else update x, b, g, r as below
    x = x * g ^ (2 ^ (r - m - 1))
    b = b * g ^(2 ^ (r - m))
    g = g ^ (2 ^ (r - m))
    r = m
```

CODE :

```
def legendre_symbol(a, p):

    a = a % p

    if a == 0:
        return 0
    elif pow(a, (p - 1) // 2, p) == 1:
        return 1
    else:
        return -1

def tonelli_shanks(a, p):
    if legendre_symbol(a, p) != 1:
        return None

    if p % 4 == 3:
```

```
        x = pow(a, (p + 1) // 4, p)
        return x

    q, s = p - 1, 0
    while q % 2 == 0:
        q //= 2
        s += 1

    z = 2
    while legendre_symbol(z, p) != -1:
        z += 1

    c = pow(z, q, p)
    x = pow(a, (q + 1) // 2, p)
    t = pow(a, q, p)
    m = s

    while t != 1:
        i = 1
        temp = pow(t, 2, p)
        while temp != 1:
            temp = pow(temp, 2, p)
            i += 1
            if i == m:
                return None
        b = pow(c, 2 ** (m - i - 1), p)
        x = (x * b) % p
        t = (t * b * b) % p
        c = (b * b) % p
        m = i

    return x

for value in ints:
    if legendre_symbol(value, p) == 1:
        root = tonelli_shanks(value, p)
```

```
if root is not None:
    print(f"x ≡ {root} mod p")
```

Chinese Remainder Theorem

states that there always exists an x that satisfies given congruences.Let num[0], num[1], ...num[k-1] be positive integers that are pairwise coprime. Then, for any given sequence of integers rem[0], rem[1], ... rem[k-1], there exists an integer x solving the following system of simultaneous congruences.

$$\begin{cases} x \equiv rem[0] & \pmod{num[0]} \\ \quad \dots & \\ x \equiv rem[k-1] & \pmod{num[k-1]} \end{cases}$$

Furthermore, all solutions x of this system are congruent modulo the product, prod = num[0] * num[1] * ... * nun[k-1]. Hence

$$x \equiv y \pmod{num[i]}, \quad 0 \le i \le k-1 \qquad \Longleftrightarrow \qquad x \equiv y \pmod{prod}.$$

CODE :

```
def chinese_remainder_theorem(a, m):
    M = 1
    for mod in m:
        M *= mod

    x = 0
    for ai, mi in zip(a, m):
        Mi = M // mi
        yi = mod_inverse(Mi, mi)
        x += ai * Mi * yi

    return x % M
```

Week 2 : Resources Used - [Cryptohack Math Module](#) | GeeksforGeeks