Phi (Φ) function - Euler's totient

Phi (Φ) function, also known as Euler's totient function, is an arithmetic function that counts the positive integers up to a given integer n that are relatively prime to n. In other words, the function returns the number of integers from 1 to n-1 that have no common factor with n other than 1.

The phi function is written as Φ(n) or ϕ(n). Here are the mathematical formulas to calculate phi function:

1. If p is a prime number, then $\Phi(p) = p - 1$
2. If p and q are distinct prime numbers, then $\Phi(pq) = (p - 1)(q - 1)$
3. For a general n, let p1, p2, ..., pm be the distinct prime factors of n. Then,

$$\Phi(n) = n * (1-1/p1) * (1-1/p2) * ... * (1-1/pm)$$

Example of JavaScript code to calculate phi function Φ(n):

```
// Compute phi function Φ(n)
function phi(n) {
    let result = n; // Initialize result with n

    // Check for all prime factors smaller or equal to sqrt(n)
    for (let i = 2; i*i <= n; i++) {
        if (n % i == 0) {
            while (n % i == 0) {
                n /= i;
            }
            result -= (result / i);
        }
    }

    // If n has a prime factor greater than sqrt(n)
    if (n > 1) {
        result -= (result / n);
    }

    return result;
}
```

Note:

This implementation of phi function uses a well-known algorithm called Euler’s Totient Function Formula.

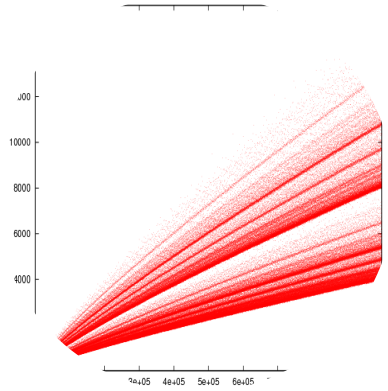
Funny note:

Christian Goldbach in a letter to the Euler make a discussion. The "Goldbach Conjecture" read more (https://en.wikipedia.org/wiki/Goldbach%27s_conjecture)

- $2m = p + q$
- $p \geqq m$
- $q \leqq m$

He calculate the chance of q & p can be prime number ...

Calculation power speed = ($\ln n \ll \sqrt{n}$) {i think: n log n^2, not sure}



ECC (Elliptic Curve Cryptography)

ECC (Elliptic Curve Cryptography) is a public-key encryption algorithm used to secure data transmission over networks. ECC is based on the mathematical concept of elliptic curves and it offers the same level of security as RSA and other public-key encryption algorithms, but with much shorter key lengths.

The ECC algorithm involves the following steps:

1. Key Generation:
    - Choose a random elliptic curve over a finite field of prime order
    - Select a point P on the elliptic curve as the base point
    - Choose a private key (a random integer) d
    - Calculate the public key (a point on the elliptic curve) Q = dP

    Public key: The point Q

    Private key: The integer d

2. Encryption:
    - Choose a random integer k
    - Calculate the elliptic curve point C1 = kP
    - Calculate the shared secret s = kQ
    - Select a symmetric encryption algorithm (for example, AES)

- Encrypt the plaintext with the shared secret using the symmetric encryption algorithm
- Attach C1 and the encrypted ciphertext to the message

3. Decryption:

- Given C1, calculate s = dC1
- Decrypt the ciphertext using the shared secret s

The mathematical formulas used in ECC algorithm:

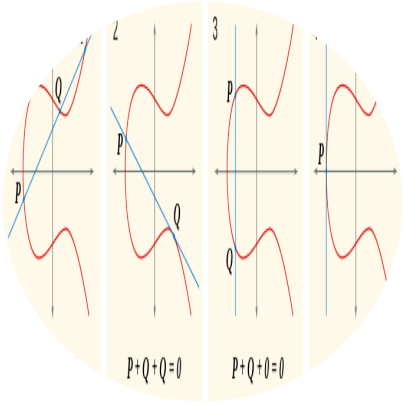
- Elliptic Curves: An elliptic curve is a curve that satisfies a specific mathematical equation. In ECC, the curve is defined over a finite field of prime order.
- Base Point (P): A point on the elliptic curve that is used as the basis for the elliptic curve calculations.
- Point Addition: ECC uses a point addition operation for addition of two points on the curve.
- Scalar Multiplication: ECC uses scalar multiplication for generation of public key from private key.

Example of JavaScript code for ECC Encryption and Decryption:

```
const crypto = require('crypto');
const eccrypto = require('eccrypto');

// ECC Key Generation
async function eccKeyGen() {
  const { publicKey, privateKey } = await eccrypto.generateKeypair();
  return { publicKey, privateKey };
}

// ECC Encryption
async function eccEncrypt(publicKey, plaintextMsg) {
  const plaintextBuffer = Buffer.from(plaintextMsg, 'utf8');
  const k = crypto.randomBytes(32); // Generate random key
  const ciphertext = await eccrypto.encrypt(publicKey, plaintextBuffer, [{ k }]);
  return ciphertext;
}

// ECC Decryption
async function eccDecrypt(privateKey, ciphertextMsg) {
  const decrypted = await eccrypto.decrypt(privateKey, ciphertextMsg);
  return decrypted.toString('utf8');
}
```

Note:

This implementation of ECC (Elliptic Curve Cryptography) formula.



Fun

```
y^2 = x^3 + 3x + b (mod p)
```

- p = the hours on the clock (generate by phi p & phi q) "phi = e = euler, read my article 04-phi-euler.md in same repo"
- b = random big num
- x, y = prime number (phi p & phi q)
- phi: p, q = have x & y axis on the curve (they are use in phi for generating randoms)

this is the ecc revers engineering!
in fact you can create a 4096 encryption machine less then 100$

Conjectur: elliptic curve is a 2d of inside the torus. explain that to your-self by imagin the shape in 3dimensions!
This conjecture is a *theorem* asap!
The way to create un-crackable cryptography...

HMAC - KDF

A **Key Derivation Function (KDF)** is a cryptographic algorithm used to derive one or more keys from a single secret value, such as a password or a seed. KDFs are designed to be computationally expensive, which makes them more resistant to brute-force attacks and other methods of cryptographic analysis.

The **HMAC (Keyed-Hash Message Authentication Code)** algorithm is a specific type of KDF that is used to verify the integrity and authenticity of messages. HMAC is based on a hash function, which generates a fixed-size output from a variable-length input. The hash function used in HMAC is typically a secure hash function like SHA-256 or SHA-512.

Here's the mathematical equation for HMAC:

```
HMAC(key, message) = H[(key XOR opad) || H((key XOR ipad) || message)]
```

In this equation, "key" is the secret key used to encrypt the message, and "message" is the message to be encrypted. The "opad" and "ipad" values are padded versions of the secret key, and "H" is the hash function used to encrypt the message.

To use HMAC in JavaScript, we can use the built-in Node.js crypto library. Here's an example:

```
// nodejs
const crypto = require('crypto');

const key = 'mysecretkey';
const message = 'Hello, world!';

const hmac = crypto.createHmac('sha256', key);
hmac.update(message);

console.log(hmac.digest('hex'));
```

In this example, we use the `crypto.createHmac()` method to create a new HMAC object using the SHA-256 hash function and the secret key "mysecretkey". We then use the `hmac.update()` method to add the message "Hello, world!" to the HMAC, and the `hmac.digest()` method to generate the final HMAC value in hexadecimal format.