RSA

Part 1 Mathematics basis

Euclidean algorithm

- given two number r_0, r_1, find the $\gcd(r_0, r_1)$
- The idea behind euclidean algorithm is to reduce (r_0, r_1) to two small number step by step

$$\begin{aligned}\gcd(r_0, r_1) &= \gcd(r_0 \% r_1, r_1) \\ &= \gcd(r_1, r_0 \% r_1)\end{aligned}$$

- Python implementation

```python
import math
import random
import time
def gcd(r0 , r1 ):
    while r1:
        r0 , r1 = r1 , r0 % r1
    return r0

while True:
    r0 , r1 = random.randint(1,10000) ,
random.randint(1,10000)
    print(gcd(r0,r1) == math.gcd(r0 ,r1))
    time.sleep(1)
```

Extended Euclidean Algorithm

- find s, t , which make that $\gcd(r_0, r_1) = s \times r_0 + t \times r_1$
- the idea behind extended euclidean algorithm is calculate a 'extended' equation along with the gcd

$$\gcd(r_0, r_1) = s_0 \times r_0 + t_0 \times r_1$$

$$\gcd(r_1, r_0 \% r_1) = s_1 \times r_0 + t_1 \times r_1$$

$$\ldots\ldots$$

- How to do this ? express the new reminder in r_0, r_1

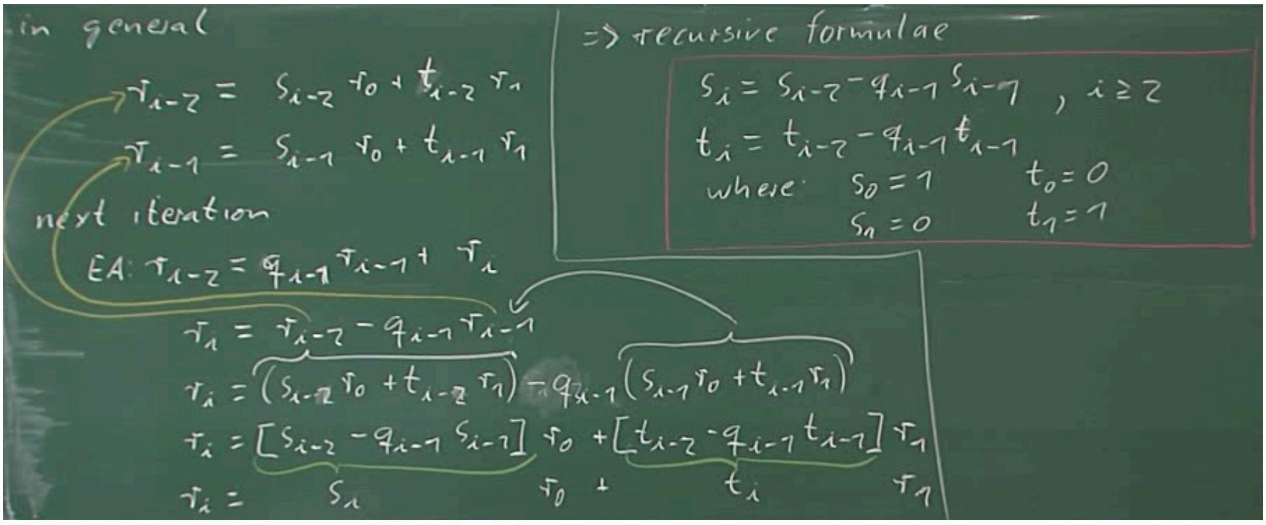
- You just need the line above and the line above above(recursion comes here)
- example: xgcd(973,301)

$$973 = 3 \times 301 + 70, 70 = r_0 + (-3)r_1$$

$$301 = 4 \times 70 + 21, 21 = r_1 + (-4) \times 70 = 13r_1 + (-4)r_0$$

$$70 = 3 \times 21 + 7, 7 = 70 + (-3) \times 21 = 13r_0 + (-42)r_1$$

- General formula



- Application : you can use xgcd to find multiply inverse

$$a \cdot a^{-1} \equiv 1 \mod n$$

$$\gcd(a, n) = sa + tn = 1$$

$$(s\%n)a \equiv 1 \mod n$$

$$a^{-1} = s\%n$$

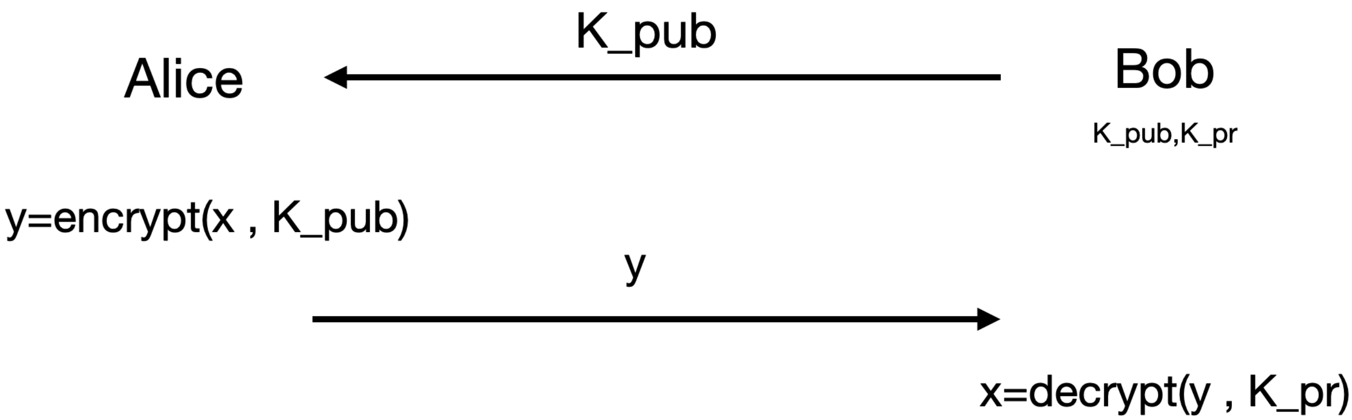
Some theorems

- Euler's Phi Function

> given integer m, how many numbers are relative prime to m in $Z_m = \{0, 1, 2, \ldots, m-1\}$
>
> $$\begin{aligned} \varphi(n) &= \varphi(p_1^{k_1})\,\varphi(p_2^{k_2})\cdots\varphi(p_r^{k_r}) \\ &= p_1^{k_1-1}(p_1-1)\,p_2^{k_2-1}(p_2-1)\cdots p_r^{k_r-1}(p_r-1) \\ &= p_1^{k_1}\left(1-\frac{1}{p_1}\right)p_2^{k_2}\left(1-\frac{1}{p_2}\right)\cdots p_r^{k_r}\left(1-\frac{1}{p_r}\right) \\ &= p_1^{k_1}p_2^{k_2}\cdots p_r^{k_r}\left(1-\frac{1}{p_1}\right)\left(1-\frac{1}{p_2}\right)\cdots\left(1-\frac{1}{p_r}\right) \\ &= n\left(1-\frac{1}{p_1}\right)\left(1-\frac{1}{p_2}\right)\cdots\left(1-\frac{1}{p_r}\right). \end{aligned}$$

- Fermat's little theorem
- Euler's theorem

Part 2 Introduction to Public Key



RSA Algorithm

Key Generation

*Unlike symmetric key system, PK system require the **computation** of the pair (K_pub,K_pr)*

1. choose **large prime** p, q
2. Compute $n = p \times q$
3. $\varphi(n) = \varphi(p) \times \varphi(q) = (p-1)(q-1)$
4. choose $K_{pub} = e$ from set $\{1, 2, 3, \ldots, \varphi(n)\}$, such that $\gcd(e, \varphi(n)) = 1$ (the existence of inverse of e modulo $\varphi(n)$)
5. compute $K_{pr} = d$,such that $e \cdot d \equiv 1 \mod \varphi(n)$

$$K_{pub} = (n, e), K_{pr} = (n, d)$$

- usually $p \geq 2^{512}, q \geq 2^{512}$ so that $n \geq 2^{1024}$, and when we talk about the length of RSA, we are referring length of n

Encryption and Decryption

Encryption

given $K_{pub} = (n, e)$, message x ($x \in Z_n, x \in \{0, 1, 2, \ldots, n-1\}$)

$$y = \text{encrypt}(K_{pub}, x) \equiv x^e \mod n$$

Decryption

given $K_{pr} = (n, d)$, $y \in Z_n$

$$x = \text{decrypt}(K_{pr}, y) \equiv y^d \mod n$$

Example

```
import random
from math import gcd
def xgcd(a:int,b:int)->tuple:
    x0 , x = 1 , 0
    y0 , y = 0 , 1
    r0 , r = a , b
    while r:
        q = r0 // r
        x0 , x = x , x0 - q*x
        y0 , y = y , y0 - q*y
        r0 , r = r , r0 - q*r
    return x0 , y0 , r0

message = 4
p,q = 3 , 11 # choose your prime here



n = p*q
```

```python
phin = (p-1)*(q-1)
e = random.randint(0, phin)
while gcd(e, phin) != 1:
    e = random.randint(0, phin)
d = xgcd(e, phin)[0]

Kpub = (n,e)
Kpr = (n,d)

def encrypt(Kpub:tuple , message):
    n , e = Kpub
    return pow(message, e , n)

def decrypt(Kpr:tuple , cipher):
    n , d = Kpr
    return pow(cipher, d , n)

print(message)
cipher  = encrypt(Kpub, message)
print(cipher)
m = decrypt(Kpr, cipher)
print(m)
```

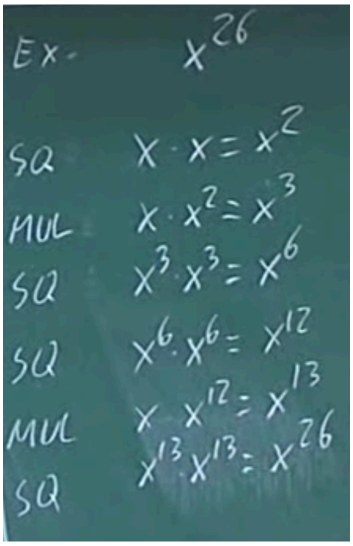
- almost no where else need handling such long number!
- usually the key is only generated once (we store it).

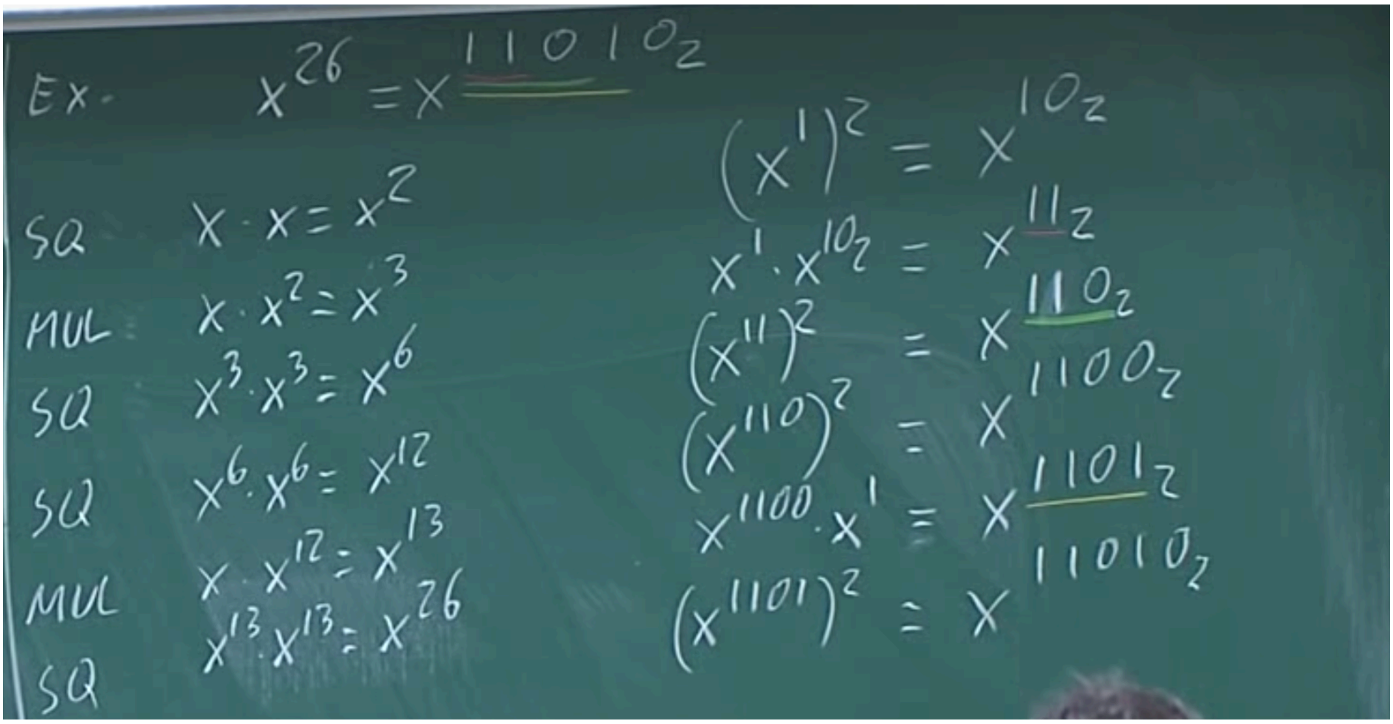
Fast Exponentiation

aka "left-to-right exponentiation" or "square and multiplication exponentiation"

The idea : we prefer exponential with 2's power(log complexity), such as $x^4, x^{16}, \ldots,$ so we break arbitrary number into sum of 2's power

The algorithm can be seen as a **mix of multiplication and squaring**





The algorithm

Referring to the binary representation of exponent(from left to right, begin with the second position), if it is 1, we do squaring and multiplication(shifting in binary), if it is 0, only do squaring

```
def fast_exponential(x , e):
    res = x
    signal = bin(e)[3:]
    for i in signal:
        res *= res
        if i == '1':
            res*=x
    return res
```