Information Security HW4

四資工三 **B10615033** 王璽禎

- 開發環境

  Visual studio Code 使用 Python 撰寫

- 操作方式

  - Encrypt 輸入參數：**python RSA.py –e {plaintext}**

    ```
    C:\Users\HP AY111TX\Desktop\HW4>python RSA.py -e hello
    ```

  - Decrypt 輸入參數：**python RSA.py –d {ciphertext}**

    然後依照指示輸入 **d, p, q**

```
C:\Users\HP AY111TX\Desktop\HW4>python RSA.py -d 12166529024,10510100501,14693280768,14693280768,16850581551
```

- 執行結果圖

  - Encrypt：

```
C:\Users\HP AY111TX\Desktop\HW4>python RSA.py -e hello
ciphertext =  12166529024,10510100501,14693280768,14693280768,16850581551
p = 763654005126058741730819057497450116203416620638125673987590741736898775109101108481269530510572199205311
949181641361871392236926294814724631877039414636 9
q = 166467407773134211530086075697562655549309723010291410529656469599062326383842063830290546512105278695811
904351683255615878646199442863713743737323480041 93
n = 127123502668906744000539876083492870718414475804345366840520580826663142051240723597124260796415455001794
417648099167933642236696171375281451049087931925987948413726286724283103152090898155376062612227598786506367 6
478487023908270425796720131220116255841746846470107867415780341792815432449220285463070677252 17
n bit = 1024
e = 5
d = 101698802135125395200431900866794296574731580643476293472416464661330513640992578877699408637132364001435
534118479334346913789356937100225160839270345540770932106318170172570229083156933910927277978976150710970820 8
747771417363500538897633370976442474674504402696670416019181859138338688083227262830434604597 25
n bit = 1024
e = 5
```

  - Decrypt：

```
C:\Users\HP AY111TX\Desktop\HW4>python RSA.py -d 12166529024,10510100501,14693280768,14693280768,16850581551
input d =
101698802135125395200431900866794296574731580643476293472416464661330513640992578877699408637132364001435534118
479334346913789356937100225160839270345540770932106318170172570229083156933910927277978976150710970820874777141
7363500538897633370976442474674504402696670416019181859138338688083227262830434604597 25
input p =
763654005126058741730819057497450116203416620638125673987590741736898775109101108481269530510572199205311949181
641361871392236926294814724631877039414636 9
input q =
166467407773134211530086075697562655549309723010291410529656469599062326383842063830290546512105278695811904351
683255615878646199442863713743737323480041 93
plaintext =  hello
```

- 程式碼

  - **Find large prime number**

```
def find_large_prime(bit):
    temp = []
    temp.append(1)       # 頭取1 : 確保bit數
    for i in range(bit-2):  # 中間隨機塞0或1
        rand = random.randint(0, 1)
        temp.append(rand)
    temp.append(1)       # 尾取1 : 確保是奇數

    arr = [str(j) for j in temp]
    prime = ''.join(arr)
    b = int(prime, 2)
    return b
```

  - **Miller Rabin Test**

```
def miller_rabin_test(n):
    if n == 2:
        return True
    if n%2 == 0:
        return False
    # n-1 = 2^s * d   ,  找 s  d
    s, d = 0, n-1 # d is odd
    while d%2 == 0:
        s += 1
        d //= 2
    # 循環測試
    # a^(n-1)≡1(mod n) 成立 -> 可能是prime ; 不成立 -> 一定是composite (witness)
    count = 10
    for i in range(count):  # 檢查次數
        a = random.randrange(1, n-1)  # [1, n-1] 之間選出 a
        x = square_and_multiply(a, d, n)  # x = a^d (mod n)
        if x == 1 or x == n-1:  # 可能是prime 繼續確認
            continue
        for _ in range(s-1):  # [0, s-1] 的範圍 x^2
            x = square_and_multiply(x, 2, n)  # x = x^(2*i) (mod n)
            if x == n-1:  # 可能是prime 繼續確認
                break
        else:  # for [0, s-1] 跑完仍無法判定是prime, 就是composite
            return False
    return True
```

    - ✓ 根據「餘數系統平方根」的特性，以質數 n 為 module，

(0~n-1) 之間只有 1 和 n-1 平方後(mod n)為 1

- ✓ 根據「費馬小定理」，**a ^ (n-1) ≡1 (mod n)**
- ✓ 選擇底數 a，待測數字 n，若 n 為 prime，則 n-1 一定為偶數
- ✓ 將 n-1 拆解為**(a ^ s) * d**，d 為奇數
- ✓ 觀察 **a ^ d (mod n)**
  - ☆ 若為 1 或 n-1，接下來平方都為 1 (->可能是 prime，繼續檢查)
- ✓ 觀察 **a ^ d ^ (2 ^ t) ~ a ^ d ^ (2 ^ (t-1)) (mod n)**，(0<=t<=s-1) 每個數都是前面數字的平方
  - ☆ 若平方後為 1 (->composite)
  - ☆ 若平方後為 n-1 (->可能是 prime，繼續檢查)
- ✓ Miller Rabin Test 無法保證通過測試的數字一定為 prime，但未通過測試的一定是 composite。檢查次數(count)越多，對的機率越高

➢ **確認是否是 prime**

```
def prime(bit):
    while 1:
        prime = find_large_prime(bit)
        if miller_rabin_test(prime) == True:
            return prime
```

- ✓ 產生大數字後，丟到 miller_rabin_test 測試是否為 prime，一直找到出現 True 為止，回傳 prime

➢ **Square and Multiply**

```
def square_and_multiply(x, e, n):   # z = x^e mod n
    e = '{0:b}'.format(e)   # 字串格式化 (decimal->binary) e=3 -> e='11'
    output = 1
    for i in range(len(e)):
        output = (output**2) % n    # 進位: (output^2)
        if(e[i]=='1'):
            output = (output*x) % n # 加1: *(x)
    return output
```

- ✓ 加速 exponent 的部分，轉成二進制，減少計算次數
- ✓ 等同於 python pow()

- 尋找乘法反元素

```python
def egcd(a, b):
    if a == 0:
        return (b, 0, 1)
    else:
        g, y, x = egcd(b % a, a)
        return (g, x - (b // a) * y, y)

def inverse(e, phi_n):
    g, x, y = egcd(e, phi_n)
    if g != 1:
        raise Exception('no inverse')
    else:
        return x % phi_n
```

- ✓ 利用擴展歐幾里得算法，egcd(a, n)，可得到 ax+ny=g (g 是 a, n 的最大公因數)
- ✓ 確定 g = 1，e 和 phi_n 互質的情況下才有反元素存在。
- ✓ 當 a==0，gcd(a, b)=b，此時 x=1, y=0
- ✓ 計算 egcd，回傳(g, 餘數, 商數)，直到遞迴結束。

- **Encrypt**

```
def Encrypt():
    # 1. choose large prime p, q
    # 2. compute n
    plaintext = sys.argv[2]
    p = prime(512)
    q = prime(513)
    n = p * q
    if len(str(n)) < len(str(plaintext)):
        raise Exception('n不能小於plaintext')
    # 3. compute phi_n
    phi_n = (p-1)*(q-1)
    # 4. select e
    e = 0
    for i in range(2, phi_n):
        if math.gcd(i, phi_n) == 1:
            e = i
            break
    # 5. compute private key d
    d = inverse(e, phi_n)
    # Encrypt
    ciphertext = []
    for i in plaintext:
        text = square_and_multiply(ord(i), e, n)
        ciphertext.append(text)
    # 將ciphertext轉成string
    arr = [str(j) for j in ciphertext]
    ciphertext = ','.join(arr)
    print("ciphertext = ", ciphertext)
    print("p =", p)
    print("q =", q)
    print("n =", n)
    print("n bit =", len(bin(n))-2)
    print("e =", e)
    print("d =", d)
```

- ✓ 自動產生 512 bits 的 p, 513 bits 的 q，確保 n 至少為 1024 bits
- ✓ 計算 n, phi_n
- ✓ 檢查 n 的長度必須大於 plaintext 的長度，否則 module 會有部分資料消失
- ✓ 選擇 e，從(2~phi_n-1)的範圍，找出與 phi_n 互質的數
- ✓ 找出 d，e 的乘法反元素
- ✓ 對每一個 plaintext 的字元分別做加密
- ✓ 最後 print 出 p, q, n, e, d

➢ **Decrypt**

```
def Decrypt():
    cipher_text = sys.argv[2]
    arr = cipher_text.split(',')
    ciphertext = []
    for i in arr:
        ciphertext.append(int(i))
    print("input d = ")
    d = int(input())
    print("input p = ")
    p = int(input())
    print("input q = ")
    q = int(input())

    # Decrypt
    plaintext = ''
    for i in ciphertext:
        temp = decrypt_with_CRT(i, d, p, q)
        plaintext += (chr(temp))
    print("plaintext = ", plaintext)
```

- ✓ plaintext = ciphertext ^ d (mod n)
- ✓ 利用 CRT 做解密

- **利用 CRT 加速 Decrypt**

```python
def decrypt_with_CRT(y, d, p, q):
    # step 1: 轉成 CRT domain (n -> p, q)
    # step 2: reduce exp (d -> dp, dq)
    dp = d % (p-1)
    dq = d % (q-1)
    q_inv = inverse(q, p)
    x1 = pow(y, dp, p)
    x2 = pow(y, dq, q)
    # step 3: inverse, CRT domain -> normal domain
    h = (q_inv*(x1-x2)) % p
    x = x2 + h*q
    return x
```

  - ✓ 收到加密資料 y 後，將 n 分為 p, q，減少 module 的大小
  - ✓ 將 d 分為 dp, dq，減少 exponent 的大小
  - ✓ 最後找出 q_inverse，計算出 h，找出 plaintext

- **心得與遇到的困難**

剛開始測試完小數字後，要將 plaintext 轉成字串形式，卻沒有將 p, q 改大，解密回來後發現 output 結果都是不可視字元，且數字都在 p*q 以內，這才發現 n 的大小要大於 plaintext，不然怎麼解密都是錯的！

在實作 Miller Rain Test 時，一開始是照著老師 ppt 上面的 source code 打的，卻發現連”5”這麼小的質數都給我 return False，上網查了很多資料，才理解這個 test 用了很多數學原理作根據，了解其中步驟後，才比較懂得 code 在寫什麼。

在計算大數時，本來想說 python 需不需要有另外的方法存大數，還特地用了 list 和”long”這個 type，結果發現 python 會自動將溢位的資料轉成 long，因此在 python 中不需要擔心範圍的問題，比在 C++上實作方便許多。

還發現 pow() 這個好用的 function，裡面的實作技巧就是 square and multiply。

從一開始連 RSA 是什麼都不知道，慢慢研究 RSA 每一個步驟在幹嘛、每一個背後的數學原理是什麼，到可以自己刻出一個 RSA 的 code，真的挺有成就感的。