Information Security HW5

- **分工**

  四資工三 B10615033 王璽禎 Key Generation
  四資工三 B10615035 陳靜萱 Sign, Verify

- **開發環境**

  Visual studio Code 使用 Python 撰寫

- **操作方式**

  ➢ 輸入參數：**python DSA.py {plaintext}**

- **執行結果圖**

```
C:\Users\HP AY111TX\Desktop\HW5>python DSA.py hello
-----Generate Key-----
p:  9208257367714768262124341117442962121832468394453537674017191994384681506402065224243494106045110406919919586689508535198670883137966293321 09
70262131241907055268039862967265393904981350065633452955637069195945449032597823694507327313255527049974235320086418547449184727767698450186335 21
4073949852973523063537
p_bits:  1024
q:  971648641527573436418476151350153781633940967561
q_bits:  160
a:  13955292358336119569964954906295248038017493822762880002387250026298340076472266303934676498841001106951029783373372738439293429479490405547 9
86072956389936243965432606116725733496623980691900825722262057093754027821641014154705753941025522089985833378987511704280486514554170380099678 87
3237427807954906288905
d:  1675709703302591903508282687514146825751622048 20
B:  86633878003301830911761337971810019997691685472209147345769223780620119114335160129290089582685976568631314221566896641484565493934851055833 4
93771373112773709448682693457559839731342008853163938208973407314598451564635393311443070571622210739513040545960331767373223355206067319510631 64
833783989393010709357
-----Signature-----
SHA:  aaf4c61ddcc5e8a2dabede0f3b482cd9aea9434d
Ke:  374251937284465100015634783217910064001292974358
Ke_inv:  471003033682815188284086018972532536923228983148
r:  959546985420702877441869000996346063573528011896
s:  [531476926754956738982999927351277345243455353471, 531476926754956738982999927351277345243455353471, 94319481211388580756647771951363246659 17
18334089, 712610524634888867405604403290696379644852398153, 501834352593013679132695813946188637455972352206, 6829679504729458075553002898856 0767
1857369396888, 271250065114016738971822497723252550509106416270, 1188744748255430998305681568567392745260367793, 11887447482554309983056815685673
92745260367793, 501834352593013679132695813946188637455972352206, 501834352593013679132695813946188637455972352206, 2119649167901306192712142709 1
307513493414041 3740, 472191778431070619282391700541099929668489350941, 653325376311002747704996176480518964069886395623, 531476926754956738982999
927351277345243455353471, 742253098796831927255908516695785087432335399418, 1188744748255430998305681568567392745260367793, 531476926754956738982
999927351277345243455353471, 308313189101984908486097949736561005327433690 58, 472191778431070619282391700541099929668489350941, 11887447482554309
98305681568567392745260367793, 472191778431070619282391700541099929668489350941, 771895672958774987106212630100873795219818400683, 94319481211388
58075664777195136324665917 18334089, 241607490952073679121518384318163842721623415005, 308313189101984908486097949736561005327433690 58, 7126105246
34888867405604403290696379644852398153, 653325376311002747704996176480518964069886395623, 742253098796831927255908516695785087432335399418, 50183
435259301367913269581394618863745597235220 6, 1188744748255430998305681568567392745260367793, 152679768466244499570606044102897719359174411210, 53
147692675495673898299992735127734524345535 3471, 472191778431070619282391700541099929668489350941, 53147692675495673898299992735127734524345535347
1, 152679768466244499570606044102897719359174411210, 712610524634888867405604403290696379644852398153, 241607490952073679121518384318163842721623
415005, 712610524634888867405604403290696379644852398153, 1188744748255430998305681568567392745260367793]
-----Verify-----
v:  [959546985420702877441869000996346063573528011896, 959546985420702877441869000996346063573528011896, 95954698542070287744186900099634606357 35
28011896, 959546985420702877441869000996346063573528011896, 959546985420702877441869000996346063573528011896, 959546985420702877441869000996346 06
3573528011896, 959546985420702877441869000996346063573528011896, 959546985420702877441869000996346063573528011896, 95954698542070287744186900099 6
346063573528011896, 959546985420702877441869000996346063573528011896, 959546985420702877441869000996346063573528011896, 9595469854207028774418690
00996346063573528011896, 959546985420702877441869000996346063573528011896, 959546985420702877441869000996346063573528011896, 959546985420702877 44
1869000996346063573528011896, 959546985420702877441869000996346063573528011896, 959546985420702877441869000996346063573528011896, 9595469854207 02
877441869000996346063573528011896, 959546985420702877441869000996346063573528011896, 959546985420702877441869000996346063573528011896, 9595469854
207028774418690009963460635735280 11896, 959546985420702877441869000996346063573528011896, 959546985420702877441869000996346063573528011896, 95954
6985420702877441869000996346063573528011896, 959546985420702877441869000996346063573528011896, 959546985420702877441869000996346063573528011896,
959546985420702877441869000996346063573528011896, 959546985420702877441869000996346063573528011896, 959546985420702877441869000996346063573528011
896, 959546985420702877441869000996346063573528011896, 959546985420702877441869000996346063573528011896, 95954698542070287744186900099634606357 35
28011896, 959546985420702877441869000996346063573528011896, 959546985420702877441869000996346063573528011896, 959546985420702877441869000996346 06
3573528011896, 959546985420702877441869000996346063573528011896, 959546985420702877441869000996346063573528011896, 95954698542070287744186900099 6
346063573528011896, 959546985420702877441869000996346063573528011896, 959546985420702877441869000996346063573528011896]
-----Output (r==v -> valid)-----
valid
```

- 程式碼
  - 生成質數
    - ✓ 沿用上次 RSA 生成質數的 code
    - ✓ 利用 find_large_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

```python
def find_large_prime(bit):
    temp = []
    temp.append(1)
    for i in range(bit-2):
        rand = random.randint(0, 1)
        temp.append(rand)
    temp.append(1)

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

```python
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 = pow(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 = pow(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
```

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

- 尋找乘法反元素

  ✓ 利用擴展歐幾里得算法找出 inverse

```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
```

- 生成 **bit**

  ✓ 隨機生成 bit 數，給 k 用的

```python
def bitGenerator(bit):
    temp = []
    for i in range(bit):
        rand = random.randint(0, 1)
        temp.append(rand)
    arr = [str(j) for j in temp]
    prime = ''.join(arr)
    b = int(prime, 2)
    return b
```

- **Generate key**
  - ✓ 生成大質數 p, q，q 是(p-1)之中的一個因數，且為 160 bits
  - ✓ 先產生 q，再隨機生成 864bits 的 k，來找出 p (p=k*q+1)，每產生一個 p 就要丟進 miller_rabin_test 確認是否為 prime
  - ✓ 為了避免 q 不對而找不到 p 的情況，每產生一個 p 就將 count++，最多找 100 次，若沒找到則再產生一次 q，找到 p 就跳出迴圈

```
### generate key
print("-----Generate Key-----")
# p(1024) = q(160) * k + 1
flag = False
while 1:
    q = prime(160)
    count = 0
    while count < 100:
        k = bitGenerator(1024-160)
        p = k * q + 1
        if (p.bit_length() == 1024 and miller_rabin_test(p)):
            flag = True
            break
        count += 1
    if flag:
        break
print("p: ", p)
print("p_bits: ",p.bit_length())
print("q: ", q)
print("q_bits: ",q.bit_length())

# a = h^((p-1)/q)
h = random.randint(2, p-1)
a = pow(h, (p-1)//q, p)
print("a: ", a)

# 1 < d < q
d = random.randint(1, q)
print("d: ", d)

# B = a^d mod p
B = pow(a, d, p)
print("B: ", B)
```

- **Signature**
  - ✓ 利用 sha1()來將 plaintext 做 hash
  - ✓ 隨機產生 ke，並找出 ke 的 inverse
  - ✓ 計算 r 和 s，用迴圈來跑 hash 後 plaintext 的長度，將所有計算結果放在 s 的陣列裡

```
### signature
print("-----Signature-----")
SHA = sha1(plaintext.encode('utf-8')).hexdigest()
print("SHA: ", SHA)
# 1 < Ke < q
Ke = random.randint(1, q)
print("Ke: ", Ke)
Ke_inv = inverse(Ke, q)
print("Ke_inv: ", Ke_inv)
# r = a^Ke mod p mod q
r = pow(a, Ke, p) % q
print("r: ", r)
# s = (SHA + d*r)*Ke_inv mod q
s = []
for i in SHA:
    i = ord(i)
    s_n = ((i + d*r) * Ke_inv) % q
    s.append(s_n)
print("s: ", s)
```

- **Verify**
  - ✓ 用迴圈來跑陣列 s 的長度
  - ✓ 計算出 s 的 inverse，再分別乘上 sha 和 r，計算出 a 和 B 的指數
  - ✓ 計算出每個 s 對應的 v
  - ✓ 若所有陣列中的 v 都等於 r，則此 signature 為有效的

```
### verify
print("-----Verify-----")
v = []
for i in range(len(s)):
    w = inverse(s[i], q)
    u1 = (w * ord(SHA[i])) % q
    u2 = (w * r) % q
    v_n = ((pow(a, u1, p) * pow(B, u2, p)) % p) % q
    v.append(v_n)
print("v: ", v)

### Compare
print("-----Output (r==v -> valid)-----")
for i in range(len(v)):
    if r != v[i]:
        print('invalid')
        break
else:
    print('valid')
```

- **心得與遇到的困難**

我覺得在實作 DSA 的部分，最困難的是在 generate key 的部分找到 p 和 q 這兩個大質數。q 需要符合是 p-1 的因數，且要是 160 bits，而為了要找到 ke 和 s 的反元素，q 也必須是質數才行。一開始我的方法是先生成 p=1024 bits 的質數，再一個一個慢慢除，當長度符合 160 bits 時確認他是否為 prime，來找到 q，結果這個方法讓程式一直在迴圈裡跑，完全找不到符合條件的 q。最後我換個方向思考，改先生成 q=160 bits 的質數，再生成 k=(1204-160)bits，因為 p = k*q+1，這個方法結果一下就出來了，也許是乘法比除法要精確吧，畢竟乘法不需要考慮到會有浮點數的出現。看到錯誤訊息不再是”no inverse”的那瞬間，真的很高興！