Practical Assignment 5 (PA5)

Name:	Ashish Kumar
Roll No:	18075068
Dept:	CSE B.Tech. 4th Year
Course:	Network Security (CSE-537)

Practical Assignment 5

Implement the El Gamal algorithm in a language of your choice.

As usual, the Practical Assignment submission must include a report (pdf) consisting of basic theory, explanation of interesting code snippets, screen shots and a Github link where your code / data is deposited.

GitHub link (Source code): https://github.com/krashish8/netsec-assignments

(Exact Link of this Assignment)

El Gamal Algorithm:

The algorithm is used for Encryption and Decryption, and also for Signing and Verifying the cryptographic signatures.

Generation of Public and Private Key:

The algorithm takes a prime number q and α, which is a primitive root of q.

- **Private Key**: X_A, which is generated as a random number lying in the range (1, q-1).
- **Public Key**: $Y_A = \alpha^{X_A} \, mod \, q$. Public key is {q, α, Y_A}.

The public key can be used for encrypting the message (and can be sent across the network), and the private key for decrypting the message (secret to the user).

Encryption:

The encryption is performed by anyone having the public key {q, α, Y_A}.

- The message is hashed in the form of an integer in the range [0, q-1]. Each character of the message can be hashed in this way.
- A random number k is chosen such that k lies between [1, q-1].
- One-time key K is computed as $K = (Y_A)^k \, mod \, q$
- The message M is encrypted as a pair (C_1, C_2), where $C_1 = \alpha^k \, mod \, q$, $C_2 = KM \, mod \, q$.

The pair (C_1, C_2) is sent to the user, as an encrypted message.

Decryption:

The decryption is performed by the user, who has the private key X_A.

- The one-time key K is recovered as $K = (C_1)^{X_A} \, mod \, q$.
- The message is decrypted as $M = (C_2 K^{-1}) \, mod \, q$, where K^{-1} is the modular inverse of K w.r.t q.

Signing a message:

The user having the private key can sign the message M.

- A random number K is chosen in the range [1, q-1] such that K is relatively prime to (q-1).
- Signature consists of the pair (S_1, S_2), where $S_1 = \alpha^K \, mod \, q$, $S_2 = K^{-1}(M - X_A S_1) \, mod \, (q - 1)$.

The pair (S_1, S_2) is sent as a signature.

Verifying the signature:

Anyone with the public key can verify the signature.

- Compute $V_1 = \alpha^M \, mod \, q$, and $V_2 = (Y_A)^{S_1}(S_1)^{S_2} \, mod \, q$
- The signature is value if $V_1 == V_2$.

Source Code:

The well-commented source code is shown below:

For complete source code, please refer to the repository link.

```
import random
import secrets


def pow(a, b, m):
    """
    Calculate a^b mod m, using the binary exponentiation algorithm
    """
    result = 1
    while b > 0:
        if b & 1:
            result = (result * a) % m
        b >>= 1
        a = (a * a) % m
    return result


def inverse(a, m):
    """
    Calculate the inverse of a mod m, when m is NOT prime,
    using the extended Euclidean algorithm
    """
    if gcd(a, m) != 1:
        raise ValueError("Inverse does not exist")
    u1, u2, u3 = 1, 0, a
    v1, v2, v3 = 0, 1, m
    while v3 != 0:
        q = u3 // v3
        v1, v2, v3, u1, u2, u3 = (
            u1 - q * v1), (u2 - q * v2), (u3 - q * v3), v1, v2, v3
    return u1 % m
```

```python
def gcd(a, b):
    """
    Calculate the greatest common divisor of a and b,
    using Euclid's algorithm
    """
    while b:
        a, b = b, a % b
    return a


class ElGamal:
    def __init__(self, q, alpha):
        """
        ElGamal encryption algorithm
        """
        # q must be a prime number
        # alpha must be a primitive root of q
        self.q = q
        self.alpha = alpha

        # Private key = x
        # Public key = (y, alpha, q)
        self.x = random.randint(1, q-1)
        self.y = pow(alpha, self.x, q)

    def encrypt(self, m):
        """
        Anyone with the public key can encrypt a message
        """
        k = random.randint(1, self.q-1)
        key = pow(self.y, k, self.q)
        c1 = pow(self.alpha, k, self.q)

        c2_list = []
        # m is a string, we encrypt each character
        # after converting the character to int
        for i in m:
            m_int = ord(i)
            c2 = (key * m_int) % self.q
```

```
            c2_list.append(c2)
        return (c1, c2_list)

    def decrypt(self, c):
        """
        Anyone with the private key can decrypt a message
        """
        c1, c2_list = c
        key = pow(c1, self.x, self.q)

        m = ""
        # each character is decrypted and then concatenated to m
        for c2 in c2_list:
            m_char = (c2 * inverse(key, self.q)) % self.q
            m += chr(m_char)
        return m

    def sign(self, m):
        """
        Anyone with the private key can sign a message
        """
        k = random.randint(1, self.q-1)
        while gcd(k, self.q-1) != 1:
            k = random.randint(1, self.q-1)
        s1 = pow(self.alpha, k, self.q)

        s2_list = []
        for i in m:
            m_int = ord(i)
            s2 = (inverse(k, self.q-1) *
                  (m_int - self.x * s1)) % (self.q - 1)
            s2_list.append(s2)
        return (s1, s2_list)

    def verify(self, m, s):
        """
        Anyone with the public key can verify a signature
        """
        s1, s2_list = s
```

```python
        v1_list = []
        for i in m:
            m_int = ord(i)
            v1 = pow(self.alpha, m_int, self.q)
            v1_list.append(v1)

        v2_list = []
        for s2 in s2_list:
            v2 = pow(self.y, s1, self.q) * pow(s1, s2, self.q) % self.q
            v2_list.append(v2)

        return v1_list == v2_list


# Creating the ElGamal object with prime number = 76403,
# and alpha = 4514, which is a primitive root of 76403
algo = ElGamal(q=76403, alpha=4514)

"""
===============================
Encryption and Decryption
===============================
"""

secret = "This is a secret message"

# e can be sent to the receiver
e = algo.encrypt(secret)

# The user decrypts the message
d = algo.decrypt(e)

print("Encrypted message:", e)
print("Decrypted message:", d)
```

```python
"""
===============================
Signing and Verifying
===============================
"""

message = "This message will be signed"

# The user signs the message
signature = algo.sign(message)

# Verifying the message
verification = algo.verify(message, signature)

print("Signature:", signature)
print("Verification:", verification)
```

Output:

```
$ python el_gamal.py

Encrypted message: (7629, [50694, 62764, 25166, 31201, 19312, 25166,
31201, 19312, 20338, 19312, 31201, 22752, 21545, 68799, 22752, 70006,
19312, 27580, 22752, 31201, 31201, 20338, 23959, 22752])

Decrypted message: This is a secret message

Signature: (47392, [47826, 25288, 20341, 47273, 75864, 553, 40129,
47273, 47273, 59917, 30235, 40129, 75864, 27485, 20341, 5500, 5500,
75864, 54970, 40129, 75864, 47273, 20341, 30235, 72008, 40129, 45076])

Verification: True
```

Screenshot:

```
ashish@ubuntu:~/Documents/acad/S8/Network Security/My Work/PA/assignment-5$ python el_gamal.py

Encrypted message: (7629, [50694, 62764, 25166, 31201, 19312, 25166, 31201, 19312, 20338, 19312, 31201,
 22752, 21545, 68799, 22752, 70006, 19312, 27580, 22752, 31201, 31201, 20338, 23959, 22752])

Decrypted message: This is a secret message

Signature: (47392, [47826, 25288, 20341, 47273, 75864, 553, 40129, 47273, 47273, 59917, 30235, 40129, 7
5864, 27485, 20341, 5500, 5500, 75864, 54970, 40129, 75864, 47273, 20341, 30235, 72008, 40129, 45076])

Verification: True
```