El Gamal Encryption

- Aim:
  Implement the ElGamal encryption algorithm in Java.

- Description:
  ElGamal encryption is an asymmetric key encryption algorithm used for secure communication. It is based on the Diffie-Hellman key exchange and involves key generation, encryption, and decryption processes. The security of the algorithm relies on the difficulty of computing discrete logarithms.

- Code:

```
import java.math.BigInteger;
import java.util.Random;

public class ElGamalEncryption {

    static BigInteger gcd(BigInteger a, BigInteger b) {
        if (b.equals(BigInteger.ZERO)) {
            return a;
        } else {
            return gcd(b, a.mod(b));
        }
    }

    static BigInteger genKey(BigInteger q) {
        Random rand = new Random();
        BigInteger key;
        do {
            key = new BigInteger(q.bitLength(), rand);
        } while (key.compareTo(q) >= 0 || !gcd(q, key).equals(BigInteger.ONE));

        return key;
    }

    static BigInteger power(BigInteger a, BigInteger b, BigInteger c) {
        return a.modPow(b, c);
    }

    static Object[] encrypt(String msg, BigInteger q, BigInteger h, BigInteger g) {
        Random rand = new Random();
        BigInteger k = genKey(q);
        BigInteger s = power(h, k, q);
        BigInteger p = power(g, k, q);
```

```
        BigInteger[] enMsg = new BigInteger[msg.length()];
        for (int i = 0; i < msg.length(); i++) {
            enMsg[i] = BigInteger.valueOf((int) msg.charAt(i)).multiply(s);
        }

        System.out.println("g^k used : " + p);
        System.out.println("g^ak used : " + s);
        return new Object[] { enMsg, p };
    }

    static String decrypt(BigInteger[] enMsg, BigInteger p, BigInteger key, BigInteger q) {
        StringBuilder drMsg = new StringBuilder();
        BigInteger h = power(p, key, q);

        for (BigInteger value : enMsg) {
            drMsg.append((char) value.divide(h).intValue());
        }

        return drMsg.toString();
    }

    public static void main(String[] args) {
        Random rand = new Random();
        String msg = "encryption";
        System.out.println("Original Message : " + msg);

        BigInteger q = new BigInteger(100, rand);
        BigInteger g = new BigInteger(q.bitLength(), rand).mod(q);

        BigInteger key = genKey(q);
        BigInteger h = power(g, key, q);

        System.out.println("g used : " + g);
        System.out.println("g^a used : " + h);

        Object[] encryptedData = encrypt(msg, q, h, g);
        BigInteger[] enMsg = (BigInteger[]) encryptedData[0];
        BigInteger p = (BigInteger) encryptedData[1];

        String decryptedMsg = decrypt(enMsg, p, key, q);
        System.out.println("Decrypted Message : " + decryptedMsg);
    }
}
```

- Output:

```
Original Message : encryption
g used : 4898139155524043607662073745 93
g^a used : 24009562714893101993121998 6875
g^k used : 4568022057132161404798483289 37
g^ak used : 2582993300113635550950702 1775
Decrypted Message : encryption
```

- Code Explanation:

1. GCD Calculation (`gcd` method): Computes the greatest common divisor of two numbers.
2. Key Generation (`genKey` method): Generates a private key that is coprime to a large prime number `q`.
3. Modular Exponentiation (`power` method): Efficiently computes `(a^b) % c` using exponentiation by squaring.
4. Encryption (`encrypt` method):
   - Generates a random ephemeral key `k`.
   - Computes shared secret `s = h^k mod q`.
   - Encrypts each character of the message using `s`.
5. Decryption (`decrypt` method):
   - Computes the decryption key `h = p^key mod q`.
   - Recovers the original message by dividing the encrypted values by `h`.
6. Main Method (`main` method):
   - Generates prime `q`, base `g`, and private key.
   - Performs encryption and decryption.
   - Prints the results.

- Time Complexity:
  - Key Generation: O(log q)
  - Modular Exponentiation: O(log b)
  - Encryption & Decryption: O(n) (where `n` is the message length)

- Space Complexity:
  - O(n) for storing the encrypted message.
  - O(1) for other auxiliary variables.