**CSI4108 - Assignment 3**
**Nicholas Gin (300107597)**

1. **[1.5 marks]** A description of the Elgamal public key encryption algorithm can be found in many places, including Stallings 7th ed., pp. 300-303 (5th ed., pp. 305-308; 6th ed., pp. 292-294).

   Implement a toy version of this algorithm with prime $q = 89$ and primitive root $\alpha = 13$ (the "implementation" can be done in software or on paper, as long as you show your work). Demonstrate that encryption and decryption perform correctly. If two messages, $m_1$ and $m_2$, have been encrypted using the random integer $k = 37$, compute the value of $m_2$ if you know that $m_1 = 56$.

**My code for Question 1 can be found in q1_part1.py and q1_part2.py.**
To implement a version of the Elgamal algorithm with prime q = 89 and primitive root α = 13 (my code can be found in q1_part1.py), I followed the algorithm on Page 292 of the 6th edition textbook. The question also indicates that the value of a message, m1, is 56.

```python
import random

# Following steps described on Page 292 of 6th Edition Textbook
q = 89 # Given in the question
alpha = 13 # Given in the question

def generate_keys():
    # Assuming that User A will only use this function once to generate their private and public keys;
    # then they will store these keys after they are returned by the function.
    x_a = random.randint(2, q-2) # A's private key (generate a random integer x_a, such that 1 < x_a < q - 1)
    y_a = pow(alpha, x_a, q) #  y_a = (alpha**x_a) mod q
    a_public_key = [q, alpha, y_a] # A's public key
    return x_a, a_public_key # Return A's private and public keys

def encrypt(a_public_key, M): # Takes A's public key (a_public_key) and the message (M) as parameters.
    # Since m1 = 56 is already an integer M in the range 0 <= M <= q - 1, m1 is represented by its actual value, 56.
    k = random.randint(1, q-1) # Generate a random integer k such that 1 <= k <= q - 1
    K = pow(a_public_key[2], k, q) # Compute a one time key K = ((y_a)**k) mod q.
    # Encrypted M as a pair of integers (C1, c2) where C1 = (alpha**k) mod q, and C2 = K*M mod q.
    C1 = pow(alpha, k, q)
    C2 = (K * M) % q
    # Return C1 and C2.
    return [C1, C2]

def recover(x_a, C1, C2): # Takes  A's private key, C1 and C2 as parameters.
    K = pow(C1, x_a, q) # Recover the one-time key by computing K = ((C1)**x_a) mod q.
    M = (C2 * (pow(K, -1, q))) % q  # Finally, recover M by computing (C2 * (K**-1)) mod q.
    # Return M.
    return M

def main():
    a_private_key, a_public_key = generate_keys()
    encrypted_pair = encrypt(a_public_key, 56)
    M = recover(a_private_key, encrypted_pair[0], encrypted_pair[1])

    print("Original, unencrypted Message:", M)
    print("Generated keys for Alice:", a_private_key, "(private key),", a_public_key, "(public key)")
    print("Message encrypted as a pair of integers (C1, C2):", encrypted_pair)
    print("Recovered Message:", M)


if __name__ == '__main__':
    main()
```

To demonstrate the encryption and decryption processes of the Elgamal algorithm, I have implemented a version of this algorithm that encrypts and decrypts a message of value 56; this is m1 from the question.

Firstly, I generate a private/public key pair from the perspective of User A, using the generate_keys function I have defined. I generate a random integer x_a, such that 1 < x_a < q - 1. I then compute y_a = (alpha**x_a) mod q. User A's private key is x_a and their public key is {q, alpha, y_a}. The generate_keys function returns both x_a and the public key. I am assuming that User A will only use this function once to generate their private and public keys; then they will store these keys after they are returned by the function.

```
def generate_keys():
    # Assuming that User A will only use this function once to generate their private and public keys;
    # then they will store these keys after they are returned by the function.
    x_a = random.randint(2, q-2) # A's private key (generate a random integer x_a, such that 1 < x_a < q - 1)
    y_a = pow(alpha, x_a, q) #  y_a = (alpha**x_a) mod q
    a_public_key = [q, alpha, y_a] # A's public key
    return x_a, a_public_key # Return A's private and public keys
```

I then encrypted the message, m1 = 56, using the encrypt function I defined. This function accepts User A's public key and the message itself as parameters. Since m1 = 56 is already an integer M in the range 0 <= M <= q - 1, m1 is represented by its actual value, 56. I then generated a random integer k such that 1 <= k <= q - 1. Using this value of k, I was able to compute a one-time key, K, such that K = ((y_a)**k) mod q. I then encrypted m1 as a pair of integers (C1, C2) where C1 = (alpha**k) mod q, and C2 = K*M mod q; these values are then returned by the function.

```
def encrypt(a_public_key, M): # Takes A's public key (a_public_key) and the message (M) as parameters.
    # Since m1 = 56 is already an integer M in the range 0 <= M <= q - 1, m1 is represented by its actual value, 56.
    k = random.randint(1, q-1) # Generate a random integer k such that 1 <= k <= q - 1
    K = pow(a_public_key[2], k, q) # Compute a one time key K = ((y_a)**k) mod q.
    # Encrypted M as a pair of integers (C1, c2) where C1 = (alpha**k) mod q, and C2 = K*M mod q.
    C1 = pow(alpha, k, q)
    C2 = (K * M) % q
    # Return C1 and C2.
    return [C1, C2]
```

To decrypt (recover) the message as User A, I used the recover function I defined. This function takes User A's private key, C1 and C2 as parameters. I recover the one-time key by computing K = ((C1)**x_a) mod q. Finally, I recover the message by computing (C2 * (K**-1)) mod q; M is then returned by the function and its value is printed out in the main.

```
def recover(x_a, C1, C2): # Takes  A's private key, C1 and C2 as parameters.
    K = pow(C1, x_a, q) # Recover the one-time key by computing K = ((C1)**x_a) mod q.
    M = (C2 * (pow(K, -1, q))) % q  # Finally, recover M by computing (C2 * (K**-1)) mod q.
    # Return M.
    return M
```

```python
def main():
    a_private_key, a_public_key = generate_keys()
    encrypted_pair = encrypt(a_public_key, 56)
    M = recover(a_private_key, encrypted_pair[0], encrypted_pair[1])

    print("Original, unencrypted Message:", M)
    print("Generated keys for Alice:", a_private_key, "(private key),", a_public_key, "(public key)")
    print("Message encrypted as a pair of integers (C1, C2):", encrypted_pair)
    print("Recovered Message:", M)
```

To answer the second part of this question, in which a value of m2 is computed such that m1 = 56 and k = 37, please refer to q1_part2.py.
Page 294 of the 6th edition textbook indicates the following formula:

If $M_1$ is known, then $M_2$ is easily computed as

$$M_2 = (C_{2,1})^{-1} C_{2,2} M_1 \bmod q$$

In the modified code, I set k = 37 in the encrypt function, as indicated in the question.

```python
def encrypt(a_public_key, M): # Takes A's public key (a_public_key) and the message (M) as parameters.
    # Since m1 = 56 is already an integer M in the range 0 <= M <= q - 1, m1 is represented by its actual value, 56.
    k = 37 # Given in the question
    K = pow(a_public_key[2], k, q) # Compute a one time key K = ((y_a)**k) mod q.
    # Encrypted M as a pair of integers (C1, c2) where C1 = (alpha**k) mod q, and C2 = K*M mod q.
    C1 = pow(alpha, k, q)
    C2 = (K * M) % q
    # Return C1 and C2.
    return [C1, C2]
```

I still assume that User A will only use the generate_keys function once to generate their private and public keys; then they will store these keys after they are returned by the function.
In the example screenshots below, the value of C2,1 is returned as 61 after running the main function. Let's say that C2,2 has been determined to have the value 88 (the professor said in class that we can assume a value of C2,2). I then substitute this value of C2,1 into the formula from the textbook to calculate M2. In this example, M2 = 2.

```python
def main():
    a_private_key, a_public_key = generate_keys()
    encrypted_pair = encrypt(a_public_key, 56)
    M = recover(a_private_key, encrypted_pair[0], encrypted_pair[1])

    M2 = (pow(encrypted_pair[1], -1, q) * 88 * M) % q

    print("Original, unencrypted Message:", M)
    print("Generated keys for Alice:", a_private_key, "(private key),", a_public_key, "(public key)")
    print("Message encrypted as a pair of integers (C1,1, C2,1):", encrypted_pair)
    print("Recovered Message (M1):", M)
    print("M2 (assuming that (C2,2) = 88):", M2)
```

```
python -u "/Users/nicholasgin/Desktop/csi4108_a3/q1_part2.py"
(base) nicholasgin@Nicholass-MacBook-Pro ~ % python -u "/Users/nicholasgin/Desktop/csi4108_a3/q1_part2.py"
Original, unencrypted Message: 56
Generated keys for Alice: 84 (private key), [89, 13, 11] (public key)
Message encrypted as a pair of integers (C1,1, C2,1): [29, 61]
Recovered Message (M1): 56
M2 (assuming that (C2,2) = 88): 2
(base) nicholasgin@Nicholass-MacBook-Pro ~ %
```

2. **[2 marks]** Implement the Miller-Rabin probabilistic primality testing algorithm to find a 14-bit integer that is probably prime with confidence $t = 5$. (Actually implement the Miller-Rabin algorithm; don't just call it from some library or toolkit.) Is your "probable prime" in this table: https://primes.utm.edu/lists/small/10000.txt ?

**My code for Question 1 can be found in q2.py.**

To implement the Miller-Rabin probabilistic primality testing algorithm to find a 14-bit integer that is probably prime with confidence t = 5, I followed the code example on Page 693 of the 6th edition textbook, and Page 6 of our Week3a(4108NumTheory) course notes.

```python
# Following code example on Page 693 of 6th Edition Textbook
# and Page 6 of Week3a(4108NumTheory) notes
import random
def miller_rabin_test(n):
    # This function implements the Miller-Rabin Test. If it returns False, it is inconclusive that a given positive
    # integer n is a composite number. If it returns True, it is conclusive that a given positive integer n is a
    # composite number.

    # Find k, q with k > 0, q odd s.t. n - 1 = (2**k) * q.
    q = n-1
    k = 0

    while (0 == (q % 2)):
        k += 1
        q = q // 2

    # Select random integer a in the range, 1 < a < (n - 1).
    a = random.randint(2, n - 2)
    a_q_mod_n = pow(a, q, n)

    # If (a**q) mod n is equal to 1, return False (inconclusive that n is composite)
    if (1 == a_q_mod_n):
        return False # Inconclusive that n is composite (probable prime)

    # For j = 0 to k - 1 do: if a**((2**j) * q) mod n is equal to n - 1, return False
    # (inconclusive if n is composite)
    e = q
    for j in range(k):
        if (n-1) == (pow(a, e, n)):
            return False # Inconclusive that n is composite (probable prime)
        e = 2*e

    return True # Conclusive that the number is composite

def k_trials(t, num):
    # Running Miller-Rabin Test for 5 trials.
    for i in range(t):
        if (miller_rabin_test(num) == True):
            # If at least one of the trials determines the number is not a probable prime,
            # return True.
            return True # Conclusive that the number is composite
    return False # Inconclusive that the number is composite

def main():
    t = 5
    for i in range(pow(2, 13) + 1, pow(2, 14) - 1): # Range of 14 bit numbers that could possibly be prime.
        # Find the first number in this range that we can not conclude is composite
        if k_trials(t, i) == False:
            print("A probable prime is:", i)
            break
    # Should print out 8209 - which is in the table from https://primes.utm.edu/lists/small/10000.txt!

if __name__ == '__main__':
    main()
```

This miller_rabin_test function I have defined implements the Miller-Rabin Test. If it returns False, it is inconclusive that a given positive integer n is a composite number. If it returns True, it is conclusive that a given positive integer n is a composite number.

Firstly, I find k, q with k > 0, and q odd such that n - 1 = (2**k) * q.

```
    # Find k, q with k > 0, q odd s.t. n - 1 = (2**k) * q.
    q = n-1
    k = 0

    while (0 == (q % 2)):
        k += 1
        q = q // 2
```

Secondly, I select a random integer a in the range, 1 < a < (n - 1).

```
    # Select random integer a in the range, 1 < a < (n - 1).
    a = random.randint(2, n - 2)
    a_q_mod_n = pow(a, q, n)
```

I then check If (a**q) mod n is equal to 1. If this is the case, return False (it is inconclusive that n is composite).

```
    # If (a**q) mod n is equal to 1, return False (inconclusive that n is composite)
    if (1 == a_q_mod_n):
        return False # Inconclusive that n is composite (probable prime)
```

For j = 0 to k - 1 do: if a**((2**j) * q) mod n is equal to n - 1, return False (it is inconclusive that n is composite).

```
    # For j = 0 to k - 1 do: if a**((2**j) * q) mod n is equal to n - 1, return False
    # (inconclusive if n is composite)
    e = q
    for j in range(k):
        if (n-1) == (pow(a, e, n)):
            return False # Inconclusive that n is composite (probable prime)
        e = 2*e
```

If you make it this far into the code without returning False, return True; it is conclusive that n is composite.

```
    return True # Conclusive that the number is composite
```

In the main function I call the function, k_trials, which I have defined. This function takes two parameters, t (the number of trials) and num (a number to test on). It tests the Miller Rabin Test on num for t trials using the miller_rabin_test function. If it is determined that num is composite during any one of the t trials, num is determined to be a composite number; else it is a probable prime.

```
def k_trials(t, num):
    # Running Miller-Rabin Test for 5 trials.
    for i in range(t):
        if (miller_rabin_test(num) == True):
            # If at least one of the trials determines the number is not a probable prime,
            # return True.
            return True # Conclusive that the number is composite
    return False # Inconclusive that the number is composite
```

Since the question states to find a 14-bit integer that is probably prime with confidence $t = 5$, call k_trials for every possible 14-bit number that can be prime and for t = 5 trials (from the main).

```python
def main():
    t = 5
    for i in range(pow(2, 13) + 1, pow(2, 14) - 1): # Range of 14 bit numbers that could possibly be prime.
        # Find the first number in this range that we can not conclude is composite
        if k_trials(t, i) == False:
            print("A probable prime is:", i)
            break
    # Should print out 8209 - which is in the table from https://primes.utm.edu/lists/small/10000.txt!
```

The first probable 14-bit prime to be determined is 8209 (this is the value outputted by the function).

```
python -u "/Users/nicholasgin/Desktop/csi4108_a3/q2.py"
(base) nicholasgin@Nicholass-MacBook-Pro ~ % python -u "/Users/nicholasgin/Desktop/csi4108_a3/q2.py"
A probable prime is: 8209
(base) nicholasgin@Nicholass-MacBook-Pro ~ %
```

The value 8209 is in the table from https://primes.utm.edu/lists/small/10000.txt.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 7001 | 7013 | 7019 | 7027 | 7039 | 7043 | 7057 | 7069 | 7079 | 7103 |
| 7109 | 7121 | 7127 | 7129 | 7151 | 7159 | 7177 | 7187 | 7193 | 7207 |
| 7211 | 7213 | 7219 | 7229 | 7237 | 7243 | 7247 | 7253 | 7283 | 7297 |
| 7307 | 7309 | 7321 | 7331 | 7333 | 7349 | 7351 | 7369 | 7393 | 7411 |
| 7417 | 7433 | 7451 | 7457 | 7459 | 7477 | 7481 | 7487 | 7489 | 7499 |
| 7507 | 7517 | 7523 | 7529 | 7537 | 7541 | 7547 | 7549 | 7559 | 7561 |
| 7573 | 7577 | 7583 | 7589 | 7591 | 7603 | 7607 | 7621 | 7639 | 7643 |
| 7649 | 7669 | 7673 | 7681 | 7687 | 7691 | 7699 | 7703 | 7717 | 7723 |
| 7727 | 7741 | 7753 | 7757 | 7759 | 7789 | 7793 | 7817 | 7823 | 7829 |
| 7841 | 7853 | 7867 | 7873 | 7877 | 7879 | 7883 | 7901 | 7907 | 7919 |
| 7927 | 7933 | 7937 | 7949 | 7951 | 7963 | 7993 | 8009 | 8011 | 8017 |
| 8039 | 8053 | 8059 | 8069 | 8081 | 8087 | 8089 | 8093 | 8101 | 8111 |
| 8117 | 8123 | 8147 | 8161 | 8167 | 8171 | 8179 | 8191 | 8209 | 8219 |
| 8221 | 8231 | 8233 | 8237 | 8243 | 8263 | 8269 | 8273 | 8287 | 8291 |
| 8293 | 8297 | 8311 | 8317 | 8329 | 8353 | 8363 | 8369 | 8377 | 8387 |
| 8389 | 8419 | 8423 | 8429 | 8431 | 8443 | 8447 | 8461 | 8467 | 8501 |
| 8513 | 8521 | 8527 | 8537 | 8539 | 8543 | 8563 | 8573 | 8581 | 8597 |
| 8599 | 8609 | 8623 | 8627 | 8629 | 8641 | 8647 | 8663 | 8669 | 8677 |
| 8681 | 8689 | 8693 | 8699 | 8707 | 8713 | 8719 | 8731 | 8737 | 8741 |
| 8747 | 8753 | 8761 | 8779 | 8783 | 8803 | 8807 | 8819 | 8821 | 8831 |
| 8837 | 8839 | 8849 | 8861 | 8863 | 8867 | 8887 | 8893 | 8923 | 8929 |
| 8933 | 8941 | 8951 | 8963 | 8969 | 8971 | 8999 | 9001 | 9007 | 9011 |
| 9013 | 9029 | 9041 | 9043 | 9049 | 9059 | 9067 | 9091 | 9103 | 9109 |
| 9127 | 9133 | 9137 | 9151 | 9157 | 9161 | 9173 | 9181 | 9187 | 9199 |
| 9203 | 9209 | 9221 | 9227 | 9239 | 9241 | 9257 | 9277 | 9281 | 9283 |
| 9293 | 9311 | 9319 | 9323 | 9337 | 9341 | 9343 | 9349 | 9371 | 9377 |
| 9391 | 9397 | 9403 | 9413 | 9419 | 9421 | 9431 | 9433 | 9437 | 9439 |

3. **[4 marks]** For this question you may use any big integer library or toolkit you wish in order to explore cryptographic algorithms with more realistically-sized numbers than are possible in a classroom setting.

   a. Using RSA with 1024-bit primes $p$ and $q$ and a public exponent $e = 65537$, encrypt the message $m = 466921883457309$. Use the Chinese Remainder Theorem to decrypt the resulting ciphertext $c$; how long does it take compared to decryption without using CRT (show your timing results if possible)?

   b. Using elliptic curve $E_p(a,b)$, where $p$ is a 256-bit prime number and $a$ and $b$ are any appropriate integers, choose private and public values for both Alice and Bob and compute their shared secret, $s$, using ECDH. Show that both parties can compute the same $s$. Compare the speed of computing $s$ using ECDH and using "ordinary" D-H at the same security level (show your timing results if possible).

**3a. My code for Question 3a can be found in q3a.sage.** I used the Sagemath toolkit to answer this question; this is a toolkit recommended and used in the 6th edition textbook (the Professor also approved the use of this toolkit in an email I sent him).

```
import time
def RSA():
    # RSA algorithm from pages 5-6 of Week7b(4108PKCToECC) course notes

    # Choose two prime numbers (ensuring that they are 1024 bit)
    p = int(random_prime((2**1024) - 1, (2**1023) + 1))
    q = int(random_prime((2**1024) - 1, (2**1023) + 1))

    # Let n = p * q
    n = p * q

    # Compute phi(n) = (p - 1) * (q - 1)
    phi = (p - 1) * (q - 1)

    # e has already been selected in the question
    # (the public exponent)
    e = 65537

    # Compute d such that 1 < d < phi(n) and e * d is congruent to 1 mod phi(n)
    # (the private exponent)
    d = int(inverse_mod (e, phi))
    # The public key is {e, nt}.
    # The private key is {d}.

    # The message to be encrypted (as given in the question)
    m = 466921883457309
    c = pow(m, e, n)

    return c, d, p, q, n

def decrypt(c, d, n):
    # Decryption formula from pages 5-6 of Week7b(4108PKCToECC) course notes
    decrypted_c = (c**d) % n
    return decrypted_c

def CRT_setup(p, q):
    # CRT_setup coded based on CRT Examples 2 and 3 from Week3b(4108NumTheory) course notes
    m1 = p
    m2 = q

    mm1 = m2
    mm2 = m1

    mm1_mod_m1 = mm1 % m1
    mm2_mod_m2 = mm2 % m2

    i1 = int(inverse_mod(mm1_mod_m1, m1))
    i2 = int(inverse_mod(mm2_mod_m2, m2))

    c1 = mm1 * i1
    c2 = mm2 * i2
    return c1, c2

def CRT(c, c1, c2, p, q, d, n):
    # CRT coded based on CRT Examples 2 and 3 from Week3b(4108NumTheory) course notes
    # Integers in the tuple
    t1 = int(pow(c % p, d % (p-1), p))
    t2 = int(pow(c % q, d % (q-1), q))
    decrypted_c_crt = (t1 * c1 + t2 * c2) % n

    return decrypted_c_crt

def main():
    ciphertext, d, p, q, n = RSA()
    print ("Ciphertext:", ciphertext, "\n")
```

```
    start = time.time()
    decrypted_c = decrypt(ciphertext, d, n)
    end = time.time()

    c1, c2 = CRT_setup(p, q)

    start2 = time.time()
    decrypted_c_crt = CRT(ciphertext, c1, c2, p, q, d, n)
    end2 = time.time()

    print ("Decrypted Ciphertext (Regular):", decrypted_c)
    print ("Duration:", end - start, "s," \n")
    print ("Decrypted Ciphertext (CRT):", decrypted_c_crt)
    print ("Duration:", end2 - start2, "s," \n")

if __name__ == '__main__':
    main()
```

Firstly, I coded the RSA algorithm as the RSA function based on pages 5-6 of Week7b(4108PKCToECC) course notes.
I chose two prime numbers, p and q, ensuring that they are both 1024 bits long.

```
    # Choose two prime numbers (ensuring that they are 1024 bit)
    p = int(random_prime((2**1024) - 1, (2**1023) + 1))
    q = int(random_prime((2**1024) - 1, (2**1023) + 1))
```

I computed n = p * q and then phi(n), which is equivalent to (p - 1) * (q -1).

```
    # Let n = p * q
    n = p * q
```

```
    # Compute phi(n) = (p - 1) * (q - 1)
    phi = (p - 1) * (q - 1)
```

The public exponent, e, is given in the question as 65537.

```
    # e has already been selected in the question
    # (the public exponent)
    e = 65537
```

I then computed a value for the private exponent, d, such that 1 < d < phi(n) and e *d is congruent to 1 mod phi(n), by finding (e**(-1)) mod phi(n), using Sagemath's inverse_mod function.

```
    # Compute d such that 1 < d < phi(n) and e * d is congruent to 1 mod phi(n)
    # (the private exponent)
    d = int(inverse_mod (e, phi))
    # The public key is {e, nt}.
    # The private key is {d}.
```

The message, m, to be encrypted is 466921883457309; this is indicated in the question. To encrypt m as a cipher text, c, I computed c = (m**e) mod n.

```
# The message to be encrypted (as given in the question)
m = 466921883457309
c = pow(m, e, n)

return c, d, p, q, n
```

The decrypt function I defined can decrypt a given cipher text, c, given the values of d and n, by computing (c**d) mod n. This is the decryption of the cipher text without using CRT.

```
def decrypt(c, d, n):
    # Decryption formula from pages 5-6 of Week7b(4108PKCToECC) course notes
    decrypted_c = (c**d) % n
    return decrypted_c
```

Since we also want to decrypt the cipher text using CRT, I have a function, CRT, to do so. To use this function, some values must be pre-computed and saved using CRT_setup, which takes the values of p and q as parameters. CRT_setup computes the values of c1 and c2, which are used in CRT; the actual code in this function follows the steps depicted in Example 2 from the

Week3b(4108NumTheory) course notes; c1 and c2 are returned by this function so they can be saved and used later. Note that CRT_setup is not included in the time measured for the CRT function to decrypt the cipher text (the Professor indicated over email it was unnecessary to include it, since the values returned by CRT_setup can be done once and saved).

```
def CRT_setup(p, q):
    # CRT_setup coded based on CRT Examples 2 and 3 from Week3b(4108NumTheory) course notes
    m1 = p
    m2 = q

    mm1 = m2
    mm2 = m1

    mm1_mod_m1 = mm1 % m1
    mm2_mod_m2 = mm2 % m2

    i1 = int(inverse_mod(mm1_mod_m1, m1))
    i2 = int(inverse_mod(mm2_mod_m2, m2))

    c1 = mm1 * i1
    c2 = mm2 * i2
    return c1, c2
```

The CRT function, which takes values of c (the cipher text), c1 and c2 (both from CRT_setup), p, q, d, and n as parameters, converts a number in the form, (base**exponent) mod number, to a single integer value. This function follows the steps depicted in Example 3 from the Week3b(4108NumTheory) course notes, in which the number is first converted into a tuple and then the values in this tuple are then combined to form a single integer value. CRT then returns the decrypted cipher text.

```
def CRT(c, c1, c2, p, q, d, n):
    # CRT coded based on CRT Examples 2 and 3 from Week3b(4108NumTheory) course notes
    # Integers in the tuple
    t1 = int(pow(c % p, d % (p-1), p))
    t2 = int(pow(c % q, d % (q-1), q))
    decrypted_c_crt = (t1 * c1 + t2 * c2) % n

    return decrypted_c_crt
```

The main function calls all necessary functions and computes the time required to decrypt the cipher text both regularly and using CRT.

```
def main():
    ciphertext, d, p, q, n = RSA()
    print ("Ciphertext:", ciphertext, "\n")

    start = time.time()
    decrypted_c = decrypt(ciphertext, d, n)
    end = time.time()

    c1, c2 = CRT_setup(p, q)

    start2 = time.time()
    decrypted_c_crt = CRT(ciphertext, c1, c2, p, q, d, n)
    end2 = time.time()

    print ("Decrypted Ciphertext (Regular):", decrypted_c)
    print ("Duration:", end - start, "s", "\n")
    print ("Decrypted Ciphertext (CRT):", decrypted_c_crt)
    print ("Duration:", end2 - start2, "s", "\n")
```

The decryption using CRT is more efficient than the decryption not using CRT. The CRT process of converting integers to tuples, doing the computation for the tuple components and then converting from the resulting tuple to an integer is faster than just doing the decryption computation on the original integers. The CRT process reduces an extremely large number into smaller, more manageable numbers to perform computations and a decryption on.

Examples of timing outputs (5 consecutive trials comparing regular and CRT decryptions):
Trial 1:

```
(base) nicholasgin@Nicholass-MacBook-Pro ~ % sage /Users/nicholasgin/Downloads/A3_300107597/q3a.sage
Ciphertext: 106907021946167529391329389117512979970114610749985480995426151201616551009409406836939670156402949018905733360439224712933852277415750156138266588202857289451751862855535306838108995729535839164116414396193694520602424976000803450352680944227453455661545339761476264138940209129959223858183397793977375396216773273308970106438203215953745581782234836947056763290131740453768532666257233137304151128341867628781229654393025292067821235592045196910394148751631716380658066859349889055618811522232322085083658328873965371599647780201766712764456070991079483664482330050959743351531444947872051189366250344606311903900 16

Decrypted Ciphertext (Regular): 466921883457309
Duration: 0.0017397403717041016 s

Decrypted Ciphertext (CRT): 466921883457309
Duration: 0.001444101333618164 s
```

Trial 2:

```
(base) nicholasgin@Nicholass-MacBook-Pro ~ % sage /Users/nicholasgin/Downloads/A3_300107597/q3a.sage
Ciphertext: 3630958057298680293422416999284235434899206484923670674840278303577973034559481090543818942400178096368251907262905092451728797767279535574437380546744567036308762420188273826440344022491025410546828059039571719695534325468052583372813768268304082507338023454215024600831709276585909090006973471355627446138746753680493918943353238826042968310997844527225313552188115651027637838655046439914473105871232002609940792495261064931468284575928993792117937822577811315774937130992748338747261295242690263130098290172561824612227991347185273935643164675721043435258315142273793771033455783292177538548726522155659905886 61

Decrypted Ciphertext (Regular): 4669218783457309
Duration: 0.0017900466918945312 s

Decrypted Ciphertext (CRT): 4669218783457309
Duration: 0.0014529228210449219 s
```

Trial 3:

```
(base) nicholasgin@Nicholass-MacBook-Pro ~ % sage /Users/nicholasgin/Downloads/A3_300107597/q3a.sage
Ciphertext: 257994123888897862135879247589002757884642521483459570519268524623841003145970869768444004256572667233863518370141395494366528726002094892140001680190677849568447140241829714588024524401224162682153055847539371004774076284322311325698832891494915844325988260094494301438356291912720125021812580817608416561227938624953527095814998318858686384705602712837262400146764859742977817924002838534572596504562969639091220092560201495263814403633316665303777056923392190075029963248525555295625486011897229741963483724728597069787899413885467199472272093042620535941623353413544789973390874487656533845143702806370436969556 6

Decrypted Ciphertext (Regular): 466921883457309
Duration: 0.0017650127410888672 s

Decrypted Ciphertext (CRT): 466921883457309
Duration: 0.00145721435546875 s
```

Trial 4:

```
(base) nicholasgin@Nicholass-MacBook-Pro ~ % sage /Users/nicholasgin/Downloads/A3_300107597/q3a.sage
Ciphertext: 215842335227807517003066024556365017114737429079183969474749595701801554720067927077477882045349201535841260657396613445277437380982120533059813745931118941902342179426314926054281646852193719871436371146985789610607431336030306080894111866347526497288418635888637302640145118979004414984878665623732477631872191311365186001790755511275952887151690392542426696444469951015455013112675661703545172810575214084027746692286490130143441335633864661007918910436296628814994828978079781342735939049406522294478527961568676011351037227761235621780499198269712028908214623575566206501088582003210109381737952312648986234985

Decrypted Ciphertext (Regular): 466921883457309
Duration: 0.0016999244689941406 s

Decrypted Ciphertext (CRT): 466921883457309
Duration: 0.0014100074768066406 s
```

Trial 5:

```
(base) nicholasgin@Nicholass-MacBook-Pro ~ % sage /Users/nicholasgin/Downloads/A3_300107597/q3a.sage
Ciphertext: 432781870260648101171526655577568051188615603202316359548085817825005268634423649826613479676191368478905095021895075554197514314329288002712714603911378447041998510071665879968629723396496551241904001207208729520779450050974551212723901149499979530894719054276400496985996334071390910789835768064008274113286963114226784292468268096987590187326373952561545280599381575133518676321093748217797419230157872888874054210178251582382833188879783013650482398740194568825546006930787284578075587104668663436401879574597062199639895868417313918576615573279330999421109639496164640441431963022102722859629756117175702456795 2

Decrypted Ciphertext (Regular): 466921883457309
Duration: 0.0017621517181396484 s

Decrypted Ciphertext (CRT): 466921883457309
Duration: 0.0014379024505615234 s
```

3. **[4 marks]** For this question you may use any big integer library or toolkit you wish in order to explore cryptographic algorithms with more realistically-sized numbers than are possible in a classroom setting.

   a. Using RSA with 1024-bit primes $p$ and $q$ and a public exponent $e = 65537$, encrypt the message $m = 466921883457309$. Use the Chinese Remainder Theorem to decrypt the resulting ciphertext $c$; how long does it take compared to decryption without using CRT (show your timing results if possible)?

   b. Using elliptic curve $E_p(a,b)$, where $p$ is a 256-bit prime number and $a$ and $b$ are any appropriate integers, choose private and public values for both Alice and Bob and compute their shared secret, $s$, using ECDH. Show that both parties can compute the same $s$. Compare the speed of computing $s$ using ECDH and using "ordinary" D-H at the same security level (show your timing results if possible).

**3b. My code for Question 3b can be found in q3b.sage.** I used the Sagemath toolkit to answer this question; this is a toolkit recommended and used in the 6th edition textbook (the Professor also approved the use of this toolkit in an email I sent him).

```
import random
import time
def ECDH():
    start = time.time()
    # Following code example from pg. 703 of 6th edition textbook and Page 4 from Week9b(4108AsymmetricKeying)
    # Generating a random 256 bit prime number
    p = random_prime((2**256) - 1, (2**255) + 1)

    F = GF(p)
    E = EllipticCurve(F, 5, 10)
    G = E.gen(0)
    q = E.order()

    # Alice computes a secret value n_a in 2...q-1
    n_a = randint(2, q-1)

    # Alice computes a public value p_a
    p_a = n_a * G

    # Bob computes a secret value n_b in 2...q-1
    n_b = randint(2, q-1)

    # Bob computes a public value p_b
    p_b = n_b * G

    # Alice computes the shared value (k_a)
    k_a = n_a * p_b

    # Bob also computes the shared value (k_b)
    k_b = n_b * p_a

    end = time.time()

    print("A's shared value (EDCH):", k_a, "\n")
    print("B's shared value (EDCH):", k_b, "\n")

    if (k_a == k_b):
        print("Using ECDH: A and B have jointly created a key for encryption.")

    duration = end - start
    print("Duration:", duration, "s", "\n")
```

```
def DH():
    # Following code example from pg. 699 of the textbook and Page 3 of Week9b(4108AsymmetricKeying) course notes.
    # To be equivalent security to ECC, private key must be 256 bits, and the public key must be 3072 bits.

    start2 = time.time()

    # p is a safety prime of length 3072 bits, which was randomly generated from this link: https://2ton.com.au/getprimes/random/3072?callback=safePrimes.
    p = 5706506157234918077697620794757832850164288054563798372894135938213062275647987966122258633068750422765412656757031356726396953590532832612651778614105809511849586196710555382454189104759450958990285411479790481437352705776283

    g = primitive_root(p)

    x_a = random.randint((2**255), (2**256) - 1) # Alice's private value
    y_a = pow(g, x_a, p) # Alice's public value

    x_b = random.randint((2**255), (2**256) - 1) # Bob's private value
    y_b = pow(g, x_b, p) # Bob's public value

    # Shared value:
    a_shared = pow(y_b, x_a)
    b_shared = pow(y_a, x_b)

    end2 = time.time()

    print("A's shared value (DH):", a_shared, "\n")
    print("B's shared value (DH):", b_shared, "\n")

    if (a_shared == b_shared):
        print("Using DH: A and B have jointly created a key for encryption.")

    duration2 = end2 - start2
    print("Duration:", duration2, "s")


def main():
    ECDH()
    DH()

if __name__ == '__main__':
    main()
```

Note that the value of p is cut off in this screenshot. The full value of p is:
5706506157234918077697620794757832850164288054563798372894135938213062275647987966122258633068750422765412656757031356726396953590532832612651778614105809511

184958619671055538245418910475945095899028541147979048143735270577628327231945
665268332353893726049719331100014368919217906890227634599072098765455150806751
562502836985794033869438478816588227469607545927347042210669730531879014789384
625312982001289864835150801555122374527889052035994750757124189436848583839226
992569385070354114912652891019821818119019183868438099199312919032379528618617
430628365941310932061881292714895761756353065421318359584527145513453282486890
035170412508759441697383386508068971410351837868102038696149732588696068875395
447812851324057514147606069300611722179364711646227334652991062948541983908288
325193852955442541198569041139684456053663803165879586852064518625040243698064
060128209438658571721483111336624193582743807582523561078247718466 3

To implement ECDH, I followed the code example from pg. 703 of the 6th edition textbook and page 4 from Week9b(4108AsymmetricKeying). I generated a 256 bit random prime number, p.

```
def ECDH():
    start = time.time()
    # Following code example from pg. 703 of 6th edition textbook and Page 4 from Week9b(4108AsymmetricKeying)
    # Generating a random 256 bit prime number
    p = random_prime((2**256) - 1, (2**255) + 1)
```

I then set up the Elliptic Curve using p.

```
    F = GF(p)
    E = EllipticCurve(F, 5, 10)
    G = E.gen(0)
    q = E.order()
```

As Alice, I computed a secret value, n_a, in 2…q-1. Then I computed a public value p_a.

```
    # Alice computes a secret value n_a in 2...q-1
    n_a = randint(2, q-1)

    # Alice computes a public value p_a
    p_a = n_a * G
```

As Bob, I compute a secret value, n_b, in 2…q-1. Then I computed a public value, p_b.

```
    # Bob computes a secret value n_b in 2...q-1
    n_b = randint(2, q-1)

    # Bob computes a public value p_b
    p_b = n_b * G
```

I can then compute the shared value, k_a, as Alice, and the shared value, k_b, as Bob.

```
# Alice computes the shared value (k_a)
k_a = n_a * p_b

# Bob also computes the shared value (k_b)
k_b = n_b * p_a
```

If Alice's shared value and Bob's shared value are actually the same value, the shared secret has been successfully computed.

```
print("A's shared value (EDCH):", k_a, "\n")
print("B's shared value (EDCH):", k_b, "\n")

if (k_a == k_b):
    print("Using ECDH: A and B have jointly created a key for encryption.")

duration = end - start
print("Duration:", duration, "s", "\n")
```

To implement DH, I followed the code example from pg. 699 of the 6th edition textbook and Page 3 of the Week9b(4108AsymmetricKeying) course notes.

I set p to a pre-computed safety prime of length 3072 bits and then set g to a primitive root of p.

```
def DH():
    # Following code example from pg. 699 of the textbook and Page 3 of Week9b(4108AsymmetricKeying) course notes.
    # To be equivalent security to ECC, private key must be 256 bits, and the public key must be 3072 bits.

    start2 = time.time()

    # p is a safety prime of length 3072 bits, which was randomly generated from this link: https://2ton.com.au/getprimes/random/3072?callback=safePrimes.
    p = 5706506157234918077697620794757832850164288054563798372894135938213062275647987966122258633068750422765412656757031356726396953590532832612651778614105809511849586196710555382454189104759450958990285411479790481437352705776283277
    g = primitive_root(p)
```

I then set Alice's private value, x_a, to be random 256 bit integer. I determined Alice's public value, y_a, by computing (g**x_a) mod p.

```
    x_a = random.randint((2**255), (2**256) - 1) # Alice's private value
    y_a = pow(g, x_a, p) # Alice's public value
```

I set Bob private value, x_b, to be random 256 bit integer. I determined Bob's public value, y_b, by computing (g**x_b) mod p.

```
    x_b = random.randint((2**255), (2**256) - 1) # Bob's private value
    y_b = pow(g, x_b, p) # Bob's public value
```

I determined Alice's shared value by computing y_b**x_a, and Bob's shared value by computing y_a**x_b.

```
    # Shared value:
    a_shared = pow(y_b, x_a)
    b_shared = pow(y_a, x_b)

    end2 = time.time()
```

If Alice's shared value and Bob's shared value are actually the same value, the shared secret has been successfully computed.

```
# Shared value:
a_shared = pow(y_b, x_a)
b_shared = pow(y_a, x_b)

end2 = time.time()

print("A's shared value (DH):", a_shared, "\n")
print("B's shared value (DH):", b_shared, "\n")

if (a_shared == b_shared):
    print("Using DH: A and B have jointly created a key for encryption.")

duration2 = end2 - start2
print("Duration:", duration2, "s")
```

The main function calls all necessary functions; the times required to compute the shared secrets (using both ECDH and DH) are displayed.

```
def main():
    ECDH()
    DH()
```

The computation of the shared secret using ECDH is much more efficient than the computation of the shared secret using DH. ECDH required the use of a 256 bit prime integer, while DH required the use of a 3072 bit prime integer, to reach the same level of security. Performing computations using the elliptic curves in ECDH proved more beneficial to the runtime than using extremely large integers in DH. It should be noted that the primitive_root function used in the code for DH is not optimized, despite the use of a safe prime to compute the value g. This function may be slowing down the DH process.

Examples of timing outputs (5 consecutive trials comparing ECDH and DH):

Trial 1:

```
(base) nicholasgin@Nicholass-MacBook-Pro ~ % sage /Users/nicholasgin/Desktop/csi4108_a3/q3b.sage
A's shared value (EDCH): (81706403326033327237705864791206870205940302847494369713821642241539540167774 : 5999581772576432959221045259045299380832943354390091688888455013414236
4121773 : 1)

B's shared value (EDCH): (81706403326033327237705864791206870205940302847494369713821642241539540167774 : 5999581772576432959221045259045299380832943354390091688888455013414236
4121773 : 1)

Using ECDH: A and B have jointly created a key for encryption.
Duration: 1.2859199047088623 s

A's shared value (DH): 737837111033414505745320557593049254327490943837152600119946114599436190635216835678331804667583478894605143033441459037603001981382537221724548634641517988716289821313439440793841457607594499773761849891324183553810309105125785475041097621659085041945886738404123298931786457354674187114553588662529461501436411055175034259132883916604611809122186503948268550955614161797059792037733146787936483017998933105648138150752359234508281054539261760244055617703278302836205508657761944938872018313939506076230733008556435458453085970730103946375499051473483639394801047411163234524116805470944086080730870079991014801424185053641856698581004325623542977557587806560270070989035196505513911812427319938359544809932780551682528837511617544847746467194861881260640982689573500847370204483888730690715028231766845474890262250929567794061754431525868641537679342555117452662158378197440260545508073594060370461665745460987502841806

B's shared value (DH): 737837111033414505745320557593049254327490943837152600119946114599436190635216835678331804667583478894605143033441459037603001981382537221724548634641517988716289821313439440793841457607594499773761849891324183553810309105125785475041097621659085041945886738404123298931786457354674187114553588662529461501436411055175034259132883916604611809122186503948268550955614161797059792037733146787936483017998933105648138150752359234508281054539261760244055617703278302836205508657761944938872018313939506076230733008556435458453085970730103946375499051473483639394801047411163234524116805470944086080730870079991014801424185053641856698581004325623542977557587806560270070989035196505513911812427319938359544809932780551682528837511617544847746467194861881260640982689573500847370204483888730690715028231766845474890262250929567794061754431525868641537679342555117452662158378197440260545508073594060370461665745460987502841806

Using DH: A and B have jointly created a key for encryption.
Duration: 139.94026231765747 s
```

Trial 2:

```
(base) nicholasgin@Nicholass-MacBook-Pro ~ % sage /Users/nicholasgin/Desktop/csi4108_a3/q3b.sage
A's shared value (EDCH): (58347297033356083963173780267586865457082682364046137712150794730726013291329 : 51509467843971738906905846150806262663497302407845886283457166217038920433922 : 1)

B's shared value (EDCH): (58347297033356083963173780267586865457082682364046137712150794730726013291329 : 51509467843971738906905846150806262663497302407845886283457166217038920433922 : 1)

Using ECDH: A and B have jointly created a key for encryption.
Duration: 3.6523520946502686 s

A's shared value (DH): 22078785783055160535733654649740859494812787760688005319797899584402243912910367685636054013144830663108586115453206039968501908160123972230035274940824642984619772771779605380892579396774484074142453942126032660049468949013316461596142460730060292753774102805474723787341894981194946056475123429615389030588923019170080578459621968173957780676946897393825587003085922153087083477539655984736167503947272666711739470551215244684656131491157538422921524237207637799517949157010200760732418714871516345955485045982540989989866587633424502221694796744966000903099945776941124718487833837043445156437072835891571645050448193028807412874806606786410746701550822177088773846374874788566742246302444961265055509486262638508789442171829981917051467458280439527690046755928144217257665814665713169815040431201042864731546478557751521985072328710948160267201923935893179784912654743327287517989448502490045423132921466638372469400 3

B's shared value (DH): 22078785783055160535733654649740859494812787760688005319797899584402243912910367685636054013144830663108586115453206039968501908160123972230035274940824642984619772771779605380892579396774484074142453942126032660049468949013316461596142460730060292753774102805474723787341894981194946056475123429615389030588923019170080578459621968173957780676946897393825587003085922153087083477539655984736167503947272666711739470551215244684656131491157538422921524237207637799517949157010200760732418714871516345955485045982540989989866587633424502221694796744966000903099945776941124718487833837043445156437072835891571645050448193028807412874806606786410746701550822177088773846374874788566742246302444961265055509486262638508789442171829981917051467458280439527690046755928144217257665814665713169815040431201042864731546478557751521985072328710948160267201923935893179784912654743327287517989448502490045423132921466638372469400

Using DH: A and B have jointly created a key for encryption.
Duration: 141.6697759628296 s
```

Trial 3:

```
(base) nicholasgin@Nicholass-MacBook-Pro ~ % sage /Users/nicholasgin/Desktop/csi4108_a3/q3b.sage
A's shared value (EDCH): (233227011894377187364087057277600917401421643957482384240043322424810923705501 : 25379213368452133979257606302431785028559990222419288479435779227368332965707 : 1)

B's shared value (EDCH): (233227011894377187364087057277600917401421643957482384240043322424810923705501 : 25379213368452133979257606302431785028559990222419288479435779227368332965707 : 1)

Using ECDH: A and B have jointly created a key for encryption.
Duration: 39.549859046936035 s

A's shared value (DH): 2891611674629961566793063557482335227970739618413572046938191101984687583725732295763082464723260467256209439258714210660637436126415600396233839620899197438766207857220182843735943989396638549469429054820267476298394389969074846805748348080658632602116501009622363816854953213817297290262709809592834395784378420343452881554335363929233182576797434151514914916585080778406272555856050512876097522615545516786847003560481148138243404067797828120253973001322210861965212980088557515250483271523536821781706170117306722796564756046367765102640908849438748191301941686930504599018078785526087204418867586089165214963262207774249463614493216590919426136325030219782878488128151664186234498018457356134241481130630115777741858555806755922258119637212879132197467508534607951507496836241962660320865309072104950799099160485556359715637483551972185909105727460751048496982417523950343697176942712194933874708121987421073208780150

B's shared value (DH): 2891611674629961566793063557482335227970739618413572046938191101984687583725732957630824647232604672562094392587142106606374361264156003962338396208991974387662078572201828437359439893966385494694290548202674762983943899690748468057483480806586326021165010096223638168549532138172972902627098095928343957843784203434528815540345335639292331825767974341515149149165850807784062725558560505128760975226155455167868470035604811481382434046077978281202539730013222108619652129800885575152504832715235368279217061701173067227956547560463677651026409084894387481913019416869305045990180787855260872044188675860891652149632622077742494636144932165909194261363250302197828784881281516641862344980184573561342414811306301157777418585558067559222581196372128791321974675085346079515074968362419626603208653090721049507990991604855563597156374835519721859091057274057510484969824175239503436971769427121949338747081219874210732087801503 3

B's shared value (DH): 2891611674629961566793063557482335227970739618413572046938191101984687583725732957630824647232604672562094392587142106606374361264156003962338396208991974387662078572201828437359439893966385494694290548202674762983943899690748468057483480806586326021165010096223638168549532138172972902627098095928343957843784203434528815540345335639292331825767974341515149149165850807784062725558560505128760975226155455167868470035604811481382434046077978281202539730013222108619652129800885575152504832715235368279217061701173067227956547560463677651026409084894387481913019416869305045990180787855260872044188675860891652149632622077742494636144932165909194261363250302197828784881281516641862344980184573561342414811306301157777418585558067559222581196372128791321974675085346079515074968362419626603208653090721049507990991604855563597156374835519721859091057274057510484969824175239503436971769427121949338747081219874210732087801503

Using DH: A and B have jointly created a key for encryption.
Duration: 143.61451387405396 s
```

Trial 4:

```
(base) nicholasgin@Nicholass-MacBook-Pro ~ % sage /Users/nicholasgin/Desktop/csi4108_a3/q3b.sage
A's shared value (EDCH): (44295526601612102498459714772645325123824197828451738805458764018816880669214 : 31053208368627713970175451642928620073985563929291955360100890657993591949208 : 1)

B's shared value (EDCH): (44295526601612102498459714772645325123824197828451738805458764018816880669214 : 31053208368627713970175451642928620073985563929291955360100890657993591949208 : 1)

Using ECDH: A and B have jointly created a key for encryption.
Duration: 1.8455917835235596 s

 A's shared value (DH): 1044436033590775808035918936520221427518709995602761117084441395610571499456598462461090115284194445285199532282013512015761443152031397668321679484922890060722795179790251056730452769059822213613545235748932741266956192871665864102392806650480243800894345857623194017708860759928007974398510643980440369359123917892328139742327841672335464447355404011196543642878293037282182229094445719559898513815537949445803991736533156445022939291386776887372623488647248139242046352764280096510800597933389908043285363164988419994057138694558754582765182379509366223662038392256670604802087055806388684503011400185549292826503423242888141807555277152781735640401470664369481825668097486574741857484628260678332372742789017466662199675048425577289048088055415883620131559617384422150590355159872804315421899561928091043423244739262075863909784780997938020086104561892114784227946690376535777852871980193456524581755324001483199939817941 4

B's shared value (DH): 1044436033590775808035918936520221427518709995602761117084441395610571499456598462461090115284194445285199532282013512015761443152031397668321679484922890060722795179790251056730452769059822213613545235748932741266956192871665864102392806650480243800894345857623194017708860759928007974398510643980440369359123917892328139742327841672335464447355404011196543642878293037282182229094445719559898513815537949445803991736533156445022939291386776887372623488647248139242046352764280096510800597933389908043285363164988419994057138694558754582765182379509366223662038392256670604802087055806388684503011400185549292826503423242888141807555277152781735640401470664369481825668097486574741857484628260678332372742789017466662199675048425577289048088055415883620131559617384422150590355159872804315421899561928091043423244739262075863909784780997938020086104561892114784227946690376535777852871980193456524581755324001483199939817941

Using DH: A and B have jointly created a key for encryption.
Duration: 139.90181231498718 s
```

Trial 5:

```
(base) nicholasgin@Nicholass-MacBook-Pro ~ % sage /Users/nicholasgin/Desktop/csi4108_a3/q3b.sage
A's shared value (EDCH): (11159816047943357582713730307381313287344988924451959687866929753324535151402 : 68888182259985392635408474002941580058287222908514625532555018945518058308459 : 1)

B's shared value (EDCH): (11159816047943357582713730307381313287344988924451959687866929753324535151402 : 68888182259985392635408474002941580058287222908514625532555018945518058308459 : 1)

Using ECDH: A and B have jointly created a key for encryption.
Duration: 16.985147953033447 s

A's shared value (DH): 4210695310264835458401461349085697495440027824634289426295191532396796199583540705548649420628867412187776504285503336491570354352325417548985586735136868755620093473982074073536993240034275842716376600261225181388664387139070119857982767595240179784390944194749294146444891637152536075315226180710892219991338953966577636390402334289973579043895037779729022932406440319349873793096717454251318234184884270700156556869981428798679442956485949351311118322655375651380582884381688657418781611413891678960683332868507904774677869465495787145108110921437474930805895563381113537115731302853268088209625976911765083929126849060661584575850806161480183591827479242061538712605862207004715991636544563389495883572029516092217914127061282083766048997592895820875435803824630650705364205410652884258085312988183930577973724276528644623530666892794305627294991238593125803490135458796348291458031391388924313396795302099907214128490267610 1

B's shared value (DH): 4210695310264835458401461349085697495440027824634289426295191532396796199583540705548649420628867412187776504285503336491570354352325417548985586735136868755620093473982074073536993240034275842716376600261225181388664387139070119857982767595240179784390944194749294146444891637152536075315226180710892219991338953966577636390402334289973579043895037779729022932406440319349873793096717454251318234184884270700156556869981428798679442956485949351311118322655375651380582884381688657418781611413891678960683332868507904774677869465495787145108110921437474930805895563381113537115731302853268088209625976911765083929126849060661584575850806161480183591827479242061538712605862207004715991636544563389495883572029516092217914127061282083766048997592895820875435803824630650705364205410652884258085312988183930577973724276528644623530666892794305627294991238593125803490135458796348291458031391388924313396795302099907214128490267610

Using DH: A and B have jointly created a key for encryption.
Duration: 140.07601809501648 s
```