1. Answers
   (a) I would see which key K_1 matched the plaintext/ciphertext pair for $X = E_{K_1}(P)$ and the two keys K_2, K_3 that matched the plaintext/ciphertext pair for $X = E_{K_2}(D_{K_3}(C))$. The found keys would be the answer since they both align to the same result.
   (b) $2^{112} + 2^56$ keys (2 keys + 1 key)
   (c) 2^56 space (each key is only 56 bits)
2. Answers
   (a) I am assuming that there is a private key to complement the public key. I am also assuming that the public key cipher with the public and private keys can be used for digital signatures. I also assume A and B have synchronized clocks.
   (b) $E_{KR_A}(timestamp)$
   (c) B will decrypt the message with KU_A to get the timestamp and then check if the timestamp is recent enough to be the current virtual circuit.
   (d) BG will not have the private key of A, so when A signs the timestamp B can be assured that A is the sender when decrypting with KU_A. The timestamp functions as a way to ensure that the message is recent enough to be used and can't be replayed by BG.
3. Answers
   (a)

$$\begin{aligned}
8^1 \pmod{11} &\equiv 8 \pmod{11}\\
8^2 \pmod{11} \equiv 8^1 \pmod{11} \times 8^1 \pmod{11} &\equiv 9 \pmod{11}\\
8^3 \pmod{11} \equiv 8^2 \pmod{11} \times 8^1 \pmod{11} &\equiv 6 \pmod{11}\\
8^4 \pmod{11} \equiv 8^3 \pmod{11} \times 8^1 \pmod{11} &\equiv 4 \pmod{11}\\
8^5 \pmod{11} \equiv 8^4 \pmod{11} \times 8^1 \pmod{11} &\equiv 10 \pmod{11}\\
8^6 \pmod{11} \equiv 8^5 \pmod{11} \times 8^1 \pmod{11} &\equiv 3 \pmod{11}\\
8^7 \pmod{11} \equiv 8^6 \pmod{11} \times 8^1 \pmod{11} &\equiv 2 \pmod{11}\\
8^8 \pmod{11} \equiv 8^7 \pmod{11} \times 8^1 \pmod{11} &\equiv 5 \pmod{11}\\
8^9 \pmod{11} \equiv 8^8 \pmod{11} \times 8^1 \pmod{11} &\equiv 7 \pmod{11}\\
8^{10} \pmod{11} \equiv 8^9 \pmod{11} \times 8^1 \pmod{11} &\equiv 1 \pmod{11}
\end{aligned}$$

   There are no repeat modular values
   $\therefore$ 8 is a primitive root of 11.
   (b) Solve for X_A given $Y_A = 6, q = 11, \alpha = 8$

$$6 = 8^{X_A} \pmod{11}$$

$$\begin{aligned}
8^1 \pmod{11} &\equiv 8 \pmod{11}\\
8^2 \pmod{11} \equiv 8^1 \pmod{11} \times 8^1 \pmod{11} &\equiv 9 \pmod{11}\\
8^3 \pmod{11} \equiv 8^2 \pmod{11} \times 8^1 \pmod{11} &\equiv 6 \pmod{11}
\end{aligned}$$

   $8^3 \pmod{11} \equiv 6 = Y_A$
   $\therefore X_A = 3$
   (c) Solve for X_B given $Y_B = 3, q = 11, \alpha = 8$

$$3 = 8^{X_B} \pmod{11}$$

$$\begin{aligned}
8^1 \pmod{11} &\equiv 8 \pmod{11}\\
8^2 \pmod{11} \equiv 8^1 \pmod{11} \times 8^1 \pmod{11} &\equiv 9 \pmod{11}\\
8^3 \pmod{11} \equiv 8^2 \pmod{11} \times 8^1 \pmod{11} &\equiv 6 \pmod{11}\\
8^4 \pmod{11} \equiv 8^3 \pmod{11} \times 8^1 \pmod{11} &\equiv 4 \pmod{11}\\
8^5 \pmod{11} \equiv 8^4 \pmod{11} \times 8^1 \pmod{11} &\equiv 10 \pmod{11}\\
8^6 \pmod{11} \equiv 8^5 \pmod{11} \times 8^1 \pmod{11} &\equiv 3 \pmod{11}
\end{aligned}$$

$8^6 \pmod{11} \equiv 3 = Y_B$
$\therefore X_B = 6$

(d) Given the equation $K = {Y_B}^{X_A} \pmod q$ where $Y_B = 3$, $X_A = 3$, and $q = 11$
$\therefore K = 3^3 \pmod{11} = 5$

(e) Given the equation $K = {Y_A}^{X_B} \pmod q$ where $Y_A = 6$, $X_B = 6$, and $q = 11$
$\therefore K = 6^6 \pmod{11} = 5$

4. Answers

(a) Given the equation $Y_A = \alpha^{X_A} \pmod q$, $q = 11$, $\alpha = 7$, and $X_A = 6$:

$$Y_A = 7^6 \pmod{11} \equiv 4$$

$\therefore Y_A = 4$

(b) Given the equation $K = {Y_A}^k \pmod q$, $\alpha = 7$, $q = 11$, $k = 2$, $M = 3$, and $Y_A = 4$:

$$K = 4^2 \pmod{11} \equiv 5$$

$\therefore K = 5$
Using the equations $C_1 = \alpha^k \pmod q$ and $C_2 = KM \pmod q$, it follows:

$$C_1 = 7^2 \pmod{11} \equiv 5$$
$$C_2 = 5 \times 3 \pmod{11} \equiv 4$$

$\therefore C_1 = 5, C_2 = 4$

(c) Given the equations $K = {C_1}^{X_A} \pmod q$ and $M = C_2K^{-1} \pmod q$, $C_1 = 5$, $C_2 = 4$, and $X_A = 6$, it follows:

$$K = 5^6 \pmod{11} \equiv 5$$

$\therefore K = 5$
K^{-1} is calculated by testing all possible values:

$$5 \times 0 \equiv 0 \pmod{11}$$
$$5 \times 1 \equiv 5 \pmod{11}$$
$$5 \times 2 \equiv 10 \pmod{11}$$
$$5 \times 3 \equiv 4 \pmod{11}$$
$$5 \times 4 \equiv 9 \pmod{11}$$
$$5 \times 5 \equiv 3 \pmod{11}$$
$$5 \times 6 \equiv 8 \pmod{11}$$
$$5 \times 7 \equiv 2 \pmod{11}$$
$$5 \times 8 \equiv 7 \pmod{11}$$
$$5 \times 9 \equiv 1 \pmod{11}$$
$$\therefore K^{-1} = 9$$

From this, M can be calculated as following:

$$M = 4 \times 9 \pmod{11} \equiv 3$$

$\therefore M$ is calculated to be 3, and A has successfully recovered the message.

5. Answers

(a) The BG could take the public key of B and replace it with their public key and send that to A.

(b) BG could take an old transmission from round 2 and send that to A instead of the legitimate transmission. The old transmission might have an old public key for B which the BG could use as their new public key.

(c) In round 6, BG could send a nonce N_2 to A and A would have no way of confirming it came from B. The nonce in round 3 is only known to A and B, so without it A cannot confirm B's identity.

(d) BG could request their public key from the authority and send it to A in round 2. A would be forced to assume that this is the response they wanted when in reality it would be the BG's public key.

Programming Problem

Self-critique

The program works as expected and its answers are verifiably correct. The only issue is the chosen plaintext for the third set of attacks. Since the plaintext is only 67 and not something higher, the first brute force attack is significantly faster than the second factoring attack. I do not believe that the first attack is faster than the second attack unless the best case occurs where M is close to 1, as seen in the program.

Program Output

```
hoodz@hoodz-laptop:/mnt/c/Users/hoodi/Desktop/Coding/CSCI-4534-Cryptography/Homework
    -2$ ./main
------------------------------------------------------
~ RSA Brute Force Attacker Program ~
------------------------------------------------------
Running Problem 1...
------------------------------------------------------
=========================
Ciphertext: 10
Public Key: 7
Number: 15
==========================
Running Attack One...
Plaintext: 10
Time to complete AttackOne: 500 nanoseconds

Running Attack Two...
Prime P: 5
Prime Q: 3
Private Key: 7
Plaintext: 10
Time to complete AttackTwo: 300 nanoseconds

------------------------------------------------------
Running Problem 2...
------------------------------------------------------
=========================
Ciphertext: 356
Public Key: 13
Number: 527
==========================
Running Attack One...
Plaintext: 271
Time to complete AttackOne: 8700 nanoseconds

Running Attack Two...
Prime P: 31
Prime Q: 17
Private Key: 37
Plaintext: 271
Time to complete AttackTwo: 1600 nanoseconds

------------------------------------------------------
Running Problem 3...
------------------------------------------------------
=========================
Ciphertext: 8567
Public Key: 53
Number: 21583
==========================
Running Attack One...
Plaintext: 67
Time to complete AttackOne: 3400 nanoseconds

Running Attack Two...
Prime P: 191
Prime Q: 113
Private Key: 20477
```

```
Plaintext: 67
Time to complete AttackTwo: 53000 nanoseconds

------------------------------------------------------
~ End of Program ~
hoodz@hoodz-laptop:/mnt/c/Users/hoodi/Desktop/Coding/CSCI-4534-Cryptography/Homework-2$
```

Human-readable code

main.cpp

```cpp
#include <chrono>
#include <iostream>
#include <iomanip>

#include "RSAAttacker.hpp"

using namespace std;

/// function for testing homework problems
void Problem(RSAAttacker& attack, Members info);

int main(){
    cout << "-------------------------------------------------------" << endl;
    cout << "~ RSA Brute Force Attacker Program ~" << endl;

    // Blank initialization
    RSAAttacker attack(0, 0, 0);

    // formatted as public key, number, ciphertext
    // private key = 7, plaintext = 10, primes = 3 & 5, totient = 8
    Members q1 { 7, 15, 10 };

    // private key = 37, plaintext = 271, primes = 17 & 31, totient = 480
    Members q2 { 13, 527, 356 };

    // private key = 20477 11143, plaintext = 67, primes = 113 & 191, totient = 21280
    Members q3 { 53, 21583, 8567 };

    cout << "-------------------------------------------------------" << endl;
    cout << "Running Problem 1..." << endl;
    cout << "-------------------------------------------------------" << endl;

    Problem(attack, q1);

    cout << "-------------------------------------------------------" << endl;
    cout << "Running Problem 2..." << endl;
    cout << "-------------------------------------------------------" << endl;

    Problem(attack, q2);

    cout << "-------------------------------------------------------" << endl;
    cout << "Running Problem 3..." << endl;
    cout << "-------------------------------------------------------" << endl;

    Problem(attack, q3);

    cout << "-------------------------------------------------------" << endl;
    cout << "~ End of Program ~" << endl;

    return 0;
}

/// @brief          a helper function to run both RSAAttacker functions on one problem set
/// @param attack   (RSAAttacker) any object that will get its values changed with info param
/// @param info     (struct Members) the specified public key, number, and ciphertext xtruct
void Problem(RSAAttacker& attack, Members info){
    // read in the information into the attacker
    attack.SetInfo(info);

    cout << "=========================" << endl;
    cout << "Ciphertext: " << info.ciphertext << endl;
    cout << "Public Key: " << info.encryptionKey << endl;
    cout << "Number:     " << info.number << endl;
    cout << "==========================" << endl;


    cout << "Running Attack One..." << endl;
    const auto start1 = chrono::high_resolution_clock::now();
    const int plaintext = attack.AttackOne();
    const auto end1 = chrono::high_resolution_clock::now();
    cout << "Plaintext: " << plaintext << endl;
```

```cpp

    const std::chrono::duration<double> time1 = end1 - start1;
    const auto timeCast1 = chrono::duration_cast<chrono::nanoseconds>(end1 - start1);

    cout << "Time to complete AttackOne: " << timeCast1.count() << " nanoseconds\n" << endl;

    cout << "Running Attack Two..." << endl;
    const auto start2 = chrono::high_resolution_clock::now();
    const attackTwoReturn vals = attack.AttackTwo();
    const auto end2 = chrono::high_resolution_clock::now();

    cout << "Prime P:     " << vals.primeP << endl;
    cout << "Prime Q:     " << vals.primeQ << endl;
    cout << "Private Key: " << vals.privKey << endl;
    cout << "Plaintext:   " << vals.plaintext << endl;

    const auto timeCast2 = chrono::duration_cast<chrono::nanoseconds>(end2 - start2);

    cout << "Time to complete AttackTwo: " << timeCast2.count() << " nanoseconds\n" << endl;
}
```

RSAAttacker.hpp

```cpp
#ifndef RSA_ATTACKER_HPP
#define RSA_ATTACKER_HPP

/// @brief internal data members of RSAAttacker class
struct Members {
    int encryptionKey;  // public key of RSA
    int number;         // number N of RSA
    int ciphertext;     // ciphertext from m ^ eKey mod number
};

/// @brief return value for AttackTwo to aggregate data and return one item
struct attackTwoReturn {
    int primeP;     // calculated primes from RSA
    int primeQ;
    int privKey;    // private key/decryption key associated with RSA encryption key
    int plaintext;  // calculated message m where m = c ^ dKey mod number
};


class RSAAttacker{
private:
    Members info;   // aggregated data members, see Members struct

public:
    /// ----------------------------------------------------------------------------
    /// @brief              main constructor for RSAAttacker helper class
    /// @param eKey         (int) encryption key/private key of RSA
    /// @param number       (int) calculated number N of RSA
    /// @param ciphertext   (int) valid ciphertext of RSA derived from m ^ eKey mod number (m is unknown to
     class)
    /// @note               this class does not check for validness of inputs
    /// ----------------------------------------------------------------------------
    RSAAttacker(int eKey, int number, int ciphertext);

    /// ----------------------------------------------------------------------------
    /// @brief          constructor for RSAAttacker helper class that allows assignment using Members struct
    /// @param info     (struct Members) values needed for attacker, see RSAAttacker(int, int, int)
    /// @note           this class does not check for validness of inputs
    /// ----------------------------------------------------------------------------
    RSAAttacker(Members info);

    ~RSAAttacker() = default;

    /// ----------------------------------------------------------------------------
    /// @brief          setter to change out internal information
    /// @param info     (struct Members) struct values to change to
    /// ----------------------------------------------------------------------------
    void SetInfo(Members info) { this->info = info; }

    /// ----------------------------------------------------------------------------
    /// @brief      calculates the Euler totient of two prime numbers
```

```
    /// @param p    (int) a prime number
    /// @param q    (int) a prime number
    /// @return     (int) totient of primes p and q
    /// ---------------------------------------------------------------------------------
    int eulerTotient(int p, int q) const { return (p-1)*(q-1); }

    /// ---------------------------------------------------------------------------------
    /// @brief  run through all possible values of m until [m ^ eKey mod number = ciphertext] is found
    /// @return (int) plaintext associated with the inputted ciphertext from info
    /// ---------------------------------------------------------------------------------
    int AttackOne();

    /// ---------------------------------------------------------------------------------
    /// @brief  factor inputted info.number into its two primes and get plaintext by calculating private key from them
    /// @return (struct attackTwoReturn) following calculated values via RSA algorithm: { (int) prime P, (int) prime Q, (int) private key, (int) plaintext }
    /// ---------------------------------------------------------------------------------
    attackTwoReturn AttackTwo();

    /// ---------------------------------------------------------------------------------
    /// @brief      solve a ^ b mod n fast using less multiplications
    /// @param a    (int) positive base of the equation
    /// @param b    (int) positive exponent of the equation
    /// @param n    (int) modulus value of the equation
    /// @return     (int) result of a ^ b mod n
    /// ---------------------------------------------------------------------------------
    int fastModExponentiation(int a, int b, int n) const;

    /// ---------------------------------------------------------------------------------
    /// @brief      helper function to calculate modular inverse, ax = 1 mod n
    /// @param a    (int) positive known term in the above equation
    /// @param n    (int) modulus value of the equation
    /// @return     (int) modular inverse of a mod n
    /// @throw      if the modular inverse cannot be found, a const char* message notification is thrown
    /// ---------------------------------------------------------------------------------
    int modInverse(int a, int n) const;

    /// ---------------------------------------------------------------------------------
    /// @brief      uses the extended Euclidean algorithm to find the solution to ax = b mod n
    /// @param a    (int) positive known term in the above equation
    /// @param b    (int) positive congruent term in the above equation
    /// @param n    (int) modulus value of the equation
    /// @return     (int) the solution x to the equation ax = b mod n
    /// @throw      if the modular inverse cannot be found, a const char* message notification is thrown
    /// ---------------------------------------------------------------------------------
    int modLinearEquationSolver(int a, int b, int n) const;

    /// ---------------------------------------------------------------------------------
    /// @brief      the extended Euclidean algorithm to find the greatest common denominator
    /// @note       remainder = ax + by = GCD(a, b)
    /// @param a    (int) the first term to find the GCD with
    /// @param b    (int) the second term to find the GCD with
    /// @param x    (int*) return parameter to return Bezout coefficient x
    /// @param y    (int*) return parameter to return Bezout coefficient y
    /// @return     (int) the GCD of a & b, as well as Bezout coefficients in x & y
    /// ---------------------------------------------------------------------------------
    int gcdExtended(int a, int b, int* x, int* y) const;
};

#endif
```

RSAAttacker.cpp

```
#include "RSAAttacker.hpp"

/// ---------------------------------------------------------------------------------
/// @brief              main constructor for RSAAttacker helper class
/// @param eKey         (int) encryption key/private key of RSA
/// @param number       (int) calculated number N of RSA
/// @param ciphertext   (int) valid ciphertext of RSA derived from m ^ eKey mod number (m is unknown to class)
/// @note               this class does not check for validness of inputs
/// ---------------------------------------------------------------------------------
RSAAttacker::
```

```
RSAAttacker(int eKey, int number, int ciphertext) {
    info.encryptionKey = eKey;
    info.number = number;
    info.ciphertext = ciphertext;
}

/// ---------------------------------------------------------------------------------
/// @brief          constructor for RSAAttacker helper class that allows assignment using Members struct
/// @param info     (struct Members) values needed for attacker, see RSAAttacker(int, int, int)
/// @note           this class does not check for validness of inputs
/// ---------------------------------------------------------------------------------
RSAAttacker::
RSAAttacker(Members info) : info(info) {}

/// ---------------------------------------------------------------------------------
/// @brief  run through all possible values of m until [m ^ eKey mod number = ciphertext] is found
/// @return (int) plaintext associated with the inputted ciphertext from info
/// ---------------------------------------------------------------------------------
int RSAAttacker::
AttackOne(){
    int guess = 0, m = 0;

    while (guess != info.ciphertext)
        guess = fastModExponentiation(++m, info.encryptionKey, info.number);

    return m;
}

/// ---------------------------------------------------------------------------------
/// @brief  factor inputted info.number into its two primes and get plaintext by calculating private key from them
/// @return (struct attackTwoReturn) following calculated values via RSA algorithm: { (int) prime P, (int) prime Q, (int) private key, (int) plaintext }
/// ---------------------------------------------------------------------------------
attackTwoReturn RSAAttacker::
AttackTwo(){
    // calculate the two prime factors (naively)
    int p, q;
    for (int i = 1; i < info.number; i++){
        if (info.number % i == 0){
            p = i; q = info.number / p;
        }
    }
    // calculate the totient of the primes
    int totient = eulerTotient(p, q);

    // calculate the private key by getting the inverse of eKey mod totient
    int privateKey = modInverse(info.encryptionKey, totient);

    // calculate plaintext from c ^ dKey mod number
    int m = fastModExponentiation(info.ciphertext, privateKey, info.number);

    return attackTwoReturn { p, q, privateKey, m };
}

/// ---------------------------------------------------------------------------------
/// @brief      solve a ^ b mod n fast using less multiplications
/// @param a    (int) positive base of the equation
/// @param b    (int) positive exponent of the equation
/// @param n    (int) modulus value of the equation
/// @return     (int) result of a ^ b mod n
/// @note       relies on the idea that modding smaller parts gets same result
/// @example    a^5 mod n = a^2 mod n * a^2 mod n * a mod n
/// ---------------------------------------------------------------------------------
int RSAAttacker::
fastModExponentiation(int a, int b, int n) const {
    int result = 1;
    a = a % n;

    while (b > 0){
        if (b & 1) result = (result * a) % n;       // if b is odd, set the new result and mod it by n (result * [a mod n])
        b >>= 1;                                    // another a^2 mod n is added, divide exponent b by 2
        a = (a * a) % n;                            // a = a^2 mod n
    }

```

```cpp
    return result;
}

/// ---------------------------------------------------------------------------------
/// @brief      helper function to calculate modular inverse, ax = 1 mod n
/// @param a    (int) positive known term in the above equation
/// @param n    (int) modulus value of the equation
/// @return     (int) modular inverse of a mod n
/// @throw      if the modular inverse cannot be found, a const char* message notification is thrown
/// ---------------------------------------------------------------------------------
int RSAAttacker::
modInverse(int a, int n) const {
    return modLinearEquationSolver(a, 1, n);
}

/// ---------------------------------------------------------------------------------
/// @brief      uses the extended Euclidean algorithm to find the solution to ax = b mod n
/// @param a    (int) positive known term in the above equation
/// @param b    (int) positive congruent term in the above equation
/// @param n    (int) modulus value of the equation
/// @return     (int) the solution x to the equation ax = b mod n
/// @throw      if the modular inverse cannot be found, a const char* message notification is thrown
/// ---------------------------------------------------------------------------------
int RSAAttacker::
modLinearEquationSolver(int a, int b, int n) const {
    int x,              // first Bezout coefficient from gcdExtended
        y;              // second Bezout coefficient from gcdExtended

    int gcd = gcdExtended(a, n, &x, &y);    // get the greatest common denominator and the Bezout coefficients
    if (b % gcd == 0){
        return ((x % n + n) % n);           // make sure solution x is within modulus range and return it
    }

    throw "Modular Linear Equation Solver cannot return a value";
}

/// ---------------------------------------------------------------------------------
/// @brief      the extended Euclidean algorithm to find the greatest common denominator
/// @note       remainder = ax + by = GCD(a, b)
/// @param a    (int) the first term to find the GCD with
/// @param b    (int) the second term to find the GCD with
/// @param x    (int*) return parameter to return Bezout coefficient x
/// @param y    (int*) return parameter to return Bezout coefficient y
/// @return     (int) the GCD of a & b, as well as Bezout coefficients in x & y
/// @cite       https://en.wikipedia.org/wiki/Extended_Euclidean_algorithm#Pseudocode
/// @date       4 Mar 2024
/// ---------------------------------------------------------------------------------
int RSAAttacker::
gcdExtended(int a, int b, int* x, int* y) const{
    int remainder   = a, newRemainder   = b;
    int bezoutX     = 1, bezoutXCalc    = 0;
    int bezoutY     = 0, bezoutYCalc    = 1;
    int quotient    = 0, temp           = 0;

    while (newRemainder != 0){                                  // while there is something to divide
        quotient = remainder / newRemainder;                    // get the quotient result
        temp = newRemainder;
        newRemainder = remainder - (quotient * newRemainder);   // calculate next remainder
        remainder = temp;

        temp = bezoutXCalc;
        bezoutXCalc = bezoutX - (quotient * bezoutXCalc);       // calculate next x Bezout coefficient
        bezoutX = temp;

        temp = bezoutYCalc;
        bezoutYCalc = bezoutY - (quotient * bezoutYCalc);       // calculate next y Bezout coefficient
        bezoutY = temp;
    }

    // cout << "Quotients of GCD: (" << bezoutYCalc << ", " << bezoutXCalc << ")" << endl;
    // cout << "Final GCD: " << remainder << endl;
    // cout << "Bezout coefficients: (" bezoutX << ", " << bezoutY << ")" << endl;

    *x = bezoutX; *y = bezoutY;                                 // assign return parameters and return GCD
    return remainder;
}
```