This quarter we have spend some time developing the fascinating mathematical ideas behind a number of modern crypto systems. For this final project, the focus will be on implementing the ideas we have developed. Some of these functions you have already developed ino your homework, others you will have to write yourself. I suggest using cocalc/python 2, as the helper code I will provide will be written in these languages, but you are welcome to use whichever language is most comfortable for you. We will spend the next three days doing this work in class, and I hope to be available to help however necessary.

Sumbission Guidelines: To turn in this work I will need a readout of all of your written code. This includes all the helper functions, and implementation of each of the 3 cryptosystems, and the results of the test cases assigned for each problem. This can be send as a shared cocalc document or a printout of your code.

I will also need you all to generate public keys for both ElGamal and RSA, which you will share with me. On the last day of class I will share public keys and you will all be able to send me messages using your decryption functions. Send me a short message describing your desired grade. I will decrypt, read, and use your public keys to respond, sending each of you ciphertext to decrypt. Send me an email with a quote of my decrypted message so I can see your functions worked.

1 Helper Functions

This first problem involves writing the the helper functions necessary to implement the Diffie-Hellman, ElGamal, and RSA cryptosystems. Some problems are followed by 2 test cases, whose results should be turned in along with your code.

1. Define a function called `divisionWithRemainder(a,b)`.

   - **Input:** 2 integers a and b.
   - **Output:** An array `[q,r]` of the whole number quotient and remainder. That the unique q and r such that $a = bq + r$ with $0 \leq r < b$.

   Test it on the following two pairs of integers:

   (a) $a = 37$ and $b = 16$.

   (b) $a = 85354354693$ and $b = 927256$

2. Define a function called `findGCD(a,b)`.

   - **Input:** 2 integers a and b.
   - **Output:** The greatest common denominator of a and b.

   You MUST use the Euclidean Algorithm rather than the brute force method, since this should be able to work with large numbers. Do not hesitate to use the `divisionWithRemainder(a,b)` function from question 1. Test your function on the following pairs of integers:

   (a) $a = 16534528044$ and $b = 8332745927$

   (b) $a = 170446930$ and $b = 596564255$

3. Define a function called `findInverse(a,N)`.

   - **Input:** An integer modulus N and an integer a.
   - **Output:** If a is invertible modulo N it will return the inverse of a in $\mathbb{Z}/N\mathbb{Z}$. If a has no inverse return an error message.

Use the extended Euclidean algorithm rather than guess an check, since this should be able to work with large numbers. Do not hesitate to use functions defined earlier. Since N is not necessarily prime, you cannot use Fermat's little theorem. Test your function on the following pairs of integers:

(a) $a = 9843$ and $N = 3006908448$

(b) $a = 8735$ and $N = 7956970$

4. Define a function called `fastPower(g,A,N)`.

   - **Input:** An integer g, integer exponent A and integer modulus N.
   - **Output:** Return $g^A \pmod N$ in $\log A$ time.

   This should not just compute g^A and reduce mod N, as for large A this will be extremely inefficient. Instead implement fast powering. You may want a helper function to convert A to binary, but a more elegant procedure is outlined in the textbook, problem 1.25 on page 53. Test your function on the following pairs of integers:

   (a) $g = 526$, $A = 1008006$, $N = 463824966$

   (b) $g = 2$, $A = 30069476292$, $N = 30069476293$

2 Diffie-Hellman

The following functions will together implement the Diffie-Hellman key exchange protocol. To begin, you should have global variables p and g, where p is a prime number and g is a principle root, that is, a generator of $(\mathbb{Z}/p\mathbb{Z})^*$. You also may want a global variable a where you can store your secret key.

1. Define a function `keyExchange(a,g,p)`.

   - **Input:** A prime number p, a principle root g, and your secret key a.
   - **Output:** Your half of the public key to swap with your partner.

   Notice, this should be a very short function using your helper functions from part 1.

2. Define a function `sharedSecret(a,B,p)`.

   - **Input:** A prime number p, the public key B that you receive from your partner, and your secret key a.
   - **Output:** You and your partner's shared secret.

   As above, your helper functions should do all the work for you.

3. Now find a partner. Agree on a large prime p and a principle generator g. You should each decide on (and NOT SHARE) secret keys. Now run `keyExchange` and swap your halves of the public key. Then run `sharedSecret` check that you defined a matching secret key.

3 ElGamal

The following functions will implement each side of the ElGamal asymetric crypto system. Don't be afraid of making liberal use of your helper functions from part 1. As in part 2, you should have global variables p and g where you can save a prime number and a principle root, as well as a global variable a where you can store your secret key.

We will first define functions you will need to receive messages.

1. Define a function `generatePublicKeyEG(a,g,p)`.

   - **Input:** A prime number p, principle root g, and secret key a.

- **Output:** Your public key.

This should look a lot like the function `keyExchange` from Diffie-Hellman.

2. Define a function `elGamalDecrypt(a,c1,c2,p)`.

   - **Input:** Your secret key a, the ciphertext c_1, c_2 you received, as well as the prime number p you used to define your public key.
   - **Output:** The plaintext message m.

   You will likely need to use your `findInverse` function from earlier.

What if you recieve a public key and would like to send a message to the person who generated it? The following function will allow you to encrypt the message into ciphertext you can safely send. Recall that the public key you receive should include a large prime p and a principle root g, make sure to save these.

1. Define a function `elGamalEncrypt(m,A,g,p)`.

   - **Input:** The prime and principle root p and g that the person your sending a message to decided on. Their public key A, and of course your message m, encoded as an integer.
   - **Output:** An array `[c1,c2]` consisting of the ElGamal ciphertext you can safely send to the recipient.

   Your function should select a different random number k each time it runs. I suggest using the random python library by putting `import random` at the top of your code. Then the line `random.randint(a,b)` will produce a 'random' number between a and b.

2. Now find a partner. Agree on a large prime p and a principle generator g. One of you (Alice) should decide on a secret key and generate a public key to share with the other (Bob). Bob now should choose a secret message and send ciphertext to Alice, which Alice can now decrypt. Swap rolls and do it all again.

4 RSA

The following functions will implement the RSA cryptosystem. If you are trying to receive message, you should save two large prime numbers p and q as global variables. These must be kept secret. You should also select an encryption exponent e. First let's define functions necessary to receive messages.

1. Define a function `generatePublicKeyRSA(p,q,e)`.

   - **Input:** Two primes p and q and an encryption exponent e.
   - **Output:** If $GCD(e,(p-1)(q-1)) = 1$ then return $N = pq$ and e. Otherwise return an error message instructing the user to select a new encryption exponent.

   You then publish N and e as your public key.

2. Define a function `computeDecryptionExponent(p,q,e)`.

   - **Input:** Two primes p and q and an RSA encryption exponent e.
   - **Output:** The RSA decryption exponent d.

   It would likely be useful to save d as a global variable so as to not have to repeatedly compute it.

3. Define a function `RSADecrypt(c,d,N)`.

   - **Input:** The ciphertext c, decryption exponent d, and modulus $N = pq$.
   - **Output:** The plaintext message you have been sent.

Notice that you don't actually need p and q if you already have your decryption exponent d and public key N computed.

If you are hoping to send someone a message, and they have shared e and N with you, the following will allow you to encrypt a message.

1. Define a function `RSAEncrypt(m,e,N)`.

   - **Input:** Your message m converted to an integer, and a the public key of person you are sending the message, which consists of an encryption exponent e and modulus N.
   - **Output:** RSA ciphertext c.

2. Now find a partner. One of you (Alice) should decide on a secret key and generate a public key to share with the other (Bob). Bob now should choose a secret message and send ciphertext to Alice, which Alice can now decrypt. Swap rolls and do it all again.