**Mathematics 3159A** Introduction to Cryptography

**Assignment 2** Due Date: October 13, 2020

## Part 1

#### Problem

Let $p$ be a prime and $a$ an integer not divisible by $p$. Show that the congruence

$$x^2 \equiv a \pmod{p}$$

has either two or no solutions in $\mathbb{Z}/$ p.

#### Solution

There are two cases: we either have a solution or we do not. If the system does not have a solution, we are done. If there is a solution, we will show there are exactly two distinct solutions. Take a solution $x_1$. We know

$$x_1^2 \equiv a \pmod{p}.$$

Thus, there exists $k_1 \in \mathbb{Z}$ so that $x_1^2 - a = k_1 p$, so $x_1^2 = k_1 p + a$. However, we see that this means $x_1$ can actually take on two satisfactory values,

$$x_1 = \sqrt{k_1 p + a} \qquad x_1 = -\sqrt{k_1 p + a}.$$

We know each of these are distinct, as the only way for them to be the same is if they are both equal to 0. However, this would require $-k_1 p = a$. This would imply $p|a$, however by our original assumption of $a$ we know this is not the case.

Thus, any one solution $x_1$ induces two distinct solutions. We will show these are the only two solutions. For suppose there exists $x_2$ which also satisfies our congruence. Then, there exists some $k_2 \in \mathbb{Z}$ so that $x_2^2 - a = k_2 p$. Recall now that $x_1^2 - a = k_1 p$. Thus,

$$x_2^2 - k_2 p = x_1^2 - k_1 p \implies (x_1 - x_2)(x_1 + x_2) = p(k_2 - k_1)$$

but this means

$$(x_1 - x_2)(x_1 + x_2) \equiv 0 \pmod{p}$$

which, as $p$ is prime and therefore $\mathbb{Z}/p$ is a field, implies that $x_1 = \pm x_2$ as one of our factors must be zero, and as there are positive and negative forms for $x_1$, we know that $x_2$ will match one of them. Thus, we only have two distinct solutions. $\square$

### 2.

#### Problem

Let $p$ and $q$ be distinct primes and $a$ an integer such that $gcd(a, pq) = 1$. Show that the congruence

$$x^2 \equiv a \pmod{pq}$$

has either four or no solutions in $\mathbb{Z}/$ pq.

**Solution**

We will call back upon the following quadratic system

$$\begin{cases} x^2 \equiv a \pmod{p} \\ x^2 \equiv a \pmod{q} \end{cases}.$$

We know $p$ and $q$ are coprime, and since $\gcd(a, pq) = 1$, then

$$\gcd(a, p) = 1 = \gcd(a, q).$$

Thus, we have the conditions necessary to apply the Chinese Remainder Theorem ('CRT') as well as question 1 of Part 1.

By the CRT, we know there is a unique integer $s$ where $0 \leq s < pq$ such that

$$\begin{cases} s \equiv a \pmod{p} \\ s \equiv a \pmod{q} \end{cases}.$$

Suppose now that solutions exist for both quadratic congruences introduced at the beginning. Then, we will have precisely two solutions for each, giving

$$\begin{cases} x_{p_1}^2 \equiv a \pmod{p} \\ x_{p_2}^2 \equiv a \pmod{p} \end{cases} \qquad \begin{cases} x_{q_1}^2 \equiv a \pmod{q} \\ x_{q_2}^2 \equiv a \pmod{q} \end{cases}.$$

However, this means $s$ is congruent to all these squares (under their respective primes). Now, recall from (1) that $x_{p_1} = -x_{p_2}$, and likewise for $q$. So, defining

$$c_p \equiv x_{p_1} \pmod{p},$$

we see $x_{p_2} \equiv -c_p \pmod{p}$, and likewise for $c_q$. Now, consider the linear system

$$\begin{cases} r_{1,1} \equiv c_p \pmod{p} \\ r_{1,1} \equiv c_q \pmod{q}. \end{cases}$$

We can solve for $r_{1,1}$ using CRT. Note then that $r_{1,1}^2 \equiv s \pmod{pq}$ by construction. In general, we denote $r_{\pm 1, \pm 1}$ to be the solution to the above congruence with $\pm c_p$ and $\pm c_q$ (respectively).

Note that the uniqueness of the solution given by CRT and the distinctness of our $x$ implies the distinctness of $r_{\pm 1, \pm 1}$. Hence, we have found precisely four distinct solutions to our desired congruence modulo $pq$. We know that we cannot have more than four solutions, as this would imply more than two solutions to the congruences modulo $q$ and modulo $p$, which we proved is not possible in Part 1 question 1. Thus, we have characterized all of our solutions.

The other case is that one of the quadratic congruences modulo $p$ or modulo $q$ has no solution. Without loss of generality, suppose it is for $p$, meaning there is no $x \in \mathbb{Z}$ such that

$$x^2 \equiv a \pmod{p}.$$

Suppose now that our $s$ from CRT did have a square root, $r$. That is,

$$r^2 \equiv s \equiv a \pmod{pq}.$$

However, this implies $r^2 \equiv a \pmod{p}$ by definition of $s$, which is a contradiction as we assumed there was no solution to the quadratic equation modulo $p$. □

**3.**

**Problem**

Based on your proof above, describe a polynomial-time algorithm that finds all solutions to the congruence

$$x^2 \equiv 1 \pmod{pq}.$$

**Solution**

It is clear that $x = 1$ is a trivial solution to this congruence. Thus, from (2) we know that there exist four unique solutions to the quadratic congruences modulo $p$ and modulo $q$, respectively $x_{p_1}, x_{p_2}, x_{q_1}, x_{q_2}$. As $1^2 \equiv 1 \pmod{pq}$, the CRT implies that

$$x_{p_1}^2 \equiv 1 \equiv x_{p_2}^2 \pmod{p}.$$

It is clear from here that $x_{p_1} = 1$ and $x_{p_2} = -1$ are the solutions, up to ordering. To obtain the other solutions to our desired congruence modulo $pq$, we apply the same technique from part 1 question 2 and have

$$\begin{cases} r_{1,1} \equiv 1 \pmod{p} \\ r_{1,1} \equiv 1 \pmod{q} \end{cases}$$

and likewise for the other $r$, where we note that $-1 \equiv p - 1 \pmod{p}$.

Hence, our algorithm will simply have 5 steps:

1. Solve for $r_{1,1}$ using CRT.
2. Solve for $r_{1,-1}$ using CRT.
3. Solve for $r_{-1,1}$ using CRT.
4. Solve for $r_{-1,-1}$ using CRT.
5. Return $r_{1,1}, \ldots, r_{-1,-1}$.

The only computational steps are the first 4, which each require one application of CRT, which itself requires applications of the Extended Euclidean Algorithm ('EEA'). Although the runtime of CRT depends on the amount of congruences, we will always only have a system of 2 equations for each $r$, and only 4 different $r$ values. So, we are bounded by the EEA, thus our algorithm runs with a bitwise time complexity of $O(\max\{p, q\})$ (as the largest prime will dominate the other).

**4.**

**Problem**

Describe a polynomial-time algorithm that given a natural number N of the form N = pq (with p and q are primes) and four elements of $a_1, a_2, a_2, a_4 \in \mathbb{Z}/N$ such that $a_i^2 \equiv 1$ mod N, finds $p$ and $q$.

**Solution**

We know $N = pq$ for some distinct primes $p$ and $q$, and so by part 1 question 2, we know there exists $a_p \in \{a_1, \ldots, a_4\}$ so that $a_p^2 \equiv 1 \pmod{p}$. Without loss of generality, suppose $a_p \equiv -1 \pmod{p}$. Thus, we know $a_p+1 \equiv 0 \pmod{p}$. Similarly, we can find $a_q \in \{a_1, \ldots, a_4\}$ so that $a_q + 1 \equiv 0 \pmod{q}$. Then,

$$(a_q + 1)(a_p + 1) = k(pq) = kN$$

for some $k \in \mathbb{Z}$. Now, note that $(N-1)^2 = N(N-1) + 1$, with $a_N \in \{a_1, \ldots, a_4\}$ so $a_N \equiv N - 1 \pmod{N}$. Thus, we know $a_N + 1 \equiv 0 \pmod{pq}$. In total, we know that there exist elements such that

$$a_N + 1 \equiv (a_p + 1)(a_q + 1) \pmod{pq}.$$

Now, note that at the beginning if we take our entire set of $a_i$ and reduce them modulo $N$, then in fact we simply have $a_p + 1 = p$, $a_q + 1 = q$, and $a_N + 1 = N$. By the uniqueness of our $a_i$, we know these $a_p$ and $a_q$ are unique.

However, we do not know what $p$ and $q$ are, so we cannot simply test these $a_i$ to see if they satisfy the congruence modulo $p$ and $q$. However, we can adapt this approach as we know they exist and they are unique, and use some specific details about our input. We simply take each $a_i$ and reduce it modulo $N$. Then, take the first $a_i$ that is neither 1 nor $N - 1$. As there are only two possible $a_i$ satisfying this, we will have $a_i + 1$ be a multiple of $p$ or $q$, and hence $\gcd(a_i + 1, N)$ will give one of our factors. Thus, our algorithm is

1. Use division with remainder on all our list of $a_i$ to reduce them modulo $N$.
2. Take the first $a_i$ that is neither 1 nor $N - 1$.
3. Compute $\gcd(a_i + 1, N)$, and $\gcd(a_i - 1, N)$, and return these two numbers.

The size of our lists are always fixed (as we will only ever have lists of length 4). Thus, the significant computational aspect above is computing the gcd which can both done using EEA. As EEA runs bitwise in $O(M)$, where $M$ corresponds to the larger of the two inputs, and our largest input will always be $N$, this algorithm will run in $O(N)$.

# Part 2

## 1. First Program

### 1.1 Python Function for $solve1(p, q)$

```python
'''
@authors: Alex Kazachek, Bradley Assaly-Nesrallah, Bohan Jiang, Wenxuan Dai
@Date: Oct 09, 2020

Math3159A - Group Assignment 2
Problem 2

This program implements the solve1(p,q) and solve2(N,a1,a2,a3,a4) functions.

'''

def extended_euclidean(a,b):
    """
    Implementation of the Extended Euclidean Algorithm in order to compute the gcd(a,b),
    as well as the two integers u and v such that a*u + b*v = gcd(a,b).

    This function is implemented iteratively and runs in O(log N) time, as seen in the textbook.

    Parameters
    ----------
    a : int
        First integer of the gcd(a,b) function.
    b : int
        Second integer of the gcd(a,b) function.

    Returns
    -------
    3-tuple containing the following values:

    gcd : int
        value of gcd(a,b)
    u : int
        integer value such that a*u + b*v = gcd(a,b).
    v : int
        integer value such that a*u + b*v = gcd(a,b).
    """

    u =1
    g =a
    x =0
    y =b
    while True:
        if y ==0:
            v =(g-a*u)//b
            return ((int(g),int(u),int(v)))
        t =g % y
        q =g//y
        s =u-q*x
        u =x
        g =y
        x =s
        y =t

def chinese_remainder_theorem(modulus_list, num_list):
    """
    Implementation of the Chinese Remainder Theorem in order to find the minimum value x
    such that x satisfies the given set of congruences:

    x is congruent to a_1 (mod m_1),
    x is congruent to a_2 (mod m_2),
    ...,
    x is congruent to a_i (mod m_i)

    Let N = len(modulus_list) and y = product of all values in modulus_list
    Then, this function runs in O(N * log(y)) time, where the log(y) time represents
    the running time of the Extended Euclidean Algorithm.

    Parameters
    ----------
    modulus_list : int list
        List of integers containing the list of modulus [m_1, m_2, ...., m_n]
    num_list : int list
        List of integers containing the list of integers [a_1, a_2, ..., a_n]


    Returns
    -------
    x : int
        Smallest integer x satisfying the set of congruences given.
    """

    # Find the product of all numbers in modulus_list
    mod_prod =1
    for modulus in modulus_list:
        mod_prod *=modulus

    # Instantiate result variable
    res =0
    # Iterate through modulus list
    for i in range(len(modulus_list)):
        # Find y, the product of all moduli divided by the current modulus
        y =mod_prod //modulus_list[i]
        # Add to res the number at num_list[i] * the modular multiplicative inverse with respect to
                                                            y
        # and the current modulus * y
        res =res +num_list[i] *extended_euclidean(y, modulus_list[i])[1] *y

    # Return res modulo the product of all moduli, which is our desired result x
    return res % mod_prod


def solve1(p,q):
    """
    Implementation of the solve1(p,q) function such that given two distinct primes p,q,
    returns all solutions to the congruence: x^2 is congruent to mod pq.

    This implementation is done using the mathematical algorithm described in Part 1 question 3.
    Hence, as proved in Part 1, this algorithm should return a tuple of 4 unique integers
                                                         corresponding
    to the solutions of the above congruence.
    Parameters
    ----------
    p : int
        A prime integer p.
    q : int
        A prime integer q.

    Returns
    -------

    res : tuple of integers
        A tuple of integers containing all solutions to the congruence.
    """

    # Add trivial solutions to res
    # These are the solutions to the following congruence systems:
    # 1 is the solution to: x is congruent to 1 (mod p) and x is congruent to 1 (mod q)
    # p*q - 1 is the solution to: x is congruent to -1 (mod p) and x is congruent to -1 (mod q)
    res =[1,p*q-1]

    # Find non-trivial solutions using CRT and add to res
    # The third solution can be found by solving the following congruence equation system:
    # x is the solution to: x is congruent to 1 (mod p) and x is congruent to -1 (mod q)
    modulus_list =[p,q]
    num_list =[1,-1]
    res.append(chinese_remainder_theorem(modulus_list, num_list))

    # Finally, the last solution can be found by solving the following congruence equation system:
    # x is the solution to: x is congruent to -1 (mod p) and x is congruent to 1 (mod q)
    num_list =[-1,1]
    res.append(chinese_remainder_theorem(modulus_list, num_list))

    # Convert to tuple and return res
    return tuple(res)
```

## 1.2 Program Output

The program *generate_input.py* was ran with the last three digits of the programmer's student number (296) as follows:

$$python\ generate\_input1.py\ 296$$

The following output was generated from the program:

$$(32416188241, 4294967513)\ (32416187761, 4294967867)\ (32416189717, 4294968187)$$

The three tuple sets generated above was then ran using the $solve1(p, q)$ function described above using the following code:

```
# Runs the code below when this file is executed
if __name__ =='__main__':
    # Execute the solve1(p,q) function using the 3 outputs generated from:
    # ./generate_input1.py 296
    print(solve1(32416188241, 4294967513))
    print(solve1(32416187761, 4294967867))
    print(solve1(32416189717, 4294968187))
```

The program *solve.py* was then ran as follows:

*python solve.py*

The following output was then generated when *solve.py* ran:

```
(1, 139226475390387614632, 25514278081758211343, 113712197308629403290)
(1, 139226484804133675786, 114364655004883899431, 24861829799249776356)
(1, 139226503578271533078, 93528137902216689609, 45698365676054843470)
```

## 2.Second Program

### 1.1 Python Function for $solve2(N, a1, a2, a3, a4)$

```
'''
@authors: Alex Kazachek, Bradley Assaly-Nesrallah, Bohan Jiang, Wenxuan Dai
@Date: Oct 09, 2020

Math3159A - Group Assignment 2
Problem 2

This program implements the solve1(p,q) and solve2(N,a1,a2,a3,a4) functions.

'''


def extended_euclidean(a,b):
    """
    Implementation of the Extended Euclidean Algorithm in order to compute the gcd(a,b),
    as well as the two integers u and v such that a*u + b*v = gcd(a,b).

    This function is implemented iteratively and runs in O(log N) time, as seen in the textbook.

    Parameters
    ----------
    a : int
        First integer of the gcd(a,b) function.
    b : int
        Second integer of the gcd(a,b) function.

    Returns
    -------
    3-tuple containing the following values:

    gcd : int
        value of gcd(a,b)
    u : int
        integer value such that a*u + b*v = gcd(a,b).
    v : int
        integer value such that a*u + b*v = gcd(a,b).
    """

    u =1
    g =a
    x =0
    y =b
    while True:
        if y ==0:
            v =(g-a*u)//b
            return ((int(g),int(u),int(v)))
        t =g % y
        q =g//y
        s =u-q*x
        u =x
        g =y
        x =s
        y =t


def solve2(N,a1,a2,a3,a4):
    """
    Implementation of the solve2(N,a1,a2,a3,a4) function such that given a natural number of the
                                                         form
    N = pq (for p,q primes) and four roots of unity a1, a2, a3, a4 is in Z/N, returns the primes p,
                                                         q

    This implementation is done using the mathematical algorithm described in Part 1 question 4.
    This algorithm finds the two primes p and q such that N = pq.

    Parameters
    ----------
    N : int
        A natural number N such that N = pq.
    a1 : int
        A root solution to such that (a1)**2 is congruent to 1 mod N.
    a2 : int
        A root solution to such that (a2)**2 is congruent to 1 mod N.
    a3 : int
        A root solution to such that (a3)**2 is congruent to 1 mod N.
    a4 : int
        A root solution to such that (a4)**2 is congruent to 1 mod N.

    Returns
    -------

    res : tuple of integers
        A tuple of integers containing the primes (p,q).
    """

    # Put all roots in a list and create a list to store all possible solutions
    roots =[a1, a2, a3, a4]
    soln =[]
    for root in roots:
        # Since we know that one root is 1 and the other is N-1, we look for a root
        # that is not equal to 1 or N-1
        if root !=1 and root !=N-1:
            # We can then find the two primes by using EEA and taking:
            # gcd(N, root+1) and gcd(N, root-1).
            # The results of the above corresponds to the unique primes p and q.
            soln.append(extended_euclidean(root+1, N)[0])
            soln.append(extended_euclidean(root-1, N)[0])
            break

    return tuple(soln)
```

### 1.2 Program Output

The program *generate_input.py* was ran with the last three digits of the programmer's student number (296) as follows:

$$python\ generate\_input2.py\ 296$$

The following output was generated from the program:

$$(147573957924025841387, (1, 147573957924025841386, 94166239817860682228, 53407718106165159159))$$
$$(147573975224154881923, (1, 147573975224154881922, 127267796233366145756, 20306178990788736167))$$
$$(147573992764803003143, (1, 147573992764803003142, 93387515657155393 77, 138235241199087463766))$$

The three tuple sets generated above was then ran using the $solve2(N, a1, a2, a3, a4)$ function described above using the following code:

```
# Runs the code below when this file is executed
if __name__ =='__main__':

    # Execute the solve2(N,a1,a2,a3,a4) function using the 3 outputs generated from:
    # ./generate_input2.py 296
    print(solve2(147573957924025841387, 1, 147573957924025841386, 94166239817860682228,
                                                             53407718106165159159))
    print(solve2(147573975224154881923, 1, 147573975224154881922, 127267796233366145756,
                                                             20306178990788736167))
    print(solve2(147573992764803003143, 1, 147573992764803003142, 9338751565715539377,
                                                             138235241199087463766))
```

The program *solve.py* was then ran as follows:

$$python\ solve.py$$

The following output was then generated when *solve.py* ran:

```
(8589934721, 17179869547)
(8589935407, 17179870189)
(8589936101, 17179870843)
```