Lab 1

Diffie-Hellman Key Exchange

Lab Objective: *Understand a method of sharing secrets over insecure channels.*

In cryptography, a *key* is a piece of secret information that is required to either encrypt or decrypt a message. A cipher (or code) is *symmetric* if it uses the same key for both encryption and decryption. People have used symmetric ciphers since ancient history. For example, the ancient Greeks used substitution ciphers, the Nazis used Enigma, and people today use AES and Twofish.

For Alice and Bob to use a symmetric cipher, they must first agree on a key. A *key exchange algorithm* is a way to do this so that even if an enemy Eve intercepts all the information passed between Alice and Bob, she will not be able to deduce the key. The Diffie-Hellman key exchange was the first such algorithm, and it is frequently used today.

One-Time Pad (Optional)

In this section we introduce a basic symmetric cipher with modern uses: the one-time pad. This cipher is outlined below.

- The key is a number.
- Encrypt a message by
    1. converting it to a number
    2. then adding the key *bitwise*. This means that binary digits are added with no carrying; for example, $1010 + 11 = 1001$.
- Decrypt the ciphertext by
    1. adding the key to it bitwise
    2. then converting the resulting number back into text.

The security of a one-time pad is compromised when the key is reused to encrypt another message.

For example, suppose we wish to encrypt the message SECRET. One way to convert this message to a number is to use the substitution cipher $A = 10, B = 11, \ldots, Z = 35$. With this rule, SECRET=281412271429. To encrypt this message, we must choose a key that is larger than 281412271429. We choose 987654321000. We perform the bitwise addition in Python with the caret operator `^`.

```
>>> message = 281412271429
>>> key = 987654321000
>>>
>>> ciphertext = message^key    # Encode the message
>>> ciphertext
706282163757
>>>
>>> message == ciphertext^key    # Decode the message
True
```

A = 10,...

Problem 1. For this problem, use the substitution cipher $A = 10, \ldots$ described above, with the additional rule `' '`=36.

1. Encrypt the message 'PRIVATE INFO' with the key 98765432109876543210 9876.
2. Decrypt the message 153931672663401143 with the key 12345678901234567890.

Diffie-Hellman key exchange algorithm

The Diffie-Hellman key exchange uses *primitive roots* modulo a prime.

Primitive roots

Let p be a prime number. A positive number a is a *primitive root modulo* p if the positive powers $a^1, a^2, \ldots, a^{p-1}$ generate all nonzero congruence classes modulo p. For example, let $p = 7$. Then $a = 2$ is not a primitive root, since

$$2^1 = 2, \quad 2^2 = 4, \quad 2^3 \equiv 1, \quad 2^4 \equiv 2, \quad 2^5 \equiv 4, \quad 2^6 \equiv 1 \pmod 7.$$

The powers of 2 never hit the classes $3, 5$, or 6 modulo 7. However, 3 is a primitive root since

$$3^1 = 3, \quad 3^2 \equiv 2, \quad 3^3 \equiv 6, \quad 3^4 \equiv 4, \quad 3^5 \equiv 5, \quad 3^6 \equiv 1 \pmod 7.$$

That is, each congruence class $\pmod 7$ appears as a power of 3.

Diffie-Hellman algorithm

Here is an outline of how Alice and Bob can perform a Diffie-Hellman key exchange:

- Alice and Bob (publicly) agree on a prime p and a primitive root g.
- Alice and Bob each secretly choose an integer, x and y respectively.
- Alice computes $A = g^x \pmod{p}$ while Bob computes $B = g^y \pmod{p}$.
- Alice and Bob exchange A and B (still keeping x and y secret).
- Alice computes $B^x \pmod{p}$ and Bob computes $A^y \pmod{p}$. They get the same answer,
$$k \equiv B^x \equiv (g^y)^x \equiv (g^x)^y \equiv A^y \pmod{p}.$$

Now Alice and Bob can use k as a secure key for a symmetric cipher. One advantage of this algorithm is that Alice and Bob can use the same prime p and root g to generate a new secure key, simply by picking new integers x and y.

The key idea of the Diffie-Hellman algorithm is that $(g^x)^y \equiv (g^y)^x \pmod{p}$. We do an example of this in Python with $p = 41$, $g = 6$, $x = 10$, and $y = 13$. The function `pow(a, b, n)` computes $a^b \pmod{n}$.

```
>>> A = pow(6, 10, 41)
>>> B = pow(6, 13, 41)
>>> pow(A, 13, 41)
32
>>> pow(B, 10, 41)
32
```

Problem 2. With a partner, perform a Diffie-Hellman key exchange with $p = 21929$ and $g = 3$.

The Diffie-Hellman can also be used to create a secure key shared by $n > 2$ people. If n people choose private exponents $x_1, \ldots, x_n$, then the secure key will be $g^{x_1 \cdots x_n}$.

Problem 3. Work out a method by which n people can use Diffie-Hellman to create a secure key. Remember that only the i^{th} person knows the i^{th} exponent x_i. With a group of at least 3 people, perform such an exchange with $p = 21929$ and $g = 3$.

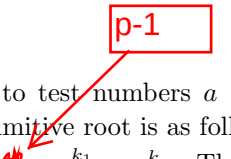
Security of Diffie-Hellman

Why is the Diffie-Hellman algorithm secure? Suppose a third party, Eve, intercepts p, g, A, and B. If she can compute x and y, she easily compute k just as Alice and Bob did. Then Eve must compute x satisfying $A \equiv g^x \pmod{p}$. This computation is known as taking a *discrete logarithm* of A and there are no fast algorithms for doing it. When p is large, say 100s of digits, calculating a discrete logarithm is effectively impossible.

Practical Considerations

For Diffie-Hellman to be a practical key exchange algorithm, we need an efficient way to find primitive roots and perform modular exponentiation.

Finding primitive roots



The only way to find the primitive roots of p is to test numbers $a < p$ until a primitive root is found. One test to see if a is a primitive root is as follows.

First, factor $p-1$ as a product of primes, say $p-1 = p_1^{k_1} \dots p_n^{k_n}$. Then, for each prime p_i in the factorization, compute $a^{(p-1)/p_i} \pmod{p}$. If for any i we find $a^{(p-1)/p_i} \equiv 1 \pmod{p}$, then a is NOT a primitive root. Otherwise, a is a primitive root.

As an example we look for a primitive root of $p = 41$. We factor $p - 1 = 40$ using the function `factorint()` from SymPy.

```
>>> from sympy import factorint
>>> factorint(40)
{2: 3, 5: 1}
```

This output means that $40 = 2^3 \cdot 5^1$. Then the powers to check are $40/2 = 20$ and $40/5 = 8$. The following code shows that 2 is not a primitive root modulo 41, since $2^{20} \equiv 1 \pmod{41}$.

```
>>> pow(2, 20, 41)
1
```

However, 6 is a primitive root modulo 41, as the following code shows.

```
>>> pow(6, 8, 41)
10
>>> pow(6, 20, 41)
40
```

The idea behind this algorithm is that if $a^n \equiv 1 \pmod{p}$ for some $n < p-1$, then a will not be a primitive root. Instead, powers of a will repeat and miss some congruence classes, as they did in the example of $2 \pmod{7}$. For "group-theoretic" reasons, if $a^n \equiv 1 \pmod{p}$ for some $n < p-1$ (i.e., a is not a primitive root), then there must be some p_i such that $a^{(p-1)/p_i} \equiv 1 \pmod{p}$. So, instead of checking all powers of a modulo p, we only need to check the powers $(p-1)/p_i$ for all i.

Problem 4. Write the following function to test if a is a primitive root modulo p.

```
def is_primitive(root, mod):
    '''Determine whether `root' is a primitive root modulo `mod'.

    INPUTS:
    root - A positive integer.
    mod  - A prime integer.
```

```
    Return a Boolean value.
    '''
```

Fast modular exponentiation

For RSA to be a reasonably fast cryptosystem, we need a fast algorithm for modular exponentiation. As an example, suppose we wish to compute $5064^{361} \pmod{19673}$. The naïve way is to multilply 5064 by itself 361 times, producing an enormous number, and then mod this number by 19673. A slightly better way is to take the modulus after each multiplication by 5064, since this way we do not store such a large number.

Problem 5. Write the following function to implement the algorithm for modular exponentiation outlined above.

```
def power1(base, exp, mod):
    ''' Return base^exp modulo `mod'.

    Multiply `base' by itself `exp' times, taking the modulus after each
    multiplication.

    Example:
    >>> power1(5064, 361, 19673)
    13172
    '''
```

Time your function against Python's built in `pow()` function. What is the difference in speed?

If you did Problem 5, you know that Python's exponentiation method is orders of magnitude faster than the naïve method outlined above. This increase in performance is acheived with the *right-to-left binary method* of exponentiation.

This idea of this method is as follows. To compute $a^{361} \pmod{19673}$, we first write 361 in binary as 101101001. Thus,

$$a^{361} = a^{2^0+2^3+2^5+2^6+2^8} = a^{2^0} a^{2^3} a^{2^5} a^{2^6} a^{2^8} \pmod{19673}. \tag{1.1}$$

We can quickly compute the integers

$$a = a^{2^0},\ a^2 = a^{2^1},\ a^{2^2},\ \ldots,\ a^{2^8} \pmod{19673}$$

by squaring each term to get the next. Then, to compute (1.1), we simply multiply the appropriate terms from this series.

Problem 6. Implement the right-to-left binary method with the following function.

```
def power2(base, exp, mod):
    ''' Return base^exp modulo `mod'.

    Compute the result using the right-to-left binary method.

    Example:
    >>> power2(5064, 361, 19673)
    994
    '''
```

Do not at any point use Python's built-in exponentiation methods. Compare the speed of this function to that of the function in Problem 5 and Python's `pow()` function. Even with a naïve implementation of the right-to-left binary method, you should see significant improvement over the `power1()` function.