Author: Tianyi Lu (Sky)

Breaking Diffie-Hellman

By the description of Diffie-Hellman key exchange protocol, we know that shared key K is calculated from Alice and Bob's private keys a and b in the follow method:

$$A \equiv g^a \bmod p$$
$$B \equiv g^b \bmod p$$
$$K \equiv A^b \equiv B^a \bmod p$$

To figure out the value of K, we start by calculating the value of a or b. We can iterate value of b from one and using brutely force to check whether the current value produce our known A. The code is as follows:

```
for i in range(p):
        if g**i % p == B:
                print(f'Found b: {i}')
                b = i
                break
```

If we plug in $g = 7, p = 61, B = 17$ in our case, we will get $b = 23$. Now we know the $K \equiv A^b \equiv 30^{23} \equiv 6 \bmod p$. So the shared key is `6`.

Problems with Big Integers

When g and p are huge integers, say larger than 600 digits, finding what a makes $A \equiv g^a \bmod p$ when A, g, p are known becomes extremely computationally costly. The fastest computer on Earth right now could run the algorithm above to the heat death of the universe without finding a solution. This is also refer to as *the discrete logarithm problem*, which is known to be difficult to solve.

Breaking RSA

We know that n in the public key is made up of two prime factors p and q, which are part of the secret key. Since n is relatively small, we can try each prime number less then n (technically less than or equal to $\sqrt{n}$, but in our case there isn't much difference).

After trying all possible prime factors, we found out that $p = 379$ and $q = 449$. Now, we can get $\phi(n)$ by using $\phi(n) = (p-1)(q-1) = 169344$. Because we also know that $d \cdot e \equiv 1 \bmod \phi(n)$

with e and $\phi(n)$ shared, we can get d by enumerating the value from 1 until the equation is satisfied. Using the method, we found out that d is 119537.

All parts of secret key are now known. Each plaintext integer p_i can be calculated from ciphertext integer c_i using $(c_i)^d \equiv p_i \bmod n$. In our case, we will have plaintext integers as: `[18537, 8258, 28514, 11808, 18727, 27936, 30561, 27755, 26990, 26400, 26226, 28525, 8302, 28535, 8303, 28206, 8281, 28533, 29216, 28769, 27692, 8257, 27753, 25445, 11808, 26740, 29808, 29498, 12079, 26223, 30062, 25697, 29801, 28526, 11885, 28538, 26988, 27745, 11887, 29287, 12133, 28207, 28786, 26998, 24931, 31086, 28532, 26990, 25452, 30052, 25956, 12129, 29300, 26979, 27749, 29487, 26996, 29485, 28518, 26217, 25449, 24940, 11619, 24946, 29485, 24946, 25901, 29800, 25901, 30575, 29299, 29741, 28786, 28516, 30051, 29741, 25441, 29797, 26479, 29305, 11639, 25901, 26721, 30309, 11621, 30309, 29229, 29285, 30313, 25975, 25956, 11622, 28530, 11632, 29289, 30305, 25465, 12032]`.

Finding Out the Encoding

Examining the plaintext integers closely, we can learn that all of them are smaller than $2^{16} = 65536$. This indicates that each integers can be the concatenation of two bytes. We can convert these base 10 integers to base 2 bytes using the following function:

```
def decimal_to_bytestring(plain_text_integers):
    bytestring = []
    for i in plain_text_integers:
            # get rid of the leading '0b' produced by bin function
        binary = bin(i)[2:]
        # there can be leading zero missing in the first byte
        bytestring.append(binary[:-8])
        # but the last 8 bits must be complete
        bytestring.append(binary[-8:])

    return bytestring
```

Now, we can translate all binary bytes in to ASCII characters and combine them together to form our readable plaintext message:

```
def bytestring_to_ascii(bytestring):
    return ''.join([chr(int(b, 2)) for b in plain_text_bytestring])
```

The human readable plaintext is: `Hi Bob. I'm walking from now on. Your pal, Alice. https://foundation.mozilla.org/en/privacynotincluded/articles/its-official-cars-are-the-`

`worst-product-category-we-have-ever-reviewed-for-privacy/` .
Luckily, I don't own a car at Carleton. Hope there're fewer privacy concerns with bikes.

Problem with the Encoding

In Alice's encoding method, every two bytes' encryption are independent to each other. This means there can be patterns in the ciphertext if plaintext contains many repetition. Given enough ciphertext and a frequency analysis of two-byte strings in the context of Alice and Bob's conversation, we can use frequency analysis attack to guess the relationship between plaintext and ciphertext byte pairs.

Problems with Big Integers

It's very hard to calculate the *Euler Totient Function* $\phi(n)$ with large n without knowing p and q. Furthermore, factoring very large integers is also known to be very time consuming. It's impossible to find out $p, q, \phi(n)$ when n has more than 2048 digits with all modern computers. Maybe someday in the quantum realm, all can be revealed.

Appendix

Code for Breaking Diffie-Hellman

```
Public = {
    'p': 61,
    'g': 7,
    'A': 30,
    'B': 17
}

def break_brute_force(p, g, A, B):
    a = b = None
    for i in range(p):
        if g**i % p == A:
            print(f'Found a: {i}')
            a = i
            break

    for i in range(p):
        if g**i % p == B:
            print(f'Found b: {i}')
            b = i
            break

    return a, b
```

```
if __name__ == '__main__':
    a, b = break_brute_force(**Public)
    K = Public['g']**(a*b) % Public['p']
    if K != Public['A']**b % Public['p'] or K != Public['B']**a % Public['p']:
        raise Exception('K is not correct')

    print(K)
```

Code for Breaking RSA

```
import math

Public = {
    'e_Bob': 17,
    'n_Bob': 170171,
}

cipher_text = [65426, 79042, 53889, 42039, 49636, 66493, 41225, 58964,
126715, 67136, 146654, 30668, 159166, 75253, 123703, 138090,
118085, 120912, 117757, 145306, 10450, 135932, 152073, 141695,
42039, 137851, 44057, 16497, 100682, 12397, 92727, 127363,
146760, 5303, 98195, 26070, 110936, 115638, 105827, 152109,
79912, 74036, 26139, 64501, 71977, 128923, 106333, 126715,
111017, 165562, 157545, 149327, 60143, 117253, 21997, 135322,
19408, 36348, 103851, 139973, 35671, 93761, 11423, 41336,
36348, 41336, 156366, 140818, 156366, 93166, 128570, 19681,
26139, 39292, 114290, 19681, 149668, 70117, 163780, 73933,
154421, 156366, 126548, 87726, 41418, 87726, 3486, 151413,
26421, 99611, 157545, 101582, 100345, 60758, 92790, 13012,
100704, 107995]

def get_prime(max_limit):
        # Using Sieve of Eratosthenes to generate prime numbers
    prime = [True for _ in range(max_limit + 1)]

    p = 2
    while p * p <= max_limit ** 2:
        if prime[p]:
            for i in range(p * p, max_limit + 1, p):
                prime[i] = False

            yield p
        p += 1
```

```
def factorize(n):
    factors = []
    for prime in get_prime(math.ceil(n)):
        while n % prime == 0:
            factors.append(prime)
            n //= prime

    return factors

def get_d(e, phi):
    d = 0
    while (d * e) % phi != 1:
        d += 1

    return d

def decrypt(cipher_text, d, n):
    return [(c ** d) % n for c in cipher_text]

def encrypt(plain_text, e, n):
    return [(p ** e) % n for p in plain_text]

def decimal_to_bytestring(plain_text_integers):
    bytestring = []
    for i in plain_text_integers:
        binary = bin(i)[2:]
        bytestring.append(binary[:-8])
        bytestring.append(binary[-8:])

    return bytestring

def bytestring_to_ascii(bytestring):
    return ''.join([chr(int(b, 2)) for b in plain_text_bytestring])

if __name__ == '__main__':
    factors = factorize(Public['n_Bob']) # impractical for large n
    if len(factors) != 2:
        raise Exception('n_Bob is not the product of two primes')

        p, q = factors
        phi = (p - 1) * (q - 1)

        d = get_d(Public['e_Bob'], phi) # impractical if don't know p and q
        print(f'd: {d}')
        plain_text_integers = decrypt(cipher_text, d, Public['n_Bob'])
```

```
    print(plain_text_integers)
    # Encrypt again to check the correctness of d.
    print(encrypt(plain_text_integers, Public['e_Bob'], Public['n_Bob']))

    plain_text_bytestring = decimal_to_bytestring(plain_text_integers)

    plain_text = bytestring_to_ascii(plain_text_bytestring)
    print(plain_text)
```