Portfolio 3: Public key cryptography

Exercises will be implemented using the Python language. A compressed file with the Python source code, organized in one or several files, will be uploaded to the corresponding task at Moodle. There are no requirements on file structure, just make sure to be clear in which part of the code solves which exercise.

1 RSA cipher

a) Generate a pair of RSA keys and post them at "RSA public key directory".

b) Send an encrypted message to the person above you in the "RSA public key directory ". Post the encrypted message in the "RSA message channel". The message should be encoded in UTF-8. Use the same strategy we used in class to translate bytes into numeric blocks. The encrypted message posted should be the bytes representation of the encrypted message in Python for ease of copy/paste. For example:

```
b's\x00\x92\x01T\x02\x8f\x01\x0b\x00\xf7\x01\xd5\x01'
```

c) Decrypt the message sent to you.

2 Diffie-Hellman

a) Implement a function to generate a random prime p of n bits and a random appropriate generator g for $G = \mathbb{Z}/p\mathbb{Z}^*$.

To check that an element is a generator, you need to check that its order is equal to the group's order. We know the group's order is $\phi(p)$, so we have to check that $\phi(p)$ is the smallest k for which $g^k \equiv 1 \bmod p$. We would need to check all divisors of $\phi(p)$, which requires a factorization.

To save effort, fix p to be of the form $p = 2q + 1$, where q is prime (q would be a Sophie Germain prime, which has other interesting properties [1])

[1] https://sitn.hms.harvard.edu/flash/2017/2-x-prime-1-200-year-old-story-sophie-germain-21st-centu

b) It is not uncommon to use a precomputed p and g that have been shown to be safe. Implement a function that returns a pair of p and g obtained from RFC 3526 `https://www.ietf.org/rfc/rfc3526.txt`. Have the function take a single argument: a number of bits. If the number of bits does not correspond to the p and g you have selected, raise an error.

Note: In the formulas shown in RFC 3526 "[2 ^ 1406 pi]" means "math.floor(2 ** 1406 * pi)". The problem is that you need to approximate pi to the required number of decimals. For that, you can use the function *approximate_pi* in the attached file **pi.py**.

```python
from decimal import Decimal, getcontext

def approximate_pi(precision: int) -> Decimal:
    """Compute Pi to precision number of decimals.

    Taken from https://docs.python.org/3/library/decimal.html#recipes

    Changes the precision of Decimal's context to accomodate precision,
    so beware

    Arguments
    ---------
    precision: int
        Number of significant digits

    Returns
    -------
    Decimal
        The approximation of the value of pi
    """
    getcontext().prec = precision + 1
    getcontext().prec += 2  # extra digits for intermediate steps
    three = Decimal(3)      # substitute "three=3.0" for regular floats
    lasts, t, s, n, na, d, da = 0, three, 3, 1, 0, 0, 24
    while s != lasts:
        lasts = s
        n, na = n+na, na+8
        d, da = d+da, da+32
        t = (t * n) / d
        s += t
    getcontext().prec -= 2
    return +s               # unary plus applies the new precision
```

c) Given $p = 7883$, $g = 2$ and a user with $g^{a_i} \equiv 1876 \bmod p$, form a common key with that user.

3 El Gamal

a) Generate some valid p, g (you can use the solution to exercise 2a) for this) and public key $\beta = g^{a_i}$ and post it in the "El Gamal key directory".

b) Send an encrypted message to the person above you in "El Gamal key directory". The message should be encoded in UTF-8. Use the same strategy we use used to translate bytes into numeric blocks in RSA. Post the encrypted message in "El Gamal message channel".

Notes:

- To simplify, we will consider an El Gamal encrypted message a list of tuples, one for each block the message is divided into:

$$encrypted = [(C_{11}, C_{21}), (C_{12}, C_{22}), \ldots, (C_{1n}, C_{2n})]$$

  Where $C_{1i} = g^{k_i} \bmod p$ and $C_{2i} = M\beta^{k_i} \bmod p$. C_{1i} will be an integer value and C_{2i} the encrypted bytes of the corresponding message block. For example:

```
[(25048, b'\x9d\xa7'), (53606, b'\xef!')]
```

- Remember that $gcd(k_i, \phi(p)) = 1$

c) Decrypt the message sent to you.

4 RSA signature

a) A hash of the message is usually signed instead of the whole message. The NIST describes a set of approved hash functions in FIPS 180-4 `https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf`. Select one of the described hash functions to be used as part of the signature process and implement the signature scheme.

Notes:

- Select a hash function that is already implemented in Python's standard library `https://docs.python.org/3/library/hashlib.html`.

- To sign the message, use the bytes representation of the hash the same way we did with plaintext messages. The hashes in Python have a method *digest()* to extract the bytes of the hash.

b) Send an encrypted and signed message to the person above you in "RSA public key directory" and post it in "RSA signed channel". Both the encrypted message and signature should be posted in their bytes representation, just as in Exercise 1.

c) Verify the signature and decrypt the message sent to you.