Lecture 20: Cryptography Wrap-Up

Print this handout

Exercise 1: Implementing the RSA cryptosystem

Last class, we studied the RSA cryptosystem. Now let's see how to implement it in Python!

We've divided up this exercise into two parts: implementing the encryption/decryption functions (which are pretty straightforward), and then going back a step implementing the key generation phase, which is more complex.

```python
import random
import math


################################################################################
# Question 1: Encryption and Decryption
################################################################################
def rsa_encrypt(public_key: tuple[int, int], plaintext: int) -> int:
    """Encrypt the given plaintext using the recipient's public key.

    Preconditions:
        - public_key is a valid RSA public key (n, e)
        - 0 < plaintext < public_key[0]
    """








def rsa_decrypt(private_key: tuple[int, int, int], ciphertext: int) -> int:
    """Decrypt the given ciphertext using the recipient's private key.

    Preconditions:
        - private_key is a valid RSA private key (p, q, d)
        - 0 < ciphertext < private_key[0] * private_key[1]
    """








################################################################################
# Question 2: Key Generation
################################################################################
def rsa_generate_key(p: int, q: int) -> \
        tuple[tuple[int, int, int], tuple[int, int]]:
    """Return an RSA key pair generated using primes p and q.

    The return value is a tuple containing two tuples:
      1. The first tuple is the private key, containing (p, q, d).
      2. The second tuple is the public key, containing (n, e).

    Preconditions:
        - p and q are prime
        - p != q

    Hints:
        - If you choose a random number e between 2 and phi(n), there isn't a guarantee
        gcd(e, phi(n)) = 1. You can use the following pattern to keep picking random num
        until you get one that is coprime to phi(n).

            e = ... # Pick an initial choice
            while math.gcd(e, ___) > 1:
                e = ... # Pick another random choice

        - We've provided copies of the modular_inverse and extended_euclidean_gcd functi
    """








################################################################################
# Helper functions
################################################################################
def extended_euclidean_gcd(a: int, b: int) -> tuple[int, int, int]:
    """Return the gcd of a and b, and integers p and q such that

    gcd(a, b) == p * a + b * q.

    Preconditions:
        - a > 0  # Simplifying assumption for now
        - b > 0  # Simplifying assumption for now
    """
    x, y = a, b

    px, qx = 1, 0
    py, qy = 0, 1

    while y != 0:
        # assert math.gcd(x, y) == math.gcd(a, b)  # Loop Invariant 1
        assert x == px * a + qx * b                # Loop Invariant 2
        assert y == py * a + qy * b                # Loop Invariant 3

        q, r = divmod(x, y)

        x, y = y, r
        px, qx, py, qy = py, qy, px - q * py, qx - q * qy

    return (x, px, qx)


def modular_inverse(a: int, n: int) -> int:
    """Return the inverse of a modulo n, in the range 0 to n - 1 inclusive.

    Preconditions:
        - gcd(a, n) == 1
        - n > 0

    >>> modular_inverse(10, 3)  # 10 * 1 is equivalent to 1 modulo 3
    1
    >>> modular_inverse(3, 10)  # 3 * 7 is equivalent to 1 modulo 10
    7
    """
    result = extended_euclidean_gcd(a, n)
    p = result[1]

    return p % n
```