ICC/Programming — Week 13 Exercises

18 May 2026

Goal. In these exercises you will practise the fundamentals of cryptography introduced in today's lecture: symmetric-key encryption (XOR cipher), public-key encryption (RSA), and a comparison of the two approaches.

These exercises are self-contained. No additional libraries are required beyond the Python standard library.

Setup. Create a new file **`crypto.py`** and work through each exercise by adding code to the bottom of the file. Run it at any point with:

```
python3 crypto.py
```

Exercise 1. Symmetric Encryption — XOR Cipher

In symmetric cryptography, Alice and Bob share a single secret key used both to encrypt and to decrypt. The simplest example is XOR encryption: each character of the message is combined with a key using the exclusive-or (XOR, $\oplus$) operation. Recall that XOR is its own inverse: $(m \oplus k) \oplus k = m$.

(a) **Basic XOR encrypt/decrypt.** Implement **`xor_cipher`**, which encrypts (or decrypts) a string by XOR-ing every character with **`key`**. Use Python's **`ord`**, **`chr`**, and the **`^`** operator. Then verify that the test below prints **`True`**:

```python
def xor_cipher(text: str, key: int) -> str:
    ...   # one line is enough

message    = "HELLO"
key        = 42
ciphertext = xor_cipher(message, key)
decrypted  = xor_cipher(ciphertext, key)   # same function, same key
print("Round-trip OK:", decrypted == message)   # must be True
```

Why does calling **`xor_cipher`** twice with the same key give back the original message? Answer using the XOR truth table from the lecture.

(b) **Encrypting a longer message.** A single byte key offers only 256 possible values. Extend the cipher so it cycles through a list of key bytes: position i of the plaintext should be XOR-ed with **`key[i % len(key)]`**. Complete the function and verify the round-trip:

```python
def xor_cipher_multi(text: str, key: list[int]) -> str:
    result = []
    for i, c in enumerate(text):
        result.append(...)   # XOR ord(c) with the correct key byte
    return ''.join(result)

key_multi = [42, 17, 99]
msg = "ATTACK AT DAWN"
enc = xor_cipher_multi(msg, key_multi)
dec = xor_cipher_multi(enc, key_multi)
print("Round-trip OK:", dec == msg)   # must be True
```

(c) **Key length and security.**

(i) How many possible keys are there for a 2-byte key? A 4-byte key? An n-byte key?

(ii) Modern symmetric ciphers such as AES use a 128-bit key. How many possible keys is that? (Express your answer as a power of 2 and as an approximate power of 10.)

Exercise 2. Public-Key Encryption — Toy RSA

Background. In asymmetric (public-key) cryptography each party has two mathematically linked keys: a public key that anyone may use to encrypt a message, and a private key that only the owner holds, used to decrypt.

RSA is the most widely used public-key algorithm. Its security rests on the fact that multiplying two large prime numbers together is easy, but reversing the process—factoring the product back into its two primes—is computationally infeasible for large enough numbers.

RSA key generation (simplified).

1. Choose two prime numbers p and q (in practice, hundreds of digits long). Compute $n = p \cdot q$.
2. Compute Euler's totient: $\phi(n) = (p-1)(q-1)$. This counts how many integers from 1 to $n-1$ share no common factor with n.
3. Choose a public exponent e such that $1 < e < \phi(n)$ and $\gcd(e, \phi(n)) = 1$ (i.e. e and $\phi(n)$ share no common factors).
4. Compute the private exponent d such that $e \cdot d \equiv 1 \pmod{\phi(n)}$, i.e. d is the modular inverse of e.

The **public key** is the pair (e, n); the **private key** is d (together with n). Encryption: $c = m^e \bmod n$. Decryption: $m = c^d \bmod n$.

Toy example (used throughout this exercise): $p = 3,\ q = 11,\ n = 33,\ \phi(n) = 20,\ e = 3,\ d = 7$, because $3 \times 7 = 21 \equiv 1 \pmod{20}$.

(a) **Verify the toy parameters by hand.** Before writing any code, confirm the following on paper (or in your head):

(i) Check that $p = 3$ and $q = 11$ are both prime.

(ii) Check that $n = p \cdot q = 33$ and $\phi(n) = (p-1)(q-1) = 20$.

(iii) Check that $e \cdot d = 3 \times 7 = 21 \equiv 1 \pmod{20}$, so d is indeed the modular inverse of e.

(iv) Encrypt the message $m = 4$ by hand: compute $4^3 \bmod 33$. Then decrypt your result: raise it to the power $d = 7$ modulo 33. Do you recover 4?

(b) **Modular exponentiation.** The naive approach—multiplying in a loop—is far too slow for large exponents. Implement **`mod_exp_naive`** by completing the loop below, then verify that it agrees with Python's built-in **`pow(base, exp, mod)`**, which uses fast (square-and-multiply) exponentiation internally, requiring only $O(\log exp)$ multiplications:

```python
def mod_exp_naive(base: int, exp: int, mod: int) -> int:
    result = 1
    for _ in range(exp):
        ...   # update result on each iteration
    return result

e, d, n = 3, 7, 33
for m in range(1, 10):   # messages 1..9, all < n = 33
    assert mod_exp_naive(m, e, n) == pow(m, e, n), f"Failed for m={m}"
print("mod_exp_naive correct for m = 1..9")
```

Why does the naive implementation become impractical for real RSA, where exponents have thousands of digits?

(c) **Encrypt and decrypt a word.** To encrypt text with RSA, treat each character as its Unicode code point (obtained with **`ord`**) and encrypt each code point independently with **`pow`**. Implement both functions, then encrypt and decrypt the message **`"HI"`** using the toy parameters:

```python
def rsa_encrypt(plaintext: str, e: int, n: int) -> list[int]:
    ...   # return the list of encrypted code points

def rsa_decrypt(ciphertext: list[int], d: int, n: int) -> str:
    ...   # return the decrypted string

e, d, n = 3, 7, 33
enc = rsa_encrypt("HI", e, n)
dec = rsa_decrypt(enc, d, n)
print("Encrypted:", enc)
print("Decrypted:", dec)   # does NOT give "HI" -- why?
```

You will notice that **`"HI"`** does not round-trip correctly. Explain why by looking up the Unicode code points of **`'H'`** and **`'I'`** and comparing them to $n = 33$.

Why must $m < n$? Encryption computes $c = m^e \bmod n$, so c always falls in $\{0, 1, \ldots, n-1\}$ regardless of how large m is. Two different values of m that differ by a multiple of n produce exactly the same ciphertext, so decryption cannot tell them apart. In other words, RSA only "sees" m mod n; any information in m beyond that is permanently lost.

(d) **Slightly larger parameters.** The parameters below use larger (but still tiny) primes that make n large enough to encrypt any printable ASCII character (code points 32–126).

```
# Key generation (for reference -- you do not need to derive these):
#   p = 61,  q = 53          (both prime)
#   n = p * q       = 3233   (public, part of both keys)
#   phi(n) = (p-1)*(q-1)
#          = 60 * 52 = 3120 (kept secret)
#   e = 17                   (public exponent, gcd(17, 3120) = 1)
#   d = 2753                 (private exponent: 17 * 2753 mod 3120 = 1)
e2, d2, n2 = 17, 2753, 3233
```

(i) Using your **`rsa_encrypt`** and **`rsa_decrypt`** functions, encrypt then decrypt **`"Hello, Bob!"`** and confirm the round-trip with an **`assert`**.

(ii) How many distinct integer messages can these parameters handle? (i.e. how many values satisfy $0 \leq m < n$?)

(iii) Verify in Python that $e_2 \cdot d_2 \bmod \phi(n) = 1$, confirming that d_2 is the correct private exponent. *Hint:* $\phi(n) = (p-1)(q-1) = 3120$.

(e) **Why is RSA hard to break?** An attacker who knows the public key (e, n) must recover d. The only known way to do this efficiently is to factor n into p and q, then recompute $\phi(n)$ and solve for d. Implement the body of the trial-division factoriser below:

```
from time import perf_counter
import math

def factorise(n: int) -> tuple[int, int]:
    """Return (p, q) such that p * q == n, or (1, n) if n is prime."""
    for p in range(2, math.isqrt(n) + 1):
        ...   # test whether p divides n and return the pair if so
    return 1, n

start   = perf_counter()
p, q    = factorise(3233)
elapsed = perf_counter() - start
print(f"Factored 3233 = {p} * {q}  in {elapsed:.6f} s")
```

(i) How many divisions does trial division need in the worst case to factor n? (Express in terms of n.)

(ii) Real RSA uses n with 2048 bits ($n \approx 10^{617}$). Estimate how many divisions trial division would need, and explain why this is completely infeasible even on the fastest modern hardware.

(iii) Once you have factored $n = 3233$ into p and q, compute $\phi(n)$ and recover the private exponent d using **`pow(e2, -1, phi_n)`**. Confirm the result equals 2753 and that it correctly decrypts a message encrypted with **`e2`**.

Exercise 3. Comparing Symmetric and Asymmetric Encryption

(a) **Speed comparison.** Write the code that measures the time needed to encrypt a 1 000-character string with your XOR cipher (key = 42) and with your toy RSA (parameters **`e2, n2`**). Use **`time.perf_counter`** and print both durations and their ratio.

```
from time import perf_counter

long_msg = "A" * 1000

# Measure and print the time for xor_cipher
...

# Measure and print the time for rsa_encrypt
...

# Print how many times slower RSA is than XOR
...
```

In practice, systems such as HTTPS use asymmetric encryption only to exchange a short random session key, then switch to symmetric encryption for all subsequent data. Why does this make sense given your results?

(b) **The key-distribution problem.** Symmetric encryption requires Alice and Bob to agree on a shared secret key before they can communicate securely — but how do they exchange that key without anyone intercepting it?

(i) If n people all want to communicate privately with every other person using symmetric encryption, how many distinct shared keys are needed in total? Fill in the table and write a general formula:

People (n)	**Keys needed**
2	
5	
10	
100	

(ii) How does public-key encryption solve the key-distribution problem? How many keys does each person need to hold? (Answer in 2–3 sentences.)