8.1 An Introduction to Cryptography

So far we've seen how the data types we introduced in Chapter 1 can be used to store a variety of different data. In our modern world, data is constantly being created, stored, sent, and received. But not all data is created equal; some data is inherently more sensitive than other data. And there are laws mandating the privacy of your data in Canada. Thanks to the explosion of data and the evolution of privacy policy, there are numerous technologies (backed by a strong theoretical underpinning) being developed to ensure data privacy and security.

After our work from the previous chapter, we now have the theoretical foundations necessary to learn about one of the coolest applications of number theory in computer science: encrypting messages so that only the sender and receiver can understand them.[1] This is only one method for ensuring data privacy, but it is pervasive—nearly every time you send or receive something on your phone or web browser, cryptography plays a role. In this section, you'll learn about the basics of *cryptography*, which is the study of theoretical and practical techniques for keeping data secure.

[1] Check out the movie The Imitation Game, which is about some amazing codebreaking work done in World War II (and a crucial piece in the history of computing).

What is cryptography?

Cryptography is the study of techniques used to keep communication secure in the face of adversaries who wish to eavesdrop on or interfere with the communication. Defining what *secure* communication between two parties means is complex, and involves several dimensions such as: confidentiality, data integrity, and authentication. In this chapter we will focus primarily on encryption, which involves turning coherent messages into seemingly-random nonsensical strings, and then back again.

As computers have become more powerful, cryptographic technologies have evolved to ensure that the "nonsense" strings are not easily converted back to the coherent message except by the intended recipient(s). But the growing power of computers is a double-edged sword; while cryptographic technologies have evolved, so have the technologies of malicious attackers and eavesdroppers who want to decipher the "nonsense" strings and gain access to sensitive data, such as passwords and social insurance numbers.

Setting the stage: Alice and Bob

The simplest setup that we study in cryptography is *two-party confidential communication*. In this setup, we have two people, who we'll call Alice and Bob, who wish to send messages to each other that only they can read, and a third person, Eve, who has access to all of the communications between Alice and Bob, and wants to discover what they're saying.

Since Eve has access to the communications between Alice and Bob, they can't just send their messages directly. So instead, Alice and Bob need to encrypt their messages using some sort of encryption algorithm, and send the encrypted versions to each other instead. The hope is that through some shared piece of information called a secret key, Alice and Bob can encrypt their messages in such a way that they will each be able to decrypt each other's messages, but Eve won't be able to decrypt the messages without knowing their secret key.

More formally, we define a **secure symmetric-key cryptosystem** as a system with the following parts:

- A set $\mathcal{P}$ of possible original messages, called **plaintext** messages. (E.g., a set of strings)
- A set $\mathcal{C}$ of possible encrypted messages, called **ciphertext** messages. (E.g., another set of strings)
- A set $\mathcal{K}$ of possible **shared secret keys** (known by both Alice and Bob, but no one else).
- Two functions $Esecret : \mathcal{K} \times \mathcal{P} \to \mathcal{C}$ and $Decrypt : \mathcal{K} \times \mathcal{C} \to \mathcal{P}$ that satisfies the following two properties:
  - (**correctness**) For all $k \in \mathcal{K}$ and $m \in \mathcal{P}$, $Decrypt(k, Esecret(k, m)) = m$. (That is, if you encrypt and then decrypt the same message with the same key, you get back the original message.)
  - (**security**) For all $k \in \mathcal{K}$ and $m \in \mathcal{P}$, if an eavesdropper only knows the value of $c = Esecret(k, m)$ but does not know k, it is computationally infeasible to find m.

Example: Caesar's substitution cipher

One of the earliest examples we have of a symmetric-key cryptosystem is the *Caesar cipher*, named after the Roman general Julius Caesar. In this system, the plaintext and ciphertext sets are simply strings, and the secret key is some positive integer k.

The idea of this cryptosystem, as well as the starting point of many others, is to associate characters with numbers, because we already have many operations and functions that work with numbers. In this example, we'll first only consider messages that consist of uppercase letters and spaces, and associate each of these letters with a number as follows:

Character	Value	Character	Value
`'A'`	0	`'O'`	14
`'B'`	1	`'P'`	15
`'C'`	2	`'Q'`	16
`'D'`	3	`'R'`	17
`'E'`	4	`'S'`	18
`'F'`	5	`'T'`	19
`'G'`	6	`'U'`	20
`'H'`	7	`'V'`	21
`'I'`	8	`'W'`	22
`'J'`	9	`'X'`	23
`'K'`	10	`'Y'`	24
`'L'`	11	`'Z'`	25
`'M'`	12	`' '`	26
`'N'`	13		

In Python, we can implement this conversion as follows:

```
LETTERS = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ '


def letter_to_num(c: str) -> int:
    """Return the number that corresponds to the given let

    Preconditions:
        - len(c) == 1 and c in LETTERS
    """
    return str.index(LETTERS, c)


def num_to_letter(n: int) -> str:
    """Return the letter that corresponds to the given num

    Preconditions:
        - 0 <= n < len(LETTERS)
    """
    return LETTERS[n]
```

In the Caesar cipher, the secret key k is an integer from the set $\mathcal{K} = \{1, 2, \ldots, 26\}$. Before sending any messages, Alice and Bob meet and decide on a secret key from this set.

Now when Alice wants to send a string message m to Bob, she *encrypts* her message as follows.

- For each character of m:
  - Alice shifts the character by adding the secret key k to its corresponding number, taking remainders modulo 27, the length of `LETTERS`. Note that the space character `' '` comes after `'Z'`.

For example, if $k = 3$, and the plaintext message is `'HAPPY'`, encryption happens as follows:

Plaintext character	Corresponding Integer	Shifted Integer	Ciphertext character
`'H'`	7	10	`'K'`
`'A'`	0	3	`'D'`
`'P'`	15	18	`'S'`
`'P'`	15	18	`'S'`
`'Y'`	24	0	`'A'`

The corresponding ciphertext is `'KDSSA'`. Note that the `Y`, when shifted by 3, wraps around to become `A`.

Then when Bob receives the ciphertext `'KDSSA'`, he decrypts the ciphertext by applying the corresponding shift in reverse (subtracting the secret key k instead of adding it). We can implement this in Python as follows:[2]

[2] Note that we've dropped the `_so_far` suffix on these accumulator variables now that you're more experience writing loops!

```
def encrypt_caesar(k: int, plaintext: str) -> str:
    """Return the encrypted message using the Caesar cipher with key k.

    Preconditions:
        - all(x in LETTERS for x in plaintext))
        - 1 <= k <= 26
    """
    ciphertext = ''

    for letter in plaintext:
        ciphertext = ciphertext + num_to_letter((letter_to_num(letter) + k) % len(LETTERS))

    return ciphertext


def decrypt_caesar(k: int, ciphertext: str) -> str:
    """Return the decrypted message using the Caesar cipher with key k.

    Preconditions:
        - all(x in LETTERS for x in ciphertext))
        - 1 <= k <= 26
    """
    plaintext = ''

    for letter in ciphertext:
        plaintext = plaintext + num_to_letter((letter_to_num(letter) - k) % len(LETTERS))

    return plaintext
```

Expanding the set of letters

In our example above, we restricted ourselves to only upper-case letters and spaces. But the key mathematical idea of the Caesar cipher, shifting letters based on a secret key k used as an offset, generalizes to larger sets of letters.

To see how to do this, first we recall two built-in Python functions from 2.9 Application: Representing Text:

```
>>> ord('A')  # Convert a character into an integer
65
>>> chr(33)  # Convert an integer into a character
'!'
```

Using these two functions, we can modify our `encrypt` and `decrypt` functions in the Caesar cipher to operate on arbitrary Python strings. For simplicity, we'll stick only to the first 128 characters, which are known as the ASCII characters.[3] Our secret key will now take on values from the set $\{1, 2, \ldots, 127\}$.

[3] You might recall from Section 2.9 that *ASCII* is one of the earliest standard for encoding characters as natural numbers on a computer.

```
def encrypt_ascii(k: int, plaintext: str) -> str:
    """Return the encrypted message using the Caesar cipher with key k.

    Preconditions:
        - all(ord(c) < 128 for c in plaintext))
        - 1 <= k <= 127

    >>> encrypt_ascii(4, 'Good morning!')
    'Ksshtqsvsmrk%'
    """
    ciphertext = ''

    for letter in plaintext:
        ciphertext = ciphertext + chr((ord(letter) + k) % 128)

    return ciphertext


def decrypt_ascii(k: int, ciphertext: str) -> str:
    """Return the decrypted message using the Caesar cipher with key k.

    Preconditions:
        - all(ord(c) < 128 for c in ciphertext))
        - 1 <= k <= 127

    >>> decrypt_ascii(4, 'Ksshtqsvsmrk%')
    'Good morning!'
    """
    plaintext = ''

    for letter in ciphertext:
        plaintext += chr((ord(letter) - k) % 128)

    return plaintext
```

Caesar cipher: correctness vs. security

So far, we've focused on the *correctness* of the Caesar cipher by implementing Python functions to encrypt and decrypt data using the cipher. But remember that a *secure* symmetric-key cryptosystem requires two properties: correctness and security!

> (**security**) For all $k \in \mathcal{K}$ and $m \in \mathcal{P}$, if an eavesdropper only knows the value of $c = Esecret(k, m)$ but does not know k, it is computationally infeasible to find m.

In practice, the Caesar cipher is definitely not secure, as it is very possible for an eavesdropper to write a Python program to try all possible secret key to decrypt a ciphertext, and pick out the most likely message that Alice sent. In other words, with only 27 (or even 127) possible keys, it is very "computationally feasible" for an eavesdropper to decrypt the messages.

The main takeaway from this example: it is usually not too hard to create (or implement) a *correct* cryptosystem, but it can be very challenging to create one that's secure! Also, don't use the Caesar cipher to transmit secret messages in real-world applications.