Cryptography

David R. Kohel

July 11, 2008

CONTENTS

Preface

When embarking on a project to write a book in a subject saturated with such books, the natural question to ask is: what niche does this book fill not satisfied by other books on the subject? The subject of cryptography attracts participants from many academic disciplines, from mathematics to computer science and engineering. The goal of this book is to provide an introduction which emphasizes the mathematical and algorithmic components and building blocks suitable for mathematics students, while liberally illustrating the theory with examples. Most textbooks for a mathematics audience limit themselves to pen and paper calculations, which fails to give the student a sense of either the asymptotic complexity for the algorithms or access to a practical range for cryptographic study. Textbooks which take a computational view usually miss the conceptual framework of the mathematics, and are either tied to a particular commercial software package or emphasize low-level computations in C or Java which requires a stronger computer science background. We choose to use the computer algebra system SAGE for experimental exploration, since this package is both freely available and designed for intuitive interactive use. We hope that this book will fill a niche by emphasizing a mathematical presentation of structures in cryptography, without sacrificing the explicit exploration of the field.

ONE

Introduction to Cryptography

Cryptography is the study of mathematical techniques for all aspects of information security. *Cryptanalysis* is the complementary science concerned with the methods to defeat these techniques. *Cryptology* is the study of cryptography and cryptanaylsis. Key features of information security information include *confidentiality* or *privacy*, *data integrity*, *authentication*, and *nonrepudiation*.

Each of these aspects of message security can addressed by standard methods in cryptography. Besides exchange of messages, tools from cryptography can be applied to sharing an access key between multiple parties so that no one person can gain access to a vault by any two of them can. Another role is in the design of electronic forms of cash.

1.1   Definitions and Terminology

Encryption = the process of disguising a message so as to hide the information it contains; this process can include both encoding and enciphering (see definitions below).

Protocol = an algorithm, defined by a sequence of steps, precisely specifying the actions of multiple parties in order to achieve an objective.

Plaintext = the message to be transmitted or stored.

Ciphertext = the disguised message.

Alphabet = a collection of symbols, also referred to as characters.

Character = an element of an alphabet.

Bit = a character 0 or 1 of the binary alphabet.

String = a finite sequence of characters in some alphabet.

Encode = to convert a message into a representation in a standard alphabet, such as to the alphabet $\{A, \ldots, Z\}$ or to numerical alphabet.

Decode = to convert the encoded message back to its original alphabet and original form — the term plaintext will apply to either the original or the encoded form. The process of encoding a message is not an obscure process, and the result that we get can be considered equivalent to the plaintext message.

Cipher = a map from a space of plaintext to a space of ciphertext.

Encipher = to convert plaintext into ciphertext.

Decipher = to convert ciphertext back to plaintext.

Stream cipher = a cipher which acts on the plaintext one symbol at a time.

Block cipher = a cipher which acts on the plaintext in blocks of symbols.

Substitution cipher = a stream cipher which acts on the plaintext by making a substitution of the characters with elements of a new alphabet or by a permutation of the characters in the plaintext alphabet.

Transposition cipher = a block cipher which acts on the plaintext by permuting the positions of the characters in the plaintext.

Example. The following are some standard alphabets.

A, . . . , Z	26 symbols	Classical alphabet (less punctuation)
ASCII	7-bit words (128 symbols)	American standard
extended	8-bit words (256 symbols)	
ISO-8859-1	8-bit words (256 symbols)	Western European standard
Binary	{0,1}	Numerical alphabet base 2
Octal	{0,. . .,7}	Numerical alphabet base 8
Decimal	{0,. . .,9}	Numerical alphabet base 10
Hexadecimal	{0,. . .,9,a,b,c,d,e,f}	Numerical alphabet base 16

Example. The following is an example of a substitution cipher:

```
A  B  C  D  E  F  G  H  ···  Z  _
↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ···  ↓  ↓
P  C  _  O  N  A  W  Y  ···  L  S
```

which takes the plaintext BAD CAFE BED to the ciphertext CPOS ANSNO.

1.2   String Monoids

An encoding or cipher is a transformation of data or text, whether expressed in the Roman alphabet, Chinese characters, or some binary, hexadecimal, or byte encoding of this

information. We introduce here some of the structures in which this text is stored, as the natural domain and codomain on which encodings and ciphers operate. First we introduce the abstract notion of a monoid before specialising to the string monoids which form the objects of interest.

Monoids

A *monoid* is a set $\mathcal{M}$ together with a binary operation

$$\cdot : \mathcal{M} \times \mathcal{M} \longrightarrow \mathcal{M},$$

which is associative:

$$(x \cdot y) \cdot z = x \cdot (y \cdot z),$$

and which contains a distinguished element e such that $e \cdot x = x \cdot e = x$. We note that a monoid in which every element x has an inverse element y in $\mathcal{M}$, i.e. such that $x \cdot y = e$ is called a *group*. If we both the axiom of existence of inverses and the axiom asserting existence of an identity, we arrive at the notion of a *semigroup*.

Monoid homomorphisms

A *monoid homomorphism* is a map $\phi : \mathcal{M}_1 \to \mathcal{M}_2$ such that

$$\phi(x \cdot y) = \phi(x) \cdot \phi(y),$$

and which takes the identity in $\mathcal{M}_1$ to the identity in $\mathcal{M}_2$. We denote the set of all monoid homomorphism $\mathcal{M}_1 \to \mathcal{M}_2$ by

$$\mathrm{Hom}_{\mathtt{Mon}}(\mathcal{M}_1, \mathcal{M}_2),$$

or just $\mathrm{Hom}(\mathcal{M}_1, \mathcal{M}_2)$ when it is clear that $\mathcal{M}_1$ and $\mathcal{M}_2$ are both monoids. In contrast, the set of all set-theoretic maps from $\mathcal{M}_1$ to $\mathcal{M}_2$ is denoted

$$\mathrm{Hom}_{\mathtt{Set}}(\mathcal{M}_1, \mathcal{M}_2).$$

Strings

Given a finite alphabet $\mathcal{A}$ (a finite set of symbols), we write $\mathcal{A}^n$ for the n-fold product $\mathcal{A} \times \cdots \times \mathcal{A}$, whose elements we call strings of length n, and let

$$\mathcal{A}^* = \bigcup_{n=0}^{\infty} \mathcal{A}^n,$$

be the disjoint union over strings of all lengths. Then $\mathcal{A}^*$ forms a monoid under string concatentation, with identify the unique string in $\mathcal{A}^0$, called the empty string. We identity $\mathcal{A}$ with its image $\mathcal{A}^1$ in $\mathcal{A}^*$, and write a string in $\mathcal{A}^n$ as $x_1 x_2 \cdots x_n$ rather than $(x_1, x_2, \ldots, x_n)$.

The string monoids $\mathcal{A}^*$ form a very special class of monoids, called *free monoids* over $\mathcal{A}$, which are characterized by the "universal" property:

The set $\mathcal{A}$ generates $\mathcal{A}^*$, and given any map $\mathcal{A} \to \mathcal{M}$ to a monoid $\mathcal{M}$, there exists a unique monoid homomorphism $\mathcal{A}^* \to \mathcal{M}$ which extends $\mathcal{A} \to \mathcal{M}$.

Since the length of strings adds when composing (i.e. concatenating) them, clearly no element other than the empty string in $\mathcal{A}^*$ has an inverse, thus it forms a monoid but not a group.

We are interested in standard string monoids used in language and computers. For example, we let ASCII be the set of 256 binary strings of length 8, which represent text in the computer. For instance, we have the following correspondence between characters in the Roman alphabet, numeric decimal values and the binary representation of the ASCII alphabet.

Character	Number	ASCII binary
A	65	01000001
B	66	01000010
$\vdots$	$\vdots$	$\vdots$
Z	90	01001110
$\vdots$	$\vdots$	$\vdots$
a	97	01100001
b	98	01100010
$\vdots$	$\vdots$	$\vdots$
z	122	01111010

Note that the space character " " is a valid symbol in the ASCII alphabet, with numeric value 32. In particular it is a symbol distinct from the identity element of the monoid ASCII*.

We now let $\mathcal{A}$ be the alphabet consisting of the 26 symbols $\{A, B, \ldots, Z\}$. There is an obvious monoid homomorphism $\iota : \mathcal{A}^* \to \text{ASCII}^*$ induced by the inclusion $\mathcal{A} \subset \text{ASCII}$. We can define a map $\text{ASCII} \to \mathcal{A}^*$ by extending the map

$$
\begin{array}{ll}
\text{A} \longmapsto \text{A} & \quad \text{a} \longmapsto \text{A} \\
\text{B} \longmapsto \text{B} & \quad \text{b} \longmapsto \text{B} \\
\quad \vdots & \qquad \vdots \\
\text{Z} \longmapsto \text{Z} & \quad \text{z} \longmapsto \text{Z}
\end{array}
$$

to all of ASCII by sending all other characters to the empty string e. This induces a monoid homomorphism $\pi : \text{ASCII}^* \to \mathcal{A}^*$ such that the composition $\iota \circ \pi$ is the identity homomorphism on $\mathcal{A}^*$, but $\pi \circ \iota$ is far from injective on ASCII*.

This monoid homomorphism was commonly applied to plaintext in classical cryptosystems, to encode it prior to enciphering. As an example we see that

$$\iota \circ \pi(\text{The cat in the hat.}) = \text{THECATINTHEHAT},$$

but strings in $\mathcal{A}^*$ map injectively into `ASCII`*:

$$\pi \circ \iota \circ \pi(\texttt{THECATINTHEHAT}) = \pi(\texttt{THECATINTHEHAT}) = \texttt{THECATINTHEHAT}.$$

The existence of the empty string is crucial to the definition of the map from `ASCII` to $\mathcal{A}^*$, which shows that the concept of a monoid, rather than a semigroup, is the correct one for study of strings and the transformations which operate on them (under the guise of encodings or ciphers).

In what follows the domain and codomain of ciphers will be string monoids or a subset $\mathcal{A}^n$ of a string monoid $\mathcal{A}^*$ (for block ciphers). The latter ciphers may be extended naturally (in what is called ECB mode in Chapter 5) to the submonoid $(\mathcal{A}^n)^*$ of $\mathcal{A}^*$ on the larger alphabet $\mathcal{A}^n$. The concept of a string monoid gives a useful framework for understanding ciphers. A first question to ask for a cipher whose domain is a string monoid is whether that cipher is a monoid homomorphism.

1.3   Cryptosystems

The notion of a cryptosystem or encryption scheme $\mathcal{E}$ captures the idea of a distinguished set of ciphers indexed over some keyspace $\mathcal{K}$:

$$\mathcal{E} = \{E_K : \mathcal{M} \to \mathcal{C} \ : \ K \in \mathcal{K}\}.$$

To every enciphering key K there exists a deciphering key K' with deciphering map $D_{K'} : \mathcal{C} \to \mathcal{M}$. Now $\mathcal{E}$ should be thought of as a pair of algorithms E and D which take inputs (K, M) and (K', C), respectively.

We formalise this definition as follows. First, we require a collection of sets:

$$
\begin{aligned}
\mathcal{A} &= \text{plaintext alphabet} & \mathcal{A}' &= \text{ciphertext alphabet} \\
\mathcal{M} &= \text{plaintext space} & \mathcal{C} &= \text{ciphertext space} \\
\mathcal{K} &= \text{(plaintext) keyspace} & \mathcal{K}' &= \text{(ciphertext) keyspace}
\end{aligned}
$$

where $\mathcal{M}$ is a subset of $\mathcal{A}^*$, $\mathcal{C}$ is a subset of $\mathcal{A}'^*$, and $\mathcal{K}$ and $\mathcal{K}'$ are finite sets. A *cryptosystem* or *encryption scheme* is a pair (E, D) of maps

$$
\begin{aligned}
E &: \mathcal{K} \times \mathcal{M} \longrightarrow \mathcal{C} \\
D &: \mathcal{K}' \times \mathcal{C} \longrightarrow \mathcal{M}
\end{aligned}
$$

such that for each K in $\mathcal{K}$ there exists a K' in $\mathcal{K}'$ such that

$$D(K', E(K, M)) = M$$

for all M in $\mathcal{M}$.

To recover to our original notion, for fixed K we write the cipher

$$E_K = E(K, \cdot) : \mathcal{M} \to \mathcal{C},$$

and similarly

$$D_{K'} = D(K', \cdot) : \mathcal{C} \to \mathcal{M}.$$

With this notation the condition on E, D, K and K' is that $D_{K'} \circ E_K$ is the identity map on $\mathcal{M}$. In this way, we may express E and D as maps

$$E : \mathcal{K} \to \mathrm{Hom}_{\mathsf{Set}}(\mathcal{M}, \mathcal{C}),$$
$$D : \mathcal{K}' \to \mathrm{Hom}_{\mathsf{Set}}(\mathcal{C}, \mathcal{M}).$$

We will refer to E_K as a *cipher*, and note that a cipher is necessarily injective. For many cryptosystems, there will exist a unique deciphering key K' associated to each enciphering key K. A cryptosystem for which the deciphering key K' equals K (hence $\mathcal{K} = \mathcal{K}'$) or for which K' can be easily obtained from K, is said to be *symmetric*. If the deciphering key K' associated to K not easily computable from K, we say that the cryptosystem is *asymmetric* or a *public key cryptosystem*.

A fundamental principle of (symmetric key) cryptography is *Kerckhoff's principle*, that the security of a cryptosystem does not rest on the lack of knowledge of the cryptosystem $\mathcal{E} = (E, D)$. Instead, security should be based on the secrecy of the keys.

Recall that a *protocol* is an algorithm, defined by a sequence of steps, precisely specifying the actions of multiple parties in order to achieve an objective. An example of a cryptographic protocol, we describe the steps for message exchange using a symmetric key cryptosystem.

1. Alice and Bob publicly agree on a cryptosystem (E, D).
2. *For each message M Alice $\to$ Bob:*
   a) Alice and Bob agree on a secret key K.
   b) Alice computes $C = E_K(M)$ and sends it to Bob.
   c) Bob computes $M = D_K(C)$ to obtain the plaintext.

The difficulty of step 2.a) was one of the fundamental obstructions to cryptography before the advent of public key cryptography. Asymmetric cryptography provides an elegant solution to the problem of distribution of private keys.

Classical Cryptography

2.1 Substitution Ciphers

Classically, cryptosystems were character-based algorithms. Cryptosystems would substitute characters, permute (or transpose) characters, or do a combination of those operations.

Notation

Throughout the course we will denote the *plaintext alphabet* by $\mathcal{A}$ and the *ciphertext alphabet* by $\mathcal{A}'$. We write E_K for the enciphering map and $D_{K'}$ for the deciphering map, where K and K' are enciphering and deciphering keys.

Substitution Ciphers

We identify four different types of substitution ciphers.

Simple substitution ciphers

In this cryptosystem, the algorithm is a character-by-character substitution, with the key being the list of substitutions under the ordering of the alphabet. In other words, a simple substitution cipher is defined by a map $\mathcal{A} \to \mathcal{A}'$.

Suppose that we first encode a message by purging all nonalphabetic characters (e.g. numbers, spaces, and punctuation) and changing all characters to uppercase. Then the key size, which bounds the security of the system, is 26 alphabetic characters. Therefore the total number of keys is 26!, an enormous number. Nevertheless, we will see that simple substitution is very susceptible to cryptanalytic attacks.

Example. Consider this paragraph, encoded in this way, to obtain the plaintext:

```
SUPPOSETHATWEFIRSTENCODEAMESSAGEBYPURGINGALLNONALPHABETI
CCHARACTERSEGNUMBERSSPACESANDPUNCTUATIONANDCHANGINGALLCH
ARACTERSTOUPPERCASETHENTHEKEYSIZEWHICHBOUNDSTHESECURITYO
FTHESYSTEMISALPHABETICCHARACTERSTHEREFORETHETOTALNUMBERO
FKEYSISOFENORMOUSSIZENEVERTHELESSWEWILLSEETHATSIMPLESUBS
TITUTIONISVERYSUSCEPTIBLETOCRYPTANALYTICATTACKS
```

then using the enciphering key `ULOIDTGKXYCRHBPMZJQVWNFSAE`, we encipher the plaintext to obtain ciphertext:

```
QWMMPQDVKUVFDTXJQVDBOPIDUHDQQUGDLAMWJGXBGURRBPBURMKULDVX
OOKUJUOVDJQDGBWHLDJQQMUODQUBIMWBOVWUVXPBUBIOKUBGXBGURROK
UJUOVDJQVPWMMDJOUQDVKDBVKDCDAQXEDFKXOKLPWBIQVKDQDOWJXVAP
TVKDQAQVDHXQURMKULDVXOOKUJUOVDJQVKDJDTPJDVKDVPVURBWHLDJP
TCDAQXQPTDBPJHPWQQXEDBDNDJVKDRDQQFDFXRRQDDVKUVQXHMRDQWLQ
VXVWVXPBXQNDJAQWQODMVXLRDVPOJAMVUBURAVXOUVVUOCQ
```

Simple substitution ciphers can be easily broken because the cipher does not change the frequencies of the symbols of the plaintext.

Affine ciphers. A special case of simple substitution ciphers are the *affine ciphers*. If we numerically encode the alphabet $\{\texttt{A},\texttt{B}\ldots,\texttt{Z}\}$ as the elements $\{0,1,\ldots,25\}$ of $\mathbf{Z}/26\mathbf{Z}$ then we can operate on the letters by transformations of the form $x \mapsto ax + b$, for any a for which $\mathrm{GCD}(a,26) = 1$. *What happens if a is not coprime to 26?*

An affine cipher for which $a = 1$ is called a *translation cipher*. Enciphering in a translation cipher is achieved by the performing b cyclic shift operations ($\texttt{A} \mapsto \texttt{B}$, $\texttt{B} \mapsto \texttt{C}$, etc.) on the underlying alphabet. Classically a translation cipher is known as **Caesar's cipher**, after Julius Caesar, who used this method to communicate with his generals. For example, using $b = 3$ we obtain the cipher $\texttt{A} \mapsto \texttt{D}$, $\texttt{B} \mapsto \texttt{E}, \ldots, \texttt{Z} \mapsto \texttt{C}$.

Homophonic substitution ciphers

In this cryptosystem the deciphering is a function from a larger alphabet $\mathcal{A}'$ to the alphabet $\mathcal{A}$, but an enciphering of the plaintext can take a character to any one of the elements in the preimage.

One way to realize a homophonic cipher is to begin with m different substitution keys, and with each substitution, make a random choice of which key to use. For instance, suppose we take $\mathcal{A}$ to be own standard 26 character alphabet, and let the cipher alphabet $\mathcal{A}'$ be the set of character pairs. Suppose now that we the pair of substitution keys in the ciphertext alphabet:

Chapter 2.   Classical Cryptography

LV MJ CW XP QO IG EZ NB YH UA DS RK TF MJ XO SL PE NU FV TC QD RK YH GW AB ZI
UD PY KG JN SH MC FT LX BQ EI VR ZA OW XP HO DJ CY RN ZV WT LA SF BM GU QK IE

as our homophonic key.

In order to encipher the message:

> "Always look on the bright side of life."

we strip it down to our plaintext alphabet to get the plaintext string:

```
ALWAYSLOOKONTHEBRIGHTSIDEOFLIFE
```

Then each of the following strings are valid ciphertext:

```
LVRKYHLVABZVRKHOHOVRHOXPWTLXQOMJNUYHFTNBTCFVYHJNQOHOMCZABQMCSH
UDZAYHUDQKZVZAHOXODSXOMJTCLXSHMJRNBQFTNBWTZVBQXPQOHOIGZABQMCSH
LVRKYHUDQKZVRKXOXODSHOXPTCLXQOPYRNBQEZNBTCFVBQXPSHHOIGZAYHMCSH
LVZABMUDABFVRKHOHODSHOXPWTLXQOPYRNBQEZNBTCZVBQXPQOXOIGZABQMCQO
```

Moreover, each uniquely deciphers back to the original plaintext.

Polyalphabetic substitution ciphers

A polyalphabetic substitution cipher, like the homophonic cipher, uses multiple keys, but the choice of key is not selected randomly, rather it is determined based on the position within the plaintext. Most polyalphabetic ciphers are *periodic substitution ciphers*, which substitutes the $(mj + i)$-th plaintext character using the i-th key, where $1 \leq i \leq m$. The number m is called the *period*.

Vigenère cipher. The Vigenère cipher is a polyalphabetic translation cipher, that is, each of the m keys specifies an affine translation.

Suppose that we take our standard alphabet $\{A, B, \ldots, Z\}$ with the bijection with $\mathbf{Z}/26\mathbf{Z} = \{0, 1, \ldots, 25\}$. Then beginning with the message:

> Human salvation lies in the hands
> of the creatively maladjusted.

This gives the encoded plaintext:

HUMANSALVATIONLIESINTHEHANDSOFTHECREATIVELYMALADJUSTED

The with the enciphering key `UVLOID`, the Vigenère enciphering is given by performing the column additions:

```
HUMANS ALVATI ONLIES INTHEH ANDSOF THECRE ATIVEL YMALAD JUSTED
UVLOID UVLOID UVLOID UVLOID UVLOID UVLOID UVLOID UVLOID UVLOID
------------------------------------------------------------
BPXOVV UGGOBL IIWWMV CIEVMK UIOGWI NCPQZH UOTJMO SHLZIG DPDHMG
```

Recall that the addition is to be carried out in $\mathbf{Z}/26\mathbf{Z}$, with the bijection defined by the following table:

```
A  B  C  D  E  F  G  H  I  J   K   L   M   N   O   P   Q   R   S   T   U   V   W   X   Y   Z
0  1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25
```

Polygram substitution ciphers

A polygram substitution cipher is a cryptosystem in which blocks of characters are substituted in groups. For instance (for a particular key) `AA` could map to `NO`, `AB` to `IR`, `JU` to `AQ`, etc. These cryptosystems make cryptanalysis harder by destroying the single character frequencies, preserved under simple substitution ciphers.

General affine ciphers. An affine cipher can be generalised to polygram ciphers. Rather than a map $m \mapsto c = ma + b$, we can apply a linear transformation of vectors

$$u = (m_1, \ldots, m_n) \mapsto (c_1, \ldots, c_n) = uA + v,$$

for some invertible matrix $A = (a_{ij})$ and vector $v = (b_1, \ldots, b_n)$. As before we numerically encode an alphabet $\{A, B \ldots, Z\}$ as the elements $\{0, 1, \ldots, 25\}$ of $\mathbf{Z}/26\mathbf{Z}$. Then each n-tuple of characters $m_1 m_2 \ldots m_n$ is identified with the vector $u = (m_1, m_2, \ldots, m_n)$. Note that matrix multiplication is defined as usual, so that

$$c_j = (\sum_{i=1}^{n} m_i a_{ij}) + b_j,$$

with the result interpretted modulo 26 as an element of $\mathbf{Z}/26\mathbf{Z}$.

As a special case, consider 2-character polygrams, so that

$$\texttt{AA} = (0,0), \ldots, \texttt{ZY} = (25,24), \texttt{ZZ} = (25,25).$$

The matrix A given by

$$\begin{pmatrix} 1 & 8 \\ 21 & 3 \end{pmatrix}$$

and vector $v = (13,14)$ defines a map

$$
\begin{aligned}
\texttt{AA} = (\ 0,\ 0) &\mapsto (13,14) = \texttt{NO} \\
\vdots \qquad\quad & \qquad\qquad \vdots \\
\texttt{ZY} = (25,24) &\mapsto (18,23) = \texttt{WA} \\
\texttt{ZZ} = (25,25) &\mapsto (18,23) = \texttt{RD}
\end{aligned}
$$

which is a simple substitution on the 2-character polygrams. Note that the number of affine ciphers is much less than all possible substutions, but grows exponentially in the number n of characters.

2.2 Transposition Ciphers

Recall that a substitution cipher permutes the characters of the plaintext alphabet, or may, more generally, map the plaintext characters into a different ciphertext alphabet. In a *transposition cipher*, the symbols of the plaintext remain the same unchanged, but their order is permuted by a permutation of the index positions. Unlike substitution ciphers, transposition ciphers are *block ciphers*.

The relation between substitution ciphers and transposition ciphers is illustrated in Table 2.1. The characters and their positions of the plaintext string $\texttt{ACATINTHEHAT}$ appear in a graph with a character axis c and a position index i for the 12 character block $1 \le i \le n$. We represented as a graph a substitution cipher (with equal plaintext and ciphertext alphabets) is realised as a permutation of the rows of the array, while a transposition cipher is realised by permuting the columns in fixed size blocks, in this case 12.

Permutation groups

The *symmetric group* S_n is the set of all bijective maps from the set $\{1, \ldots, n\}$ to itself, and we call an elements π of S_n a permutation. We denote the n-th composition of π with itself by π^n. As a function write π on the left, so that the image of j is $\pi(j)$. An element of S_n is called a *transposition* if and only if it exhanges exactly two elements, leaving all others fixed.

	1	2	3	4	5	6	7	8	9	10	11	12
Z												
Y												
X												
W												
V												
U												
T				T			T					T
S												
R												
Q												
P												
O												
N						N						
M												
L												
K												
J												
I					I							
H								H		H		
G												
F												
E									E			
D												
C		C										
B												
A	A		A								A	

Table 2.1: Transposition and substitution axes for `ACATINTHEHAT`

List notation for permutations

The map $\pi(j) = i_j$ can be denoted by $[i_1, \ldots, i_n]$. This is the way, in effect, that we have described a key for a substitution cipher — we list the sequence of characters in the image of A, B, C, etc. Although these permutations act on the set of the characters $A, \ldots, Z$ rather than the integers $1, \ldots, n$, the principle is identical.

Cycle notation and orbit structure

Given a permutation π in S_n there exists a unique *orbit decomposition*:

$$\{1, \ldots, n\} = \bigcup_{k=1}^{t} \{\pi^j(i_k) \ : \ j \in \mathbf{Z}\},$$

where union can be taken over disjoint sets, i.e. i_k is not equal to $\pi^j(i_\ell)$ for any j unless $k = \ell$. The sets $\{\pi^j(i_k) \ : \ j \in \mathbf{Z}\}$ are called the *orbits* of π, and the cycle lengths of π are the sizes $d_1, \ldots, d_t$ of the orbits.

Asociated to any orbit decomposition we can express an element π as

$$\pi = \left(i_1, \pi(i_1), \ldots, \pi^{d_1-1}(i_1)\right) \cdots \left(i_t, \pi(i_t), \ldots, \pi^{d_t-1}(i_t)\right)$$

Note that if $d_k = 1$, then we omit this term, and the identity permutation can be written just as 1. This notation gives more information about the permutation π and is more compact for simple permutations such as transpositions.

Simple columnar transposition

A classical example of a transposition cipher is an (r, s)-simple columnar transposition. In this cryptosystem the plaintext is written in blocks as r rows of fixed length s. The ciphertext is read off as the columns of this array. Suppose we begin with the plaintext:

```
I was riding on the Mayflower
When I thought I spied some land
I yelled for Captain Arab
I have yuh understand
Who came running to the deck
Said, "Boys, forget the whale
Look on over yonder
Cut the engines
Change the sail
Haul on the bowline"
We sang that melody
Like all tough sailors do
When they are far away at sea
```

Stripped to our plaintext alphbet and written in lines of 36 characters each, we have the plaintext:

```
IWASRIDINGONTHEMAYFLOWERWHENITHOUGHT
ISPIEDSOMELANDIYELLEDFORCAPTAINARABI
HAVEYUHUNDERSTANDWHOCAMERUNNINGTOTHE
DECKSAIDBOYSFORGETTHEWHALELOOKONOVER
YONDERCUTTHEENGINESCHANGETHESAILHAUL
ONTHEBOWLINEWESANGTHATMELODYLIKEALLT
OUGHSAILORSDOWHENTHEYAREFARAWAYATSEA
```

Reading off the columns, we obtain the following ciphertext under the columnar transposition cipher:

```
IIHDYOOWSAEONUAPVCNTGSIEKDHHREYSEESIDUARBA
DSHICOIIOUDUWLNMNBTLOGEDOTIROLEYHNSNARSEED
TNSFEWOHDTONEWEIARGSHMYNGIAEAEDENNNYLWTEGT
FLHTSTHLEOHCHEODCEHAYWFAWATAEOMHNMRRREAGEE
WCRLELFHAUETOAEPNLHDRNTNOEYAIAIOSLWTINKAIA
HNGOIKYOATNLEAUROOHATGATVALSHBHEULETIERLTA
```

Simple columnar transpositions impose unnecessarily restrictive conditions on the form of the transposition, but were widely used as a component of ciphers used until the 1950's. More general columnar transpositions allow for permuations of the columns before reading them off.

General transposition ciphers

A general transposition cipher of block length m allows $m!$ different permuations. For $m = 7$, this is a mere 5040 permutations, but for block length 36, this gives

$$371993326789901217467999448150835200000000$$

possibilities. One such permutation, given in cycle notation, is:

$$(1, 12, 5, 36, 30, 31, 4, 28, 33, 22, 26, 17, 10, 16, 14, 23, 18, 35, 32)$$
$$(2, 9, 3, 25, 15, 7, 21, 6, 29, 34, 11, 27, 19, 24)(8, 13, 20).$$

Applied to the above ciphertext this gives a the ciphertext

```
NNWNTIOTAMERLEDHGHRIIHYWEAFUGHSIWOOT
AMCTIADNPYPEEOSDEBRODALSIELRANIIFLAI
RNRNEICSVNNYOMHTDHEUUUWAADHOTGEHAETN
SBLOROEFCGLSHHIOOEADAETERETOVOKDWYNK
ETEELSHENIHECNCNTUGURTEOGNSHAIDYAHLA
ELLYTLAWTADEHMOEILEWBOGNSNTALKHOTNEI
DOFAAWYOGERSERIWREELAATUHNHTSYHOASAA
```

Despite the large number of possible permutations, the unmasked stucture of the plaintext permits an adversary to decipher transposition ciphertext.

Exercises

Substitution ciphers

Exercise 2.1 *Determine the number of possible keys for the affine substitution ciphers. Is this sufficient to have a secure cryptosystem?*

Transposition ciphers

Exercise 2.2 *Show that for every π in S_n, there exists an positive integer m, such that π^m is the identity map, and such that m divides $n!$. The smallest such m is called the order of π.*

Exercise 2.3 *How many transpositions exist in S_n? Describe the elements of order 2 in S_n and determine their number.*

Exercise 2.4 *Show that every element of S_n can be expressed as the composition of at most n transpositions.*

Exercise 2.5 *What is the order of a permutation with cycle lengths $d_1, \ldots, d_t$? How does this solve the previous exercise concerning the order of a permutation?*

Exercise 2.6 *What is the block length m of an (r, s)-simple columnar transposition? Describe the permutation. Hint: it may be easier to describe the permutation if the index set is $\{0, \ldots, m-1\}$.*

Exercise 2.7 *Show that the (r, r)-simple columnar transposition has order 2. What is the order of the cipher for $(r, s) = (3, 5)$? Determine the permutation in cycle notation for this cipher. Determine the permutation in cycle notation for the $(7, 36)$-simple columnar transposition used in this chapter.*

Elementary Cryptanalysis

The most direct attack on a cryptosystem is an *exhaustive key search* attack. The key size therefore provides a lower bound on the security of a cryptosystem. As an example we compare the key sizes of several of the cryptosystems we have introduced so far. We assume that the alphabet for each is the 26 character alphabet.

> Substitution ciphers:
>> Simple substitution ciphers: 26!
>> Affine substitution ciphers: $\varphi(26) \cdot 26 = 12 \cdot 26 = 312$
>> Translation substitution ciphers: 26
> Transposition ciphers:
>> Transposition ciphers (of block length m): $m!$
> Enigma :
>> Rotor choices (3 of 5): 60
>> Rotor positions: $26^3 = 17576$
>> Plugboard settings: 105578918576
>> Total combinations: 111339304373506560

The size of the keyspace is a naive measure, but provides an upper bound on the security of a cryptosystem. This measure ignores any structure, like character frequencies, which might remain intact following encryption.

3.1   Classification of Cryptanalytic Attacks

We do not consider enumeration of all keys a valid cryptanalytic attack, since no well-designed cryptosystem is susceptible to such an approach. The types of legitimate attacks which we consider can be classified in three categories: ciphertext-only attack, known plaintext attack, and chosen plainext attack.

Ciphertext-only Attack

The cryptanalyst intercepts one or more messages all encoded with the same encryption algorithm.

Goal: Recover the original plaintext or plaintexts, to discover the deciphering key or find an algorithm for deciphering subsequent messages enciphered with the same key.

Known Plaintext Attack

The cryptanalyst has access to not only the ciphertext, but also the plaintext for one or more of the messages.

Goal: Recover the deciphering key or find an algorithm for deciphering subsequent messages (or the remaining plaintext) enciphered which use the same key.

Chosen Plainext Attack

The cryptanalyst has access to ciphertext for which he or she specified he plaintext.

Goal: Recover the discover the deciphering key or find an algorithm for deciphering subsequent messages enciphered with the same key.

For the remainder of the chapter we consider cryptanalytic techniques employed against classical cryptosystems for ciphertext-only attacks.

3.2 Cryptanalysis by Frequency Analysis

Given a sample of an English language newspaper text (stripped of spaces, punctuation and other extraneous characters) the following gives the approximate percentage of occurrences of each character.

A	B	C	D	E	F	G	H	I	J	K	L	M
7.3	0.9	3.0	4.4	13	2.8	1.6	3.5	7.4	0.2	0.3	3.5	2.5

N	O	P	Q	R	S	T	U	V	W	X	Y	Z
7.8	7.4	2.7	0.3	7.7	6.3	9.3	2.7	1.3	1.6	0.5	1.9	0.1

The relative frequencies can change according to subject matter and style of writing, but it is still possible to pick out those characters with a high frequency of occurence and those which are rare:

High frequency: $\{E, I, O, A\}$ (vowels)   **Low frequency:** $\{J, K, Q, X, Z\}$
$\{T, N, R, S\}$ (consonants)

Examples of Cryptanalysis

Let's begin with the substitution ciphertext constructed previously:

```
QWMMPQDVKUVFDTXJQVDBOPIDUHDQQUGDLAMWJGXBGURRBPBURMKULDVX
OOKUJUOVDJQDGBWHLDJQQMUODQUBIMWBOVWUVXPBUBIOKUBGXBGURROK
UJUOVDJQVPWMMDJOUQDVKDBVKDCDAQXEDFKXOKLPWBIQVKDQDOWJXVAP
TVKDQAQVDHXQURMKULDVXOOKUJUOVDJQVKDJDTPJDVKDVPVURBWHLDJP
TCDAQXQPTDBPJHPWQQXEDBDNDJVKDRDQQFDFXRRQDDVKUVQXHMRDQWLQ
VXVWVXPBXQNDJAQWQODMVXLRDVPOJAMVUBURAVXOUVVUOCQ
```

We find the following character counts, scaled to that of a 1000 character input text.

A	24.0964	J	54.2169	S	0
B	54.2169	K	51.2048	T	15.0602
C	9.0361	L	24.0964	U	81.3253
D	129.5181	M	36.1446	V	99.3976
E	6.0241	N	6.0241	W	39.1566
F	12.0482	O	57.2289	X	57.2289
G	18.0723	P	45.1807	Y	0
H	18.0723	Q	96.3855	Z	0
I	12.0482	R	39.1566		

The distributions look like a frequency preserving substitution cipher. We guess that the enciphering takes $E \mapsto D$ and $T \mapsto V$ or $T \mapsto Q$. The most frequent characters are D, V, Q, V, U, O, J, K, B and E, N, S, Y, Z are of lowest frequency.

Equating high frequency and low frequency characters, we might first guess

$$\{E, I, O, A, T, N, R, S\} \mapsto \{D, V, Q, U, O, J, K, B\}$$

and

$$\{J, K, Q, X, Z\} \mapsto \{E, N, S, Y, Z\}$$

How would you go about reconstructing the entire text?

Index of Coincidence

In the 1920's William Friedman introduced the *index of coincidence* as a measure of the variation of character frequencies in text from a uniform distribution. The index of coincidence of a text space (e.g. that of all plaintext or ciphertext) is defined to be the probability that two randomly chosen characters are equal. In a language over an alphabet

of size n, suppose that p_i is the probability of a random character is the i-th character in a string of length N. Then, in the limit as the string length N goes to infinity, the index of coincidence in that language is:

$$\sum_{i=1}^{n} p_i^2.$$

Over an alphabet of 26 characters, the coincidence index of random text is

$$\sum_{i=1}^{26} (1/26)^2 = 1/26 \cong 0.0385.$$

For English text, the coincidence index is around 0.0661. For a finite string of length N, we the index of coincidence is defined to be:

$$\frac{\sum_{i=1}^{n} n_i(n_i - 1)}{N(N-1)},$$

where n_i is the number of occurrences of the i-th character in the string.

Theorem 3.1 *The expected index of coincidence of a ciphertext of length N, output from a period m cipher, which is defined by m independent substitutions ciphers at each position in arithmetic progression the $i + jm$, is*

$$\frac{1}{m}\left(\frac{N-m}{N-1}\right) i_{\mathrm{X}} + \frac{(m-1)}{m}\left(\frac{N}{N-1}\right) i_n,$$

where i_{X} is the index of coincidence of the space X, the size of the alphabet is n, and and $i_n = 1/n$ is the index of coincidence of random text in that space.

Example 3.2 *Consider the ciphertext enciphered with a Vigenère cipher:*

```
OOEXQGHXINMFRTRIFSSMZRWLYOWTTWTJIWMOBEDAXHVHSFTRIQKMENXZ
PNQWMCVEJTWJTOHTJXWYIFPSVIWEMNUVWHMCXZTCLFSCVNDLWTENUHSY
KVCTGMGYXSYELVAVLTZRVHRUHAGICKIVAHORLWSUNLGZECLSSSWJLSKO
GWVDXHDECLBBMYWXHFAOVUVHLWCSYEVVWCJGGQFFVEOAZTQHLONXGAHO
GDTERUEQDIDLLWCMLGZJLOEJTVLZKZAWRIFISUEWWLIXKWNISKLQZHKH
WHLIEIKZORSOLSUCHAZAIQACIEPIKIELPWHWEUQSKELCDDSKZRYVNDLW
TMNKLWSIFMFVHAPAZLNSRVTEDEMYOTDLQUEIIMEWEBJWRXSYEVLTRVGJ
KHYISCYCPWTTOEWANHDPWHWEPIKKODLKIEYRPDKAIWSGINZKZASDSKTI
TZPDPSOILWIERRVUIQLLHFRZKZADKCKLLEEHJLAWWVDWHFALOEOQW
```

The coincidence index of this text is 0.0439. This would suggest a period of approximately 5. We will see that this is a bad estimate. Note that this doesn't disprove the theorem, it just shows that the statistical errors are too great and that we would need a much larger sample size to converge to this theoretical expectation, or that the substitutions employed were not independent.

Exercise. Explain why ciphertext for a particular key need not follow the behavior predicted by the theorem.

Decimation of Sequences

For a sequence $S = s_1 s_2 s_3 \ldots$ and positive integers m and k such that $1 \leq k \leq m$, we denote the k-th decimation of period m as the sequence $s_k s_{m+k} s_{2m+k} \ldots$. If S is a ciphertext string (a sequence of characters in the alphabet $\mathcal{A}'$) enciphered by a cipher with period m, then the decimations of period m capture the structure of the cipher without periods.

Example 3.3 *If we take the previous ciphertext and average the coincidence indices of each of the k-the decimated sequences of period m, we find:*

m	CI	m	CI	m	CI
1	0.0439	6	0.0424	11	0.0653
2	0.0438	7	0.0442	12	0.0408
3	0.0435	8	0.0414	13	0.0445
4	0.0434	9	0.0438	14	0.0418
5	0.0421	10	0.0407	15	0.0423

From this table, the correct period, 11, is obvious.

Exercise. What do you expect to see in such a table if the period is composite? Hint: consider the period of the decimated sequence, and apply the theorem.

Kasiski method

The Prussian military officer Friedrich Kasiski made the following observation on the Vigenère cipher in 1863. If a frequently occurring pattern, such as THE is aligned at the same position with respect to the period, then the same three characters will appear in the ciphertext, at a distance which is an exact multiple of m. By looking for frequently occurring strings in the ciphertext, and measuring the most frequent divisors of the displacements of these strings, it is often possible to identify the period, hence to reduce to a simple substitution.

Example 3.4 *Returning to the ciphertext of Example 3.2, we find that the three substrings* SYE*,* ZKZ*, and* KZA *each occur three times. The positions of these occurrences are:*

$$
\begin{array}{ll}
\texttt{SYE}: & 122,\ 196,\ 383 \\
\texttt{ZKZ}: & 252,\ 439,\ 472 \\
\texttt{KZA}: & 253,\ 440,\ 473
\end{array}
$$

Note that ZKZ *and* KZA *are substrings of the four character string* ZKZA *appearing three times in the ciphertext! Moreover two of the occurences of the string* SYE *appear as substrings of the longer string* SYEV.

Now we look for common divisors of the differences between the positions of the frequently occurring substrings.

$$
\begin{array}{llll}
196 - 122 & = 2 \cdot 37 & 439 - 252 & = 11 \cdot 17 \\
383 - 196 & = 11 \cdot 17 & 472 - 439 & = 3 \cdot 11 \\
383 - 122 & = 3^2 \cdot 29 & 472 - 252 & = 2^2 \cdot 5 \cdot 11
\end{array}
$$

We see that our guess of 11 *for the period appears as a divisor of the distances between each of the occurrences of the common four character substring, and divides one of the differences of the other three string characters.*

Exercise. If 11 is the correct period, why does it not appear in all of the differences above? Which of the occurences can be attributed as random?

3.3   Breaking the Vigenère Cipher

Now that we have established that the period is 11, we can write the ciphertext in blocks and look at the strings which occur frequently at the same position within blocks.

	1	2	3	4
1	OOEXQGHXINM	FRTRIFSSMZR	WLYOWTTWTJI	WMOBEDAXHVH
2	SFTRIQKMENX	ZPNQWMCVEJT	WJTOHTJXWYI	FPSVIWEMNUV
3	WHMCXZTCLFS	CVNDLWTENUH	SYKVCTGMGYX	SYELVAVLTZR
4	VHRUHAGICKI	VAHORLWSUNL	GZECLSSSWJL	SKOGWVDXHDE
5	CLBBMYWXHFA	OVUVHLWCSYE	VVWCJGGQFFV	EOAZTQHLONX
6	GAHOGDTERUE	QDIDLLWCMLG	ZJLOEJTVLZK	ZAWRIFISUEW
7	WLIXKWNISKL	QZHKHWHLIEI	KZORSOLSUCH	AZAIQACIEPI
8	KIELPWHWEUQ	SKELCDDSKZR	YVNDLWTMNKL	WSIFMFVHAPA
9	ZLNSRVTEDEM	YOTDLQUEIIM	EWEBJWRXSYE	VLTRVGJKHYI
10	SCYCPWTTOEW	ANHDPWHWEPI	KKODLKIEYRP	DKAIWSGINZK
11	ZASDSKTITZP	DPSOILWIERR	VUIQLLHFRZK	ZADKCKLLEEH
12	JLAWWVDWHFA	LOEOQW		

Some of the longer strings which appear more than one at distances which are a multiple of 11 are given in the following table. The first column indicates the number of times the full string appears.

#	1	2	3	4	5	6	7	8	9	10	11
3	Z	A								Z	K
2			N	D	L	W	T				
2				P	W	H	W	E			
2	V								S	Y	E
2						L	W	C			

Now let's guess what the translations are at each of the periods. The following is a table of common characters in each of the 11 decimations of period 11, organized by the numbers of their appearances.

i	9	8	7	6	5
1				S,W,Z	V
2				L	A
3				F	T
4			D	O	R
5			L		I,W
6	W				L
7	T			H	W
8				I,S,X	E
9			E		H
10			Z		E,Y
11			I		

These characters are not themselves the characters in the key, but if we assume that one of these frequently occuring characters is the image of E, then we can make a guess at the key. The table below gives the enciphering characters which take the corresponding character in the previous table to E.

i	9	8	7	6	5
1				M,I,F	J
2				T	E
3				Z	L
4			B	Q	N
5			T		W,I
6	I				T
7	L			X	I
8				W,M,H	A
9			A		X
10			F		A,G
11			W		

Checking possible keys, the partial key `I****IL*A*W` gives the following text which is suggestive of English:

```
              1               2               3               4
    1   W****OS*I*I    N****ND*M*N    E****BE*T*E    E****LL*H*D
    2   A****YV*E*T    H****UN*E*P    E****BU*W*E    N****EP*N*R
    3   E****HE*L*O    K****EE*N*D    A****BR*G*T    A****IG*T*N
    4   D****IR*C*E    D****TH*U*H    O****AD*W*H    A****DO*H*A
    5   K****GH*H*W    W****TH*S*A    D****OR*F*R    M****YS*O*T
    6   O****LE*R*A    Y****TH*M*C    H****RE*L*G    H****NT*U*S
    7   E****EY*S*H    Y****ES*I*E    S****WW*U*D    I****IN*E*E
    8   S****ES*E*M    A****LO*K*N    G****EE*N*H    E****NG*A*W
    9   H****DE*D*I    G****YF*I*I    M****EC*S*A    D****OU*H*E
   10   A****EE*O*S    I****ES*E*E    S****ST*Y*L    L****AR*N*G
   11   H****SE*T*L    L****TH*E*N    D****TS*R*G    H****SW*E*D
   12   R****DO*H*W    T****E
```

3.4   Cryptanalysis of Transposition Ciphers

A transposition cipher can easily be recognized by an analysis of character frequencies. Iterating transposition ciphers can greatly increase security, but as with substitution ciphers, almost all such ciphers can be broken. Although many modern cryptosystems incorporate transposition ciphers, the operation on large blocks has the disadvantage of requiring a lot of memory.

3.5   Statistical Measures

So far we have focused on Vigenère ciphers, and their reduction to monoalphabetic substitutions. Here we show how to use SAGE to complete the final step of breaking these ciphers. Recall that the reduction to monoalphabetic substitution is done by the process of *decimation*, by which we lose all 2-character frequency structure of the language. A more sophisticated approach will be necessary for breaking more complex ciphers.

Correlation. We first introduce the concept of correlation of two functions. Let X and Y be discrete random variables on a space Ω of n symbols, with values $(x_1, x_2, \ldots, x_n)$ and $(y_1, y_2, \ldots, y_n)$, respectively. For simplicity we assume that all n symbols of Ω occur with equal likihood. We define the correlation of the two sequences to be

$$\text{Corr}(X, Y) = \frac{\sum_{i=1}^{n}(x_i - \mu(X))(y_i - \mu(Y))}{\sigma(X)\sigma(Y)}$$

and where $\mu(\mathrm{X})$ and $\mu(\mathrm{Y})$ are the respective *means* of X and Y:

$$\mu(\mathrm{X}) = \frac{1}{n} \sum_{i=1}^{n} x_i, \quad \mu(\mathrm{Y}) = \frac{1}{n} \sum_{i=1}^{n} y_i,$$

and the terms in the denomiators are:

$$\sigma(\mathrm{X}) = \Big(\sum_{i=1}^{n} (x_i - \mu(\mathrm{X}))^2 \Big)^{1/2}, \quad \sigma(\mathrm{Y}) = \Big(\sum_{i=1}^{n} (y_i - \mu(\mathrm{Y}))^2 \Big)^{1/2},$$

called the *standard deviations* of X and Y. The correlation of two sequences will be a real number between 1 and -1, which measures the linear relation between two sequences. When the random variables X and Y are probability functions, the means each reduce to $1/n$ (on a probability space Ω with equal probabilities).

Exercises

One important measure of a cryptographic text is the *coincidence index*. For random text (of uniformly distributed characters) in an alphabet of size 26, the coincidence index is approximately 0.0385. For English text, this value is closer to 0.0661. Therefore we should be able to pick out text which is a simple substitution or a transposition of English text, since the a coincidence index remains unchanged.

The SAGE crypto string functions

$$\texttt{coincidence\_index} \text{ and } \texttt{frequency\_distribution}$$

provide functionality for analysis of the ciphertexts in the exercises. Moreover, for a SAGE string $\texttt{s}$ the k-th *decimation* of period m for that string is given by $\texttt{s[k::m]}$ (short for $\texttt{s[k:len(s):m]}$).

Exercise 3.1 *Complete the deciphering of the Vigenère ciphertext of Section 3.3 . What do you note about the relation between the text and the enciphering or deciphering key? A useful tool for this task could be the following javascript application for analyzing Vigenère ciphers:*

http://echidna.maths.usyd.edu.au/ kohel/tch/Crypto/vigenere.html

Consider those ciphertexts from previous exercises which come from a Vigenère cipher, and determine the periods and keys for each of the ciphertext samples.

Exercise 3.2 *For each of the cryptographic texts from the course web page, compute the coincidence index of the ciphertexts. Can you tell which come from simple substitution or transposition ciphers? How could you distinguish the two?*

Exercise 3.3 *For each of the cryptographic texts from the course web page, for various periods extract the substrings of im + j-th characters. For those which are not simple substitutions, can you identify a period?*

Exercise 3.4 *For each of the ciphertexts which you have reduced to simple substitutions, consider the frequency distribution of the simple substitution texts. Now recover the keys and original plaintext.*

Exercise 3.5 (Correlations of sequence translations) *Suppose that* `pt` *and* `ct` *are plaintext and ciphertext whose frequency distributions are to be compared. Assume we have defined:*

```
sage: S = AlphabeticStrings()
sage: E = SubstitutionCryptosystem(S)
```

The following code finds the correlations between the affine translations of two sequences.

```
sage: X = pt.frequency_distribution()
sage: Z = ct.frequency_distribution()
sage: Y = DiscreteRandomVariable(X,Z.function())
sage: for j in range(26):
...        K = S([ (j+k)%26 for k in range(26) ])
...        print "%s: %s" % (j, X.translation_correlation(Y,E(K)))
```

What does `frequency_distribution` *return, and what are the ciphers e constructed in the* `for` *loop? What does* `translation_correlation` *return? Note that* Y *must be created as a discrete random variable on the probability space* X *in order to compute their correlations.*

Exercise 3.6 (Breaking Vigenère ciphers) *A Vigenère cipher is reduced to an translation cipher by the process of decimation. How does the above exercise solve the problem of finding the affine translation?*

Apply this exercise to the Vigenére ciphertext sample `cipher01.txt` *from the course web page, and the break the enciphering. Recall that you will have to use the decimation (by* `ct[i::m]`*) and* `coincidence_index` *to first reduce a Vigenère ciphertext to the output of a monoalphabetic cipher.*

```
sage: X = pt.frequency_distribution()
sage: m = 11
sage: r = 0.75
sage: match = [ [] for i in range(m) ]
sage: for i in range(m):
...      Z = ct[i::m].frequency_distribution()
...      Y = DiscreteRandomVariable(X,Z.function())
...      for j in range(26):
...          K = S([ (j+k)%26 for i in range(26) ])
...          corr = X.translation_correlation(Y,E(K))
...          if corr > r:
...              match[i].append(j)
```

Exercise 3.7 (Breaking substitution ciphers) *Suppose that rather than an affine translation, you have reduced to an arbitrary simple substitution. We need to undo an arbitrary permutation of the alphabet. For this purpose we define maps into Euclidean space:*

1. $\mathcal{A} \rightarrow \mathcal{A}^2 \rightarrow \mathbf{R}^2$ *defined by*

$$x \longmapsto xx \longmapsto \big(P(x), P(xx)\big).$$

2. $\mathcal{A} \rightarrow \mathcal{A}^2 \rightarrow \mathbf{R}^3$ *defined by*

$$x \longmapsto xy \longmapsto \big(P(x), P(xy \,|\, y), P(yx \,|\, y)\big),$$

for some fixed character y.

See the document

http://echidna.maths.usyd.edu.au/ kohel/tch/Crypto/digraph_frequencies.pdf

for standard vectors for the English language.

Exercise 3.8 (Breaking transposition ciphers) *In order to break transposition ciphers it is necessary to find the period m, of the cipher, and then to identify positions i and j within each block $1 + km \leq i, j \leq (k+1)m$ which were adjacent prior to the permutation of positions. Suppose we guess that m is the correct period. Then for a ciphertext sample $C = c_1 c_2 \ldots$, and a choice of $1 \leq i < j \leq m$, we can form the digraph decimation sequence $c_i c_j, c_{i+m} c_{j+m}, c_{i+2m} c_{j+2m}, \ldots$.*

Two statistical measures that we can use on ciphertext to determine if a digraph sequence is typical of the English language are a digraph coincidence index

$$\sum_{x \in \mathcal{A}} \sum_{y \in \mathcal{A}} \frac{n_{xy}(n_{xy} - 1)}{N(N-1)}$$

where N is the total number of character pairs, and n_{xy} is the number of occurrences of the pair xy, and the coincidence discriminant:

$$\sum_{x \in \mathcal{A}} \sum_{y \in \mathcal{A}} \left(\frac{n_{xy}}{N} - \left(\sum_{z \in \mathcal{A}} \frac{n_{xz}}{N} \right) \left(\sum_{z \in \mathcal{A}} \frac{n_{zy}}{N} \right) \right)^2.$$

The first term is the frequency of xy, and the latter is the product over the frequencies of x as a first character and y as a second character. The coincidence discriminant measures the discrepancy between the probability space of pairs xy and the product probability space.

What behavior do you expect for the coincidence index and coincidence discriminant of the above digraph decimation, if i and j were the positions of originally adjacent characters? Test your hypotheses with decimations of "real" English text, using the SAGE implementations of `coincidence_index` *and* `coincidence_discriminant`.

Why can we assume that $i < j$ in the digraph sequence? What is the obstacle to extending these statistical measures from two to more characters?

Information Theory

Information theory concerns the measure of information contained in data. The security of a cryptosystem is determined by the relative content of the key, plaintext, and ciphertext.

For our purposes a *discrete probability space* – a finite set X together with a probability function on X – will model a language. Such probability spaces may represent the space of keys, of plaintext, or of ciphertext, and which we may refer to as a space or a language, and to an element x as a *message*. The probability function $P : \mathrm{X} \to \mathbf{R}$ is defined to be a non-negative real-valued function on X such that

$$\sum_{x \in \mathrm{X}} P(x) = 1.$$

For a naturally occuring language, we reduce to a finite model of it by considering finite sets consisting of strings of length N in that language. If X models the English language, then the function P assigns to each string the probability of its appearance, among all strings of length N, in the English language.

4.1 Entropy

The *entropy* of a given space with probability function P is a measure of the information content of the language. The formal definition of entropy is

$$H(\mathrm{X}) = \sum_{x \in \mathrm{X}} P(x) \log_2(P(x)^{-1}).$$

For $0 < P(x) < 1$, the value $\log_2(P(x)^{-1})$ is a positive real number, and we define $P(x) \log_2(P(x)^{-1}) = 0$ when $P(x) = 0$. The following exercise justifies this definition.

Exercise. Show that the limit

$$\lim_{x \to 0^+} x \log_2(x^{-1}) = 0.$$

What is the maximum value of $x \log_2(x)$ and at what value of x does it occur?

An *optimal encoding* for a probability space X is an injective map from X to strings over some alphabet, such that the expected string length of encoded messages is minimized. The term $\log_2(P(x)^{-1})$ is the expected bit-length for the encoding of the message x in an optimal encoding, if one exists, and the entropy is the expected number of bits in a random message in the space.

As an example, English text files written in 8-bit ASCII can typically be compressed to 40% of the original size without loss of information, since the structure of the language itself encodes the remaining information. The human genome encodes data for producing sequences of 20 different amino acid, each with a triple of letters in the alphabet $\{A, T, C, G\}$. The 64 possibile "words" (codons in genetics) includes more than 3-fold redundancy, in specifying one of these 20 amino acids. Moreover, huge sections of the genome are repeats such as $AAAAAA\ldots$, whose information can be captured by an expression like A^n. More accurate models for the languages specified by English or by human DNA sequences would permit greater compression rates for messages in these languages.

Example 4.1 *Let* X *be the probability space* $\{A, B, C\}$ *of three elements, and assume that* $P(A) = 1/2$, $P(B) = 1/4$, *and* $P(C) = 1/4$. *The entropy of the space is then*

$$P(A) \log_2(2) + P(B) \log_2(4) + P(C) \log_2(4) = 1.5.$$

An optimal encoding is attained by the encoding of A *with* 0, B *with* 10, *and* C *with* 11. *With this encoding one expects to use an average of 1.5 bits to transmit a message in this encoding.*

The following example gives methods by which we might construct models for the English language.

Example 4.2 (Empirical models for English) *First, choose a standard encoding — this might be an encoding as strings in the set* $\{A, \ldots, Z\}$ *or as strings in the ASCII alphabet. Next, choose a sample text. The text might be the complete works of Shakespeare, the short story* Black cat *of Edgar Allan Poe, or the U.S. East Coast version of the* New York Times *from 1 January 2000 to 31 January 2000. The following are finite probability spaces for English, based on these choices:*

1. *Let* X *be the set of characters of the encoding and set* $P(c)$ *to be the probability that the character c occurs in the sample text.*

2. *Let* X *be the set of character pairs over the encoding alphabet and set* $P(x)$ *to be the probability that the pair $x = c_1 c_2$ occurs in the sample text.*

3. *Let* X *be the set of words in the sample text, and set* $P(x)$ *to be the probability that the word x occurs in the sample text.*

For each of these we can extend our model for the English language to strings of length n.

How well do you think each of these model the English language?

4.2   Rate and Redundancy

Let X be a discrete probability space. We define the *rate* of X to be

$$r(\mathrm{X}) = \frac{H(\mathrm{X})}{\log_2(|\mathrm{X}|)},$$

and the *redundancy* to be $1 - r(\mathrm{X})$. The redundancy in a language derives from the structures such as character frequency distributions, digram frequency distributions (the probabilities of ordered, adjacent character pairs), and more generally n-gram frequency distributions. Global structures of a natural language such as vocabulary and grammar rules determine yet more structure, adding to the redundancy of the language.

4.3   Conditional Probability

We would now like to have a concept of conditional probability for cryptosystems. Let E be a cryptosystem, $\mathcal{M}$ a plaintext space, $\mathcal{K}$ a key space, and $\mathcal{C}$ a ciphertext space. For a symmetric key system the space of plaintext and ciphertext coincide, but the probability distributions on them may differ in the context of the cryptosystem.

We use P for both the probability function on the plaintext space $\mathcal{M}$ and on $\mathcal{K}$. We can now define a probability function on $\mathcal{C}$ relative to the cryptosystem E:

$$P(y) = \sum_{K \in \mathcal{K}} P(K) \sum_{\substack{x \in \mathcal{M} \\ E_K(x)=y}} P(x).$$

We can now define $P(x, y)$, for $x \in \mathcal{M}$ and $y \in \mathcal{C}$ to be the probability that the pair (x, y) appears as a plaintext–ciphertext pair. Assuming the independence of plaintext and key spaces, we can define this probability as:

$$P(x, y) = \sum_{\substack{K \in \mathcal{K} \\ E_K(x)=y}} P(K)P(x).$$

x and y are said to be *independent* if $P(x, y) = P(x)P(y)$. For ciphertext y and plaintext x, define the conditional probability $P(y|x)$ by

$$P(y|x) = \begin{cases} \dfrac{P(x, y)}{P(x)} & \text{if } P(x) \neq 0 \\[2ex] 0 & \text{if } P(x) = 0 \end{cases}$$

4.4 Conditional Entropy

We can now define the conditional entropy $H(\mathcal{M}|y)$ of the plaintext space with respect to a given ciphertext $y \in \mathcal{C}$.

$$H(\mathcal{M}|y) = \sum_{x \in \mathcal{M}} P(x|y) \log_2(P(x|y)^{-1})$$

The conditional entropy $H(\mathcal{M}|\mathcal{C})$ of a cryptosystem (more precisely, of the plaintext with respect to the ciphertext) as an expectation of the individual conditional entropies:

$$H(\mathcal{M}|\mathcal{C}) = \sum_{y \in \mathcal{C}} P(y) H(\mathcal{M}|y)$$

This is sometimes referred to as the *equivocation* of the plaintext space $\mathcal{M}$ with respect to the ciphertext space $\mathcal{C}$.

4.5 Perfect secrecy and one-time pads

Perfect Secrecy. A cryptosystem is said to have *perfect secrecy* if the entropy $H(\mathcal{M})$ equals the conditional entropy $H(\mathcal{M}|\mathcal{C})$.

Let $K = k_1 k_2 \ldots$ be a key stream of random bits, and let $M = m_1 m_2 \ldots$ be the plaintext bits. We define a ciphertext $C = c_1 c_2 \ldots$ by

$$c_i = m_i \oplus k_i,$$

where $\oplus$ is the addition operation on bits in $\mathbf{Z}/2\mathbf{Z}$. In the language of computer science, this is the `xor` operator:
$$\begin{aligned} 0 \oplus 0 &= 0, & 1 \oplus 0 &= 1, \\ 0 \oplus 1 &= 1, & 1 \oplus 1 &= 0. \end{aligned}$$

In general such a cryptosystem is called the *Vernam cipher*. If the keystream bits are generated independently and randomly, then this cipher is called a *one-time pad*.

Note that neither the Vernam cipher nor the one-time pad has to be defined with respect to a binary alphabet. The bit operation `xor` can be replaced by addition in $\mathbf{Z}/n\mathbf{Z}$, where n is the alphabet size, using any bijection of the alphabet with the set $\{0, \ldots, n-1\}$.

Perfect secrecy of one-time pads

Recall that $P(x|y)$ is defined to be $P(x|y) = P(x,y)/P(y)$ if $P(y) \neq 0$ and is zero otherwise. If $\mathcal{M}$ is the plaintext space and $\mathcal{C}$ the ciphertext space (with probability function defined

in terms of the cryptosystem), then the condidtional entropy $H(\mathcal{M}|\mathcal{C})$ is defined to be:

$$
\begin{aligned}
H(\mathcal{M}|\mathcal{C}) &= \sum_{y \in \mathcal{C}} P(y) H(\mathcal{M}|y) \\
&= \sum_{y \in \mathcal{C}} P(y) \sum_{x \in \mathcal{M}} P(x|y) \log_2(P(x|y)^{-1}) \\
&= \sum_{y \in \mathcal{C}} \sum_{x \in \mathcal{M}} P(x,y) \log_2(P(x|y)^{-1}).
\end{aligned}
$$

If for each $x \in \mathcal{M}$ and $y \in \mathcal{C}$ the joint probability $P(x,y)$ is equal to $P(x)P(y)$ (i.e. the plaintext and ciphertext space are independent) and thus $P(x|y) = P(x)$, then the above expression simplifies to:

$$
\begin{aligned}
H(\mathcal{M}|\mathcal{C}) &= \sum_{x \in \mathcal{M}} \sum_{y \in \mathcal{C}} P(x)P(y) \log_2(P(x)^{-1}) \\
&= \left(\sum_{y \in \mathcal{C}} P(y) \right) \sum_{x \in \mathcal{M}} P(x) \log_2(P(x)^{-1}) \\
&= \sum_{x \in \mathcal{M}} P(x) \log_2(P(x)^{-1}) = H(\mathcal{M}).
\end{aligned}
$$

Therefore the cryptosystem has perfect secrecy.

Entropy of the key space

It can be shown that perfect secrecy (or unconditional security) requires the entropy $H(\mathcal{K})$ of the key space $\mathcal{K}$ to be at least as large as the entropy $H(\mathcal{M})$ of the plaintext space $\mathcal{M}$. If the key space is defined to be the set of N-bit strings with uniform distribution, then the entropy of $\mathcal{K}$ is N, and this is the maximum entropy for a space of N-bit strings (see exercise). This implies that in order to achieve perfect secrecy, the number of bits of strings in the key space should be at least equal the entropy $H(\mathcal{M})$ of the plaintext space.

Exercises

In order to understand naturally occurring languages, we consider the models for finite languages X consisting of strings of fixed finite length N together with a probability function P which models the natural language. In what follows, for two strings x and y we denote their concatenation by xy.

Exercise 4.1 *Show that N is the maximum entropy for a probability function on bit strings of length N.*

Exercise 4.2 *Show that the rate of a uniform probability space is* 1 *and that this is the maximal value for any probability space.*

Exercise 4.3 *For a given cryptosystem, show that the definition*

$$P(y) = \sum_{K \in \mathcal{K}} P(K) \sum_{\substack{x \in \mathcal{M} \\ E_K(x)=y}} P(x).$$

determines a probability function on the ciphertext space. Then verify the equalities:

$$P(y) = \sum_{x \in \mathcal{M}} P(x,y), \quad and \quad P(x) = \sum_{y \in \mathcal{C}} P(x,y).$$

Exercise 4.4 *Consider the language of* 1*-character strings over* $\{A, B, C, D\}$ *with associated probabilities* $1/3$*,* $1/12$*,* $1/4$*, and* $1/3$*. What is its corresponding entropy?*

Exercise 4.5 *Consider the language* X^2 *of all strings of length* 2 *in* $\{A, B, C, D\}$ *defined by the probability function of Exercise 1 and 2-character independence:* $P(xy) = P(x)P(y)$*. What is the entropy of this language?*

Exercise 4.6 *Let* $\mathcal{M}$ *be the strings of length* 2 *over* $\{A, B, C, D\}$ *with the following frequency distribution:*

$$
\begin{array}{llll}
P(\mathtt{AA}) = 5/36 & P(\mathtt{BA}) = 0 & P(\mathtt{CA}) = 1/12 & P(\mathtt{DA}) = 1/9 \\
P(\mathtt{AB}) = 1/36 & P(\mathtt{BB}) = 1/144 & P(\mathtt{CB}) = 1/48 & P(\mathtt{DB}) = 1/36 \\
P(\mathtt{AC}) = 7/72 & P(\mathtt{BC}) = 1/48 & P(\mathtt{CC}) = 1/16 & P(\mathtt{DC}) = 5/72 \\
P(\mathtt{AD}) = 5/72 & P(\mathtt{BD}) = 1/18 & P(\mathtt{CD}) = 1/12 & P(\mathtt{DD}) = 1/8
\end{array}
$$

Show that the 1-character frequencies in this language are the same as for the language in Exercise 2.

Exercise 4.7 *Do you expect the entropy of the language of Exercise 3 to be greater or less than that of Exercise 2? What is the entropy of each language?*

Exercise 4.8 *Consider the infinite language of all strings over the alphabet* $\{A\}$*, with probability function defined such that* $P(\mathtt{A} \ldots \mathtt{A}) = 1/2^n$*, where* n *is the length of the string* $\mathtt{A} \ldots \mathtt{A}$*. Show that the entropy of this language is* 2*.*

Block Ciphers

Data Encryption Standard

The Data Encryption Standard, or DES, is one of the most important examples of a Feistel cryptosystem. DES was the result of a contest set by the U.S. National Bureau of Standards (now called the NIST) in 1973, and adopted as a standard for unclassified applications in 1977.

The winning standard was developed at IBM, as a modification of the previous system called LUCIFER. The DES is widely used for encryption of PIN numbers, bank transactions, and the like. DES is also specified as an Australian banking standard.

The DES is an example of a Feistel cipher, which operates on blocks of 64 bits at a time, with an input key of 64 bits. Every 8th bit in the input key is a parity check bit which means that in fact the key size is effectively reduced to 56 bits.

Advanced Encryption Standard

In 1997, the NIST called for submissions for a new standard to replace the aging DES. The contest terminated in November 2000 with the selection of the Rijndael cryptosystem as the Advanced Encryption Standard (AES).

5.1   Product ciphers and Feistel ciphers

As a precursor to the description of DES, we make the following definitions, which describe various aspects of the constructions, specific properties, and design components of DES.

A *product cipher* is a composite of two or more elementary ciphers with the goal of producing a cipher which is more secure that any of the individual components. A *substitution-permutation network* is a product cipher composed of stages, each involving substitutions and permutations, in which the blocks can be partitioned into smaller blocks for substitutions and recombined with permutations. An *iterated block cipher* is a block cipher
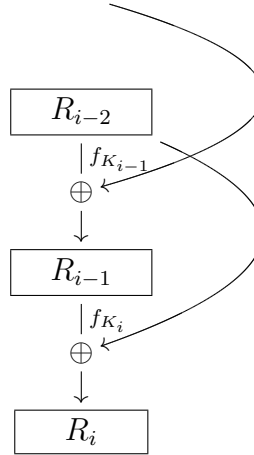
involving the repetition of an internal *round function*, which may involve a key as input. Each of the sequential steps is termed a *round*.

We now describe in more detail an example of an iterated block cipher, called a *Feistel cipher*. In a Feistel cipher the input block is of even length $2t$, of the form $L_0 R_0$, and outputs ciphertext of the form $R_r L_r$. For each i such that $1 \le i \le r$, the round map takes $L_{i-1} R_{i-1}$ to $L_i R_i$, where $L_i = R_{i-1}$ and $R_i = L_{i-1} \oplus f_{K_i}(R_{i-1})$, where f_{K_i} is a cipher which depends only on an input subkey K_i, which is derived from the cipher key K.

The flow of the Feistel cipher therefore looks something like:

$$
\begin{array}{cc}
\boxed{L_{i-1}} & \boxed{R_{i-1}} \\
 & \quad \downarrow f_{K_i} \\
 & \oplus \\
 & \downarrow \\
\boxed{L_i} & \boxed{R_i}
\end{array}
$$
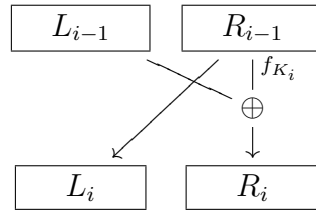
We can eliminate the L_i by defining $R_{-1} = L_0$, so that the input is $R_{-1} R_0$, and the round operations are of the form $R_i = R_{i-2} \oplus f_{K_i}(R_{i-1})$, in which case the flow diagram looks like:

$$
\begin{array}{c}
\boxed{R_{i-2}} \\
\downarrow f_{K_{i-1}} \\
\oplus \\
\downarrow \\
\boxed{R_{i-1}} \\
\downarrow f_{K_i} \\
\oplus \\
\downarrow \\
\boxed{R_i}
\end{array}
$$

The final output of the Feistel cipher is the inverted pair $R_r L_r = R_r R_{r-1}$, which allows the Feistel cipher to be inverted by running through the same algorithm with the key sequence reversed.

We now prove that reversing the internal key sequence gives the inverse cipher, by a comparison of the enciphering and deciphering sequences $\{R_i\}$ and $\{R'_j\}$.

Enciphering. A message $M = L_0 R_0 = R_{-1} R_0$, is enciphered via the iteration:

$$R_{i+1} = R_{i-1} \oplus f_{K_{i+1}}(R_i), \tag{5.1}$$

with respect to a key sequence $K_1, K_2, \ldots, K_r$.

Deciphering. Suppose we begin with $C = R_r R_{r-1} = R'_{-1} R'_0$, and a reversed key sequence $K'_1, K'_2 \ldots, K'_r = K_r, K_{r-1} \ldots, K_1$. The deciphering follows the same algorithm as enciphering with respect to this key sequence:

$$R'_{j+1} = R'_{j-1} \oplus f_{K'_{j+1}}(R'_j). \tag{5.2}$$

Setting $j = r - i - 1$, we have $K'_{j+1} = K'_{r-i} = K_{i+1}$. We moreover want to show the relations

$$R'_{-1} = R_r, \ R'_0 = R_{r-1}, \ldots, R'_{r-1} = R_0, \ R'_r = R_{-1}.$$

In other words, we want to show that $R'_j = R_i$ whenever $i + j = r - 1$.

Clearly this relation holds for $(i, j) = (r, -1)$ and $(i, j) = (r - 1, 0)$. Assuming it holds for $j - 1$ and j we prove that it holds for $j + 1$. The deciphering sequence (5.2) can be replaced by

$$R'_{j+1} = R'_{j-1} \oplus f_{K'_{j+1}}(R'_j) = R'_{r-i-2} \oplus f_{K'_{r-i}}(R'_{r-i-1}) = R_{i+1} \oplus f_{K_{i+1}}(R_i)$$

The expression $R_{i+1} = R_{i-1} \oplus f_{K_{i+1}}(R_i)$ in (5.2) can be rearranged by adding (= substracting) $f_{K_{i+1}}(R_i)$ to both sides to get $R_{i+1} \oplus f_{K_{i+1}}(R_i) = R_{i-1}$. We conclude that $R'_{j+1} = R_{i-1}$, so the equality holds by induction.

Example 5.1 (Feistel cipher) *Let f_{K_i} be the block cipher, of block length 4, which is the composition of the following maps:*

1. *The transposition cipher $T = [4, 2, 1, 3]$; followed by*

2. *A bit-sum with the 4-bit key K_i; followed by*

3. *A substitution cipher S applied to the 2-bit blocks*

$$S(00) = 10, \quad S(10) = 01, \quad S(01) = 11, \quad S(11) = 00,$$

i.e. $b_1 b_2 b_3 b_4 \mapsto S(b_1 b_2) S(b_3 b_4)$.

Let C be the 3-round Feistel cryptosystem of key length 12, where the three internal keys K_1, K_2, K_3 are the first, second, and third parts of the input key K, and the round function is f_{K_i}.

Exercise. *Compute the enciphering of the text $M = 11010100$, using the key $K = 001011110011$.*

5.2 Digital Encryption Standard Overview

The DES is a 16-round Feistel cipher, which is preceeded and followed by an initial permutation IP and its inverse IP^{-1}. That is, we start with a message M, and take $L_0 R_0 = IP(M)$ as input to the Feistel cipher, with output $IP^{-1}(R_{16} L_{16})$. The 64-bits of the key are used to generate 16 internal keys, each of 48 bits. The steps of the round function f_K is given by the following sequence, taking on 32-bit strings, expanding them to 48-bit strings, and applying a 48-bit block function.

1. Apply a fixed *expansion permutation E* — this function is a permutation the 32 bits with repetitions to generate a 48-bit block $E(R_i)$.

2. Compute the bit-sum of $E(R_i)$ with the 48-bit key K_i, and write this as 8 blocks $B_1, \ldots, B_8$ of 6 bits each.

3. Apply to each block $B_j = b_1 b_2 b_3 b_4 b_5 b_6$ a substitution S_j. These substitutions are specified by *S-boxes*, which describe the substitution as a look-up table. The output of the substitution cipher is a 4-bit string C_j, which results in the 32-bit string $C_1 C_2 C_3 C_4 C_5 C_6 C_7 C_8$.

4. Apply a fixed 32-bit permutation P to $C_1 C_2 C_3 C_4 C_5 C_6 C_7 C_8$, and output the result as $f_{K_i}(R)$.

This completes the description of the round function f_{K_i}.

5.3 Advanced Encryption Standard Overview

In 1997, the NIST called for submissions for a new standard to replace the aging DES. The contest terminated in November 2000 with the selection of the Rijndael cryptosystem as the Advanced Encryption Standard (AES).

The Rijndael cryptosystem operates on 128-bit blocks, arranged as 4×4 matrices with 8-bit entries. The algorithm consists of multiple iterations of a round cipher, each of which is the composition of the following four basic steps:

- *ByteSub* transformation. This step is a nonlinear substition, given by a S-box (look-up table), designed to resist linear and differential cryptanalysis.

- *ShiftRow* transformation. Provides a linear mixing for diffusion of plaintext bits.

- *MixColumn* transformation. Provides a similar mixing as in the ShiftRow step.

- *AddRoundKey* transformation. Bitwise `XOR` with the round key.

The Advanced Encryption Standard allows Rijndael with key lengths 128, 192, or 256 bits.

The eight-bit byte blocks which form the matrix entries are interpreted as elements of the finite field of $2^8 = 256$ elements. The finite field is represented by the quotient ring

$$\mathbf{F}_{2^8} = \mathbf{F}_2[X]/(X^8 + X^4 + X^3 + X + 1),$$

whose elements are polynomials $c_7 X^7 + c_6 X^6 + c_5 X^5 + c_4 X^4 + c_3 X^3 + c_2 X^2 + c_1 X + c_0$.

We denote by BS, SR, MC, and ARK these four basic steps. There exist corresponding inverse operations IBS, ISR, IMC, IARK. The flow of the algorithms for enciphering and deciphering are as follows:

1. ARK	1. ARK
2. BS, SR, MC, ARK	2. IBS, ISR, IMC, IARK
$\vdots$	$\vdots$
3. BS, SR, MC, ARK	3. IBS, ISR, IMC, IARK
4. BS, SR, ARK	4. IBS, ISR, ARK

ByteSub. The ByteSub operation is given by the S-box look-up table. Alternatively the S-box has a description in terms of the structure of the finite fields and linear algebra. Let x' be the inverse of x in $\mathbf{F}_{2^8}$ if $x \neq 0$ and set $x' = x = 0$ otherwise. Then the ByteSub step is given by $x \mapsto X^6 + X^5 + X + 1 + x'A$ where A is the matrix:

$$A = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{pmatrix}$$

5.4   Modes of Operation

Block ciphers can be applied to longer ciphertexts using one of various *modes of operation*. We assume that the input is plaintext $M = M_1 M_2 \ldots$, the block enciphering map for given key K is E_K, and the output is $C = C_1 C_2 \ldots$. The possible block cipher *modes of operation* which we treat are identified by the acronyms ECB, CBC, CFB, and OFB. In each case we assume that we have a cipher of block length n, with enciphering maps E_K and deciphering maps D_K for each key K.

5.4.1   Electronic Codebook Mode (ECB)

Electronic codebook mode is the most obvious way to use a block cipher.

Enciphering.

Input:
k-bit key K
n-bit plaintext blocks $M = M_1 M_2 \ldots M_t$.

Algorithm:
$$C_j = E_K(M_j).$$

Output:
n-bit ciphertext blocks $C = C_1 C_2 \ldots C_t$.

Deciphering.

Input:
k-bit key K
n-bit ciphertext blocks $C = C_1 C_2 \ldots C_t$.

Algorithm:
$$M_j = D_K(C_j).$$

Output:
n-bit plaintext blocks $M = M_1 M_2 \ldots M_t$.

To explain the name, one should think of this mode as being defined by a lookup table or *codebook*. Consider, for example, DES, which operates on 64 bit (binary) strings. These describe, for instance, 8 characters in 8-bit ASCII (or in 7-bit ASCII with one parity check bit). For each key K, the codebook contains the ciphertext image of each of these 8 character strings as a lookup table. In order to encipher the message, the electronic codebook is consulted for the ciphertext encoding of each block. Note that the number of such hypothetical codebooks is itself enormous – for DES there are 2^{56} possible keys, each with its own codebook.

We now consider some of the properties and limitations of ECB mode. The categories below are chosen for comparison with the modes of operations which follow.

Properties:
1. Identical plaintext. The same plaintext block always maps to the same ciphertext block.
2. Chaining dependencies. Reordering the plaintext blocks induces a reordering of the same ciphertext blocks.
3. Error propagation. An error in a ciphertext block results in a deciphering error only in the corresponding plaintext block.

Security Remarks:

1. Malicious substitution of a ciphertext block C_j results in substitution of message block M_j.

2. Blocks C_j do not hide patterns – the same block M_j is enciphered in the same way.

Conclusion. Although commonly used, electronic codebook mode is not recommended for use if $t > 1$ with the same key. Security can be improved by inclusion of random padding bits in each block.

5.4.2   Cipher Block Chaining Mode (CBC)

Cipher block chaining mode involves a vector bit sum operation of the message block with the previous ciphertext block prior to enciphering. The ciphertext blocks are initialized with a randomly chosen message which may be transmitted openly, i.e. the security of the cryptosystem is based on the secrecy of the key, not on the secrecy of initialization vector.

Enciphering.

Input:
k-bit key K
n-bit initialization vector C_0
n-bit plaintext blocks $M = M_1 M_2 \ldots M_t$.

Algorithm:
$$C_j = E_K(C_{j-1} \oplus M_j).$$

Output:
n-bit ciphertext blocks $C = C_0 C_1 \ldots C_t$.

Deciphering.

Input:
k-bit key K
n-bit ciphertext blocks $C = C_0 C_1 \ldots C_t$.

Algorithm:
$$M_j = C_{j-1} \oplus D_K(C_j).$$

Output:
n-bit plaintext blocks $M = M_1 M_2 \ldots M_t$.

Properties:
1. **Identical plaintext.** The same sequence of ciphertext blocks result when the same key and the same initialization vector are used.
2. **Chaining dependencies.** The chaining mechanism causes C_j to depend on C_{j-1} and M_j, so enciphering is not independent of reordering.
3. **Error propagation.** An error in a ciphertext block C_j affects decipherment of C_j and C_{j+1}. For a reasonable enciphering algorithm, a single bit error affects 50% of the bits in

the deciphered plaintext block M_j', while the bit error affects only that bit of M_{j+1}'.

3. Error recovery. The cryptosystem is said to be self-recovering, in the sense that while an error in C_j results in incorrectly deciphered plaintext M_j' and M_{j+1}', the ciphertext C_{j+2} correctly deciphers to $M_{j+2}' = M_{j+2}$.

5.4.3   Cipher Feedback Mode (CFB)

Cipher feedback mode allows one to process blocks of size $r < n$ at a time. The typical value for r is 1, while n may be of size 64, using DES.

Enciphering.

Input:
k-bit key K
n-bit initialization vector I_1
r-bit plaintext blocks $M = M_1 M_2 \ldots M_t$.

Algorithm:
$$C_j = M_j \oplus L_r(E_K(I_j)),$$
$$I_{j+1} = R_{n-r}(I_j) \, || \, C_j,$$

where L_r and R_{n-r} are the operators which take the left-most r-bits and the right-most $n - r$-bits, and $||$ is the concatenation operator.

The vector I_j should be thought of as a *shift register*, a block of n-bits of memory which stores some state of the algorithm. The formation of I_{j+1} is a left-shift by r of this block, discarding the left-most r bits, with the right-most r bits replaced by C_j.

Deciphering.

Input:
k-bit key K
n-bit initialization vector I_1
r-bit ciphertext blocks $C = C_1 C_2 \ldots C_t$.

Algorithm:
Compute $I_1, \ldots, I_t$ as in the enciphering algorithm, which can be generated independently of the deciphered message text, and then compute

$$M_j = C_j \oplus L_r(E_K(I_j)).$$

Note that CFB deciphering requires only the block cipher E_K, not the inverse block deciphering map D_K.

Properties:
1. Identical plaintext. The same sequence of ciphertext blocks results when the same key and initialization vector is used. Changing the initialization vector changes the ciphertext.
2.   Chaining dependencies. Ciphertext block C_j depends on the previous plaintext

blocks $M_{j-1}, \ldots, M_1$ as well as M_j, so the ciphertext blocks are not reordering independent.

3. Error propagation. An error in C_j affects the decipherment of the next $[n/r]$ plaintext blocks. The recovered plaintext M_j' will differ from M_j at exactly the bits for which C_j was in error. These bit errors will appear in subsequent blocks M_{j+k}' at translated positions.

4. Error recovery. Proper deciphering requires the shift register to be correct, for which the previous $[n/r]$ ciphertext blocks are required. The decipherment is self-recovering from errors, but only after $[n/r]$ blocks (approximately the same n-bits of the ciphertext block in error).

5. Throughput. The rate of enciphering and deciphering is reduced by a factor of n/r, that is, for every r bits of output the algorithm must carry out one n-bit enciphering operation.

5.4.4 Output Feedback Mode (OFB)

Output feedback mode has a similar use as cipher feedback mode, but is relevant to applications for which error propagation must be avoided. Output feedback mode is an example of a *synchronous stream cipher* (constructed from a block cipher), in which the keystream is created independently of the plaintext stream.

Enciphering.

Input:
k-bit key K
n-bit initialization vector I_0
r-bit plaintext blocks $M = M_1 M_2 \ldots M_t$.

Algorithm:
$$I_j = E_K(I_{j-1})$$
$$C_j = M_j \oplus L_r(I_j)$$

Deciphering.

Input:
k-bit key K
n-bit initialization vector I_0
r-bit ciphertext blocks $C = C_1 C_2 \ldots C_t$.

Algorithm:
Compute $I_1, \ldots, I_t$ as in the enciphering algorithm.

$$M_j = C_j \oplus L_r(I_j)$$

Properties:
1. Identical plaintext. The same comments for CBC and CFB apply.
2. Chaining dependencies. The ciphertext output is order dependent, but the keystream $I_1, I_2, \ldots$ is plaintext independent.

3. Error propagation. An error in a ciphertext bit affects only that bit of the plaintext.

4. Error recovery. The cipher is self-synchronizing, and bit errors in a ciphertext block affect only that bit of the recovered plaintext. It recovers immediately from bit errors, but bit losses affect alignment.

5. Throughput. As with CFB, the rate of enciphering and deciphering is reduced by a factor of n/r, however the vectors I_j can be precomputed from K and I_0, independently of the ciphertext blocks.

Exercises

We summarise the modes of operation covered in this chapter.

Electronic Codebook Mode. For a fixed key K, the output ciphertext is given by $C_j = E_K(M_j)$ with output $C_1 C_2 \ldots$.

Ciphertext Block Chaining Mode. For input key K, and initialization vector $IV = C_0$, the output ciphertext is given by $C_j = E_K(C_{j-1} \oplus M_j)$, with output $C_0 C_1 C_2 \ldots$.

Ciphertext Feedback Mode. Given plaintext $M_1 M_2 \ldots$ in r-bit blocks, a key K, an n-bit cipher E_K, and an n-bit initialization vector $IV = I_1$, the ciphertext is computed as:

$$C_j = M_j \oplus L_r(E_K(I_j))$$
$$I_{j+1} = R_{n-r}(I_j) \,\|\, C_j$$

where R_{n-r} and L_r are the operators which take the right-most $n-r$ bits and the left-most r bits, respectively, and $\|$ is concatenation.

Output Feedback Mode. Given plaintext $M_1 M_2 \ldots$ in r-bit blocks, a key K, an n-bit cipher E_K, and an n-bit initialization vector $IV = I_0$, the ciphertext is computed as:

$$I_j = E_K(I_{j-1})$$
$$C_j = M_j \oplus L_r(I_j),$$

where L_r is the operator which takes the left-most r bits.

Exercise 5.1 *What mode of operation has been used in the assignment and in class up to this point, and why? What are the security disadvantages of this mode of operation?*

Exercise 5.2 *Let E_K be the 4-bit cipher defined by:*

$$E_K(M) = (K \oplus M) \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix} = (X_1 + X_3, X_2 + X_4, X_2 + X_3, X_1 + X_4)$$

where $X = K \oplus M = (X_1, X_2, X_3, X_4)$. Encipher the message M given by

$$11010110111001110010010001001000,$$

using the key $K = 1011$, in (i) ECB mode, in (ii) CBC mode with initialization vector 1001, and in (iii) CFB mode with initialization vector 1001 and $r = 1$.

Exercise 5.3 *How many steps are required for error recovery from a ciphertext transmission error in ECB and CBC modes?*

Exercise 5.4 *If $n = 64$ and $r = 8$, how many steps in CFB mode does it take to recover from an error in a ciphertext block? What about in OFB mode?*

Stream Ciphers

A stream cipher enciphers individual characters, usually bits, of a plaintext message one at a time, with a cipher that varies with time. Block ciphers are *memoryless*, in the sense that the same function is used to encipher successive blocks. Stream ciphers, in contrast, must have memory. As such they are *state functions* because the current state S_i of the function is recorded in a memory buffer. We saw how various modes of operation (CFB, OFB) turn a memoryless block cipher into a state function by feedback buffer. A *keystream* is a sequence of characters generated from the key and the current state, as input to the stream cipher.

6.1   Types of Stream Ciphers

In a **synchronous stream ciphers** the keystream is generated independently of the plaintext message (or ciphertext). Given a key K, if the initial state is designated S_0, then, for each cycle $i = 0, 1, 2, \ldots$, the following equations describe the generation of the keystream, ciphertext, and next state:

$$\begin{aligned} \textit{Key stream function:} \quad & k_i = g_K(S_i) \\ \textit{Output function:} \quad & c_i = h(k_i, m_i) \\ \textit{Next state function:} \quad & S_{i+1} = f_K(S_i) \end{aligned}$$

An additive binary stream cipher is defined to be a synchronous stream cipher in which $h = $ `XOR`.

A **self-synchronizing** or **asynchronous** stream cipher is a stream cipher in which the keystream is a function of the key and a fixed number of previous ciphertext characters. Given a key K and initial state $S_0 = (c_{-t}, \ldots, c_{-1})$, the keystream and ciphertext are generated for each cycle $i = 0, 1, 2, \ldots$ as for a synchronous stream cipher:

$$\begin{aligned} \textit{Key stream function:} \quad & k_i = g_K(S_i) \\ \textit{Output function:} \quad & c_i = h(k_i, m_i) \end{aligned}$$

with the next state set to $S_{i+1} = (c_{i+1-t}, \ldots, c_i)$.

## 6.2 Properties of Stream Ciphers

Synchronous stream ciphers

- Synchronization: Sender and receiver are required to be synchronized in terms of both state and key.

- Error propagation: None — a bit error in the ciphertext affects precisely one bit in the deciphered plaintext, provided that synchronization is maintained.

- Attacks and features: Property (1) means that an active adversary can use insertion, deletion, or replay of ciphertext; property (2) implies that the affects of these changes effect direct changes on the deciphered plaintext, which might be exploited.

Asynchronous stream ciphers

- Synchronization: An insertion, deletion, or change in ciphertext characters results in loss of only a fixed number of deciphered plaintext characters, after which the deciphering self-synchronizes.

- Error propagation: A ciphertext error in transmission affects at most t characters of the deciphered plaintext.

- Attacks and features: The error propagation makes active modification more easily detected, while self-synchronization makes insertion, deletion, or reply of ciphertext blocks more difficult to detect. Since each plaintext character influences subsequent ciphertext, an asynchronous stream cipher is better at masking plaintext structure or redundancies.

6.3 Linear Feedback Shift Registers

A linear feedback shift register implements a keystream function, and which can be simply described by a schematic diagram of the following form:
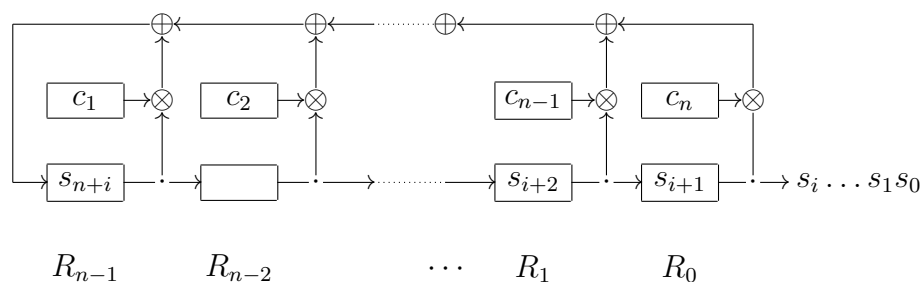
Before discussing the mathematical definition of linear feedback shift registers (LFSR's), we address the question "Why?". A LFSR is essentially an elementary algorithm for generating a keystream, which has the following desirable properties:

1. Easy to implement in hardware.

2. Produce sequences of long period.

3. Produce sequences with good statistical properties.

4. Can be readily analyzed using algebraic techniques.

A LFSR is defined by n *stages*, labelled $R_{n-1}, \ldots, R_1, R_0$, each storing one bit, and having one input and output, and a timer which mark clock cycles $i = 0, 1, 2, \ldots$. At the i-th clock cycle:

1. The contents of stage 0 is output;

2. The contents of R_i moves to R_{i-1}, for $1 \leq i \leq n - 1$; and

3. Stage R_{n-1} is the bit sum of a prescribed subset of stages $0, 1, \ldots, n - 2$.

We may denote the contents of stage R_j at time i by s_{i+j}, and the algorithm for updating the contents of stage R_{n-1} gives a recurrence relation

$$s_{n+i} = \sum_{j=0}^{n-1} c_{n-j} s_{i+j},$$

where c_j, $1 \leq j \leq n$ are fixed bit constants specifying the stages which contribute to the bit sum. By setting $c_0 = 1$ we can express the relation as $\sum_{j=0}^{n} c_{n-j} s_{i+j} = 0$.

We identify the constants c_k with coefficients of a polynomial

$$g(x) = \sum_{k=0}^{n} c_k x^k,$$

which we call the *connection polynomial* of the LFSR. Moreover, if can take the LFSR output bits s_j as the coefficients of a power series

$$s(x) = \sum_{j=0}^{\infty} s_j x^j,$$

then the recurrence relation expresses the fact that $s(x)g(x)$ is a polynomial $f(x)$ of degree less than n. In other words, the power series takes the form $s(x) = f(x)/g(x)$.

Exercise. Verify that the equality $f(x) = s(x)g(x)$, for $f(x)$ a polynomial of degree less than n, gives rise to the the stated recurrence for the coefficients of $s(x)$.

The LFSR is said to be *nonsingular* if $c_n \neq 0$. The condition that $c_n = \cdots = c_{n-k} = 0$ gives rise to a LFSR in which the feedback reduces to at most $n - k - 1$ terms, hence, after the initial k bits are output, the LFSR sequence reduces to one which can be modelled by a LFSR of length $n - k - 1$. For this reason we hereafter make the assumption that the LFSR is nonsingular.

We note that since the next state of the shift register (i.e. the contents of the collection of stages) depends only on the current contents, and there are 2^n possible states, the output sequence is eventually periodic. Since the all zero initial state maps to itself, it is clear that the maximal period for any LFSR of length n is $2^n - 1$. The connection polynomial is said to be *primitive* if the period of the LFSR output sequence, beginning at any nonzero state, is $2^n - 1$.

We note that the output sequence has period N if and only if $(X^N + 1)s(x)$ is a polynomial of degree at most $N - 1$. On the other hand, since $s(x) = f(x)/g(x)$, if $f(x)$ and $g(x)$ have no common factor, then it follows by the unique factorization of polynomials that $g(x)$ divides $X^N + 1$. In particular, if $g(x)$ is *irreducible*, since $\deg(f(x)) < \deg(g(x))$, it follows that $f(x)$ and $g(x)$ have no common factors. In summary, an irreducible connection polynomial of a LFSR must divide $x^N + 1$ where N is the period of any nonzero output sequence.

The theorem below shows that in fact every polynomial $g(x)$ in $\mathbf{F}_2[x]$ with nonzero constant term must divides $X^N + 1$ for some N. The special feature of irreducible connection polynomials, and especially primitive polynomials, is that we will be able to compute the value of N and, for primitive polynomials, that it is takes the the maximal possible value.

Lemma 6.1 *If $g(x)$ is not divisible by x, then there exists a polynomial $u(x)$ such that $x\,u(x) \bmod g(x) = 1$.*

Proof. Since the constant term of $g(x)$ is 1, the polynomial $u(x) = (g(x) + 1)/x$ satisfies $x\,u(x) = g(x) + 1$, from which the lemma follows. $\square$

Theorem 6.2 *Every polynomial $g(x)$ in $\mathbf{F}_2[x]$ coprime to x divides $x^N + 1$ for some N.*

Proof. Consider the sequence of remainders $\bmod\ g(x)$:

$$1 \bmod g(x), \ x \bmod g(x), \ x^2 \bmod g(x), \ x^3 \bmod g(x), \ldots$$

Since every remainder is a unique polynomial of degree at most $n - 1$, there are at most 2^n distinct elements in this sequence. It follows that there is some N such that $x^i \bmod g(x)$ equals $x^{N+i} \bmod g(x)$ for all sufficiently large i. Since $g(x)$ is not divisible by x, it follows from the previous lemma that we can cancel the powers of x^i to obtain $x^N \bmod g(x) = 1$. We conclude that $x^N + 1$ is divisible by $g(x)$. $\square$

Periods of LFSR's

We begin with some theorems regarding LFSR's and their connection polynomials. First, we make or recall some standard definitions. We define a polynomial $g(x)$ to be *irreducible* if the only factorization $g(x) = h(x)k(x)$ is with $h(x)$ or $k(x)$ equal to the constant polynomial. We define the order of x modulo $g(x)$ to be the smallest power x^N of x such that $x^N \bmod g(x)$ equals 1. A polynomial $g(x)$ is of degree n is said to be *primitive* if the order of x modulo $g(x)$ is $2^n - 1$. The next theorem shows that the definition of primitive given in the previous lecture agrees with the current one.

Theorem 6.3 *The period of a sequence generated by a LFSR is independent of the nonzero initial state if the connection polynomial is irreducible, and the period takes the maximal value $2^n - 1$ if and only if the connection polynomial is primitive.*

Since there are exactly $2^n - 1$ possible nonzero states, it is clear that a LFSR that produces an output sequence with this period in fact cycles through all such states, so the period is independent of the initial state. As a consequence of the theorem, a primitive polynomial is irreducible. We now prove the theorem.

Proof. We first note that all possible $2^n - 1$ output sequences are given by the rational expressions $s(x) = f(x)/g(x)$, where $f(x)$ runs through the all nonzero polynomials of degree less than the connection polynomial $g(x)$. If the $g(x)$ is irreducible, then this expression is minimal — $f(x)$ and $g(x)$ have no common factors, so there is no cancellation.

Next we note that the minimal period N of any power series $s(x)$ is the degree of the smallest $x^N + 1$ for which $(x^N + 1)s(x)$ is a polynomial. If also $s(x) = f(x)/g(x)$ is in minimal form, then for any such $x^N + 1$, the denominator $g(x)$ divides $x^N + 1$. If the connection polynomial $g(x)$ is irreducible, then the denominator equals $g(x)$ independently of $s(x)$ and the initial state which defines it, and so also is the period constant.

Finally the last statement follows by noting that $g(x)$ divides $x^N + 1$ if and only if $x^N \bmod g(x) = 1$. $\square$

For cryptographic purposes, it is desirable to have sequences which have very long period. The advantage of LFSR's in this respect is that the period grows exponentially in the length of the shift register. For small value of n, 2, 3, or 4, the value of the maximal possible period, $N = 2^n - 1$, is still trivially small. But as the table below shows, with modest values of n we are able to efficiently generate sequences with enormous period.

n	$2^n - 1$	n	$2^n - 1$	n	$2^n - 1$
1	1	11	2047	21	2097151
2	3	12	4095	22	4194303
3	7	13	8191	23	8388607
4	15	14	16383	24	16777215
5	31	15	32767	25	33554431
6	63	16	65535	26	67108863
7	127	17	131071	27	134217727
8	255	18	262143	28	268435455
9	511	19	524287	29	536870911
10	1023	20	1048575	30	1073741823

As particular examples, the primitive trinomials such as $x^{23}+x^5+1$, $x^{29}+x^2+1$, $x^{31}+x^3+1$, and $x^{41}+x^3+1$ define very efficiently computable recurrence relations for maximal length LFSR's.

6.4   Linear Complexity

In addition to practical applications for generating pseudo-random sequences, LFSR's are a useful theoretical tool for the characterization of other binary sequences. The *linear complexity* of a binary sequence is a measure of the structure of the sequence – a low linear complexity implies a cryptographically weak sequence.

An infinite sequence $s = s_0, s_1, \ldots$ is said to be generated by a LFSR if it is the output sequence of the shift register for some initial state. The linear complexity $L(s)$ for an infinite sequence s is defined to be 0 if s is the all zero sequence, equal to the minimum length of a LFSR which generates it if s is periodic, and equal to ∞ otherwise. The linear complexity $L(s)$ of a finite sequence $s = s_0, s_1, \ldots, s_{n-1}$ is defined to be the minimum length of a shift register which generates some sequence with initial segment s. The *linear complexity profile* of an infinite sequence s is the sequence $L_1(s), L_2(s), \ldots$, where $L_i(s)$ is the linear complexity of first i terms of s.
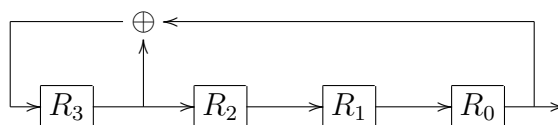
Exercises

Exercise 6.1 *Identify each of the key stream, output, and next state functions for the synchronous stream cipher determined by a block cipher in OFB mode of operation.*

Exercise 6.2 *Identify the the key stream and output functions for the asynchronizing stream cipher determined by a block cipher in each of CBC and 1-bit CFB modes.*

Linear feedback shift registers (LFSR's) are an efficient way of describing and generating certain sequences in hardware implementations. We derive and work with equivalent mathematical descriptions of the sequences produced by a LFSR, along with some generalized sequences which do not arise in this way.

A linear feedback shift register is composed of a shift register R which contains a sequence of bits and a feedback function f which is the bit sum (`xor`) of a subset of the entries of the shift register. The shift register contains n memory cells, or stages, labelled $R_{n-1}, \dots, R_1, R_0$, each holding one bit. Each time a bit is needed the entry in stage R_0 is output while the entry in cell R_i is passed to cell R_{i-1} and the top stage R_{n-1} is updated with the value $f(R)$.

Exercise 6.3 *Consider the following schematic of a linear feedback shift register:*

Let the initial entries of stages R_i be s_i, for $0 \le i \le n$. For each of the following initial entries below:

	s_3	s_2	s_1	s_0
a)	0	1	1	0
b)	1	1	1	0
c)	1	0	1	0
d)	1	1	0	0

compute the first 16 bits in the output sequence. Show that the output sequence is defined by the initial entries and the recursion $s_{i+4} = s_{i+3} + s_i$.

Exercise 6.4 *Show that every linear feedback register defines and is defined by a recursion of the form $s_{i+n} = \sum_{j=0}^{n-1} c_j s_{i+j}$, where the c_j are bits in $\mathbf{Z}/2\mathbf{Z}$; the products $c_j s_{i+j}$ and the summation are operations in $\mathbf{Z}/2\mathbf{Z}$.*

N.B. The ring $\mathbf{Z}/2\mathbf{Z}$ is also referred to as $\mathbf{F}_2$, the unique finite field of two elements. Note that the addition operation is the same `xor` that we have been using and the multiplication operation is the logical `and` operation.)

Exercise 6.5 *For a linear feedback register of length n, define a power series*

$$s(x) = \sum_{i=1}^{\infty} s_i x^i$$

from the output sequence s_i. Suppose that the linear feedback register defines the recursion $s_{i+n} = \sum_{j=0}^{n-1} c_{n-j} s_{i+j}$. Define a polynomial $g(x) = 1 + \sum_{j=1}^{n-1} c_j x^j$. Show that $f(x) = g(x)s(x)$ is a polynomial, that is, all of its coefficients are eventually zero. What is the polynomial $f(x)$?

Exercise 6.6 *In the previous exercise we showed that the power series $s(x)$ has the form $f(x)/g(x)$ in the power series ring $\mathbf{F}_2[[x]]$. In* SAGE *it is possible to form power series rings in the following way*

```
sage: F2 = FiniteField(2)
sage: PS.<x> = PowerSeriesRing(F2)
sage: f = x^2 + x
sage: g = x^3 + x + 1
sage: f/g + O(x^16)
x + x^4 + x^5 + x^6 + x^8 + x^11 + x^12 + x^13 + x^15 + O(x^16)
```

Consider the linear feedback shift register at the beginning of the worksheet. Construct the corresponding power series and verify that these are the same of the output sequences that you computed.

Statistical Properties. The output of a linear feedback shift operator of length n has a period, which must divide $2^n - 1$. The period is independent of the initial state, provided it is non-zero. If the period equals $2^n - 1$, the output sequence is said to be an m-sequence. The following theorem describes the statistical properties of m-sequences.

Theorem 6.4 *Let $s = s_0 s_1 \ldots$ be an m-sequence, and let k be an integer with $1 \le k \le n$. Then in each subsequence of s of length $2^n + k - 2$, every finite nonzero binary sequence of length k appears as a subsequence exactly 2^{n-k} times, and the length k zero subsequence appears exactly $2^{n-k} - 1$ times.*

A polynomial $g(x)$ in $\mathbf{F}_2[x]$ is *irreducible* if it is not the product of two polynomials of degree greater than zero. An irreducible polynomial $g(x)$ of degree n is *primitive* if $g(x)$ divides $x^N - 1$ for $N = 2^n - 1$ and no smaller value of N. (Equivalently, $g(x)$ is primitive if the powers x^i are distinct modulo $g(x)$, for i with $1 \le i \le 2^n - 1$.)

Linear Complexity. A linear feedback shift register is said to generate a binary sequence s if there exists some initial state for which its output sequence is s. The linear complexity $L(s)$ of an infinite sequence s is defined to be zero if s is the zero sequence, infinity if s is generated by no linear feedback shift register, and otherwise equal to the minimal length of a linear feedback shift register generating s. The linear complexity $L(s)$ of a finite sequence s is defined to be the minimal length of a linear feedback shift register with initial sequence s for some initial state.

Linear Complexity Profile. For a sequence s, define $L_j(s)$ to be the linear complexity of the first j terms of the sequence. The linear complexity profile of an infinite sequence s is defined to be the infinite sequence $L_1(s), L_2(s), \ldots$, and for a finite sequence $s = s_0 s_1 \ldots s_{N-1}$ is defined to be the finite sequence $L_1(s), L_2(s), \ldots, L_n(s)$.

LFSR Cryptosystems We introduce new utilities for binary stream cryptosystems based on linear feedback shift registers. The functions `binary_encoding` and `binary_decoding` convert ASCII text into its bit sequence and back. In addition, the new binary cryptosystems are:

<div align="center">

LFSRCryptosystem
ShrinkingGeneratorCryptosystem

</div>

Unlike the encoding function `strip_encoding` we have used so far, the function `binary_encoding` is information-preserving, taking 8-bit ASCII input and returning the binary encoding string. The inverse function `binary_decoding` recovers the original text.

A linear feedback shift register cryptosystem is created in SAGE using the function `LFSRCryptosystem`, taking no arguments. A key is defined by means of a pair, consisting of the connection polynomial $g(x)$ over $\mathbf{F}_2$ and a initial bit sequence of length equal to the degree of the sequence. A sample use of the cryptosystem follows. The shrinking generator cryptosystem is a cryptosystem based on a pair of LFSR's as defined in class.

```
sage: F2 = FiniteField(2)
sage: P2.<x> = PolynomialRing(F2)
sage: g = x^17 + x^5 + 1
sage: IS = [ F2.random_element() for i in range(17) ]
sage: LFSR = LFSRCryptosystem()
sage: PT = LFSR.encoding("The dog ate my assignment."); PT
01010100011010000110010100100000011001000110111101100110
01000000110000101110100011001010010000001101101011110010010
1000000110000101110011011110011011010010110011101101110011
011010110010101101110011101000001011110
sage: K = (g,IS)
sage: e = LFSR(K)
sage: CT = e(PT)
sage: PT == e(CT)
True
```

Note that the encoding of the message is not ciphertext – this is the standard ASCII bit encoding.

Exercise 6.7 *Consider the coefficient sequence for $f(x)/g(x)$ in $\mathbf{F}_2[[x]]$, where $g(x) = 1 + x + x^4$ and $f(x) = 1 + x^3$. Is $g(x)$ an irreducible polynomial? A primitive polynomial? Draw the associated linear feedback shift register. What is the initial state of the shift register?*

Exercise 6.8 *Compute the linear complexity of the sequences* 11, 1011, 10101, 10110, *and* 10011.

Exercise 6.9 *Compute the first 8 terms of the linear complexity profile of the coefficient sequence from Exercise 1.*

Exercise 6.10 *Practice encoding and enciphering with the LFSR stream cryptosystem. The function* `binary_decoding` *easily converts this back to ASCII text. Use these functions to verify that* `PT` *is just the binary encoding of the original plaintext message and that the ciphertext is enciphered.*

Exercise 6.11 *Since the LFSR is the bitsum of the binary keystream, generated by the connection polynomial and initial state, why must the inverse key be equal to the key itself?*

Elementary Number Theory

In this lecture we assume that R is one of the rings $\mathbf{Z}$ or $\mathbf{F}_2[x]$, m is an element of R, and we denote by (m) or mR the set $\{mx \,:\, x \in R\}$, which is called an *ideal* of R. The principle goal is to introduce the *quotient* or *residue class rings* R/mR and to understand how to work with its elements. We refer to m as the *modulus* of R/mR.

7.1   Quotient rings

The residue class ring R/mR is a commutative ring, whose elements are sets, called *cosets* of mR, of the form
$$\overline{a} = a + mR = \{a + mx \,:\, x \in R\},$$
and multiplication and addition laws are derived from that on R:
$$\overline{a} + \overline{b} = (a + mR) + (b + mR) = (a + b) + mR = \overline{(a + b)},$$
$$\overline{a} * \overline{b} = (a + mR) * (b + mR) = (a * b) + mR = \overline{(a * b)}.$$
The fact that R/mR is a ring means that the addition $(+)$ and multiplication $(*)$ are well-defined on cosets, and satisfy the usual associative and distributive laws.

Example 7.1 *Consider the ring $R/mR = \mathbf{Z}/m\mathbf{Z}$ with modulus $m = 21$. We consider the addition and muliplication of $\overline{2} = \overline{23}$ and $\overline{-2} = \overline{19}$, and show that in each pair of equal elements, we can use either the first or the second representative to define the sum and product. First, for addition, we find:*
$$\overline{2} + \overline{-2} = \overline{2 + (-2)} = \overline{0},$$
but on the other hand:
$$\overline{23} + \overline{19} = \overline{23 + 19} = \overline{42},$$
which equals $\overline{0}$ since 42 is in $21\mathbf{Z}$. Multiplication is similarly independent of the represen-tatives we chose:
$$\overline{2} * \overline{-2} = \overline{2 * (-2)} = \overline{-4} = \overline{17},$$

which holds since $-4 = 17 + (-1) * 21$, *or*

$$\overline{23} + \overline{19} = \overline{437} = \overline{17},$$

where the latter identity is determined by $437 = 17 + 420 = 17 + 20 * 21$.

7.2 The mod operator

In both rings $R = \mathbf{Z}$ and $R = \mathbf{F}_2[x]$, we have an operator $\bmod m$ for producing a canonical smallest representative for elements of the quotient rings R/mR. This means that we can work with this smallest or reduced representative in computations in R/mR. In particular, we note that working with this representative is well-defined:

$$((a \bmod m) + (b \bmod m)) \bmod m = (a + b) \bmod m$$
$$((a \bmod m) * (b \bmod m)) \bmod m = (a * b) \bmod m$$

since, $a \bmod m = b \bmod m$ if and only if $\overline{a} = \overline{b}$.

The value $a \bmod m$ can be computed by long division — successively subtracting off multiples until the result is smaller, until the final result is smaller than m. The definition of x smaller than y is $x < y$ for positive x, y in $\mathbf{Z}$, and $\deg(x) < \deg(y)$ for polynomials x, y in $\mathbf{F}_2[x]$.

Definition 7.2 *Using the* mod *operator we define a boolean operator* $\_ \equiv \_ \bmod m$ *such that the value* $a \equiv b \bmod m$ *is* True *if and only if* $(a - b) \bmod m$ *is zero.*

Example 7.3 *We use the operator* mod *to determine a canonical representative for* x^7 *in* $\mathbf{F}_2[x]/(x^2 + x + 1)$. *First we write* $x^7 = (x^3)^2 * x$, *and compute:*

$x^3 \bmod (x^2 + x + 1)$
$$= (x^3 + x * (x^2 + x + 1)) \bmod (x^2 + x + 1)$$
$$= (x^2 + x) \bmod (x^2 + x + 1)$$
$$= ((x^2 + x) + (x^2 + x + 1)) \bmod (x^2 + x + 1) = 1.$$

It follows that $x^7 \bmod x^2 + x + 1 = (1^2 * x) \bmod (x^2 + x + 1) = x$. *By explicit long division:*

$$
\begin{array}{r}
x^5 + x^4 + \qquad\quad x^2 + x \\
\hline
x^2 + x + 1 \,) \, x^7 \qquad\qquad\qquad\qquad\qquad \\
x^7 + x^6 + x^5 \qquad\qquad\qquad \\
\hline
x^6 + x^5 \qquad\qquad\qquad \\
x^6 + x^5 + x^4 \qquad\qquad \\
\hline
x^4 \qquad\qquad \\
x^4 + x^3 + x^2 \\
\hline
x^3 + x^2 \\
x^3 + x^2 + x \\
\hline
x
\end{array}
$$

we find similarly that $x^7 = (x^5 + x^4 + x^2 + x) * (x^2 + x + 1) + x$, verifying the equality $x^7 \bmod (x^2 + x + 1) = x$.

7.3 Primes and Irreducibles

A nonzero ideal (p) in R $(= \mathbf{Z}$ or $\mathbf{F}_2[x])$ is said to be a prime ideal if p is a prime number or an irreducible polynomial. The following theorem is a generalization of Fermat's Little Theorem.

Theorem 7.4 *Let (p) be a prime ideal of R and let N equal $\#R/(p) - 1$. Then $\bar{a}^N = 1$ for every nonzero $\bar{a}$ in $R/(p)$. Conversely if there exists an element $\bar{a}$ in $R/(p)$ of exact order N, then (p) is prime.*

N.B. Recall that we define the polynomial $g(x)$ to be primitive if and only if the element $\bar{x}$ has exact order N in $R/(g(x))$.

Constructing irreducible polynomials

We now want to enumerate the the irreducible polynomials in $\mathbf{F}_2[x]$ of low degree, and in the process explain some of the steps for more efficiently determining [ir]reducibility of polynomials.

Degree 1: The linear polynomials x, $x + 1$ are necessarily irreducible.

Degree 2: The polynomial $x^2 + x + 1$ is irreducible by the previous theorem and the fact that $\bar{x}$, $\bar{x}^2 = \bar{x} + 1$, and $\bar{x}^3 = 1$. Conversely, it is clear to see that the only other candidates: x^2, $x^2 + x$, and $x^2 + 1 = (x + 1)^2$ are reducible.

Lemma 7.5 *If $f(x)$ is a polynomial, then $f(x) \bmod (x - a) = f(a)$, and in particular $f(x) = (x - a)g(x)$ if and only if $f(a) = 0$*

For polynomials over $\mathbf{F}_2$, the value $f(0)$ is the constant term, and $f(1)$ is the number of nonzero coefficients $\bmod 2$, which gives an easy test for divisibility by linear polynomials.

Degree 3: By the previous test, it is clear that the only nontrivial candidates to consider are

$$x^3 + x + 1, \quad x^3 + x^2 + 1,$$

and that these are automatically irreducible, since they have no linear factor.

Degree 4: We first exclude $(x^2 + x + 1)^2 = x^4 + x^2 + 1$, the only degree four polynomial which is divisible by an irreducible polynomial of degree 2. Every other reducible polynomial

---

must therefore have a divisor of degree 1, and we apply the lemma to reduce to the list of irreducible polynomials:

$$x^4 + x^3 + 1, \quad x^4 + x + 1, \quad x^4 + x^3 + x^2 + x + 1.$$

Degree 5: As in degree 4, we exclude those polynomials which have a divisor of degree 2:

$$(x^2 + x + 1)(x^3 + x + 1) = x^5 + x^4 + 1$$
$$(x^2 + x + 1)(x^3 + x^3 + 1) = x^5 + x + 1,$$

after which we conclude that all other polynomials of degree 5 with constant term 1 and an odd number of coefficients are irreducible:

$$\begin{array}{cc} x^5 + x^3 + 1, & x^5 + x^2 + 1, \\ x^5 + x^4 + x^3 + x^2 + 1, & x^5 + x^4 + x^3 + x + 1, \\ x^5 + x^4 + x^2 + x + 1, & x^5 + x^3 + x^2 + x + 1. \end{array}$$

Exercise. Determine which of the above polynomials are primitive.

Cyclotomic polynomials

In the previous lecture we found that there are six irreducible polynomials of degree five. In order to understand and to count the numbers of irreducible and primitive polynomials, we first introduce cyclotomic polynomials.

Definition 7.6 *The cyclotomic polynomials* $\Phi_N(x)$ *are defined recursively by the identity:*

$$x^N - 1 = \prod_{m|N} \Phi_m(x).$$

Example 7.7 *To demonstrate how this serves to define the cyclotomic polynomials, we compute the first few examples:*

$$\begin{array}{ll} \Phi_1(x) = x - 1 & \Phi_4(x) = x^2 + 1 \\ \Phi_2(x) = x + 1 & \Phi_5(x) = x^4 + x^3 + x^3 + x^2 + x + 1 \\ \Phi_3(x) = x^2 + x + 1 & \Phi_6(x) = x^2 - x + 1 \end{array}$$

Moreover, if p is a prime, then

$$\Phi_p(x) = \frac{x^p - 1}{x - 1} = x^{p-1} + \cdots + x + 1.$$

So far, the definition of cyclotomic polynomials does not make use of polynomials being defined over $\mathbf{F}_2$, and if we instead let the coefficient ring be $\mathbf{Z}$, then we have the following classical result.

Theorem 7.8 *The cyclotomic polynomial $\Phi_N(x)$ is irreducible over $\mathbf{Z}$, of degree $\varphi(N)$.*

The function $\varphi(N)$ is called the Euler φ-function, and is defined by

$$\varphi(N) = \prod_{p^r || N} p^{r-1}(p-1),$$

where $p^r || N$ means that p^r divides N but that p^{r+1} does not divide N.

The analogous statement about irreducibility over $\mathbf{F}_2$ is false, but we can make a very precise statement of the form of the factorization of cyclotomic polynomials over $\mathbf{F}^2$.

Theorem 7.9 *An irreducible polynomial $g(x) \in \mathbf{F}_2[x]$ of degree n divides the polynomial $x^N + 1$ and no other polynomial $x^m + 1$ for $m < N$ if and only if $g(x)$ divides $\Phi_N(x)$. The integer N equals $2^n - 1$ if and only if $g(x)$ is primitive.*

Corollary 7.10 *The cyclotomic polynomial $\Phi_N(x) \in \mathbf{F}_2[x]$ for $N = 2^n - 1$ is the product of the distinct primitive polynomials of degree n.*

Example 7.11 *Previously we found that there were 6 irreducible polynomials of degree 5 in $\mathbf{F}_2[x]$. Since $N = 2^5 - 1 = 31$ is prime, every irreducible polynomial of degree 5 is in fact primitive. Since the degree of $\Phi_{31}(x)$ is $\varphi(31) = 31 - 1 = 30$, we could have concluded in advance that there were exactly 6 primitive and irreducible polynomials of this degree.*

LFSR cryptosystems

Since the period $N = 2^n - 1$ of the LFSR output sequence, with primitive connection polynomial, grows exponentially in the size of n, LFSR's provide good constructions for sequences of large period. Moreover a LFSR can be made computationally efficient by choosing a sparse primitive polynomial such as

$$\begin{array}{rl}
14: & x^{14} + x^7 + x^5 + x^3 + 1 \\
15: & x^{15} + x^5 + x^4 + x^2 + 1 \\
16: & x^{16} + x^5 + x^3 + x^2 + 1 \\
17: & x^{17} + x^3 + 1
\end{array}$$

A naïve stream cryptosystem can be built from a LFSR by taking the bit sum of the keystream with the message stream to produce ciphertext. Unfortunately, knowledge of just $2n$ bits of the LFSR keysteam allows the determination of the entire sequence, by an algorithm due to Berlekamp and Massey. Therefore such a LFSR cryptosystem should be considered insecure. A relatively new stream cryptosystem, called the *shrinking generator* cryptosystem, using two LFSR's in unison, has so far resisted any such algorithms.

Shrinking Generator cryptosystem

Let L_1 and L_2 be two LFSR's with output sequences $a_0a_1a_2\ldots$ and $s_0s_1s_2\ldots$. The first sequence is called the input sequence and the second sequence the controllingt sequence. At clock cycle i bits a_i and s_i are output. If s_i is 0 then the bit a_i is discarded, and otherwise a_i is output as part of the output keystream. The resulting keystream

$$z_1z_2z_3\ldots z_n\cdots = a_{i_1}a_{i_2}a_{i_3}\ldots a_{i_n}\ldots$$

is used for forming the bit sum with the message stream to form ciphertext.

Exercises

Reduction modulo a polynomial $g(x)$ or modulo an integer m plays a central role in the mathematics of cryptography. Recall that for a monic polynomial $g(x)$ of positive degree, we define $a(x) \bmod g(x)$ to the unique polynomial $a_0(x)$ with $\deg a_0(x) < \deg g(x)$ such that

$$a(x) = a_0(x) + a_1(x)g(x).$$

For an integer m, we define $a \bmod m$ to be the unique integer a_0 with $0 \le a_0 < m$ such that $a = a_0 + a_1 m$.

Fermat's little theorem. If p is a prime, then the relation $a^{p-1} \equiv 1 \bmod p$ holds for any integer a not divisible by p.

Note. The SAGE function `mod` operates on integers, with the syntax:

```
sage: m = 101
sage: (2^31).mod(m)
34
```

The same mathematical result can be achieved with the `powermod` function (for modular powering):

```
sage: 2.powermod(31,m)
34
```

The latter construction, however, is more efficient.

Chinese remainder theorem. Let p and q be distinct primes, then for every integer a and b there exists a unique integer c with $0 \le c < pq$ such that $c \equiv a \bmod p$ and $c \equiv b \bmod q$.

If a, b, and c are as above, then for any integral polynomial $f(x)$, the integer $f(c)$ satisfies $f(c) \equiv f(a) \bmod p$ and $f(c) \equiv f(b) \bmod q$. Therefore $f(c) \bmod pq$ is the unique solution to the Chinese remainder theorem.

Analogues of Fermat's little theorem also hold for polynomials.

Polynomial analogue of Fermat. If $g(x)$ is an irreducible polynomial of degree n over $\mathbf{F}_2$, then the relation $a(x)^{2^n-1} \equiv 1 \bmod g(x)$ holds for any polynomial $a(x)$ not divisible by $g(x)$.

Chinese remainder theorem. Let $g(x)$ and $h(x)$ be monic polynomials with no common factors. Given any polynomials $a(x)$ and $b(x)$, there exists a unique polynomial $c(x)$ such that $c(x) \equiv a(x) \bmod g(x)$ and $c(x) \equiv b(x) \bmod h(x)$.

We can create and work with polynomials over $\mathbf{F}_2$ as demonstrated by the following SAGE code.

```
sage: F2 = FiniteField(2)
sage: P2.<x> = PolynomialRing(F2)
sage: f = x^17 + x^5 + 1
sage: f.factor()
x^17 + x^5 + 1
sage: g = x^13 + x^5 + 1
sage: g.factor()
(x^2 + x + 1) * (x^11 + x^10 + x^8 + x^7 + x^5 + x^4 + x^3 + x + 1)
```

Exercise 7.1 *Let p be the prime $2^{31} - 1 = 2147483647$. Use the SAGE function* `powermod` *to verify Fermat's little theorem for several values of a. Why would it be a bad idea to compute a^{p-1} and then reduce modulo p?*

Exercise 7.2 *Let p be as above and let $q = (2^{61} + 1)/3 = 768614336404564651$. Compute $a^{p-1} \bmod pq$ for various primes using* `powermod`. *Then reduce the result modulo p. How do you explain the result in terms of the Chinese remainder theorem and Fermat's little theorem?*

Exercise 7.3 *Let $g(x) = x^{17} + x^5 + 1$, and use the function* `powermod` *to verify the polynomial analogue of Fermat's little theorem for the polynomials x, $x^2 + x + 1$, etc.*

Exercise 7.4 *Let $h(x) = x^{17} + x^{15} + x^{10} + x^5 + 1$ and compute $a(x)^{2^{17}-1} \bmod g(x)h(x)$ for various $a(x)$. What is the result reduced modulo $g(x)$? Why does the same not hold true for $a(x)^{2^{17}-1} \bmod g(x)h(x)$, reduced modulo $h(x)$?*

Public Key Cryptography

The theory of public key cryptography was introduced by Diffie and Hellman in 1976. Public key cryptography does not displace symmetric key cryptography — they solve different problems. The recent NIST contest which resulted in the Advanced Encryption Standard did not resulted in a new symmetric key, not public key standard. Why? Symmetric key algorithms do the bulk of encryption, and are orders of magnitude faster than public key systems of comparable security. They dispense with the restrictive conditions needed to build the split of public and private keys, and focus on speed. Public key cryptography, on the other hand, solves the key exchange problem — how to establish a common key between two parties that may have never met. It also finds use in specialized algorithms for digital signatures and message authentication.

The foundational concept of public key cryptography is that of the invertible one-way function $f : X \rightarrow Y$ — a function which is efficiently computable on any value $x \in X$, but for which the inverse is hard: given y finding an x such that $y = f(x)$ is computationally hard. Most public key systems rely on *trapdoor* one-way functions. For such a function the constructor of the function has priviledged information which allows the efficient inversion of the function.

We begin with a description of symmetric and public protocols for message exchange in order to understand the role of each in cryptography.

8.1   Public and Private Key Protocols

We describe the standard symmetric and public key cryptography protocols, then describe a hybrid protocol, which could be used to exchange messages efficiently using public key cryptography for key exchange and symmetric key encryption for the message content. Each of these protocols describes the sequence of steps by which Alice can send a secure message to Bob. The word *public* or *sends* refer to events or information to which an adversary can have access. The word *private* refers to information or an exchange which is protected from evesdropping.

Symmetric Cryptography Protocol

Initial setup:
1. Alice and Bob publicly agree on a cryptosystem.
For each message Alice → Bob:
1. Alice and Bob privately agree on a key.
2. Alice enciphers her plaintext using the agreed key.
3. Alice sends the ciphertext message to Bob.
4. Bob deciphers the ciphertext message to obtain the plaintext.

The need to exchange keys for every exchange was a critical problem prior to the theoretical solution with public key cryptography by Diffie and Hellman. The public key protocol proceeds as follows, requiring only one initial public key exchange.

Public Key Cryptography Protocol

Initial setup:
1. Alice and Bob publicly agree on a cryptosystem.
2. Bob sends Alice his public key.
For each message Alice → Bob:
1. Alice enciphers her plaintext using Bob's public key.
2. Alice sends the ciphertext message to Bob.
3. Bob deciphers the ciphertext message using his private key.

As noted in the preceeding discussion, public key cryptosystems are several orders of magnitude slower than comparable symmetric key cryptosystems. Moreover, a public key cryptosystem is susceptible to chosen plaintext attacks: an adversary can create a lookup table of pairs $(E_K(M), M)$ for the known public key K. Therefore the amount of public key ciphertext which is transmitted should be strictly limited.

These considerations lead to the following hybrid protocol, in which the public key cryptosystem is used for key exchange and a fast symmetric key cryptosystem is used for message encryption.

Hybrid Cryptographic Protocol

Initial setup:
1. Alice and Bob publicly agree on a public key cryptosystem and a symmetric key cryptosystem.
2. Bob sends Alice his public key.

For each message Alice $\rightarrow$ Bob:
1. Alice generates a random session key K.
2. Alice enciphers K using Bob's public key.
3. Alice enciphers the plaintext message using K.
4. Alice sends Bob the enciphered session key and the ciphertext message.
5. Bob deciphers the enciphered session key using his private key.
6. Bob deciphers the ciphertext message using the session key.

An additional layer can be added to the hybrid exchange, involving a trusted authority, which carries out the function of certifying the identity of individuals and their public keys, managing a database of public keys, and handling expiration of public keys. The public key exchange step, in which Bob sends Alice his public key, in the hybrid protocol can therefore be replaced with any of the following:

- Alice gets Bob's key from Bob.
- Alice gets Bob's key from a trusted authority's database.
- Alice gets Bob's key from her private database.

Trapdoor one-way functions

Several years elapsed between when Diffie and Hellman presented their *New directions in cryptography* with the theory of public key cryptography based on trapdoor one-way functions, and the discovery of a practical example of trapdoor one-way functions for cryptographic use. The principal public key cryptosystems in use today, based on trapdoor one-way functions, are the RSA cryptosystem and the ElGamal cryptosystem. We describe the underlying mathematical functions used in the RSA and ElGamal constructions.

RSA.

The trapdoor one-way function used in RSA is a fixed exponentiation in $\mathbf{Z}/n\mathbf{Z}$ for a composite integer n.

$$\mathbf{Z}/n\mathbf{Z} \longrightarrow \mathbf{Z}/n\mathbf{Z}$$
$$m \longmapsto m^e$$

The public data is the exponent e and the modulus n, which is presumed to be of the form pq for distinct primes p and q. The presumed difficulty of inverting this function is based

on the difficulty of factoring n, or on the difficulty of a weaker problem version called the RSA problem. The trapdoor for this function is the knowledge of the factorization of n.

ElGamal.

The ElGamal function is based on the exponentiation of a fixed multiplicative generator a for $\mathbf{F}_p^*$, the group of nonzero elements of the finite field of p elements. The map takes m to the power $a^m \bmod p$:

$$\mathbf{Z}/(p-1)\mathbf{Z} \longrightarrow \mathbf{F}_p^*$$
$$m \longmapsto a^m$$

The public data is the generator a and the prime p, and inverting the cryptographic function is solved by the discrete logarithm problem – finding m given a and a^m, or on a weaker problem called the Diffie Hellman problem.

Elliptic Curves.

Alternatively, the ElGamal construction can be solved using the discrete logarithm problem on an elliptic curve E over a finite field $\mathbf{F}_q$. An elliptic curve is defined by an equation of the form

$$y^2 + (a_1 x + a_3)y = x^3 + a_2 x^2 + a_4 x + a_6$$

with fixed coefficients a_1, a_2, a_3, a_4, a_6 in $\mathbf{F}_q$. The set $E(\mathbf{F}_q)$ of points (x, y) in $\mathbf{F}_q^2$ solving this equation plus a distiguished point O at infinity forms a group law under an addition, which we denote $+$. For groups $E(\mathbf{F}_q)$ and $\mathbf{F}_p^*$ of comparable size, the discrete logarithm problem on the elliptic curve is believed to be a harder problem than the classical one in $\mathbf{F}_p^*$. Elliptic curves will not be a topic of discussion in this book, but are becoming commonplace in implementations of smart cards and cryptography for low-power devices.

8.2   RSA Cryptosystems

The RSA cryptosystem is based on the difficulting of finding an inverse to exponentiation by a fixed e on $\mathbf{Z}/n\mathbf{Z}$ for composite n, applied for n equal to a product of two large primes p and q. An efficient solution to integer factorization, or a solution to the possibly weaker *RSA problem* would render the RSA cryptosystem insecure. The trapdoor for RSA encryption is the knowledge of the prime factors of n (which allows the determination of an inverse deciphering exponent d for any enciphering exponent e).

RSA Problem. Given integers e and $n = pq$ such that $\mathrm{GCD}(e, p-1) = \mathrm{GCD}(e, q-1) = 1$, and an integer c, find an m such that $c = m^e \bmod n$.

Rule: Let p be a prime, then the reduction $m^x \bmod p$ remains unchanged whenever

(a) m is changed by a multiple of p, or
(b) x is changed by a multiple of $p - 1$.

We apply this rule in the RSA algorithm for $x \equiv 1 \bmod p - 1$ to conclude that $m = m^x \bmod p$.

The correctness of the deciphering follows from the construction of d such that $ed \equiv 1 \bmod n$. By the above rule, it follows that

$$m \equiv m^{ed} \bmod p \text{ and } m \equiv m^{ed} \bmod q,$$

from which it follows that $m = m^{ed} \bmod pq$.

Example. Let $p = 11$ and $q = 23$, so $n = 253$, and $e = 3$. We verify that $\mathrm{GCD}(e, p-1) = 1$ and $\mathrm{GCD}(e, q - 1) = 1$. Suppose that the ciphertext is $c = 29$ in $\mathbf{Z}/253\mathbf{Z}$. To construct the inverse of exponentiation by e, we need to find d_1 and d_2 such that

$$e\, d_1 \equiv 1 \bmod (p - 1)$$
$$e\, d_2 \equiv 1 \bmod (q - 1)$$

First we compute the extended GCD's of the pairs $(e, p-1) = (3, 10)$ and $(e, q-1) = (3, 22)$. These are given by the relations:

$$-3 \cdot 3 + 1 \cdot 10 = 1$$
$$-7 \cdot 3 + 1 \cdot 22 = 1$$

giving the equalities $-3e \equiv 1 \bmod 10$ and $-7e \equiv 1 \bmod 22$. Therefore if $ed_1 \equiv 1 \bmod 10$, then we have

$$d_1 \equiv (-3\, e)\, d_1 \bmod 10 \equiv -3\, (e\, d_1) \bmod 10 \equiv -3 \bmod 10 \equiv 7 \bmod 10.$$

Similarly if $ed_2 \equiv 1 \bmod 22$, then we have

$$d_2 \equiv (-7\, e)\, d_2 \bmod 22 \equiv -7(e\, d_2) \bmod 22 \equiv -7 \bmod 22 \equiv 15 \bmod 22.$$

Now we can compute the plaintext message m such that $c = m^3 \bmod 253$. First we compute

$$m_1 = c^{d_1} \bmod 11 \equiv 7^7 \bmod 11 \equiv 7^4\, 7^2\, 7 \bmod 11$$
$$\equiv 3 \cdot 5 \cdot 7 \bmod 11 \equiv 3 \cdot 2 \bmod 11 \equiv 6 \bmod 11,$$

and similarly

$$m_2 = c^{d_2} \bmod 23 \equiv 6^{15} \bmod 23$$
$$\equiv 6^8 \, 6^4 \, 6^2 \, 6 \bmod 23 \equiv (-5 \cdot 8)\,(13 \cdot 6) \bmod 23$$
$$\equiv 6 \cdot 9 \bmod 23 \equiv 8 \bmod 23.$$

Now we can combine $m_1 = 6 \bmod 11$ and $m_2 = 8 \bmod 23$ by the Chinese remainder theorem. We expressed the extended GCD of 11 and 23 as:

$$rp + sq = -2 \cdot 11 + 1 \cdot 23 = 1.$$

Setting $m = m_2 + kq$, we find $m_1 = (m_2 + kq) \bmod p$, whence

$$s(m_1 - m_2) \equiv s(k\,q) \bmod p \equiv k(s\,q) \bmod p \equiv k \bmod p.$$

So we have solved for $k \equiv s(m_1 - m_2) \bmod p \equiv 1\,(6 - 8) \equiv -2 \bmod p$. Therefore $m \equiv m_1 + kq \bmod 253 \equiv 6 - 46 \bmod 253 \equiv 213 \bmod 253$.

RSA with exponent 3. A commonly used exponent for RSA encryption is $e = 3$. This allows efficient enciphering using only two arithmetic operations (two multiplications or one squaring and one multiplication). No such gain is achieved for deciphering.

However, this presents algorithm presents the following problem for security. Let m be a message to be sent to three parties, with RSA moduli n_1, n_2, and n_3. The encoding of the message satisfies $0 \le m < n_i$. By means of the Chinese remainder theorem, we can recover $c = m^3 \bmod n_1 n_2 n_3$ from the three enciphered messages $c_1 = m^3 \bmod n_1$, $c_2 = m^3 \bmod n_2$, and $c_3 = m^3 \bmod n_3$. While the latter messages c_i, as the modular representatives of some huge integer, appear random. But from the bounds on m, the cube satisfies the bound:

$$0 \le m^3 < n_1 n_2 n_3,$$

hence the smallest modular representative c equals m^3, and the cube root can be extracted as an integer to recover m.

A valid protocol to overcome this dilemma, for $e = 3$, is to never send the same message to more than one party. This is achieved by adding unique random padding to every message prior to enciphering. This turns the message m into three distinct messages m_1, m_2, and m_3. The Chinese remainder theorem then solves for some integer

$$0 \le c < n_1 n_2 n_3$$

such that $c \equiv m_i \bmod n_i$, but this integer no longer bears any relation to any cube m^3.

8.3   ElGamal Cryptosystems

The ElGamal Cryptosystem is implicitly based on the difficultly of finding a solution to the discrete logarithm in $\mathbf{F}_p^*$: given a primitive element a of $\mathbf{F}_p^*$ and another element b,

the discrete logarithm problem (DLP) is the computational problem of finding $x = \log_a(b)$ such that $b = a^x$.

Efficient algorithms for the discrete logarithm problem would render the ElGamal Cryptosystem insecure, the possibly weaker Diffie-Hellman problem (DHP) is the precise problem on which the cryptosystem is based: given $b = a^x$ and $c = a^y$ in $\mathbf{F}_p^*$, compute a^{xy}.

Note that a^{xy} can not be formed as any obvious algebraic combination of a^x and a^y like $a^x a^y = a^{x+y}$. In fact, other cryptosystems rely on the difficult of the Decision Diffie-Hellman problem (DDHP) being hard: given a^x, a^y and c, decide whether or not $c = a^{xy}$. Both the DHP and the DDHP are easy if the DLP is easy.

Definition. Recall that an element a of $\mathbf{F}_p^*$ is said to be *primitive* if and only if

$$1, a, a^2, \ldots, a^{p-2}$$

are all distinct. Primitive elements always exist in any finite field.

ElGamal Protocol

Public key: (a, a^x, p) where p is a prime, a is a primitive element of $\mathbf{F}_p^*$, and x is an integer $1 \le x < p - 1$.
Private key: The integer x.
Initial setup:
1. Alice obtains Bob's public key (a, a^x, p).
For each message m Alice $\rightarrow$ Bob:
1. Alice chooses a private element y randomly in $1 \le y < p - 1$.
1. Alice computes $r = a^y$ and $s = ma^{xy}$.
2. Alice sends the ciphertext message $c = (r, s)$ to Bob.
3. Bob deciphers the ciphertext message as $m = r^{-x} s \bmod p$.

The correctness of the deciphering is verified as follows:

$$r^{-x}s = (a^y)^{-x} ma^{xy} = ma^{-yx} a^{xy} = ma^{xy-yx} = m.$$

Disrete Logarithms

The main known attack on an ElGamal cryptosystem is to solve the discrete logarithm problem: given both a and a^x (in the finite field $\mathbf{F}_p$), find the value for x. In order for the discrete logarithm problem (DLP) to be hard, it is not enough to choose any prime p. One needs to select a prime p such that $p - 1$ has a large prime factor. Suppose, on the contrary, that $p - 1$ is divisibly only by primes less than some positive integer B. Such a number is said to be B-smooth. The DLP can be reduced to solving a small number of discrete logarithm problems of "size" B rather than of size $p - 1$.

As an example, let r be a prime divisor of $p-1$, and let $m = (p-1)/r$. Suppose that we want to solve for x such that $b = a^x$. The exponent is defined up to multiples of $p-1$. If we

raise both sides to the power m, then for the problem $b^m = a^{mx}$ a solution x is well-defined up to multiples of r:

$$a^{m(x+r)} = a^{mx+mr} = a^{mx}a^{p-1} = a^{mx},$$

since $a^{p-1} = 1$.

If we now find that $p - 1 = r_1 r_2 \cdots r_t$ for pairwise distinct primes r_i, the by the Chinese remainder theorem the value of $x \bmod p - 1$ can be determined from its modular values $x \bmod r_i$, for all $1 \leq i \leq t$. So the hardness of the DLP determined by the size of the largest prime divisor of $p - 1$.

Exercise. Suppose that a prime power r^k divides $p - 1$. How would you solve the DLP for $x \bmod r^k$?

Algorithmic Considerations

A naïve algorithm for solving the discrete logarithm problem for $\log_a(b)$ is to compute $1, a, a^2, \ldots$ until a match is found with b. As we have just seen, it is possible to replace a with $a_1 = a^m$ and b with $b_1 = b^m$ in order to solve $\log_{a_1}(b_1)$ modulo r such that $rm = p-1$. In this way we have to build the list $1, a_1, a_1^2, \ldots, a_1^x$ of length at most r before finding b_1.

An alternative approach is called the baby-step, giant-step method. We set $s = [\sqrt{r}] + 1$ and to form a first list $1, a_1, a_1^2, \ldots, a_1^{s-1}$ of length s, called the baby steps, then form the second list $b_1, a_1^s b_1, a_1^{2s} b_1, \ldots, a_1^{s^2} b_1$ of giant steps, to find a match.

If a match is found, say $a_1^i = b_1 a_1^{js}$, then we have found $b_1 = a_1^{i-js}$, so $x = i - js \bmod r$. On the other hand, if x is a solution to the DLP $\bmod r$, then we can write $x = i - js$ for some $0 \leq i, -j \leq s$, so the above algorithm finds a match.

8.4 Diffie–Hellman Key Exchange

Diffie and Hellman proposed the following scheme for establishing a common key. The scheme is widely used because of the simplicity of its implementation, however an naive implementation without identity authentication leaves the protocol subject to a man-in-the-middle attack.

1. A and B decide on a large prime number p and a primitive element a of $\mathbf{Z}/p\mathbf{Z}$, both of which can be made public.

2. A chooses a secret random x with $\mathrm{GCD}(x, p-1) = 1$ and B chooses a secret random y with $\mathrm{GCD}(y, p-1) = 1$.

3. A sends Bob $a^x \bmod p$ and Bob sends Alice $a^y \bmod p$.

4. Each is able to compute a session key $K = a^{xy} = (a^x)^y = (a^y)^x$.

An eavesdropper only has knowledge of p, a, a^x and a^y, and would need to break the Diffie-Hellman problem to be able to come up with the session key.

Man in the Middle Attack

The man-in-the-middle attack is a protocol for an eavesdropper E to intercept a message exchange between A and B. The attack is premised on a Diffie-Hellman key exchange, but the principle applied to any public key cryptosystem for which the keys used for public key exchange is not certified with a cerification authority.

We assume that A and B have agreed on a prime p and a primitive element a of $\mathbf{Z}/p\mathbf{Z}$, and that E is positioned between A and B. Having observed this Diffie-Hellman initialization E prepares for the man-in-the-middle attack.

1. A chooses a secret key x, creates a public key a^x, and sends it to B, which is intercepted by E.

2. E chooses a private integer z at random, and creates the alternative public key a^z which she sends to B, pretending to be A. At the same time she sends same key a^z to A, now posing as B.

3. Now E has established a common session key a^{xz} with A and common session key a^{yz} with B. Message exchanges between A and B pass through E and can be deciphered, read, modified, re-enciphered, and resent in transit.

The breakdown of the key exchange protocol is due to lack of identity authentication of the communicating parties. If, for instance the public key (a, a^x, p) of A could be confirmed with an independent certification authority, then B would not have confused E with A.

Exercises

The RSA cryptosystem is based on the difficulty of factoring large integers into its composite primes.

Based on Fermat's little theorem, we know that $a^m \equiv 1 \bmod p$ exactly when $p-1$ divides m. Therefore we recover the identity $a^u \equiv a \bmod p$ where u is of the form $1 + (p-1)r$. Now given any e such that e and $p-1$ have no common divisors, there exists a d such that $ed \equiv 1 \bmod p-1$. In other words, $u = ed$ is of the form $1 + (p-1)r$. This means that the map

$$a \mapsto a^e \bmod p$$

followed by

$$a^e \bmod p \mapsto (a^e \bmod p)^d \bmod p \equiv a^{ed} \bmod p = a \bmod p$$

are inverse maps. This only works for a prime p.

Exercise 8.1 *Use* SAGE *to find a large prime p and to compute inverse exponentiation pairs e and d. The following functions are of use:*

$$\texttt{random\_prime}, \texttt{gcd}, \texttt{xgcd}, \textit{and } \texttt{inverse\_mod}.$$

The RSA cryptosystem is based on the fact that for primes p and q and any integer e with no common factors with $p-1$ and $q-1$, it is possible to find an d_1 such that

$$ed_1 \equiv 1 \bmod (p-1),$$
$$ed_2 \equiv 1 \bmod (q-1).$$

Using the Chinese remainder theorem, it is possible to then find the unique d such that

$$d = d_1 \bmod (p-1) \text{ and } d = d_2 \bmod (q-1)$$

in the range $1 \le d < (p-1)(q-1)$. This d has the property that

$$a^{ed} \equiv a \bmod n.$$

The send a message securely, the public key (e, n) is used. First we encoding the message as an integer $a \bmod n$, then form the ciphertext $a^e \bmod n$. The recipient recovers the message using the secret exponent d.

Exercise 8.2 *Use your exponents e, d, verify the identities mod p:*

$$(a^e)^d \equiv a \bmod n, \;\; (a^d)^e \equiv a \bmod n, \;\; \text{and } a^{ed} \equiv a \bmod n,$$

for various random values of a.

Note that after construction of d, the primes p and q are not needed, but that without knowing the original factorization of n, Fermat's little theorem does not apply, and finding the inverse exponent for e is considered a hard problem.

Exercise 8.3 *Use the above factorization to reproduce the private key $\texttt{L}$ (generated but not printed above) for this $\texttt{K}$.*

Exercise 8.4 *Why is the choice for which key is the public key and which key is the private key arbitrary? Practice encoding, decoding, enciphering, and deciphering with the RSA cryptosystem. Why do the member functions $\texttt{enciphering}$ and $\texttt{deciphering}$ return the same values?*

An ElGamal cryptosystem is based on the difficulty of the Diffie–Hellman problem: Given a prime p, a primitive element a of $(\mathbf{Z}/p\mathbf{Z})^* = \{c \in Z/pZ : c \ne 0\}$, and elements $c_1 = a^x$ and $c_2 = a^y$, find the element a^{xy} in $(\mathbf{Z}/p\mathbf{Z})^*$.

Exercise 8.5 *Recall the discrete logarithm problem: Given a prime p, a primitive element a of $(\mathbf{Z}/p\mathbf{Z})^*$, and an element c of $(\mathbf{Z}/p\mathbf{Z})^*$, find an integer x such that $c = a^x$. Explain how a general solution to the discrete logarithm problem for p and a implies a solution to the Diffie–Hellman problem.*

Exercise 8.6 *Fermat's little theorem tells us that $a^{p-1} = 1$ for all a in $(\mathbf{Z}/p\mathbf{Z})^*$. Recall that a primitive element a has the property that $\mathbf{Z}/(p-1)\mathbf{Z} \to (\mathbf{Z}/p\mathbf{Z})^*$ given by $x \mapsto a^x$ is a bijection.*

1. *Show that a is primitive if and only if $a^x = 1$ only when $p - 1$ divides x.*

2. *Let p be prime $2^{32} + 15$. Show that $a = 3$ is a primitive element of $(\mathbf{Z}/p\mathbf{Z})^*$. Use the* SAGE *function* `log` *to compute discrete logarithms of elements of* `FiniteField(p)` *with respect to a.*

3. *Let p be the prime $2^{32} + 61$. Show that the element $a = 2$ is a primitive element for $(\mathbf{Z}/p\mathbf{Z})^*$. Use the* SAGE *function* `log` *to compute discrete logarithms of elements of* `FiniteField(p)` *with respect to a.*

Exercise 8.7 *Compare the times to compute discrete logarithms in the previous exercise. Now factor $p - 1$ for each p. What difference do you note? Explain the timings in terms of the Chinese remainder theorem for $\mathbf{Z}/(p-1)\mathbf{Z}$.*

Exercise 8.8 *Let p be the prime $2^{131} + 1883$ and verify the factorization*

$$p - 1 = 2 \cdot 3 \cdot 5 \cdot 37 \cdot 634466267339108669 \cdot 3865430919824322067.$$

Let $a = 109$ and $c = 1014452131230551128319928312434869768346$ and set

$$n_5 = (p - 1) \text{ div } 634466267339108669$$
$$n_6 = (p - 1) \text{ div } 3865430919824322067.$$

Then verify that $c^{n_5} = a^{129n_5}$ and $c^{n_6} = a^{127n_6}$. Find similar relations for

$$n_1 = (p - 1) \text{ div } 2 \quad n_3 = (p - 1) \text{ div } 5,$$
$$n_2 = (p - 1) \text{ div } 3 \quad n_4 = (p - 1) \text{ div } 37.$$

and use this information to find the discrete logarithm of c with respect to a.

Digital Signatures

A digital signature is the digital analogue of a handwritten signature. The signature of a message is data dependent on some secret known only to the signer and on the content of the message. A digital signature must be verifiable without access to the signer's private key.

9.1 RSA Signature Scheme

The RSA signature scheme is a signature scheme with *message recovery* — the signed message is recovered from the signature.

Key generation. This step is exactly as for RSA enciphering. The signer generates a public key (e, n) and guards a private key (d, n), where $n = pq$ is the product of two large primes.

Signature generation. Encode the message m in $\mathbf{Z}/n\mathbf{Z}$, and output the signature $s = m^d \in \mathbf{Z}/n\mathbf{Z}$, computed using the private key (d, n).

Verification. Compute $m = s^e \in \mathbf{Z}/n\mathbf{Z}$.

9.2 ElGamal Signature Scheme

The ElGamal signature scheme requires an encoding of the message m as an element of $\mathbf{Z}/(p-1)\mathbf{Z}$.

Key generation. This step is exactly as for ElGamal enciphering. The signer generates a public key (p, a, c), where $c = a^x \bmod p$, and guards the private key (p, a, x), where a is a primitive element of $\mathbf{Z}/p\mathbf{Z}$ and x is an integer in the range $1 \leq x < p - 1$ with $\mathrm{GCD}(x, p-1) = 1$.

Signature generation. The signer selects a random secret integer k in the range $1 \leq$

$k < p - 1$ with $\text{GCD}(k, p - 1) = 1$, and computes

$$r = a^k \bmod p \text{ and } s = l(m - rx) \bmod (p - 1),$$

where $l = k^{-1} \bmod (p - 1)$, and the signature (r, s) is output.

Note that r is well-defined in $\mathbf{Z}/p\mathbf{Z}$, but that to form s it is necessary to choose a minimal positive integer representative and reinterpret it $\bmod (p - 1)$.

Verifcation. The signature is verified first that $1 \leq r \leq p - 1$, or rejected. The values

$$v_1 = c^r r^s \bmod p, \text{ and } v_2 = a^m \bmod p,$$

are next computed, and the equality $v_1 = v_2$ is verified or the signature rejected.

Proof of equality. $v_1 = c^r a^{kl(m-xr)} = a^{xr} a^{m-xr} = a^m = v_2$.

9.3   Chaum's Blind Signature Scheme

Chaum's blind signature scheme is an RSA-based scheme, adapted for blind signatures. In the protocol below we assume that Bob has set up a public RSA key (e, n) with corresponding private key (d, n), so that Bob's RSA signature functions is $S_B(m) = m^d$.

1. *Initial setup:* Alice obtains Bob's public key (e, n) and chooses a random public session key k, such that $0 < k < n$ and $\text{GCD}(k, n) = 1$.
2. *Blinding:* Alice computes $m^* = mk^e$, and sends m^* to Bob.
3. *Signing:* Bob computes $s^* = m^{*d}$, which he sends back to Alice.
4. *Unblinding:* Alice computes $s = k^{-1}s^*$, which equals $S_B(m) = m^d$.

As an application we mention a naive digital cash scheme. Suppose that Alice wants to withdraw a digital \$100 from her account to be spent anonymously at a later date. She writes 1000 notes from the bank, each certifying its value to be \$100, and blinds them, each with a separate session key. The bank asks for the session keys to 999 of these notes, verifies that each has the correct value, and blindly signs the last one, deducting \$100 from her account, and returns the blinded signed \$100 note to Alice for use as cash.

9.4   Digital Cash Schemes

We won't go into details of a particular digital cash protocol, but list the ideal properties which such a scheme should satisfy, as spelled out by Okomoto and Ohta in 1991 (Crypto'91).

1. Digital cash can be sent securely through an insecure channel.

2. Digital cash can not be copied or reused.

3. The spender remains anonymous under legitimate use of the protocol.

4. Spending does not require communication with a bank or external agency.

5. The cash is transferable.

6. The cash can be subdivided.

There are several proposed digital cash schemes, which provide both partial and full solutions to these sets of conditions. Okamoto and Ohta provide a solution to all six of these conditions. Chaum has proposed a variety of schemes which give partial solutions to different subsets of the above, and Brands has a scheme which satisfies the first four properties. The complexity of the scheme is largely dependent on the number of these properties which it satisfies, so that the most complete scheme may not be the easiest to describe or to implement.

We note that anonymity, property three of this list, relies on an analogue blind signatures called restricted blind signatures, as in the naive example above. The naive example fails the above criteria, for instance, failing to ensure against multiple spending.

Secret Sharing

A secret sharing scheme is a means for n parties to carry *shares* or *parts* s_i of a message s, called the *secret*, such that the complete set $s_1, \ldots s_n$ of the parts determines the message. The secret sharing scheme is said to be *perfect* if no proper subset of shares leaks any information regarding the secret.

Two party secret sharing. Let s be a secret, encoding as an integer in $\mathbf{Z}/m\mathbf{Z}$. Let $s_1 \in \mathbf{Z}/m\mathbf{Z}$ be generated at random by a trusted party. Then the two shares are defined to be s_1 and $s - s_1$. The secret is recovered as $s = s_1 + s_2$.

Multiple party secret sharing. Let $s \in \mathbf{Z}/m\mathbf{Z}$ be a secret to be shared among n parties. Generate the first $n-1$ shares $s_1, \ldots, s_{n-1}$ at random and set

$$s_n = s - \sum_{i=1}^{n-1}.$$

The secret is recovered as $s = \sum_{i=1}^n s_i$.

A (t, n) *threshold* secret sharing scheme is a method for n parties to carry shares s_i of a message s such that any t of the them to reconstruct the message, but so that no $t-1$ of them can easy do so. The threshold scheme is *perfect* if knowledge of $t-1$ or fewer shares provides no information regarding s.

Shamir's (t, n)-threshold scheme. A scheme of Shamir provide an elegant construction of a perfect (t, n)-threshold scheme using a classical algorithm called Lagrange interpolation. First we introduce Lagrange interpolation as a theorem.

Theorem 10.1 (Lagrange interpolation) *Given t distinct points (x_i, y_i) of the form $(x_i, f(x_i))$, where $f(x)$ is a polynomial of degree less that t, then $f(x)$ is determined by*

$$f(x) = \sum_{i=1}^{t} y_i \prod_{\substack{1 \leq j \leq t \\ i \neq j}} \frac{x - x_j}{x_i - x_j}. \tag{10.1}$$

Shamir's scheme is defined for a secret $s \in \mathbf{Z}/p\mathbf{Z}$ with p prime, by setting $a_0 = s$, and choosing $a_1, \ldots, a_{t-1}$ at random in $\mathbf{Z}/p\mathbf{Z}$. The trusted party computes $f(i)$, where

$$f(x) = \sum_{k=0}^{t-1} a_k x^k,$$

for all $1 \le i \le n$. The shares $(i, f(i))$ are distributed to the n distinct parties. Since the secret is the constant term $s = a_0 = f(0)$, the secret is reovered from any t shares $(i, f(i))$, for $I \subset \{1, \ldots, n\}$ by

$$s = \sum_{i \in I} c_i f(i), \text{ where each } c_i = \prod_{\substack{j \in I \\ j \neq i}} \frac{i}{j - i}.$$

Properties. Shamir's secret sharing scheme is (1) *perfect* — no information is leaked by the shares, (2) *ideal* — every share is of the same size p as the secret, and (3) involves no unproven hypotheses. In comparison, most public key cryptosystems rely on certain well-known problems (integer factorization, discrete logarithm problems) to be hard in order to guarantee security.

Proof of Lagrange interpolation theorem. Let $g(x)$ be the right hand side of (10.1). For each x_i in we verify directly that $f(x_i) = g(x_i)$, so that $f(x) - g(x)$ is divisible by $x - x_i$. It follows that

$$\prod_{i=1}^{t} (x - x_i) \big| (f(x) - g(x)), \tag{10.2}$$

but since $\deg(f(x) - g(x)) \le t$, the only polynomial of this degree satisfying equation (10.2) is $f(x) - g(x) = 0$.

Example. Shamir secret sharing with $p = 31$. Let the threshold be $t = 3$, and the secret be $7 \in \mathbf{Z}/31\mathbf{Z}$. We choose elements at random $a_1 = 19$ and $a_2 = 21$ in $\mathbf{Z}/31\mathbf{Z}$, and set $f(x) = 7 + 19x + 21x^2$. As the trusted pary, we can now generate as many shares as we like,

$$
\begin{aligned}
(1, f(1)) &= (1, 16) & (5, f(5)) &= (5, 7) \\
(2, f(2)) &= (2, 5) & (6, f(6)) &= (6, 9) \\
(3, f(3)) &= (3, 5) & (7, f(7)) &= (7, 22) \\
(4, f(4)) &= (4, 16) & (8, f(8)) &= (8, 15)
\end{aligned}
$$

which are distributed to the holders of the share recipients, and the original polynomial $f(x)$ is destroyed. The secret can be recovered from the formula

$$f(x) = \sum_{i=1}^{t} y_i \prod_{\substack{1 \le i \le t \\ i \neq j}} \frac{x - x_j}{x_i - x_j} \quad \Longrightarrow \quad f(0) = \sum_{i=1}^{t} y_i \prod_{\substack{1 \le i \le t \\ i \neq j}} \frac{x_j}{x_j - x_i}$$

using any t shares $(x_1, y_1), \ldots, (x_t, y_t)$. If we take the first three shares $(1, 16)$, $(2, 5)$, $(3, 5)$, we compute

$$f(0) = \frac{16 \cdot 2 \cdot 3}{(1-2)(1-3)} + \frac{5 \cdot 1 \cdot 3}{(2-1)(2-3)} + \frac{5 \cdot 1 \cdot 2}{(3-1)(3-2)}$$

$$= 3 \cdot 2^{-1} + 15 \cdot (-1) + 10 \cdot 2^{-1} = 17 - 15 + 5 = 7.$$

This agrees with the same calculation for the shares $(1, 16)$, $(5, 7)$, and $(7, 22)$,

$$f(0) = \frac{16 \cdot 5 \cdot 7}{(1-5)(1-7)} + \frac{7 \cdot 1 \cdot 7}{(5-1)(5-7)} + \frac{22 \cdot 1 \cdot 5}{(7-1)(7-5)}$$

$$= 2 \cdot 24^{-1} + 18 \cdot (-8)^{-1} + 17 \cdot 12^{-1} = 13 + 21 + 4 = 7.$$

Exercises

SAGE Constructions

SAGE is a computer algebra system providing a common mathematical interface to many common open source computer algebra packages. The SAGE shell is a customised shell `iPython` to the standard `Python` language. The design of SAGE aims to provide an intuitive, mathematically rigorous interface to a large code base of algorithms. Its name is an acronym for

```
Software for Algebra and Geometry Experimentation.
```

In this course we use SAGE to experiment with cryptographic algorithms and constructions with a custom cryptosystem package.

Obtaining SAGE

SAGE is a freely available computer algebra system which can be downloaded from any of the following mirrors:

1. http://sage.math.washington.edu/sage
2. http://modular.fas.harvard.edu/sage
3. http://echidna.maths.usyd.edu.au/sage
4. http://sage.scipy.org/sage
5. http://cocoa.mathematik.uni-dortmund.de/sage

The SAGE Shell

To start the SAGE shell, just type:

```
> sage
```

You should see something like:

```
> sage
--------------------------------------------------------
| SAGE Version 1.5.3, Build Date: 2007-01-05            |
| Distributed under the GNU General Public License V2.  |
--------------------------------------------------------



sage:
```

Entering commands.

Basic types like integers and rational numbers are built in and can be typed directly at the command line.

```
sage: 1 + 1
2
sage: (2^64+1).factor()
274177 * 67280421310721
sage: 231/23 * 2/55
42/115
```

(The symbol `sage:` is the prompt and not typed.) Basic operations such as `+`, `-`, `*`, `/` invoke underlying commands in the SAGE library of functions (with automatically recognition of the types of the arguments and application the correct algorithm which applies). More advanced functions like `factor` are also linked into the underlying code base of SAGE.

Types and Parents

Every object in SAGE have a `type` and a `parent`. The type may be either a standard Python classes or a class implemented in SAGE.

Appendix A. SAGE Constructions

```
sage: x = 2
sage: type(x)
<type 'sage.rings.integer.Integer'>
sage: y = 2/1
sage: type(y)
<type 'sage.rings.rational.Rational'>
sage: z = '2'
sage: type(z)
<type 'str'>
```

The class of an object determines which functions apply to it. The printing of an object does not in general determine the object itself.

Each class of object has special *member functions* which apply to it; the result depends on the function definition. A member function is called by typing the object name, a '.', followed by the function name and any arguments it takes in parentheses. For instance, the integer 2 is not a unit in **Z**, since there is no element $1/2$ in **Z**. In contrast, the element 2 as a rational number is a unit.

```
sage: x.is_unit()
False
sage: y.is_unit()
True
```

An easy way to determine what member functions exist is to type the first few characters and a TAB to see what functions complete it:

```
sage: x.is
x.is_nilpotent    x.is_prime       x.is_squarefree  x.is_zero
x.is_one          x.is_square      x.is_unit        x.isqrt
sage: x.is
```

To determine what a function does, type (the object, '.', and) the function name, followed by a ? (and return):

```
sage: x.is_unit?
Type:           builtin_function_or_method
Base Class:     <type 'builtin_function_or_method'>
String Form:    <built-in method is_unit of sage.rings.integer.Integer object at 0x8667
Namespace:      Interactive
Docstring:

        Returns true if this integer is a unit, i.e., 1 or -1.

        EXAMPLES:
            sage: for n in srange(-2,3):
                      print n, n.is_unit()
            -2 False
            -1 True
            0 False
            1 True
            2 False
```

Typing a double question mark ?? returns the same documentation string *plus* the source code which implements the function in Python.

The parent of an object is itself a object in SAGE, which represents the mathematical structure to which an object belongs:

```
sage: parent(x)
Integer Ring
sage: parent(y)
Rational Field
```

These parent classes admit their own member functions:

```
sage: ZZ = parent(x)
sage: ZZ.is_field()
False
sage: QQ = parent(y)
sage: QQ.is_field()
True
```

Assignment and output

As we have seen, the assignment operator is =. To print an object in SAGE, just type it

at the command line:

```
sage: x = 2
sage: y = 2/1
sage: z = '2'
sage: x
2
sage: y
2
sage: z
'2'
```

The operator == tests *mathematical* equality, which may involve an evaluation of one object in the parent of the other to carry out the comparison.

```
> x == y
True
> z == y
False
```

In the first line above, the integer x is interpretted as a rational number and found to be equal to y (and vice versa).

Booleans and boolean operators

The boolean truth values `True` and `False` have their own types in SAGE (in fact in Python), which take the special binary boolean operators `and`, `or`, and unary operator `not`.

```
sage: a = True
sage: type(a)
<type 'bool'>
sage: b = False
sage: a and b
False
sage: a or b
True
sage: not a
False
```

Lists, tuples, sets, and dictionaries

Python (hence SAGE) has useful datastructures called lists, tuples, and dictionaries which can be used to collect objects in SAGE.

```
sage: type([])
<type 'list'>
sage: type(())
<type 'tuple'>
sage: type({})
<type 'dict'>
```

The list and tuple types collects sequences of data which is indexed like strings:

```
sage: s = [ 16, 9, 4, 1, 0, 1, 4, 9, 16 ]
sage: t = ( 16, 9, 4, 1, 0, 1, 4, 9, 16 )
sage: [ s[i] == t[i] for i in range(9) ]
[True, True, True, True, True, True, True, True, True]
```

Note that all strings, lists, and tuples are indexed from 0 and range(n) returns the list of elements from 0 to 9:

```
sage: range(9)
[0, 1, 2, 3, 4, 5, 6, 7, 8]
```

The main distinction is that tuples are *immutable*:

```
sage: t[0] = 4
----------------------------------------------------------------------------
<type 'exceptions.TypeError'>              Traceback (most recent call last)

...

<type 'exceptions.TypeError'>: 'tuple' object does not support item assignment
```

while list elements can be reassigned:

```
sage: s[0] = 4
sage: s[8] = 4
sage: s
[4, 9, 4, 1, 0, 1, 4, 9, 4]
```

A set in SAGE (i.e. Python) represents a mathematical set – an unordered collection of objects with each object represented only once.

```
sage: s = set([ 16, 9, 4, 1, 0, 1, 4, 9, 16 ])
sage: s
set([16, 9, 4, 0, 1])
sage: 16 in s
True
sage: len(s) # the cardinality of the set
5
```

The set type has very fast hashed lookup, so that membership test is efficient for sets of large (finite) cardinality.

A dictionary is a useful tool for specifying a lookup table of data indexed by *keys*:

```
sage: u = { 0 : 'yes', 1 : 'no', 2 : 'no', 3 : 'yes' }
sage: u[0]
'yes'
sage: u[-1] = 'no'
sage: u[-2] = 'no'
sage: u[-3] = 'yes'
sage: u
{0: 'yes', 1: 'no', 2: 'no', 3: 'yes', -1: 'no', -3: 'yes', -2: 'no'}
sage: u.keys()
[0, 1, 2, 3, -1, -3, -2]
```

Loops and flow control

Recall that `range(n)` returns the sequence of integers (actually Python `int`'s) from `0` up to `n`. We demonstrate the use of `for` and `if` loops by printing the elements in `0,..,12` which are coprime to `12`:

```
sage: n = 12
sage: for i in range(n):
....:     if gcd(i,n) == 1:
....:         print i # now hit enter twice
....:
1
5
7
11
```

Putting together our use of sets, we demonstrate the use of `if` and `while` loops to construct the same set of integers in `0,..,n`:

```
sage: n = 12
sage: r = euler_phi(n)
sage: i = 1
sage: S = set([i])
sage: while len(S) < r:
....:     i += 1
....:     if GCD(i,n) == 1:
....:         S.add(i) # hit enter twice
....:
sage: S
set([1, 11, 5, 7])
```

In each instance the indentation level is crucial to determine the limits of the loops, which can consist of several lines of commands.

Python Strings

A useful Python types for this course will be strings, which can be created by enclosing input in double quotes *or* in single quotes.

```
sage: S = 'This\n  is\n    a \n    string'
sage: S
'This\n  is\n    a \n    string'
sage: print S
This
  is
    a
     string
sage: T = "This\n  is\n    a \n    string"
sage: T
'This\n  is\n    a \n    string'
sage: S == T
True
```

As seen above, a string can contain newline characters and spaces. The two characters \\, ", and ' have special functionality, and must be typed as \\\\, \\", and \\', respectively (not quite `True`). The newline string can be created directly by `"\n"` or carriage return `"\r"`. Strings are like lists of characters, for which `S[i]` gives access to the `i`-th character of string `S`. The member function `join` can be used to concatenate strings, but must be applied to a string object! We demonstrate the use of these operators in the following construction:

```
sage: s = "But Angie, Angie, ain't it time we said good-bye?\n"
sage: t = "With no loving in our souls "
sage: u = "and no money in our coats\n"
sage: v = "You can't say we're satisfied\n\n"
sage: w = "...they can't say we never tried"
sage: null = ''
sage: angie = null.join([s,t,u,v,w])
sage: print angie
But Angie, Angie, ain't it time we said good-bye?
With no loving in our souls and no money in our coats
You can't say we're satisfied
<BLANKLINE>
...they can't say we never tried
```

and subsequently we cat deconstruct and reassemble our strings:

```
sage: I = [55+i for i in range(3)] + [124 + i for i in range(6)]
sage: I += [4,143,138,164,56,55]
sage: print null.join([ angie[i] for i in I ])
no satisfAction
```

SAGE Cryptosystems

String monoids

The main classes of string monoids are classical alphabetic string monoids, binary, and hexadecimal string monoids.

```
sage: S = AlphabeticStrings()
sage: S
Free alphabetic string monoid on A-Z
sage: H = HexadecimalStrings()
sage: H
Free hexadecimal string monoid
sage: B = BinaryStrings()
sage: B
Free binary string monoid
```

Elements of these strings can be created either by accessing the generators:

```
sage: S.gens()
(A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z)
sage: B.gens()
(0, 1)
sage: H.gens()
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f)
```

For example

```
sage: I = [7, 4, 11, 11, 14, 22, 14, 17, 11, 3]
sage: hello = S('')
sage: for i in I:
....:    hello *= S.gen(i)
....:
....:
sage: hello
HELLOWORLD
```

Alternatively we can either recognize a Python string in the given string monoid:

```
sage: S('ABC')
ABC
sage: H('0a91')
0a91
sage: B('0110')
0110
```

or use standard encodings to map ASCII strings to the monoid:

```
sage: S.encoding('abc')
ABC
sage: H.encoding('abc')
616263
sage: B.encoding('abc')
011000010110001001100011
```

Note that the first construction gives a non-injective map from ASCII strings to upper case (stripping away non-alphabetic characters by mapping them to the empty string). The latter two give the hexadecimal and binary encodings of the underlying ASCII bytes for the characters. These altter are injective maps:

```
sage: S.encoding('abc').decoding()
ABC
sage: H.encoding('abc').decoding()
abc
sage: B.encoding('abc').decoding()
abc
```

Cryptosystems

Specific cryptosystem can be created in SAGE with the following commands:

```
sage: S = AlphabeticStrings()
sage: S
Free alphabetic string monoid on A-Z
sage: E = SubstitutionCryptosystem(S)
sage: T = TranspositionCryptosystem(S,15)
sage: T
Transposition cryptosystem on Free alphabetic string monoid on A-Z
of block length 15
```

The latter constructor (of the transposition cryptosystem) specifies a key length of 15 characters. The SAGE cryptosystem represents a ...

```
sage: K = E.random_key()
sage: K
HYTIKQRWXUPZBJSCANEFODLVMG # random
```

```
sage: K = S('HYTIKQRWXUPZBJSCANEFODLVMG')
sage: e = E(K)
sage: e
HYTIKQRWXUPZBJSCANEFODLVMG
sage: e(S('THECATINTHEHAT'))
FWKTHFXJFWKWHF
```

```
sage: m = E.encoding("This is sample message text to be encoded.")
sage: m
THISISSAMPLEMESSAGETEXTTOBEENCODED
sage: c = e(m)
sage: c
FWXEXEEHBCZKBKEEHRKFKVFFSYKKJTSIKI
sage: L = E.inverse_key(K)
sage: E.enciphering(L,c)
THISISSAMPLEMESSAGETEXTTOBEENCODED
sage: E.deciphering(K,c)
THISISSAMPLEMESSAGETEXTTOBEENCODED
```

Exercises

Read over the above SAGE tutorial and become familiar with the concepts of type, parent, assignment, basic constructions of integers, rationals, and strings, and with simple looping and boolean operations.

Exercise B.1 *For the function* `strip_encoding`*, type the function name followed by a* ? *at the* SAGE *prompt to display the docstring for the function. Do the same for*

$$\text{SubstitutionCryptosystem } \textit{and} \text{ TranspositionCryptosystem.}$$

What are the components of this information, and what does it tell you?

Create a cryptosystem `E` *and type*

$$\text{E.encoding?, E.random\_key? } \textit{and} \text{ E.enciphering?}$$

to read the corresponding docstrings.

Solution.   The docstring details of the path for the function and a description of its use.

Exercise B.2 *Create the string in* SAGE

"I am standing up at the water's edge in my dream"

and assign it to a variable W*. Next create the string monoid* `S` *of* `AlphabeticStrings`*. Apply the member function* `encoding` *to* W*, while reassign* W *to be the output. What is the encoded plaintext that you obtain?*

Solution.   The command encoding gives `IAMSTANDINGUPATTHEWATERSEDGEINMYDREAM`.

Exercise B.3 *Define* K *to be the output of* `E.random_key()` *for a substitution cryptosystem. What is your key? Using the cipher* `E(K)`*, to find the enciphering of* W *with respect to the key* K*.*

Solution.   The substitution key `UVLOIDTGKYZCRHBPMJQWXNFSAE` enciphers the above plaintext as `KURQWUHOKHTXPUWWGIFUWIJQIOTIKHRAOJIUR`.

Exercise B.4 *Show that the deciphering map with respect to* K *is also a simple substitution. What is the inverse substitution key with respect to your particular* K*? Verify this by creating the inverse key and enciphering the ciphertext with respect to it.*

Solution. The inverse of the above substitution key is

YOLFZWHNERICQVDPSMXGABTUJK.

This can be verified by the following lines in SAGE:

```
sage: S = AlphabeticStrings()
sage: K = S('UVLOIDTGKYZCRHBPMJQWXNFSAE')
sage: E = SubstitutionCryptosystem(S)
sage: E.inverse_key(K)
YOLFZWHNERICQVDPSMXGABTUJK
sage: L = E.inverse_key(K)
sage: E.en
E.enciphering   E.encoding
sage: E.enciphering(K,L)
ABCDEFGHIJKLMNOPQRSTUVWXYZ
sage: E.enciphering(L,K)
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

Note that ABCDEFGHIJKLMNOPQRSTUVWXYZ is the identity substitution key.

Solutions to Exercises

Introduction to Cryptography

Classical Cryptography

Substitution ciphers

Exercise 2.1 *Determine the number of possible keys for the affine substitution ciphers. Is this sufficient to have a secure cryptosystem?*

Transposition ciphers

Exercise 2.2 *Show that for every π in S_n, there exists an positive integer m, such that π^m is the identity map, and such that m divides $n!$. The smallest such m is called the order of π.*

Exercise 2.3 *How many transpositions exist in S_n? Describe the elements of order 2 in S_n and determine their number.*

Exercise 2.4 *Show that every element of S_n can be expressed as the composition of at most n transpositions.*

Exercise 2.5 *What is the order of a permutation with cycle lengths $d_1, \ldots, d_t$? How does this solve the previous exercise concerning the order of a permutation?*

Exercise 2.6 *What is the block length m of an (r, s)-simple columnar transposition? Describe the permutation. Hint: it may be easier to describe the permutation if the index set is $\{0, \ldots, m-1\}$.*

Solution. The block length is the number of characters which are involved in each permutation, which equals rs for an (r,s)-simple columnar transposition. In terms of maps of indices, the k-th position maps to (i,j) in an array where $i = ((k-1) \bmod s) + 1$ and $j = (k-1) \operatorname{div} s$ (for k in $1, \ldots, rs$). The transpose maps (i,j) to (j,i), which goes to the new position $\pi(k) = i + (j-1)r$ in the ciphertext. The map $k \mapsto \pi(k)$ determines the *inverse* of the permutation on indices. The inverse permutation is determined by exchanging the roles of r and s.

Exercise 2.7 *Show that the (r,r)-simple columnar transposition has order 2. What is the order of the cipher for $(r,s) = (3,5)$? Determine the permutation in cycle notation for this cipher. Determine the permutation in cycle notation for the $(7,36)$-simple columnar transposition used in this chapter.*

Solution. For a (r,r)-simple columnar transposition one writes each r^2-block into an $r \times r$ array, applies a transposition, and reads the columns off as rows. It has order 2 since two-fold application of the cipher acts as the identity on every r^2-block.

In general, an (r,s)-simple columnar tranposition cipher does not have order 2 since the transpose matrix does not have the same form, so r characters are read from each column, while s characters are written into each row at the next application of the cipher. The inverse of an (r,s)-simple columnar transposition is an (s,r)-simple columnar transposition.

The $(3,5)$-simple columnar transposition is determined by the following map of indices

$$1 \mapsto 1, \ 2 \mapsto 6, \ 3 \mapsto 11, \ldots$$

giving the map in list notation

$$[1, 6, 11, 2, 7, 12, 3, 8, 13, 4, 9, 14, 5, 10, 15]$$

In SAGE we can construct this sequence and determine the cycle notation for the permuation as follows:

```
sage: (r,s) = (3,5)
sage: G = SymmetricGroup(r*s)
sage: S = [ i+s*j for i in range(1,s+1) for j in range(r) ]
sage: S
[1, 6, 11, 2, 7, 12, 3, 8, 13, 4, 9, 14, 5, 10, 15]
sage: G(S)
(2,6,12,14,10,4)(3,11,9,13,5,7)
```

The equivalent construction for $(r,s) = (7,36)$ follows.

```
sage: (r,s) = (7,5)
sage: G = SymmetricGroup(r*s)
sage: S = [ i+s*j for i in range(1,s+1) for j in range(r) ]
sage: G(S)
(2,6,26,24,14,32,20,28,34,30,10,12,22,4,16,8)
  (3,11,17,13,27,29,5,21,33,25,19,23,9,7,31,15)
```

Elementary Cryptanalysis

One important measure of a cryptographic text is the *coincidence index*. For random text (of uniformly distributed characters) in an alphabet of size 26, the coincidence index is approximately 0.0385. For English text, this value is closer to 0.0661. Therefore we should be able to pick out text which is a simple substitution or a transposition of English text, since the a coincidence index remains unchanged.

The SAGE crypto string functions

<p style="text-align:center">coincidence_index and frequency_distribution</p>

provide functionality for analysis of the ciphertexts in the exercises. Moreover, for a SAGE string s the k-th *decimation* of period m for that string is given by s[k::m] (short for s[k:len(s):m]).

Exercise 3.1 *Complete the deciphering of the Vigenère ciphertext of Section 3.3 . What do you note about the relation between the text and the enciphering or deciphering key? A useful tool for this task could be the following javascript application for analyzing Vigenère ciphers:*

<p style="text-align:center">http://echidna.maths.usyd.edu.au/ kohel/tch/Crypto/vigenere.html</p>

Consider those ciphertexts from previous exercises which come from a Vigenère cipher, and determine the periods and keys for each of the ciphertext samples.

Exercise 3.2 *For each of the cryptographic texts from the course web page, compute the coincidence index of the ciphertexts. Can you tell which come from simple substitution or transposition ciphers? How could you distinguish the two?*

Solution. The coincidence index for each of the ciphertext samples is given in the table below.

1. 0.0438722554890219
2. 0.0657023320387985
3. 0.0447239692522711
4. 0.0629545310820211
5. 0.0412801243845555
6. 0.0655225068362645
7. 0.0412339115637173
8. 0.0674918061066068
9. 0.0685573482676622
10. 0.0657341758094024
11. 0.0665847779993272

All but ciphertexts 1, 3, 5 and 7 are consistent with output from a simple substitution or transposition cipher. It is likely that the exceptional ones employ a polyalphabetic cipher. In order to distinguish substitution and transposition ciphers, it is necessary to look at character distributions, e.g. a close match with the frequency distribution of English (or a modern language) suggests a transposition cipher.

Exercise 3.3 *For each of the cryptographic texts from the course web page, for various periods extract the substrings of $im + j$-th characters. For those which are not simple substitutions, can you identify a period?*

Solution. Using the average coincidence index of the ciphertext decimations, we find that the periods of the ciphertexts 1, 3, 5, and 7 are 11, 6, 14, and 9, respectively. The code to verify this is:

```
sage: ct05 = strip_encoding(open("Ciphertext/cipher05.txt").read())
sage: n = len(ct05)
sage: for m in range(1,10):
          cis = [ ct05[i:n:m].coincidence_index() for i in range(m) ]
          print "%s : %s" % (m, sum(cis)/m)
```

with output:

```
1 : 0.0412801243845555
2 : 0.0424744413115725
3 : 0.0411679856535807
4 : 0.0432876773672201
5 : 0.0406338756102603
6 : 0.0431854895350470
7 : 0.0547229569294580
8 : 0.0428701713243436
9 : 0.0423160002107370
```

Noting the minor peak at $m = 7$, we continue with the loop:

```
sage: for m in range(10,19):
          cis = [ coincidence_index(ct05[i:n:m]) for i in range(m) ]
          print "%s : %s" % (m, sum(cis)/m)
```

to find the true period is $m = 14$:

```
10 : 0.0420455268414958
11 : 0.0418377321603127
12 : 0.0438681932102984
13 : 0.0391335947128655
14 : 0.0728669127225357
15 : 0.0393529497877323
16 : 0.0442427680090754
17 : 0.0417945039178898
18 : 0.0446147902288252
19 : 0.0393532851737185
```

Note that the coincidence index for each even test period is slightly higher, since these involve an averaging over only 7, rather than 14, distinct substitutions.

Exercise 3.4 *For each of the ciphertexts which you have reduced to simple substitutions, consider the frequency distribution of the simple substitution texts. Now recover the keys and original plaintext.*

Solution. The first step in recovering the keys and plaintext is to determine the type of cipher; further techniques are studied in later tutorials. Note that the ciphertexts 1, 3, 5, and 7 are the result of Vigenère cryptosystems, and can be deciphered by statistical

analysis of the each of the decimations with respect to their periods. A javascript program from the course web page can be used for this purpose.

Exercise 3.5 (Correlations of sequence translations) *Suppose that* `pt` *and* `ct` *are plaintext and ciphertext whose frequency distributions are to be compared. Assume we have defined:*

```
sage: S = AlphabeticStrings()
sage: E = SubstitutionCryptosystem(S)
```

The following code finds the correlations between the affine translations of two sequences.

```
sage: X = pt.frequency_distribution()
sage: Z = ct.frequency_distribution()
sage: Y = DiscreteRandomVariable(X,Z.function())
sage: for j in range(26):
...         K = S([ (j+k)%26 for k in range(26) ])
...         print "%s: %s" % (j, X.translation_correlation(Y,E(K)))
```

What does `frequency_distribution` *return, and what are the ciphers e constructed in the* `for` *loop? What does* `translation_correlation` *return? Note that* Y *must be created as a discrete random variable on the probability space* X *in order to compute their correlations.*

Solution. Given two strings $S_1 = $ `pt` and $S_2 = $ `ct` with this computes their frequency distributions, X_1 and X_2 (as discrete probability spaces). At each iteration of the `for` loop, an affine translation cipher (a cyclic shift by k characters) is constructed. Then for such cipher e, the correlation of X_1 with $X_2 \circ e$ is constructed. By comparing a standard plaintext `pt` against affine translations of decimations of ciphertext `ct` we are able to break the Vigenère enciphering in the exercise below.

Exercise 3.6 (Breaking Vigenère ciphers) *A Vigenère cipher is reduced to an translation cipher by the process of decimation. How does the above exercise solve the problem of finding the affine translation?*

Apply this exercise to the Vigenére ciphertext sample `cipher01.txt` *from the course web page, and the break the enciphering. Recall that you will have to use the decimation (by* `ct[i::m]`*) and* `coincidence_index` *to first reduce a Vigenère ciphertext to the output of a monoalphabetic cipher.*

```
sage: X = pt.frequency_distribution()
sage: m = 11
sage: r = 0.75
sage: match = [ [] for i in range(m) ]
sage: for i in range(m):
...      Z = ct[i::m].frequency_distribution()
...      Y = DiscreteRandomVariable(X,Z.function())
...      for j in range(26):
...          K = S([ (j+k)%26 for i in range(26) ])
...          corr = X.translation_correlation(Y,E(K))
...          if corr > r:
...              match[i].append(j)
```

Solution. We have already surmised that the first sample ciphertext, `cipher01.txt`, is output from a Vigenére cipher of period 11. We the definitions as below:

```
sage: S = AlphabeticStrings()
sage: E = SubstitutionCryptosystem(S)
sage: pt = S.encoding(open("Plaintext/blackcat.txt").read())
sage: ct = S.encoding(open("Ciphertext/cipher01.txt").read())
```

the output of the above code gives

```
sage: match
[[], [7], [0], [], [], [18], [3], [4], [0], [], [4]]
```

A translation by 7 corresponds to the character `H`, by 0 to `A`, by 18 to `S`, and by 3 and 4 to `D` and `E`, respectively. This gives the partial enciphering key `*HA**SDEA*E`. Deciphering with respect to this key gives the plaintext blocks `*HE**OETI*I`, `*KT**NPOM*N`, etc.

Relaxing the bound from $r = 0.75$ to 0.50, one finds multiple solutions among them the correct solution `SHAKESPEARE`, giving the plaintext

WHENMOSTIWINKTHENDOMINEEYESBESTSEEFORALLTHEDAYTHEYVIEWTHINGSUNRESP

Why is this a bad key choice?

Exercise 3.7 (Breaking substitution ciphers) *Suppose that rather than an affine translation, you have reduced to an arbitrary simple substitution. We need to undo an*

109

arbitrary permutation of the alphabet. For this purpose we define maps into Euclidean space:

1. $\mathcal{A} \to \mathcal{A}^2 \to \mathbf{R}^2$ *defined by*

$$x \longmapsto xx \longmapsto \big(P(x), P(xx)\big).$$

2. $\mathcal{A} \to \mathcal{A}^2 \to \mathbf{R}^3$ *defined by*

$$x \longmapsto xy \longmapsto \big(P(x), P(xy \,|\, y), P(yx \,|\, y)\big),$$

   for some fixed character y.

See the document

for standard vectors for the English language.

Solution. These maps can be applied to the solution of substitution ciphers by finding nearest elements to a known standard for the English language. For instance, assume that the ciphertext image x of E has been identified, one can look for pairs xy which are the image of the plaintext pair ER, by searching for a nearest vector to:

$$(0.05674, 0.13071, 0.11352).$$

This will determine the ciphertext image y of R. This bootstrapping procedure successively determines the substitution from the digraph frequencies of the ciphertext.

Exercise 3.8 (Breaking transposition ciphers) *In order to break transposition ciphers it is necessary to find the period m, of the cipher, and then to identify positions i and j within each block $1 + km \le i, j \le (k+1)m$ which were adjacent prior to the permutation of positions. Suppose we guess that m is the correct period. Then for a ciphertext sample $C = c_1 c_2 \ldots$, and a choice of $1 \le i < j \le m$, we can form the digraph decimation sequence $c_i c_j, c_{i+m} c_{j+m}, c_{i+2m} c_{j+2m}, \ldots$.*

Two statistical measures that we can use on ciphertext to determine if a digraph sequence is typical of the English language are a digraph coincidence index

$$\sum_{x \in \mathcal{A}}^{n} \sum_{y \in \mathcal{A}}^{n} \frac{n_{xy}(n_{xy} - 1)}{N(N-1)}$$

where N is the total number of character pairs, and n_{xy} is the number of occurrences of the pair xy, and the coincidence discriminant:

$$\sum_{x \in \mathcal{A}} \sum_{y \in \mathcal{A}} \left(\frac{n_{xy}}{N} - \left(\sum_{z \in \mathcal{A}} \frac{n_{xz}}{N}\right)\left(\sum_{z \in \mathcal{A}} \frac{n_{zy}}{N}\right) \right)^2.$$

The first term is the frequency of xy, and the latter is the product over the frequencies of x as a first character and y as a second character. The coincidence discriminant measures the discrepancy between the probability space of pairs xy and the product probability space.

What behavior do you expect for the coincidence index and coincidence discriminant of the above digraph decimation, if i and j were the positions of originally adjacent characters? Test your hypotheses with decimations of "real" English text, using the SAGE implementations of `coincidence_index` *and* `coincidence_discriminant`.

Why can we assume that $i < j$ in the digraph sequence? What is the obstacle to extending these statistical measures from two to more characters?

Solution. If i and j are the ciphertext images of adjacent positions k, $k+1$, in each block of length m, then the sequence

$$c_i c_j, \ c_{i+m} c_{j+m}, \ c_{i+2m} c_{j+2m}, \ldots$$

will have the coincidence index and coincidence discriminant of the plaintext. Note that the measures are invariant under a substitution, so can be used to break a combination substitution-transposition cipher, by first breaking the transposition. The result will be a sequence $i_1, i_2, \ldots, i_m$ of indices of positions which "want" to be associated.

Note that the measures, coicidence index and coincidence discriminant, will also be the same for the sequence

$$c_j c_i, \ c_{j+m} c_{i+m}, \ c_{j+2m} c_{i+2m}, \ldots$$

so we do not directly distinguish the correct order from its reverse, $i_m, \ldots, i_2, i_1$. This one bit of information can be determined at the end, with the savings of being able to assume $i < j$ in testing for statistically associated pairs $\{i, j\}$.

Note also that for the incorrect period m there will be little or no tendency for statistical association of characters, so by first varying the triples (i, j, m), with fixed $i = 1$ and $1 < j \le m$, we can determine the probable period m and then recover the entire sequence $i_1, i_2, \ldots i_m$ by letting i vary.

Information Theory

In order to understand naturally occurring languages, we consider the models for finite languages X consisting of strings of fixed finite length N together with a probability function P which models the natural language. In what follows, for two strings x and y we denote their concatenation by xy.

Exercise 4.1 *Show that N is the maximum entropy for a probability function on bit strings of length N.*

Solution. None provided.

Exercise 4.2 *Show that the rate of a uniform probability space is 1 and that this is the maximal value for any probability space.*

Solution. None provided.

Exercise 4.3 *For a given cryptosystem, show that the definition*

$$P(y) = \sum_{K \in \mathcal{K}} P(K) \sum_{\substack{x \in \mathcal{M} \\ E_K(x)=y}} P(x).$$

determines a probability function on the ciphertext space. Then verify the equalities:

$$P(y) = \sum_{x \in \mathcal{M}} P(x,y), \quad and \quad P(x) = \sum_{y \in \mathcal{C}} P(x,y).$$

Solution. None provided.

Exercise 4.4 *Consider the language of 1-character strings over $\{\mathtt{A}, \mathtt{B}, \mathtt{C}, \mathtt{D}\}$ with associated probabilities 1/3, 1/12, 1/4, and 1/3. What is its corresponding entropy?*

Solution. *The entropy of the language is*

$$\frac{1}{3}\log_2(3) + \frac{1}{12}\log_2(12) + \frac{1}{4}\log_2(4) + \frac{1}{3}\log_2(3)$$

$$= \frac{2}{3}\log_2(3) + \frac{1}{12}(2 + \log_2(3)) + \frac{1}{2} = \frac{2}{3} + \frac{3}{4}\log_2(3),$$

which is approximately 1.855.

Exercise 4.5 *Consider the language X^2 of all strings of length 2 in $\{\mathtt{A}, \mathtt{B}, \mathtt{C}, \mathtt{D}\}$ defined by the probability function of Exercise 1 and 2-character independence: $P(xy) = P(x)P(y)$. What is the entropy of this language?*

Solution. By rearranging the sum

$$\sum_{x \in X} \sum_{y \in X} P(xy) \log_2(P(xy))$$
$$= \sum_{x \in X} \sum_{y \in X} P(x)P(y)\big(\log_2 P(x) + \log_2 P(y)\big)$$

one finds the entropy to be double that of Exercise 1, or about 3.711. This is consistent with the interpretation of the entropy as the length of a random element of a language in some theoretically optimal encoding.

Exercise 4.6 *Let $\mathcal{M}$ be the strings of length 2 over $\{$A, B, C, D$\}$ with the following frequency distribution:*

$$
\begin{array}{llll}
P(\text{AA}) = 5/36 & P(\text{BA}) = 0 & P(\text{CA}) = 1/12 & P(\text{DA}) = 1/9 \\
P(\text{AB}) = 1/36 & P(\text{BB}) = 1/144 & P(\text{CB}) = 1/48 & P(\text{DB}) = 1/36 \\
P(\text{AC}) = 7/72 & P(\text{BC}) = 1/48 & P(\text{CC}) = 1/16 & P(\text{DC}) = 5/72 \\
P(\text{AD}) = 5/72 & P(\text{BD}) = 1/18 & P(\text{CD}) = 1/12 & P(\text{DD}) = 1/8
\end{array}
$$

Show that the 1-character frequencies in this language are the same as for the language in Exercise 2.

Solution. The 1-character frequencies can be defined as the average of the character frequencies in the 1st and 2nd positions, but these turn out to be the same for each character, and agree witht the frequencies of Exercise 1.

Exercise 4.7 *Do you expect the entropy of the language of Exercise 3 to be greater or less than that of Exercise 2? What is the entropy of each language?*

Solution. The entropy of the language of Exercise 3 is approximately 3.633, compared to an entropy of about 3.711 for that of Exercise 2.

The language of Exercise 2 is the most random space with given 1 character frequences. The lower entropy in Exercise 3 could have been predicted since the probabilities agrees with the 1 character frequencies, while additional structure (less uncertainty) is built into the 2 character probabilities, since in general $P(\text{XY}) \neq P(\text{YX})$.

Exercise 4.8 *Consider the infinite language of all strings over the alphabet $\{$A$\}$, with probability function defined such that $P(\text{A}\ldots\text{A}) = 1/2^n$, where n is the length of the string $\text{A}\ldots\text{A}$. Show that the entropy of this language is 2.*

Solution. One must verify the equality

$$\sum_{n=1}^{\infty} \frac{1}{2^n} \log_2(2^n) = \sum_{n=1}^{\infty} \frac{n}{2^n} = 2.$$

We do this by first verifying the equality

$$\sum_{n=0}^{\infty} \frac{1}{2^n} + \sum_{n=1}^{\infty} \frac{n}{2^n} = 2\sum_{n=1}^{\infty} \frac{n}{2^n},$$

together with the standard identity $\sum_{n=0}^{\infty} 1/2^n = 2$.

Block Ciphers

We summarise the modes of operation covered in this chapter.

Electronic Codebook Mode. For a fixed key K, the output ciphertext is given by $C_j = E_K(M_j)$ with output $C_1 C_2 \ldots$.

Ciphertext Block Chaining Mode. For input key K, and initialization vector $IV = C_0$, the output ciphertext is given by $C_j = E_K(C_{j-1} \oplus M_j)$, with output $C_0 C_1 C_2 \ldots$.

Ciphertext Feedback Mode. Given plaintext $M_1 M_2 \ldots$ in r-bit blocks, a key K, an n-bit cipher E_K, and an n-bit initialization vector $IV = I_1$, the ciphertext is computed as:

$$C_j = M_j \oplus L_r(E_K(I_j))$$
$$I_{j+1} = R_{n-r}(I_j) \,||\, C_j$$

where R_{n-r} and L_r are the operators which take the right-most $n-r$ bits and the left-most r bits, respectively, and $||$ is concatenation.

Output Feedback Mode. Given plaintext $M_1 M_2 \ldots$ in r-bit blocks, a key K, an n-bit cipher E_K, and an n-bit initialization vector $IV = I_0$, the ciphertext is computed as:

$$I_j = E_K(I_{j-1})$$
$$C_j = M_j \oplus L_r(I_j),$$

where L_r is the operator which takes the left-most r bits.

Exercise 5.1 *What mode of operation has been used in the assignment and in class up to this point, and why? What are the security disadvantages of this mode of operation?*

Solution. Electronic codebook mode – it leaves the cipher most open to analysis of its statistical properties, so that we can demonstrate the methods to crack it. It is also the most natural and naïve way to apply a block cipher.

Exercise 5.2 *Let E_K be the 4-bit cipher defined by:*

$$E_K(M) = (K \oplus M) \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix} = (X_1 + X_3, X_2 + X_4, X_2 + X_3, X_1 + X_4)$$

where $X = K \oplus M = (X_1, X_2, X_3, X_4)$. Encipher the message M given by

$$11010110111001110010010001001000,$$

using the key $K = 1011$, in (i) ECB mode, in (ii) CBC mode with initialization vector 1001, and in (iii) CFB mode with initialization vector 1001 and $r = 1$.

Solution. The ciphertext output for each of ECB, CBC, and CFB modes is:

ECB mode:	11001010001111111100000000001111
CBC mode:	00001010000011111100111100001111
CFB mode:	00000111010000111110001011010001

Here the leading initialization vector 1001 is omitted in the CBC output.

Exercise 5.3 *How many steps are required for error recovery from a ciphertext transmission error in ECB and CBC modes?*

Solution. The blocks in ECB mode are independent, so error recovery is immediate, i.e. an error affects only the block in which it occurs. In CBC mode recovery from erros occurs after two blocks.

Exercise 5.4 *If $n = 64$ and $r = 8$, how many steps in CFB mode does it take to recover from an error in a ciphertext block? What about in OFB mode?*

Solution. Recovery in CFB mode occurs after $[n/r] = 64/8 = 8$ blocks. In OFB mode recovery is immediate, provided synchronization is not lost.

Stream Ciphers

Exercise 6.1 *Identify each of the key stream, output, and next state functions for the synchronous stream cipher determined by a block cipher in OFB mode of operation.*

Solution. None provided.

Exercise 6.2 *Identify the the key stream and output functions for the asynchronizing stream cipher determined by a block cipher in each of CBC and 1-bit CFB modes.*
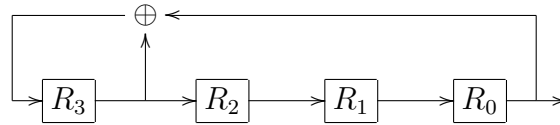
Solution. None provided.

Linear feedback shift registers (LFSR's) are an efficient way of describing and generating certain sequences in hardware implementations. We derive and work with equivalent mathematical descriptions of the sequences produced by a LFSR, along with some generalized sequences which do not arise in this way.

A linear feedback shift register is composed of a shift register R which contains a sequence of bits and a feedback function f which is the bit sum (`xor`) of a subset of the entries of the shift register. The shift register contains n memory cells, or stages, labelled $R_{n-1}, \ldots, R_1, R_0$, each holding one bit. Each time a bit is needed the entry in stage R_0 is output while the entry in cell R_i is passed to cell R_{i-1} and the top stage R_{n-1} is updated with the value $f(R)$.

Exercise 6.3 *Consider the following schematic of a linear feedback shift register:*

Let the initial entries of stages R_i be s_i, for $0 \le i \le n$. For each of the following initial entries below:

	s_3	s_2	s_1	s_0
a)	0	1	1	0
b)	1	1	1	0
c)	1	0	1	0
d)	1	1	0	0

compute the first 16 bits in the output sequence. Show that the output sequence is defined by the initial entries and the recursion $s_{i+4} = s_{i+3} + s_i$.

Solution. The recursion $s_{i+4} = s_{i+3} + s_i$ is immediately apparent as that specified by the diagram of the LFSR. From this recursion, the given initial states expand to the following sequences:

a)	`0110010001111010...`
b)	`1110101100100011...`
c)	`1010110010001111...`
d)	`1100100011110101...`

Exercise 6.4 *Show that every linear feedback register defines and is defined by a recursion of the form $s_{i+n} = \sum_{j=0}^{n-1} c_j s_{i+j}$, where the c_j are bits in $\mathbf{Z}/2\mathbf{Z}$; the products $c_j s_{i+j}$ and the summation are operations in $\mathbf{Z}/2\mathbf{Z}$.*

N.B. The ring $\mathbf{Z}/2\mathbf{Z}$ is also referred to as $\mathbf{F}_2$, the unique finite field of two elements. Note that the addition operation is the same `xor` that we have been using and the multiplication operation is the logical `and` operation.)

Solution. The data of a LFSR diagram, of a linear recurrences relation, and of a connection polynomial are equivalent — they express the same information. The connection polynomial $g(x) = \sum_j c_j x^j$ encodes the wiring of a LFSR which implements a recurrence relation. Thinking of x^j as a shift operator acting on the sequence $s_0, s_1, s_2, \ldots$, the behaviour of $g(x)$ in the product $g(x)s(x)$ (below) is precisely this recurrence relation.

Exercise 6.5 *For a linear feedback register of length n, define a power series*

$$s(x) = \sum_{i=1}^{\infty} s_i x^i$$

from the output sequence s_i. Suppose that the linear feedback register defines the recursion $s_{i+n} = \sum_{j=0}^{n-1} c_{n-j} s_{i+j}$. Define a polynomial $g(x) = 1 + \sum_{j=1}^{n-1} c_j x^j$. Show that $f(x) = g(x)s(x)$ is a polynomial, that is, all of its coefficients are eventually zero. What is the polynomial $f(x)$?

Solution. The expression $f(x) = s(x)g(x)$ for a polynomial $f(x)$ of degree less than $n = \deg(g(x))$ is another equivalent formulation of the recurrence relation. The initial n coefficients of $s(x)$ are the entries of the shift register, and the n coefficients of $f(x)$ is a linear combination of these coefficients. Although the coefficients of $f(x)$ are not equal to the initial state, for a nonsingular LFSR, the initial states are in bijection with the numerator polynomials $f(x)$.

Exercise 6.6 *In the previous exercise we showed that the power series $s(x)$ has the form $f(x)/g(x)$ in the power series ring $\mathbf{F}_2[[x]]$. In* SAGE *it is possible to form power series rings in the following way*

```
sage: F2 = FiniteField(2)
sage: PS.<x> = PowerSeriesRing(F2)
sage: f = x^2 + x
sage: g = x^3 + x + 1
sage: f/g + O(x^16)
x + x^4 + x^5 + x^6 + x^8 + x^11 + x^12 + x^13 + x^15 + O(x^16)
```

Consider the linear feedback shift register at the beginning of the worksheet. Construct the corresponding power series and verify that these are the same of the output sequences that you computed.

Solution. The power series expansions of the first question are:

$$s(x) = x + x^2 + x^5 + x^9 + x^{10} + x^{11} + x^{12} + x^{14} + \cdots$$
$$s(x) = 1 + x + x^2 + x^4 + x^6 + x^7 + x^{10} + x^{14} + x^{15} + \cdots$$
$$s(x) = 1 + x^2 + x^4 + x^5 + x^8 + x^{12} + x^{13} + x^{14} + x^{15} + \cdots$$
$$s(x) = 1 + x + x^4 + x^8 + x^9 + x^{10} + x^{11} + x^{13} + x^{15} + \cdots$$

Multiplying each through by the connection polynomial $g(x) = x^4 + x + 1$, we find the numerator polynomials for each of the sequences:

$$f(x) = x + x^3$$
$$f(x) = 1 + x^3$$
$$f(x) = 1 + x + x^2 + x^3$$
$$f(x) = 1 + x^2$$

This verifies that the sequences output are consistent with their expected structure as coefficients of a rational power series.

Statistical Properties. The output of a linear feedback shift operator of length n has a period, which must divide $2^n - 1$. The period is independent of the initial state, provided it is non-zero. If the period equals $2^n - 1$, the output sequence is said to be an m-sequence. The following theorem describes the statistical properties of m-sequences.

Theorem 6.1 *Let $s = s_0 s_1 \ldots$ be an m-sequence, and let k be an integer with $1 \le k \le n$. Then in each subsequence of s of length $2^n + k - 2$, every finite nonzero binary sequence of length k appears as a subsequence exactly 2^{n-k} times, and the length k zero subsequence appears exactly $2^{n-k} - 1$ times.*

A polynomial $g(x)$ in $\mathbf{F}_2[x]$ is *irreducible* if it is not the product of two polynomials of degree greater than zero. An irreducible polynomial $g(x)$ of degree n is *primitive* if $g(x)$ divides $x^N - 1$ for $N = 2^n - 1$ and no smaller value of N. (Equivalently, $g(x)$ is primitive if the powers x^i are distinct modulo $g(x)$, for i with $1 \le i \le 2^n - 1$.)

Linear Complexity. A linear feedback shift register is said to generate a binary sequence s if there exists some initial state for which its output sequence is s. The linear complexity $L(s)$ of an infinite sequence s is defined to be zero if s is the zero sequence, infinity if s is generated by no linear feedback shift register, and otherwise equal to the minimal length of a linear feedback shift register generating s. The linear complexity $L(s)$ of a finite sequence s is defined to be the minimal length of a linear feedback shift register with initial sequence s for some initial state.

Linear Complexity Profile. For a sequence s, define $L_j(s)$ to be the linear complexity of the first j terms of the sequence. The linear complexity profile of an infinite sequence s is defined to be the infinite sequence $L_1(s), L_2(s), \ldots$, and for a finite sequence $s = s_0 s_1 \ldots s_{N-1}$ is defined to be the finite sequence $L_1(s), L_2(s), \ldots, L_n(s)$.

LFSR Cryptosystems We introduce new utilities for binary stream cryptosystems based on linear feedback shift registers. The functions `binary_encoding` and `binary_decoding` convert ASCII text into its bit sequence and back. In addition, the new binary cryptosystems are:

LFSRCryptosystem
ShrinkingGeneratorCryptosystem

Unlike the encoding function `strip_encoding` we have used so far, the function `binary_encoding` is information-preserving, taking 8-bit ASCII input and returning the binary encoding string. The inverse function `binary_decoding` recovers the original text.

A linear feedback shift register cryptosystem is created in SAGE using the function `LFSRCryptosystem`, taking no arguments. A key is defined by means of a pair, consisting of the connection polynomial $g(x)$ over $\mathbf{F}_2$ and a initial bit sequence of length equal to the degree of the sequence. A sample use of the cryptosystem follows. The shrinking generator cryptosystem is a cryptosystem based on a pair of LFSR's as defined in class.

```
sage: F2 = FiniteField(2)
sage: P2.<x> = PolynomialRing(F2)
sage: g = x^17 + x^5 + 1
sage: IS = [ F2.random_element() for i in range(17) ]
sage: LFSR = LFSRCryptosystem()
sage: PT = LFSR.encoding("The dog ate my assignment."); PT
0101010001101000011001010010000001100100011011110110011 10
0100000011000010111010001100101001000000110110101111001 00
1000000110000101110011011110011011010010110011101101110011
0110101100101011011100111010000101110
sage: K = (g,IS)
sage: e = LFSR(K)
sage: CT = e(PT)
sage: PT == e(CT)
True
```

Note that the encoding of the message is not ciphertext – this is the standard ASCII bit encoding.

Exercise 6.7 *Consider the coefficient sequence for $f(x)/g(x)$ in $\mathbf{F}_2[[x]]$, where $g(x) = 1 + x + x^4$ and $f(x) = 1 + x^3$. Is $g(x)$ an irreducible polynomial? A primitive polynomial? Draw the associated linear feedback shift register. What is the initial state of the shift register?*

Solution. The polynomial $x^4 + x + 1$ is an irreducible polynomial, which is primitive. The LFSR with this connection polynomial was given in the previous tutorial. The primitivity follows since none of the sequences, computed last week, had a period shorter than 15. The initial state corresponding to the polynomial $f(x) = x^3 + 1$ was the second given value 1110 of the previous tutorial.

Exercise 6.8 *Compute the linear complexity of the sequences* 11, 1011, 10101, 10110, *and* 10011.

Solution. The linear complexity of the sequences 11, 1011, 10101, 10110, and 10011 is 1, 2, 2, 2, and 3. The initial values follow from extending the sequences with period 1, 3, 2, and 3, with connection polynomials $x + 1$, $x^2 + x + 1$, $x^2 + 1$, and $x^2 + x + 1$. The third sequence can be extended to a sequence with period no better than 4, so it generated by no LFSR of length 2. A possible connection polynomial is $x^4 + 1 = (x+1)^4$, giving a LFSR of length 4 which generates it. However, the divisor $x^3 + x^2 + x + 1 = (x+1)^3$ defines a recursion for a LFSR of length 3. Hence the linear complexity for this sequence is 3.

Exercise 6.9 *Compute the first 8 terms of the linear complexity profile of the coefficient sequence from Exercise 1.*

Solution. The first 8 terms of the linear complexity profile for the sequence of the first question are:
$$[1, 1, 1, 3, 3, 3, 4, 4].$$
On the other hand, since the sequence is generated by a LFSR of length 4 we know that the full infinite sequence becomes constant at 4.

Exercise 6.10 *Practice encoding and enciphering with the LFSR stream cryptosystem. The function* `binary_decoding` *easily converts this back to ASCII text. Use these functions to verify that* `PT` *is just the binary encoding of the original plaintext message and that the ciphertext is enciphered.*

Solution. The encoding and decoding member functions associated with a LFSR are just a wrapper around `binary_encoding` and `binary_decoding`:

```
sage: LFSR = LFSRCryptosystem()
sage: PT = LFSR.encoding('The dog ate my assignment.'); PT
01010100011010000110010100100000011001000110111101100111
01000000110000101110100011001010010000001101101011110010
100000011000010111001101111001101101010010110011101101110011
0110101100101011011100111010000101110
sage: LFSR.decoding(PT)
'The dog ate my assignment.'
```

We verify using that `binary_decoding` that the binary string also returns the ASCII message:

```
sage: PT.binary_decoding()
'The dog ate my assignment.'
```

Exercise 6.11 *Since the LFSR is the bitsum of the binary keystream, generated by the connection polynomial and initial state, why must the inverse key be equal to the key itself?*

Solution. Since LFSR ciphertext is the bitsum of plaintext with a keystream, a subsequent bitsum with the same keystream gives the original plaintext:

$$c_i + s_i = (m_i + s_i) + s_i = m_i + (s_i + s_i) = m_i + 0 = m_i,$$

Therefore enciphering map is equal to deciphering map; in particular, the enciphering and deciphering keys are the same.

Elementary Number Theory

Reduction modulo a polynomial $g(x)$ or modulo an integer m plays a central role in the mathematics of cryptography. Recall that for a monic polynomial $g(x)$ of positive degree, we define $a(x) \bmod g(x)$ to the unique polynomial $a_0(x)$ with $\deg a_0(x) < \deg g(x)$ such that

$$a(x) = a_0(x) + a_1(x)g(x).$$

For an integer m, we define $a \bmod m$ to be the unique integer a_0 with $0 \le a_0 < m$ such that $a = a_0 + a_1 m$.

Fermat's little theorem. If p is a prime, then the relation $a^{p-1} \equiv 1 \bmod p$ holds for any integer a not divisible by p.

Note. The SAGE function `mod` operates on integers, with the syntax:

```
sage: m = 101
sage: (2^31).mod(m)
34
```

The same mathematical result can be achieved with the `powermod` function (for modular powering):

```
sage: 2.powermod(31,m)
34
```

The latter construction, however, is more efficient.

Chinese remainder theorem. Let p and q be distinct primes, then for every integer a and b there exists a unique integer c with $0 \le c < pq$ such that $c \equiv a \bmod p$ and $c \equiv b \bmod q$.

If a, b, and c are as above, then for any integral polynomial $f(x)$, the integer $f(c)$ satisfies $f(c) \equiv f(a) \bmod p$ and $f(c) \equiv f(b) \bmod q$. Therefore $f(c) \bmod pq$ is the unique solution to the Chinese remainder theorem.

Analogues of Fermat's little theorem also hold for polynomials.

Polynomial analogue of Fermat. If $g(x)$ is an irreducible polynomial of degree n over $\mathbf{F}_2$, then the relation $a(x)^{2^n-1} \equiv 1 \bmod g(x)$ holds for any polynomial $a(x)$ not divisible by $g(x)$.

Chinese remainder theorem. Let $g(x)$ and $h(x)$ be monic polynomials with no common factors. Given any polynomials $a(x)$ and $b(x)$, there exists a unique polynomial $c(x)$ such that $c(x) \equiv a(x) \bmod g(x)$ and $c(x) \equiv b(x) \bmod h(x)$.

We can create and work with polynomials over $\mathbf{F}_2$ as demonstrated by the following SAGE code.

```
sage: F2 = FiniteField(2)
sage: P2.<x> = PolynomialRing(F2)
sage: f = x^17 + x^5 + 1
sage: f.factor()
x^17 + x^5 + 1
sage: g = x^13 + x^5 + 1
sage: g.factor()
(x^2 + x + 1) * (x^11 + x^10 + x^8 + x^7 + x^5 + x^4 + x^3 + x + 1)
```

Exercise 7.1 *Let p be the prime $2^{31} - 1 = 2147483647$. Use the SAGE function* `powermod` *to verify Fermat's little theorem for several values of a. Why would it be a bad idea to compute a^{p-1} and then reduce modulo p?*

Solution. The function `a.powermod(e,p)` computes the result of $a^e \bmod p$ by doing an optimal number of squarings and multiplications, and reducing the intermediate results. The size of the expanded result a^e for large e, such as for $e = p - 1 = 2^{31} - 2$, would overflow the internal storage capacity of a computer, so it would be unwise to attempt to structure the algorithm as $a \mapsto a^e$ then to reduce modulo p.

Exercise 7.2 *Let p be as above and let $q = (2^{61} + 1)/3 = 768614336404564651$. Compute $a^{p-1} \bmod pq$ for various primes using* `powermod`. *Then reduce the result modulo p. How do you explain the result in terms of the Chinese remainder theorem and Fermat's little theorem?*

Solution. For primes $p = 2^{31} - 1$ and $q = (2^{61} - 1)/3$, we compute for $a = 2$ the power `2.powermod`$(p-1, pq) = 1031616713356184101960358$. If we reduce modulo q, then result

is 624499148328708779 — pretty much a random number of size q. On the other hand, if we reduce modulo p, the result is 1. This follows from Fermat's little theorem, since 2.powermod$(p-1, pq)$ mod p is equal to the result 2.powermod$(p-1, p)$.

Exercise 7.3 *Let $g(x) = x^{17} + x^5 + 1$, and use the function* powermod *to verify the polynomial analogue of Fermat's little theorem for the polynomials x, $x^2 + x + 1$, etc.*

Solution. For the polynomial $g(x) = x^{17} + x^5 + 1$, we should use exponent $e = 2^{17} - 1$, which we note is prime. We verify that each of the results x.powermod(e, g) and $(x^2 + x + 1)$powermod(e, g) is 1. Since e is prime, this proves that $g(x)$ is not only irreducible, but also primitive.

Exercise 7.4 *Let $h(x) = x^{17} + x^{15} + x^{10} + x^5 + 1$ and compute $a(x)^{2^{17}-1} \bmod g(x)h(x)$ for various $a(x)$. What is the result reduced modulo $g(x)$? Why does the same not hold true for $a(x)^{2^{17}-1} \bmod g(x)h(x)$, reduced modulo $h(x)$?*

Solution. With $g(x)$ as above and $h(x) = x^{17} + x^{15} + x^{10} + x^5 + 1$, the results x.powermod(e,gh).mod(g) equals 1 holds as expected, exactly as in the previous exercise. In this case, if $h(x)$ is also irreducible, then the result:

$$x.\text{powermod}(e, gh) \bmod h = x^{16} + x^{15} + x^{14} + x^{11} + x^{10} + x^8 + x^6 + x^3 + 1$$

would also have been 1. The fact that this result does not give 1 is a consequence of the reducibility of h:

$$h = (x^3 + x^2 + 1)(x^{14} + x^{13} + x^{11} + x^8 + x^5 + x^4 + x^3 + x^2 + 1).$$

Public Key Cryptography

The RSA cryptosystem is based on the difficulty of factoring large integers into its composite primes.

Based on Fermat's little theorem, we know that $a^m \equiv 1 \bmod p$ exactly when $p-1$ divides m. Therefore we recover the identity $a^u \equiv a \bmod p$ where u is of the form $1 + (p-1)r$. Now given any e such that e and $p-1$ have no common divisors, there exists a d such that $ed \equiv 1 \bmod p-1$. In other words, $u = ed$ is of the form $1 + (p-1)r$. This means that the map

$$a \mapsto a^e \bmod p$$

followed by

$$a^e \bmod p \mapsto (a^e \bmod p)^d \bmod p \equiv a^{ed} \bmod p = a \bmod p$$

are inverse maps. This only works for a prime p.

Exercise 8.5 *Use* SAGE *to find a large prime* p *and to compute inverse exponentiation pairs* e *and* d. *The following functions are of use:*

$$\texttt{random\_prime}, \texttt{gcd}, \texttt{xgcd}, \textit{and } \texttt{inverse\_mod}.$$

The RSA cryptosystem is based on the fact that for primes p *and* q *and any integer* e *with no common factors with* $p - 1$ *and* $q - 1$, *it is possible to find an* d_1 *such that*

$$ed_1 \equiv 1 \bmod (p - 1),$$
$$ed_2 \equiv 1 \bmod (q - 1).$$

Using the Chinese remainder theorem, it is possible to then find the unique d *such that*

$$d = d_1 \bmod (p - 1) \textit{ and } d = d_2 \bmod (q - 1)$$

in the range $1 \leq d < (p - 1)(q - 1)$. *This* d *has the property that*

$$a^{ed} \equiv a \bmod n.$$

The send a message securely, the public key (e, n) *is used. First we encoding the message as an integer* $a \bmod n$, *then form the ciphertext* $a^e \bmod n$. *The recipient recovers the message using the secret exponent* d.

Solution. The function call $\texttt{random\_prime}(2^{100})$ returns a random prime of up to 100 bits. Suppose that the primes

$$p = 1172991670841347272989353064539,$$
$$q = 3009975179695075520611104346547,$$

are found with this function, and set $e = 5$. We want to build the inverse exponent d such that $ed \equiv 1 \bmod (p - 1)$ and $ed \equiv 1 \bmod (q - 1)$. Note first that $\gcd(e, p - 1) = 1$ and $\gcd(e, q - 1) = 1$, so that such a d must exist. We first compute each of $d \bmod (p - 1)$ and $d \bmod (q - 1)$.

```
sage: p = 1172991670841347272989353064539
sage: q = 3009975179695075520611104346547
sage: e = 5
sage: d1 = inverse_mod(e,p-1)
sage: d1
703795002504808363793611838723
sage: d2 = inverse_mod(e,q-1)
sage: d2
2407980143756060416488834777237
```

The value of d can now be computed modulo the value $\mathrm{lcm}(p-1, q-1)$ — this is sufficient to determine the inverse, rather than the larger value of the product $(p - 1)(q - 1)$.

We would like to compute the value of d, but the SAGE function `crt` complains that the moduli $p-1$ and $q-1$ have a common factor.

```
sage: gcd(p-1,q-1);
6
sage: (p-1).factor()
2 * 3^3 * 13 * 23767 * 19475307419 * 3609932889503
sage: (q-1).factor()
2 * 3 * 17 * 5297 * 22123 * 152417 * 165217231734649
```

We can divide $q-1$ by 6 to remove the common factor, and so compute the Chinese remainder lifting as follows. Note first that the system is consistent — d_1 and d_2 are the same modulo 6 since they are both inverses to e mod 6.

```
sage: d1 mod 6
5
sage: d2 mod 6
5
```

Since $(q-1)/6$ is not divisible by 2 or 3, we can proceed with the Chinese remainder lifting with $p-1$ and $(q-1)/6$.

```
sage: d = crt([d1,d2],[p-1,(q-1).div(6)])
sage: d
353067581522151113489975704430723410964207885999877055538575
```

Alternatively we could have computed the inverse exponent d in one step by

```
sage: d = inverse_mod(e,lcm(p-1,q-1))
sage: d
353067581522151113489975704430723410964207885999877055538575
```

Exercise 8.6 *Use your exponents e, d, verify the identities mod p:*
$$(a^e)^d \equiv a \bmod n, \quad (a^d)^e \equiv a \bmod n, \quad and \ a^{ed} \equiv a \bmod n,$$
for various random values of a.

Note that after construction of d, the primes p and q are not needed, but that without knowing the original factorization of n, Fermat's little theorem does not apply, and finding the inverse exponent for e is considered a hard problem.

125

Solution. Now we can verify that e and d are inverses modulo $p-1$ and modulo $q-1$, and, moreover, that they determine inverse exponential maps modulo the RSA modulus $n = pq$.

```
sage: (e*d).mod(p-1)
1
sage: (e*d).mod(q-1)
1
sage: n = p*q
sage: m = random_prime(n)
sage: c = m.powermod(e,n)
sage: m == c.powermod(d,n)
True
sage: m == m.powermod(e*d,n)
True
```

We use the RSA cryptosystem in SAGE as follows. First begin with encoding ASCII text numerically:

```
sage: E := RSACryptosystem(128)
sage: m = E.encoding('The dog ate my lunch.'); m
01010100011010000110010100100000011001000110111101100011100100\
00001100001011101000110010100100000011011010111100100100000001\
10110001110101011011100110001101101000000101110
sage: E.decoding(m)
'The dog ate my lunch.'
```

Note that, as with LFSR cryptosystems, RSA encoding uses the information-preserving ASCII bit encoding, and encoding and decoding are true inverses. **Caution:** we note that printing "decoded" ciphertext might render an terminal nonfunctional, since the resulting ASCII text might contain escape characters which reset the terminal display.

To encipher, first we must create a key pair:

```
sage: (K, L) = E.random_key_pair()
sage: K
(493389218628303818077602918189940340453, 863986773687927680675564524563117433331)
```

This returns a pair of inverse keys `K` and `L`. We will consider `K` to be the public `K` and `L` to be the private key.

N.B. The argument to `RSACryptosystem` specifies the number of bits in the RSA modulus.

With a value of 128, the modulus is of size 2^{128}, or about 39 decimal digits. Each of the primes is of size approximately 20 decimal digits. This particular example can be easily broken by the factorization:

```
sage: x = K[0]; x
86398677368792768067556452456311743331
sage: x.factor()
6046864213681032211 * 14288178850339607921
```

Exercise 8.7 *Use the above factorization to reproduce the private key* L *(generated but not printed above) for this* K.

Solution.   Given the factorization

$$86398677368792768067556452456311743331$$
$$= 6046864213681032211 \cdot 14288178850339607921,$$

we can find the inverse to the exponent

$$e = 49338921862830381807760291818994034053.$$

```
sage: e = 49338921862830381807760291818994034053
sage: p = 6046864213681032211
sage: q = 14288178850339607921
sage: d = inverse_mod(e,lcm(p-1,q-1))
sage: d
28548445760572555940025914187603517
```

It is now possible to verify as above that (e, n) and (d, n) server as inverse RSA keys.

Exercise 8.8 *Why is the choice for which key is the public key and which key is the private key arbitrary? Practice encoding, decoding, enciphering, and deciphering with the RSA cryptosystem. Why do the member functions* enciphering *and* deciphering *return the same values?*

Solution.   Provided that e is chosen as a random number in the range

$$1 \le e \le \operatorname{lcm}(p - 1, q - 1),$$

which has no common factors with $p - 1$ or $q - 1$, then its inverse is a similarly random value in this range. Therefore after creation, the decision of which key to publish as the public key, and which key to guard as the private key is arbitrary.

N.B. Sometimes a special value, such as 3, 5, 17, 257, or 65537, is chosen as the public exponent. These are each of the form $2^r + 1$, so that the enciphering can be done rapidly using only r squarings and one multiplication. In such a case it is clear that no such "obvious" value is suitable for the private key, and the symmetry of the choice between public and private keys is broken.

An ElGamal cryptosystem is based on the difficulty of the Diffie–Hellman problem: Given a prime p, a primitive element a of $(\mathbf{Z}/p\mathbf{Z})^* = \{c \in Z/pZ : c \neq 0\}$, and elements $c_1 = a^x$ and $c_2 = a^y$, find the element a^{xy} in $(\mathbf{Z}/p\mathbf{Z})^*$.

Exercise 8.9 *Recall the discrete logarithm problem: Given a prime p, a primitive element a of $(\mathbf{Z}/p\mathbf{Z})^*$, and an element c of $(\mathbf{Z}/p\mathbf{Z})^*$, find an integer x such that $c = a^x$. Explain how a general solution to the discrete logarithm problem for p and a implies a solution to the Diffie–Hellman problem.*

Solution. Suppose that the discrete logarithm problem has an efficient solution. Then, given a primitive element a of $\mathbf{F}_p$, for every a^x and a^y we could solve for $x = \log_a(a^x)$ and for $y = \log_a(a^y)$. It follows that we could then produce the value a^{xy}, which solves the Diffie-Hellman problem.

Exercise 8.10 *Fermat's little theorem tells us that $a^{p-1} = 1$ for all a in $(\mathbf{Z}/p\mathbf{Z})^*$. Recall that a primitive element a has the property that $\mathbf{Z}/(p-1)\mathbf{Z} \to (\mathbf{Z}/p\mathbf{Z})^*$ given by $x \mapsto a^x$ is a bijection.*

1. *Show that a is primitive if and only if $a^x = 1$ only when $p - 1$ divides x.*

2. *Let p be prime $2^{32} + 15$. Show that $a = 3$ is a primitive element of $(\mathbf{Z}/p\mathbf{Z})^*$. Use the* SAGE *function* `log` *to compute discrete logarithms of elements of* `FiniteField(p)` *with respect to a.*

3. *Let p be the prime $2^{32} + 61$. Show that the element $a = 2$ is a primitive element for $(\mathbf{Z}/p\mathbf{Z})^*$. Use the* SAGE *function* `log` *to compute discrete logarithms of elements of* `FiniteField(p)` *with respect to a.*

Solution. The statement of the definition of primitive is a formal statement equivalent to that which follows. An element a of $\mathbf{Z}/p\mathbf{Z}$ is primitive if and only if

$$1, a, a^2, \ldots, a^{p-2}$$

are all distinct, and therefore enumerate all nonzero elements of $\mathbf{Z}/p\mathbf{Z}$.

1. Fermat's little theorem tells us that the next value, a^{p-1} in this list is 1, and therefore $a^x = 1$ for all $x = r(p-1)$, and indeed, we have run out of nonzero elements so must have a repeat at this point.

   Conversely for any nonzero element a there must be some value t such that $a^t = 1$, hence $a^{rt} = 1$ for all r. We may assume that t divides $p - 1$, since if $t' = \gcd(t, p-1)$ then there exist r and s such that $t' = rt + s(p-1)$, so

   $$1 = a^{rt}a^{s(p-1)} = a^{rt+s(p-1)} = a^{t'},$$

   and we can replace t by t'. Therefore the maximum length of a cycle $1, a, a^2, \ldots, a^{t-1}$ divides $p - 1$ and is equal l to $p - 1$ exactly when a is primitive.

2. For $p = 2^{32} + 15$, the factorization of $p - 1$ is $2 \cdot 3^2 \cdot 5 \cdot 131 \cdot 364289$. We need to check that 3^x is not 1 mod p for any divisor of $p - 1$.

   ```
   sage: p = 2^32+15
   sage: m = p-1
   sage: (m1, m2, m3, m4, m5) = (m.div(q) for q in (2,3,5,131,364289))
   sage: powermod(3,m1,p);
   4294967310
   sage: powermod(3,m2,p);
   2086193154
   sage: powermod(3,m3,p);
   239247313
   sage: powermod(3,m4,p);
   1859000016
   sage: powermod(3,m5,p);
   1338913740
   ```

   How does this prove that 3 is a primitive element?

   By producing random elements in $\mathbf{F}_p$ and computing discrete logarithms with respect to a, we find that the time to compute discrete logarithms in $\mathbf{F}_p = \mathbf{Z}/p\mathbf{Z}$ is trivial.

   ```
   sage: FF = FiniteField(p)
   sage: x = FF(3)
   sage: for i in range(4):
   ...          y = FF.random_element()
   ...          time n = log(y,x)
   ```

3. For $p = 2^{32} + 61$, the factorization of $p - 1$ is $2^2 \cdot 1073741839$. We repeat the same test as in the previous part.

```
sage: p = 2^32+61
sage: m = (p-1).quo_rem(2)[0]
sage: 2.powermod(m,p)
4294967356
sage: m = (p-1).quo_rem(1073741839)[0]
sage: 2.powermod(m,p)
16
```

The shows that 2 is a primitive element. Next we find that, for this prime p, that the time to compute discrete logarithms in $\mathbf{F}_p = \mathbf{Z}/p\mathbf{Z}$ is nontrivial.

```
sage: FF = FiniteField(p)
sage: x = FF(2)
sage: for i in range(4):
...          y = FF.random_element()
...          time n = log(y,x)
816373
986893
931102
62625
```

Exercise 8.11 *Compare the times to compute discrete logarithms in the previous exercise. Now factor $p - 1$ for each p. What difference do you note? Explain the timings in terms of the Chinese remainder theorem for $\mathbf{Z}/(p-1)\mathbf{Z}$.*

Solution. The nontrivial time for the discrete logarithm is due to the large prime divisor of $p - 1$. The amount of time required to compute a discrete logarithm in $\mathbf{F}_p$ is dependent on the size of the largest prime divisor of $p - 1$. The discrete logarithm can be computed independently for each prime divisor of $p - 1$ — more correctly for prime power divisor — and the discrete logarithm can be recovered by the Chinese remainder theorem, as is the next example.

Exercise 8.12 *Let p be the prime $2^{131} + 1883$ and verify the factorization*

$$p - 1 = 2 \cdot 3 \cdot 5 \cdot 37 \cdot 634466267339108669 \cdot 3865430919824322067.$$

Let $a = 109$ and $c = 1014452131230551128319928312434869768346$ and set

$$n_5 = (p - 1) \text{ div } 634466267339108669$$
$$n_6 = (p - 1) \text{ div } 3865430919824322067.$$

Then verify that $c^{n_5} = a^{129 n_5}$ and $c^{n_6} = a^{127 n_6}$. Find similar relations for

$$n_1 = (p-1) \text{ div } 2 \quad n_3 = (p-1) \text{ div } 5,$$
$$n_2 = (p-1) \text{ div } 3 \quad n_4 = (p-1) \text{ div } 37.$$

and use this information to find the discrete logarithm of c with respect to a.

Solution. We set up the problem in SAGE in the following way.

```
sage: p = 2^131+1883
sage: fac = (p-1).factor(); fac
2 * 3 * 5 * 37 * 634466267339108669 * 3865430919824322067
sage: primes = [ f[0] for f in (p-1).factor() ]
sage: (p1, p2, p3, p4, p5, p6) = primes
sage: expons = ( (p-1) // r for r in primes )
sage: (n1, n2, n3, n4, n5, n6) = expons
```

In this way, p_1, p_2, p_3, p_4, p_5, and p_6 are assigned to be the prime divisors of $p-1$ and n_1, n_2, n_3, n_4, n_5, and n_6 their cofactors.

Raising both generator a and its power c to the large exponents n_1, n_2, n_3, and n_4 reduces the solution to the discrete logarithm to one modulo $p_1 = 2$, $p_2 = 3$, $p_3 = 5$, and $p_4 = 37$, which can be easily solved by enumerating all possibilities.

```
sage: FF = FiniteField(p)
sage: a = FF(109)
sage: c = FF(1014452131230551128319928312434869768346)
sage: [ a^(n1*i) for i in range(2) ].index(c^n1)
1
sage: [ a^(n2*i) for i in range(3) ].index(c^n2)
2
sage: [ a^(n3*i) for i in range(5) ].index(c^n3)
4
sage: [ a^(n4*i) for i in range(37) ].index(c^n4)
29
sage: crt([1,2,4,29],[2,3,5,37])
29
```

For the larger primes

$$p_5 = 634466267339108669 \text{ and } p_6 = 3865430919824322067,$$

we verify the given discrete logarithms.

```
sage: a5 := a^n5
sage: c5 := c^n5
sage: a5^129
1106532280219457618983939634726858708298
sage: c5
1106532280219457618983939634726858708298
sage: a6 = a^n6
sage: c6 = c^n6
sage: a6^127
8095792859180089801332726438583202198
sage: c6
8095792859180089801332726438583202198
```

The discrete logarithm x can be recovered from the discrete logarithms $x_i = \log_{a_i}(c_i)$ where $a_i = a^{n_i}$ and $c_i = c^{n_i}$ by using the function `crt` to find the Chinese remainder lifting.

```
sage: x = CRT([29,129,127],[2*3*5*37,p5,p6]);
sage: x
107521720347655517565250443822403779
sage: a^x eq c
True
```

Digital Signatures

Secret Sharing

Exercise 10.1 *Verify the correctness of the formula for the secret secret in Shamir's secret sharing scheme by substituting into the formula of Lagrange's interpolation theorem.*

Revision Exercises

Let $\mathcal{A}$ be the alphabet $\{A, B, C, D, E\}$. Given the message A BAD CAB A DEAD DAD, we form the strip–encoded plaintext

$$M = \text{ABADCABADEADDAD}$$

by removing all characters not in the alphabet.

Exercise D.1 *Encipher the message M using the substitution key $K = $ BDEAC. Find the inverse key and verify the correctness by deciphering your ciphertext.*

Solution. Recall that the key $K = $ BDEAC specifies the map $A \mapsto B$, $B \mapsto D$, etc. This results in the enciphering

$$M = \text{ABADCABADEADDAD} \mapsto C = \text{BDBAEBDBACBAABD}.$$

Exercise D.2 *Let $\mathcal{A} \to \mathbf{Z}/5\mathbf{Z}$ be the bijection $A \mapsto 0, B \mapsto 1, \dots, E \mapsto 4$. Encipher the message M using the Vigenère key $K = $ ADECB in ECB mode, then encipher the same message using the same key and initialization vector BBBB, in CFB and OFB modes with the block length $n = 5$ and $r = 1$. Rather than bit sum, use summation in $\mathbf{Z}/5\mathbf{Z}$ for the feedback. Verify the correctness of your results by then deciphering the ciphertext.*

Solution. We make the identification $M = \text{ABADCABADEADDAD} = $ 010320103403303 over $\mathbf{Z}/5\mathbf{Z}$. In the same way, we write $K = $ ADECB $= $ 03421. The enciphering in ECB mode is then 044030440001223 $= $ AEEADAEEAAABCCD. The enciphering in CBC mode begins with $C_0 = $ BBBBB $= $ 11111, and since $E_K(C_{j-1} \oplus M_j) = C_{j-1} \oplus E_K(M_j)$, we just add in the previous ciphertext block to get

$$11111100141441410132 = \text{BBBBBBAABEBEEBEBABDC}.$$

The application of this function E_K to form the state vectors I_j is particularly weak, since only the first character of the key K and the first character of the initialization vector. Since $K = \mathtt{A} * * * *$, this means $I_j = \mathtt{B} * * * *$, and so the ciphertext output is

BBBBBBCBEDBCBEABEEBE.

Exercise D.3 *Let $K = [3, 5, 4, 1, 2]$ be a transposition key. Encipher the message M in ECB mode and in CBC mode. Verify the correctness of your results by deciphering the ciphertext.*

Solution. The key $K = [3, 5, 4, 1, 2]$ specifies a transposition, under which $M \mapsto$ ACDABAEDABDDAAD in ECB mode. If we use the addition $\oplus$ of the previous question for the feedback, then we get ciphertext

BBBBBACDABDADADCCAAC.

Check this work carefully for errors – no guarantees.

Exercise D.4 *Which of the modes of operation leaves Vigenère ciphertext open to attack by the Kasiski method? Which mode of operation was used for the block ciphers in the course assignments, and why?*

Solution. We made use of the ECB mode in order to preserve the structure of a Vigenère ciphertext. This leaves this and other classical cryptosystems open to classical attacks such as the Kasiski method.

Mathematics of LFSR's

Next we focus on some of the mathmematical problems which arise in stream ciphers and public key cryptography. The problems given are of a size which can be computed by hand, with minimal effort if the proper method is used.

Exercise D.5 *Let S be the set $\{x^6 + x + 1, x^6 + x^3 + 1, x^6 + x^5 + 1, x^6 + x^2 + 1\}$ of polynomials in $\mathbf{F}_2[x]$.*

1. *Which of the polynomials are irreducible?*

2. *Which of the polynomials are primitive?*

3. *What are the periods of the linear feedback shift registers with the above connections polynomials?*

4. (∗) *The polynomial* $g(x) = x^6 + x^5 + x^4 + x^3 + 1$ *is not irreducible. What is its factorization, and what are the periods of output sequence of a linear feedback shift register with* $g(x)$ *as connection polynomial and initial states* 010011, 010010, *and* 111111?

Solution. Let S be the set $\{x^6 + x + 1, x^6 + x^3 + 1, x^6 + x^5 + 1, x^6 + x^2 + 1\}$ of polynomials in $\mathbf{F}_2[x]$.

1. The polynomials $x^6 + x + 1$, $x^6 + x^3 + 1$, and $x^6 + x^5 + 1$ are irreducible, but $x^6 + x^2 + 1 = (x^3 + x + 1)^2$.

2. Of the three irreducible polynomials, we find that $x^6 + x^3 + 1$ generates LFSR output of period 9, so is not primitive. The other two irreducible polynomials generate output of period greater than $21 = 63/3$, so must be primitive.

3. The periods are therefore 63, 9, 63, and (at most) 14. The period of 14 can be determined for a specific value, but poor choices, like 1101001 can result in a period of 7, since the connection polynomial is not irreducible.

4. The polynomial $g(x) = x^6 + x^5 + x^4 + x^3 + 1$ factors as $(x^2 + x + 1)(x^4 + x + 1)$. The LFSR outputs for initial states 010011, 010010, and 111111 with connection polynomial $g(x)$ are:

$$010011100110000010011100110000001\ldots$$
$$010010101000011010010101010000001101\ldots$$
$$111111000101110111111100010111011\ldots$$

These each have periods 15, which equals $2^4 - 1$ (rather than $2^6 - 1$, which would be the case if $g(x)$ where irreducible.)

Mathematics of RSA

Exercise D.6 *Let* $G = (\mathbf{Z}/15\mathbf{Z})^*$.

1. *What are the elements of* G?

2. *Show that* $a = 2$ *is a primitive element for* $(\mathbf{Z}/3\mathbf{Z})^*$ *and* $a = 3$ *is a primitive element for* $(\mathbf{Z}/5\mathbf{Z})^*$.

3. *Find an element* a *in* $\mathbf{Z}$ *which is primitive for both* $(\mathbf{Z}/3\mathbf{Z})^*$ *and* $(\mathbf{Z}/5\mathbf{Z})^*$.

4. (∗) *Why does it not make sense to speak of a primitive element for* G?

5. (∗) *How many elements a of G have the property of being primitive for both* $(\mathbf{Z}/3\mathbf{Z})^*$ *and* $(\mathbf{Z}/5\mathbf{Z})^*$?

Solution.

1. The elements of $(\mathbf{Z}/15\mathbf{Z})^*$ are

$$\{1, 2, 4, 7, 8, 11, 13, 14\},$$

   the elements of $\mathbf{Z}/15\mathbf{Z}$ coprime to 3 and 5.

2. Since $\{1, 2\} = (\mathbf{Z}/3\mathbf{Z})^*$ and $\{1, 3, 9 = 4, 27 = 2\} = (\mathbf{Z}/5\mathbf{Z})^*$, 2 and 3 are primitive elements for these moduli.

3. The integer 8 is primitive in $(\mathbf{Z}/3\mathbf{Z})^*$ and $(\mathbf{Z}/5\mathbf{Z})^*$, since 8 is a CRT lift of the pair $(2, 3)$ in $\mathbf{Z}/3\mathbf{Z} \times \mathbf{Z}/5\mathbf{Z}$, i.e. $8 \equiv 2 \bmod 3$ and $8 \equiv 3 \bmod 5$.

4. There is no primitive element for $\mathbf{Z}/15\mathbf{Z}^*$ since no single element generates all of them. For instance the powers of 8 are:

$$1, 8, 64 = 4, 32 = 2, 16 = 1,$$

   which generates a cycle of length only 4, whereas $(\mathbf{Z}/15\mathbf{Z})^*$ has eight elements.

5. The elements of $(\mathbf{Z}/15\mathbf{Z})^*$ which are primitive for both $(\mathbf{Z}/3\mathbf{Z})^*$ and $(\mathbf{Z}/5\mathbf{Z})^*$ are the two CRT images of the pairs $(2, 3)$ and $(2, 2)$.

Mathematics of Diffie–Hellman

Exercise D.7 *Let* $G_1 = (\mathbf{Z}/89\mathbf{Z})^*$ *and* $G_2 = (\mathbf{Z}/97\mathbf{Z})^*$.

1. *Show that* 7 *is a primitive element for* G_1 *and for* G_2.

2. *Solve the discrete logarithm problem* $\log_7(2)$ *in* G_1 *and in* G_2.

3. (∗) *Which discrete logarithm is harder, and why?*

Solution.

1. To show that 7 is a primitive element for $(\mathbf{Z}/89\mathbf{Z})^*$ and $(\mathbf{Z}/97\mathbf{Z})^*$, we need to show that
$$7^{44} \not\equiv 1 \bmod 89 \quad 7^{48} \not\equiv 1 \bmod 97$$
$$7^8 \not\equiv 1 \bmod 89 \quad 7^{32} \not\equiv 1 \bmod 97$$

   These values can be computed using products of successive squares of 7, e.g. $7^{44} = 7^4 7^8 7^{32}$, so $7^4 \equiv 87 \bmod 89$, $7^8 \equiv (-2)^2 \equiv 4 \bmod 89$, $7^{16} \equiv 16 \bmod 89$. Therefore $7^{44} \equiv -1 \bmod 89$, etc.

2. We find $\log_7(2) = 48$ in $\mathbf{F}_{89}$ and $\log_7(2) = 94$ in $\mathbf{F}_{97}$ using the baby-step, giant-step method.

3. A discrete logarithm in $(\mathbf{Z}/97\mathbf{Z})^*$ is theoretically easier to solve because $96 = 2^5 3$, so we solve the discrete logarithms using this factorization.

N.B. Verify that you can solve $\log_7(2)$ using $\log_{7^m}(2^m)$ where $m = 32$, and $m = 48, 24, 12, 6, 3$, and at each of the latter steps you only need to determine one additional bit of information.

Mathematics of Shamir's Secret Sharing Scheme

Recall the Lagrange interpolation theorem:

Theorem D.1 (Lagrange) *Let k be a field and let $f(x)$ be a polynomial over k of degree less than t. Given t distinct elements $x_1, x_2, \ldots, x_t$ of k, then $f(x)$ equals*

$$\sum_{i=1}^{t} f(x_i) \prod_{\substack{j=1 \\ j \neq i}}^{t} \frac{x - x_j}{x_i - x_j}$$

Exercise D.8 *Let $\mathbf{F}_{31} = \mathbf{Z}/31\mathbf{Z}$ be the finite field of 31 elements, and let*

$$\{(1,1), (2,16), (3,25), (4,28)\}$$

be a set of pairs of the form $(i, f(i))$ for some polynomial $f(x)$.

1. *Find the value $f(0)$ of the polynomial $f(x)$ of degree 2 which interpolates the first three points.*

2. *Find the polynomial $f(x)$ of degree 2 which interpolates the first three points.*

3. *Show that the same polynomial passes through the fourth point.*

4. *Use the Lagrange interpolation theorem to conclude that $f(x)$ is the unique polynomial of degree less than 4 which passes through these four points.*

Solution. Let $\mathbf{F}_{31} = \mathbf{Z}/31\mathbf{Z}$ be the finite field of 31 elements, and let

$$\{(1,1), (2,16), (3,25), (4,28)\}$$

be a set of pairs of the form $(i, f(i))$ for some polynomial $f(x)$.

1. The value $f(0)$ of the polynomial $f(x)$ of degree 2 which interpolates the first three points is given, using the first three shares, by

$$f(0) = 1 \cdot \frac{(2)(3)}{(2-1)(3-1)} + 16 \cdot \frac{(1)(3)}{(1-2)(3-2)} + 25 \cdot \frac{(1)(2)}{(1-3)(2-3)}$$
$$= 1 \cdot (3) + 16 \cdot (-3) + 25 \cdot (1) = 3 + 14 + 25 = 11.$$

Using the last three shares we find:

$$f(0) = 16 \cdot \frac{(3)(4)}{(3-2)(4-2)} + 25 \cdot \frac{(2)(4)}{(2-3)(4-3)} + 28 \cdot \frac{(2)(3)}{(2-4)(3-4)}$$
$$= 16 \cdot (6) + 25 \cdot (-8) + 28 \cdot (3) = 3 + 17 + 22 = 11.$$

N.B. Be careful to do any inversions modulo 31!!

2. Using the full formula, we get $f(x) = 28x^2 + 24x + 11$.

3. Verify: $f(4) = 28 \cdot 4^2 + 24 \cdot 4 + 11 = 14 + 3 + 11 = 28$.

4. Since the polynomial $f(x)$ agrees with the *four* points, this must be the unique polynomial of degree less than four which does so.