The Laws of Cryptography with Java Code

by Neal R. Wagner

Table of Contents
Parts and Chapters

Foreword . . . . . **xi**

Introduction . . . . . **xiii**

I. Preliminaries **2**

1. Cryptographers' Favorites **3**
2. Cryptographers' Favorite Algorithms **14**

II. Coding and Information Theory **22**

3. Coding and Information Theory **23**
4. Visualizing Channel Capacity **31**
5. The Huffman Code for Compression **35**
6. The Hamming Code for Error Correction **44**
7. Coping with Decimal Numbers **48**
8. Verhoeff's Decimal Error Detection **54**

III. Introduction to Cryptography **60**

9. Cryptograms and Terminology **61**
10. The One-Time Pad **66**
11. Conventional Block Cipher Cryptosystems **70**
??. Conventional Stream Cipher Cryptosystems

IV. Public Key Cryptography **78**

12. Public Key Distribution Systems **79**
13. Public Key Cryptography: Knapsacks **82**
14. The RSA Public Key Cryptosystem **86**
15. Rabin's Version of RSA **93**
??. Elliptic Curve Cryptosystems
??. Other Public Key Cryptosystems

V. Random Number Generation **98**

16. Traditional Random Number Generators **99**
17. Random Numbers From Chaos Theory **106**
18. Statistical Tests and Perfect Generators **112**

VI. The Advanced Encryption Standard (AES) **114**

19. Introduction to the AES 115
20. The Finite Field GF(256) 119
21. The S-Boxes 127
22. Key Expansion 130
23. Encryption 133
24. Decryption 136

VII. Hash Functions and Digital Signatures

??. One-Way Functions and Hash Functions
??. Digital Signatures

VIII. Randomization Techniques

??. Simple Randomization
??. More Complex Tricks
??. The Rip van Winkle Cipher and Rabin's Scheme

IX. Identification and Key Distribution 140

25. Passwords 141
26. Zero-Knowledge Protocols 144
27. Identification Schemes 150
28. Threshold Schemes 153
??. Case Study: the Secure Shell (ssh)

Java Programs 159

Appendices 307

A. Using Printed Log Tables 309
B. Unsigned bytes in Java 310
C. Projects 313

Table of Contents
Laws

I. Preliminaries

1. Cryptographers' Favorites
   - **Law XOR-1** **3**
   - **Law XOR-2** **5**
   - **Law LOG-1** **5**
   - **Law LOG-2** **7**
   - **Law GROUP-1** **8**
   - **Law FIELD-1** **9**
   - **Law FIELD-2** **10**
   - **Law FERMAT-1** **10**
2. Cryptographers' Favorite Algorithms
   - **Law GCD-1** **14**
   - **Law EXP-1** **17**
   - **Law PRIME-1** **19**
   - **Law PRIME-2** **20**

II. Coding and Information Theory

3. Coding and Information Theory
   - **Law ENTROPY-1** **23**
   - **Law ENTROPY-2** **24**
   - **Law INFORMATION-1** **25**
   - **Law SHANNON-1** **29**
4. Visualizing Channel Capacity
5. The Huffman Code for Compression
   - **Law SHANNON-2** **35**
   - **Law COMPRESSION-1** **36**
6. The Hamming Code for Error Correction
   - **Law HAMMING-1** **46**
7. Coping with Decimal Numbers
   - **Law DECIMAL-1** **48**
   - **Law DECIMAL-2** **53**
8. Verhoeff's Decimal Error Detection
   - **Law DECIMAL-3** **57**

III. Introduction to Cryptography

9. Cryptograms and Terminology

Law CRYPTOGRAPHY-1a **62**
Law CRYPTOGRAPHY-1b **62**
Law CRYPTOGRAPHY-2 . . . . . **63**
Law CRYPTANALYSIS-1 . . . . . **64**
Law CRYPTANALYSIS-2 . . . . . **64**
Law CRYPTANALYSIS-3 . . . . . **64**
Law CRYPTANALYSIS-4 . . . . . **65**
Law CRYPTANALYSIS-5 . . . . . **65**
10. The One-Time Pad
Law PAD-1 . . . . . **69**
11. Conventional Block Cipher Cryptosystems
Law BLOCKCIPHER-1 . . . . . **71**
Law BLOCKCIPHER-2 . . . . . **73**
??. Conventional Stream Cipher Cryptosystems

IV. Public Key Cryptography

12. Public Key Distribution Systems
13. Public Key Cryptography: Knapsacks
14. The RSA Public Key Cryptosystem
Law RSA-1 . . . . . **86**
Law RSA-2 . . . . . **91**
15. Rabin's Version of RSA
Law RABIN-1 . . . . . **93**
??. Elliptic Curve Cryptosystems
??. Other Public Key Cryptosystems

V. Random Number Generation

16. Traditional Random Number Generators
Law RNG-1 . . . . . **99**
Law RNG-2 . . . . . **99**
Law RNG-3 . . . . . **100**
17. Random Numbers From Chaos Theory
18. Statistical Tests and Perfect Generators

VI. The Advanced Encryption Standard (AES)

19. Introduction to the AES
Law AES-1 . . . . . **115**
20. The Finite Field GF(256)
21. The S-Boxes
22. Key Expansion
23. Encryption

24. Decryption

VII. Hash Functions and Digital Signatures

??. One-Way Functions and Hash Functions
??. Digital Signatures

VIII. Randomization Techniques

??. Simple Randomization
??. More Complex Tricks
??. The Rip van Winkle Cipher and Rabin's Scheme

IX. Identification and Key Distribution

25. Passwords
26. Zero-Knowledge Protocols
27. Identification Schemes
28. Threshold Schemes
Law THRESHOLD-1 157
??. Case Study: the Secure Shell (ssh)

Java Programs

Appendices

A. Using Printed Log Tables
B. Unsigned bytes in Java
Law JAVA-BYTES-1 311
Law JAVA-BYTES-2 312

Table of Contents
Java Programs

I. Preliminaries

1. Cryptographers' Favorites
   - **a. Demonstration of Xor 4 161**
   - **b. Formulas for logs 5 162**
   - **c. Fermat's Theorem Illustrated 12 163**
2. Cryptographers' Favorite Algorithms
   - **a. Basic GCD Algorithm 15 165**
   - **b. Extended GCD Algorithm 16 166**
   - **c. Extended GCD Algorithm (debug version) 17 167**
   - **d. Testing Two Exponential Algorithms 19 169**

II. Coding and Information Theory

3. Coding and Information Theory
   - **a. Formula for Channal Capacity 26 172**
   - **b. Table of Channal Capacities 26 173**
   - **c. Inverse of the Channal Capacity formula 26 175**
   - **d. Table of Repetition Codes 28 177**
4. Visualizing Channel Capacity
   - **a. The Simulation Program 33 179**
5. The Huffman Code for Compression
   - **a. The Huffman Algorithm 41 183**
   - **b. Two Distinct Huffman Codes 43 191**
6. The Hamming Code for Error Correction
   - **a. The Hamming Algorithm 47 193**
7. Coping with Decimal Numbers
   - **a. U.S. Banking Scheme 53 203**
   - **b. IBM Scheme 53 206**
   - **c. ISBN mod 11 Scheme 53 209**
   - **d. Mod 97 Scheme 53 212**
   - **e. Hamming mod 11 Scheme, Error Correction 53 215**
   - **f. Hamming mod 11 Scheme, Double Errors 53 219**
8. Verhoeff's Decimal Error Detection
   - **a. Use of the Dihedral Group 57 223**
   - **b. Verhoeff's Scheme 57 226**

III. Introduction to Cryptography

9. Cryptograms and Terminology
   a. Cryptogram Program 62 229
10. The One-Time Pad
   a. Caesar Cipher 67 232
   b. Beale Cipher 67 235
   c. Generate a One-time Pad 69 239
   d. Wheels to Encrypt/Decrypt With a Pad 69 242
11. Conventional Block Cipher Cryptosystems
??. Conventional Stream Cipher Cryptosystems

IV. Public Key Cryptography

12. Public Key Distribution Systems
13. Public Key Cryptography: Knapsacks
14. The RSA Public Key Cryptosystem
   a. RSA Implementation 90 246
   b. Faster RSA, Using Chinese Remainder Theorem . . . 92 251
15. Rabin's Version of RSA
   a. Square Roots mod n = p*q 94 256
??. Elliptic Curve Cryptosystems
??. Other Public Key Cryptosystems

V. Random Number Generation

16. Traditional Random Number Generators
   a. Linear Congruence Random Number Generators . . 103 259
   b. Exponential and Normal Distributions 103 262
17. Random Numbers From Chaos Theory
   a. The logistic Lattice as a RNG 110 266
18. Statistical Tests and Perfect Generators
   a. Maurer's Universal Test 112 270
   b. The Blum-Blum-Shub Perfect Generator 112 272

VI. The Advanced Encryption Standard (AES)

19. Introduction to the AES
20. The Finite Field GF(256)
   a. Generate Multiplication Tables 125 273
   b. Compare Multiplication Results 126 275
21. The S-Boxes

a. Generate AES Tables 127 277
22. Key Expansion
23. Encryption
a. AES Encryption 135 282
24. Decryption
a. AES Decryption 138 290
b. Test Runs of the AES Algorithm 138 293

VII. Hash Functions and Digital Signatures

??. One-Way Functions and Hash Functions
??. Digital Signatures

VIII. Randomization Techniques

??. Simple Randomization
??. More Complex Tricks
??. The Rip van Winkle Cipher and Rabin's Scheme

IX. Identification and Key Distribution

25. Passwords and Key Distribution
26. Zero-Knowledge Proofs
27. Identification Schemes
28. Threshold Schemes
a. Shamir's Threshold Schemes 157 299
??. Case Study: the Secure Shell (ssh)

Foreword

There are excellent technical treatises on cryptography, along with a number of popular books. In this book I am trying to find a middle ground, a "gentle" introduction to selected topics in cryptography without avoiding the mathematics. The material is aimed at undergraduate computer science students, but I hope it will be accessible and of interest to many others. The idea is to cover a limited number of topics carefully, with clear explanations, sample calculations, and illustrative Java implementations.

The emphasis is on the underlying systems and their theory, rather than on details of the use of systems already implemented. For example, the notes present material on the RSA cryptosystem, its theory and a Java implementation, but there is no discussion of a commercial implementation such as PGP ("Pretty Good Privacy").

The Java class libraries now give considerable support for commercial cryptography, and there are whole books just on this subject, but this book doesn't cover this topic.

The reader should not actively dislike mathematics, although the amount and difficulty of the mathematics requirements vary. One of my goals is to cover the necessary mathematics without hiding details, but also without requiring material from an undergraduate mathematics degree. Also a number of subjects and results do not include full mathematical proofs.

The notes contain "maxims" or "laws" designed to emphasize important points, sometimes in an amusing way — hence the title of the overall work.

I refer interested readers to the *Handbook of Applied Cryptography*, by Menezes, van Oorschot, and Vanstone (CRC Press, 1997). That work gives a comprehensive survey of the whole field, leaving many details to technical articles that the handbook refers to, and presenting "techniques and algorithms of greatest interest to the current practitioner". In contrast, my work is more idiosyncratic, occasionally presenting odd or obscure material, and not trying to be comprehensive.

The Java programs that accompany this book are demonstration implementations to help readers and students understand the concepts. I have kept the code simple to further this goal, rather than strive for code that could be included into commercial or open source projects, which would require far longer and more complex code (and be much harder for me to write). The complexities then would get in the way of understanding. Readers need some familiarity with programming and with Java to understand these programs, but most of the exposition is independent of Java.

The book also contains various tables of values along with sample or "toy" calculations. In every case I've found it easier and quicker to write Java programs to generate this material rather than to do the calculations by hand. In many cases the Java programs directly output HTML source to display a table. Tables in this book use Latex source, but I do not include Java code that outputs Latex, since HTML is far more accessible. Thus when I say: "The Java program on page xxx creates Table X.Y," this means that the Java program creates a nearly identical HTML table.

The Java programs in the book are available online in machine-readable form on the author's web page:

`http://www.cs.utsa.edu/~wagner/lawsbook/`

This book was partly inspired by two undergraduate courses in cryptography taught at the University of Texas at San Antonio during the Spring 2002 and Spring 2003 semesters. The web page for the course has many links and other information:

`http://www.cs.utsa.edu/~wagner/CS4953/index.html`

A one-semester undergraduate course in cryptography might cover the following material:

- Part I. *Introductory Material on Functions and Algorithms*, referring back to it as needed.
- Part II. *Coding and Information Theory*, without the Huffman or Hamming codes, and with emphasis on Verhoeff's detection method.
- Part III. *Introduction to Cryptography*, covered quickly.
- Part IV. *Public Key Cryptography*, the first four chapters.
- Part V. *Random Number Generation*, the first two chapters.
- Part VI. *The Advanced Encryption Standard (AES)*, all.
- Plus selected remaining topics as desired.

The author would like to thank his mother for giving birth to him, but can't think of anyone else to thank at this time.

San Antonio, Texas
June, 2003

Introduction

Mankind has used the science of cryptography or "secret messages" for thousands of years to transmit and store information needing secrecy. Until recently the military expended most of the effort and money involved. However, starting in 1976 with the introduction in the open literature of public key cryptography by Diffie and Hellman, the non-military and academic pursuit of cryptography has exploded. The computer revolution has given people the means to use far more complicated cryptographic codes, and the same revolution has made such widespread and complex codes necessary. At the start of a new millennium, even non-technical people understand the importance of techniques to secure information transmission and storage.

Cryptography provides four main types of services related to data that is transmitted or stored:

- **Confidentiality:** keep the data *secret*.
- **Integrity:** keep the data *unaltered*.
- **Authentication:** be certain *where the data came from*.
- **Non-repudiation:** so someone cannot *deny sending the data*.

Consider first *confidentiality*. This is just a big word meaning "secrecy" — keeping the data secret. For this one uses *encryption*, a process of taking readable and meaningful data, and scrambling or transforming it so that someone who happens to intercept the data can no longer understand it. As part of the process, there has to be a way for authorized parties to unscramble or *decrypt* the encrypted data.

Integrity means keeping the data in unaltered form, while *authentication* means to know where the data came from and who sent it. Neither of these services has anything to do with secrecy, though one might also want secrecy. Consider, for example, the transfer of funds involving U.S. Federal Reserve Banks (and other banks). While secrecy might be desirable, it is of small importance compared with being sure who is asking for the transfer (the authentication) and being sure that the transfer is not altered (the integrity). One important tool that helps implement these services is the *digital signature*. A digital signature has much in common with an ordinary signature, except that it works better: when properly used it is difficult to forge, and it behaves as if the signature were scrawled over the entire document, so that any alteration to the document would alter the signature. In contrast, ordinary signatures are notoriously easy to forge and are affixed to just one small portion of a document.

The final service, *non-repudiation*, prevents someone from claiming that they had not sent a document that was authenticated as coming from them. For example, the person might claim that their private key had been stolen. This service is important but difficult to implement, and is discussed in various of the books referred to in the references.

Refinements and extensions of these basic services fall into a category I call *cryptographic trickery*: clever capabilities that might initially seem impossible, such as public keys, zero knowledge proofs, and threshold schemes. I include examples of this material to entice readers into the fascinating field of cryptography.

Taken all together, cryptography and its uses and implementations have become essential for mankind's technical civilization. The future promise is for the smooth functioning of these and other services to allow individuals, businesses, and governments to interact without fear in the new digital and online world.

Part I
Favorites

1
The Laws of Cryptography
Cryptographers' Favorites

1.1 Exclusive-Or.

The function known as **exclusive-or** is also represented as **xor** or a plus sign in a circle, $\oplus$. The expression $a \oplus b$ means either a or b *but not both*. Ordinary *inclusive-or* in mathematics means either one or the other *or both*. The exclusive-or function is available in C / C++ / Java for bit strings as a hat character: `^`. (Be careful: the hat character is often used to mean exponentiation, but Java, C, and C++ have no exponentiation operator. The hat character also sometimes designates a control character.) In Java `^` also works as exclusive-or for **`boolean`** type.

Law XOR-1:
The cryptographer's favorite function is *Exclusive-Or*.

Exclusive-or comes up continually in cryptography. This function plays an essential role in the one-time pad (Chapter 10), stream ciphers (Chapter ??), and the Advanced Encryption Standard (Part VI), along with many other places.

Recall that the boolean constant **`true`** is often written as a **`1`** and **`false`** as a **`0`**. Exclusive-or is the same as *addition mod 2*, which means ordinary addition, followed by taking the remainder on division by 2.

For single bits 0 and 1, Table 1.1 gives the definition of their exclusive-or.

The exclusive-or function has many interesting properties, including the following, which hold for any bit values or bit strings a, b, and c:

Exclusive-Or		
a	b	$a \oplus b$
0	0	0
0	1	1
1	0	1
1	1	0

Table 1.1 Definition of Exclusive-Or.

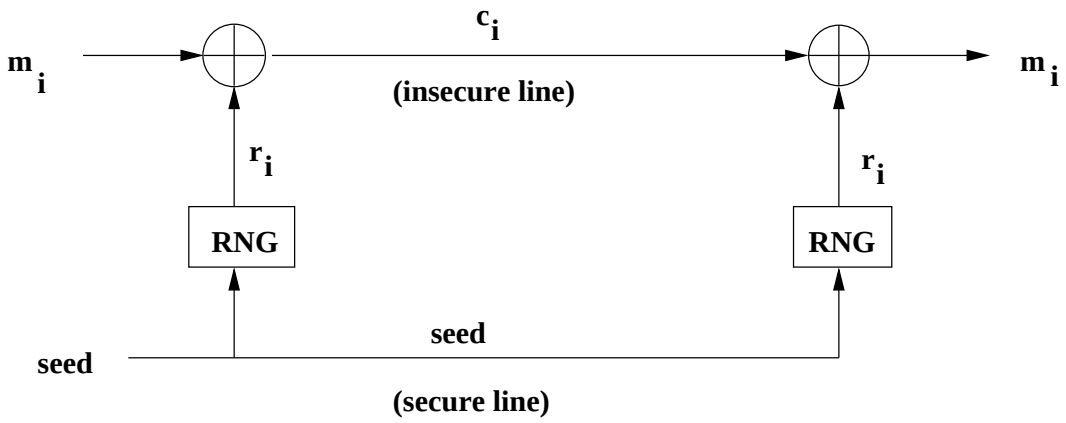


Figure 1.1 Cryptosystem Using XOR.

$a \oplus a = 0$
$a \oplus 0 = a$
$a \oplus 1 = \sim a$, where $\sim$ is bit complement.
$a \oplus b = b \oplus a$ (commutativity)
$a \oplus (b \oplus c) = (a \oplus b) \oplus c$ (associativity)
$a \oplus a \oplus a = a$
if $a \oplus b = c$, then $c \oplus b = a$ and $c \oplus a = b$.

Beginning programmers learn how to exchange the values in two variables **a** and **b**, using a third temporary variable **temp** and the assignment operator **=** :

```
temp = a;
a = b;
b = temp;
```

The same result can be accomplished using **xor** without an extra temporary location, regarding **a** and **b** as bit strings. (A Java program that demonstrates interchange using exclusive-or is on page 161).

```
a = a xor b;
b = a xor b;
a = a xor b;
```

For an example of exclusive-or used in cryptography, consider taking the xor of a pseudo-random bit stream r_i with a message bit stream m_i to give an encrypted bit stream c_i, where $c_i = r_i \oplus m_i$. To decrypt, xor the same pseudo-random bit stream r_i with c_i to give m_i back: $r_i \oplus c_i = r_i \oplus r_i \oplus m_i = 0 \oplus m_i = m_i$. Figure 1.1 illustrates this process.

Law XOR-2:
Cryptographers love exclusive-or because it immediately gives a cryptosystem.

1.2 Logarithms.

By definition, $y = \log_b x$ means the same as $b^y = x$. One says: "y is the logarithm of x to base b," or "y is the log base b of x." Stated another way, $\log_b x$ (also known as y) is the *exponent* you raise b to in order to get x. Thus $b^{(\log_b x)} = x$. In more mathematical terms, the logarithm is the *inverse* function of the exponential.

Law LOG-1:
The cryptographer's favorite logarithm is *log base 2*.

One uses logs base 2 in cryptography (as well as in most of computer science) because of the emphasis on binary numbers in these fields.

So $y = \log_2 x$ means the same as $2^y = x$, and a logarithm base 2 of x is the exponent you raise 2 to in order to get x. In symbols: if $y = \log_2 x$, then $x = 2^y = 2^{\log_2 x}$. In particular $2^{10} = 1024$ means the same as $\log_2 1024 = 10$. Notice that $2^y > 0$ for all y, and inversely $\log_2 x$ is not defined for $x \leq 0$.

Here are several other formulas involving logarithms:

$$\begin{aligned}
&\log_2(ab) = \log_2 a + \log_2 b, \text{ for all } a, b > 0\\
&\log_2(a/b) = \log_2 a - \log_2 b, \text{ for all } a, b > 0\\
&\log_2(1/a) = \log_2(a^{-1}) = -\log_2 a, \text{ for all } a > 0\\
&\log_2(a^r) = r \log_2 a, \text{ for all } a > 0, r\\
&\log_2(a + b) = \text{(Oops! No simple formula for this.)}
\end{aligned}$$

Table 1.2 gives a few examples of logs base 2.

Some calculators, as well as languages like Java, do not directly support logs base 2. Java does not even support logs base 10, but only logs base e, the "natural" log. However, a log base 2 is just a fixed constant times a natural log, so they are easy to calculate if you know the "magic" constant. The formulas are:

$$\begin{aligned}
\log_2 x &= \log_e x / \log_e 2, \text{ (mathematics)}\\
&= \texttt{Math.log(x)/Math.log(2.0);} \text{ (Java).}
\end{aligned}$$

The magic constant is: $\log_e 2 = 0.69314\ 71805\ 59945\ 30941\ 72321$, or $1/\log_e 2 = 1.44269\ 50408\ 88963\ 40735\ 99246$. (Similarly, $\log_2 x = \log_{10} x / \log_{10} 2$, and $\log_{10} 2 = 0.30102999566398114$.)

Logarithms base 2	
$x = 2^y = 2^{\log_2 x}$	$y = \log_2 x$
1 073 741 824	30
1 048 576	20
1 024	10
8	3
4	2
2	1
1	0
1/2	−1
1/4	−2
1/8	−3
1/1024	−10
1/1048576	−20
1/1073741824	−30
0	$-\infty$
< 0	undefi ned

Table 1.2 Logarithms base 2.

A Java program that demonstrates these formulas is found on page 162.
Here is a proof of the above formula:

$2^y = x$, or $y = \log_2 x$ (then take $\log_e$ of each side)
$\log_e(2^y) = \log_e x$ (then use properties of logarithms)
$y \log_e 2 = \log_e x$ (then solve for y)
$y = \log_e x / \log_e 2$ (then substitute $\log_2 x$ for y)
$\log_2 x = \log_e x / \log_e 2$.

Law LOG-2:
The log base 2 of an integer x tells how many bits it takes to represent x in binary.

Thus $\log_2 10000 = 13.28771238$, so it takes 14 bits to represent 10000 in binary. (In fact, $10000_{10} = 10011100010000_2$.) Exact powers of 2 are a special case: $\log_2 1024 = 10$, but it takes 11 bits to represent 1024 in binary, as 10000000000_2.

Similarly, $\log_{10}(x)$ gives the number of decimal digits needed to represent x.

1.3 Groups.

A *group* is a set of *group elements* with a *binary operation* for combining any two elements to get a unique third element. If one denotes the group operation by $\#$, then the above says that for any group elements a and b, $a\#b$ is defined and is also a group element. Groups are also *associative*, meaning that $a\#(b\#c) = (a\#b)\#c$, for any group elements a, b, and c. There has to be an *identity element* e satisfying $a\#e = e\#a = a$ for any group element a. Finally, any element a must have an *inverse* a' satisfying $a\#a' = a'\#a = e$.

If $a\#b = b\#a$ for all group elements a and b, the group is *commutative*. Otherwise it is *non-commutative*. Notice that even in a non-commutative group, $a\#b = b\#a$ might sometimes be true — for example if a or b is the identity.

A group with only finitely many elements is called *finite*; otherwise it is *infinite*.

Examples:

1. The *integers* (all whole numbers, including 0 and negative numbers) form a group using ordinary addition. The identity is 0 and the inverse of a is $-a$. This is an infinite commutative group.

2. The *positive rationals* (all positive fractions, including all positive integers) form a group if ordinary multiplication is the operation. The identity is 1 and the inverse of r is $1/r = r^{-1}$. This is another infinite commutative group.

3. The *integers mod n* form a group for any integer $n > 0$. This group is often denoted Z_n. Here the elements are 0, 1, 2, ..., $n - 1$ and the operation is addition followed by

+	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	0	1	2	3	4	5	6	7	8	9
1	1	2	3	4	5	6	7	8	9	0
2	2	3	4	5	6	7	8	9	0	1
3	3	4	5	6	7	8	9	0	1	2
4	4	5	6	7	8	9	0	1	2	3
5	5	6	7	8	9	0	1	2	3	4
6	6	7	8	9	0	1	2	3	4	5
7	7	8	9	0	1	2	3	4	5	6
8	8	9	0	1	2	3	4	5	6	7
9	9	0	1	2	3	4	5	6	7	8

Table 1.3 Addition in the integers mod 10, Z_{10}.

remainder on division by n. The identity is 0 and the inverse of a is $n - a$ (except for 0 which is its own inverse). This is a finite commutative group.

4. For an example of a non-commutative group, consider 2-by-2 non-singular matrices of real numbers (or rationals), where the operation is matrix multiplication:

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

Here a, b, c, and d are real numbers (or rationals) and $ad - bc$ must be non-zero (non-singular matrices). The operation is matrix multiplication. The above matrix has inverse

$$\frac{1}{ad - bc} \begin{pmatrix} d & -b \\ -c & a \end{pmatrix}$$

and the identity is

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}.$$

This is an infinite non-commutative group.

5. The chapter on decimal numbers gives an interesting and useful example of a finite *non*-commutative group: the *dihedral* group with ten elements.

Law GROUP-1:
The cryptographer's favorite group is the *integers mod n*, Z_n.

In the special case of $n = 10$, the operation of addition in Z_{10} can be defined by $(x + y)$ mod 10, that is, divide by 10 and take the remainder. Table 1.3shows how one can also use an *addition table* to define the integers modulo 10:

1.4 Fields.

A *field* is an object with a lot of structure, which this section will only outline. A field has two operations, call them $+$ and $*$ (though they will not necessarily be ordinary addition and multiplication). Using $+$, all the elements of the field form a commutative group. Denote the identity of this group by 0 and denote the inverse of a by $-a$. Using $*$, all the elements of the field except 0 must form another commutative group with identity denoted 1 and inverse of a denoted by a^{-1}. (The element 0 has no inverse under $*$.) There is also the *distributive identity*, linking $+$ and $*$: $a * (b + c) = (a * b) + (a * c)$, for all field elements a, b, and c. Finally, one has to exclude *divisors of zero*, that is, non-zero elements whose product is zero. This is equivalent to the following *cancellation property:* if c is not zero and $a * c = b * c$, then $a = b$.

Examples:

1. Consider the *rational numbers* (fractions) **Q**, or the *real numbers* **R**, or the *complex numbers* **C**, using ordinary addition and multiplication (extended in the last case to the complex numbers). These are all infinite fields.

2. Consider the *integers mod p*, denoted Z_p, where **p** is a prime number (2, 3, 5, 7, 11, 13, 17, 19, 23, 29, . . .). Regard this as a group using $+$ (ordinary addition followed by remainder on division by p). The elements with 0 left out form a group under $*$ (ordinary multiplication followed by remainder on division by p). Here the identity is clearly 1, but the inverse of a non-zero element a is not obvious. In Java, the inverse must be an element x satisfying $(x * a)\%p == 1$. It is always possible to find the unique element x, using an algorithm from number theory known as the *extended Euclidean algorithm*. This is the topic in the next chapter, but in brief: because p is prime and a is non-zero, the greatest common divisor of p and a is 1. Then the extended Euclidean algorithm gives ordinary integers x and y satisfying $x * a + y * p = 1$, or $x * a = 1 - y * p$, and this says that if you divide $x * a$ by p, you get remainder 1, so this x is the inverse of a. (As an integer, x might be negative, and in this case one must add p to it to get an element of Z_p.)

Law FIELD-1:
The cryptographer's favorite field is the *integers mod p*, denoted Z_p, where *p* is a prime number.

The above field is the only one with p elements. In other words, the field is unique up to renaming its elements, meaning that one can always use a different set of symbols to represent the elements of the field, but it will still be essentially the same.

There is also a unique finite field with p^n elements for any integer $n > 1$, denoted $GF(p^n)$. Particularly useful in cryptography is the special case with $p = 2$, that is, with 2^n elements for $n > 1$. The case $2^8 = 256$ is used, for example, in the new U.S. Advanced Encryption Standard (AES). It is more difficult to describe than the field Z_p. The chapter about multiplication for the AES will describe this field in more detail, but here are some of its properties in brief for now: It has 256 elements, represented as all possible strings of 8 bits. Addition in the field is just the same as bitwise exclusive-or (or bitwise addition mod 2). The zero element is 00000000, and the identity element is 00000001. So far, so good, but multiplication is more problematic: one has to regard an element as a degree 7 polynomial with coefficients in the field Z_2 (just a 0 or a 1) and use a special version of multiplication of these polynomials. The details will come in the later chapter on the AES.

Law FIELD-2:
The cryptographer's *other* favorite field is *GF(2^n)*.

1.5 Fermat's Theorem.

In cryptography, one often wants to raise a number to a power, modulo another number. For the integers mod p where p is a prime (denoted Z_p), there is a result know as Fermat's Theorem, discovered by the 17th century French mathematician Pierre de Fermat, 1601-1665.

Theorem (Fermat): If p is a prime and a is any non-zero number less than p, then

$$a^{p-1} \bmod p = 1$$

Law FERMAT-1:
The cryptographer's favorite theorem is Fermat's Theorem.

Table 1.4 illustrates Fermat's Theorem for $p = 13$. Notice below that the value is always 1 by the time the power gets to 12, but sometimes the value gets to 1 earlier. The initial run up to the 1 value is shown in bold italic in the table. The lengths of these runs are always numbers that divide evenly into 12, that is, 2, 3, 4, 6, or 12. A value of a for which the whole row is bold italic is called a *generator*. In this case 2, 6, 7, and 11 are generators.

Because a to a power mod p always starts repeating after the power reaches $p - 1$, one can reduce the power mod $p - 1$ and still get the same answer. Thus no matter how big the power x might be,

$$a^x \bmod p = a^{x \bmod (p-1)} \bmod p.$$

p	a	a^1	a^2	a^3	a^4	a^5	a^6	a^7	a^8	a^9	a^{10}	a^{11}	a^{12}
13	2	***2***	***4***	***8***	***3***	***6***	***12***	***11***	***9***	***5***	***10***	***7***	***1***
13	3	***3***	***9***	***1***	3	9	1	3	9	1	3	9	1
13	4	***4***	***3***	***12***	***9***	***10***	***1***	4	3	12	9	10	1
13	5	***5***	***12***	***8***	***1***	5	12	8	1	5	12	8	1
13	6	***6***	***10***	***8***	***9***	***2***	***12***	***7***	***3***	***5***	***4***	***11***	***1***
13	7	***7***	***10***	***5***	***9***	***11***	***12***	***6***	***3***	***8***	***4***	***2***	***1***
13	8	***8***	***12***	***5***	***1***	8	12	5	1	8	12	5	1
13	9	***9***	***3***	***1***	9	3	1	9	3	1	9	3	1
13	10	***10***	***9***	***12***	***3***	***4***	***1***	10	9	12	3	4	1
13	11	***11***	***4***	***5***	***3***	***7***	***12***	***2***	***9***	***8***	***10***	***6***	***1***
13	12	***12***	***1***	12	1	12	1	12	1	12	1	12	1

Table 1.4 Fermat's Theorem for $p = 13$.

Thus modulo p in the expression requires modulo $p - 1$ in the exponent. (Naively, one might expect to reduce the exponent mod p, but this is not correct.) So, for example, if $p = 13$ as above, then

$$a^{29} \bmod 13 = a^{29 \bmod 12} \bmod 13 = a^5 \bmod 13.$$

The Swiss mathematician Leonhard Euler (1707-1783) discovered a generalization of Fermat's Theorem which will later be useful in the discussion of the RSA cryptosystem.

Theorem (Euler): If n is any positive integer and a is any positive integer less than n with no divisors in common with n, then

$$a^{\phi(n)} \bmod n = 1,$$

where $\phi(n)$ is the *Euler phi function*:

$$\phi(n) = n(1 - 1/p_1) \ldots (1 - 1/p_m),$$

and $p_1, \ldots, p_m$ are all the prime numbers that divide evenly into n, including n itself in case it is a prime.

If n is a prime, then using the formula, $\phi(n) = n(1 - 1/n) = n(\frac{n-1}{n}) = n - 1$, so Euler's result is a special case of Fermat's. Another special case needed for the RSA cryptosystem comes when the modulus is a product of two primes: $n = pq$. Then $\phi(n) = n(1 - 1/p)(1 -$

n	a	a^1	a^2	a^3	a^4	a^5	a^6	a^7	a^8	a^9	a^{10}	a^{11}	a^{12}	a^{13}	a^{14}
15	2	***2***	***4***	***8***	***1***	2	4	8	1	2	4	8	1	2	4
15	3	***3***	***9***	***12***	***6***	***3***	***9***	***12***	***6***	***3***	***9***	***12***	***6***	***3***	***9***
15	4	***4***	***1***	4	1	4	1	4	1	4	1	4	1	4	1
15	5	***5***	***10***	***5***	***10***	***5***	***10***	***5***	***10***	***5***	***10***	***5***	***10***	***5***	***10***
15	6	***6***	***6***	***6***	***6***	***6***	***6***	***6***	***6***	***6***	***6***	***6***	***6***	***6***	***6***
15	7	***7***	***4***	***13***	***1***	7	4	13	1	7	4	13	1	7	4
15	8	***8***	***4***	***2***	***1***	8	4	2	1	8	4	2	1	8	4
15	9	***9***	***6***	***9***	***6***	***9***	***6***	***9***	***6***	***9***	***6***	***9***	***6***	***9***	***6***
15	10	***10***	***10***	***10***	***10***	***10***	***10***	***10***	***10***	***10***	***10***	***10***	***10***	***10***	***10***
15	11	***11***	***1***	11	1	11	1	11	1	11	1	11	1	11	1
15	12	***12***	***9***	***3***	***6***	***12***	***9***	***3***	***6***	***12***	***9***	***3***	***6***	***12***	***9***
15	13	***13***	***4***	***7***	***1***	13	4	7	1	13	4	7	1	13	4
15	14	***14***	***1***	14	1	14	1	14	1	14	1	14	1	14	1

Table 1.5 Euler's theorem for $n = 15$ and $\phi(n) = 8$.

$1/q) = (p-1)(q-1)$. Table 1.5 illustrates Euler's theorem for $n = 15 = 3 \cdot 5$, with $\phi(15) = 15 \cdot (1 - 1/3) \cdot (1 - 1/5) = (3-1) \cdot (5-1) = 8$. Notice here that a 1 is reached when the power gets to 8 (actually in this simple case when the power gets to 2 or 4), but only for numbers with no divisors in common with 15. For other base numbers, the value never gets to 1.

The tables above were generated by the Java program on page 163.

In a way similar to Fermat's Theorem, arithmetic in the exponent is taken mod $\phi(n)$, so that, assuming a has no divisors in common with n,

$$a^x \bmod n = a^{x \bmod \phi(n)} \bmod p.$$

If $n = 15$ as above, then $\phi(n) = 8$, and if neither 3 nor 5 divides evenly into a, then $\phi(n) = 8$. Thus for example,

$$a^{28} \bmod 15 = a^{28 \bmod 8} \bmod 15 = a^4 \bmod 15.$$

The proof in Chapter 14 that the RSA cryptosystem works depends on the above fact.

Exercises

1. For any bit string a, what is $a \oplus a \oplus a \oplus a \oplus a$ equal to?

2. Prove in two ways that the three equations using exclusive-or to interchange two values work. One way should use just the definition of **`xor`** in the table, and the other way should use the properties of **`xor`** listed above. (On computer hardware that has an **`xor`** instruction combined with assignment, the above solution may execute just as fast as the previous one and will avoid the extra variable.)

3. Use the notation $\vee$ to mean "inclusive-or", $\wedge$ to mean "and", and $\sim$ to mean "not". With this notation, show, using either truth tables or algebraically that

$$\begin{aligned} a \oplus b &= (a\wedge \sim b) \vee (\sim a \wedge b), \text{ and} \\ &= (a \vee b) \wedge (\sim (a \wedge b)). \end{aligned}$$

4. Show how to use exclusive-or to compare the differences between two bit strings.

5. Given a bit string a, show how to use another *mask* bit string m of the same length to reverse a fixed bit position i, that is, change 0 to 1 and 1 to 0, but just in position i.

6. How many bits are needed to represent a number that is 100 decimal digits long? How many decimal digits are needed to represent a number that is 1000 bits long? How many decimal digits are needed to represent a number that is 100 decimal digits long?

7. Write a Java function to return the log base b of a, where $b > 1$ and $a > 0$. Test your function.

8. In the example of 2-by-2 matrices, verify that the product of a non-zero element and its inverse in the identity.

2

The Laws of Cryptography
Cryptographers' Favorite Algorithms

2.1 The Extended Euclidean Algorithm.

The previous section introduced the field known as the *integers mod p*, denoted Z_p or $GF(p)$. Most of the field operations are straightforward, since they are just the ordinary arithmetic operations followed by remainder on division by p. However the multiplicative inverse is not intuitive and requires some theory to compute. If a is a non-zero element of the field, then a^{-1} can be computed efficiently using what is known as *the extended Euclidean algorithm*, which gives the greatest common divisor (gcd) along with other integers that allow one to calculate the inverse. It is the topic of the remainder of this section.

Law GCD-1:

The cryptographer's first and oldest favorite algorithm is the *extended Euclidean algorithm*, which computes the greatest common divisor of two positive integers *a* and *b* and also supplies integers *x* and *y* such that *x*a + y*b = gcd(a, b)*.

The Basic Euclidean Algorithm to give the gcd: Consider the calculation of the greatest common divisor (gcd) of 819 and 462. One could factor the numbers as: $819 = 3 \cdot 3 \cdot 7 \cdot 13$ and $462 = 2 \cdot 3 \cdot 7 \cdot 11$, to see immediately that the gcd is $21 = 3 \cdot 7$. The problem with this method is that there is no efficient algorithm to factor integers. In fact, the security of the RSA cryptosystem relies on the difficulty of factoring, and we need an extended gcd algorithm to implement RSA. It turns out that there is another better algorithm for the gcd — developed 2500 years ago by Euclid (or mathematicians before him), called (surprise) the *Euclidean* algorithm.

The algorithm is simple: just repeatedly divide the larger one by the smaller, and write an equation of the form "larger = smaller * quotient + remainder". Then repeat using the two numbers "smaller" and "remainder". When you get a 0 remainder, then you have the gcd of the original two numbers. Here is the sequence of calculations for the same example as before:

$$
\begin{array}{rcrcrcrl}
819 & = & 462 & \cdot & 1 & + & 357 & \text{(Step 0)} \\
462 & = & 357 & \cdot & 1 & + & 105 & \text{(Step 1)} \\
357 & = & 105 & \cdot & 3 & + & 42 & \text{(Step 2)} \\
105 & = & 42 & \cdot & 2 & + & 21 & \text{(Step 3, so GCD = 21)} \\
42 & = & 21 & \cdot & 2 & + & 0 & \text{(Step 4)}
\end{array}
$$

The proof that this works is simple: a common divisor of the first two numbers must also be a common divisor of all three numbers all the way down. (Any number is a divisor of 0, sort of on an honorary basis.) One also has to argue that the algorithm will terminate and not go on forever, but this is clear since the remainders must be smaller at each stage.

Here is Java code for two versions of the GCD algorithm: one iterative and one recursive. (There is also a more complicated *binary* version that is efficient and does not require division.)

Java function: gcd (two versions)

```
public static long gcd1(long x, long y) {
   if (y == 0) return x;
   return gcd1(y, x%y);
}

public static long gcd2(long x, long y) {
   while (y != 0) {
      long r = x % y;
      x = y; y = r;
   }
   return x;
}
```

A complete Java program using the above two functions is on page 165.

The *Extended* GCD Algorithm: Given the two positive integers 819 and 462, the extended Euclidean algorithm finds unique integers a and b so that $a \cdot 819 + b \cdot 462 = \gcd(819, 462) = 21$. In this case, $(-9) \cdot 819 + 16 \cdot 462 = 21$.

For this example, to calculate the magic a and b, just work backwards through the original equations, from step 3 back to step 0 (see above). Below are equations, where each shows two numbers a and b from a step of the original algorithm, and corresponding integers x and y (in **bold**), such that $x \cdot a + y \cdot b = gcd(a, b)$. Between each pair of equations is an equation in *italics* that leads to the next equation.

```
1*105 + (-2)* 42 = 21                                  (from Step 3 above)
(-2)*357 + (-2)(-3)*105 = (-2)*42 = (-1)*105 + 21    (Step 2 times -2)
(-2)*357 +  7*105 = 21          (Combine and simplify previous equation)
7 *462 + (7)(-1)* 357 = 7*105 = 2*357 + 21            (Step 1 times 7)
7*462 + (-9)*357 = 21           (Combine and simplify previous equation)
(-9)*819 + (-9)(-1)*462 = (-9)*357 = (-7)*462 + 21     (Step 0 * (-9))
(-9)*819 +  16*462 =  21                 (Simplify -- the final answer)
```

It's possible to code the extended gcd algorithm following the model above, first using a loop to calculate the gcd, while saving the quotients at each stage, and then using a second loop as above to work back through the equations, solving for the gcd in terms of the original two numbers. However, there is a much shorter, tricky version of the extended gcd algorithm, adapted from D. Knuth.

Java function: GCD (extended gcd)

```
public static long[] GCD(long x, long y) {
   long[] u = {1, 0, x}, v = {0, 1, y}, t = new long[3];
   while (v[2] != 0) {
      long q = u[2]/v[2];
```

```
      for (int i = 0; i < 3; i++) {
         t[i] = u[i] - v[i]*q; u[i] = v[i]; v[i] = t[i];
      }
   }
   return u;
}
```

A complete Java program using the above function is on page 166.

The above code rather hides what is going on, so with additional comments and checks, the code is rewritten below. Note that at every stage of the algorithm below, if **`w`** stands for any of the three vectors **`u`**, **`v`** or **`t`**, then one has **`x*w[0] + y*w[1] = w[2]`**. The function **`check`** verifies that this condition is met, checking in each case the vector that has just been changed. Since at the end, **`u[2]`** is the gcd, **`u[0]`** and **`u[1]`** must be the desired integers.

Java function: GCD (debug version)

```
public static long[] GCD(long x, long y) {
   long[] u = new long[3];
   long[] v = new long[3];
   long[] t = new long[3];
   // at all stages, if w is any of the 3 vectors u, v or t, then
   //   x*w[0] + y*w[1] = w[2]  (this is verified by "check" below)
   // vector initializations: u = {1, 0, u}; v = {0, 1, v};
   u[0] = 1; u[1] = 0; u[2] = x; v[0] = 0; v[1] = 1; v[2] = y;

   while (v[2] != 0) {
      long q = u[2]/v[2];
      // vector equation:  t = u - v*q
      t[0] = u[0] - v[0]*q; t[1] = u[1] - v[1]*q; t[2] = u[2] - v[2]*q;
                              check(x, y, t);
      // vector equation:  u = v;
      u[0] = v[0]; u[1] = v[1]; u[2] = v[2]; check(x, y, u);
      // vector equation:  v = t;
      v[0] = t[0]; v[1] = t[1]; v[2] = t[2]; check(x, y, v);
   }
   return u;
}

public static void check(long x, long y, long[] w) {
   if (x*w[0] + y*w[1] != w[2])
      System.exit(1);
}
```

Here is the result of a run with the data shown above:

```
q       u[0]    u[1]    u[2]    v[0]    v[1]    v[2]}

1        0       1      462      1      -1      357
1        1      -1      357     -1       2      105
3       -1       2      105      4      -7       42
2        4      -7       42     -9      16       21
2       -9      16       21     22     -39        0
```

```
gcd(819, 462) = 21
(-9)*819 + (16)*462 = 21
```

Here is a run starting with 40902 and 24140:

```
q       u[0]    u[1]    u[2]    v[0]    v[1]    v[2]}

1         0       1    24140      1      -1    16762
1         1      -1    16762     -1       2     7378
2        -1       2     7378      3      -5     2006
3         3      -5     2006    -10      17     1360
1       -10      17     1360     13     -22      646
2        13     -22      646    -36      61       68
9       -36      61       68    337    -571       34
2       337     571       34   -710    1203        0

gcd(40902, 24140) = 34
(337)*40902 + (-571)*24140 = 34
```

A complete Java program with the above functions, along with other example runs appears on page 167.

2.2 Fast Integer Exponentiation (Raise to a Power).

A number of cryptosystems require arithmetic on large integers. For example, the RSA public key cryptosystem uses integers that are at least 1024 bits long. An essential part of many of the algorithms involved is to raise an integer to another integer power, modulo an integer (taking the remainder on division).

Law EXP-1:
Many cryptosystems in modern cryptography depend on a fast algorithm to perform integer exponentiation.

It comes as a surprise to some people that in a reasonable amount of time one can raise a 1024 bit integer to a similar-sized power modulo an integer of the same size. (This calculation can be done on a modern workstation in a fraction of a second.) In fact, if one wants to calculate x^{1024} (a 10-bit exponent), there is no need to multiply x by itself 1024 times, but one only needs to square x and keep squaring the result 10 times. Similarly, 20 squarings yields $x^{1048576}$ (a 20-bit exponent), and an exponent with 1024 bits requires only that many squarings if it is an exact power of 2. Intermediate powers come from saving intermediate results and multiplying them in. RSA would be useless if there were no efficient exponentiation algorithm.

There are two distinct fast algorithms for raising a number to an integer power. Here is pseudo-code for raising an integer x to power an integer Y:

Java function: exp (first version)

```
int exp(int x, int Y[], int k) {
   //  Y = Y[k] Y[k-1] ... Y[1] Y[0]  (in binary)
   int y = 0, z = 1;
```

```
   for (int i = k; i >= 0; i--) {
      y = 2*y;
      z = z*z;
      if (Y[i] == 1) {
         y++;
         z = z*x;
      }
   }
   return z;
}
```

The variable y is only present to give a loop invariant, since at the beginning and end of each loop, as well as just before the if statement, the invariant $x^y = z$ holds, and after the loop terminates $y = Y$ is also true, so at the end, $z = x^Y$. To find $x^y \bmod n$ one should add a remainder on division by n to the two lines that calculate z.

Here is the other exponentiation algorithm. It is very similar to the previous algorithm, but differs in processing the binary bits of the exponent in the opposite order. It also creates those bits as it goes, while the other assumes they are given.

Java function: exp (second version)

```
int exp(int X, int Y) {
   int x = X, y = Y, z = 1;
   while (y > 0) {
      while (y%2 == 0) {
         x = x*x;
         y = y/2;
      }
      z = z*x;
      y = y - 1;
   }
   return z;
}
```

The loop invariant at each stage and after the each iteration of the inner while statement is:

$$z * x^y = X^Y.$$

Here is a mathematical proof that the second algorithm actually calculates X^Y. Just before the while loop starts, since $x = X$, $y = Y$, and $z = 1$, it is obvious that the loop invariant is true. (In these equations, the $=$ is a mathematical equals, not an assignment.)

Now suppose that at the start of one of the iterations of the while loop, the invariant holds. Use x', y', and z' for the new values of x, y, and z after executing the statements inside one iteration of the inner while statement. Notice that this assumes that y is even. Then the following are true:

$$\begin{aligned} x' &= x * x \\ y' &= y/2 \text{ (exact integer because y is even)} \\ z' &= z \\ z' * (x')^{y'} &= z * (x * x)^{y/2} = z * x^y = X^Y. \end{aligned}$$

This means that the loop invariant holds at the end of each iteration of the inner while statement for the new values of x, y, and z. Similarly, use the same prime notation for the variables at the end of the while loop.

$$x' = x$$
$$y' = y - 1$$
$$z' = z * x$$
$$z' * (x')^{y'} = z * x * (x)^{y-1} = z * x^y = X^Y.$$

So once again the loop invariant holds. After the loop terminates, the variable y must be 0, so that the loop invariant equation says:

$$X^Y = z * x^y = z * x^0 = z.$$

For a complete proof, one must also carefully argue that the loop will always terminate.
A test of the two exponentiation functions implemented in Java appears on page 169.

2.3 Checking For Probable Primes.

For 2500 years mathematicians studied prime numbers just because they were interesting, without any idea they would have practical applications. Where do prime numbers come up in the real world? Well, there's always the 7-Up soft drink, and there are sometimes a prime number of ball bearings arranged in a circle, to cut down on periodic wear. Now finally, in cryptography, prime numbers have come into their own.

Law PRIME-1:
A source of large random prime integers is an essential part of many current cryptosystems.

Usually large random primes are created (or found) by starting with a random integer n, and checking each successive integer after that point to see if it is prime. The present situation is interesting: there are reasonable algorithms to check that a large integer is prime, but these algorithms are not very efficient (although a recently discovered algorithm is guaranteed to produce an answer in running time no worse that the number of bits to the twelth power). On the other hand, it is very quick to check that an integer is "probably" prime. To a mathematician, it is not satisfactory to know that an integer is only probably prime, but if the chances of making a mistake about the number being a prime are reduced to a quantity close enough to zero, the users can just discount the chances of such a mistake.

Tests to check if a number is probably prime are called *pseudo-prime tests*. Many such tests are available, but most use mathematical overkill. Anyway, one starts with a property of a prime number, such as Fermat's Theorem, mentioned in the previous chapter: if p is a prime and a is any non-zero number less than p, then $a^{p-1} \bmod p = 1$. If one can find a number a

for which Fermat's Theorem does not hold, then the number p in the theorem is *definitely not a prime*. If the theorem holds, then p is called a *pseudo-prime with respect to* a, and it might actually be a prime.

So the simplest possible pseudo-prime test would just take a small value of a, say 2 or 3, and check if Fermat's Theorem is true.

> **Simple Pseudo-prime Test:** If a very large random integer p (100 decimal digits or more) is not divisible by a small prime, and if $3^{p-1} \bmod p = 1$, then the number is prime except for a vanishingly small probability, which one can ignore.

One could just repeat the test for other integers besides 3 as the base, but unfortunately there are non-primes (called *Carmichael numbers*) that satisfy Fermat's theorem for all values of a even though they are not prime. The smallest such number is $561 = 3 \cdot 11 \cdot 17$, but these numbers become very rare in the larger range, such as 1024-bit numbers. Corman et al. estimate that the chances of a mistake with just the above simple test are less than 10^{-41}, although in practice commercial cryptosystems use better tests for which there is a proof of the low probability. Such better tests are not really needed, since even if the almost inconceivable happened and a mistake were made, the cryptosystem wouldn't work, and one could just choose another pseudo-prime.

Law PRIME-2:
Just one simple pseudo-prime test is enough to test that a very large random integer is probably prime.

Exercises

1. Prove that the long (debug) version of the Extended GCD Algorithm works.
   (a) First show that `u[2]` is the gcd by throwing out all references to array indexes `0` and `1`, leaving just `u[2]`, `v[2]`, and `t[2]`. Show that this still terminates and just calculates the simple gcd, without reference to the other array indexes. (This shows that at the end of the complicated algorithm, `u[2]` actually is the gcd.)
   (b) Next show mathematically that the three special equations are true at the start of the algorithm, and that each stage of the algorithm leaves them true. (One says that they are left *invariant*.)
   (c) Finally deduce that algorithm is correct.

Part II
Coding and Information Theory

3
The Laws of Cryptography
Coding and Information Theory

3.1 Shannon's Information Theory and Entropy.

The term *information theory* refers to a remarkable field of study developed by Claude Shannon in 1948. Shannon's work was like Einstein's gravitation theory, in that he created the whole field all at once, answering the most important questions at the beginning. Shannon was concerned with "messages" and their transmission, even in the presence of "noise". For Shannon, the word "information" did not mean anything close to the ordinary English usage. Intuitively, the *information* in a message is the amount of "surprise" in the message. No surprise means zero information. (Hey, that's somewhat intuitive. However, it is not intuitive that a random message has the most information in this sense.)

Shannon defined a mathematical quantity called *entropy* which measures the amount of information in a message, if the message is one of a collection of possible messages, each with its own probability of occurrence. This entropy is the average number of bits needed to represent each possible message, using the best possible encoding. If there are n messages $X = X_1, \ldots, X_n$, with probabilities of occurrence: $p(X_1), \ldots, p(X_n)$ (with sum equal 1), then the *entropy* $H(X)$ of this set of messages is:

$$H(X) = p(X_1)\log_2(1/p(X_1)) + ... + p(X_n)\log_2(1/p(X_n)).$$

Intuitively, the entropy is just the weighted average of the number of bits required to represent each message, where the weights are the probabilities that each message might occur.

Law ENTROPY-1:
The entropy of a message is just the number of bits of information in the message, that is, the number of bits needed for the shortest possible encoding of the message.

It is possible to list reasonable properties of any entropy function and to prove that only the above formula gives a function with those properties.

For example, if we have two messages $X = \text{male}, \text{female}$, each having probability $1/2$, then the entropy is

$$H(X) = (1/2)(\log_2(1/(1/2)) + (1/2)(\log_2(1/(1/2))$$

$$\begin{aligned} &= (1/2)(\log_2 2) + (1/2)(\log_2 2) \\ &= 1/2 + 1/2 = 1. \end{aligned}$$

Thus in this case, as we would intuitively expect, there is one bit of information in such a message.

Suppose $p(X_1) = 1$ and the remaining probabilities are zero. In this case the entropy works out to be 0 as one would expect, since one is only going to receive the message X_1, so there is no information and no surprise in receiving this message. The actual message X_1 might be complex, with many bits representing it, but its probability is 1, so only this message can occur, with no information or "surprise" on its receipt, even if it is complex. (In the calculation of entropy, the term $0 \cdot \log_2(1/0)$ comes up, which looks like $0 \cdot \log_2(\infty)$. This term would be indeterminate, but the first part tends to 0 much faster than $\log_2(\infty)$ tends to ∞, so that in practice such terms are regarded as 0.)

As another example, suppose $n = 3$ and $p(X_1) = 1/2$, $p(X_2) = 1/4$, and $p(X_3) = 1/4$. Then the entropy works out to be 1.5. It is possible to encode these messages as follows: $X_1 : 0$, $X_2 : 10$, and $X_3 : 11$. In this case the average code length is the same as the entropy. One doesn't normally expect to be able to represent a collection of messages with a code whose average length is exactly equal to the entropy; it is *never* possible to get the average length less than the entropy.

Finally, suppose there are 1000 equally probably messages. Then the entropy is:

$$\begin{aligned} H(X) &= (1/1000)(\log_2(1/(1/1000)) + ... + (1/1000)(\log_2(1/(1/1000)) \\ &= (1/1000)\log_2(1000) + ... + (1/1000)\log_2(1000) \\ &= 1000(1/1000)\log_2(1000) \\ &= \log_2(1000) = 9.965784285. \end{aligned}$$

Thus the entropy value of these messages means that there are nearly 10 bits of information in each message. Similarly, if there are n equally likely messages, then the entropy of a message is $\log_2 n$. The equal probable case gives the largest possible value for the entropy of a collection of messages.

Law ENTROPY-2:
A random message has the most information (the greatest entropy). [Claude Shannon]

3.2 The Three Kinds of Codes.

The terms *code* and *coding* refer to ways of representing information. We will usually be using *binary* codes, that is, codes that use only the binary bits 0 and 1. There are three kinds of coding:

1. *Source Coding.* This usually involves *data compression*: representing the data with as few bits as possible. Notice that one always needs at least as many bits to encode a message as the entropy of the message. The next chapter talks more about source coding, and presents one special example: the Huffman code.

2. *Channel Coding.* Here one uses *error detection* and *error correction* to improve the reliability of the channel. This is accomplished by adding extra redundant bits. The rest of this chapter presents material on channel capacity and error correction codes. A later chapter talks about one particular error correcting code: the Hamming code.

3. *Secrecy Coding.* For secrecy, one uses *cryptography* to scramble the message so that it may not be intelligible to an eavesdropper. Much of the rest of the material in these notes is concerned with cryptographic coding. Often the scrambled message has the same number of bits as the original message.

Law INFORMATION-1:
In all coding theory, information transmission is essentially the same as information storage, since the latter is just transmission from *now* to *then*.

It is possible to have a single code that combines two or even all three of these functions, but the codes are usually kept separate. Normally one would compress a message (making the message smaller, to save storage or channel bandwidth), then transform it cryptographically for secrecy (without changing the message length), and finally add bits to the message to allow for error detection or correction.

3.3 Channel Capacity.

Shannon also introduced the concept of *channel capacity*, which is the maximum rate at which bits can be sent over an unreliable (noisy) information channel with arbitrarily good reliability. The channel capacity is represented as a fraction or percentage of the total rate at which bits can be sent physically over the channel. Shannon proved that there always exist codes that will signal arbitrarily close to the channel capacity with arbitrarily good reliability. Thus by choosing a larger and more complicated code, one can reduce the number of errors to as small a percentage as one would like, while continuing to signal as close as one wants to 100% of the channel capacity. In practice the theory does not provide these good codes, though they are known to exist. It is not possible to signal with arbitrarily good reliability at a rate greater than the channel capacity.

The simplest example of such a channel is the *binary symmetric channel*. Here every time a bit is transmitted, there is a fixed probability p, with $0 \leq p \leq 1$ such that a transmitted 0 is received as a 0 with probability p and received as a 1 with probability $1 - p$. The errors occur at random.

Probability	Channel Capacity
0.00 or 1.00	1.000000000000000
0.05 or 0.95	0.713603042884044
0.10 or 0.90	0.531004406410719
0.15 or 0.85	0.390159695283600
0.20 or 0.80	0.278071905112638
0.25 or 0.75	0.188721875540867
0.30 or 0.70	0.118709100769307
0.35 or 0.65	0.065931944624509
0.40 or 0.60	0.029049405545331
0.45 or 0.55	0.007225546012192
0.50 or 0.50	0.000000000000000

Table 3.1 Channel Capacity.

For example, if $p = 1$ there are no errors at all on the channel, and the channel capacity is 1 (meaning 100%). If $p = 0$, the capacity is still 1 as long as you realize that all bits are reversed. If $p = 0.5$, then on receipt of a bit, both 0 and 1 are equally likely as the bit that was sent, so one can never say anything about the original message. In this case the channel capacity is 0 and no information can be sent over the channel.

For binary symmetric channels there is a simple formula for the capacity C (a Java program that calculates channel capacity is on page 172):

$$C \quad = \quad 1 + p \log_2(p) + (1 - p) \log_2(1 - p).$$

Alternatively, one can write this formula as:

$$C \quad = \quad 1 - H(X),$$

where X consists of two messages with probabilities p and $1 - p$. One can argue intuitively that this formula makes use of the amount of information lost during transmission on this noisy channel, or one can show this formula mathematically using concepts not introduced here.

Table 3.1 gives values of the channel capacity and was produced by the Java program on page 173:

Exercise: Use a numerical approximation algorithm to find the inverse of the channel capacity formula. Then write a Java program implementing the algorithm and printing a table of channel capacities and corresponding probabilities (just the reverse of the above table). The results might look like Table 3.2 .

[Ans: A Java program that uses the simple bisection method to print the table is on page 175.]

Channel Capacity	Probability p	Probability $1-p$
0.00	0.50000000	0.50000000
0.10	0.31601935	0.68398065
0.20	0.24300385	0.75699615
0.30	0.18929771	0.81070229
0.40	0.14610240	0.85389760
0.50	0.11002786	0.88997214
0.60	0.07938260	0.92061740
0.70	0.05323904	0.94676096
0.80	0.03112446	0.96887554
0.90	0.01298686	0.98701314
1.00	0.00000000	1.00000000

Table 3.2 Inverse channel capacity.

3.4 A Simple Error Correcting Code.

Let us work out a simple example of the binary symmetric channel with $p = 0.75$. Keep in mind that this means the error rate is 25%: an extremely high figure. In such a case every fourth bit on the average will be reversed, with the reversals occurring completely at random. The capacity of the binary symmetric channel with $p = 0.75$ is:

$$\begin{aligned} C &= 1 + (1/4)\log_2(1/4) + (3/4)\log_2(3/4) \\ &= 1 + (1/4)(-2) + (3/4)(\log_2 3 - 2) \\ &= 0.18872\ldots \end{aligned}$$

A channel capacity of 0.18872 means that one can at best signal at a rate of slightly less than 19% of the true channel bandwidth, because there are so many errors.

Here is a simple code to improve the error rate: *send each bit 3 times and let the majority decide at the other end*. In this case a single error can be corrected, since if 0 is transmitted and something like 010 is received, the majority rule would interpret the received 3 bits as a 0.

With a success rate of $\frac{3}{4}$, there will be 0 errors in the 3 bits with probability $\frac{27}{64}$, and there will be 1 error also with probability $\frac{27}{64}$. Then 2 errors will occur $\frac{9}{64}$ of the time and 3 errors will occur $\frac{1}{64}$ of the time. Only the last two cases represent an actual error with this triple redundant system, so the new error rate is $\frac{10}{64}$ or about 16%. In summary, by reducing the transmission rate from 100% to 33%, one can reduce the error rate from 25% to 16%. (This is a very simple-minded and inefficient code.)

One could continue in this way, using more and more duplicates and letting the majority rule. (An even number of duplicates is not a good choice because the result is indeterminate in

Repetition Codes, Error Probability is 0.25			
Number of Duplicates	**Transmission Rate**	**Error Rate**	**Success Rate**
1	100%	25%	75%
3	33%	16%	84%
5	20%	10%	90%
7	14%	7%	93%
9	11%	5%	95%
11	9.1%	3.4%	96.6%
25	4.0%	0.337%	99.663%
49	2.04%	0.008%	99.992%
99	1.01%	0.0000044%	99.9999956%
199	0.5025%	1.805E-12%	99.9999999999982%

Table 3.3 Repetition codes.

case equal numbers of 0s and 1s are received.) Table 3.3 gives the results (a Java program that prints the table is given as the answer to the next exercise).

It should be clear that you can get as low an error rate as you like by using more and more duplicates, reducing the transmission rate to near zero. At a little less than the channel capacity (7 duplicates and a transmission rate of 14%), you can get the error rate down to 7%. Shannon's theory says that there are other more complicated codes that will also take the error rate arbitrarily close to zero, while maintaining a transmission rate close to 18%. (These other codes can get very large and complicated indeed. No general and practical codes have ever been discovered.)

Exercise: Find the channel capacity for $p = \frac{2}{3}$ [Ans: 0.08170.] Do up a table for $p = \frac{2}{3}$ similar to the one for $p = \frac{3}{4}$. [Ans: see the program on page 177.]

3.5 A Slightly Better Error Correcting Code.

Instead of having code words just for the two bits 0 and 1, it is possible to find codes for longer blocks of bits: for example, to take two bits at a time and use some distinct code word for each of the four possibilities: 00, 01, 10, and 11. Table 3.4 shows a scheme that encodes blocks of three bits at a time:

If the information bits are designated a_1, a_2, and a_3, then the code word bits are: a_1, a_2, a_3, a_4, a_5, a_6, where $a_4 = a_2 \oplus a_3$, $a_5 = a_1 \oplus a_3$, and $a_6 = a_1 \oplus a_2$.

These code words have the property that any two of them differ from one another in at least three positions. (One says that the *Hamming distance* between any two of them is at least 3.)

If there is a single error in the transmission of a code word, there is a unique code word that

Information Word	Code Word
000	000 000
100	100 011
010	010 101
001	001 110
011	011 011
101	101 101
110	110 110
111	111 000

Table 3.4 Error correcting code.

differs from the transmitted word in only one position, whereas all others differ in at least two positions. For this reason, this code is single error correcting, just like the previous triple code, where each bit is transmitted three times.

Notice a difference between this code and the triple transmission code: this code has a transmission rate of 50%, while the triple code has a rate of only 33.3%, even though both do single-error correction.

In the previous code, after transmitting each bit 3 times, one got a better error rate by transmitting each bit 5 times, or 7 times, or 9 times, etc. The transmission rate went to zero, while the error rate also went to zero. In the same way, one could repeat the codes in this section, and the transmission rates would be slightly better than the earlier ones, though they would still go to zero. What is wanted is a way to keep a high transmission rate while lowering the error rate.

3.6 Shannon's Random Codes.

Claude Shannon proved the existence of arbitrarily good codes in the presence of noise. The codes he constructed are completely impractical, but they show the limits of what is possible.

Law SHANNON-1:
Over a noisy channel it is always possible to use a long enough random code to signal arbitrarily close to the channel capacity with arbitrarily good reliability [known as *Shannon's Noisy Coding Theorem*].

The proof chooses an integer n for the blocksize of the information words — n must be very large indeed. The information words are all 2^n strings of 0s and 1s of length n. Corresponding to each information word one chooses a codeword completely at random. The length of the

codewords must be greater than the blocksize divided by the channel capacity. Then it is possible to prove Shannon's result by choosing n large enough. (Notice that it is possible to choose two identical random codewords. This will contribute another error, but as the blocksize gets large, the probability of these errors will also tend to zero.)

The actual situation is just a little more complicated: a code table chosen at random might be anything at all, and so might not perform well. Shannon's proof of his result takes the average over all possible codes. For large enough n, the average satisfies his theorem, so there must exist an individual specific code table that satisfies his theorem. In practice, for large n a random code is very likely to work.

These codes are impractical because the system requires keeping a table of all the random code words. Assuming the table would fit into memory, encoding could be done efficiently. However, for decoding one must find the code word in the table that most closely matches the received code word (that has errors from the noise on the channel).

Exercise: Create a simulation of Shannon's result for specific parameter values and for a specific random code table (with codewords chosen at random). Use $p = \frac{3}{4}$ for the error probability in the binary symmetric channel. Thus the channel capacity C will be 0.18872. Now try a specific value for the blocksize n, say $n = 15$, so that there will be 32768 information words and 32768 entries in the code/decode table. Shannon's theory says that these random code words must be at least $n/C = 15/0.18872 = 79.483$ bits long. Try a longer length for the code words, say 100 bits, and find the error rate of the simulation in this case. (We know that for n large enough this error rate will be as small as we like in this case, but without some estimates, we don't know how it will work out for $n = 15$.) Try a shorter length, say 65 for the code word lengths. (We know that if the code word lengths are chosen less than n/C the error rate can never tend to zero, no matter how big n is chosen.) Also try the value 80 (or 79) for comparison.

[Ans: See an answer in the next chapter.]

4

The Laws of Cryptography
Visualizing Channel Capacity

This section starts with work on an answer to the exercise at the end of the chapter on *Coding and Information Theory*. The exercise asks for a simulation of Shannon's random codes, so the simulation starts with a random code table. This is not the same as Shannon's proof of his theorem, which uses the average of all possible code tables. In practice however, a single specific large random table will almost certainly show the behavior expected by Shannon's theorem. (I don't know if it's possible to find an analytic solution to this simulation.)

The simulation is a challenge in several ways, since to get a reasonable code, a very large blocksize is required by Shannon's theorem. Large blocksizes require exponentially larger code tables, and these quickly exhaust the memory of any actual system.

Please keep in mind that all this discussion assumes a binary symmetric channel with probability $p = 0.75$, that is, on the average, one out of every four bits transmitted is reversed — an extremely high error rate. The corresponding channel capacity is approximately 0.18872.

Table 4.1 gives sample sizes for the random codeword tables, where the block sizes are just arbitrarily set to multiples of 5, that is 5, 10, 15, 20, 25, . . ., and where the codeword lengths are taken to be at the channel capacity (which would be the minimum possible value for a good code).

In practice, simulation runs need longer codeword lengths, perhaps up to twice the channel capacity values. I was able to run the simulation for blocksize = 20, using up to 30 Megabytes of main memory on an iMac. Hypothetical runs for blocksizes of 25 or 30 would take up to 1 Gigabyte or 40 Gigabytes of memory. (The memory requirements grow exponentially as the blocksize grows: adding 5 to the blocksize makes the memory requirements about 40 times as great.)

Blocksize (bits)	Number of Entries	Min Codeword Length (bits)	Minimum Code Table Size (in K bytes)
5	32	27	0.1 K
10	1 024	53	6 K
15	32 768	80	327 K
20	1 048 576	106	13 894 K
25	33 554 432	132	553 648 K
30	1 073 741 842	159	21 206 401 K

Table 4.1 Capacity Simulation Parameters.

Initial results of simulation runs were discouraging and equivocal. It seemed that I would need a far larger blocksize than my computer could run in order to get reasonable results. For example, the results of runs answering the original questions in the exercise were:

> For a blocksize of 15, runs with code word lengths of 64, 80, and 104 produce respective error rates of 56%, 36%, and 15%.

Thus at substantially under the channel capacity for codeword lengths (104 bits or 131% of capacity), the error rate is still 15%, indicating that the blocksize is not remotely close to a value needed for reasonable accuracy. More discouraging was that these runs gave no hint of a magic channel capacity value behind the scenes.

4.1 Visualizing Shannon's Noisy Coding Theorem.

I continued with simulation runs for block sizes of 5, 10, 15, and 20 and for a variety of codeword lengths (all divisible by 8 for convenience of the simulation program). Eventually I plotted all the data in a graph, and got the astonishing picture shown in Figure 4.1. (The Postscript file that creates the graph can be found in the online material.)

The remarkable graph in Figure 4.1 shows how far away the simulation is from the good results predicted by Shannon's theorem. At the same time, the graph gives a clear idea of what the results would be if one could keep increasing the blocksize: the graph would be increasingly vertical as it crosses the vertical dashed line marking the channel capacity.

This specific graph shows the actual simulation results, with points connected by straight lines. (No curve-fitting was carried out.) The black line (for blocksize = 5), is accurate since each point represents 10000000 trials, but there are only 14 points, so this graph has an angular look. The graphs for blocksizes of 10 and 15 are smoother (because more points are plotted), but less accurate (smaller number of trials per point). Finally, the red graph (blocksize = 20) is somewhat irregular because only 10000 trials per point did not produce the accuracy of the other graphs.

It is important to realize that the graph only illustrates what Shannon's theorem proves. The graph and these simulation results prove nothing, but just give an indication of what might be true.

4.2 The Converse of the Noisy Coding Theorem.

The graph also illustrates another theorem known as the *Converse of Shannon's Noisy Coding Theorem*. Roughly speaking, the converse says that if one is signaling at a fixed rate more than channel capacity (to the left of the vertical dashed line in the picture), and if the block size gets arbitrarily large, then the error rate will get arbitrarily close to 100%. Stated another way, at more than channel capacity, as the block size gets larger and larger, the error rate must get closer and closer to 100%. Contrast this with Shannon's theorem, which says that if one signals at less than channel capacity, and if the block size get arbitrarily large, then the error rate will

get arbitrarily close to 0. In the limit as the block size tends to infinity, the graph will look like a step function: at 100 to the left of the dashed vertical line and at 0 to the right.

The proofs of both of Shannon's theorems are found in books on information theory. In the terms of this course they are very difficult and technical.

The Java simulation program in three files is found on page 179.

Figure 4.1. Simulation of Shannon's Random Codes

(Error Probability = 0.25)

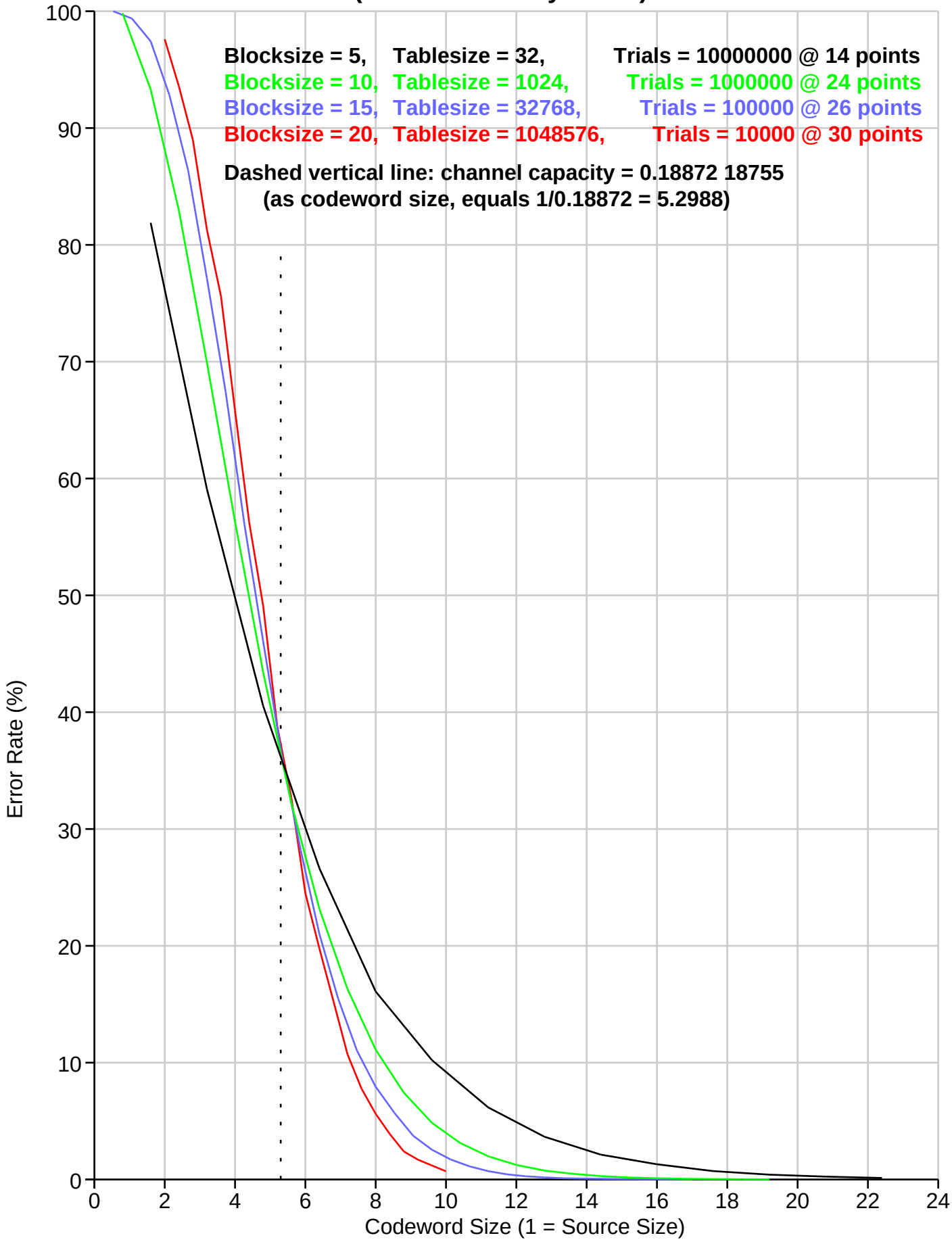


Figure 4.1 Simulation of Shannon's Random Codes

5
The Laws of Cryptography
The Huffman Code for Compression

5.1 Lossless Compression.

Starting with a a file (that is, a message), one wants a *compressed* file smaller than the original from which one can recover the original file exactly. This chapter focuses on *lossless compression*, meaning that not one bit of information is lost during the compression/decompression process.

Claude Shannon proved the following significant result:

Law SHANNON-2:
In the absence of noise, it is always possible to encode or transmit a message with a number of bits arbitrarily close to the entropy of the message, but never less than the entropy [known as *Shannon's Noiseless Coding Theorem*].

To achieve this result, it may be necessary to lump together a large number of messages. In contrast to the proof of Shannon's Noisy Coding Theorem (discussed in the chapter on coding theory), Shannon's Noiseless Coding Theorem has a constructive proof given below as a reasonable method for data compression, though the method is not used any more for actual compression.

Intuitively, a random message has the largest entropy and allows no compression at all. A message that is not random will have some "regularity" to it, some predictability, some "patterns" in the sequence of its bits. Such patterns could be described in a more succinct way, leading to compression.

These concepts provide a new way to describe random sequences: A finite sequence is *random* if it has no succinct representation, that is, any program or algorithm that will generate the sequence is at least as long as the sequence itself. This is the concept of *algorithmic information theory*, invented by Chaitin and Kolmogoroff, which is beyond the scope of this discussion.

Still speaking intuitively, the result of a good compression algorithm is a file that appears random. (If it did not look random, it would be possible to compress it further.) Also, a good compression algorithm will end up relatively close to the entropy, so one knows that no further compression is possible.

Law COMPRESSION-1:
Just like Niven's law (never fire a laser at a mirror), one says: never compress a message that's already compressed.

The joke is to take a large file, Shakespeare's plays, say, and repeatedly compress the file until the result is a single bit. Intuitively, one realizes that there really is information in the plays. They could be compressed, but many bits would still be needed to represent the true information in them. (Of course, the entropy of his plays gives the smallest size of any compressed version.)

Just as with inventors of perpetual motion machines, crackpots fixated on compression regularly announce fantastic compression schemes without knowledge of the limitations imposed by information theory. As I write this (2002), a small company is claiming to compress any random file and recover every bit of the original during decompression, assuming the original file is "large enough". Simply counting the number of possible files to compress and the number of their compressed forms easily shows that these claims are impossible, and the argument requires nothing subtle from information theory. People making claims about perpetual motion or faster than light travel are just suggesting a violation of the accepted laws of physics, something that might be true here or in another universe, but the compression crazies are suggesting a violation of the laws of *logic* — impossible for reasonable people to imagine.

For one very simple example of compression, consider a file with many sequences of blank characters. One needs a special character (or sequence of characters) to represent a sequence of blanks. Use a Ascii blank itself for this purpose. This special character must always be followed by an 8-bit number representing the number of blanks. Thus a single blank is represented by two bytes: a normal blank, followed by a 1. If the file contains many long sequences of blanks, each such sequence shorter than 256 blanks could be represented by two bytes. This might provide a large compression. On the other hand, if the file mostly consists of isolated blanks, the above technique will replace single blanks by two-byte sequences, so the "compression" algorithm will output a larger file than the input.

A later section below presents Huffman's compression algorithm, which in a sense is optimal. Huffman's code provides an explicit solution to Shannon's Noiseless Coding Theorem, but Huffman's algorithm has significant disadvantages. It usually needs an initial statistical analysis of the file itself, and it usually requires transmitting a large decoding table along with the file. For these and other reasons, Huffman's code has been replaced with a large number of other clever compression codes. The complete description is far beyond the scope of this book, but the *.gif* images or the bit stream processed by a modem use very sophisticated algorithms that adapt to the nature of the source file. These methods allow the receiving station to construct a decode table "on the fly" as it carries out decompression.

Images with a *.gif* suffix use the *LZW* compression algorithm, which has an interesting history. Two researches named Lempel and Ziv came up with a remarkable compression algorithm which they published in the scientific literature, though their companies also patented it: the (surprise) *Lempel-Zip* method. Later an employee of Unisys named Welch made minor modifications to Lempel-Ziv to produce the LZW algorithm used in .gif images. Unisys patented the algorithm and after its use became widespread started demanding payments for it. I personally

resent this situation because Unisys and even Welch had relatively little to do with the breakthrough — Lempel and Ziv were the ones with the insight — yet Unisys wants money for its minor modification of a standard algorithm.

5.2 Lossy Compression.

Lossy compression (as opposed to lossless) means that the process of compression followed by decompression does not have to yield exactly the original file. One accepts a loss of quality. (Be careful not to spell the word as "lousy" or "loosey", etc.) When the file is executable computer code, this would be unacceptable — the lower-quality file would not likely be executable any more. If the original file is a picture, however, the recovered file (after compression/decompression) may look nearly as good as the original.

Lossy compression is a huge area with many applications for our society. These include the *Jpeg* standard for individual pictures and the *Mpeg* standard for motion pictures. Both these standards are marvels of compression technology. Jpeg is used in modern digital cameras, allowing the image to be saved in far less memory than the original representation of the color of each pixel in the image. Mpeg is the basis for modern DVD movie players and for satellite transmission of motion pictures.

5.3 Huffman Lossless Compression.

Variable Length and Prefix Codes: Elsewhere in this book codewords only occur that are all the same length (for a given code). However the Huffman code in this section is a *variable length* code. Another well-known but old-fashioned variable length code is the *Morse code* widely used for radio and telegraph transmissions. Table 5.1 gives the code. The idea was to use short codewords for the most commonly occurring letters and longer codewords for less frequent letters.

Morse code presents an immediate decoding problem, since for example, an **N** is **"-."**, but the codewords for **B**, **C**, **D**, **K**, **X**, and **Y**, also start with **"-."**. In fact, the code for **C** is the same as the code for **N** repeated twice. Just given a sequence of dots and dashes, it is not possible to uniquely break the sequence into letters. For this reason, Morse code requires an extra "symbol": a short pause between letters. (There is a longer pause between words.)

The Huffman code that is the subject here does not have this problem. It is a *prefix code*, meaning that no codeword is a prefix of another codeword. Not only is it possible to separate any string of 0s and 1s uniquely into codewords, but the decoding is very easy, since a unique entry always matches the next part of the input. There are other codes that do not have the prefix property, but that are nevertheless uniquely decodable. Such codes are not desirable because of the difficulty of decoding.

The Huffman code starts with a sequence of symbols (a "file") and computes the percent frequency of each symbol.

Example 1. For example, if the sequence (or file) is **aaaabbcd**, then the frequency table is:

International Morse Code							
Letter	Morse		Letter	Morse		Digit	Morse
A	.-		N	-.		0	-----
B	-...		O	---		1	.----
C	-.-.		P	.--.		2	..---
D	-..		Q	--.-		3	...--
E	.		R	.-.		4	....-
F	..-.		S	...		5	.....
G	--.		T	-		6	-....
H	....		U	..-		7	--...
I	..		V	...-		8	---..
J	.---		W	.--		9	----.
K	-.-		X	-..-			
L	.-..		Y	-.--			
M	--		Z	--..			

Table 5.1 The Morse Code.

```
Symbol: a, Weight: 0.5
Symbol: b, Weight: 0.25
Symbol: c, Weight: 0.125
Symbol: d, Weight: 0.125
```

The Huffman algorithm starts with a simple list of each symbol, regarding the list as a sequence of single-element trees with the symbol and the frequency at the root node of each tree (the only node in this case). Then the algorithm repeatedly combines the two *least frequent* root nodes as left and right subtrees of a new root node with symbol and frequency the sum of the two previous frequencies. (I use the symbol `@` as the symbol for the root of the combined tree.)

Thus, referring to Figure 5.1, the first step combines the single-node trees for letters **`c`** and **`d`**, each with frequencies **`0.125`**, into a single tree, with subtrees the trees above, and a root node with frequency **`0.125 + 0.125 = 0.25`**. Now this process is repeated with the new node and the old node for the letter **`b`** with frequency **`0.25`**. The final combination yields a single tree with frequency **`1.0`**.

If there are multiple choices for the “least frequent” root nodes, then make any choice. The resulting Huffman trees and the resulting codes may not be the same, but the average code lengths must always work out the same.

So step 1 in Figure 5.1combines **`c`** and **`d`**, yielding a new root node of frequency **`0.25`**. Then step 2 combines the result of step 1 with **`b`**. That result is combined in step 3 with **`a`**, to give a single root node with frequency **`1.0`**.

In the final part of the algorithm, one heads down the tree from the root as shown above, building a code string as one goes, adding a **`0`** as one goes up and a **`1`** as one goes down.

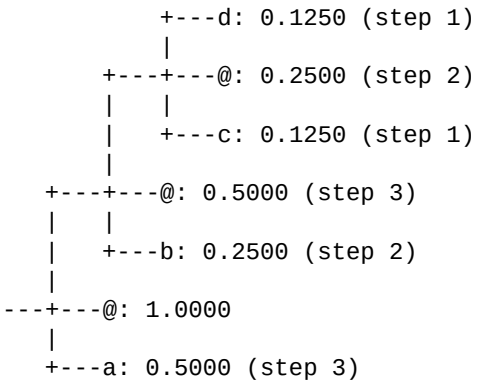


Figure 5.1 Huffman Tree: Entropy equal average code length.

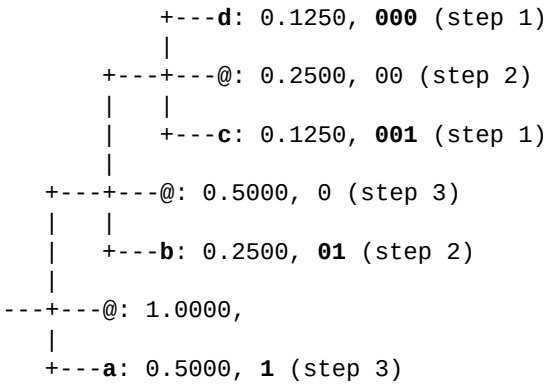


Figure 5.2 Huffman Tree: Codewords.

The resulting binary codes for symbols are shown in bold in Figure 5.2(note the codes for intermediate nodes also):

To encode using the result of the Huffman algorithm, one makes up a *code table* consisting of each symbol followed by the corresponding codeword. To encode, look up each symbol in the table, and fetch the corresponding codeword. (Encoding can be efficient if the table is arranged so that binary search or hashing is possible.)

```
Symbol: a, Codeword: 1
Symbol: b, Codeword: 01
Symbol: c, Codeword: 001
Symbol: d, Codeword: 000
```

The same table could be used for decoding, by looking up successive sequences of code symbols, but this would not be efficient. The process of decoding can be made simple and efficient by using the above Huffman coding tree itself. Start at the root (left side) of the tree

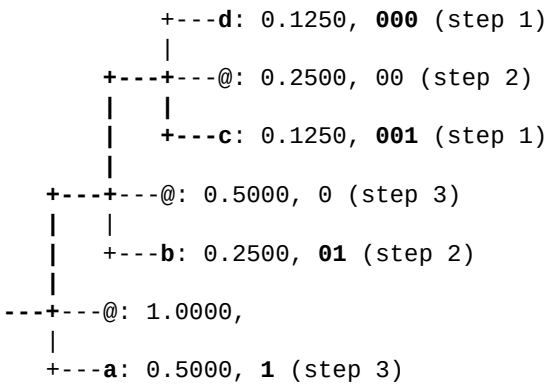


Figure 5.3 Huffman Tree: Decoding.

and process the code symbols **0** and **1** one at a time. For a **0** head upward and for a **1** head downward. When a leaf node (one with no subnodes) is reached, the symbol at that node is the one being decoded. For example, in decoding the string **001**, start at the root, head up because of the first (leftmost) **0**. Then head up again because of the second **0**. Finally head down because of the final **1**. This is now a leaf node holding **c**, so that is the symbol decoded from **001**. The diagram in Figure 5.3shows the path through the tree in boldface:

The entropy of the four symbols above with the given probabilities is **1.75**, and this is exactly the same as the average code length, given by

```
0.5 * 1 + 0.25 * 2 + 0.125 * 3 + 0.125 * 3 = 1.75.
```

Huffman codes are always optimal (the best possible), but this particular code has average code length equal to the entropy, and it is never possible to create a code with shorter average length. Most Huffman codes have average code length greater than the entropy (unless all frequencies are a fraction with numerator and denominator a power of 2). The next simple example shows this.

Example 2. Start with the file **aaaaaabc**. Here is the frequency table, and the tree along with the the code strings is in Figure 5.4:

```
Symbol: a, Weight: 0.75
Symbol: b, Weight: 0.125
Symbol: c, Weight: 0.125
```

In this case, the entropy is **1.061278** while the average code length is **1.25**.

Product Codes: Now suppose one forms the "product" code of the code in Example 2 by considering all possible pairs of symbols and their respective probabilities, which are the products of the probabilities for individual symbols:

```
Symbol: A (for aa), Weight: 0.5625   = 0.75  * 0.75
Symbol: B (for ab), Weight: 0.09375  = 0.75  * 0.125
Symbol: C (for ba), Weight: 0.09375  = 0.125 * 0.75
Symbol: D (for ac), Weight: 0.09375  = 0.75  * 0.125
Symbol: E (for ca), Weight: 0.09375  = 0.125 * 0.75
```

```
          +---c: 0.1250, 00 (step 1)
          |
      +---+---@: 0.2500, 0 (step 2)
      |   |
      |   +---b: 0.1250, 01 (step 1)
      |
------+---@: 1.0000,
      |
      +---a: 0.7500, 1 (step 2)
```

Figure 5.4 Huffman Tree: Entropy less than average code length.

```
Symbol: F (for bb), Weight: 0.015625 = 0.125 * 0.125
Symbol: G (for bc), Weight: 0.015625 = 0.125 * 0.125
Symbol: H (for cb), Weight: 0.015625 = 0.125 * 0.125
Symbol: I (for cc), Weight: 0.015625 = 0.125 * 0.125
```

Figure 5.5gives the corresponding Huffman tree, along with the code words:

With this product code, the entropy and average code length work out to be 2.122556 and 2.15625, but each new symbol (upper-case letter) represents two of the original symbols. Dividing by 2 gives the original value for the entropy, but the average code length (per original symbol, not per new symbol) is 1.078125, which is much closer to the entropy value of 1.061278 than the previous 1.25.

If one takes three symbols at a time, the average code length goes down to 1.0703125. Continuing with four at a time, five at a time, and so forth, it can be proved that the resulting average code lengths get arbitrarily close to the entropy: 1.061278. This, in turn, proves Shannon's Noiseless Coding Theorem, stated earlier.

Even though Huffman's code is optimal (it yields the best possible code for a collection of symbols and frequencies), the other adaptive algorithms (LZ or LZW) usually do a much better job of compressing a file. How can this be, since Huffman is optimal? Suppose one gathers statistics of frequencies of letters in English and creates an optimal Huffman code. Such an analysis does not consider strings of letters in English, but only individual letters. The Huffman product code for two letters at a time just assumes the frequencies are the products of individual frequencies, but this is not true for actual English. For example, for pairs of letters with "q" as the first letter, most pairs have probability 0, while the pair "qu" is about as common as "q" by itself. It is possible to gather statistics about pairs of letters in English and create a Huffman code for the pairs. One could proceed to statistics about longer strings of letters and to corresponding Huffman codes. The resulting Huffman codes would eventually perform better than the adaptive algorithms, but they would require an unacceptable amount of statistical processing, and an unacceptably large code table. In contrast, the adaptive algorithms do not need to transmit a code table, and they eventually adapt to long strings that occur over and over.

A computer implementation of the Huffman algorithm appears on page 183. This software generates printouts of the Huffman trees and other data reproduced above, but it does not read or write actual binary data, nor does it transmit the dictionary (or the frequency table) along

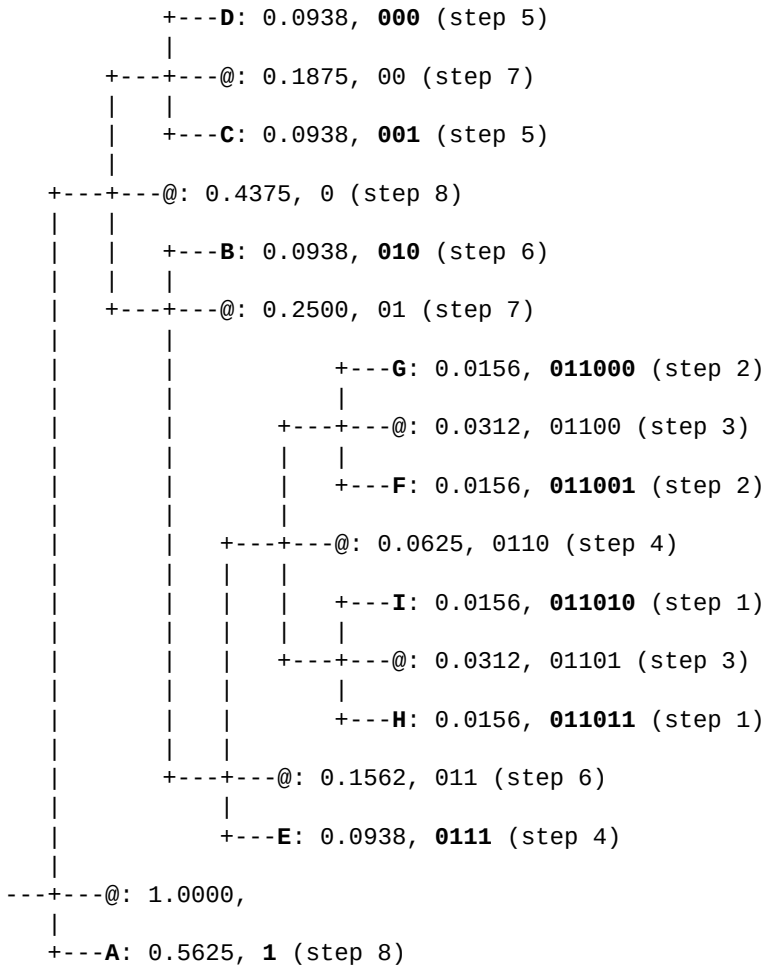


Figure 5.5 Huffman Tree: Product code.

with the data, so that decoding is possible at the other end.

Exercises

1. Devise a simple example where there are different choices for the least trees and where the Huffman algorithm yields different answers. Get an example where there are even two different distributions for the lengths of the codewords. Verify that the average code lengths are the same for the two examples. [Ans: See page 191 for an answer.]

2. After looking at the answer to the previous problem, see if you can create the simplest possible example and argue that there is no simpler example.

3. Write a program that will translate *to* and *from* Morse code. In the coded text, use a blank between the Morse codes for letters and three blanks between the codes for words. Only insert newlines between words.

4. Expand the Huffman code implementation to handle binary files using techniques similar to those of the Hamming code implementation. Comments at the end of the code indicate how to handle these binary files.

6

The Laws of Cryptography
The Hamming Code for Error Correction

6.1 Error correcting codes.

Codes that *correct* errors are essential to modern civilization and are used in devices from modems to planetary satellites. The theory is mature, difficult, and mathematically oriented, with tens of thousands of scholarly papers and books, but this chapter will only describe a simple and elegant code, discovered in 1949.

6.2 Description of the Hamming Code.

Richard Hamming found a beautiful binary code that will *correct* any single error and will *detect* any double error (two separate errors). The Hamming code has been used for computer RAM, and is a good choice for randomly occurring errors. (If errors come in bursts, there are other good codes that are particularly of interest to electrical engineers.) Unlike most other error-correcting codes, this one is simple to understand.

The code uses extra redundant bits to check for errors, and performs the checks with special check equations. A *parity check* equation of a sequence of bits just adds the bits of the sequence and insists that the sum be even (for *even* parity) or odd (for *odd* parity). This chapter uses even parity. Alternatively, one says that the sum is taken modulo 2 (divide by 2 and take the remainder), or one says that the sum is taken over the integers mod 2, Z_2.

A simple parity check will detect if there has been an error in one bit position, since even parity will change to odd parity. (Any odd number of errors will show up as if there were just 1 error, and any even number of errors will look the same as no error.)

One has to force even parity by adding an extra *parity bit* and setting it either to 1 or to 0 to make the overall parity come out even. It is important to realize that the extra parity check bit participates in the check and is itself checked for errors, along with the other bits.

The *Hamming code* uses parity checks over portions of the positions in a block. Suppose there are bits in consecutive positions from 1 to $n-1$. The positions whose position number is a power of 2 are used as check bits, whose value must be determined from the data bits. Thus the check bits are in positions 1, 2, 4, 8, 16, ..., up to the largest power of 2 that is less than or equal to the largest bit position. The remaining positions are reserved for data bits.

Each check bit has a corresponding check equation that covers a portion of all the bits, but always includes the check bit itself. Consider the binary representation of the *position numbers*: $1 = 1_2$, $2 = 10_2$, $3 = 11_2$, $4 = 100_2$, $5 = 101_2$, $6 = 110_2$, and so forth. If the position number

Position	***1***	***2***	**3**	***4***	**5**	**6**	**7**	***8***	**9**	**10**	**11**	**12**	**13**	**14**	**15**	***16***	**17**
Bin Rep	***1***	***10***	**11**	***100***	**101**	**110**	**111**	***1000***	**1001**	**1010**	**1011**	**1100**	**1101**	**1110**	**1111**	***10000***	**10001**
Check:1	***x***		x		x		x		x		x		x		x		x
Check:2		***x***	x			x	x			x	x			x	x		
Check:4				***x***	x	x	x					x	x	x	x		
Check:8								***x***	x	x	x	x	x	x	x		
Check:16																***x***	x

Table 6.1 Position of Parity Check Bits.

Position	***1***	***2***	**3**	***4***	**5**	**6**	**7**	***8***	**9**	**10**	**11**
Binary	***1***	***10***	**11**	***100***	**101**	**110**	**111**	***1000***	**1001**	**1010**	**1011**
Word	***1***	***1***	1	***0***	1	0	1	***0***	1	0	1
Check:1	***1***		1		1		1		1		1
Check:2		***1***	1			0	1			0	1
Check:4				***0***	1	0	1				
Check:8								***0***	1	0	1

Table 6.2 Determining the Check Bit Values.

has a 1 as its rightmost bit, then the check equation for check bit 1 covers those positions. If the position number has a 1 as its next-to-rightmost bit, then the check equation for check bit 2 covers those positions. If the position number has a 1 as its third-from-rightmost bit, then the check equation for check bit 4 covers those positions. Continue in this way through all check bits. Table 6.1 summarizes this.

In detail, Table 6.1 shows the parity checks for the first 17 positions of the Hamming code. (Check bits are in positions 1, 2, 4, 8, and 16, in ***bold italic*** in the table.)

Table 6.2 assumes one starts with data bits 1101101 (in plain type). The check equations above are used to determine values for check bits in positions 1, 2, 4, and 8, to yield the word ***11***1***0***101***0***101 below, with check bits in bold italic here and in Table 6.2 .

Intuitively, the check equations allow one to "zero-in" on the position of a single error. For example, suppose a single bit is transmitted in error. If the first check equation fails, then the error must be in an odd position, and otherwise it must be in an even position. In other words, if the first check fails, the position number of the bit in error must have its rightmost bit (in binary) equal to one; otherwise it is zero. Similarly the second check gives the next-to-rightmost bit of the position in error, and so forth.

Table 6.3 below gives the result of a single error in position 11 (changed from a 1 to a 0). Three of the four parity checks fail, as shown below. Adding the position number of each failing check gives the position number of the error bit, 11 in this case.

The above discussion shows how to get single-error correction with the Hamming code.

Position	***1***	***2***	**3**	***4***	**5**	**6**	**7**	***8***	**9**	**10**	**11**	**Result**
Binary	***1***	***10***	**11**	***100***	**101**	**110**	**111**	***1000***	**1001**	**1010**	**1011**	
Word	*1*	*1*	1	*0*	1	0	1	*0*	1	0	**0 (err)**	
Check:1	*1*		1		1		1		1		0	**1 fail**
Check:2		*1*	1			0	1			0	0	**2 fail**
Check:4				*0*	1	0	1					**- pass**
Check:8								*0*	1	0	0	**8 fail**

Table 6.3 Correcting a 1-bit Error.

One can also get double-error detection by using a single extra check bit, which is in position 0. (All other positions are handled as above.) The check equation in this case covers all bits, including the new bit in position 0. Values for all the other check bits are calculated in any order, but the overall check must come at the end, to determine the 0th check bit.

In case of a single error, this new check will fail. If only the new equation fails, but none of the others, then the position in error is the new 0th check bit, so a single error of this new bit can also be corrected. In case of *two* errors, the overall check (using position 0) will *pass*, but at least one of the other check equations must fail. This is how one detects a double error. In this case there is not enough information present to say anything about the positions of the two bits in error. Three or more errors at the same time can show up as no error, as two errors detected, or as a single error that is "corrected" in the wrong position.

Notice that the Hamming code without the extra 0th check bit would react to a double error by reversing some bogus position as if it were a single error. Thus the extra check bit and the double error detection are very important for this code. Notice also that the check bits themselves will also be corrected if one of them is transmitted in error (without any other errors).

Law HAMMING-1:
The binary Hamming code is particularly useful because it provides a good balance between error correction (1 error) and error detection (2 errors).

6.3 Block sizes for the Hamming Code.

The Hamming code can accommodate any number of data bits, but it is interesting to list the maximum size for each number of check bits. Table 6.4 below includes the overall check bit, so that this is the full binary Hamming code, including double error detection.

For example, with 128 bits or 16 bytes, one gets 120 bits of data and uses 8 bits for the check bits. Thus an error-prone storage or transmission system would only need to devote 1 out

Check Bits	Max Data Bits	Max Total Size
3	1	4
4	4	8
5	11	16
6	26	32
7	57	64
8	**120**	**128**

Table 6.4 Sizes for the Binary Hamming Code.

of each 16 bytes or 6.25% to error correction/detection. Because of these pleasant properties, this code has been used for RAM memory correction (before memory got so reliable).

6.4 A Java implementation of the Hamming Code.

The Hamming code lends itself to efficient implementation in hardware, and even a software implementation just needs bit masks for the various parity checks and attention to detail. The code given in this text in for reference, intended to be simple and understandable, rather than efficient. The method here uses an array of bits (stored in `int` type). This means that actual bit string input must be *unpacked* into the bit array, and then must be *packed* again at the end.

The implementation appears on page 193. The program allows any number of message bits from 1 to 120, with corresponding total number of bits from 4 to 128. For example, if the program uses the bold line in the table, there are 15 bytes of data and 1 byte devoted to check bits. This gives an expansion factor of $16/15 = 1.0666$, using 6.25% of the coded file for check bits. The program does not just implement the lines in the table, but allows all values less than or equal to 120 for the number of message bits.

7
The Laws of Cryptography
Coping with Decimal Numbers

7.1 Decimal Error Detection and Error Correction.

The previous discussions have focused on coding theory applied to binary numbers. In coding theory, the binary number system is regarded as the finite field Z_2 with two elements. The theory can be extended to any finite field, but decimal numbers do not work because there is no finite field with 10 elements. It is possible to apply normal coding theory to the fields of order 2 and 5 separately, and then handle a message modulo 2 and modulo 5 separately, combining results at the end. It is also possible to convert a decimal number to binary and use binary error detection and correction. For decimal numbers, neither of these approaches is satisfactory. In the first case the extra error digits would be different numbers of binary and base 5 digits, which would not fit together and could not be combined into individual checkable decimal digits. More importantly for both approaches, the possible errors in a message will be errors in *decimal* digits. If the number is expressed in binary, a decimal error might change any number of binary bits. If the number is represented using 4 bits per decimal digit, a single decimal error could still change up to 4 bits. For these and other reasons, one needs new decimal-oriented approaches to handle decimal numbers

Law DECIMAL-1:
One needs special methods for decimal-oriented applications; general binary coding theory is not useful.

7.2 Simple Error Detection Schemes.

In applications involving people, numbers with decimal digits are almost always used. (Called the *simian* preference, because humans have 10 fingers – well, most of them anyway, and if you count thumbs.) Humans and machines processing these numbers make *decimal* mistakes in them. The most common mistakes that humans make are to alter a single digit or to transpose two adjacent digits. It is also common to drop a digit or to add an extra digit. Machines can misread a digit, though they would not transpose digits. Applications often add an extra *check digit* to catch as many errors as possible. There is usually also a *check equation* which all digits, including the check digit, must satisfy. On the average, one expects to detect about 90

Suppose one has $n-1$ digits in an application, such as a credit card number: $a_1, a_2, \ldots, a_{n-1}$. A check digit will be added in position 0 as a_0.

A Simple Check: The simplest check equation just adds all digits modulo 10 (form the sum and take the remainder on division by 10) and expects to get 0. So the equation looks like:

$$(a_0 + a_1 + a_2 + \ldots + a_{n-1}) \bmod 10 = 0$$

where "mod" is given in Java or C by the operator **%**. Starting with the data digits, one needs to choose the check digit a_0 so that the check equation will hold. This is easy to do by setting

$$a_0 = (10 - (a_1 + a_2 + \ldots + a_{n-1})) \bmod 10$$

The above simple scheme catches all errors in a single digit. In fact the value of the check equation will tell the amount of the error but not which position is incorrect. This method might be suitable for an optical scanning system that occasionally has trouble reading a digit but knows the position in error. However, this scheme misses all adjacent transpositions, so it is not desirable because transpositions are a common human error.

The U.S. Banking Check: A different and better check scheme is used by U.S. banks for an 8-digit processing number placed on bank checks. They add a check digit and use the check equation:

$$\begin{aligned}(a_0 + 3 * a_1 + 7 * a_2 + a_3 + 3 * a_4 + 7 * a_5 + \\ a_6 + 3 * a_7 + 7 * a_8) \bmod 10 = 0\end{aligned}$$

It is essentially the same scheme to repeat the coefficients 1, 3, and 7 indefinitely. This scheme also catches all single errors and it catches all adjacent transpositions of digits that do not differ by 5. Thus 88.888% of adjacent transposition errors are caught (80 out of 90). (Computer programs later in the book verify this fact; it is also easy to prove mathematically.)

The IBM Check: The most common check now in use, for example on credit card numbers, is often called the "IBM check". Here the check equation is:

$$(a_0 + 2\#a_1 + a_2 + 2\#a_3 + a_4 + \ldots) \bmod 10 = 0$$

where $2\#a_i$ means to multiply a_i by 2 and add the decimal digits. In Java **`2#a[i] = (2*a[i])/10 + (2*a[i])%10`**.

For example, if the account number is 54996, then the check equation without the check digit is:

$$\begin{aligned}&(2\#5 + 4 + 2\#9 + 9 + 2\#6) \bmod 10 = \\ &((1 + 0) + 4 + (1 + 8) + 9 + (1 + 2)) \bmod 10 = \\ &(1 + 4 + 9 + 9 + 3) \bmod 10 = 26 \bmod 10 = 6,\end{aligned}$$

so that the check digit must equal 4 to make the check equation true. Actual credit cards currently have 16 digits and place the check digit on the right, but they treat the other digits

as above, so that the first (leftmost) digit is acted on by $2\#$, while the final 16th digit (the rightmost digit, which is also the check digit) is just added in to the check sum. For example, if a VISA card has number 4270710015912024, then the rightmost 4 is the check digit, chosen so that the check equation will work out to zero:

$$\begin{aligned}&(2\#4+2+2\#7+0+2\#7+1+2\#0+0+\\&2\#1+5+2\#9+1+2\#2+0+2\#2+4) \bmod 10 =\\&(8+2+5+0+5+1+0+0+\\&2+5+9+1+4+0+4+4) \bmod 10 = 0\end{aligned}$$

This scheme detects all single-digit errors as well as all adjacent transpositions except for 09 and 90. Thus it catches 97.777% of all adjacent transposition errors (88 out of 90).

The ISBN Check: The ISBN number of a book uses a mod 11 check. The check equation is:

$$(a_0 + 2 * a_1 + 3 * a_2 + 4 * a_3 + \ldots + n * a_{n-1}) \bmod 11 = 0, \quad \text{for } n < 11$$

If $n \geq 11$ just keep repeating the weights from 1 to 10. Actual ISBN numbers use $n = 10$ and write the digits in reverse order. For example if the ISBN number is 0140046569, then the rightmost digit of 9 is chosen so that the following check equation is true:

$$\begin{aligned}&(10 * 0 + 9 * 1 + 8 * 4 + 7 * 0 + 6 * 0 +\\&5 * 4 + 4 * 6 + 3 * 5 + 2 * 6 + 9) \bmod 11 = 0\end{aligned}$$

The check catches all single errors and all transpositions, whether adjacent or not. (If $n \geq 11$ then transpositions are caught if they are no more than 10 positions apart.) Unfortunately, in this check the check "digit" has a value from 0 to 10, and the ISBN number uses an X to represent 10. (I guess they were thinking of "ten" in Roman numerals.) Because the check calculates numbers modulo 11 and requires an extra symbol for the "digit" 10, it is not properly speaking a decimal check at all. Here is an ISBN number with an X in it: $374661046X$ (the ISBN for Franz Kafka's *Der Prozess*).

All these checks have disadvantages. The IBM check seems best, except that it *does* miss two transpositions. The ISBN check performs much better, but it has the serious disadvantage of an extra X inside every 11th number on the average.

Verhoeff's Scheme: The next chapter presents a more interesting decimal error detecting scheme developed by a Dutch mathematician Verhoeff: ¡a href="verhoeff.html"¿Verhoeff's Check¡/a¿. This method has a section to itself because it is more challenging mathematically.

7.3 Error Detection Using More Than One Check Digit.

For a very low error rate, one could use *two* check digits and a modulo 97 check, with weights successive powers of 10. This check catches 100% of all errors on Verhoeff's list (except for

additions or omissions), and 99.94% of adjacent double errors. Overall, it catches 99% of all errors, compared with 90% for the single check digit schemes.

Keep in mind that the mod 97 check "digit" itself would be represented as *two* decimal digits. These two digits would be protected against all double errors. However, a double error would only be caught 99% of the time if it involved adjacent digits where one is a check digit (part of the mod 97 check digit), and the other is a data digit.

For even better performance, one could use a modulo 997 or a modulo 9973 check, again with weights successive powers of 10, and with 3 or 4 check digits, respectively. (The numbers 97, 997, and 9973 are each the largest prime less than the respective power of 10.)

7.4 Decimal Error *Correction*.

The Hamming code can be carried out modulo any prime number, in particular modulo 11, using the field Z_{11}. This would allow single error *correction*. Such a check would work with *base* 11 digits, but one would just treat the base 10 number as if it were base 11. However, any check digits might have the additional value of 10, so there must be some way to represent this value, such as X (as in ISBN numbers) or A (as in hexadecimal).

The Hamming mod 11 Code With Two Check Digits: With two mod 11 check digits and up to 9 data digits, the code would use the following two check equations:

$$(a_1 + 2 * a_2 + 3 * a_3 + \ldots + 9 * a_9 + 10 * a_{10}) \bmod 11 = 0$$
$$(a_0 + a_1 + a_2 + \ldots + a_9 + a_{10}) \bmod 11 = 0$$

Here the two check digits are a_1 in the first equation and a_0 in the second. One starts with data digits a_2 through a_{10}, then determines the check digit a_1 so that the first check equation is true, and then determines the check digit a_0 so that the second check equation is true. To correct any single error, get the amount of the error from the second check equation and the position of the error from the first.

This system will correct any single error (including errors in the check digits themselves). Unlike the binary Hamming code, this code will interpret many double errors as if they were some other single error.

The Hamming mod 11 Code With Three Check Digits: Nine data digits are probably too few for current applications, but the Hamming code will work fine with a third check digit, allowing up to 118 data digits. Again, such a system would correct any single error, including a single error in one of the check digits. Keep in mind that all these mod 11 checks require the possibility of an X (or something else) as the value for any of the check digits.

Here the digits go from a_0 up to as high as a_{120}. The check digits are a_0, a_1, and a_{11}. (An actual implementation might collect these three digits at one end.) There are three check equations:

$$
\begin{aligned}
(&0a_0+ \;\; 1a_1+ \;\; 2a_2+ \;\; 3a_3+\ldots+ \;\; 9a_9+ \;\; 10a_{10}+\\
&0a_{11}+ \;\; 1a_{12}+ \;\; 2a_{13}+ \;\; 3a_{14}+\ldots+ \;\; 9a_{20}+ \;\; 10a_{21}+\\
&\ldots \quad +\\
&0a_{110}+1a_{111}+2a_{112}+3a_{113}+\ldots+9a_{119}+10a_{120}) \bmod 11 = 0
\end{aligned}
$$

$$
\begin{aligned}
(&0\,a_0+ \;\; 0\,a_1+ \;\; 0\,a_2+ \;\; 0\,a_3+\ldots+ \;\; 0\,a_9+ \;\; 0\,a_{10}+\\
&1\,a_{11}+ \;\; 1\,a_{12}+ \;\; 1\,a_{13}+ \;\; 1\,a_{14}+\ldots+ \;\; 1\,a_{20}+ \;\; 1\,a_{21}+\\
&\ldots \quad +\\
&10\,a_{110}+10\,a_{111}+10\,a_{112}+10\,a_{113}+\ldots+10\,a_{119}+10\,a_{120}) \bmod 11 = 0
\end{aligned}
$$

$$(a_0 + a_1 + a_2 + a_3 + \ldots + a_{119} + a_{120}) \bmod 11 = 0$$

The above equations explicitly show multipliers by 0 to help keep the pattern clear. Start with up to 118 data digits a_2, a_3, ..., a_{10}, a_{12}, ..., a_{119}, a_{120}. In either order, determine the check digits a_1 and a_{11} using the first two equations. Then determine the check digit a_0 using the third equation. If there are fewer than 118 data digits, just set the remaining ones equal to zero in the equations above (and leave them off in transmission and storage). As before, the third equation gives the value of the error, and the first two equations give the location of the error. If p is the location of the error ($p < 121$), then the first equation gives $p\%11$, while the second gives $p/11$, and together these give p.

Suppose this scheme were used for current credit card numbers. These use the IBM scheme and have 15 data digits (expandable to any number of digits) and one check digit. One would replace these with 15 data digits (expandable to 118 data digits) and 3 mod 11 check digits. Thus the new numbers would have two disadvantages compared with the old: two extra digits and the possibility of an X in three of the digits. In exchange, the system would correct any single error, and would have an extremely low error rate for random errors ($1/11^3 = 0.075\%$ compared with the current error rate of 10%).

With the full 121 digits, this scheme interprets any two errors as if they were a single error in some other position. However, some of these miscorrections try to put a ten, that is, an X into a position other than 0, 1 or 11, and so are recognized as a double error. Thus with 121 digits, about 18.5% of double errors are detected, or 81.5% are undetected.

With only 18 digits, most double errors (75%) will be interpreted as an error in a position greater than 18 and so will be detected as a double error. Also if a double error is interpreted as a "correction" of a position other than 0, 1, or 11 to an X, or if the amount of error comes up as 0, these also represents a detected double error, detecting an additional 9.3%, for a total of 84.3% of double errors detected, or 15.7% undetected.

In summary, the three-digit Hamming mod 11 scheme would have an extremely low error rate if used only for detection. It would allow the correction of any single error. However, for

a size such as 15 data digits, about 15% of double errors would be erroneously interpreted as a single error, compounding the problems. For these and other reasons, it does not seem likely that anyone will want to use this code for practical decimal-oriented applications.

In contrast, the *binary* Hamming code (see the chapter on the Hamming code) corrects any single error and *detects* any double error. In the mod 11 Hamming code above, the overall check gives the amount of the error. With the binary Hamming code, the corresponding check is not needed, since if there is an error, its amount must be 1 (that is, a 1 changed to a 0 or vice versa). Moreover, any double error will still show up as an error according to the other checks, but a double *binary* error appears as *no* error in the overall check. Thus the double error detection works only in the special case of the base **2** Hamming code. Because of this the binary Hamming code has often been implemented for hardware RAM memory. The absence of double error detection in the Hamming mod 11 code, and more importantly, the fact that double errors can mask as a single correctable errors, are fatal flaws.

Law DECIMAL-2:
Hamming codes exist for prime bases other two, but because they do not support double error detection, and because they may misinterpret a double error as a correctable single error, they are not useful.

7.5 Java Implementation of the Schemes.

- *The U.S. Bank Scheme* (all but 10 transpositions detected): see the program on page 203.
- *The "IBM" Scheme* (all but 2 transpositions detected): see the program on page 206.
- *The ISBN Scheme* (all transpositions detected): see the program on page 209.
- *The mod 97 Scheme* (all transpositions detected): see the program on page 212.
- *Hamming mod 11 Code, Test single error correction*: see the program on page 215.
- *Hamming mod 11 Code, Test handling of double errors*: see the program on page 219.

8

The Laws of Cryptography
Verhoeff's Decimal Error Detection

In the past, researchers have given "proofs" that it is impossible for a check to detect all adjacent transpositions (as well as all single errors). It is true that if one uses a simple sum of digits with weights on the individual locations, then such a check is mathematically impossible. However, the more general scheme in this section works.

8.1 Types of Decimal Errors.

In 1969, a Dutch mathematician named Verhoeff carried out a study of errors made by humans in handling decimal numbers. He identified the following principal types:

- *single errors:* a changed to b (60 to 95 percent of all errors)
- *adjacent transpositions:* ab changed to ba (10 to 20 percent)
- *twin errors:* aa changed to bb (0.5 to 1.5 percent)
- *jump transpositions:* acb changed to bca (0.5 to 1.5 percent)
- *jump twin errors:* aca changed to bcb (below 1 percent)
- *phonetic errors:* $a0$ changed to $1a$ (0.5 to 1.5 percent; "phonetic" because in some languages the two have similar pronunciations, as with *thirty* and *thirteen*)
- *omitting or adding a digit:* (10 to 20 percent)

8.2 The Dihedral Group D_5.

Verhoeff had the clever idea to use some other method besides addition modulo 10 for combining the integers from 0 to 9. Instead he used the operation of a group known as the *dihedral group* D_5, represented by the symmetries of a pentagon, which has ten elements that one can name 0 through 9. The discussion here will use $\#$ for the group operation and 0 for the identity element. This is *not* a commutative group, so that $a\#b$ will not always equal $b\#a$. Table 8.1 gives the multiplication table for this group. (Each table entry gives the result of the bold table entry on the left combined with the bold table entry across the top, written left to right. The ***bold italic*** entries show results which are not commutative.)

The reader should realize that D_5 is a complex entity whose elements are somewhat arbitrarily mapped to the integers from 0 to 9, and the group operation $\#$ is not at all like ordinary

#	0	1	2	3	4	5	6	7	8	9
0	0	1	2	3	4	5	6	7	8	9
1	1	2	3	4	0	***6***	***7***	***8***	***9***	***5***
2	2	3	4	0	1	***7***	***8***	***9***	***5***	***6***
3	3	4	0	1	2	***8***	***9***	***5***	***6***	***7***
4	4	0	1	2	3	***9***	***5***	***6***	***7***	***8***
5	5	***9***	***8***	***7***	***6***	0	***4***	***3***	***2***	***1***
6	6	***5***	***9***	***8***	***7***	***1***	0	***4***	***3***	***2***
7	7	***6***	***5***	***9***	***8***	***2***	***1***	0	***4***	***3***
8	8	***7***	***6***	***5***	***9***	***3***	***2***	***1***	0	***4***
9	9	***8***	***7***	***6***	***5***	***4***	***3***	***2***	***1***	0

Table 8.1 Multiplication in the Dihedral Group D_5.

addition or multiplication. Keep in mind that the ten symbols 0 through 9 are the only symbols in the group. When two are combined, you get another of them. There is no number 29, but only separate group elements 2 and 9 that could be combined in two ways: $2\#9 = 6$ and $9\#2 = 7$. There is no concept of order such as $<$ or $>$ in this group. The five numbers 0 through 4 combine in the group just as they do with addition in the group Z_5, but the remaining numbers are quite different, since each of 5, 6, 7, 8, and 9 is its own inverse. (With ordinary addition in Z_{10}, only 5 is its own inverse.)

Figure 8.1 at the end of this chapter shows a way to visualize this dihedral group. It is known as the *group of symmetries of a pentagon*, meaning all rigid motions in the 2-dimensional plane that will transform a regular pentagon onto itself. First there are clockwise rotations by 0, 72, 144, 216, and 288 degrees. These rotations correspond to the group elements 0, 1, 2, 3, and 4, respectively. Any other possible rotation that would take a pentagon to itself is equivalent to one of these, as for example a counter-clockwise rotation by 432 degrees is the same as element 4: a clockwise rotation by 288 degrees. These 5 group elements are illustrated in the left column of Figure 8.1.

The remaining 5 group elements can be visualized as each of the earlier rotations followed by a reflection of the pentagon (flipping it over) along the vertical axis of symmetry. The right column of Figure 8.1 shows the action at the left in each case followed by a reflection about the vertical axis. These actions correspond to group elements 5, 6, 7, 8, and 9, as shown in the figure. These actions show where the entries in Table 8.1 come from. For example, element 9 is a rotation by 288 degrees followed by a vertical reflection. If one wants to see the result of $2\#9$, first do the action specified by 2, that is, a rotation by 144 degrees. Then follow that by the action specified by 9: a further rotation by 288, for a rotation by 144 + 288 = 432, which is the same as a rotation by 72 (group element 1), followed by the vertical reflection, which will result finally in group element 6. This shows that $2\#9 = 6$, as we also see from the table. All other results of combining two group elements can be see to agree with the table entries in the same way. The result of *any* combination of rotations (by any angle that is a multiple of 72

degrees) and of reflections (about any of the 10 axes of symmetry of the pentagon) must end up equivalent to one of the 10 different group elements.

8.3 Verhoeff's Scheme.

If one just used the check equation

$$a_0 \# a_1 \# a_2 \# \ldots \# a_{n-1} = 0$$

where $\#$ is the group operation in D_5, this would be much better than simply adding the digits modulo 10, since in both cases single errors are caught, but in D_5 two-thirds of adjacent transpositions are caught (60 out of 90, represented by the bold italic entries in the table above), whereas ordinary addition catches no transpositions. This suggests that stirring things up a little more would give the answer.

Verhoeff considered check equations of the form

$$f_0(a_0) \# f_1(a_1) \# f_2(a_2) \# \ldots \# f_{n-1}(a_{n-1}) = 0$$

where each f_i is a permutation of the ten digits. He was able to get an excellent check in the special case where f_i is the ith iteration of a fixed permutation f.

As the Java implementation will show, this check is not hard to program and is efficient, but it employs several tables and could not be carried out by hand as all the earlier checks could. This discussion will employ Java notation to keep the subscripts straight.

First the check needs the group operation, defined as a two-dimensional array **`op[i][j]`**, for **`i`** and **`j`** going from **`0`** to **`9`** giving the result of combining the two numbers in D_5 (the same as the table above):

```
int[][] op= {
   {0, 1, 2, 3, 4, 5, 6, 7, 8, 9},
   {1, 2, 3, 4, 0, 6, 7, 8, 9, 5},
   {2, 3, 4, 0, 1, 7, 8, 9, 5, 6},
   {3, 4, 0, 1, 2, 8, 9, 5, 6, 7},
   {4, 0, 1, 2, 3, 9, 5, 6, 7, 8},
   {5, 9, 8, 7, 6, 0, 4, 3, 2 ,1},
   {6, 5, 9, 8, 7, 1, 0, 4, 3, 2},
   {7, 6, 5, 9, 8, 2, 1, 0, 4, 3},
   {8, 7, 6, 5, 9, 3, 2, 1, 0, 4},
   {9, 8, 7, 6, 5, 4, 3, 2, 1, 0} };
```

Then comes an array **`inv`**, where **`inv[i]`** gives the inverse of each digit **`i`** in D_5:

```
int[] inv = {0, 4, 3, 2, 1, 5, 6, 7, 8, 9};
```

Finally, the check requires another two-dimensional array giving the special permutation and iterations of it:

```
int[][] F = new int[8][];
F[0] = new int[]{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}; // identity perm
F[1] = new int[]{1, 5, 7, 6, 2, 8, 3, 0, 9, 4}; // "magic" perm
for (int i = 2; i < 8; i++) { // iterate for remaining perms
```

```
      F[i] = new int[10];
      for (int j = 0; j < 10; j++)
         F[i][j] = F[i-1] [F[1][j]];
   }
```

Now the check equation takes the form:

```
   public static boolean doCheck(int[] a) {
      int check = 0;
      for (int i = 0; i < a.length; i++)
         check = op[check][ F[i % 8][a[i]] ];
      if (check != 0) return false;
      else return true;
   }
```

The check digit can be inserted in position **`a[0]`** using the array **`Inv`**, as is show in the actual implementation later in this section.

Verhoeff's check above catches all single errors and all adjacent transpositions. It also catches 95.555% of twin errors, 94.222% of jump transpositions and jump twin errors, and 95.3125% percent of phonetic errors (assuming a ranges from 2 to 9).

Law DECIMAL-3:
It is possible to use a single check digit to detect all single errors and all adjacent transpositions, *but this method is seldom used*.

I had earlier formulated the above law using the words "and nobody uses this method" at the end. However, Professor Ralph-Hardo Schulz of the Freie Universität in Berlin pointed out that Verhoeff's method was used for serial numbers on German currency, before the introduction of the Euro.

8.4 Java Implementation of Verhoeff's Scheme.

- *Use of the Dihedral Group* (all but 30 transpositions detected): see the program on page 223.
- *Verhoeff's Scheme* (all transpositions detected): see the program on page 226.

Part III
Introduction to Cryptography

9
The Laws of Cryptography
Cryptograms and Terminology

9.1 Cryptograms.

Newspapers in the U.S. have long presented to their readers a special puzzle called a *cryptogram*. The puzzle has taken a quotation in capital letters and substituted another letter for each given letter. The trick is to guess the substitutions and recover the original quotation. Here is an example of a cryptogram:

```
ZFY TM ZGM LMGM ZA HF Z YZGJRBFI QRZBF
ATMQX TBXL WHFPNAMY ZRZGVA HP AXGNIIRM ZFY PRBILX,
TLMGM BIFHGZFX ZGVBMA WRZAL UO FBILX.
YHCMG UMZWL, VZXLMT ZGFHRY
```

It looks like complete gibberish, but if one knows, deduces, or guesses the translation scheme, the *key* for uncovering the quotation, then it is understandable. In this case the key is:

```
Alphabet:      ABCDEFGHIJKLMNOPQRSTUVWXYZ
Translated to: ZUWYMPILBDJRVFHQSGAXNCTKOE
```

Given the quotation, the person making up this cryptogram would write **Z** for each **A**, **U** for each **B**, **W** for each **C**, and so forth. Given the cryptogram as above, one just has to go backwards, changing each **Z** back to **A**, and so forth. In this way, knowing the translation key, it is easy to recover the original quotation:

```
AND WE ARE HERE AS ON A DARKLING PLAIN
SWEPT WITH CONFUSED ALARMS OF STRUGGLE AND FLIGHT,
WHERE IGNORANT ARMIES CLASH BY NIGHT.
DOVER BEACH, MATHEW ARNOLD
```

I have never solved one of these puzzles, but my parents often used to spend an hour or so recovering such quotations. (I must try one sometime.) I remember that my mother would first focus on a word that is a single letter as above, since this letter must be either an **A** or an **I** in ordinary English. After trying **I** for awhile, assume that **Z** is an **A**. Then there is a three-letter word **ZFY** that one now guesses starts with an **A**. This word appears twice, so one might guess that it is **AND**. From this, the last word (last name of an author?) becomes **A_N_D**. There is only one well-known author whose name looks like this, and the quotation is perhaps his most famous one, so one would have solved the puzzle immediately.

As another approach, my mother would check the frequencies of all the letters. In the scrambled quotation (leaving off the last line), they are far from uniform: **Z:11**, **M:10**, **F:9**, **G:8**, **B:7**, **A:7**, etc. Now, **E** is the most frequent letter in English, and it is the next-to-most-

frequent in this quotation. One can also look for words with double letters or with other unusual features. With trial and error, and some luck, one soon has the quotation.

A Java program to produce cryptograms at random, using whatever quotation you wish appears on page 229.

9.2 Terminology from Cryptography.

The "quotation" above is ordinarily called a *message* or *plaintext* in cryptography. The cryptogram is the *ciphertext*. The process of transforming the plaintext into ciphertext is *encryption*, while the reverse process of recovering the plaintext from the ciphertext is *decryption*. The 26 letters used for encryption and decryption is called the *key*. The particular method of translating plaintext to ciphertext is called a *cryptosystem*.

It is important to realize that a single key could transform an arbitrarily long piece of plaintext. Thus instead of keeping a large message secret, one uses cryptography so that one need only keep a short key secret. This leads to a law:

Law CRYPTOGRAPHY-1a:
Cryptography reduces the problem of keeping an arbitrarily long message secret to the problem of keeping a short key secret. What an impressive improvement!

Although the techniques of cryptography are wonderful and powerful, one also needs to realize the limitations of these tools. There still remains something to keep secret, even if it is short:

Law CRYPTOGRAPHY-1b:
Cryptography reduces the problem of keeping an arbitrarily long message secret to the problem of keeping a short key secret. This is little if any improvement, since the problem of keeping something secret still remains.

Keeping keys secret and distributing them to users are fundamental difficult problems in cryptography which this book will take up later.

9.3 Security From Cryptography.

Cryptography has many uses, but first and foremost it provides security for data storage and transmission. The security role is so important that older books titled *Network Security* covered only cryptography. Times have changed, but cryptography is still an essential tool for

achieving security. Network "sniffers" work because packets are in the clear, unencrypted. In fact, we make as little use of cryptography as we do because of a long-standing policy of the U.S. government to suppress and discourage work in the area and uses of it, outside classified military applications. If a transmission line needs security there are still only two basic options: physical security, with fences, rasorwire, and guard dogs, or security using cryptography. (The emerging field of *quantum cryptography* may yield a fundamentally different solution.) With cryptography, it doesn't matter if the line goes across a field or across the world.

Law CRYPTOGRAPHY-2:
Cryptography is the only means of securing the transmission of information where the cost is independent of the distance.

9.4 Cryptanalysis.

The early part of this section regarded a cryptogram as a special (simple) *cryptographic code*. The process of recovering the original quotation is a process of breaking this code. This is called *cryptanalysis*, meaning the analysis of a cryptosystem. In this case the cryptanalysis is relatively easy. One simple change would make is harder: just realize that revealing where the blanks (word boundaries) are gives a lot of information. A much more difficult cryptogram would leave out blanks and other punctuation. For example, consider the cryptogram:

```
OHQUFOMFGFMFOBEHOQOMIVAHZJVOAHBUFJWUAWGKEHDPBFQOVOMLBEDBWMPZZVF
OHQDVAZGWUGFMFAZHEMOHWOMLAFBKVOBGXTHAZGWQENFMXFOKGLOWGFUOMHEVQ
```

One might also present this just broken into groups of five characters for convenience in handling:

```
OHQUF OMFGF MFOBE HOQOM IVAHZ JVOAH BUFJW UAWGK EHDPB FQOVO
MLBED BWMPZ ZVFOH QDVAZ GWUGF MFAZH EMOHW OMLAF BKVOB GXTHA
ZGWQE NFMXF OKGLO WGFUO MHEVQ
```

Now there are no individual words to start working on, so it is a much more difficult cryptogram to break. However, this is an encoding of the same quotation, and there is the same uneven distribution of letters to help decrypt the cryptogram. Eventually, using the letter distributions and a dictionary, along with distributions of pairs of letters, one could get the quotation back:

```
Alphabet:      ABCDEFGHIJKLMNOPQRSTUVWXYZ
Translated to: OXKQFDZGACIVLHEJSMBWPNURTY

ANDWEAREHEREASONADARKLINGPLAINSWEPTWITHCONFUSEDALARMSOFSTRUGGLE
ANDFLIGHTWHEREIGNORANTARMIESCLASHBYNIGHTDOVERBEACHMATHEWARNOLD
```

Even here there are problems breaking the text into words, since it seems to start out with **AND WEAR E HE REASON ....**

Notice that the uneven statistical distribution of symbols is still a strong point of attack on this system. A much better system uses multiple ciphertext symbols to represent the more common plaintext letters. This is called a *homophonic* code, and it can be arbitrarily hard to cryptanalyze if one uses enough additional ciphertext symbols.

The cryptanalysis above assumed that the ciphertext (the cryptogram) was available, but nothing else. However, often much more information is at hand, and good cryptosystems must be resistant to analysis in these cases also. Often the cryptanalyst has both plaintext and matching ciphertext: a *known plaintext* attack. In the case of cryptograms, the code would be known for all letters in that particular plaintext, and this would effectively break the code immediately unless the plaintext were very plain indeed. Sometimes the cryptanalyst can even choose the plaintext and then view his own choice of plaintext along with the corresponding ciphertext: a *chosen plaintext* attack.

Amateurs in cryptography sometimes think they should keep the method of encryption secret, along with the particular key. This is a bad idea though, because sooner or later the underlying method with be discovered or bought or leaked.

Law CRYPTANALYSIS-1:
The method or algorithm of a cryptosystem must not be kept secret, but only the key. All security must reside in keeping the key secret.

In most of computer science, an algorithm that only works once in a while is no reasonable algorithm at all. The situation is reversed in cryptography because it is intolerable if ciphertext can be decrypted even “once in a while”.

Law CRYPTANALYSIS-2:
Ordinarily an algorithm that only occasionally works is useless, but a cryptanalysis algorithm that occasionally works makes the cryptosystem useless.

People naively think that as computers get faster, it gets easier to break a cryptosystem, but this is actually backwards logic. The utility of cryptography depends on the asymptotic ease of encryption and decryption compared with the asymptotic difficulty of cryptanalysis. Faster machines simply increase this disparity.

Law CRYPTANALYSIS-3:
The faster computers get, the more powerful cryptography gets. [Radia Perlman]

9.5 Inventing Cryptosystems.

As I write this (2002), I just attended a talk and just visited a web site where in each case a entirely new cryptosystem was put forward saying "the set of keys is so large that cryptanalysis by trying them all is entirely impossible." No mention was made of attacks other than brute force ones, not even known plaintext attacks. The cryptograms in this section have $26! = 4.0329 \times 10^{26}$ different keys — the equivalent of an 88-bit binary key. This is far too many keys for a brute force attack, yet a cryptogram is easily broken with only ciphertext available, and is trivial to break under a known plaintext attack.

Law CRYPTANALYSIS-4:
While a large number of possible keys is a requirement for a strong cryptosystem, this does not ensure strength.

Getting a new cryptosystem approved for reasonable use by individuals, companies, and governments is now an involved process. It can even start with a proposal like one of the systems mentioned above, but better is a method like that for the Advanced Encryption Standard (AES), where a number of different systems were proposed and evaluated by a committee of experts and by the larger community. Mathematical evaluations of the strength of a new system are desirable, as well as the "test of time": a long period during which many people worldwide evaluate the system and try to break it. The AES has already undergone an enormous amount of scrutiny, and this examination will continue indefinitely. No new cryptosystem should ever be used for sensitive (unclassified) work, but only systems that have been thoroughly studied.

Law CRYPTANALYSIS-5:
A cryptosystem should only be used after it has been widely and publicly studied over a period of years.

For example, the public key cryptosystem proposed by Diffie and Hellman in 1976 (based on the integer knapsack problem) was broken in its strongest form in 1985. On the other hand, the RSA cryptosystem, proposed in 1978, still resists all attacks. (Most attacks are based on factoring large composite integers, and no efficient algorithm for this problem has been found.)

10
The Laws of Cryptography
Perfect Cryptography
The One-Time Pad

10.1 The Caesar Cipher.

People have used cryptography for thousands of years. For example, the Caesar Cipher, which was used during the time of Julius Caesar, wraps the alphabet from **A** to **Z** into a circle. The method employs a fixed shift, say of **3**, to transform **A** to **D**, **B** to **E**, and so on until **W** to **Z**, **X** to **A**, **Y** to **B**, and **Z** to **C**. Thus a message **ATTACK** becomes **DWWDFN** and appears incomprehensible to someone intercepting the message. (Well, incomprehensible to someone not very smart.) At the other end, one can reverse the transformation by stepping **3** letters in the opposite direction to change **DWWDFN** back to **ATTACK**.

This example illustrates many concepts and terminology from cryptography. The original message is also called the *plaintext*. The transformed message is also called the *ciphertext* or the *encrypted message*, and the process of creating the ciphertext is *encryption*. The process of getting the original message back is called *decryption*, using a *decryption algorithm*. Thus one *decrypts* the *ciphertext*.

The basic method used, moving a fixed distance around the circle of letters, is the *encryption algorithm*. In this case the decryption algorithm is essentially the same. The specific distance moved, **3** in this case, is the *key* for this algorithm, and in this type of *symmetric key* system, the key is the same for both encryption and decryption. Usually the basic algorithm is not kept secret, but only the specific key. The idea is to reduce the problem of keeping an entire message secure to the problem of keeping a single short key secure, following Law C1 in the Introduction to Cryptography.

For this simple algorithm there are only 26 possible keys: the shift distances of 0, 1, 2, etc. up to 25, although 0 leaves the message unchanged, so a key equal to 0 is not going to keep many secrets. If the key is greater than 25, just divide by 26 and take the remainder. (Thus the keys just form the *integers modulo* 26, the group Z_{26} described in the chapter *Cryptographer's Favorites*.)

If an interceptor of this message suspects the nature of the algorithm used, it is easy to try each of the 25 keys (leaving out 0) to see if any meaningful message results – a method of breaking a code known as *exhaustive search*. In this case the search is short, though it still might pose problems if the letters in the ciphertext are run together without blanks between words.

The Caesar Cipher is just a special case of the cryptograms from the previous chapter, since

with a shift of 3 for example, the cyprtogram key is:

```
Alphabet:      ABCDEFGHIJKLMNOPQRSTUVWXYZ
Translated to: DEFGHIJKLMNOPQRSTUVWXYZABC
```

Here is a computer implementation of the Caesar cipher: see page 232.

10.2 The Beale Cipher.

The Beale Cipher is a just simple extension of the Caesar Cipher, but it is easy to use by hand and it provides excellent security.

Consider the Caesar cipher of the previous section, and associate the letters **A** through **Z** with the numbers 0 through 25, that is, **A** is associated with 0, **B** with 1, **C** with 2, and so on until **Z** with 25. One can represent the previous shift of 3 in the example by the letter **D**, so that each letter specifies a shift. A special encryption method called the *Beale cipher* starts with a standard text (the *key* in this case) like the U.S. Constitution **(WE THE PEOPLE . . .)** and with the message to encrypt, say **ATTACK**. Write down the letters of the standard text on one line, followed by the letters of the message on the next line. In each column, the upper letter is interpreted as a shift to use in a Caesar cipher on the letter in the second row. Thus below in the second column, the **E** in the first row means a shift of 4 is applied to the letter **T** in the second row, to get the letter **X**.

```
Standard text (key):  WETHEP
Message:              ATTACK
Encrypted message:    WXMHGZ
```

The person receiving the encrypted message must know what the standard text is. Then this receiver can reverse the above encryption by applying the shifts in the opposite direction to get the original message back. This method will handle a message of any length by just using more of the standard text. Notice that in this example the two **T**s came out as different letters in the encrypted message. For more security, one should not use a standard text as well known as the one in this example. Instead the sender and receiver could agree on a page of a book they both have with them as the start of their standard text.

In fact, the original historical Beale cipher consisted of three messages: one in the clear and the other two encrypted. The first encrypted message used the start of the U.S. Constitution just as above, and told of a buried treasure. The third message was to tell where to find the treasure, but it has never been decrypted. In fact, if the standard text is not known, it can be very hard to cryptanalyze a Beale cipher.

All the security of this system resides with the secrecy of the standard text. There are a number of subtle pitfalls with this method, as with most of cryptography. For example, suppose you make a trip to, ummmm, Karjackistan, and you want to communicate in secret with your friend back home. You buy two copies of a cheap detective novel, and agree on a page as above. The Karjackistan Secret Police might notice the novel you are carrying, and might digitize the entire book and try all possible starting points within its text, as possible ways to decrypt your transmissions. If that didn't work, they could try taking every third letter from every starting point, or try other more complex schemes.

Here is a computer implementation of the Beale cipher: see page 235.

10.3 Perfect Cryptography: The One-Time Pad.

It may be surprising to the reader that there exist simple "perfect" encryption methods, meaning that there is a mathematical proof that cryptanalysis is impossible. The term "perfect" in cryptography also means that after an opponent receives the ciphertext he has no more information than before receiving the ciphertext.

The simplest of these perfect methods is called the *one-time pad*. Later discussion explains why these perfect methods are not practical to use in modern communications. However, for the practical methods there is always the possibility that a clever researcher or even a clever hacker could break the method. Also cryptanalysts can break these other methods using brute-force exhaustive searches. The only issue is how long it takes to break them. With current strong cryptographic algorithms, the chances are that there are no short-cut ways to break the systems, and current cryptanalysis requires decades or millennia or longer to break the algorithms by exhaustive search. (The time to break depends on various factors including especially the length of the cryptographic key.) To summarize, with the practical methods there is no absolute ¡em¿guarantee¡/em¿ of security, but experts expect them to remain unbroken. On the other hand, the One-Time Pad is completely unbreakable.

The One-Time Pad is just a simple variation on the Beale Cipher. It starts with a random sequence of letters for the standard text (which is the key in this case). Suppose for example one uses **RQBOPS** as the standard text, assuming these are 6 letters chosen completely at random, and suppose the message is the same. Then encryption uses the same method as with the Beale Cipher, except that the standard text or key is not a quotation from English, but is a random string of letters.

```
Standard text (random key): RQBOPS
Message:                    ATTACK
Encrypted message:          RJUORC
```

So, for example, the third column uses the letter **B**, representing a rotation of 1, to transform the plaintext letter **T** into the ciphertext letter **U**. The receiver must have the same random string of letters around for decryption: **RQBOPS** in this case. As the important part of this discussion, I want to show that this method is ¡em¿perfect¡/em¿ as long as the random standard text letters are kept secret. Suppose the message is **GIVEUP** instead of **ATTACK**. If one had started with random letters **LBYKXN** as the standard text, instead of the letters **RQBOPS**, then the encryption would have taken the form:

```
Standard text (random key): LBYKXN
Message:                    GIVEUP
Encrypted message:          RJUORC
```

The encrypted message (ciphertext) is the same as before, even though the message is completely different. An opponent who intercepts the encrypted message but knows nothing

about the random standard text gets *no information* about the original message, whether it might be **ATTACK** or **GIVEUP** or any other six-letter message. Given any message at all, one could construct a standard text so that the message is encrypted to yield the ciphertext **RJUORC**. An opponent intercepting the ciphertext has no way to favor one message over another. It is in this sense that the one-time pad is perfect.

In this century spies have often used one-time pads. The only requirement is text (the pad) of random letters to use for encryption or decryption. (In fact, even now I would not want to be found in a hostile country with a list of random-looking letters.) The party communicating with the spy must have exactly the same text of random letters. This method requires the secure exchange of pad characters: as many such characters as in the original message. In a sense the pad behaves like the encryption key, except that here the key must be as long as the message. But such a long key defeats a goal of cryptography: to reduce the secrecy of a long message to the secrecy of a short key. If storage and transmission costs keep dropping, the one-time pad might again become an attractive alternative.

Law PAD-1:
The one-time pad is a method of *key* transmission, not message transmission. [Bob Blakeley]

During World War II the Germans used an intricate machine known as *Enigma* for encryption and decryption. As an important event of the war, British intelligence, with the help of Alan Turing, the twentieth century's greatest computer genius, managed to break this code. I find it sobering to think that if the Germans had not been so confident in the security of their machine but had used a one-time pad instead, they would have had the irritation of working with pad characters, keeping track of them, and making sure that each ship and submarine had a sufficient store of pad, but they would have been able to use a completely unbreakable system. No one knows what the outcome might have been if the allies had not been able to break this German code.

10.4 Random Characters For the One-Time Pad.

Later sections will dwell more on random number generation, but for now just note that the one-time pad requires a truly random sequence of characters. If instead, one used a random number generator to create the sequence of pad characters, such a generator might depend on a single 32-bit integer seed for its starting value. Then there would be only 2^{32} different possible pad sequences and a computer could quickly search through all of them. Thus if a random number generator is used, it needs to have at least 128 bits of seed, and the seed must not be derived solely from something like the current date and time. (Using the current time and date would be terrible, allowing immediate cryptanalysis.)

Exercise: Write a program that will generate duplicate copies of sequences of random characters, for use in a one-time pad. [Ans: for programs that will generate a one-time pad, see page 239. For a pair of wheels that makes the one-time pad easy to use, see page 242.]

11
The Laws of Cryptography
Conventional Block Ciphers

11.1 Conventional Block Ciphers.

The word "conventional" is typically the same as "symmetric key", or "single key", or "classical", indicating something from ancient history. But this is internet time, so in this case an ancient theory is anything dated before 1976. After that came the cryptosystems variously described as "asymmetric", or "two-key", or "public key" (actually mis-labeled, since this latter is a two-key system where one key is public and the other private). Later sections will describe asymmetric cryptosystems.

Block ciphers are in contrast to *stream* ciphers. A stream cipher handles one bit of key and one bit of plaintext at a time, usually combining them with an exclusive-or to produce one bit of ciphertext. Since the encryption cannot just depend on a single bit of key, the system must have *state* or *memory*, so that what is done with one bit depends on what was done before.

Block ciphers take a block of plaintext, whose size depends on the cryptosystem, and use a fixed key of some block length also depending on the cryptosystem, to produce a block of ciphertext, usually the same length as the block of plaintext. Such encryption is "stand-alone" and does not depend on what happened before. (Such a block cipher does not have state or memory.) Having ciphertext block size the same as plaintext block size is important, because then there is no *data expansion* with encryption.

The identical block size of plaintext and ciphertext should not be too small or the system would not be practical. In practice a convenient size is chosen, nowadays usually either 64 or 128 bits. The size of the key must be large enough to prevent brute-force attacks. Thus the ancient (25-year old) Data Encryption Standard (DES) had a 128-bit key and then 64 bits in the original proposals. This was finally cut to 56 bits on a transparently false pretext that eight out of 64 bits should be used for parity. It is now clear that the U.S. National Security Agency (NSA) wanted a key size that they could just barely break using a brute force attack and that no one else could break. A 64-bit key requires 256 times as much work to break as does a 56-bit one, but this is still obviously inadequate in today's world. The new U.S. Advanced Encryption Standard (AES) requires at least 128 bits in a key, and this is now regarded as a minimum desirable size. There has been recent talk that this latter size will also soon give way to a brute-force attack, but this is nonsense — DES is still not trivial to break, and the AES would be at least 2^{72} or roughly $4.722 * 10^{21}$ harder, a very large number. (Note that this estimate is just for a brute-force attack — there might be easier ways to break any given system.) For additional security, the AES also has key sizes of 192 and 256 bits available.

The section describing Claude Shannon's noisy coding theorem used a *random* code for

error correction of data sent over a noisy channel. In cryptography, a random code with no duplicate code words could be considered for a cryptographic code, but the code table would need to be unacceptably large, and the decoding algorithm would be difficult to make efficient. (Here the code table would take the place of the key.) Instead of using random ciphertexts, which is not practical, one wants to have ciphertexts that *appear* to be random.

Encryption itself provides a function from each possible plaintext block to a ciphertext block, with no duplicate values occurring. Similarly decryption gives the inverse mapping, which would not be uniquely defined if there were duplicates. Encryption in this type of cryptosystem is essentially a parameterized collection of encryption functions, one for each key value.

The usual characterization of a block cipher assumes no *memory* or *state* from block to block, that is, each plaintext block is always transformed to the same ciphertext block, assuming the same key. The ciphers with state are *stream* ciphers. If the blocksize is small and there is no state, one could just use attacks in the next section to try to determine the ciphertext corresponding to each possible plaintext block.

Law BLOCKCIPHER-1:
In a block cipher, the blocksize must be large, as least 64 bits, unless the cipher has state.

Thus a cipher with small blocksize needs to actually be a stream cipher. The cipher block chaining mode described later in this chapter is so important because it converts a block cipher to a stream cipher.

11.2 Possible Attacks.

The basic method used for encryption should not be secret, but instead all security should rely on the secrecy of the key. Thus opponents who know everything about the encryption and decryption algorithms should still not be able to break the system unless they know the particular key.

The key is now chosen large enough to eliminate brute-force attacks, but just a large key is not enough to ensure security of a cryptosystem. The opponent may have additional information about the cryptosystem. There are three simple types of attacks against a cipher (along with other fancier ones not described here):

- *Ciphertext only:* here the opponent has access only to any number of ciphertexts. This is the weakest assumption and would always be true. An opponent who cannot even intercept encrypted messages obviously cannot determine the plaintexts.
- *Known plaintext:* This case assumes that an opponent can obtain plaintext/ciphertext pairs. As an example, an embassy might first get permission for a press report by sending the report in encrypted form on one day, and then actually releasing the report the next.

- *Chosen plaintext:* This scenario assumes the opponents can ask for encryption of plaintexts of their choosing, and see the resulting ciphertexts. This is the strongest information normally assumed of opponents: that they essentially have access to an "encryption box" with the key buried inaccessibly inside.

One always wants a cryptosystem to be secure against a chosen plaintext attack, so of course the AES appears to resist this attack. Notice that a cryptosystem may often be strong (resistant to attacks) and yet not require as much work to break as a brute force attack. For example, the cryptograms in an earlier section have keys of size $26!$ or roughly $4.0329 * 10^{26}$. This is a very large key space for brute-force search, but in fact a cryptogram is easily broken.

In addition to trying to break the cipher, an opponent can attempt various *replay* attacks: retransmitting individual blocks or entire messages made up of many blocks. For example, on one day an army might receive an encrypted **RETREAT** message. The next day, when they are supposed to attack, the encrypted **ATTACK** message could be intercepted and replaced with the previous day's message. The army would decrypt the message and think they should again retreat.

There are several ways to protect against these replay attacks. Additional data could be included in each block, such as a sequence number, or the date. However, a better method, explained in detail in the section on "modes of operation" below, makes each encrypted block depend on all the previous blocks.

Other attacks include inserting, deleting or rearranging blocks of ciphertext, as well as *ciphertext searching*: looking for a block that matches the block in another encrypted message, without being able to decrypt the block.

11.3 Meet-In-the-Middle Attack on Double Encryption.

From the beginning, critics of the DES's short key were told that they could use double or triple DES encryption, thus using two or three 56-bit DES keys, and getting an effective key length of 112 or 168 bits. For example, *double encryption* uses two keys K_1 and K_2, encrypting first with the first key, and then encrypting the resulting ciphertext with the second key. A brute-force attack on all pairs of keys would indeed require 2^{112} steps, but such a double encryption should not be regarded as nearly as secure as a cryptosystem designed from scratch with a 112-bit key. In fact, there is a special attack on such a system, called *meet-in-the-middle*.

In order to carry out an attack, one needs enough information to recognize when the attack is successful. One possibility is that the plaintexts might have extra information (such as padding with 0 bits, or representing ASCII characters) that will allow their recognition. More common is a known plaintext attack: Suppose one has several pairs of corresponding plaintext : ciphertext from known plaintext information: $P_1 : C_1$, $P_2 : C_2$, etc. These correspond to double DES encryption using two unknown keys in succession: K_1 and K_2. The objective is to determine these unknown 56-bit keys.

First calculate the ciphertexts $E_K(P_1) = C$ for all 2^{56} possible keys K. These should be stored as a (very large) hash table to allow efficient lookup. Then for each possible key K,

calculate $C' = E_K^{-1}(C_1)$. Look up each C' in the hash table. If an entry is found, then it satisfies: $E_{K'}(P_1) = C'$ and $E_{K''}(C') = C_1$, for some keys K' and K''. This might represent a *false alarm*, so one needs to check these two keys against another plaintext : ciphertext pair as a second check. On the average, in 2^{55} steps, the desired pair of keys will be found.

Thus, instead of 2^{112} steps (which is at present completely unrealistic), one can get by with at most 2^{56} and 2^{56} blocks of storage. Of course, these figures are also very large, but perhaps not so far beyond what is possible. There is also a refinement of this attack called the *time-memory trade-off* that uses 2^x steps of execution and 2^y blocks of storage, where $x + y = 112$. (See the *Handbook of Applied Cryptography* for details.) So even if 2^{56} blocks of storage is not possible, one can trade a smaller amount of storage for a larger amount of execution time.

There are clever ways to use block ciphers, as illustrated in the next section, that will eliminate these meet-in-the-middle attacks.

11.4 Modes of Operation.

The discussion below assumes a fixed conventional (single key) block encryption scheme, such as the Advanced Encryption Standard discussed in a later section. The methods work for any such block cipher.

Electronic Codebook (ECB) Mode: The first method of using a block cipher is called The *Electronic Codebook (ECB) Mode* . In this method, each block is encrypted independently of each other block. This method obviously invites the replay of blocks mentioned earlier.

Cipher Block Chaining (CBC) Mode: This uses an *Initialization Vector* (IV) the size of one block. The IV is exclusive-ored with the first message block before encryption to give the first ciphertext block. Each subsequent message block is exclusive-ored with the *previous* ciphertext block. The process is reversed on decryption. Figure 11.1illustrates the CBC mode (adapted from the *Handbook of Applied Cryptography*):

In the image above, a sequence of plaintext blocks: P_1, P_2, P_3, . . . is being encrypted using a key K and block encryption algorithm E. Step j of the algorithm uses plaintext P_j, key K, and the ciphertext C_{j-1} produced by the previous step. Step 1 requires a special *initialization vector* $IV = C_0$. As shown, step j of decryption uses the inverse decryption algorithm E^{-1} and the same key K, along with the ciphertext block C_j and *the previous ciphertext block* C_{j-1}.

This CBC mode has so many pleasant properties that no one should consider using ECB mode in its place.

Law BLOCKCIPHER-2:
Always use the cipher block chaining (CBC) mode instead of the electronic code book (ECB) mode.

Properties of the CBC mode:

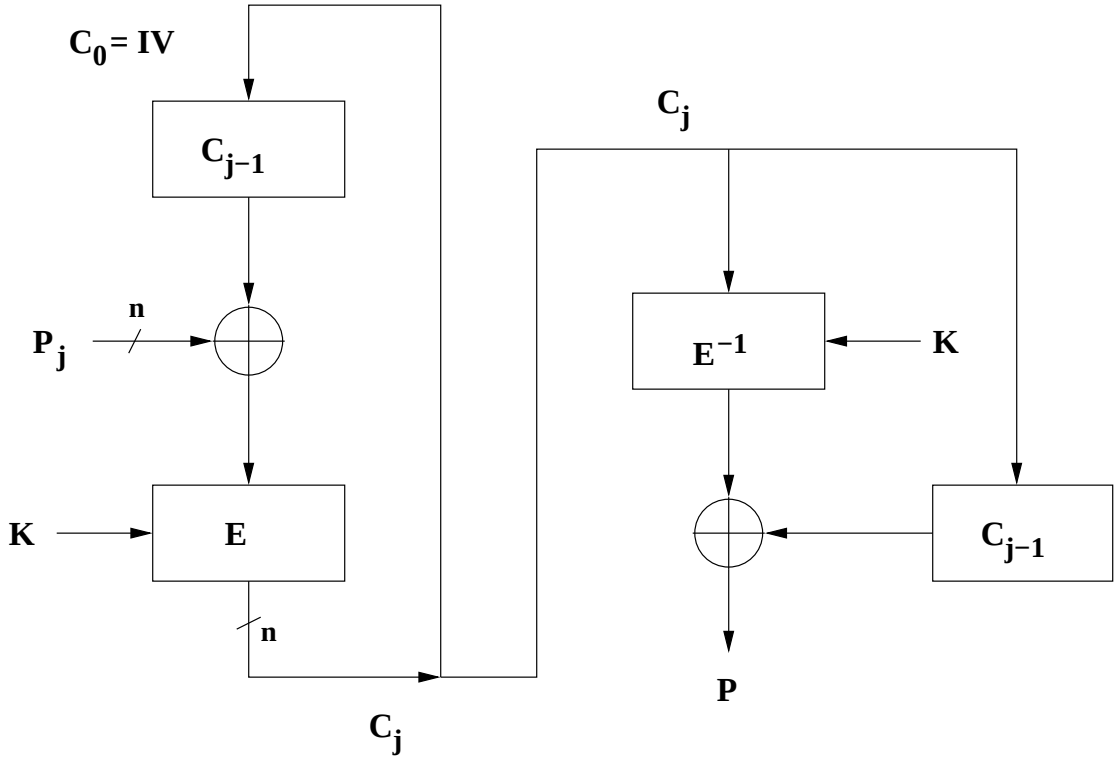


Figure 11.1 The Cipher Block Chaining Mode.

- *What is transmitted:* At each stage, one ciphertext block is transmitted. It must also be arranged that the same secret key K and the same initialization vector IV are at both ends of the transmission, although the IV could be included in an initial transmission.
- *Initialization Vector (IV):* This needs to be the same size as the plaintext and ciphertext. It does not need to be kept secret, but an opponent should not be able to modify it. For less security, the IV could be all zeros, making the first step of CBC the same as ECB. For extra security, the IV could be kept secret along with the key, and then an opponent will not be able to obtain even one pair of plaintext : ciphertext corresponding to the given key.
- *CBC converts a block cipher to a stream cipher:* The CBC mode is essentially a stream cipher that handles one block's worth of bits at a time. The state or memory in the system is the previous ciphertext.
- *Each ciphertext block depends on the current plaintext and on all plaintext that came before:* If a single bit of the initialization vector of of the first plaintext block is changed, then all ciphertext blocks will be randomly altered (50% of bits different from what they were on the average).
- *CBC is secure against the various attacks mentioned earlier:* This includes all the ways of fiddling with and searching for encrypted blocks.
- *Use of CBC as a file checksum:* The final ciphertext block depends on the entire sequence of plaintexts, so it works as a checksum to verify the integrity of all the plaintexts (that they are unaltered and are complete and in the correct order). As with all checksums, there is a very small probability that the checksum will fail to detect an error. Using a 128-bit block, for example, the probability of error is so small that an advanced civilization could use this method for millennia and expect a vanishingly small chance of a single undetected error.
- *Recovery from errors in the transmission of ciphertext blocks:* If one or more bits of ciphertext block C_j are transmitted in error, the error will affect the recovery of plaintext blocks P_j and P_{j+1}. The recovered block P_j will be completely randomized (50% errors on the average), while plaintext block P_{j+1} will only have errors in the same places where C_j has errors. All the remaining plaintexts will come out free of error. Thus the CBC mode is *self-synchronizing* in the sense that it recovers from bit transmission errors with only two recovered blocks affected.

Cipher Feedback (CFB) Mode: This is used for applications requiring that a portion of a block be transmitted immediately. See the *Handbook of Applied Cryptography* for details.

Part IV
Public Key Cryptography

12
The Laws of Cryptography
Public Key Distribution Systems

The first ideas of public key cryptography can best be explained using a clever method for two people to exchange a common secret key using only public communications. The following example is not intended to be practical, but it illustrates the ideas. In fact, it would work in practice, but there are better methods.

12.1 Merkle's Puzzles.

Imagine that person A ("Alice") wants to establish secure communication with a distant individual B ("Bob"). Alice and Bob have made no previous arrangements and can only communicate over a public channel that anyone can listen in on. Another person ("Boris") listens to all their communications and has more computing power than they have. Also Alice and Bob have no special secret methods not known to Boris. However, Boris can only listen in and cannot change messages. In spite of all this, it is still possible for the two of them to establish secure communication that Boris cannot understand. This surprising technique was developed by R.C. Merkle before the introduction of true public key cryptography. It helps prepare students of cryptography for even more surprising ideas.

Alice and Bob must be able to create "puzzles" that are hard but not impossible to solve. For example, they could agree that all but the low 50 bits of a 128-bit AES key would be zero bits. Then they could encrypt two blocks of information with this key. A brute-force breaking of the "puzzle" would mean trying 2^{49} keys on the average. The encrypted information would also start with a block of zero bits, so that they can tell when the puzzle is broken. Suppose it takes Alice and Bob an hour of computer time to break one of these puzzles. It is important to emphasize that Boris also knows exactly how the puzzles are constructed. Suppose Boris has a much more powerful computer and can break a puzzle in just one minute.

Bob creates 14400 of these puzzles, each containing a 128-bit random key in one block and a sequence number identifying the puzzle (a number from 1 to 14400 in binary with leading zeros to make it 128 bits long). Bob transmits all of these puzzles to Alice, in random order. Alice chooses one puzzle at random and breaks it in one hour. She then keeps the random key and sends the sequence number back to Bob. (Boris listens in to this sequence number, but it does him no good because he doesn't know which number goes with which puzzle.) Bob has saved the content of the puzzles (or has generated them pseudo-randomly) so that he can look up the sequence number and find the same random key that Alice has. Boris, their opponent, must break half of the puzzles on the average to recover the common key that Alice and Bob are using. Even with his much faster machines, it still takes Boris $14400/2$ minutes or 5 days on the average to break Alice and Bob's system. Thus Alice and Bob get 5 days of secret

communications. If they want more time, say 50 days, then Bob just needs to send ten times as many messages initially, that is, 144000. If the disparity between their computing power and that of the listeners is even greater, then again Bob must simply send more puzzles initially.

If Boris can actually alter messages (change or inject or delete them) as well as listening in, then he might pretend to be Alice and communicate with Bob, or vice versa. In this case Alice and Bob must be able to describe shared experiences or to communicate shared information to one another to be sure they are not communicating with a stranger (Boris).

If Boris can intercept and change all messages between Alice and Bob, then he could answer both Alice's and Bob's requests to communicate, and pretend to be each to the other. Once Boris has secure communication established with both Alice and Bob, he could relay messages back and forth between them, translating between the two cryptosystems. Then even if they authenticated each other, he would still be listening in. This is called a *man-in-the-middle attack*. In this extreme case, the method does not work. This shows the great care that must be taken with cryptographic protocols. There are more complicated methods relying on a trusted server that will foil this and other attacks

12.2 Commuting Ciphers.

Alice and Bob still want to communicate securely, without having set it up ahead of time, but they would like a simpler and more mathematical system. The ground rules are the same as above: Boris listens in, knows everything they know, and has more computing power.

If they each had their own *secret commuting* cipher, say Alice had E_A and Bob had E_B, then, using a common public integer a, Alice could send $E_A(a)$ to Bob, and Bob could send $E_B(a)$ to Alice. Then Alice computes $E_A(E_B(a))$ and Bob computes $E_B(E_A(a))$. Because the two ciphers *commute*, these quantities are the same. The two people can use this common value or a portion of it to make up a secret key for conventional cryptography. If Boris can't break the cipher, he has no way of knowing this secret value.

Actually, in this system, all that is needed is secret commuting *one-way functions*. Such functions are discussed in a later section, though not commuting ones.

It is a fact that cryptosystems almost never commute. The next section describes the only exception I know of.

12.3 Exponentiation and the Discrete Logarithm.

One can imagine a cryptosystem in which a fixed number a is raised to a power modulo a fixed prime p, and the power is the plaintext m. The result is the ciphertext c. (There is a cryptosystem called the *Pohlig-Hellman System* similar to this but a bit more complicated.) So one fixes a (large) prime p and an integer a that is a generator for multiplication in the field Z_p. (For this generator, see the section on Fermat's Theorem in the "Favorites" chapter.) From a message m, calculate a ciphertext c as described above:

$$c = a^m \bmod p$$

If the quantities above were ordinary real numbers, and the equation were $c = a^m$, then solving for m would give $m = \log_a c$, the "logarithm base a of c". Because of this notation from real numbers, one also refers to m above as the "*discrete* logarithm of c to base a modulo p". It turns out that there is no known efficient way to calculate discrete logarithms, even knowing c, a, and p. Using a brute force approach, one could just try all possible values of m, but there are ways to do better than this, including a "meet-in-the-middle" approach similar to the attack of the same name on double encryption with block ciphers. There are algorithms that are efficient if $p - 1$ has no large prime factors. With the best known approaches, if the prime p is large and random, and if $p - 1$ has a large prime factor, then there is no efficient algorithm to calculate discrete logarithms.

12.4 The Diffie-Hellman Key Distribution System.

With this system, Alice and Bob work with a large public prime p, and a public generator a. Alice chooses a secret integer A, while Bob chooses a secret B. Alice sends $a^A \bmod p$ to Bob, and Bob sends $a^B \bmod p$ to Alice. Then in secret, Alice computes $\left(a^B\right)^A \bmod p = a^{(B*A)} \bmod p$. Similarly in secret, Bob computes $\left(a^A\right)^B \bmod p = a^{(A*B)} \bmod p$, and these two quantities are the same, a common secret value that they can use for further communication. Boris listening in cannot compute either A or B, and cannot discover this common secret.

Just as with the Merkle puzzles, if Boris can do more than listen in on the communications, he could pretend to be Alice to Bob or vice versa, and he could even use a man-in-the-middle attack to make Alice and Bob think they are communicating in secret when they are not. There are more elaborate methods involving authenticated public keys that both Alice and Bob have, and these methods allow them to establish a common secret key even in the presence of an active opponent like Boris.

In order to set up a practical system as above, first choose a "large enough" random integer n. The size needed depends on how fast the computers have become and on whether there has been progress in computing discrete logs; at present n should be at least 1024 bits long. Then test the numbers n, $n + 1$, $n + 2$, etc., until the next prime q is found. It is not enough just to have a random prime, but the prime minus one must have a large prime factor. So just find a larger prime p with q as a factor. For this purpose, test $2 * q + 1$ to see if it is prime. If it is not a prime, start over again with another random n. Finally, one will have a prime p with the property that $p - 1 = 2 * q$, for a prime q of about half the size of p. In order to get a generator, choose an a at random, or start with $a = 2$. Check this a to see if $a^2 \bmod p = 1$ or if $a^q \bmod p = 1$. When an a doesn't satisfy either of these, one knows the value is a generator, and can be used for key distribution. (A random a would also probably work, but not with certainty.)

13
The Laws of Cryptography
Public Key Cryptography

13.1 Beginnings of Public Key Cryptography.

In 1976 Diffie and Hellman started a revolution with their article introducing the concepts of public key cryptography. Others discovered the same concepts earlier while doing classified military applications, but the open publication caught many people's imagination world-wide.

Instead of using a single key for encryption and decryption, the idea is to have one *public* key for encryption (or one *public* encryption algorithm), a key that anyone can access and use. This key will normally be available online from a trusted *key server*, in a way similar to numbers in a phone book.

The key for decryption (or the algorithm for decryption), will be secret and known only to the party for whom the message is encrypted. It must not be feasible to guess or deduce or calculate the decryption key from a knowledge of the encryption key.

Each user has his or her own pair of encryption and decryption keys, distint from one another.

13.2 Structure of a Public Key Cryptosystem.

Each separate user constructs a *public encryption algorithm* E and a *private decryption algorithm* D. (Usually the algorithms are fixed, and the only information needed are *keys* supplied to the algorithms.) Thus Alice has a pair E_A and D_A, and similarly Bob has E_B and D_B.

Each matching pair E and D of algorithms has the properties:

1. *Encryption followed by decryption works:* $D(E(m)) = m$, for any plaintext m for which the algorithm E is defined.
2. *Can encrypt efficiently:* The algorithm E can be calculated "efficiently".
3. *Can decrypt efficiently:* The algorithm D can be calculated "efficiently".
4. *Public and private keys stay that way:* For an opponent ("Boris") who knows E, it is still an "intractible" computation to discover D.

The RSA public key cryptosystem also has the unusual and useful property that decryption works the same as encryption.

5. *Signing followed by verifying works:* The set of messages m is the same as the set of ciphertexts $c = E(m)$, for all m, so that the decryption algorithm can be

applied to a message, resulting in what is called a *signed message* or a *signature*. If $s = D(m)$ is the signature corresponding to some plaintext m, then $m = E(s)$, that is, $m = E(D(m))$, for any message m.

The word "efficient" above means that that calculation uses an acceptable amount of resources, while "intractible" means roughly that this computation uses more resources than the secrecy of the message is worth. (This varies with time, as computations get cheaper.)

Alice makes her public algorithm publicly available, by publishing it or by putting it online, and so does every other user. In practice a public key server in needed to supply authenticated public keys to users. Alice keeps her secret algorithm completely to herself, just as every other user does. (No one else must know the secret algorithm.)

13.3 Modes of Operation for Public Key Cryptosystems.

Here the term "key" is used to mean the same as "algorithm" as explained above. Suppose as above that Alice and Bob have public key and private key pairs: (E_A, D_A) for Alice and (E_B, D_B) for Bob. There must be some key distribution authority that keeps the public keys E_A and E_B and makes them available to anyone requesting them. (Using signature techniques discussed later, the authority can convince a requester that he or she is getting the correct keys.)

Bob sends a *secret* message m to Alice: Bob gets Alice's public key E_A from the authority. Bob calculates $E_A(m) = c$ and sends c to Alice. Only Alice knows the decryption key, so only Alice can compute $D_A(c) = D_A(E_A(m)) = m$. Unless there is some failure in the system, Alice knows that no one intercepting the transmission of c will be able to recover the message m. However, anyone might have sent her this message, and even Bob might have leaked the message to others. Notice that this mode does not require the special property 5 of RSA, but is available for any public key cryptosystem.

Bob *signs* a message m and sends it to Alice: This mode uses the special property 5. above of the RSA cryptosystem. Other systems usually don't have property 5., but it is still possible to create digital signatures in a more complicated way as a later chapter will discuss. Bob uses his secret decryption key on the message itself (rather than on ciphertext). For RSA, the collection of all possible messages is the same as the collection of all possible ciphertexts: any integer less than the fixed integer n used for that instance of RSA. Thus Bob calculates $D_B(m) = s$. At the other end, Alice can retrieve Bob's public key E_B from the key authority and use it to recover the message, by calculating $E_B(s) = E_B(D_B(m)) = m$. Anyone can fetch Bob's public key and do this calculation, so there is no secrecy here, but assuming the system does not break down (that the key authority works and that the cryptosystem is not leaked or stolen or broken), only Bob can have signed this message, so it must have originated with him.

Bob can use this same method to broadcast a message intended for everyone, and anyone can verify using Bob's public key that the message can only have originated with him.

Bob *signs* a *secret* message m and sends it to Alice: This can be done in two ways,

with Bob using his secret key D_B and Alice's public key E_A. (Once again, this assumes RSA with property 5 is used.) Calculate either $E_A(D_B(m)) = c$, or $D_B(E_A(m)) = c'$. In either case, Alice reverses the process, using her secret key D_A and Bob's public key E_B. There is one other slight problem with the RSA cryptosystem, because the maximum size of plaintext or ciphertext is less than an integer n_A for Alice, and is less than a *different* integer n_B for Bob. What happens then depends on which of these two integers is larger, for if n_A is the larger, then $E_A(m)$ might be too large to be handled by D_B using a single block, so one would have the awkward business of breaking this into two blocks. However, in this case one can calculate $D_B(m)$ first, and this is definitely less than the block size of the key E_A. In case the sizes are reversed, just do the two steps above in the opposite order also, so there is no need for more than one block even for a signed and secret message.

Notice that Alice knows the message must have originated with Bob, and that no one else can read it, giving both authentication and secrecy.

Bob uses a *hash function* to sign an *arbitrarily long* message using only one block: Here one just signs the hash code of the message. These matters will be discussed more thoroughly in the chapter on hash functions.

13.4 The Integer Knapsack Cryptosystem.

Diffie and Hellman proposed a specific public key cryptosystem in their 1976 paper: one using an interesting problem known as the *Integer Knapsack Problem*. This particular approach has been broken in most forms. This chapter includes it because it is a good illustration of how public key cryptography works.

The knapsack problem is an example of an *NP-complete* problem — a class of problems that are usable for various purposes, but thought to be intractible. A later chapter describes this theory in much more detail. The knapsack problem occurs as a *decision problem*, that is, one with just a Yes-No answer, and as an *optimization problem*, one with a specific answer.

> **Integer Knapsack Decision Problem:** Given n positive integers $a_i, i = 1, \ldots, n$ and a positive integer s, is there a subset of the a_i that adds up exactly to s? (Yes or No answer.)

The sum has to add up to s exactly — being close counts for nothing. The optimization problem asks just which numbers must be added. Is is an interesting exercise to see that if one had an efficient algorithm to answer the decision problem, then using this algorithm a relatively small number of times, one could get another algorithm that would also answer the optimization problem.

There are many different kinds of instances of the knapsack problem. Some of them are easy to solve and some are hard. The trick with using this problem for public key cryptography is to make breaking the problem look like a hard instance, but it is really a disguised hard instance. If you know the trick, the disguise, you can transform it into an easy instance, and use the secret key to decrypt.

One type of easy instance is one with relatively small numbers for the numbers a_i. Then standard dynamic programming algorithms will solve the problem. Another easy instance, the one we will use here, is for the a_i to form a *superincreasing* sequence, meaning that each value is larger than the sum of all the values that came before.

14
The Laws of Cryptography
The RSA Public Key Cryptosystem

14.1 History of the RSA Cryptosystem.

The history of RSA is still fascinating to me because I watched it unfold. In 1976, as discussed in the previous chapter, Diffie and Hellman introduced the idea of a public key cryptosystem. (Actually, the concept had been discovered earlier in classified work by British and American military researchers, but no one knew this at the time.) Then a 1977 Scientific American article by Martin Gardener talked about a new public key implementation by MIT researchers Rivest, Shamir, and Adelman. This article caught my attention (along with many others), but did not contain the details needed to fully understand the system. A year later the details were finally published and the revolution in cryptography was in full motion. After more than twenty years of research, RSA remains secure and has become the most popular public key cryptosystem.

Law RSA-1:
The RSA cryptosystem is the de facto world-wide standard for public key encryption.

14.2 Description of the RSA Cryptosystem.

The RSA system is an *asymmetric public key cryptosystem* in the terms of the previous chapter. Recall that this means that there are any number of pairs of algorithms (E, D) both defined on the same set of values. E is the *public encryption algorithm* and D is the *private decryption algorithm*. These satisfy:

1. *Encryption followed by decryption works:* If $c = E(m)$ is the ciphertext corresponding to some plaintext m, then $m = D(c)$. (In other words: $m = D(E(m))$, for any message m.)

2. *Can encrypt efficiently:* For any message m, there is an efficient algorithm to calculate $E(m)$.

3. *Can decrypt efficiently:* For any message or ciphertext x, there is an efficient algorithm to calculate $D(x)$.

4. *Public and private keys stay that way:* From a knowledge of E, there is no efficient way to discover D.

5. *Signing followed by verifying works:* The set of messages m is the same as the set of ciphertexts $c = E(m)$, for all m, so that the decryption algorithm can be applied to a message, resulting in what is called a *signed message* or a *signature*. If $s = D(m)$ is the signature corresponding to some plaintext m, then $m = E(s)$. (In other words: $m = E(D(m))$, for any message m.)

As mentioned earlier, RSA is unique in having property 5, which makes signatures using it particularly easy.

Users A, B, ... can create their own pairs (E_A, D_A), (E_B, D_B), ... of RSA key pairs. The encryption algorithms are "published" or made available on a secure public key server, while the decryption algorithms are kept secret from everyone except the originator. The previous chapter has gone over how these can be used.

In RSA, the plaintexts and ciphertexts are just large positive integers, up to a certain size depending on the specific key pair. The underlying algorithms are not secret but only certain information used in them. The RSA system itself is constructed as follows:

Algorithm: RSA cryptosystem construction.

1. Choose random "large" prime integers p and q of roughly the same size, but not too close together.
2. Calculate the product $n = pq$ (ordinary integer multiplication).
3. Choose a random *encryption exponent* e less than n that has no factors in common with either $p - 1$ or $q - 1$.
4. Calculate the (unique) *decryption exponent* satisfying

$$ed \bmod (p-1)(q-1) = 1.$$

5. The **encryption function** is $E(m) = m^e \bmod n$, for any message m.
6. The **decryption function** is $D(c) = c^d \bmod n$, for any ciphertext c.
7. The **public key (published)** is the pair of integers (n, e).
8. The **private key (kept secret)** is the triple of integers (p, q, d).

There is more to the story about each of the above items:

1. At present, "large" means at least 512 bits. For better security each prime should be at least 1024 bits long. There are efficient algorithms for generating random numbers of a given size that are almost certainly prime (see below).

2. n is then either at least 1024 or 2048 bits long.

3. The encryption exponent e can be just 3. If one is using this exponent, then the primes must be such that $p-1$ and $q-1$ are not divisible by 3.
4. The decryption exponent must be calculated, and there are efficient algorithms to do this, but they require a knowledge of p and q (see the chapter on favorite algorithms). The modulus for division, $(p-1)(q-1)$, is the *Euler phi function* ϕ of $n=pq$, where this is a function studied in number theory. One of the function's properties is important in proving that RSA works.
5. There are efficient algorithms for carrying out the modular exponentiation needed here (see below).
6. The same efficient algorithm works here.
7. If it is known that 3 is the encryption exponent, then only n needs to be published.
8. Only d needs to be kept as the secret data for decryption (along with the public n and e). However, p and q can be efficiently calculated from the other numbers, and they are needed anyway for the most efficient form of modular exponentiation. (See the RSA implementation using the Chinese remainder theorem below.)

Some people are surprised that RSA just deals with large integers. So how does it represent data? Suppose the value of n is at least 1024 bits long. This is the same as 128 bytes. In principle then, one can just run 128 bytes of Ascii text together and regard the whole as a single RSA plaintext (a single large integer) to be encrypted or signed. In practice, the protocols will demand additional data besides just the raw message, such as a timestamp, but there is room for a lot of data in a single RSA block.

14.3 RSA Works: Decryption is the Inverse of Encryption.

To show that RSA decryption reverses what RSA encryption does, one only needs to show that:

$$D(E(m)) = m,$$

for any message m, or specifically to show that

$$(m^e)^d \bmod n = m.$$

But recall that

$ed \bmod (p-1)(q-1) = 1$, so that
$(m^e)^d \bmod n = m^{ed} \bmod n = m^{ed \bmod (p-1)(q-1)} \bmod n = m^1 \bmod n = m.$

The last line follows from the chapter on favorite algorithms which shows that the exponent can be reduced modulo

$$\phi(n) = \phi(pq) = (p-1)(q-1).$$

14.4 Java Implementation of the Basic RSA System.

RSA uses arithmetic on integers at least 1024 bits long. RSA has been implemented many times in hardware, but if it is only used for key exchange, a software implementation is fast enough. Any such implementation must start with routines to do extended precision arithmetic on the large integers. Writing such routines is perhaps a reasonable project for an undergraduate CS major as part of one course, with division causing the most grief. (Twenty years ago, I laid out one weekend for such a project, but I ended up devoting more than a week to it.) Many implementations are available, including the Java **`BigInteger`** class, and implementations in symbolic algebra packages such as Maple or Mathematica.

This Java implementation of the RSA cryptosystem uses the Java **`BigInteger`** library class. This arbitrary precision integer arithmetic class has all the methods one needs to implement RSA without difficulty. In fact, it seems as if a number of specialized methods were included just to make RSA implementation easy.

Here are additional comments about this particular implementation:

- *Key generation:*
  - Using public keys of size 1024 bits, it took about 15-60 seconds to generate two sets of keys on a Sun Ultra 10.
  - The key generation has no unusual features. Primes p and q are chosen at random, differing in length by 10-20 bits. (If the primes are too close to $sqrt(n)$, then factoring might be easier than it should be.) The primes are also chosen so that $p-1$ and $q-1$ do not have 3 as a factor, because this implementation uses 3 as the encryption exponent.
  - The only weak point with this key generation that I know of is with the random number generation. For a good implementation, one would need a special generator, with more bits for the seed. (The current generator just uses the number of milliseconds since 1 Jan 1970, and that is clearly insecure.)
- *Encryption and Verification:*
  This uses an exponent of 3. The main known weakness here is that the message m must be bigger than the cube root of n, since otherwise the ciphertext will be m^3, without any modular division. Smaller messages must be padded to make them long enough.

- *Decryption and Signing:*
  This can be sped up using the Chinese Remainder Theorem, as is shown in the next subsection.
- *Combination of Signing and Encrypting:*
  This common combination, used to keep the message secret and to authenticate its source, is done in a simple way that checks the lengths of the public n values first, using the longer one before the shorter one. Otherwise one might need to use two RSA blocks in some cases.
- *The Test:*
  There is just a simple test of this software, though 1024 bits is a realistic size.

For the implementation code and the simple test see page 246.

14.5 Faster RSA Using the Chinese Remainder Theorem.

Here is an altered implementation of the RSA cryptosystem, using the the Chinese Remainder Theorem (CRT) to speed up decryption. Please refer first to the comments in the earlier subsection and to other material about the RSA cryptosystem.

- *Algorithm.*
  The algorithm presented here is described in items 14.71 and 14.75 in the *Handbook of Applied Cryptography*, by Menezes, van Oorschot and Vanstone, CRC Press, 1996. If c is ciphertext, then RSA decryption calculates

$$c^d \bmod n, \text{ where } n = pq.$$

  Suppose one calculates

$$v_1 = c^d \bmod p, \text{ and}$$
$$v_2 = c^d \bmod q$$

  instead. The Chinese Remainder Theorem (and associated algorithm) allows one to deduce $x \bmod (pq)$ from a knowledge of $x \bmod p$ and $x \bmod q$.
  Arithmetic $\bmod p$ should be done $\bmod (p-1)$ in an exponent, because

$$a^{p-1} \bmod p = 1 \text{ (Fermat's theorem).}$$

  Thus we can use the simpler calculation:

$$v_1 = c^{d \bmod (p-1)} \bmod p, \text{and}$$
$$v_2 = c^{d \bmod (q-1)} \bmod q.$$

Finally, following the algorithm 14.71 referred to above, calculate

$$C_2 = p^{-1} \bmod q, \text{ and}$$
$$u = (v_2 - v_1)C_2 \bmod q.$$

The final answer is:

$$c^d \bmod n = v_1 + up.$$

(In calculating u in my implementation, I had to check for a result less than 0, and I had to add q to the result in that case.)

- *Security.*
  The CRT version of decryption requires the primes p and q, as well as the decryption exponent d, so this might seem to be an extra source of insecurity. However, it is simple to factor the modulus n given the decryption exponent d, so no security is lost in using this method.

- *Performance.*
  Theory predicts that the CRT decryption should be 4 times as fast. I tried 600 1024-bit decryptions using a Sun Ultra 10 workstation. The average decryption time for the normal method was about 0.157 seconds per decryption. With the CRT method here, the average decryption time was about 0.046 seconds per decryption, giving a speedup by a factor of about 3.4. The more complicated algorithm has various sources of extra overhead, so it is not surprising that the full speedup by a factor of 4 is not achieved.

- *Summary.*
  If one uses an encryption and verifying exponent of 3 as I am with this software, then these operations are quite fast compared with decryption and signing (at least 100 times faster). A speedup by a factor of 3.4 for decryption and signing is significant. The extra algorithmic complexity is minimal, so no one would want an RSA algorithm without this speedup factor.

Law RSA-2:
RSA encryption should use exponent 3, making it hundreds of time faster, and RSA decryption should use the Chinese Remainder Theorem, making it four times as fast.

One does have to be careful with exponent 3 in two ways: if the message is less than the cube root of m, then the encrypted message will be the same as the message, and if someone obtains ciphertext for a message encrypted under several different public keys, it may be possible to calculate the message.

The implementation code can be found on page 251.

Exercise: Write a "toy" implementation of RSA in the Java language, using the `long` type (64-bit integers) for the calculations. This should be a working implementation in every respect except that the integers cannot be very large.

15
The Laws of Cryptography
Rabin's Version of RSA

15.1 Rabin's Public Key Cryptosystem.

Michael Rabin discovered what I like to call a version of RSA, although it is more properly regarded as a public key cryptosystem in its own right. During its early history, this system was considered of theoretical, but not practical interest because of a "fatal flaw" (a quote from Donald Knuth) that made it vulnerable to a chosen plaintext attack. However, there are ways around the flaw, making this system a real competitor to RSA.

Law RABIN-1:
Rabin's cryptosystem is a good alternative to the RSA cryptosystem, though both depend on the difficulty of factoring for their security.

15.2 Discrete Square Roots.

In the integers modulo n, using both addition and multiplication modulo n, if n in not a prime, then not every non-zero element has a multiplicative inverse. But also of interest here are elements that have a *square root*. The square root of an element a is an element b such that $(b * b)\%n = a$. Some elements have several square roots, and some have none. In fact, number theorists have been interested in these matters for hundreds of years; they even have a special term for a number that has a square root: a *quadratic residue*. Thus this theory is not something new invented just for cryptography.

In elementary algebra, one learns that positive numbers have two square roots: one positive and one irrational. In the same way, for the integers modulo a prime, non-zero numbers that are squares each have two square roots. For example, if $n = 11$ then in Z_{11}, $1^2 = 1$, $2^2 = 4$, $3^2 = 9$, $4^2 = 5$, $5^2 = 3$, $6^2 = 3$, $7^2 = 5$, $8^2 = 9$, $9^2 = 4$, and $10^2 = 1$. Table 15.1 shows those numbers that have square roots:

Notice that 1, 4, and 9 have their "ordinary" square roots of 1, 2, and 3, as well as an extra square root in each case, while 3 and 5 each also have two square roots, and 2, 6, 7, 8, and 10 each have no square roots at all.

Rabin's system uses $n = p * q$, where p and q are primes, just as with the RSA cryptosystem. It turns out that the formulas are particularly simple in case $p\%4 = 3$ and $q\%4 = 3$ (which

Numbers mod 11	
Square	**Square Roots**
1	1, 10
3	5, 6
4	2, 9
5	4, 7
9	3, 8

Table 15.1 Square Roots Modulo a Prime.

Numbers mod 21 = 3*7	
Square	**Square Roots**
1	1, 8, 13, 20
4	2, 5, 16, 19
7	***7, 14***
9	***3, 18***
15	***6, 15***
16	4, 10, 11, 17
18	***9, 12***

Table 15.2 Square Roots Modulo a Product of Two Primes.

is true for every other prime on the average), so the rest of this chapter makes that assumption about the primes used. The simplest such case has $p = 3$ and $q = 7$. In this case Table 15.2 gives the square roots.

Here the "normal" situation is for a square to have four different square roots. However, certain squares and square roots have either p or q as a divisor. In this case, each square has two square roots (shown in bold italic above). Of course, all the numbers not appearing in the left column don't have a square root. A program that creates the above table appears on page 256. This same section gives a table for $p = 7$ and $q = 11$, again satisfying the special Rabin property. In these tables, it looks as if there are a lot of bold italic entries, but in fact there are $(p + q)/2 - 1 = O(p + q)$ such squares with p or q as a factor, while there are $(pq + p + q - 3)/4 = O(pq)$ squares altogether. An actual Rabin instance will use very large primes, so that only a vanishingly small number of them have the divisibility property, and the chances of this happening at random can be ignored.

15.3 Rabin's Cryptosystem.

Each user chooses two primes p and q each equal to 3 modulo 4, and forms the product $n = pq$.

- *Public key*: the number n.
- *Private key*: the numbers p and q.
- *Encryption*: to encrypt a message m, form the ciphertext $c = m^2 \bmod n$.
- *Decryption*: given ciphertext c, use the formulas below to calculate the four square roots modulo n of c: m_1, m_2, m_3, and m_4. One of the four is the original message m, a second square root is $n - m$, and the other two roots are negatives of one another, but otherwise random-looking. Somehow one needs to determine the original message from the other three roots (see below).

In the special case in which both primes when divided by 4 give remainder 3, there are simple formulas for the four roots:

Formulas for the four square roots of a square c. Calculate in order:

a and b, satisfying $ap + bq = 1$, (extended GCD algorithm).

$r = c^{(p+1)/4} \bmod p$.

$s = c^{(q+1)/4} \bmod q$.

$x = (aps + bqr) \bmod n$.

$y = (aps - bqr) \bmod n$.

Now the four square roots are $m_1 = x$, $m_2 = -x$, $m_3 = y$, and $m_4 = -y$. In case m and hence c have p or q as a divisor, the formulas will only yield two square roots, each also with p or q as a factor. For the large primes used in an instance of Rabin, there is a vanishingly small chance of this happening. (Picture the chances that a 512-bit random prime number happens to divide evenly into a message!)

15.4 Cryptanalysis: the Complexity of Rabin's Cryptosystem.

The complexity of Rabin's system (the difficulty of breaking it) is exactly equivalent to factoring the number n. Suppose one has a Rabin encryption/decryption machine that hides the two primes inside it. If one can factor n, then the system is broken immediately, since the above formulas allow the roots to be calculated. Thus in this case one could construct the Rabin machine. On the other hand, if one has access to a Rabin machine, then take any message m, calculate $c = m^2$, and submit c to the Rabin machine. If the machine returns all four roots, then m and $-m$ give no additional information, but either of the other two roots minus m will have one of p or q as a factor. (Take the greatest common divisor of it with n.)

The same proof that breaking Rabin is equivalent to factoring n provides what has been called a "fatal flaw" in Rabin's system. The above argument is just a chosen ciphertext attack.

It is not wise to allow an opponent to mount such an attack, but one would also not want a cryptosystem vulnerable to the attack, which is the case with Rabin's system. (However, see the next section.)

15.5 Redundancy in the Message.

In order to distinguish the true message from the other three square roots returned, it is necessary to put redundant information into the message, so that it can be identified except for an event of vanishingly small probability. The *Handbook of Applied Cryptography* suggests replicating the last 64 bits of any message. Or one could use 0s as the last 64 bits. In these or similar cases, the Rabin machine would be programmed to return only messages with the proper redundancy, and if 2^{-64} is not a small enough margin of error, then just choose more than 64 redundant bits. Then the attack described above doesn't work any more because the Rabin machine will only return a decrypted message (a square root) with the proper redundancy. Thus the Rabin machine returns at most one square root, and possibly none if someone is trying to cheat. (The probability of having two square roots with the given redundancy is again vanishingly small.) Breaking the new system is longer formally equivalent to factoring n, but it is hard to imagine any cryptanalysis that wouldn't also factor n.

Hugh Williams gave another variation of Rabin's cryptosystem that avoids the "fatal flaw" in a mathematically more elegant way.

15.6 A Simple Example.

Here is an example with tiny values for the primes. Of course a real example would use primes in the range from 512 to 1024 bits long, just as in the case of RSA.

Take $p = 7$, $q = 11$, and $n = 77$. Then $(-3) * 7 + 2 * 11 = 1$, so that $a = -3$ and $b = 2$.

Suppose one uses 3-bit messages whose bits are then replicated to give 6 bits, up to the number 63. Messages must be in the range from 1 to 70, so this system of redundancy will work. Start with data bits 101_2 or 5_{10}. The replication gives 101101_2 or 45_{10}. Then $c = m^2 \bmod 77 = 23$. Continuing the calculations, $r = 23^2 \bmod 7 = 4$, and $s = 23^3 \bmod 11 = 1$. Finally, $x = ((-3) * 7 * 1 + 2 * 11 * 4) \bmod 77 = 67$ and $y = ((-3) * 7 * 1 - 2 * 11 * 4) \bmod 77 = 45$. These are two of the four square roots, and the remaining two are $-x \bmod 77 = 10$ and $-y \bmod 77 = 32$. In binary, the four square roots are $67 = 1000011_2$, $45 = 101101_2$, $10 = 001010_2$, and $32 = 100000_2$. Only 45 has the required redundancy, so this is the only number that this modified Rabin machine will return.

Part V
Random Number Generation

16
The Laws of Cryptography
Random Number Generation

16.1 True Random Versus Pseudo-random Numbers.

Random numbers are very widely used in simulations, in statistical experiments, in the Monte Carlo methods of numerical analysis, in other randomized algorithms, and especially in cryptography. The connection with cryptography is very close, since any pseudo-random bit stream along with exclusive-or provides a cryptosystem (though not necessarily a strong system), and any good ciphertext should look like a pseudo-random bit stream (perhaps occurring in blocks). This section focuses on random number generators used in simulation and numerical analysis, but for use in cryptography the recommended random number generators are derived from cryptosystems, both conventional and public key.

Law RNG-1:
Good ciphertext has the appearance of a true-random bit stream.

From the beginning (where "beginning" is the 1940s, the start of the computer age) there was interest in so-called "true" random numbers, that is, numbers generated by a random process in the world. Physical events such as the radioactive decay of particles are unpredictable except for their behavior averaged over time, and so could be used as a source of random numbers, but these events have been difficult to utilize and have been disappointing in practice. More promising recently are possibilities from quantum theory, but such matters are outside the scope of this discussion.

By far the most common source of random numbers is some *deterministic* process, such as a software algorithm. These provide "random-looking" numbers, but the numbers are not *really* random — there is always an exact algorithm for specifying them. This is the reason that researchers now describe such numbers using the word "pseudo", which means "false". These are not *true* random numbers, but for most applications they can be just as useful. Sometimes they can be more useful, as for example when one wants to repeat a simulation with exactly the same random or pseudo-random numbers.

Law RNG-2:
Anyone who uses software to produce random numbers is in a "state of sin". [John von Neumann]

At first one might think that the best way to get random-looking numbers is to use a "random" algorithm – one that does crazy operations, everything imaginable, in every possible order. Donald Knuth tried out such an algorithm as an example, and showed that its performance was no good at all. In its first run, Knuth's "random" algorithm almost immediately converged to a fixed point. Knuth was arguing that one should use science and great care in generating pseudo-random numbers.

Law RNG-3:
One should not use a random method to generate random numbers. [Donald Knuth]

An early suggested source of pseudo-random numbers was an equation which was much later to become a part of modern "chaos" theory. The next chapter describes a generator derived from this equation.

Another early idea for a source of random numbers was to use the bits or digits in the expansion of a transcendental number such as π, the ratio of the circumference of a circle to its diameter.

```
3.14159 26535 89793 23846 26433 83279 50288 41972 ...  (decimal)
3.11037 55242 10264 30215 14230 63050 56006 70163 21122 ...  (octal)
```

It has long been conjectured that this is a very good source of pseudo-random numbers, a conjecture that has still not been proved. In 1852 an English mathematician named William Shanks published 527 digits of π, and then in 1873 another 180 digits for a total of 707. These numbers were studied statistically, and an interesting excess of the number 7 was observed in the last 180 digits. In 1945 von Neumann wanted to study statistical properties of the sequence of digits and used one of the early computers to calculate several thousand. Fortunately for Shanks his triumph was not spoiled during his lifetime, but his last 180 digits were in error and his last 20 years of effort were wasted. Also there was no "excess of 7s". The number π has now been calculated to many billions of places, but the calculation of its digits or bits is too hard to provide a good source of random numbers. The later digits are harder to calculate than earlier ones, although a recent clever algorithm allows calculation of the nth binary (or hexadecimal) digit without calculating the previous ones.

Later work focused on a particularly simple approach using a congruence equation, as described below.

16.2 Linear Congruence Generators.

This approach uses a *linear congruence equation* of the form:

$$x_{n+1} = (k \cdot x_n + a) \bmod m$$

where all terms are integers, k is the *multiplier*, a (usually taken to be 0) is the *increment*, and m is the *modulus*. An initial *seed* is $s = x_0$. Each successive term is transformed into the

m	k
2^{31}	65539 (Beware! RANDU)
$2^{31}-1$	16807
$2^{31}-249$	40692
$2^{31}-1$	48271
$2^{31}-1$	62089911
2^{32}	69069
2^{48}	31167285
2^{64}	6364136223846793005

Table 16.1 Parameters for Linear Congruence Generators.

next, so that a function to return random numbers has the unusual property of automatically cycling itself to the next number. The pseudo-random terms are in the range from 1 to $m-1$. To get (more-or-less) uniformly distributed floating point numbers between 0 and 1, just do a floating point division by m. Assuming that $a = 0$, the quality of the numbers produced depends heavily on k and m.

This type of generator can produce at most $m-1$ different numbers before it starts to repeat. To get this behavior, one can start with a prime number for m and use a *generator* for k so that all $m-1$ numbers will be produced in a repeating cycle, starting with whatever the seed s is.

The old generator provided by the C standard library used 16-bit integers, and so had a maximum possible cycle length of $2^{16} = 65237$ — a ridiculously small cycle, making the generator useless except for toy applications. The C Standard Library still allows this function to return numbers in the range from 0 to 32767, although a larger range is now also possible.

When 32-bit machines first became popular in the 1960s, the multiplier for RANDU, the most common generator at that time, was $2^{16}+3 = 65539$, taken modulo 2^{31}. This multiplier gave extremely poor performance and was eventually replaced by better ones. The most common replacements used the fact that $2^{31}-1$ is a prime and searched for a good muliplier. The multiplier frequently used (starting in 1969) was $7^5 = 16807$ and the constant a was taken to be zero. This generator is quite efficient, and has a cycle length of $2^{31}-2 = 2147483646$. The multiplier was chosen so that various statistical properties of the sequence would be similar to the results for a true random sequence. In the 1970s when I first started using this sequence the cycle length seemed quite long – now it seems short since I have frequently run experiments with hundreds of billions of trials.

Knuth in his chapter on conventional random number generators approves the values $m = 2^{31}-1$ and $k = 16807$ above as "adequate", but he has suggestions for better values, such as those given in Table 16.1 (except for RANDU in the first line).

Knuth suggests other more complicated generators, including one that combines the first two table entries above:

$$x_n = 48271 \cdot x_{n-1} \bmod (2^{31} - 1),$$
$$y_n = 40692 \cdot y_{n-1} \bmod (2^{31} - 249),$$
$$z_n = (x_n - y_n) \bmod (2^{31} - 1),$$

where independent seeds are needed for x_0 and y_0, and the sequence of the z_n make up the output random numbers. The period is nearly the square of the component generators. Knuth also recommends:

$$x_n = (271828183 \cdot x_{n-1} - 314159269 \cdot x_{n-2}) \bmod (2^{31} - 1),$$

which has very good performance and whose period is the square of m. Of course two independent seeds x_0 and x_1 are needed to start the sequence off with x_2.

16.3 Other Distributions.

Random numbers other than the uniform distribution are sometimes needed. The two most common of these are the *normal distribution* and the *exponential distribution*. The easiest way to generate numbers according to these distributions is to transform uniformly distributed numbers to the given distribution.

The formula for the exponential distribution is especially simple:

$$X = -\log_e U$$

If U is uniformly distributed on the interval from 0 to 1, then X will be exponentially distributed with mean 1. For random events which occur once every unit of time on the average, the times between such events will be distributed according to this exponential distribution.

Similarly, there is a more complicated formula giving normally distributed numbers:

$$X_1 = \sqrt{-2 \log_e U_1}\ \cos 2\pi U_2$$
$$X_2 = \sqrt{-2 \log_e U_1}\ \sin 2\pi U_2$$

If U_1 and U_2 are independent and uniformly distributed on the interval from 0 to 1, then X_1 and X_2 will be independent and normally distributed with mean 0 and variance 1. Note that a generator based on these formulas will produce normal random numbers two-at-a-time.

There are other equivalent transformations to the normal distribution that are more efficient (see Donald Knuth, *Seminumerical Algorithms*, 3rd Ed., pages 122–132), but the above formulas should serve all but the most demanding needs. In fact Java has a library method returning normally distributed numbers (**`nextGaussian()`**), and this method uses one of the more efficient transformations than the equations given here

16.4 Commentary.

Knuth has other suggestions for efficient random number generators of high quality, where "quality" is measured by a variety of statistical tests that compare the output of the pseudo-random generator with true random numbers. If for a given test the comparison says the two sets of numbers look similar, then one says the generator "passes" this particular test. A generator that passes all the popular tests that people can devise is of high quality.

However, even generators of high quality are mostly not usable in cryptography. For example, given several successive numbers of a linear congruence generator, it is possible to compute the modulus and the multiplier with reasonable efficiency. One could make the generator more complex in order to resist this attack, but there would still be no proof or assurance of the difficulty of "reverse engineering" the generator. Instead, if generators are needed in cryptographic applications, one is usually created using a conventional cipher such as the Advanced Encryption Standard or using a public key cipher such as RSA or one of its variants. The AES-based generator will be efficient and will satisfy most practical requirements, but the RSA-based systems, while extremely slow compared to the others, have a very strong property of being *cryptographically secure*, a term that means the generator will pass all possible efficient statistical tests. These matters will be defined and discussed in the chapter after the next one.

16.5 Java Implementations of these Generators.

Each of the generators in the previous table is implemented with the Java program on page 259. For simplicity this program uses the Java **`BigInteger`** class for all the implementations. In the resulting code one does not need to worry about overflow, but the generators run very much slower than if they were carefully tuned to the available hardware. However even the inefficient implementation will generate millions of random numbers in just a few minutes on current personal computers, and this will be fast enough for most applications.

In case faster generators are needed, various sources show how to use the 32-bit hardware units directly. Knuth also presents a practical additive generator implemented in C that is very fast. (See Donald Knuth, *Seminumerical Algorithms*, 3rd Ed., page 286.)

For a Java implementation of Knuth's two-seed generator of the previous section, along with transformations to the other distributions, see page 262, which gives an applet displaying the three distributions.

The implementation above uses Java **`long`** type to avoid integer overflow. On conventional machines without 64-bit integers (for example, if programming in C or C++), even the implementation of a simple generator such as the very common

$$x_{n+1} = (16807 \cdot x_n) \bmod (2^{31} - 1)$$

poses problems because the multiply step overflows a 32-bit integer. This generator was usually coded in assembly languages on IBM 360++ machines, where the ready access to all 64 bits of

a product makes implementation easy. On machines with only 32-bit integers, one can break the integers into pieces during the calculations.

Java/C/C++ function: rand2

```
// rand2: version using ints. Works on all hardware, by
//   breaking up numbers to avoid overflow.
int seed2 = 11111;
double rand2() {
   int a = 16807,
       m = 2147483647,
       q = 127773,
       r = 2836;
   int hi, lo, test;
   hi = seed2/q;
   lo = seed2 - q*hi;
   test = a*lo - r*hi;
   if (test > 0) seed2 = test;
   else seed2 = test + m;
   return((double)seed2/(double)m);
}
```

This function, in exactly the form given above, works in Java, C, and C++.

In another approach, one can use the **`double`** type, which includes exact 52-bit integer arithmetic as a special case. If the multiplier is small enough to not overflow a 52-bit integer, then everything can be done using doubles. (In C the operator **`%`** does not work for doubles, while it does in Java.) Here is the C versions of this function. (For this to work, you may need to include a special library such as **`math.h`** to get the function **`floor.h`**.)

C/C++ function: rand1

```
// rand1: version using doubles.  Works on all hardware.
double seed1 = 11111.0;
double rand1() {
   double a = 16807.0,
          m = 2147483647.0;
   double q;
   seed1 = a*seed1;
   q = floor(seed1/m);
   seed1 = seed1 - q*m;
   return(seed1/m);
}
```

This particular generator once represented the minimum standard for a random number generator. I suggest that one now ought to use Knuth's double generator as the minimum standard, shown here in C:

Java function: rand1

```
// rand: version using doubles.  Works on all hardware.
// seed1 = 48271*seed1 mod 2^31 - 1
// seed2 = 40691*seed1 mod 2^31 - 249
// seed  = (seed1 - seed2) mod 2^31 -1
double seed1 = 11111.0;
double seed2 = 11111.0;
```

```
double seed;
double rand() {
   double a1 = 48271.0,      a2 = 40692.0,
          m = 2147483647.0,  m2 = 2147483399;
   double q1,                q2;
   double q, diff;
   seed1 = a1*seed1;         seed2 = a2*seed2;
   q1 = floor(seed1/m);      q2 = floor(seed2/m2);
   seed1 = seed1 - q1*m;     seed2 = seed2 - q2*m2;
   // now combine results
   if ((diff = seed1 - seed2) < 0.0) diff = diff + m;
   q = floor(diff/m);
   seed = diff - q*m;   return(seed/m);
}
```

To convert this to Java, one just needs to write **`Math.floor`** in place of **`floor`**. In the past such a generator might be slow because of all the floating point operations, including 4 floating point divides, but now extremely fast floating point hardware is commonplace.

Exercise: Use **`BigInteger`** to implement Knuth's two more complex generators described above.

17
The Laws of Cryptography
Random Numbers from Chaos Theory

Many interesting results have come from the field known as "chaos theory", but I know of only two published methods for obtaining pseudo-random numbers from this theory. Wolfram proposed a discrete method based on cellular automata, and I made my own proposal based on a common equation from chaos theory called the "logistic equation". This method is unusual because it works directly with floating point numbers, producing uniformly distributed values.

17.1 The Logistic equation.

The logistic equation is the iterative equation

$$f(x) = 4x(1-x), 0 \leq x \leq 1.$$

It is historically interesting as an early proposed source of pseudo-random numbers. Ulam and von Neumann suggested its use in 1947. The equation was mentioned again in 1949 by von Neumann, and much later in 1969 by Knuth, but it was never used for random number generation.

To use the equation to produce random numbers, start with a real number x_0 and define $x_n = f(x_{n-1})$, for $n = 1, 2, \ldots$. In this way one gets a sequence of numbers by just repeatedly applying the function f. The resulting sequence $x_0, x_1, x_2, x_3, \ldots$ will all lie between 0 and 1, but they will not be uniformly distributed in the interval. However, the numbers have a precise algebraic distribution that one can transform to the uniform distribution as shown below.

Another function closely related to the logistic equation is the "tent" function g defined by

$$g(x) = \begin{cases} -2u + 2, \text{ for } 1/2 \leq u \leq 1, \\ 2u, \text{ for } 0 \leq u \leq 1/2 \end{cases}$$

In contrast with f, this function does directly yield numbers that are uniformly distributed. The function f can be transformed to g using the equation

$$T(u) = \sin^2((\pi/2)u).$$

The inverse transformation (following equation) will transform numbers produced by f to those produced by g, in other words, to the uniform distribution.

$$T^{-1}(v) = (2/\pi)\arcsin(\sqrt{v}).$$

Thus the sequence

$$y_n = T^{-1}(x_n), \text{ where}$$
$$x_n = f(x_{n-1}), \text{ for } n = 0, 1, 2, 3, \ldots$$

gives uniformly distributed numbers.

Using the notation $f^2(x) = f(f(x))$, $f^3(x) = f(f(f(x)))$, and so forth, the sequence takes the form

$$y_n = T^{-1}(f^n(x)), \text{ for } n = 0, 1, 2, 3, \ldots$$

This sequence will be uniformly distributed for "almost all" starting values x.

17.2 Behavior in Infinite Precision.

The above equations give uniformly distributed sequences of real numbers if one could use what are called "infinite precision" real numbers, that is, mathematical reals. Even in this case, the sequence $T^{-1}(f^n(x))$ does not at all behave like a true random sequence. A *cycle* occurs in the sequence $f^n(x)$ in case it repeats after finitely many iterations. There are infinitely many finite cycles, even though "almost all" real numbers will not belong to such a cycle. For example, $3/4$ is transformed to itself by f. Starting with a random initial value, one would avoid a finite cycle with probability 1, but even the *influence* of the short cycles will have a bad effect, producing non-randomness. For example, if one starts with an x value very close to $3/4$, successive values will also be close to $3/4$ (though each new value not *as* close), so even this theoretical sequence is definitely not always random-looking.

17.3 Behavior in Finite Precision.

Using actual computer floating point hardware that implements the IEEE **`float`** or **`double`** type (32 bits, about 7 digits of precision, or 64 bits, about 16 digits of precision), the behavior of these functions is quite different from the theoretical behavior in infinite precision. The tent function converges almost immediately to zero, since each iteration adds another low-order 0 bit. The actual logistic equation acts a little better, but still has a relatively short initial run followed by a relatively short cycle. Table 17.1 gives results of experiments run to determine the cycle structure. For a **`float`** one can try out all $4\,194\,305$ possible starting values to see the cycle lengths that occur and their percent occurrence, as shown in the table. Notice that after an initial run of a few thousand values, the function falls into the cycle of length 1 (the cycle that maps 0 to itself) 93% of the time. The second part of Table 17.1 gives results of 7000 trials for **`double`**s, using random starting points and recording the percentages.

At this point, the logistic equation would seem useless for random number generation, since it has non-random behavior and often falls into a stationary point. However, I came up with two separate ideas that together make this approach useful.

Ordinary Logistic Equation		
Cycle Structure, `float`		
Cycle length	**Percent occurrence**	**Average initial run**
1	**93.0%**	**2034**
930	5.6%	340
431	1.0%	251
106	0.35%	244
205	0.1%	83
5	0.002%	31
4	0.0004%	7
3	0.00005%	2
1	0.00002%	0
Cycle Structure, `double`		
5 638 349	69.4%	54 000 000
1	**15.2%**	**10 000 000**
14 632 801	11.3%	8 500 000
10 210 156	1.5%	5 900 000
2 625 633	1.3%	3 800 000
2 441 806	1.2%	5 200 000
1 311 627	0.028%	240 000
960 057	0.014%	200 000

Remapped Logistic Equation		
Cycle Structure, `float`		
Cycle length	**Percent occurrence**	**Average initial run**
13753	89.9%	4745
3023	5.4%	1150
2928	3.4%	670
1552	0.66%	355
814	0.6%	266
9	0.035%	191
1	0.00017%	14
3	0.000024%	1.5
1	0.000012%	1
Cycle Structure, `double`		
112 467 844	80.5%	105 000 000
61 607 666	5.7%	23 000 000
35 599 847	4.3%	19 000 000
1 983 078	3.6%	39 000 000
4 148 061	3.3%	60 000 000
15 023 672	2.5%	19 000 000
12 431 135	0.084%	5 500 000
705 986	0.084%	670 000

Table 17.1 Cycles of the logistic equation (boldface = the cycle (0)).

17.4 The Re-mapped Logistic Equation.

The logistic equation yields numbers very close to 0 (on the positive side) and very close to 1. Available floating point numbers "pile up" near 0, but there is no similar behavior near 1. I was able to restructure the equation so that values occurring near 1 are re-mapped to the negative side of 0. The following definition does this, mapping the interval from -1 to 1 to itself (now called F_β to distinguish from the similar function f that maps the interval from 0 to 1 to itself):

$$F_\beta = \begin{cases} 2|x|(2-|x|), \text{ for } |x| \le \beta, \\ -2(1-|x|)^2, \text{ for } \beta < |x| \le 1, \end{cases}$$

where $\beta = 1 - (1/\sqrt{2} = 0.292893218813452476$.

In infinite precision, this re-mapped equation behaves exactly like the original, but with floating point numbers there is no longer any convergence to the cycle of length 1. Besides this cycle, the other cycles have lengths about 5 or 10 times as long. The previous table also gives results of experiments on the remapped equation, similar to those before, trying all 8 388 609 possible starting values for a **`float`** and over 1000 starting values for a **`double`**.

A slightly different function S will again transform these numbers to the uniform distribution from 0 to 1:

$$S(x) = \begin{cases} (2/\pi)\arcsin(\sqrt{x/2}), \text{ for } 0 \le x < 1, \\ (2/\pi)\arcsin(\sqrt{-x/2}) + 0.5, \text{ for } -1 < x < 0, \end{cases}$$

This equation does much better, but it is still not remotely good enough. The final piece of the puzzle is to combine multiple logistic equations into a lattice and couple them weakly together.

17.5 The Logistic Lattice.

Researchers in chaos theory often use a *lattice* of numbers to which the logistic equations are applied. Adjacent lattice nodes affect one another to a small degree. This model is a greatly simplified version of a fluid undergoing turbulence. In the 1-dimensional case, the nodes are an array of m numbers:

$$x_0, x_1, x_2, ..., x_{m-1}$$

The equation applied to each node i at the nth stage, producing the $n+1$st stage is:

$$x^i_{n+1} = f(x^i_n) + \nu[f(x^{i-1}_n) - 2f(x^i_n) + f(x^{i+1}_n)], \text{ for } 0 \le i < m,$$

where ν is a special *viscosity* constant, and the calculation of $i-1$ and $i+1$ is done modulo m, that is, by dividing by m and taking the remainder. In effect, this wraps the linear array of nodes into a circle. Combining terms gives:

$$x_{n+1}^{i} = (1-2\nu)f(x_n^i) + \nu[f(x_n^{i-1}) + f(x_n^{i+1})], \text{ for } 0 \le i < m,$$

In two dimensions, the equations take the form:

$$x_{n+1}^{i,j} = (1-4\nu)f(x_n^{i,j}) + \nu[f(x_n^{i,j-1}) + f(x_n^{i,j+1}) + f(x_n^{i-1,j}) + f(x_n^{i+1,j})],$$
$$\text{for } 0 \le i < m, 0 \le j < m,$$

Here again arithmetic with both subscripts is carried out modulo m. In both the 1- and 2-dimensional versions above, the numbers should be **`double`**s, and the constant ν should be small, so as not to disturb the uniform distribution and to promote more turbulence. I have used 10^{-13} for this constant. It is necessary to iterate the equations before outputting a value; I have used 120 iterations. The sizes should be at least size 7 in one dimension and size 3×3 in two dimensions.

If the initial values are symmetric about some axis, then the lattice will repeat as if there were just single logistic equations, so it would be best to use another random number generator to initialize the lattice values. The individual nodes are probably be independent of one another, so that this will produce m or m^2 random reals at a time, depending on whether it is 1- or 2-dimensional.

If one is worried about a symmetric set of values coming up (a highly improbable occurrence), one could use a variation of the equations that is not symmetric, such as:

$$x_{n+1}^{i,j} = (1-4\nu)f(x_n^{i,j}) + \nu[\,(1.1)\cdot f(x_n^{i,j-1}) + (0.9)\cdot f(x_n^{i,j+1}) +$$
$$(1.2)\cdot f(x_n^{i-1,j}) + (0.8)\cdot f(x_n^{i+1,j})],$$
$$\text{for } 0 \le i < m, 0 \le j < m,$$

As a way of visualizing the 2-dimensional lattice of size 3×3 (with 9 nodes), if the left and right sides are pasted together, the lattice would form a vertical hollow cylinder. Then if the top and bottom sides are pasted together, it would form a donut-shaped object (called a *torus* by mathematicians). (The picture is similar to the old "Pac Man" games, where the Pac Man would exit on one side and immediately come in from the other side.)

The pseudo-random number generator based on this lattice seems to be a very good source of random numbers, but from the nature of this theory, it is not possible to prove results about the probable length of cycles or about the quality of its random behavior. It seems likely that for almost all starting values (nine **`double`**s), the generator will not cycle for a very long time. It has been tested for random initial values and did not cycle for billions of iterations. The numbers produced gave good results when subjected to statistical tests. Nevertheless, the "perfect" generators of the previous section are to be preferred to this one.

17.6 Java Implementation of this Generator.

In the Java implementation it was necessary to iterate each step enough times so that each node would fill completely with noise and so that any possible symmetries would completely disappear. Java code to determine these parameters appears on page 266. The random number generator itself, for the 3×3 lattice, appears in the same section.

18
The Laws of Cryptography
Statistical Tests and Perfect Generators

18.1 Maurer's universal statistical test.

Maurer's universal statistical test can be found on page 270.

18.2 The Blum-Blum-Shub perfect generator.

The Blum-Blum-Shub perfect generator appears on page 272.

Part VI
The Advanced Encryption Standard

19
The Laws of Cryptography
The Advanced Encryption Standard

19.1 Overview.

The new U.S. *Advanced Encryption Standard* (AES) is a block cipher with block size of 128 bits, or 16 bytes. Keys for the cipher come in one of three lengths: 128, 192, or 256 bits, which is 16, 24, or 32 bytes. The algorithm is oriented toward bytes (8 bits), but there is also emphasis on what the AES specification calls *words*, which are arrays of 4 bytes (or 32 bits, the size of an **`int`** in Java). My Java implementation presented here tends to de-emphasize the words.

The main mathematical difficulty with the algorithm is that it uses arithmetic over the field $GF(2^8)$. Even the field itself only poses real difficulties with multiplication of field elements and with multiplicative inverses. These topics are covered in Section 2 below.

Otherwise, the AES algorithm is just an annoying number of details to orchestrate, but not really difficult. Section 3 below covers the *S-Boxes*, while Section 4 shows how the keys are handled. Section 5 covers the remainder of the encryption algorithm, and Section 6 covers decryption.

I have implemented the algorithm following the AES specification and using B. Gladman's commentary. I haven't worried about efficiency, but mostly have just tried to produce a clear, simple, working Java program. One exception is to give Gladman's more efficient implementation of multiplication in $GF(2^8)$ because it is interesting (see Section 2). Gladman has produced an optimized C implementation and has a lot to say on the subject of efficient implementation, especially with methods using tables.

Law AES-1:
Conventional block ciphers are always ugly, complicated, inelegant brutes, and the AES is no exception.

In order to create such a cryptosystem, one must remember that anything done by encryption must be undone during decryption, using the same key since it is a conventional (symmetric key) system. Thus the focus is on various invertible operations. One standard technique in using the key is to derive a string somehow from the key, and use **`xor`** to combine it with the emerging ciphertext. Later the same **`xor`** reverses this. Otherwise there are "mixing" operations that move data around, and "translation" (or "substitution") operations that replace one piece of data with another. This last operation is usually carried out on small portions of ciphertext using so-called "S-boxes", which define replacement strings. One set of mixing, replacements,

Key Sizes versus Rounds			
	Key Block Size (Nk words)	**Plaintext Block Size (Nb words)**	**Number of Rounds (Nr)**
AES-128	4	4	10
AES-192	6	4	12
AES-256	8	4	14

Table 19.1 Parameters for the AES.

and exclusive-or with a string derived from the key is called a *round*. Then there will typically be a number of rounds. The AES uses different numbers of rounds for the different key sizes according to Table 19.1 above. This table uses a variable **Nb** for the plaintext block size, but it is always **4** words. Originally the AES was going to support different block sizes, but they settled on just one size. However, the AES people (at the NIST) recommend keeping this as a named constant in case a change is ever wanted.

Remember that a *word* is **4** bytes or **32** bits. The names **Nk**, **Nb**, and **Nr** are standard for the AES. In general, I try to use the names in the AES specification, even when they do not conform to Java conventions.

The particular form of this type of algorithm, with its rounds of mixing and substitution and exclusive-or with the key, was introduced with the official release of the Data Encryption Standard (DES) in 1977 and with work preceding the release. The DES has a block size of **64** bits and a very small key size of **56** bits. From the beginning the key size of the DES was controversial, having been reduced at the last minute from **64** bits. This size seemed at the edge of what the National Security Agency (but no one else) could crack. Now it is easy to break, and completely insecure. The AES, with its minimum of **128** bits for a key should not be breakable by brute force attacks for a very long time, even with great advances in computer hardware.

19.2 Outline of the AES Algorithm.

Here is the AES algorithm is outline form, using Java syntax for the pseudo-code, and much of the AES standard notation:

```
Constants: int Nb = 4; // but it might change someday
           int Nr = 10, 12, or 14; // rounds, for Nk = 4, 6, or 8
Inputs: array  in  of 4*Nb bytes // input plaintext
        array out of  4*Nb bytes // output ciphertext
        array w of 4*Nb*(Nr+1) bytes // expanded key
Internal work array:
        state, 2-dim array of 4*Nb bytes, 4 rows and Nb cols
Algorithm:
```

```
void Cipher(byte[] in, byte[] out, byte[] w) {
   byte[][] state = new byte[4][Nb];
   state = in; // actual component-wise copy
   AddRoundKey(state, w, 0, Nb - 1); // see Section 4 below
   for (int round = 1; round < Nr; round++)
      SubBytes(state); // see Section 3 below
      ShiftRows(state); // see Section 5 below
      MixColumns(state); // see Section 5 below
      AddRoundKey(state, w,
          round*Nb, (round+1)*Nb - 1); // Section 4
   }
   SubBytes(state); // see Section 3 below
   ShiftRows(state); // see Section 5 below
   AddRoundKey(state, w, Nr*Nb, (Nr+1)*Nb - 1); // Section 4
   out = state; // component-wise copy
}
```

Let's go down the items in the above pseudo-code in order:

- **`Multiplication in GF(256)`**: this is not mentioned explicitly above, but the individual functions use it frequently. It is described in Section 2 below.
- **`Nb`**: Right now, this is always **`4`**, the constant number of **`32`**-bit **`words`** that make up a block for encryption and decryption.
- **`Nr`**: the number of *rounds* or main iterations of the algorithm. The possible values depend on the three different possible key sizes the earlier table showed.
- **`in`**: the input block of **`128`** bits of plaintext, arranged as **`4*Nb = 16`** bytes, numbered **`0`** to **`15`** and arranged in sequence.
- **`out`**: the output block of **`128`** bits of ciphertext, arranged the same as **`in`**.
- **`state`**: the internal array that is worked on by the AES algorithm. It is arranged as a **`4`** by **`Nb`** 2-dimensional array of bytes (that is, **`4`** by **`4`**).
- **`w`**: the expanded key. The initial key is of size **`4*Nk`** bytes (see table earlier), and this is expanded to the array **`w`** of **`4*Nb*(Nr+1) = 16*(Nr+1)`** bytes for input to the encryption algorithm. Each round uses **`4*Nb`** bytes, and each portion of **`w`** is used only once. (There are **`Nr+1`** full rounds, and an extra use before and after in partial rounds, for a total of **`Nr+1`** uses.) This function for expanding the key is described in Section 4 below.
- **`SubBytes(state)`**: this takes each byte of the **`state`** and independently looks it up in an "S-box" table to substitute a different byte for it. (The same S-box table is also used in the key expansion.) Section 3 shows how the S-box table is defined and constructed.
- **`ShiftRows(state)`**: this simply moves around the rows of the **`state`** array. See Section 5 below.

- **`MixColumns(state)`**: this does a much more complicated mix of the columns of the **state** array. See Section 5 below.
- **`AddRoundKey(state, w, param1, param2)`**: this takes the **`4*Nb*(Nr+1)`** bytes of the expanded key, **`w`**, and does an exclusive-or of successive portions of the expanded key with the changing **`state`** array. The integer values **`param1`** and **`param2`** take on different values during execution of the algorithm, and they give the inclusive range of columns of the expanded key that are used. My implementation of this function doesn't use these parameters, because each round just uses the next **`4*Nb`** bytes of **`w`**. The details of this function appear in Section 4 below.

Well, that's pretty much it. Now the remaining sections just have to fill in a large number of missing details. Section 6 gives the inverse algorithm for decryption, but this is not a big problem, since the parameters and functions are either identical or similar.

20
The Laws of Cryptography
The Finite Field GF(256)

20.1 Finite Fields.

A *field* is an algebraic object with two operations: addition and multiplication, represented by $+$ and $*$, although they will not necessarily be ordinary addition and multiplication. Using $+$, all the elements of the field must form a commutative group, with identity denoted by 0 and the inverse of a denoted by $-a$. Using $*$, all the elements of the field except 0 must form another commutative group with identity denoted 1 and inverse of a denoted by a^{-1}. (The element 0 has no inverse under $*$.) Finally, the *distributive identity* must hold: $a*(b+c) = (a*b)+(a*c)$, for all field elements a, b, and c.

There are a number of different infinite fields, including the rational numbers (fractions), the real numbers (all decimal expansions), and the complex numbers. Cryptography focuses on *finite* fields. It turns out that for any prime integer p and any integer n greater than or equal to 1, there is a unique field with p^n elements in it, denoted $GF(p^n)$. (The "GF" stands for "Galois Field", named after the brilliant young French mathematician who discovered them.) Here "unique" means that any two fields with the same number of elements must be essentially the same, except perhaps for giving the elements of the field different names.

In case n is equal to 1, the field is just the *integers mod p*, in which addition and multiplication are just the ordinary versions followed by taking the remainder on division by p. The only difficult part of this field is finding the multiplicative inverse of an element, that is, given a non-zero element a in Z_p, finding a^{-1}. This is the same as finding a b such that $a * b\%p = 1$. This calculation can be done with the extended Euclidean algorithm, as is explained elsewhere in these notes.

20.2 The Finite Field *GF*(2^n).

The case in which n is greater than one is much more difficult to describe. In cryptography, one almost always takes p to be 2 in this case. This section just treats the special case of $p = 2$ and $n = 8$, that is, $GF(2^8)$, because this is the field used by the new U.S. *Advanced Encryption Standard* (AES).

The AES works primarily with *bytes* (8 bits), represented from the right as:

$$b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0 .$$

The 8-bit elements of the field are regarded as *polynomials* with coefficients in the field Z_2:

$$b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x^1 + b_0.$$

The field elements will be denoted by their sequence of bits, using two hex digits.

20.3 Addition in $GF(2^n)$.

To add two field elements, just add the corresponding polynomial coefficients using addition in Z_2. Here addition is modulo 2, so that $1 + 1 = 0$, and addition, subtraction and exclusive-or are all the same. The identity element is just zero: 00000000 (in bits) or $0x00$ (hex).

20.4 Multiplication in $GF(2^n)$.

Multiplication is this field is more complicated and harder to understand, but it can be implemented very efficiently in hardware and software. The first step in multiplying two field elements is to multiply their corresponding polynomials just as in beginning algebra (except that the coefficients are only 0 or 1, and $1 + 1 = 0$ makes the calculation easier, since many terms just drop out). The result would be an up to degree 14 polynomial — too big to fit into one byte. A finite field now makes use of a fixed degree eight irreducible polynomial (a polynomial that cannot be factored into the product of two simpler polynomials). For the AES the polynomial used is the following (other polynomials could have been used):

$$\begin{aligned} m(x) &= x^8 + x^4 + x^3 + x + 1 \\ &= 0x11b \text{ (hex).} \end{aligned}$$

The intermediate product of the two polynomials must be divided by $m(x)$. The remainder from this division is the desired product.

This sounds hard, but is easier to do by hand than it might seem (though error-prone). To make it easier to write the polynomials down, adopt the convention that instead of $x^8 + x^4 + x^3 + x + 1$ just write the exponents of each non-zero term. (Remember that terms are either zero or have a 1 as coefficient.) So write the following for $m(x)$: (84310).

Now try to take the product $(75421) * (6410)$ (which is the same as $0xb6 * 0x53$ in hexadecimal). First do the multiplication, remembering that in the sum below only an odd number of like powered terms results in a final term:

```
(7 5 4 2 1) * (6 4 1 0)      gives (one term at a time)

(7 5 4 2 1) * (6)    =   (13    11 10   8 7)
(7 5 4 2 1) * (4)    =         (11    9 8   6 5)
(7 5 4 2 1) * (1)    =                (8   6 5   3 2)
(7 5 4 2 1) * (0)    =    +              7   5 4   2 1)
                         ---------------------------------
                         (13       10 9 8     5 4 3   1)
```

The final answer requires the remainder on division by $m(x)$, or (8431). This is like ordinary polynomial division, though easier because of the simpler arithmetic.

```
                          (13       10 9 8     5 4 3   1)
(8 4 3 1 0) * (5)    =    (13          9 8   6 5)
                          -------------------------------
                                  (10       6   4 3   1)
(8 4 3 1 0) * (2) =               (10       6 5   3 2)
                          -------------------------------
                                              (5 4   2 1)
                                          (the remainder)
```

Here the first element of the quotient is x^5 and the second element of the quotient is x^2. Thus the final result says that $0xb6 * 0x53 = 0x36$ in the field. (When I did the calculations above, I made two separate mistakes, but checked my work with techniques below.)

20.5 Improved Multiplication in *GF*(2^n).

The above calculations could be converted to a program, but there is a better way. One does the calculations working from the low order terms, and repeatedly multiplying by (1). If the result is of degree 8, just add (the same as subtract) $m(x)$ to get degree 7. Again this can be illustrated using the above notation and the same example operands: $r * s = (75421) * (6410)$

```
i    powers of  r: r * (i)     Simplified Result          Final Sum
=============================================================================
0                               (7   5 4   2 1  )      (7   5 4   2 1  )
-----------------------------------------------------------------------------
1   (7   5 4   2 1  )*(1)  =  (8   6 5   3 2)
                             +(8       4 3   1 0)
                             ---------------------
                                  (6 5 4   2 1 0)     +  (6 5 4   2 1 0)
                                                          ------------------
                                                           (7 6           0)
-----------------------------------------------------------------------------
2     (6 5 4   2 1 0)*(1)  =    (7 6 5   3 2 1)
-----------------------------------------------------------------------------
3   (7 6 5   3 2 1)  *(1)  =  (8 7 6   4 3 2)
                             +(8       4 3   1 0)
                             ---------------------
                                (7 6       2 1 0)
-----------------------------------------------------------------------------
4   (7 6       2 1 0)*(1)  =  (8 7       3 2 1)
                             +(8       4 3   1 0)
                             ---------------------     (7 6           0)
                                (7     4   2   0)    + (7     4   2   0)
                                                      ------------------
                                                         (6   4   2)
-----------------------------------------------------------------------------
5   (7     4   2   0)*(1)  =  (8     5   3   1)
                             +(8       4 3   1 0)
                             ---------------------
```

```
                                        (5 4       0)
                                                        (6   4   2)
6       (5 4       0)*(1)  =      (6 5       1)      +   (6 5       1)
                                                        -----------------
                                                              (5 4   2 1)
```

The final answer is the same as before. Here is an algorithm (almost) in Java that realizes the above calculations:

```
public byte FFMul(unsigned byte a, unsigned byte b) {
   unsigned byte aa = a, bb = b, r = 0, t;
   while (aa != 0) {
      if ((aa & 1) != 0)
         r = r ^ bb;
      t = bb & 0x80;
      bb = bb << 1;
      if (t != 0)
         bb = bb ^ 0x1b;
      aa = aa >> 1;
   }
   return r;
}
```

Unfortunately, Java has no **`unsigned byte`** type, and the logical operations produce a 32-bit integer. Finally, one ought to be able to use Java's "right shift with zero fill" operator **`>>>`**, but it doesn't work as it is supposed to. See Appendix B for a discussion of the problems encountered in converting the above "Java" program to actual Java. Later examples below show the code for this function.

20.6 Using Logarithms to Multiply.

When I was young (a *long* time ago) there were no pocket calculators. We had to do without modern conveniences like Gatorade. Mathematical calculations, if not done with a slide rule, were carried out by hand. Often we used printed tables of logarithms to turn multiplications into easier additions. (In these "elder" days, believe it or not, the printed tables included tables of the logarithms of trig functions of angles, so that you got the log directly for further calculations.)

As a simple example, suppose one wanted the area of a circle of radius 23.427 cm. Use the famous formula πr^2 ("pie are square, cake are round"), so one needs to calculate

$$A = 23.427 * 23.427 * 3.1416.$$

We would look up the logarithm (base 10) of each number in the printed table: $\log(23.427) = 1.369716$ and $\log(3.1416) = .497156$. Now add two copies of the first number and one of the second:

$$1.369716 + 1.369716 + .497156 = 3.236588.$$

Finally, take the "anti-log" (that is, take 10 to the power 3.236588) to get the final answer: 1724.2. This works because

$$\log(A) = \log(\pi r^2) = \log(pi) + \log(r) + \log(r).$$

The actual use of log tables was much more horrible than the above might indicate. In case you want to find out how it *really* worked, look at Appendix A, but have an air sickness bag handy.

In a similar way, in finite fields one can replace the harder multiplication by the easier addition, at the cost of looking up "logarithms" and "anti-logarithms."

20.7 Generators in Fields.

First must come the concept of a *generator* of a finite field. Generators also play a role is certain simple but common random number generators, as is detailed in another section. A generator is an element whose successive powers take on every element except the zero. For example, in the field Z_{13}, try successive powers of several elements, looking for a generator:
Try powers of 5, taken modulo 13:

$$\begin{aligned}
&5^1 = 5,\\
&5^2\%13 = 25\%13 = 12,\\
&5^3\%13 = (12 * 5)\%13 = 8,\\
&5^4\%13 = (8 * 5)\%13 = 1,
\end{aligned}$$

so successive powers of 5 just take on the values 5, 12, 8, 1, and repeat, so that 5 is not a generator.
Now try powers of 4, taken modulo 13:

$$\begin{aligned}
&4^1 = 4,\\
&4^2\%13 = 16\%13 = 3,\\
&4^3\%13 = (3 * 4)\%13 = 12,\\
&4^4\%13 = (12 * 4)\%13 = 9,\\
&4^5\%13 = (9 * 4)\%13 = 10,\\
&4^6\%13 = (10 * 4)\%13 = 1,
\end{aligned}$$

so successive powers make a longer cycle, but still not all elements: 4, 3, 12, 9, 10, 1, and repeat, so 4 is also not a generator.
Finally try successive powers of 2, taken modulo 13:

$$\begin{aligned}
&2^1 = 2,\\
&2^2\%13 = 4\%13 = 4,\\
&2^3\%13 = 8\%13 = 8,\\
&2^4\%13 = (8 * 2)\%13 = 3,
\end{aligned}$$

Table of "Exponential" Values																	
E(rs)		s															
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**a**	**b**	**c**	**d**	**e**	**f**
r	**0**	01	03	05	0f	11	33	55	ff	1a	2e	72	96	a1	f8	13	35
	1	5f	e1	38	48	d8	73	95	a4	f7	02	06	0a	1e	22	66	aa
	2	e5	34	5c	e4	37	59	eb	26	6a	be	d9	70	90	ab	e6	31
	3	53	f5	04	0c	14	3c	44	cc	4f	d1	68	b8	d3	6e	b2	cd
	4	4c	d4	67	a9	e0	3b	4d	d7	62	a6	f1	08	18	28	78	88
	5	83	9e	b9	d0	6b	bd	dc	7f	81	98	b3	ce	49	db	76	9a
	6	b5	c4	57	f9	10	30	50	f0	0b	1d	27	69	bb	d6	61	a3
	7	fe	19	2b	7d	87	92	ad	ec	2f	71	93	ae	e9	20	60	a0
	8	fb	16	3a	4e	d2	6d	b7	c2	5d	e7	32	56	fa	15	3f	41
	9	c3	5e	e2	3d	47	c9	40	c0	5b	ed	2c	74	9c	bf	da	75
	a	9f	ba	d5	64	ac	ef	2a	7e	82	9d	bc	df	7a	8e	89	80
	b	9b	b6	c1	58	e8	23	65	af	ea	25	6f	b1	c8	43	c5	54
	c	fc	1f	21	63	a5	f4	07	09	1b	2d	77	99	b0	cb	46	ca
	d	45	cf	4a	de	79	8b	86	91	a8	e3	3e	42	c6	51	f3	0e
	e	12	36	5a	ee	29	7b	8d	8c	8f	8a	85	94	a7	f2	0d	17
	f	39	4b	dd	7c	84	97	a2	fd	1c	24	6c	b4	c7	52	f6	01

Table 20.1 Table of Powers of 0x03, the "exponentials".

$$
\begin{aligned}
2^5 \% 13 &= (3 * 2) \% 13 = 6, \\
2^6 \% 13 &= (6 * 2) \% 13 = 12, \\
2^7 \% 13 &= (12 * 2) \% 13 = 11, \\
2^8 \% 13 &= (11 * 2) \% 13 = 9, \\
2^9 \% 13 &= (9 * 2) \% 13 = 5, \\
2^9 \% 13 &= (5 * 2) \% 13 = 10, \\
2^9 \% 13 &= (10 * 2) \% 13 = 7, \\
3^6 \% 13 &= (7 * 2) \% 13 = 1,
\end{aligned}
$$

so successive powers take on all non-zero elements: 2, 4, 8, 3, 6, 12, 11, 9, 5, 10, 7, 1, and repeat, so finally 2 is a generator. (Wheh! In fact, the generators are 2, 6, 7, and 11.)

However 2 is *not* a generator for the field Z_{23}, so it doesn't work to just try 2. (The generators in Z_{23} are 5, 7, 10, 11, 14, 15, 17, 19, 20, and 21.)

The above random search shows that generators are hard to discover and are not intuitive. It turns out that $0x03$, which is the same as $x + 1$ as a polynomial, is the simplest generator for $GF(2^8)$. Its powers take on all 255 non-zero values of the field. In fact Table 20.1 , a table of "exponentials" or "anti-logs", gives each possible power. (The table is really just a simple linear table, not really 2-dimensional, but it has been arranged so that the two hex digits are on different axes.) Here $E(rs)$ is the field element given by 03^{rs}, where these are hex numbers, and the initial "$0x$" is left off for simplicity.

Similarly, Table 20.2 is a table of "logarithms", where the entry $L(rs)$ is the field element

Table of "Logarithm" Values																	
L(rs)		s															
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**a**	**b**	**c**	**d**	**e**	**f**
r	**0**	–	00	19	01	32	02	1a	c6	4b	c7	1b	68	33	ee	df	03
	1	64	04	e0	0e	34	8d	81	ef	4c	71	08	c8	f8	69	1c	c1
	2	7d	c2	1d	b5	f9	b9	27	6a	4d	e4	a6	72	9a	c9	09	78
	3	65	2f	8a	05	21	0f	e1	24	12	f0	82	45	35	93	da	8e
	4	96	8f	db	bd	36	d0	ce	94	13	5c	d2	f1	40	46	83	38
	5	66	dd	fd	30	bf	06	8b	62	b3	25	e2	98	22	88	91	10
	6	7e	6e	48	c3	a3	b6	1e	42	3a	6b	28	54	fa	85	3d	ba
	7	2b	79	0a	15	9b	9f	5e	ca	4e	d4	ac	e5	f3	73	a7	57
	8	af	58	a8	50	f4	ea	d6	74	4f	ae	e9	d5	e7	e6	ad	e8
	9	2c	d7	75	7a	eb	16	0b	f5	59	cb	5f	b0	9c	a9	51	a0
	a	7f	0c	f6	6f	17	c4	49	ec	d8	43	1f	2d	a4	76	7b	b7
	b	cc	bb	3e	5a	fb	60	b1	86	3b	52	a1	6c	aa	55	29	9d
	c	97	b2	87	90	61	be	dc	fc	bc	95	cf	cd	37	3f	5b	d1
	d	53	39	84	3c	41	a2	6d	47	14	2a	9e	5d	56	f2	d3	ab
	e	44	11	92	d9	23	20	2e	89	b4	7c	b8	26	77	99	e3	a5
	f	67	4a	ed	de	c5	31	fe	18	0d	63	8c	80	c0	f7	70	07

Table 20.2 Table of Inverse Powers of 0x03, the "logarithms".

that satisfies $rs = 03^{L(rs)}$, where these are hex numbers, and again the initial "$0x$" is left off.

These tables were created using the multiply function in the previous section. A Java program that directly outputs the HTML source to make the tables appears on page 273.

As an example, suppose one wants the product $b6 * 53$ (the same product as in the examples above, leaving off the "$0x$"). Use the L table above to look up $b6$ and 53: $L(b6) = b1$ and $L(53) = 30$. This means that

$$\begin{aligned}(b6) * (53) &= (03)^{(b1)} * (03)^{(30)} \\ &= (03)^{(b1+30)} = (03)^{(e1)}.\end{aligned}$$

If the sum above gets bigger than ff, just subtract 255 as shown. This works because the powers of 03 repeat after 255 iterations. Now use the E table to look up $(03)^{(e1)}$, which is the answer: (36).

Thus these tables give a much simpler and faster algorithm for multiplication:

```
public byte FFMulFast(unsigned byte a, unsigned byte b){
   int t = 0;;
   if (a == 0 || b == 0) return 0;
   t = L[a] + L[b];
   if (t > 255) t = t - 255;
   return E[t];
}
```

Multiplicative Inverses of Each Element																	
inv(rs)		s															
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**a**	**b**	**c**	**d**	**e**	**f**
r	**0**	–	01	8d	f6	cb	52	7b	d1	e8	4f	29	c0	b0	e1	e5	c7
	1	74	b4	aa	4b	99	2b	60	5f	58	3f	fd	cc	ff	40	ee	b2
	2	3a	6e	5a	f1	55	4d	a8	c9	c1	0a	98	15	30	44	a2	c2
	3	2c	45	92	6c	f3	39	66	42	f2	35	20	6f	77	bb	59	19
	4	1d	fe	37	67	2d	31	f5	69	a7	64	ab	13	54	25	e9	09
	5	ed	5c	05	ca	4c	24	87	bf	18	3e	22	f0	51	ec	61	17
	6	16	5e	af	d3	49	a6	36	43	f4	47	91	df	33	93	21	3b
	7	79	b7	97	85	10	b5	ba	3c	b6	70	d0	06	a1	fa	81	82
	8	83	7e	7f	80	96	73	be	56	9b	9e	95	d9	f7	02	b9	a4
	9	de	6a	32	6d	d8	8a	84	72	2a	14	9f	88	f9	dc	89	9a
	a	fb	7c	2e	c3	8f	b8	65	48	26	c8	12	4a	ce	e7	d2	62
	b	0c	e0	1f	ef	11	75	78	71	a5	8e	76	3d	bd	bc	86	57
	c	0b	28	2f	a3	da	d4	e4	0f	a9	27	53	04	1b	fc	ac	e6
	d	7a	07	ae	63	c5	db	e2	ea	94	8b	c4	d5	9d	f8	90	6b
	e	b1	0d	d6	eb	c6	0e	cf	ad	08	4e	d7	e3	5d	50	1e	b3
	f	5b	23	38	34	68	46	03	8c	dd	9c	7d	a0	cd	1a	41	1c

Table 20.3 Table of Inverses of Each Field Element.

As before, this is Java as if it had an **`unsigned byte`** type, which it doesn't. The actual Java code requires some short, ugly additions. (See Unsigned bytes in Java in Appendix B to convert the above "Java" program to actual Java.)

This section has presented two algorithms for multiplying field elements, a slow one and a fast one. As a check, here is a program that compares the results of all 65536 possible products to see that the two methods agree (which they do): see Compare multiplications on page 275.

20.8 The Multiplicative Inverse of Each Field Element.

Later work with the AES will also require the multiplicative inverse of each field element except 0, which has no inverse. This inverse is easy to calculate, given the tables above. If g is the generator 03 (leaving off the "$0x$"), then the inverse of g^{rs} is g^{ff-rs}, so that for example, to find the inverse of $6b$, look up in the "log" table to see that $6b = g^{54}$, so the inverse of $6b$ is $g^{ff-54} = g^{ab}$, and from the "exponential" table, this is df. Table 20.3 gives the result of carrying out the above procedure for each non-zero field element.

The next chapter has a table generating program that will calculate and print HTML source for the above table.

21
The Laws of Cryptography
Advanced Encryption Standard: S-Boxes

21.1 S-Boxes and the SubBytes Transformation.

Many different block ciphers use a special substitution called an "S-box". The AES also has these S-boxes, which it terms the "SubBytes Transformation". S-boxes provide an invertible (reversible) transformation of segments of plaintext during encryption, with the reverse during decryption. With the AES it is a single simple function applied over and over again to each byte during stages of the encryption, returning a byte. Each of the 256 possible byte values is transformed to another byte value with the **`SubBytes`** transformation, which is a *full permutation*, meaning that every element gets changed, and all 256 possible elements are represented as the result of a change, so that no two different bytes are changed to the same byte. The **`SubBytes`** transformation changes a single entry as follows (here b_i stands for the ith bit of a byte value b).

```
byte SubBytesEntry(byte b) {
   byte c = 0x63;
   if (b != 0)}  // leave 0 unchanged {\timesbf
      b = multiplicativeInverse(b);
   for (i = 0; i < 8; i++)
      b[i] = b[i] ^ b[(i+4)%8] ^ b[(i+5)%8] ^
                     b[(i+6)%8] ^ b[(i+7)%8] ^ c[i];
   return b;
}
```

In practice, the 256 values given by the transformation of each byte should be precomputed and stored in a table. Because the table is computed only once before the start of encryption, there is less need to optimize its performance. Here is a copy of the table. This and the next table were printed using the Java program on page 277.)

21.2 The Inverse SubBytes Transformation.

The table of the inverse SubBytes transformation could be generated using the inverse of the formula for SubBytes given above (a similar function). However, it is even easier to just directly invert the previous table, without recalculating.

21.3 The Function SubBytes(state).

S-Box Values																	
S(rs)		s															
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
r	0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
	1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
	2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
	3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
	4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
	5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
	6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
	7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
	8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
	9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
	a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
	b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
	c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
	d	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
	e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
	f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Table 21.1 S-Box Values.

Inverse S-Box Values																	
iS(rs)		s															
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
r	0	52	09	6a	d5	30	36	a5	38	bf	40	a3	9e	81	f3	d7	fb
	1	7c	e3	39	82	9b	2f	ff	87	34	8e	43	44	c4	de	e9	cb
	2	54	7b	94	32	a6	c2	23	3d	ee	4c	95	0b	42	fa	c3	4e
	3	08	2e	a1	66	28	d9	24	b2	76	5b	a2	49	6d	8b	d1	25
	4	72	f8	f6	64	86	68	98	16	d4	a4	5c	cc	5d	65	b6	92
	5	6c	70	48	50	fd	ed	b9	da	5e	15	46	57	a7	8d	9d	84
	6	90	d8	ab	00	8c	bc	d3	0a	f7	e4	58	05	b8	b3	45	06
	7	d0	2c	1e	8f	ca	3f	0f	02	c1	af	bd	03	01	13	8a	6b
	8	3a	91	11	41	4f	67	dc	ea	97	f2	cf	ce	f0	b4	e6	73
	9	96	ac	74	22	e7	ad	35	85	e2	f9	37	e8	1c	75	df	6e
	a	47	f1	1a	71	1d	29	c5	89	6f	b7	62	0e	aa	18	be	1b
	b	fc	56	3e	4b	c6	d2	79	20	9a	db	c0	fe	78	cd	5a	f4
	c	1f	dd	a8	33	88	07	c7	31	b1	12	10	59	27	80	ec	5f
	d	60	51	7f	a9	19	b5	4a	0d	2d	e5	7a	9f	93	c9	9c	ef
	e	a0	e0	3b	4d	ae	2a	f5	b0	c8	eb	bb	3c	83	53	99	61
	f	17	2b	04	7e	ba	77	d6	26	e1	69	14	63	55	21	0c	7d

Table 21.2 Inverse S-Box Values.

The Java pseudo-code for this part is now very simple, using the **Sbox** array defined above:

```
void SubBytes(byte[][] state) {
   for (int row = 0; row < 4; row++)
      for (int col = 0; col < Nb; col++)
         state[row][col] = Sbox[state[row][col]];
}
```

22
The Laws of Cryptography
AES Key Expansion

22.1 Overview of Key Expansion.

In a simple cipher, one might exclusive-or the key with the plaintext. Such a step is easily reversed by another exclusive-or of the same key with the ciphertext. In the case of the AES, there are a number of rounds, each needing its own key, so the actual key is "stretched out" and transformed to give portions of key for each round. This is the key expansion that is the topic of this section.

The key expansion routine, as part of the overall AES algorithm, takes an *input key* (denoted **`key`** below) of **`4*Nk`** bytes, or **`Nk`** 32-bit words. **`Nk`** has value either **`4`**, **`6`**, or **`8`**. The output is an *expanded key* (denoted **`w`** below) of **`4*Nb*(Nr+1)`** bytes, where **`Nb`** is always **`4`** and **`Nr`** is the number of *rounds* in the algorithm, with **`Nr`** equal **`10`** in case **`Nk`** is **`4`**, **`Nr`** equal **`12`** in case **`Nk`** is **`6`**, and **`Nr`** equal **`14`** in case **`Nk`** is **`8`**.

The key expansion routine below states most of the actions in terms of *words* or 4-byte units, since the AES specification itself emphasizes words, but my implementation uses bytes exclusively.

```
Constants: int Nb = 4; // but it might change someday
Inputs:  int Nk = 4, 6, or 8; // the number of words in the key
         array key  of 4*Nk bytes or Nk words // input key
Output: array w of Nb*(Nr+1) words or 4*Nb*(Nr+1) bytes // expanded key
Algorithm:

void KeyExpansion(byte[] key, word[] w, int Nw) {
   int Nr = Nk + 6;
   w = new byte[4*Nb*(Nr+1)];
   int temp;
   int i = 0;
   while ( i < Nk) {
      w[i] = word(key[4*i], key[4*i+1], key[4*i+2], key[4*i+3]);
      i++;
   }
   i = Nk;
   while(i < Nb*(Nr+1)) {
      temp = w[i-1];
      if (i % Nk == 0)
         temp = SubWord(RotWord(temp)) ^ Rcon[i/Nk];
      else if (Nk > 6 && (i%Nk) == 4)
         temp = SubWord(temp);
      w[i] = w[i-Nk] ^ temp;
```

Expanded Key Sizes in Words		
Key Length ***(Nk words)***	**Number of Rounds** *(Nr)*	**Exp. Key Size** ***(Nb(Nr+1) words)***
4	10	44
6	12	52
8	14	60

Table 22.1 Expanded Key Sizes.

Powers of x = 0x02															
i	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**
x^i	01	02	04	08	10	20	40	80	1b	36	6c	d8	ab	4d	9a

Table 22.2 Parameters for Linear Congruence Generators.

```
    i++;
}
```

Discussion of items in the above pseudo-code in order:

- **The constant `Nb = 4`**: This was mentioned earlier. **`Nb`** is the number of words in an AES block, and right now it is always **`4`**.
- **The key, `key`**: the input key consists of **`Nk`** words, or **`4*Nk`** bytes.
- **The expanded key, `w`**: This consists of **`Nb*(Nk+1)`** words, or **`4*Nb*(Nk+1)`** bytes. The range of sizes are in Table 22.1 .
- **`RotWord()`**: This does the following simple cyclic permutation of a word: change $[a_0, a_1, a_2, a_3]$ to $[a_1, a_2, a_3, a_0]$.
- **`Rcon[i]`**: This is defined as the word: $[x^{i-1}, 0, 0, 0]$. Table 22.2 contains values of powers of x:

  Notice that in the algorithm for key expansion, the first reference to **`Rcon`** is **`Rcon[i/Nk]`**, where **`i`** has value **`Nk`**, so that the smallest index to **`Rcon`** is **`0`**, and this uses x^0.
- **`SubWord()`**: This just applies the S-box value used in **`SubBytes`** to each of the **`4`** bytes in the argument.

22.2 Use of Key Expansion in the AES Algorithm.

The function **`KeyExpansion()`** merely supplies a much expanded (and transformed) key

for use by the **`AddRoundKey()`** function in the main AES algorithm (see Section 1). This does a byte-wise exclusive-or of **`4*Nb = 16`** bytes at a time of the key with the **`4*Nb = 16`** bytes of the state. Successive segments of **`4*Nb = 16`** bytes of the expanded key are exclusive-ored in before the rounds of the algorithm, during each round, and at the end of the rounds. In the end, there are **`Nr`** rounds, but **`Nr+1`** exclusive-ors of parts of the expanded key. Since none of the expanded key is used more than once, this means that algorithm needs **`4*Nb*(Nr+1) = 16*(Nr+1)`** bytes of expanded key, and this is just the amount provided by the **`KeyExpansion()`** function.

23
The Laws of Cryptography
AES Encryption

23.1 Final Parts of the AES Algorithm.

The previous 4 subsections have covered most of the details needed to complete the AES algorithm. One still needs a description and code for the following routines:

- **`ShiftRows()`**
- **`MixColumns()`**
- **`AddRoundKey()`**

One also needs to organize a number of minor details to get a complete working Java program.

In the first two parts, the AES is moving around and "stirring up" data in the 4-by-4 array of bytes named **`state`**.

23.2 Implementation of ShiftRows().

The action of shifting rows is particularly simple, just performing left circular shifts of rows 1, 2, and 3, by amounts of 1, 2, and 3 bytes. Row 0 is not changed. The actual Java code below does this.

```
void ShiftRows(byte[][] state) {
   byte[] t = new byte[4];
   for (int r = 1; r < 4; r++) {
      for (int c = 0; c < Nb; c++)
         t[c] = state[r][(c + r)%Nb];
      for (int c = 0; c < Nb; c++)
         state[r][c] = t[c];
   }
}
```

23.3 Implementation of MixColumns().

The action of mixing columns works on the columns of the **`state`** array, but it is much more complicated that the shift columns action. As described in the AES specification, it treats each

column as a four-term polynomial with coefficients in the field $GF(2^8)$. All this is similar to the description of the field itself, except with an extra layer of complexity. These polynomials are added and multiplied just using the operations of the field $GF(2^8)$ on the coefficients, except that the result of a multiplication, which is a polynomial of degree up to **6**, must be reduced by dividing by the polynomial $x^4 + 1$ and taking the remainder.

The columns are each multiplied by the fixed polynomial:

$$a(x) = (03)x^3 + (01)x^2 + (01)x + (02),$$

where (03) stands for the field element $0x03$. This sounds horrible, but mathematical manipulations can reduce everything to the following simple algorithm, where multiplication in the field is represented below by $\#$. The principle change needed to convert this to actual Java is to replace $\#$ with a call to **`FFMul()`**. (Gladman gives a shorter but more obscure version of this code.)

```
void MixColumns(byte[][] s) {
   byte[] sp = new byte[4];
   for (int c = 0; c < 4; c++) {
      sp[0] = (0x02 # s[0][c]) ^ (0x03 # s[1][c]) ^
                      s[2][c]  ^          s[3][c];
      sp[1] =         s[0][c]  ^ (0x02 # s[1][c]) ^
              (0x03 # s[2][c]) ^          s[3][c];
      sp[2] =         s[0][c]  ^          s[1][c]  ^
              (0x02 # s[2][c]) ^ (0x03 # s[3][c]);
      sp[3] = (0x03 # s[0][c]) ^          s[1][c]  ^
                       s[2][c] ^ (0x02 # s[3][c]);
      for (int i = 0; i < 4; i++) s[i][c] = sp[i];
   }
}
```

23.4 Implementation of AddRoundKey().

As described before, portions of the expanded key **`w`** are exclusive-ored onto the state matrix **`Nr+1`** times (once for each round plus one more time). There are **`4*Nb`** bytes of state, and since each byte of the expanded key is used exactly once, the expanded key size of **`4*Nb*(Nr+1)`** bytes is just right. The expanded key is used, byte by byte, from lowest to highest index, so there is no need to count the bytes as they are used from **`w`**, but just use them up and move on, as the following near-Java code shows. This code assumes the key has already been expanded into the array **`w`**, and it assumes a global counter **`wCount`** initialized to **`0`**. The function **`AddRoundKey`** uses up **`4*Nb = 16`** bytes of expanded key every time it is called.

```
void AddRoundKey(byte[][] state) {
   for (int c = 0; c < Nb; c++)
      for (int r = 0; r < 4; r++)
         state[r][c] = state[r][c] ^ w[wCount++];
}
```

23.5 Java Implementation of Encryption Using AES.

My Java implementation of AES encryption uses six classes:

- **`AESencrypt`**, which provides the principle functions for AES encryption,
- **`Tables`**, which gives values from computed tables and various untility functions.
- **`GetBytes`**, which reads plaintexts and keys,
- **`Copy`**, which gives two simple copy functions needed by AES,
- **`Print`**, which prints 1- and 2-dimensional arrays of bytes for debugging, and
- **`AEStest`**, which which is a driver for the tests.

A combined listing of all the ecryption classes is found on page 282.

See the next chapter for test runs of the program, where encryption is followed by decryption.

24
The Laws of Cryptography
AES Decryption

24.1 Modifications to AES for Decryption.

The following functions need minor (or more major) revision for decryption:

- **`Cipher()`**, changed to **`InvCipher()`**, which is the main decryption outline. It is of course very similar to the **`Cipher()`** function, except that many of the subfunctions are themselves inverses, and the order of functions within a round is different.
- **`ShiftRows()`**, changed to **`InvShiftRows()`** – just minor changes.
- **`MixColumns()`**, changed to **`InvMixColumns()`** – the inverse function, similar but with different constants in it.
- **`AddRoundKey()`**, changed to **`InvAddRoundKey()`** – just works backwards along the expanded key.

As before, one also needs to organize a number of minor details to get a complete working Java program.

24.2 Implementation of InvCipher().

Here is Java pseudo-code for the inverse cipher. The various steps must be carried out in reverse order. These are arranged into rounds as with encryption, but the functions in each round are in a slightly different order than the order used in encryption. The AES specification has also supplied an equivalent inverse cipher in which the individual parts of each round are in the same order as with encryption. This might make a hardware implementation easier, but I have not used it here.

```
Constants: int Nb = 4; // but it might change someday
           int Nr = 10}, 12}, or 14; }// rounds, for Nk = 4, 6, or 8
Inputs: array in  of 4*Nb bytes // input ciphertext
        array out of 4*Nb bytes // output plaintext
        array w of 4*Nb*(Nr+1) bytes // expanded key
Internal work array:
       state, 2-dim array of 4*Nb bytes, 4 rows and Nb cols
Algorithm:
```

```
void InvCipher(byte[] in, byte[] out, byte[] w) {
   byte[][] state = new byte[4][Nb];
   state = in; // actual component-wise copy
   AddRoundKey(state, w, Nr*Nb, (Nr+1)*Nb - 1);
   for (int round = Nr-1; round >= 1; round--) {
      InvShiftRows(state);
      InvSubBytes(state);
      AddRoundKey(state, w, round*Nb, (round+1)*Nb-1);
      InvMixColumns(state);
   }
   InvShiftRows(state);
   InvSubBytes(state);
   AddRoundKey(state, w, 0, Nb - 1);
   out = state; // component-wise copy
```

24.3 Implementation of InvShiftRows().

This just does the inverse of **`ShiftRows`**: doing a *left* circular shift of rows 1, 2, and 3, by amounts of 1, 2, and 3 bytes. The actual Java code below does this.

```
void InvShiftRows(byte[][] state) {
   byte[] t = new byte[4];
   for (int r = 1; r < 4; r++) {
      for (int c = 0; c < Nb; c++)
         t[(c + r)%Nb] = state[r][c];
      for (int c = 0; c < Nb; c++)
         state[r][c] = t[c];
   }
}
```

24.4 Implementation of InvMixColumns().

The **`MixColumns()`** function was carefully constructed so that it has an inverse. I will add in the theory of this here (or elsewhere) later. For now, it suffices to say that the function multiplied each column by the *inverse* polynomial of $a(x)$:

$$a^{-1}(x) = (0b)x^3 + (0d)x^2 + (09)x + (0e),$$

The resulting function, when simplified, takes the following form in Java pseudo-code, where as before **#** indicates multiplication in the field:

```
void InvMixColumns(byte[][] s) {
   byte[] sp = new byte[4];
   for (int c = 0; c < 4; c++) {
      sp[0] = (0x0e # s[0][c]) ^ (0x0b # s[1][c]) ^
              (0x0d # s[2][c]) ^ (0x09 # s[3][c]);
      sp[1] = (0x09 # s[0][c]) ^ (0x0e # s[1][c]) ^
              (0x0b # s[2][c]) ^ (0x0d # s[3][c]);
      sp[2] = (0x0d # s[0][c]) ^ (0x09 # s[1][c]) ^
```

```
                (0x0e # s[2][c]) ^ (0x0b # s[3][c]);
         sp[3] = (0x0b # s[0][c]) ^ (0x0d # s[1][c]) ^
                (0x09 # s[2][c]) ^ (0x0e # s[3][c]);
         for (int i = 0; i < 4; i++) s[i][c] = sp[i];
      }
   }
```

24.5 Implementation of InvAddRoundKey().

Since the AES specification uses a parameterized **`AddRoundKey()`** function, it is its own inverse, using the parameters in the opposite order. My implementation just lets **`AddRoundKey()`** exclusive-or in another **`16`** bytes every time it is called, so I need a slightly different function, where **`wCount`** is initialized to **`4*Nb*(Nr+1)`**:

```
void InvAddRoundKey(byte[][] state) {
   for (int c = Nb - 1; c >= 0; c--)
      for (int r = 3; r >= 0 ; r--)
         state[r][c] = state[r][c] ^ w[--wCount];
}
```

24.6 Java Implementation of Decryption Using AES.

As before, it's a matter of putting it all together, with a number of details to make the Java work correctly. My Java implementation uses the old **`Tables`**, **`GetBytes`**, **`Copy`**, and **`Print`** classes along with the new classes:

- **`AESdecrypt`**, which provides the principle functions for AES decryption, and
- **`AESinvTest`**, to test decryption.

A combined listing of all the decryption classes appears on page 290.

Test runs of the program, where encryption is followed by decryption, appear on page 293.

Part VII
Identification and Key Distribution

25
The Laws of Cryptography
Passwords

25.1 Passwords.

Passwords are at present the most common method for verifying the identity of a user. This is a flawed method; systems continue to use passwords because of their ease of use and ease of implementation. Among many problems are the successful guessing of user's passwords, and the intercepting of them or uncovering them online. To prevent guessing and for additional security, the NSA recommends using a random 8-letter password that is regularly changed. Since such a stream of passwords is almost impossible to remember (certainly for me), the hapless user is forced to write these passwords down, adding to the insecurity. Thus passwords need to be protected by cryptographic techniques, whether they are stored or transmitted.

Several simple techniques can help make the old-fashioned form of passwords easier to memorize. First, the system can present a user with a list of possible random passwords from which to choose. With such a choice, there may be one password that is easier for a given user to remember. Second, the most common passwords are limited to 8 characters, and experience has shown that users have a hard time picking such a short password that turns out to be secure. If the system allows passwords of arbitrary length (fairly common now), then users can employ *pass phrases*: a phrase or sentence that is not going to be in dictionaries yet is easy for the given user to remember. My favorite pass phrase is "Dexter's mother's bread," but I won't be able to use it any more.

Personal physical characteristics form the basis for a number of identification methods now in use. The characteristics or *biometrics* range from fingerprints to iris patterns, from voice to hand geometry, among many examples. These techniques are outside the scope of this book. The remaining two sections study two uses of one-way functions to help secure passwords.

25.2 Password Files.

A simple system password scheme would just have a secret file holding each user's account name and the corresponding password. There are several problems with this method: if someone manages to read this file, they can immediately pretend to be any of the users listed. Also, someone might find out about a user's likely passwords from passwords used in the past.

For the reasons above and others, early Unix systems protected passwords with a one-way function (described in an earlier chapter). Along with the account name, the one-way function applied to the password is stored. Thus given a user A (Alice, of course), with account name N_A and password P_A, and given a fixed one-way function h, the system would store N_A and

$h(P_A)$ as a table entry in the password file, with similar entries for other users. When Alice supplies her password to the system, the software computes h of her password and compares this result with the table entry. In this way the systems administrators themselves will not know the passwords of users and will not be able to impersonate a user.

In early Unix systems it was a matter of pride to make the password file world readable. A user would try to guess other's passwords by trying a guess P: first calculate $h(P)$ and then compare this with all table entries. There were many values of P to try, such as entries in a dictionary, common names, special entries that are often used as passwords, all short passwords, and all the above possibilities with special characters at the beginning or the end. These "cracker" programs have matured to the point where they can always find at least some passwords if there are quite a few users in the system. Now the password file is no longer public, but someone with root privileges can still get to it, and it sometimes leaks out in other ways.

To make the attack in the previous paragraph harder (that attack is essentially the same as *ciphertext searching*), systems can first choose the one-way function h to be more execution-time intensive. This only slows down the searches by a linear factor. Another approach uses an additional random table entry, called a *salt*. Suppose for example that each password table entry has another random t-bit field (the salt), different for each password. When Alice first puts her password into the system (or changes it), she supplies P_A. The system chooses the salt S_A and calculates $E_A = h(P_A, S_A)$, where h is fixed up to handle two inputs instead of one. The password file entry for Alice now contains A, S_A, and E_A. With this change, an attack on a single user is the same, but the attack of the previous paragraph on all users at the same time now takes either an extra factor of time equal to either 2^t or the number of users, whichever is smaller. Without the salt, an attacker could check if "Dexter" were the password of any user by calculating $h(\text{"Dexter"})$ and doing a fast search of the password file for this entry. With the salt, to check if Alice is using "Dexter" for example, the attacker must retrieve Alice's salt S_A and calculate $h(\text{"Dexter"}, S_A)$. Each user requires a different calculation, so this simple device greatly slows down the dictionary attack.

25.3 Lamport's Scheme.

The schemes mentioned above are useless if an opponent can intercept Alice's communication with the computer when she is trying to log in. She would be sending her password in the clear (without encryption). There are schemes to use encryption to protect the password transmission, but in 1981 Lamport developed a simple scheme that is proof against eavesdroppers.

With Lamport's scheme, Alice first decides how many times she wants to log in without redoing the system, say, 1000 times. She chooses a random secret initial value w_0, and uses a one-way function h to calculate

$$w_1 = h(w), w_2 = h^2(w), \ldots, w_{999} = h^{999}(w), w_{1000} = h^{1000}(w),$$

where $h^2(w_0)$ means $h(h(w_0))$, $h^3(w_0) = h(h(h(w_0)))$, and so forth. Alice wants to authenticate herself repeatedly with the computer system C. She must transfer w_{1000} to C

in some secure way, possibly by just personally dropping it off. This value is stored in the computer system along with a counter value of 1000. When Alice wants to authenticate herself to C, she sends w_{999} and 999 to C. Now C calculates $h(w_{999}) = w_{1000}$ and compares it with the value w_{1000} already stored, authenticating Alice the first time. Now C replaces the old values with w_{999} and 999. Only Alice knows any w value except w_{1000}, so only Alice can calculate and supply w_{999} to C. (Alice could keep all the w_i around, or she could calculate any desired one of them from scratch starting with w_0.) The *next* time Alice wants to authenticate herself to C, she will supply w_{998} and 998.

Even if someone hijacked Alice's communication with C, intercepting w_{998} and pretending to be her, they could only do it once, since they could not calculate the previous value w_{997}. Alice's use of these passwords continues for up to 1000 authentications, at which time Alice must restart the whole system with a new initial value w_0. This system is proof against eavesdropping, and even a more active opponent can only impersonate Alice once without intercepting another of Alice's authentications. If the authentication ever fails for Alice, she will not retry the same w_i, but will use w_{i-1} the next time.

In case of a communication breakdown, the system C may have (say) w_{654} and 654 stored, and may receive from Alice w_{651} and 651, for example. In this case C knows to calculate $h(h(h(w_{651})))$ before comparing with the stored entry, and of course C will store w_{651} and 651 as the new entry.

26
The Laws of Cryptography
Zero-Knowledge Protocols

26.1 The Classes NP and NP-complete.

26.2 Zero-Knowledge Proofs.

26.3 Hamiltonian Cycles.

An *NP*-complete problem known as the *Hamiltonian Cycle Problem* gives a good illustration of a simple zero-knowledge proof. The problem starts with an *undirected graph* $G = (V, E)$, where V is a finite set of *vertices*, E is a set of ordered pairs of vertices forming *edges*, and if (u, v) is an edge, so is (v, u). A *path* in G is a sequence of vertices $v_0, v_1, v_2, \ldots, v_{n-1}$ such for each $0 \leq i \leq n-2$, (v_i, v_{i+1}) is an edge. A path is a *cycle* (ends where it starts) if $v_{n-1} = v_0$. A path is *simple* (doesn't cross itself) if no vertex appears more than once. A path is *complete* if it includes every vertex. The *Hamiltonian Cycle Problem* (**HC**) asks if a given graph has a simple complete cycle. It turns out that **HC** is an *NP*-complete problem, so in general no efficient algorithm is known.

If one had an efficient algorithm to solve **HC**, then one would also have an efficient algorithm to actually obtain the Hamitonian cycle itself. First check the entire graph to see if there is such a cycle. Then try deleting each edge in turn, checking again if there is still a Hamilintonian cycle, until only the edges of a Hamiltonian cycle remain. (There may be more than one such cycle, but this method will find one of them.)

For a given graph, even a large one, it may be easy to decide this problem, but there is no known efficient algorithm to solve the general problem as the number of vertices increases.

Consider now the specific simple graph in Figure 26.1. The graph illustrated in this figure is the same as the vertices and edges of a dodecahedron (a 12-sided regular polyhedron with each side a pentagon). All that is present is a wire framework of the edges, and the framework has been opened up from one side and squashed onto the plane.

This graph is not complicated, but it still takes most people at least a minute or two to find one of the Hamiltonian cycles in the graph. Try to do it now, without peeking ahead (and without writing in the book). Once you have found a cycle, read on. The cycle is shown later in Figure 26.2 as a thicker set of lines.

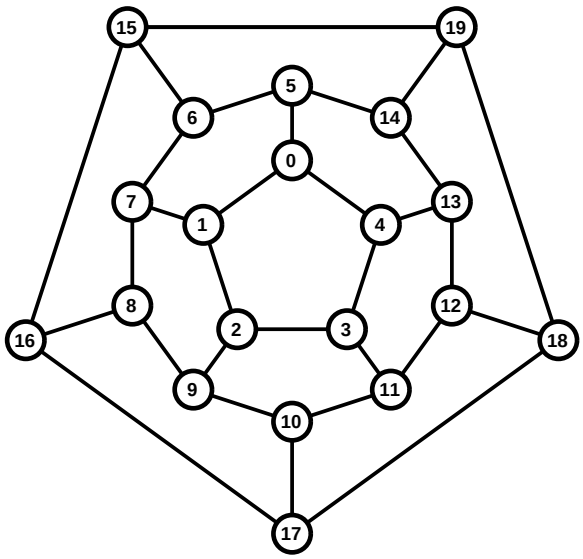


Figure 26.1 Vertices and Edges of a Dodecahedron.

Dodecahedron Graph		
Vertex	**Edges**	**Cycle**
0	1, 4, 5	1
1	0, 2, 7	7
2	1, 3, 9	3
3	2, 4, 11	4
4	0, 3, 13	0
5	0, 6, 14	14
6	5, 7, 15	5
7	1, 6, 8	8
8	7, 9, 16	16
9	2, 8, 10	2
10	9, 11, 17	9
11	3, 10, 12	10
12	11, 13, 18	11
13	4, 12, 14	12
14	5, 13, 19	13
15	6, 16, 19	6
16	8, 15, 17	17
17	10, 16, 18	18
18	12, 17, 19	19
19	14, 15, 18	15

Table 26.1 Dodecahedron Graph.

In computer algorithms, graphs are often represented as a list of vertices, and for each vertex, a list of the vertices which together with the first one make up an edge. Suppose that the vertices are numbered from 0 to $n-1$, or in this case, to 19. Table 26.1 gives information describing the graph in this way. The rightmost column also shows a Hamiltonian cycle by giving, for each vertex, the next vertex in the cycle.

Now on to a zero-knowledge proof by Alice to Bob that she has a Hamiltonian cycle of this graph while revealing nothing about the cycle itself.

Alice must carry out a number of steps in the process of a probablistic proof. After n stages, Bob can be almost certain that Alice has a Hamiltonian cycle for the graph, with only a probability of 2^{-n} that she does not have a cycle, but is cheating. At each stage Alice chooses a new (true) random permutation of the vertices in the graph. In the example below, we assume she has chosen the rearrangement of vertices given in Table 26.2 .

Alice rearranges the table by sorting the newly renumbered vertices into order again. For each vertex, the list of vertices at the other end of an edge must be made all the same length, using extra dummy entries. (In the case here all the lists are already the same length.) Finally

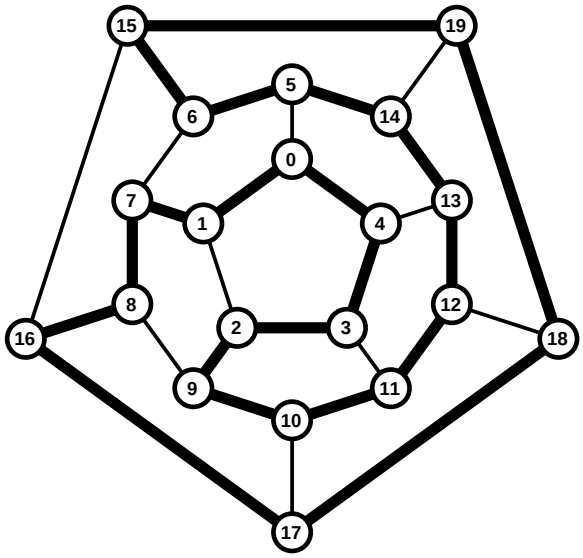


Figure 26.2 Hamiltonian Cycle on a Dodecahedron.

Original vertex:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Changed to:	12	17	16	19	8	13	7	3	10	5	18	11	2	9	15	6	0	14	4	1

Table 26.2 Permutation of Vertices.

Permuted Dodecahedron Graph			
Vertex	**Edges**	**Cycle**	**Permutation**
0	**14**, 6, 10	**14**	16
1	**6**, 4, 15	**6**	19
2	**11**, 4, 9	**11**	12
3	**10**, 7, 17	**10**	7
4	14, **1**, 2	**1**	18
5	**16**, 10, 18	**16**	9
6	0, 1, **7**	**7**	15
7	3, **13**, 6	**13**	6
8	**12**, 9, 19	**12**	4
9	15, **2**, 8	**2**	13
10	5, 3, **0**	**0**	8
11	19, 2, **18**	**18**	11
12	**17**, 8, 13	**17**	0
13	12, 7, **15**	**15**	5
14	18, **4**, 0	**4**	17
15	1, 13, **9**	**9**	14
16	**19**, 17, 5	**19**	2
17	12, 16, **3**	**3**	1
18	11, **5**, 14	**5**	10
19	16, **8**, 11	**8**	3

Table 26.3 Permuted Dodecahedron Graph.

she must randomize the order of these lists. Table 26.3 shows the results of all these changes.

Alice needs a means to selectively reveal parts of this last table, but she has to commit to the entire table without being able to cheat. This is easy to do with any cryptosystem, such as the AES, for example. For each item in the table needing concealment, use a separate encryption with a separate key. (Or similar items can be handled with a single key, as long as an extra random string is included to foil ciphertext comparisons.) Alice sends all the information in the table, with each item encrypted. Notice that if Alice encrypts the vertex number 17 with some AES key, it is not feasible for her to find another key that decrypts the same ciphertext to some other number, say 16, assuming there are a large number of redundant bits in the ciphertext.

So Alice sends the entire table in permuted and encrypted form to Bob. Bob then asks Alice to show him one of two parts of the table:

- the original graph, or
- the Hamiltonian cycle.

In the second case, Alice sends the keys needed to reveal only the parts of the table in **boldface**.

(The vertex numbers are implicit, starting at 0.) This shows each succesive vertex in the cycle, and shows that the vertex is actually on an edge. All this is assuming that the hidden graph is really the correct one. If Alice wanted to cheat, it would be easy for her to make up numbers so that it would look like a Hamiltonian cycle was present.

In the second case, Alice sends the keys needed to reveal all parts of the table except for the column labeled "cycle". The extra column labeled "permutation" is there to make it easy for Bob to verify that the permuted graph is still the same as the original. (There are no efficient algorithms to show that two graphs are really the same.)

If Alice doesn't know a cycle, she can cheat and answer either question correctly, but she has to commit herself to an answer before knowing what part Bob will ask for. In the first case she reveals nothing at all about how the Hamiltonian cycle is situated in the graph, but only that it exists. In the second case she reveals nothing at all about the Hamiltonian cycle. At each stage of this probablistic proof, Alice chooses a new permutation of the vertices, rearranges the table again, and chooses new keys for encrypting. At each step, if she is cheating, she will caught half the time on the average.

Alice could conceivably use **HC** to establish her identity. First she constructs a large, complex graph G, making sure at each stage of the construction that G has a Hamiltonian cycle. Alice securely transfers the graph G to a trusted server T. Then if Alice wanted to prove her identity to Bob, she could have Bob retrieve the graph from T, and then she could prove (probablistically, in zero knowledge) that she has a Hamiltonian cycle. Even an eavesdropper would not learn the Hamiltonian cycle. This particular method has two separate flaws as a way for Alice to identify herself:

- The method is subject to replay attacks, where Boris just sends someone else exactly the same messages Alice transmits as a way to "prove" that he is Alice.
- It is necessary for Alice to create a "hard" instance of **HC**, that is, an instance where known algorithms take a very long time to find a Hamiltonian cycle.

Alice could handle the first flaw by adding a timestamp to the part of the graph that she encrypts. The second flaw above is the difficult one, since finding hard instances of an *NP*-complete problem is often a difficult computation itself, and that especially seems to be the case with **HC**. The next chapter shows how to use factoring as the hard problem, and factoring has been studied much more extensively than **HC**.

26.4 Other Zero-Knowledge Proofs.

27
The Laws of Cryptography
Identification Schemes

27.1 Use of Public Keys.

27.2 Zero-Knowledge Schemes.

This section considers protocols for proving identity, starting with the *Fiat-Shamir Identification Protocol*, a zero-knowledge proof. The *Handbook of Applied Cryptography* lists a number of possible attacks on identification protocols (Section 10.6), including *impersonation*, *replay*, *interleaving*, *reflection*, and *forced delay*. The impersonation attack is the standard attempt to try to break an identification system, and will be discussed below. A replay attack is also standard, and is kept from working by the nature of the challenge-response protocol considered in this section.

The interleaving attack can be very powerful, since there are so many possibilities when two of more protocols are carried out in a concurrent fashion. To give just one example of this difficulty, consider the following strategy for an low-ranked chess player to improve his ranking at online or correspondence chess. The player (call him A signs up for two games with two high-ranked players (call them B and C), where B plays black and C plays white. Player A waits for C to make the first move as white. Then A plays this move in his game with B. Then A waits for B's response as black, and A makes this response to C's move. Continuing in this way, A is guaranteed either a win and a loss or two draws against the good players, and either outcome will improve A's ranking.

There is another attack which I call the *repetitive divide-and-conquer* strategy. This is illustrated with a simple scam used by gamblers. Suppose every Sunday there are sports games which people bet on. Suppose the mark (call him M), gets a letter early in the week predicting the outcome of the game on the next Sunday. M doesn't think much about it, but notices that the prediction is correct. The next week, another prediction comes in the mail that also is correct. This continues for ten weeks, each time a correct prediction of the game outcome arriving days before the game. At this point M receives another letter, offering to predict an eleventh game for $10000 dollars. The letter guarantees that the prediction will be correct, and suggests that M will be able to make far more than this amount of money by betting on the game. The final prediction might or might not be correct, but M was set up. In fact, the letter writer started with (say) 4096 letters, half predicting Sunday's game one way, and half predicting it the other way. After the first game, roughly 2048 of the recipients got a wrong prediction, but they get

no further letters, because the letter writer ignores them, focusing on the (roughly) 2048 people who got a correct first prediction. At the end of 10 weeks, and after writing approximately 8184 letters, one would expect an average of 4 individuals to have received 10 correct predictions, including M above.

Here is the protocol:

Protocol: Fiat-Schamir Identification. A (Alice) proves knowledge of a secret s to B (Bob), using a trusted center T.

1. *Setup for T:* The trusted center T chooses large random primes p and q and forms the product $n = pq$. (These should fit the requirements for the RSA cryptosystem.) T publishes n for all users, and keeps p and q secret, or just destroys them.

2. *Setup for each user:* Each user A chooses a random *secret* s satisfying $1 \leq s \leq n-1$ which has no factors in common with n. Then A calculates $v = s^2 \bmod n$ and lets T store this value in public. (Each user has his own unique secret value s and each user stores a unique square of this value with T. Notice that no secrecy is needed at T, but users need to rely on the integrity of information supplied by T. If someone wants to calculate the secret s just knowing the public value v, they would have to take the discrete square root of s, and this problem is computationally equivalent to factoring n, as was discussing in the chapter on Rabin's cryptosystem.)

3. *Witness step:* A chooses a random number r satisfying $1 \leq r \leq n-1$ to use as a *committment*, calculates $x = r^2 \bmod n$, and sends x to B.

4. *Challenge step:* B chooses a random *challenge bit* e and sends it to A.

5. *Response step:* In case $e = 0$, A sends $y = r$ back to B. In case $e = 1$, A calculates $y = rs \bmod n$, and sends this back to B.

6. *Verification step:* B verifies the response y from A by calculating $z = y^2 \bmod n$. If $e = 0$, then z should be equal to the x that was first sent, that is, z is a square root of the value r^2, a calculation that is only efficiently possible for A. If $e = 1$, then z should be equal to $r^2 s^2 \bmod n = xv \bmod n$, and B must verify this equality.

At each iteration of the above protocol, the chances that A is not who she claims to be and is in fact cheating goes down by a factor of 2. One should repeat the protocol at least 50 times, to reduce the probability of successfully cheating to less than $1/2^{50}$.

As part of the analysis of this protocol, suppose someone else, say C (Carlos), wished to pretend to be A to B. We are not assuming that C can intercept an ongoing protocol between A and B, but we do assume that C can communicate with B, claiming to be A. Of course C can get A's public value v from T, but C does not know A's secret s. C could try to guess what B's challenge bit might be:

- If C guesses that B will respond with a challenge of $e = 0$, then C would only need to send a random $x = r^2 \bmod n$ just like A, and respond to the challenge with the original r. B would accept this round of the protocol.

- If C guesses that B will respond with a challenge of $e = 1$, then C must provide a response y so that when B calculates $y^2 \bmod n$, this is the same as $xv \bmod n$, where x is the initial value sent by C to B, and v is A's public value that anyone (including C) can look up. Notice that C doesn't need to send the square of a random value at the first step, but only needs to send a value x such that later C can send another value y, with the property that when B calculates $y^2 \bmod n$ and $xv \bmod n$, these will work out to be the same. In other words, it needs to be true that $y^2 \bmod n = xv \bmod n$, or that $x = y^2 v^{-1}$, where v^{-1} is the inverse of v using multiplication mod n. (v has such an inverse because s, and therefore v have no factors in common with n.) Thus the initial x that C should send in this case should be the square of some random y times the inverse of v. If C sends this value and if B challenges with the bit $e = 1$, then C will be able to cheat successfully that time. The problem for C is that if he sends such a value for the initial x, and if B challenges with $e = 0$, C should respond with the square root of the original value, but B will not be able to calculate the square root of $x = y^2 v^{-1}$, and so C will fail at this step.

In summary, a cheating C could succeed in either part of the protocol if he knew ahead of time which challenge bit B would send back. C can choose the initial x value to succeed either way, but no single value succeeds both ways for C. Thus C's cheating is detected half of the time at each iteration of the protocol.

28
The Laws of Cryptography
Threshold Schemes

28.1 Introduction to Threshold Schemes.

When all security depends on a single secret key, this key becomes a point of attack and a source of vulnerability. It would be better to have several people *share* the responsibility of maintaining secret keys, and the schemes in this chapter are just what is needed.

Suppose there are 6 executives in a company that maintains a locked room with company secrets inside. When the company was incorporated, the founders wanted to make sure that only a majority (that is, at least 4) of the executives could together gain access to the room. For this purpose, the founders created a special steel door with a sliding rod on its outside. The rod slides through holes in two flanges on the door and through a hole in another flange beside the door. The rod in turn has a ring welded onto it so that it cannot be removed from the two flanges on the door. Between the door's flanges, there are 6 special padlocks around the rod. The dimensions are chosen so that if any four of the six padlocks are removed, the rod will slide just far enough so that the door will open. If fewer than four padlocks are removed, the rod will still barely catch, and the door is still locked. (See the figure for an illustration of this Rube Goldberg contraption.) Each executive is given a key to one of the locks, and then everyone can be sure that only four of more of the executives of the company, working together, can open the door. (There is nothing special about the numbers 4 and 6 in this example.)

In general, a (t, n) *threshold scheme* starts with a *secret* S, with n *users* who can share the secret, and with a *threshold value* of t of the users needed to recover the secret, where $t \leq n$. Each user i is given a *share* of the secret: S_i, for $1 \leq i \leq n$. Then the threshold scheme must somehow arrange that any t of these shares can be used to produce (or compute) the secret, whereas any $t - 1$ or fewer will not allow the secret to be recovered. The door with its rod and locks above give a simple example of a $(4, 6)$ (or "four out of six") threshold scheme, except that its crude mechanical nature is limiting.

28.2 *(t, t)* Threshold Schemes.

Suppose one wants to hide information in computer storage, say a secret book S, regarding it as a string of bits. Given a source of true random numbers, this is easy to do. Create a true random file R_1 with as many bits in it as the information to be secured. Let $R_2 = R_1 \oplus S$. Now keep the files R_1 and R_2 separate from one another, and the book is safe. To recover the book, just calculate $S = R_1 \oplus R_2$. An opponent who knows only R_1 or R_2 has *no information* about the book S. This is obvious for R_1, since it was true random and had nothing to do

with S. The file R_2 was defined using S, but since it is the exclusive-or of a true random bit string, it also has all the properties of a true random bit string, assuming it is taken in isolation. Thus by themselves, each of R_1 and R_2 is true random and each gives no information about the book S. However the two are not independent of one another. The two together include the complete information in S. If S were some illegal book, then just possessing R_1 or R_2 could not possibly be illegal, and only possessing them both together could be illegal. This is an example of a $(2, 2)$ threshold scheme. This scheme gives *perfect security*: just knowing one or the other share gives no information at all about the book.

This method extends easily to a (t, t) threshold scheme, starting with the book S to be kept secret, and with $t-1$ random files $R_1, R_1, \ldots, R_{t-1}$. Define $R_t = R_1 \oplus R_1 \oplus \ldots \oplus R_{t-1} \oplus S$. Then the Rs work as t shares, where again any $t-1$ of them give no information about the book S (all books of the same size are possible and equally likely), but the exclusive-or of all t of them gives S immediately.

28.3 Threshold Schemes Using Planes in Space.

This section describes one method for obtaining a threshold scheme without giving a full implementation. The purpose is to increase intuition about this concept.

A line in the plane is uniquely determined by 2 points on it. This property allows the construction of a $(2, n)$ threshold scheme. Suppose the secret S is a real number. Associate the secret with the point $(0, S)$ in the plane. Choose a non-vertical line L through $(0, S)$ at random, of the form $f(x) = y = ax + S$. Each user i gets the point $(i, f(i)) = (x_i, y_i)$ as his share. From the information that any one user i has, any value for S is possible, because the line might go through (x_i, y_i) and $(0, S)$ for any S. However, if two users get together, their two points uniquely determine the line L, and setting $x = 0$ in the equation for the line gives S immediately.

Similarly a plane in 3-dimentional space is uniquely determined by any three points on it. Just as in the previous paragraph, this leads to a $(3, n)$ threshold scheme. The same thing can be done in t-dimensional space, consisting of all ordered lists of t numbers: $(x_1, x_2, x_3, \ldots, x_t)$. Then a random linear equation of the form $x_t = a_{t-1}x_{t-1} + \ldots a_2x_2 + a_1x_1 + S$ specifies a hyperplane in this space. As above, with only $t-1$ points on this hyperplane, any value of S is still possible, but with t points there will be t equations in t unknowns (the coefficients a_i along with S), which can be solved for all unknowns including the crucial S.

The scheme in this section can be made to work, but I will not finish it, instead turning to another approach. The method of this section would need to switch from floating point numbers, and would still need to decide what picking an equation "at random" might mean.

28.4 Shamir's *(t, n)* Threshold Scheme.

Adi Shamir developed a threshold scheme based on polynomial interpolation. The scheme is based on the fact that a polynomial function of degree $t-1$ is uniquely determined by any t

points on it.

Example. Suppose $t = 3$ and the "secret" is 2. Choose a "random" polynomial $f(x) = y = x^2 - 3x + 2$. Then $f(0) = 2$, the secret. Points on the graph of this function yield shares, for example: $(1, 0), (3, 2), (4, 6), (1/2, 3/4), \ldots$ Using only the first three shares and assuming an equation of the form $y = a_1x^2 + a_2x + a_3$, one gets three linear equations in three unknowns by plugging in the values from the three shares.

$$\begin{aligned} 0 &= \quad a_1 \quad + a_2 + a_3 \\ 2 &= \ 9a_1 + 3a_2 + a_3 \\ 6 &= 16a_1 + 4a_2 + a_3 \end{aligned}$$

These equations are easy to solve, for $a_1 = 1$, $a_2 = -3$, and $a_3 = 2$, and so just from the shares, the secret is seen to be 2.

Here is the general case of Shamir's (t, n) threshold scheme:

1. Start with a secret S, a desired number of shares n, and a threshold t, where all three are integers and $2 \leq t \leq n$.

2. Choose a prime p bigger than both S and n. Everything will be done over the finite field Z_p.

3. Choose a random polynomial of degree $t-1$: $f(x) = a_{t-1}x^{t-1} + \ldots + a_2x^2 + a_1x + S$ by choosing the coefficients a_i uniformly and at random from the interval from 0 to $p-1$ inclusive, that is, from Z_p.

4. Compute n shares as points $(i, f(i))$ on the graph of f. (The x-coordinates do not have to be consecutive integers, but no x-coordinate can be zero, since that would immediately reveal the secret.) These shares are distributed securely to each of the n users.

5. If any t users get together with their shares, they know t distinct points on the polynomial's graph, and so the users can compute the polynomial's coefficients, including the constant term, which is the secret.

There are a number of ways to calculate the polynomial from the shares in the situation above, but perhaps the most convenient way is to use the *Lagrange interpolation formula*: A polynomial $f(x)$ of degree t is uniquely determined by t points, (x_i, y_i) for $1 \leq i \leq t$, assuming that the x_i are all distinct. The polynomial is given by the formula:

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

Here the Greek Σ (sigma) means to add up terms obtained by setting $i = 1, 2, \ldots, t$. Similarly the Greek Π (pi) means to multiply terms following it together, where j takes on values from 1 to t, but leaving off i in each case. The computations are all done in Z_p, that is,

modulo p. In the case of the $x_i - x_j$ in the denominator, one needs to use the multiplicative inverse of $x_i - x_j$ in Z_p.

It is easy to see that this formula works, because it is a degree $t-1$ polynomial in x that agrees with each of the points (x_i, y_i) for $1 \le i \le t$, and this polynomial must be unique. To get a feeling for the equation, look at a special case, say, $t = 3$. Then the equation is

$$f(x) = y_1\left[\frac{x-x_2}{x_1-x_2}\right]\left[\frac{x-x_3}{x_1-x_3}\right] + y_2\left[\frac{x-x_1}{x_2-x_1}\right]\left[\frac{x-x_3}{x_2-x_3}\right] + y_3\left[\frac{x-x_1}{x_3-x_1}\right]\left[\frac{x-x_2}{x_3-x_2}\right].$$

Now evaluate $f(x_1)$, that is, set $x = x_1$ throughout. The multipliers of y_1 both reduce to 1, while there is a 0 times each of y_2 and y_3, so this reduces to just y_1. Similarly, $f(x_2) = y_2$ and $f(x_3) = y_3$. The general formula behaves in exactly the same way.

It is not necessary to determine the entire random polynomial f, but only $f(0)$, which is the secret S. By substituting $x = 0$ above, the equations become

$$f(0) = S = \sum_{i=1}^{t} y_i c_i, \text{ where}$$

$$c_i = \prod_{1 \le j \le t, j \ne i} \frac{x_j}{x_j - x_i}.$$

This is a particularly easy way to calculate the secret from the t shares.

28.5 Properties of Shamir's Scheme.

Shamir's approach to thresholds has a number of pleasant properties (see the *Handbook of Applied Cryptography*):

1. It is *perfect* is the sense that the secret can be computed from t shares, but even $t-1$ shares gives *no information* about the secret. In other words, given $t-1$ shares, all possible values for the secret are still equally probable.

2. One can calculate *new shares* and distribute them to new users along with the ones already passed out.

3. One can distribute more than one share to a user and in this way give that user more power over the secret.

4. In case the secret is too large for the convenient computer arithmetic of a given implementation, the secret can be broken into two or more blocks, and security will still be perfect. Thus there is no reason for large integers and extended precision arithmetic in this example.

Law THRESHOLD-1:
Shamir's (*t*, *n*) threshold scheme gives perfect security for a shared secret, since *t* users can recover the secret, while *t* – 1 or fewer users still have no information about the secret.

28.6 Java Implementation of Shamir's Scheme.

An implementation of Shamir's scheme appears on page 299. Here is the output of a run of the software, where the parameters on the command line are **`s t n p`**, in that order:

```
% java ThresholdTest 88 3 5 97
New (3,5) threshold scheme, with p = 97 and s = 88
Function f(x) = 88*x^0 + 53*x^1 + 43*x^2
All 5 Output Shares: (1,87) (2,75) (3,52) (4,18) (5,70)

Recover secret from t = 3 shares, with p = 97
All 3 Input Shares: (4,18) (2,75) (3,52)
C[0] = 2/(2-4)  ( or 96) 3/(3-4)  ( or 94) = 3
C[1] = 4/(4-2)  ( or 2) 3/(3-2)  ( or 3) = 6
C[2] = 4/(4-3)  ( or 4) 2/(2-3)  ( or 95) = 89
Secret = 3*18 + 6*75 + 89*52 = 88
```

Here is another larger run. (By an unlikely coincidence, the values of the polynomial below are symmetric going down from 2 and up from 3. Something like this will always be true, but it is surprising that it appears here for such small values of x.)

```
% java ThresholdTest 1111 3 5 1999
New (3,5) threshold scheme, with p = 1999 and s = 1111
Function f(x) = 1111*x^0 + 1199*x^1 + 971*x^2
All 5 Output Shares: (1,1282) (2,1396) (3,1453) (4,1453) (5,1396)

Recover secret from t = 3 shares, with p = 1999
All 3 Input Shares: (4,1453) (2,1396) (3,1453)
C[0] = 2/(2-4)  ( or 1998) 3/(3-4)  ( or 1996) = 3
C[1] = 4/(4-2)  ( or 2) 3/(3-2)  ( or 3) = 6
C[2] = 4/(4-3)  ( or 4) 2/(2-3)  ( or 1997) = 1991
Secret = 3*1453 + 6*1396 + 1991*1453 = 1111
```

Yet another run. This would overflow with 32-bit Java **`int`** type, but works because the Java 64-bit **`long`** type is used.

```
% java ThresholdTest 2222222 3 5 10316017
New (3,5) threshold scheme, with p = 10316017 and s = 2222222
Function f(x) = 2222222*x^0 + 8444849*x^1 + 2276741*x^2
All 5 Output Shares: (1,2627795) (2,7586850) (3,6783370) (4,217355) (5,8520839)

Recover secret from t = 3 shares, with p = 10316017
All 3 Input Shares: (4,217355) (2,7586850) (3,6783370)
C[0] = 2/(2-4)  ( or 10316016) 3/(3-4)  ( or 10316014) = 3
C[1] = 4/(4-2)  ( or 2) 3/(3-2)  ( or 3) = 6
```

```
C[2] = 4/(4-3)  ( or 4) 2/(2-3)  ( or 10316015) = 10316009
Secret = 3*217355 + 6*7586850 + 10316009*6783370 = 2222222
```

Java Programs

Program I.1.a
Demonstration of Xor

Referred to from page 4.

The code below shows how **`xor`** can be used to interchange data elements.

Java class: Xor

```
// Xor.java: test xor function ^ for interchanges
public class Xor {

   // main function to try out Base class
   public static void main (String[] args) {
      int a = 123456789, b = -987654321;
      printThem(a, b);
      // interchange a and b
      a = a^b;
      b = a^b;
      a = a^b;
      printThem(a, b);
      a = 234234234; b = -789789789;
      printThem(a, b);
      // interchange a and b
      a ^= b;
      b ^= a;
      a ^= b;
      printThem(a, b);
   } // end of main

   private static void printThem(int a, int b) {
      System.out.println("a: " + a + ", \tb: " + b);
   }
}
```

Here is the output of a run:

```
% java Xor
a: 123456789,   b: -987654321
a: -987654321,  b: 123456789
a: 234234234,   b: -789789789
a: -789789789,  b: 234234234
```

Program I.1.b
Formulas for Logarithms

Referred to from page 5.

Java supplies a function to calculate natural logs, base $e = 2.718281828459045$. To calculate logs to other bases, you need to multiply by a fixed constant: for a log base b multiply by $1/\log_e b$.

Java class: Logs

```
// Logs.java: try out logarithm formulas
public class Logs {

   // main function to try out Logs class
   public static void main (String[] args) {
      System.out.println("log base 2 of 1024 = " + log2(1024));
      System.out.println("log base 10 of 1000 = " + log10(1000));
      System.out.println("log 2 = " + Math.log(2));
      System.out.println("1/log 2 = " + 1/Math.log(2));
      System.out.println("log base 10 of 2 = " + log10(2));
   } // end of main

   // log2:  Logarithm base 2
   public static double log2(double d) {
      return Math.log(d)/Math.log(2.0);
   }

   // log10: Logarithm base 10
   public static double log10(double d) {
      return Math.log(d)/Math.log(10.0);
   }
}
```

Here is the output of a run:

```
% java Logs
log base 2 of 1024 = 10.0
log base 10 of 1000 = 2.9999999999999996
log 2 = 0.6931471805599453
1/log 2 = 1.4426950408889634
log base 10 of 2 = 0.30102999566398114
```

Program I.1.c
Fermat's Theorem Illustrated

Referred to from page 12.

Recall that Fermat's theorem says that given a prime p and a non-zero number a, $a^{p-1} \bmod p$ is always equal to 1. Here is a table for $p = 11$ illustrating this theorem. Notice below that the value is always 1 by the time the power gets to 10, but sometimes the value gets to 1 earlier. The initial run up to the 1 value is shown in boldface in the table. A value of a for which the whole row is bold is called a *generator*. In this case **2**, **6**, **7**, and **8** are generators.

p	a	a^1	a^2	a^3	a^4	a^5	a^6	a^7	a^8	a^9	a^{10}
11	2	**2**	**4**	**8**	**5**	**10**	**9**	**7**	**3**	**6**	**1**
11	3	**3**	**9**	**5**	**4**	**1**	3	9	5	4	1
11	4	**4**	**5**	**9**	**3**	**1**	4	5	9	3	1
11	5	**5**	**3**	**4**	**9**	**1**	5	3	4	9	1
11	6	**6**	**3**	**7**	**9**	**10**	**5**	**8**	**4**	**2**	**1**
11	7	**7**	**5**	**2**	**3**	**10**	**4**	**6**	**9**	**8**	**1**
11	8	**8**	**9**	**6**	**4**	**10**	**3**	**2**	**5**	**7**	**1**
11	9	**9**	**4**	**3**	**5**	**1**	9	4	3	5	1
11	10	**10**	**1**	10	1	10	1	10	1	10	1

Java code to produce the table above and the one below.

Java class: Fermat

```
// Fermat.java: given a prime integer p, calculate powers of a
//   fixed element a mod p.  Output html table
public class Fermat {

   // main function to do all the work
   public static void main (String[] args) {
      long p = (Long.parseLong(args[0])); // the fixed prime base
      System.out.println("<table border nosave >");
      System.out.println("<tr><th>p</th><th>a</th><th></th>");
      for (int col = 1; col < p; col++)
         System.out.print("<th>a<sup>" + col + "</sup></th>");
      System.out.println("</tr><tr colspan=" + (p+2) + "></tr>");
      for (long row = 2; row < p; row++) {
         System.out.print("<tr align=right><td>" + p);
         System.out.print("</td><td>" + row + "</td><td></td>");
         boolean firstCycle = true;
         for (long col = 1; col < p; col++) {
            if (firstCycle)
               System.out.print("<td><b><font color=FF0000>" +
                  pow(row, col, p) + "</font></b></td>");
            else
               System.out.print("<td>" + pow(row, col, p) + "</td>");
```

```
            if (firstCycle)
               if (pow(row, col, p) == 1) firstCycle = false;
         }
         System.out.println("</tr>");
      }
      System.out.println("</table>");
   } // end of main

   // pow: calculate x^y mod p, without overflowing
   //    (Algorithm from Gries, The Science of Programming, p. 240
   public static long pow(long x, long y, long p) {
      long z = 1;
      while (y > 0) {
         while (y%2 == 0) {
            x = (x*x) % p;
            y = y/2;
         }
         z = (z*x) % p;
         y = y - 1;
      }
      return z;
   }
}
```

Here is a larger table with **`p = 23`**. There are **`10`** generators.

p	a	a^1	a^2	a^3	a^4	a^5	a^6	a^7	a^8	a^9	a^{10}	a^{11}	a^{12}	a^{13}	a^{14}	a^{15}	a^{16}	a^{17}	a^{18}	a^{19}	a^{20}	a^{21}	a^{22}
23	2	**2**	**4**	**8**	**16**	**9**	**18**	**13**	**3**	**6**	**12**	**1**	2	4	8	16	9	18	13	3	6	12	1
23	3	**3**	**9**	**4**	**12**	**13**	**16**	**2**	**6**	**18**	**8**	**1**	3	9	4	12	13	16	2	6	18	8	1
23	4	**4**	**16**	**18**	**3**	**12**	**2**	**8**	**9**	**13**	**6**	**1**	4	16	18	3	12	2	8	9	13	6	1
23	5	**5**	**2**	**10**	**4**	**20**	**8**	**17**	**16**	**11**	**9**	**22**	**18**	**21**	**13**	**19**	**3**	**15**	**6**	**7**	**12**	**14**	**1**
23	6	**6**	**13**	**9**	**8**	**2**	**12**	**3**	**18**	**16**	**4**	**1**	6	13	9	8	2	12	3	18	16	4	1
23	7	**7**	**3**	**21**	**9**	**17**	**4**	**5**	**12**	**15**	**13**	**22**	**16**	**20**	**2**	**14**	**6**	**19**	**18**	**11**	**8**	**10**	**1**
23	8	**8**	**18**	**6**	**2**	**16**	**13**	**12**	**4**	**9**	**3**	**1**	8	18	6	2	16	13	12	4	9	3	1
23	9	**9**	**12**	**16**	**6**	**8**	**3**	**4**	**13**	**2**	**18**	**1**	9	12	16	6	8	3	4	13	2	18	1
23	10	**10**	**8**	**11**	**18**	**19**	**6**	**14**	**2**	**20**	**16**	**22**	**13**	**15**	**12**	**5**	**4**	**17**	**9**	**21**	**3**	**7**	**1**
23	11	**11**	**6**	**20**	**13**	**5**	**9**	**7**	**8**	**19**	**2**	**22**	**12**	**17**	**3**	**10**	**18**	**14**	**16**	**15**	**4**	**21**	**1**
23	12	**12**	**6**	**3**	**13**	**18**	**9**	**16**	**8**	**4**	**2**	**1**	12	6	3	13	18	9	16	8	4	2	1
23	13	**13**	**8**	**12**	**18**	**4**	**6**	**9**	**2**	**3**	**16**	**1**	13	8	12	18	4	6	9	2	3	16	1
23	14	**14**	**12**	**7**	**6**	**15**	**3**	**19**	**13**	**21**	**18**	**22**	**9**	**11**	**16**	**17**	**8**	**20**	**4**	**10**	**2**	**5**	**1**
23	15	**15**	**18**	**17**	**2**	**7**	**13**	**11**	**4**	**14**	**3**	**22**	**8**	**5**	**6**	**21**	**16**	**10**	**12**	**19**	**9**	**20**	**1**
23	16	**16**	**3**	**2**	**9**	**6**	**4**	**18**	**12**	**8**	**13**	**1**	16	3	2	9	6	4	18	12	8	13	1
23	17	**17**	**13**	**14**	**8**	**21**	**12**	**20**	**18**	**7**	**4**	**22**	**6**	**10**	**9**	**15**	**2**	**11**	**3**	**5**	**16**	**19**	**1**
23	18	**18**	**2**	**13**	**4**	**3**	**8**	**6**	**16**	**12**	**9**	**1**	18	2	13	4	3	8	6	16	12	9	1
23	19	**19**	**16**	**5**	**3**	**11**	**2**	**15**	**9**	**10**	**6**	**22**	**4**	**7**	**18**	**20**	**12**	**21**	**8**	**14**	**13**	**17**	**1**
23	20	**20**	**9**	**19**	**12**	**10**	**16**	**21**	**6**	**5**	**8**	**22**	**3**	**14**	**4**	**11**	**13**	**7**	**2**	**17**	**18**	**15**	**1**
23	21	**21**	**4**	**15**	**16**	**14**	**18**	**10**	**3**	**17**	**12**	**22**	**2**	**19**	**8**	**7**	**9**	**5**	**13**	**20**	**6**	**11**	**1**
23	22	**22**	**1**	22	1	22	1	22	1	22	1	22	1	22	1	22	1	22	1	22	1	22	1

Program I.2.a
Basic GCD Algorithm

Referred to from page 15.

This class gives (in **bold** below) two versions of the simple greatest common division algorithm: the first recursive and the second iterative.

Java class: GCD

```
// GCD: Greatest Common Divisor
public class GCD {

   public static long gcd1(long x, long y) {
      if (y == 0) return x;
      return gcd1(y, x % y);
   }

   public static long gcd2(long x, long y) {
      while (y != 0) {
         long r = x % y;
         x = y; y = r;
      }
      return x;
   }

   public static void main(String[] args) {
      long x = Long.parseLong(args[0]);
      long y = Long.parseLong(args[1]);
      long z = GCD.gcd1(x, y);
      System.out.println("Method 1: gcd(" + x + ", " +
                                            y + ") = " + z);
           z = GCD.gcd2(x, y);
      System.out.println("Method 2: gcd(" + x + ", " +
                                            y + ") = " + z);
   }
}
```

Here are several runs of this program:

```
% java GCD 819 462
Method 1: gcd(819, 462) = 21
Method 2: gcd(819, 462) = 21
% java GCD 40902 24140
Method 1: gcd(40902, 24140) = 34
Method 2: gcd(40902, 24140) = 34
```

Program I.2.b
Extended GCD Algorithm

Referred to from page 16.

This extended greatest common divison algorithm is the version in Knuth's *Seminumerical Algorithms*, Third Edition. When the algorithm finishes, **`x*u[0] + y*u[1] = u[2] = gcd(x, y)`**.

Java class: ExtGCDsimple

```
// ExtGCDsimple: Extended GCD
public class ExtGCDsimple {

   public static long[] GCD(long x, long y) {
      long[] u = {1, 0, x}, v = {0, 1, y}, t = new long[3];
      while (v[2] != 0) {
         long q = u[2]/v[2];
         for (int i = 0; i < 3; i++) {
            t[i] = u[i] -v[i]*q; u[i] = v[i]; v[i] = t[i];
         }
      }
      return u;
   }

   public static void main(String[] args) {
      long[] u = new long[3];
      long x = Long.parseLong(args[0]);
      long y = Long.parseLong(args[1]);
      u = ExtGCDsimple.GCD(x, y);
      System.out.println("gcd(" + x + ", " + y + ") = " + u[2]);
      System.out.println("(" + u[0] + ")*" + x + " + " +
                         "(" + u[1] + ")*" + y + " = " + u[2]);
   }
}
```

Here are several runs of this program:

```
% java ExtGCDsimple 819 462
gcd(819, 462) = 21
(-9)*819 + (16)*462 = 21
% java ExtGCDsimple 40902 24140
gcd(40902, 24140) = 34
(337)*40902 + (-571)*24140 = 34
```

Program I.2.c
Extended GCD Algorithm (debug version)

Referred to from page 17.

The long (debug oriented) version of this program was discussed in the text of the book.

Java class: ExtGCD

```
// ExtGCD: Extended GCD (long version)
public class ExtGCD {

   public static long[] GCD(long x, long y) { // assume not 0 or neg
      long[] u = new long[3];
      long[] v = new long[3];
      long[] t = new long[3];
      // at all stages, if w is any of the 3 vectors u, v or t, then
      //   x*w[0] + y*w[1] = w[2]  (this is verified by "check" below)
      // u = (1, 0, u); v = (0, 1, v);
      u[0] = 1; u[1] = 0; u[2] = x; v[0] = 0; v[1] = 1; v[2] = y;
      System.out.println("q\tu[0]\tu[1]\tu[2]\tv[0]\tv[1]\tv[2]");

      while (v[2] != 0) {
         long q = u[2]/v[2];
         // t = u - v*q;
         t[0] = u[0] -v[0]*q; t[1] = u[1] -v[1]*q; t[2] = u[2] -v[2]*q;
                              check(x, y, t);
         // u = v;
         u[0] = v[0]; u[1] = v[1]; u[2] = v[2]; check(x, y, u);
         // v = t;
         v[0] = t[0]; v[1] = t[1]; v[2] = t[2]; check(x, y, v);
         System.out.println(q + "\t"+ u[0] + "\t" + u[1] + "\t" + u[2] +
                              "\t"+ v[0] + "\t" + v[1] + "\t" + v[2]);
      }
      return u;
   }

   public static void check(long x, long y, long[] w) {
      if (x*w[0] + y*w[1] != w[2]) {
         System.out.println("*** Check fails: " + x + " " + y);
         System.exit(1);
      }
   }

   public static void main(String[] args) {
      long[] u = new long[3];
      long x = Long.parseLong(args[0]);
      long y = Long.parseLong(args[1]);
      u = ExtGCD.GCD(x, y);
      System.out.println("\ngcd(" + x + ", " + y + ") = " + u[2]);
```

```
            System.out.println("(" + u[0] + ")*" + x + " + " +
                               "(" + u[1] + ")*" + y + " = " + u[2]);
        }
    }
```

Here is a sample run (with a few extra tabs inserted by hand):

```
% java ExtGCD 123456789 987654321
q               u[0]    u[1]    u[2]            v[0]            v[1]            v[2]

0               0       1       987654321       1               0               123456789
8               1       0       123456789       -8              1               9
13717421        -8      1       9               109739369       -13717421       0

gcd(123456789, 987654321) = 9
(-8)*123456789 + (1)*987654321 = 9

% java ExtGCD 1122334455667788 99887766554433
q               u[0]            u[1]            u[2]            v[0]            v[1]            v[2]

11              0               1               99887766554433  1               -11             23569023569025
4               1               -11             23569023569025  -4              45              5611672278333
4               -4              45              5611672278333   17              -191            1122334455693
4               17              -191            1122334455693   -72             809             1122334455561
1               -72             809             1122334455561   89              -1000           132
8502533754      89              -1000           132             -756725504178   8502533754809   33
4               -756725504178   8502533754809   33              3026902016801   -34010135020236 0

gcd(1122334455667788, 99887766554433) = 33
(-756725504178)*1122334455667788 + (8502533754809)*99887766554433 = 33

% java ExtGCD 384736948574637 128475948374657
q       u[0]            u[1]            u[2]            v[0]            v[1]            v[2]

2       0               1               128475948374657 1               -2              127785051825323
1       1               -2              127785051825323 -1              3               690896549334
184     -1              3               690896549334    185             -554            660086747867
1       185             -554            660086747867    -186            557             30809801467
21      -186            557             30809801467     4091            -12251          13080917060
2       4091            -12251          13080917060     -8368           25059           4647967347
2       -8368           25059           4647967347      20827           -62369          3784982366
1       20827           -62369          3784982366      -29195          87428           862984981
4       -29195          87428           862984981       137607          -412081         333042442
2       137607          -412081         333042442       -304409         911590          196900097
1       -304409         911590          196900097       442016          -1323671        136142345
1       442016          -1323671        136142345       -746425         2235261         60757752
2       -746425         2235261         60757752        1934866         -5794193        14626841
4       1934866         -5794193        14626841        -8485889        25412033        2250388
6       -8485889        25412033        2250388         52850200        -158266391      1124513
2       52850200        -158266391      1124513         -114186289      341944815       1362
825     -114186289      341944815       1362            94256538625     -282262738766   863
1       94256538625     -282262738766   863             -94370724914    282604683581    499
1       -94370724914    282604683581    499             188627263539    -564867422347   364
1       188627263539    -564867422347   364             -282997988453   847472105928    135
2       -282997988453   847472105928    135             754623240445    -2259811634203  94
1       754623240445    -2259811634203  94              -1037621228898  3107283740131   41
2       -1037621228898  3107283740131   41              2829865698241   -8474379114465  12
3       2829865698241   -8474379114465  12              -9527218323621  28530421083526  5
2       -9527218323621  28530421083526  5               21884302345483  -65535221281517 2
2       21884302345483  -65535221281517 2               -53295823014587 159600863646560 1
2       -53295823014587 159600863646560 1               28475948374657 -384736948574637 0

gcd(384736948574637, 128475948374657) = 1
(-53295823014587)*384736948574637 + (159600863646560)*128475948374657 = 1
```

Program I.2.d
Testing Two Exponential Algorithms

Referred to from page 19.

The code below the two algorithms for carrying out integer exponentiation that were presented in the text. Each function has additional debug statements to provide extra output.

Java class: Exp

```
// Exp: test Java versions of two exponentiation algorithms
public class Exp {

   // exp1: uses binary representation of the exponent.
   //   Works on binary bits from most significant to least.
   //   Variable y only present to give loop invariant:
   //     x^y = z, and y gives the leading bits of Y.
   public static long exp1(long x, long Y[], int k) {
      long y = 0, z = 1;
      int round = 0;
      dump1("Initial.  ", x, y, z);
      for (int i = k; i >= 0; i--) {
         y = 2*y;
         z = z*z;
         dump1("Round: " + (round) + ", ", x, y, z);
         if (Y[i] == 1) {
            y++;
            z = z*x;
            dump1("Round: " + (round++) + ", ", x, y, z);
         }
      }
      return z;
   }

   // dump1: function to spit out debug information
   private static void dump1(String s, long x, long y, long z) {
      System.out.println(s + "x: " + x + ",\ty: " + y +
         ",\tz: " + z + ",\t(x^y): " + (exp(x, y)));
   }

   // exp2: uses binary rep of exponent, without constructing it.
   //   Works on binary bits from least significant to most.
   //   Loop invariant is:  z*x^y = X^Y
   public static long exp2(long X, long Y) {
      long x = X, y = Y, z = 1;
      dump2("Initial.  ", x, y, z);
      int round = 1;
      while (y > 0) {
         while (y%2 == 0) {
            x = x*x;
            y = y/2;
            dump2("Round: " + (round) + ", ", x, y, z);
```

```
            }
             z = z*x;
             y = y - 1;}
             dump2("Round: " + (round++) + ", ", x, y, z);
          {\em{}
          return z;
       }

       // exp: extra copy of exp2 function without debug code
       public static long exp(long X, long Y) {
          long x = X, y = Y, z = 1;
          while (y > 0) {
             while (y%2 == 0) {
                x = x*x;
                y = y/2;
             }
             z = z*x;
             y = y - 1;
          }
          return z;
       }

       // dump2: function to spit out debug information
       private static void dump2(String s, long x, long y, long z) {
          System.out.println(s + "x: " + x + ",\ty: " + y +
             ",\tz: " + z + ",\tz*(x^y): " + (z*exp(x, y)));
       }

       public static void main(String[] args) {
          long x = Long.parseLong(args[0]);
          long y = Long.parseLong(args[1]);
          // Convert y to array Y of bits
          long Y[] = new long[50];
          int k = 0;
          long yt = y;
          while (yt > 0) {
             Y[k++] = yt % 2;
             yt = yt/2;
          }
          k--;
          System.out.println("Try first exponentiation algorithm ...");
          long z1 = Exp.exp1(x, Y, k);
          System.out.println("Method 1: exp1(" + x + ", " + y +
             ") = " + z1 + "\n");
          System.out.println("Try second exponentiation algorithm ...");
          long z2 = Exp.exp2(x, y);
          System.out.println("Method 2: exp2(" + x + ", " + y +
             ") = " + z2);
       }
    }
```

Here are results of test runs, with a few extra blanks to improve readability:

```
% java Exp 3 12
Try first exponentiation algorithm ...
Initial.  x: 3, y: 0,   z: 1,       (x^y): 1
Round: 1, x: 3, y: 0,   z: 1,       (x^y): 1
Round: 1, x: 3, y: 1,   z: 3,       (x^y): 3
Round: 2, x: 3, y: 2,   z: 9,       (x^y): 9
Round: 2, x: 3, y: 3,   z: 27,      (x^y): 27
Round: 3, x: 3, y: 6,   z: 729,     (x^y): 729
Round: 3, x: 3, y: 12,  z: 531441,  (x^y): 531441
Method 1: exp1(3, 12) = 531441

Try second exponentiation algorithm ...
Initial.  x: 3,         y: 12,  z: 1,        z*(x^y): 531441
Round: 1, x: 9,         y: 6,   z: 1,        z*(x^y): 531441
Round: 1, x: 81,        y: 3,   z: 1,        z*(x^y): 531441
Round: 1, x: 81,        y: 2,   z: 81,       z*(x^y): 531441
Round: 2, x: 6561,      y: 1,   z: 81,       z*(x^y): 531441
Round: 2, x: 6561,      y: 0,   z: 531441,   z*(x^y): 531441
Method 2: exp2(3, 12) = 531441

% java Exp 2 23
Try first exponentiation algorithm ...
Initial.  x: 2, y: 0,   z: 1,         (x^y): 1
Round: 1, x: 2, y: 0,   z: 1,         (x^y): 1
Round: 1, x: 2, y: 1,   z: 2,         (x^y): 2
Round: 2, x: 2, y: 2,   z: 4,         (x^y): 4
Round: 2, x: 2, y: 4,   z: 16,        (x^y): 16
Round: 2, x: 2, y: 5,   z: 32,        (x^y): 32
Round: 3, x: 2, y: 10,  z: 1024,      (x^y): 1024
Round: 3, x: 2, y: 11,  z: 2048,      (x^y): 2048
Round: 4, x: 2, y: 22,  z: 4194304,   (x^y): 4194304
Round: 4, x: 2, y: 23,  z: 8388608,   (x^y): 8388608
Method 1: exp1(2, 23) = 8388608

Try second exponentiation algorithm ...
Initial.  x: 2,         y: 23,  z: 1,        z*(x^y): 8388608
Round: 1, x: 2,         y: 22,  z: 2,        z*(x^y): 8388608
Round: 2, x: 4,         y: 11,  z: 2,        z*(x^y): 8388608
Round: 2, x: 4,         y: 10,  z: 8,        z*(x^y): 8388608
Round: 3, x: 16,        y: 5,   z: 8,        z*(x^y): 8388608
Round: 3, x: 16,        y: 4,   z: 128,      z*(x^y): 8388608
Round: 4, x: 256,       y: 2,   z: 128,      z*(x^y): 8388608
Round: 4, x: 65536,     y: 1,   z: 128,      z*(x^y): 8388608
Round: 4, x: 65536,     y: 0,   z: 8388608,  z*(x^y): 8388608
Method 2: exp2(2, 23) = 8388608
```

Program II.3.a
Formula for Channal Capacity

Referred to from page 26.

Program with the simple formula for channel capacity:

Java class: Capacity

```
// Capacity.java: calculate channel capacity, binary symmetric channel
//   p: the channel probability for a binary symmetric channel
public class Capacity {

   // main function to do calculation
   public static void main (String[] args) {
      double p = Double.parseDouble(args[0]); // channel probability
      System.out.println("Probability: " + p +
          ", Capacity: " + capacity(p));
   } // end of main

   // capacity: the capacity of the binary symmetric channel
   private static double capacity(double p) {
      if (p == 0 || p == 1) return 1;
      return 1 + p*log2(p) + (1 - p)*log2(1 - p);
   }

   // log2:  Logarithm base 2
   public static double log2(double d) {
      return Math.log(d)/Math.log(2.0);
   }
}
```

Typical output:

```
% javac Capacity.java
% java Capacity 0.3
Probability: 0.3, Capacity: 0.11870910076930735
% java Capacity 0.999
Probability: 0.999, Capacity: 0.9885922422625388
% java Capacity 0.001
Probability: 0.0010, Capacity: 0.9885922422625388
% java Capacity 0.5
Probability: 0.5, Capacity: 0.0
% java Capacity 0.5001
Probability: 0.5001, Capacity: 2.8853901046232977E-8
% java Capacity 0.51
Probability: 0.51, Capacity: 2.8855824719009604E-4
```

Program II.3.b
Table of Channal Capacities

Referred to from page 26.

Here is the program to print an HTML table of channel capacities. The resulting table (when interpreted by an HTML browser) will print a table that looks much like the one in the text.

Java class: CapacityTable

```
// CapacityTable.java: print table of capacities
//   p: the channel probability for a binary symmetric channel
import java.text.DecimalFormat;
public class CapacityTable {
   static final int TABLE_SIZE = 20;
   static DecimalFormat twoDigits = new DecimalFormat("0.00");
   static DecimalFormat fifteenDigits = new
                DecimalFormat("0.000000000000000");

   // main function to do calculation
   public static void main (String[] args) {
      double p; // channel probability
      System.out.println("<table border>");
      System.out.println("<tr><td><b>Probability</b></td>");
      System.out.println("<td><b>Channel Capacity</b></td></tr>");
      System.out.println("<tr><td></td><td></td></tr>");
      for (int i = 0; i <= TABLE_SIZE/2; i++) {
         p = (double)i/TABLE_SIZE;
         System.out.print("<tr><td>" + twoDigits.format(p));
         System.out.print(" or " + twoDigits.format(1-p));
         System.out.println("</td><td>" +
            fifteenDigits.format(capacity(p)) + "</td></tr>");
      }
      System.out.println("</table>");
   } // end of main

   // capacity: the capacity of the binary symmetric channel
   private static double capacity(double p) {
      if (p == 0 || p == 1) return 1;
      return 1 + p*log2(p) + (1 - p)*log2(1 - p);
   }

   // log2:  Logarithm base 2
   public static double log2(double d) {
      return Math.log(d)/Math.log(2.0);
   }
}
```

Here is the output, as an HTML table:

```
<table border>
<tr><td><b>Probability</b></td>
```

```
<td><b>Channel Capacity</b></td></tr>
<tr><td></td><td></td></tr>
<tr><td>0.00 or 1.00</td><td>1.000000000000000</td></tr>
<tr><td>0.05 or 0.95</td><td>0.713603042884044</td></tr>
<tr><td>0.10 or 0.90</td><td>0.531004406410719</td></tr>
<tr><td>0.15 or 0.85</td><td>0.390159695283600</td></tr>
<tr><td>0.20 or 0.80</td><td>0.278071905112638</td></tr>
<tr><td>0.25 or 0.75</td><td>0.188721875540867</td></tr>
<tr><td>0.30 or 0.70</td><td>0.118709100769307</td></tr>
<tr><td>0.35 or 0.65</td><td>0.065931944624509</td></tr>
<tr><td>0.40 or 0.60</td><td>0.029049405545331</td></tr>
<tr><td>0.45 or 0.55</td><td>0.007225546012192</td></tr>
<tr><td>0.50 or 0.50</td><td>0.000000000000000</td></tr>
</table>
```

Program II.3.c
Inverse of the Channal Capacity formula

Referred to from page 26.

Here is a Java program that prints a table of channel capacities and corresponding channel probabilities. (The function calculating the inverse of the channel capacity function is given in **boldface**.)

Java class: CapacityInverse

```
// CapacityInverse.java: print table of inverse capacities
//   p: the channel probability for a binary symmetric channel
import java.text.DecimalFormat;
public class CapacityInverse {
   static final int TABLE_SIZE = 20;
   static DecimalFormat eightDigits =
         new DecimalFormat("0.00000000");
   // main function to do calculation
   public static void main (String[] args) {
      double p; // channel probability
      double c; // channel capacity
      System.out.println("<table border><tr align=center>");
      System.out.println("<td><b>Channel<br>Capacity</b></td>");
      System.out.println("<td><b>Probability<br>p</b></td>");
      System.out.println("<td><b>Probability<br>1-p</b></td></tr>");
      System.out.println("<tr><td></td><td></td></tr>");
      for (int i = 0; i <= TABLE_SIZE; i++) {
         c = (double)i/TABLE_SIZE;
         System.out.print("<tr><td>" + c);
         if ((int)(10*c) == 10*c) System.out.print("0");
         System.out.print("</td><td>" +
            eightDigits.format(capacityInverse(c)) + "</td>");
         System.out.println("</td><td>" +
            eightDigits.format(1 - capacityInverse(c)) +
               "</td></tr>");
      }
      System.out.println("</table>");
   } // end of main

   // capacity: the capacity of the binary symmetric channel
   private static double capacity(double p) {
      if (p == 0 || p == 1) return 1;
      return 1 + p*log2(p) + (1 - p)*log2(1 - p);
   }

   // capacityInverse: the inverse of the capacity function,
   //    uses simple bisection method
   private static double capacityInverse(double c) {
      if (c < 0 || c > 1) return -1;
      double lo = 0, hi = 0.5, mid, cLo, cHi, cMid;
```

```
      do {
         mid = (lo + hi)/2;
         cLo = capacity(lo);
         cHi = capacity(hi);
         cMid = capacity(mid);
         if (c > cMid) hi = mid;
         else lo = mid;
      } while (hi - lo > 1.0E-15);
      return mid;
   }

   // log2:  Logarithm base 2
   public static double log2(double d) {
      return Math.log(d)/Math.log(2.0);
   }
}
```

Here is the table printed by the above program (roughly as it would look in a browser):

Channel Capacity	Probability p	Probability 1 - p
0.00	0.50000000	0.50000000
0.05	0.36912775	0.63087225
0.10	0.31601935	0.68398065
0.15	0.27604089	0.72395911
0.20	0.24300385	0.75699615
0.25	0.21450174	0.78549826
0.30	0.18929771	0.81070229
0.35	0.16665701	0.83334299
0.40	0.14610240	0.85389760
0.45	0.12730481	0.87269519
0.50	0.11002786	0.88997214
0.55	0.09409724	0.90590276
0.60	0.07938260	0.92061740
0.65	0.06578671	0.93421329
0.70	0.05323904	0.94676096
0.75	0.04169269	0.95830731
0.80	0.03112446	0.96887554
0.85	0.02153963	0.97846037
0.90	0.01298686	0.98701314
0.95	0.00560717	0.99439283
1.00	0.00000000	1.00000000

Program II.3.d
Table of Repetition Codes

Referred to from page 28.

Here is a Java program that will generate the entire table, for any input probability p:

Java class: RepetitionTable

```
// RepetitionTable.java: given p, calculate a table of duplicates
//   p: the channel probability for a binary symmetric channel
public class RepetitionTable {

   // main function to do calculation
   public static void main (String[] args) {
      double p = Double.parseDouble(args[0]); // channel probability
      int[] values = {1, 3, 5, 7, 9, 11, 25, 49, 99, 199};
      System.out.println("<table BORDER NOSAVE >");
      System.out.println("<tr><td align=center><b>Number of</b>");
      System.out.println(" <br><b>Duplicates</b></td>");
      System.out.println(" <td align=center><b>Transmission</b>");
      System.out.println(" <br><b>Rate</b></td>");
      System.out.println(" <td align=center><b>Error</b>");
      System.out.println(" <br><b>Rate</b></td>");
      System.out.println(" <td align=center><b>Success</b>");
      System.out.println(" <br><b>Rate</b></td></tr>");
      for (int len = 0; len < values.length; len++) {
         int n = values[len];
         double result = 0;
         for (long i = n; i > n/2; i--)
            result += comb(n,i)*Math.pow(p,i)*Math.pow(1-p,n-i);
         System.out.println("<tr><td>" + n + "</td><td>" +
             100.0/n + "\%</td><td>" + (100.0 - 100.0*result) +
             "\%</td><td>" + 100.0*result + "\%</td></tr>" );
      }
      System.out.println("</table>");
   } // end of main

   // comb(n, i): # of combinations of n things taken i at a time
   private static double comb(long n, long i) {
      double result = 1.0;
      if (i < n/2) i = n-i;
      for (long j = n; j > i; j--)
         result *= ((double)j/(j-i));
      return result;
   }
}
```

Here is the table generated for p = 2/3:

Number of Duplicates	Transmission Rate	Error Rate	Success Rate
1	100.0%	33.3333333333334%	66.6666666666666%
3	33.333333333333336%	25.925925925926023%	74.07407407407398%
5	20.0%	20.987654320987772%	79.01234567901223%
7	14.285714285714286%	17.329675354366827%	82.67032464563317%
9	11.11111111111111%	14.484580602550523%	85.51541939744948%
11	9.090909090909092%	12.208504801097504%	87.7914951989025%
25	4.0%	4.151367840779045%	95.84863215922095%
49	2.0408163265306123%	0.7872479136560173%	99.21275208634398%
99	1.0101010101010102%	0.030913686260717554%	99.96908631373928%
199	0.5025125628140703%	6.250990635692233E-5%	99.99993749009364%

Here is the same table “cleaned up” a bit by hand:

Number of Duplicates	Transmission Rate	Error Rate	Success Rate
1	100.0%	33.3%	66.7%
3	33.3%	25.9%	74.1%
5	20.0%	20.99%	79.01%
7	14.3%	17.33%	82.67%
9	11.1%	14.48%	85.52%
11	9.1%	12.21%	87.79%
25	4.0%	4.15%	95.85%
49	2.0%	0.787%	99.213%
99	1.0%	0.0309%	99.9691%
199	0.5%	0.0000625%	99.9999375%

Program II.4.a
The Simulation Program

Referred to from page 33.

Here is the Java simulation program in three files. The program uses the blocksize (variable **N** in file **Shannon.java**, accessible as a command line argument). The program calculates **2**N** as the size of the code table (variable **expN** in file **Shannon.java**). The length of each codeword in bytes is also a variable (**CWS** in file **Shannon.java**) accessible as a command line argument. Thus the number of bits in each codeword is **8*CWS**. The main data structure is the coding table: **expN** entries each of size **CWS** bytes. Each entry is the class **Word**, and the table itself is of class **Table**. This coding table is allocated inside **Table.java**, and each entry is allocated inside **Word.java** and filled with random bits. The simulation is repeated **simSize** many times (another command line argument inside **Shannon.java**). At each iteration, a random index in the coding table is chosen (length **N** bits), and the corresponding codeword (length **CWS** bytes) is fetched from the table. The codeword is "perturbed" by reversing each bit with probability **1 - p = 0.25**, where **p** is a variable inside **Shannon.java**. The table is then checked for the closest match to this new perturbed word. Here "closest" means to check each entry to see the number of bit positions in which it differs from the perturbed word. The program focuses on the word or words in the table that differ from the perturbed word in the smallest number of bit positions. If there is more than 1 "closest match", this is regarded as an error, as is the case in which the closest match is a word different from the original unperturbed word. (In case of more than one closest match, one could choose a word at random, but this program does not do that.) The error rate is simply the percent of errors compared with all trials. The program uses a reasonably clever and efficient method for comparing codewords (as bit strings). They are compared byte-by-byte. To compare two bytes, say **b1** and **b1**, in function **countDiffs** inside file **Table.java**, the function first calculates **b = b1 ĥ2** (the bit-wise exclusive-or). A **1** bit in **b** represents a difference in the two byte values, so one needs only to count the number of **1**s in the byte **b**. This is done with a table lookup in the array **c**, declared in **Word.java**, but used in **Table.java**. The variable **b** ranges from **-128** to **127** inclusive, so it is necessary to access **c[b+128]** and to create **c** to give the correct answers when used in this way. The array of Strings **s** (inside **Word.java**) gives the bit representation of each value of **b**, but this was only used for debugging.

Java class: Word

```
// Word.java: an array of CWS (codeword size) bytes
import java.util.Random;
public class Word {
   public static int[] c = {
    // number of 1 bits in 2s complement value (use value+128)
    // used in class Table
    1,2,2,3,2,3,3,4, 2,3,3,4,3,4,4,5, 2,3,3,4,3,4,4,5, 3,4,4,5,4,5,5,6,
    2,3,3,4,3,4,4,5, 3,4,4,5,4,5,5,6, 3,4,4,5,4,5,5,6, 4,5,5,6,5,6,6,7,
    2,3,3,4,3,4,4,5, 3,4,4,5,4,5,5,6, 3,4,4,5,4,5,5,6, 4,5,5,6,5,6,6,7,
    3,4,4,5,4,5,5,6, 4,5,5,6,5,6,6,7, 4,5,5,6,5,6,6,7, 5,6,6,7,6,7,7,8,
```

```
     0,1,1,2,1,2,2,3, 1,2,2,3,2,3,3,4, 1,2,2,3,2,3,3,4, 2,3,3,4,3,4,4,5,
     1,2,2,3,2,3,3,4, 2,3,3,4,3,4,4,5, 2,3,3,4,3,4,4,5, 3,4,4,5,4,5,5,6,
     1,2,2,3,2,3,3,4, 2,3,3,4,3,4,4,5, 2,3,3,4,3,4,4,5, 3,4,4,5,4,5,5,6,
     2,3,3,4,3,4,4,5, 3,4,4,5,4,5,5,6, 3,4,4,5,4,5,5,6, 4,5,5,6,5,6,6,7};

   public byte[] w; // the only data field in this class

   // Word: construct and fill bytes with random values
   public Word(Random ranNumGen) {
      w = new byte[Shannon.CWS]; // allocate CWS bytes
      for (int j = 0; j < Shannon.CWS; j++)
         w[j] = (byte)(256*ranNumGen.nextDouble() - 128);
   }

   // Word: construct and copy input Word u into new class
   public Word(Random ranNumGen, Word u) {
      w = new byte[Shannon.CWS];
      for (int j = 0; j < Shannon.CWS; j++)
         w[j] = u.w[j];
   }
}
```

Java class: Table

```
//Table.java: the code table for Shannon's random code
import java.util.Random;
public class Table {
   public Word[] t; // the only data field in this class

   // Table: constructor.  Allocate expN = 2**N random words
   public Table(Random ranNumGen) {
      t = new Word[Shannon.expN];
      for (int i = 0; i < Shannon.expN; i++)
         t[i] = new Word(ranNumGen);
   }

   // search: search Table t for an input word w
   public int search (Word w) {
      int comp;
      int minComp = Shannon.CWS*8 + 1;
      int minCompCount = -100000000;
      int index = -200000000;
      for (int i = 0; i < Shannon.expN; i++) {
         comp = compare(t[i], w); // count bits that differ
         if (comp == minComp) // an old minimum
            minCompCount++;
         if (comp < minComp) { // a new minimum
            index = i;
            minComp = comp;
            minCompCount = 1;
         }
      }
      if (minCompCount == 1)  return index; // unique minimum
      else return -minCompCount; // several different minimums
```

```
    }

    // compare: return count of differences of bits of input words
    private int compare(Word u, Word v) {
       int diffs = 0;
       for (int i = 0; i < Shannon.CWS; i++)
         diffs += countDiffs(u.w[i], v.w[i]);
       return diffs;
    }

    // countDiffs: return count of differences of bits of input bytes
    private int countDiffs(byte b1, byte b2) {
       byte b = (byte)(b1^b2); // xor gives 1 where bytes differ
       return Word.c[b+128]; // table lookup gives # of 1 bits
    }

    // getWord: fetch a word at a given index: part of simulation
    public Word getWord(int index) {
       return t[index];
    }

    // printTable: print the whole table, debug only
    public void printTable() {
       for (int i = 0; i < Shannon.expN; i++) {
          System.out.print("Entry " + i + ": ");
          t[i].printWord();
       }
    }
}
```

Java class: Shannon

```
// Shannon.java: a simulation of Shannon's random coding
import java.util.Random; // use fancy rng for reproducability
public class Shannon {
   public static final double P = 0.75; // prob of no error
   public static int N; // blocksize, from command line
   public static int expN; // = 2**N, table size, calculated from N
   public static final double C = capacity(P);
   public static int CWS; // the codeword size, bytes, from cmd line
   private static Random ranNumGen = new Random(); // diff each time

   public static double log2(double d) { // for log2 in Java
      return Math.log(d)/Math.log(2.0);
   }

   public static double capacity(double p) { // channel capacity
      if (p == 0 || p == 1) return 1;
      return 1 + p*log2(p) + (1 - p)*log2(1 - p);
   }

   public static int randInt(int i) { // rand int, between 0 and i-1
      return (int)(ranNumGen.nextDouble()*i);
   }
```

```
   // perturb: alter bits of input word, each time with prob 1-P
   public static Word perturb(Word v) {
      Word u = new Word(ranNumGen, v);
      int[] mask = {1, 2, 4, 8, 16, 32, 64, -128};
      for (int i = 0; i < Shannon.CWS; i++)
         for (int j = 0; j < 8; j++)
            if (ranNumGen.nextDouble() > Shannon.P) {
               u.w[i] = (byte)(mask[j]^u.w[i]);
            }
      return u;
   }

   public static void main(String[] args) {
      int simSize = Integer.parseInt(args[0]); // # of trials
      N           = Integer.parseInt(args[1]); // block size
      CWS          = Integer.parseInt(args[2]); // codeword size
      expN = 1;
      for (int i = 0; i < N; i++)
         expN = expN*2;  // expN = 2**N, table size in Table.java
      System.out.println("simSize: " + simSize +
                         ", Blocksize: " + Shannon.N +
                         ", Codeword size (bytes): " + Shannon.CWS +
                         ", expN: " + Shannon.expN);
      // count matches and two kinds of mismatches
      int numMatch = 0, numNonMatch = 0, numMultiMatch = 0;
      Table tab = new Table(ranNumGen); // the coding table
      for (int k = 0; k < simSize; k++) {
         int ind = randInt(Shannon.expN); // index of rand code word
         Word w = tab.getWord(ind); // w is the random code word
         Word u = perturb(w); // u is w with random noise added
         int ind2 = tab.search(u); // closest match, perturbed code word
         if (ind2 == ind) numMatch++;
         else if (ind2 >= 0) { // matched wrong code word, not one sent
            numNonMatch++;
         }
         else if (ind2 < 0) numMultiMatch++; // multiple matches
         if (k%500 == 499) {
             System.out.print("Error Rate: " +
               (k+1 - numMatch)/(double)(k+1));
             System.out.println(", Match: " + numMatch +
            ", Non-Match: " + numNonMatch +
            ", Multiples: " + numMultiMatch);
         }
      } // for
      System.out.print("Error Rate: " +
            (simSize - numMatch)/(double)simSize);
      System.out.println(", Match: " + numMatch +
            ", Non-Match: " + numNonMatch +
            ", Multiples: " + numMultiMatch);
   }
}
```

Program II.5.a
The Huffman Algorithm

Referred to from page 41.

Here is a Huffman code program in 6 files, coded in Java. The program is for demonstration purposes, and needs additional code to perform practical file compression, as is detailed at the end of this section. The program below either reads a file directly from standard input, or if the file name is on the command line, it uses that for the input. The program analyzes the input file to get the symbol frequencies, and then calculates the code words for each symbol. It also creates the output coded file. However, this file is a string of **`0`** and **`1`** *ascii characters*, not binary numbers. The code also produces a human-readable version of the Huffman decoding tree, as well as the entropy of the file and the average code length of the resulting Huffman code.

The encode algorithm (function **`encode`** inside **`Huffman.java`**) just uses sequential search, although the corresponding decode algorithm makes efficient use of the Huffman tree. The priority queue (implemented in the file **`PQueue.java`**) just uses a simple list and sequential search, whereas a good priority queue should be implemented with a heap.

Java class: Entry

```
// Entry.java: entry in the code frequency table
class Entry {
   public char symb; // character to be encoded
   public double weight; // prob of occurrence of the character
   public String rep; // string giving 0-1 Huffman codeword for char
}
```

Java class: Table

```
// Table.java: Huffman code frequency table
import java.io.*;
class Table {
   public final int MAXT = 100; // maximum # of different symbols
   public int currTableSize; // current size as table constructed
   public Entry[] tab; // the table array, not allocated
   private Reader in; // internal file name for input stream
   String file = ""; // the whole input file as a String
   private boolean fileOpen = false; // is the file open yet?
   private String fileName; // name of input file, if present
   private int totalChars = 0; // total number of chars read
   char markerChar = '@'; // sentinal at end of file

   // Table: constructor, input parameter: input file name or null
   public Table(String f) {
      fileName = f;
      currTableSize = 0;
      tab = new Entry[MAXT];
   }

   // getNextChar: fetches next char.  Also opens input file
```

```
private char getNextChar() {
   char ch = ' '; // = ' ' to keep compiler happy
   if (!fileOpen) {
      fileOpen = true;
      if (fileName == null)
         in = new InputStreamReader(System.in);
      else {
         try {
            in = new FileReader(fileName);
         } catch (IOException e) {
            System.out.println("Exception opening " + fileName);
         }
      }
   }
   try {
      ch = (char)in.read();
   } catch (IOException e) {
      System.out.println("Exception reading character");
   }
   return ch;
}

// buildTable: fetch each character and build the Table
public void buildTable() {
   char ch = getNextChar();
   while (ch != 65535 && ch != markerChar) { // EOF or sentinal #
      totalChars++;
      file += ch;
      int i = lookUp(ch);
      if (i == -1) { // new entry
         tab[currTableSize] = new Entry();
         tab[currTableSize].symb = ch;
         tab[currTableSize].weight = 1.0;
         tab[currTableSize].rep = "";
         currTableSize++;
      }
      else { // existing entry
         tab[i].weight += 1.0;
      }
      // System.out.print(ch); // for debug
      ch = getNextChar();
   } // while
   // finish calculating the weights
   for (int j = 0; j < currTableSize; j++)
      tab[j].weight /= (double)totalChars;
}

// lookUp: loop up the next char in the Table tab
public int lookUp(char ch) {
   for (int j = 0; j < currTableSize; j++)
      if (tab[j].symb == ch) return j;
   return -1;
}
```

```
   // log2: Logarithm base 2
   public double log2(double d) {
      return Math.log(d) / Math.log(2.0);
   }

   // entropy: calculate entropy of the Table
   public double entropy() {
      double res = 0.0;
      for (int i = 0; i < currTableSize; i++)
         res += tab[i].weight * log2(1.0/tab[i].weight);
      return res;
   }

   // aveCodeLen: calculate average code length
   public double aveCodeLen() {
      double res = 0.0;
      for (int i = 0; i < currTableSize; i++)
         res += tab[i].weight * tab[i].rep.length();
      return res;
   }
}
```

Java class: TreeNode

```
// TreeNode.java: node in the Huffman tree, used for encode/decode
class TreeNode {
   public double weight; // probability of symb occurring
   public char symb; // the symbol to be encoded
   public String rep; // string of 0's and 1's, huffman code word
   public TreeNode left, right; // tree pointeres
   public int step; // step # in construction (for displaying tree)
}
```

Java class: ListNode

```
// ListNode.java: node in linked list of trees, initially root nodes
class ListNode {
   public TreeNode hufftree;
   public ListNode next;
}
```

Java class: PQueue

```
// PQueue.java: implement a priority queue as a linked list of trees
//   Initialize it as a linked list of singleton trees
class PQueue {
   ListNode list = null; // this points to the main list

   // insert: insert new entry into the list
   public void insert(TreeNode t) {
      ListNode l = new ListNode();
      l.hufftree = t;
      l.next = list;
      list = l;
   }

   // buildList: create the initial list with singleton trees
   public void buildList(Entry[] tab, int n) {
```

```
      int i;
      TreeNode tNode;
      for (i = 0; i < n; i++) {
         tNode = new TreeNode();
         tNode.weight = tab[i].weight;
         tNode.left = tNode.right = null;
         tNode.symb = tab[i].symb;
         tNode.rep = "";
         insert(tNode);
      }
   }

   // least: Remove and return from the list that tree with greatest
   //    root weight; sort of a pain in the ass to write
   public TreeNode least() {
      ListNode l, oldl, minl = null, oldminl = null; // for compiler
      double minw = 1000000;
      oldl = list;
      l = list;
      while (l != null) {
         if (l.hufftree.weight < minw) {
            minw = l.hufftree.weight;
            oldminl = oldl;
            minl = l;
         }
         oldl = l;
         l = l.next;
      }
      if (minl == oldminl) {
            list = list.next;
            return minl.hufftree;
      }
      oldminl.next = minl.next;
      return minl.hufftree;
   }
}
```

Java class: Huffman

```
// Huffman.java: the Huffman tree algorithm
import java.text.DecimalFormat;
class Huffman {
   public TreeNode tree; // the decoding tree
   public Table t; // the frequency and encoding table
   public PQueue p; // priority queue for building the Huffman tree
   private int depth; // depth variable for debug printing of tree
   String encodedFile, decodedFile; // files as Strings
   char markerChar = '@'; // sentinal at end of file
   public DecimalFormat fourDigits = new DecimalFormat("0.0000");

   // Huffman: constructor, does all the work
   public Huffman(String fileName) {
      t = new Table(fileName);
      t.buildTable();
```

```
      p = new PQueue();
      p.buildList(t.tab, t.currTableSize);
      tree = huffman(t.currTableSize);
      insertRep(tree, t.tab, t.currTableSize, "");
      displayTree(tree);
      t.dumpTable();
      encodedFile = encode(t.file);
      System.out.println("Entropy: " + t.entropy() +
         ", Ave. Code Length: " + t.aveCodeLen());
   }

   // encode: translate the input file to binary Huffman file
   public String encode(String file) {
      String returnFile = ""; // encoded file to return (as a String)
      for (int i = 0; i < file.length(); i++) {
         int loc = t.lookUp(file.charAt(i));
         if (loc == -1) {
            System.out.println("Error in encode: can't find: " +
               file.charAt(i));
            System.exit(0);
         }
         returnFile += t.tab[loc].rep;
      }
      return returnFile;
   }

   // decode: translate the binary file (as a string) back to chars
   public String decode(String file) {
      String returnFile = ""; // decoded file to return (as a String)
      TreeNode treeRef; // local tree variable for chasing into tree
      int i = 0; // index in the Huffman String
      while (i < file.length()) { // keep going to end of String
         treeRef = tree; // start at root of tree
         while (true) {
            if (treeRef.symb != markerChar) { // at a leaf node
               returnFile += treeRef.symb;
               break;
            }
            else if (file.charAt(i) == '0') { // go left with '0'
               treeRef = treeRef.left;
               i++;
            }
            else  { // go right with '1'
               treeRef = treeRef.right;
               i++;
            }
         } // while (true)
      } // while
      return returnFile;
   }

   // huffman: construct the Huffman tree, for decoding
   public TreeNode huffman(int n) {
```

```
   int i;
   TreeNode tree = null; // = null for compiler
   for (i = 0; i < n-1; i++) {
      tree = new TreeNode();
      tree.left = p.least();
      tree.left.step = i + 1; // just for displaying tree
      tree.right = p.least();
      tree.right.step = i + 1; // just for displaying tree
      tree.weight = tree.left.weight +
         tree.right.weight;
      tree.symb = markerChar; // must not use '@' in input file
      tree.rep = "";
      p.insert(tree);
   }
   return tree;
}

// displayTree: print out tree, with initial and final comments
public void displayTree(TreeNode tree) {
   System.out.println("\nDisplay of Huffman coding tree\n");
   depth = 0;
   displayTreeRecurs(tree);
}

// displayTreeRecurs: need recursive function for inorder traveral
public void displayTreeRecurs(TreeNode tree) {
   depth++; // depth of recursion
   String s = "";
   if (tree != null) {
      s = display(tree.rep + "0");
      System.out.println(s);
      displayTreeRecurs(tree.left);
      s = display(tree.rep);
      System.out.print(s + "+---");
      if (depth != 1) {
         if (tree.symb == markerChar) System.out.print("+---");
      }
      System.out.print(tree.symb + ": " +
         fourDigits.format(tree.weight) + ", " + tree.rep);
      if (depth != 1)
         System.out.println(" (step " + tree.step + ")");
      else System.out.println();
      displayTreeRecurs(tree.right);
      s = display(tree.rep + "1");
      System.out.println(s);
   }
   depth--;
}

// display: output blanks and verical lines to display tree
//    (tricky use of rep string to display correctly)
private String display(String rep) {
   String s = "   ";
```

```
      for (int i = 0; i < rep.length() - 1; i++) { // initial chars
         if (rep.charAt(i) != rep.charAt(i+1) ) s += "|";
         else s += " ";
         s += "   ";
      }
      return s;
   }

   // insertRep: tricky function to use Huffman tree to create rep
   public void insertRep(TreeNode tree, Entry tab[], int n, String repr) {
   //recursive function to insert Huffman codewords at each node.
   //  this could just insert at the leaves.
      String s1, s2;
      tree.rep = repr;
      if ((tree.left) == null && (tree.right) == null) {
         for (int i = 0; i < n; i++)
            if (tree.symb == tab[i].symb)
               tab[i].rep = tree.rep;
         return;
      }
      s1 = repr; s1 += "0";
      insertRep(tree.left, tab, n, s1); // recursive call to left
      s2 = repr; s2 += "1";
      insertRep(tree.right, tab, n, s2); // recursive call to right
   }

   // main: doesn't do much; just feeds in input file name
   public static void main(String[] args) {
      Huffman huff;
      // pass an input file name if present on command line
      if (args.length > 0) huff = new Huffman(args[0]);
      else huff = new Huffman(null);
   }
}
```

Here is an input file:

```
% cat Testit.txt
aaaaabbbbbccccccccccccccddddddddddddddddddeeeeeeeeeeeeeeeeee
ffffffffffffffffffffffffffffffffffff#
```

Here is the output with various debug dumps:

```
% java Huffman Testit.txt
Display of Huffman coding tree
   +---f: 0.4000, 0 (step 5)
   |
   +---@: 1.0000,
   |
   |           +---b: 0.0500, 1000 (step 1)
   |           |
   |       +---+---@: 0.1000, 100 (step 2)
   |       |   |
   |       |   +---a: 0.0500, 1001 (step 1)
```

```
    |       |
    |   +---+---@: 0.2500, 10 (step 4)
    |   |   |
    |   |   +---c: 0.1500, 101 (step 2)
    |   |
    |   +---+---@: 0.6000, 1 (step 5)
    |       |
    |       |   +---d: 0.1700, 110 (step 3)
    |       |   |
    |       +---+---@: 0.3500, 11 (step 4)
    |           |
    |           +---e: 0.1800, 111 (step 3)

Dump of Table ----->
  Size: 6
Entry 0. Symbol: a, Weight: 0.05, Representation: 1001
Entry 1. Symbol: b, Weight: 0.05, Representation: 1000
Entry 2. Symbol: c, Weight: 0.15, Representation: 101
Entry 3. Symbol: d, Weight: 0.17, Representation: 110
Entry 4. Symbol: e, Weight: 0.18, Representation: 111
Entry 5. Symbol: f, Weight: 0.4, Representation: 0
----> End Dump of Table

Entropy: 2.251403369717592, Ave. Code Length: 2.3

Input file (as a String):
aaaaabbbbbccccccccccccccddddddddddddddddeeeeeeeeeeeeeeeeee
ffffffffffffffffffffffffffffffffffffff

Encoded file (as a String):
10011001100110011001100010001000100010001011011011011011011
11011011011011011011011011101101101101101101101101101101101
01101101101101101111111111111111111111111111111111111111111
11111111110000000000000000000000000000000000000000

Decoded file (as a String):
aaaaabbbbbccccccccccccccddddddddddddddddeeeeeeeeeeeeeeeeee
ffffffffffffffffffffffffffffffffffffff
```

A Program to Produce Actual Compressed Binary Huffman Files: The changes and additions involve the methods **`encode`** and **`decode`** in the class **`Huffman`**. First, the input file would be read twice, once to build the frequency table, and a second time to translate to the Huffman code. Next, the method needs to write the frequency table in some form. (It is also possible to write the actual Huffman tree in coded form.) Then **`encode`** would write *bits* in a way similar to the *Hamming* algorithm in the next section.

The **`decode`** method would first read the frequency table and build an the same Huffman tree used inside **`encode`**. Then the method would read bits, again in a way similar to that of the *Hamming* algorithm. The final translation to the original symbols would be the same.

Program II.5.b
Two Distinct Huffman Codes

Referred to from page 43.

There are simple examples of Huffman codes, where there are two structurally distinct Huffman trees for the same set of symbols and frequencies. Because the Huffman code is optimal, these codes must have the same average code length. Here is one simple example: **a:9**, **b:5**, **c:4**, **d:3**, and **e:3**. There are clearly two distinct ways to construct the tree, resulting in Huffman codes for the symbols with a different set of lengths for the codewords. In both cases the average code length is **2.25** bits per symbol.
Here are two different sets of lengths:

```
        +---a: 0.3750, 0 (step 4)
        |
    ---+---@: 1.0000,
        |
        |       +---e: 0.1250, 100 (step 1)
        |       |
        |   +---+---@: 0.2500, 10 (step 3)
        |   |   |
        |   |   +---d: 0.1250, 101 (step 1)
        |   |
        |   +---+---@: 0.6250, 1 (step 4)
        |       |
        |       |   +---c: 0.1667, 110 (step 2)
        |       |   |
        |       +---+---@: 0.3750, 11 (step 3)
        |           |
        |           +---b: 0.2083, 111 (step 2)

    Entry 0. Symbol: a, Weight: 0.3750, Representation: 0
    Entry 1. Symbol: d, Weight: 0.1250, Representation: 101
    Entry 2. Symbol: b, Weight: 0.2083, Representation: 111
    Entry 3. Symbol: e, Weight: 0.1250, Representation: 100
    Entry 4. Symbol: c, Weight: 0.1667, Representation: 110

    Entropy: 2.1829, Ave. Code Length: 2.25

           +---c: 0.1667, 00 (step 2)
           |
        +---+---@: 0.3750, 0 (step 4)
        |   |
        |   +---b: 0.2083, 01 (step 2)
        |
    ---+---@: 1.0000,
        |
        |       +---e: 0.1250, 100 (step 1)
        |       |
        |   +---+---@: 0.2500, 10 (step 3)
```

```
   |   |   |
   |   |   +---d: 0.1250, 101 (step 1)
   |   |
   +---+---@: 0.6250, 1 (step 4)
       |
       +---a: 0.3750, 11 (step 3)

Entry 0. Symbol: a, Weight: 0.3750, Representation: 11
Entry 1. Symbol: b, Weight: 0.2083, Representation: 01
Entry 2. Symbol: c, Weight: 0.1667, Representation: 00
Entry 3. Symbol: d, Weight: 0.1250, Representation: 101
Entry 4. Symbol: e, Weight: 0.1250, Representation: 100

Entropy: 2.1829, Ave. Code Length: 2.25
```

Program II.6.a
The Hamming Algorithm

Referred to from page 47.

This section presents an implementation of the binary Hamming code. The Java source was designed for simplicity and ease of understanding, rather than for efficiency. The basic Hamming algorithm is implemented using arrays of "bits" (0 or 1 stored in an **`int`**), with from 1 to 120 message bits and from 4 to 128 bits in the codeword. In order to read and write files of *bytes*, it is necessary to unpack each byte so that the Hamming routines can work on it, and then to pack the result for writing. (A more efficient implementation would not use arrays of bits in this way and so would not need the packing and unpacking.)

The complete array-based implementation of the Hamming code is in the Java class **`Hamming`**. This is a straightforward implementation. Here are comments about individual features:

- *The constructor* **`Hamming`***:* this just builds the necessary masks, described next.
- *The 2-dimensional array* **`m`***:* nine 128-bit masks used to decide which bits to use in each parity check: **`m[0]`**, **`m[1]`**, **`m[2]`**, **`m[3]`**, **`m[4]`**, **`m[5]`**, **`m[6]`**, **`m[7]`**. The mask **`m[i]`** gives the bits to check for the check bit in position **`pow(2,i)`**. Thus **`m[0]`** is all 1 bits, **`m[1]`** has odd-numbered bits 1, **`m[2]`** has alternating pairs of 0's and 1's, and so forth. The mask **`m0`** has a 1 bit in those positions used for check bits: all powers of 2.
- *The* **`encodeMessage`** *method:* Takes an input array of message bits (where the length of the array gives the number of bits), and produces an output array of bits that represents the Hamming codeword obtained by inserting extra check bits. Again the length of the codeword is given by the length of the array.
- *The* **`insertMessage`** *method:* Called by **`encodeMessage`**, this inserts message bits into each non-check bit postion.
- *The* **`insertCheckBits`** *method:* Called by **`encodeMessage`**, this inserts the proper check bit values.
- *The* **`decodeMessage`** *method:* This first checks for errors (**`checkErrors`**), then corrects a single error if one is found, and finally extracts the message without check bits and returns it. In case of a detected double error, a **`null`** is returned.
- *Expanding the implementation size:* This is easy to do. For example, to double the maximum size, change **`MAX_CHK_LEN`** from 8 to 9, **`MAX_RES_LEN`** from 128 to 256, and add 128 to end of list defining **`checkPos`**. (The class **`HammingDecode`** accesses the instatiation of **`Hamming`** and builds another array based on these sizes.)
- *Making use of the class:* The following code show how to use the class **`Hamming`**, as it is used in the classes **`HammingEncode`** and **`HammingDecode`** below:

Using class Hamming

```
Hamming ham = new Hamming();
int[] mess, res;
// create mess and fill it with one block's worth of message bits
res = ham.encodeMessage(mess); // res allocated inside ham
// create res and fill it with one block's worth of message and check bits
mess = ham.decodeMessage(res); // mess allocated inside ham
```

Here is Java code for the class **`Hamming`**, with a few extra debug lines reporting the number of errors corrected.

Java class: Hamming

```
// Hamming: implement Hamming code
//   Uses arrays of "bits": mess = uncoded input, res = coded result
public class Hamming {
   public final int MAX_CHK_LEN = 8; // max number of check digits
   public final int MAX_RES_LEN = 128; // 2^(MAX_CHK_LEN - 1)
   public int[] checkPos = {0,1,2,4,8,16,32,64}; // positions to check
   public final int MAX_MESS_LEN = MAX_RES_LEN - MAX_CHK_LEN; // 120
   private int[][] m = new int[MAX_CHK_LEN][MAX_RES_LEN]; // check masks
   private int[] m0 = new int[MAX_RES_LEN]; // mask for message insertion
   private int[] buf = new int[MAX_RES_LEN]; // buffer for coded messages
   public int errCount; // ****** extra counter for debugging ******

   // Hamming: constructor to create masks
   public Hamming() {
      for (int i = 0; i < MAX_CHK_LEN; i++)
         for (int j = 0; j < MAX_RES_LEN; j++)
            if (i == 0) m[i][j] = 1;
            else m[i][j] = (j >> (i - 1))%2;
      for (int i = 0; i < MAX_RES_LEN; i++) m0[i] = 1;
      for (int i = 0; i < MAX_CHK_LEN; i++) m0[checkPos[i]] = 0;

   }

   // encodeMessge: insert message bits and then set check bits
   public int[] encodeMessage(int[] mes) {
      int res[] = insertMessage(mes);
      insertCheckBits(res);
      return res;
   }

   // insertMessage: put message bits into non-check positions
   public int[] insertMessage(int[] mess) {
      for (int i = 0; i < MAX_RES_LEN; i++) buf[i] = 0;
      int loc = 0, i = 0;
      while (i < mess.length) {
         if (m0[loc] == 1) buf[loc] = mess[i++];
         if (loc >= MAX_RES_LEN) System.exit(1000 + i);
         loc++;
      }
      int[] res = new int[loc];
```

```
   for (int j = 0; j < loc; j++) res[j] = buf[j];
   return res;
}

// insertCheckBits: add the parity check bits
public void insertCheckBits(int[] res) {
   for (int i = MAX_CHK_LEN - 1; i >= 0; i--) {
      int checkRes = 0; // holds sum of bits for this parity check
      for (int j = 0; j < res.length; j++)
         if (m[i][j] == 1) checkRes += res[j];
      if (checkPos[i] < res.length)
         res[checkPos[i]] = checkRes%2;
   }
}

// decodeMessage: correct errors and extract message bits
public int[] decodeMessage(int[] res) {
   int errCode = checkErrors(res);
   if (errCode >= 0) { // single error in position errCode
      res[errCode] ^= 1; // correct single error
      errCount++; // ******* extra count for debugging ********
   }
   if (errCode >= -1) return extractMessage(res); // no errors left
   return null; // errCode == -2 means DOUBLE ERROR
}

// extractMessage: get back message bits from non-check positions
public int[] extractMessage(int[] res) {
   for (int i = 0; i < MAX_RES_LEN; i++) buf[i] = 0;
   int loc = 0, i = 0;
   while (i < res.length) {
      if (m0[i] == 1) buf[loc++] = res[i];
      if (loc >= MAX_RES_LEN) System.exit(2000 + i);
      i++;
   }
   int[] mess = new int[loc];
   for (int j = 0; j < loc; j++) mess[j] = buf[j];
   return mess;
}

// checkErrors: do error check, return position of error
//   return -1 for no error, return -2 for double error
public int checkErrors(int[] res) {
   int[] checkRes = new int[MAX_CHK_LEN];
   int errorPos = 0;
   for (int i = 0; i < MAX_CHK_LEN; i++) {
      checkRes[i] = 0;
      for (int j = 0; j < res.length; j++)
         if (m[i][j] == 1) checkRes[i] += res[j];
      checkRes[i] %= 2;
   }
   for (int i = 1; i < MAX_CHK_LEN; i++)
      if (checkRes[i] == 1) errorPos += checkPos[i];
```

```
      if (errorPos == 0 && checkRes[0] == 0) return -1; // no error
      if (errorPos == 0 && checkRes[0] == 1) return 0; // error at 0
      if (errorPos > 0  && checkRes[0] == 1) return errorPos; // error
      if (errorPos > 0  && checkRes[0] == 0) return -2; // double error
      return 999;
   }
}
```

The final three classes in this section implement the encoding and decoding of an arbitrary binary file using the Hamming code. The method here encodes anywhere from 1 bit to 120 bits at a time. This requires 4 to 128 bits in the Hamming codeword. The test below shows the results of encoding a 3116-byte binary PDF file into files of varying sizes depending on the message length. The coded file is then decoded to get the original back.

The coded file starts with a byte giving the number of message bits in each codeword. This file does not usually have a number of bits divisible by 8, so the last byte of the file indicates how many bits of the next-to-the-last byte are part of the coded result. As part of the debugging, single errors were simulated at some random bit position in each codeword before the decoding step.

The class **`HammingEncode`** reads bytes of a source file (using **`encodeHammingBit`**). Each bit of each byte is sent of another function (**`encodeBit`**), which accumulates them until there is a block the size of the desired message size. This block is transformed to a coded Hamming block when an instance of **`Hamming`** adds check bits. Then the resulting block is sent one bit at a time to a function **`writeBit`** which accumulates them until it has 8 to write as a byte.

The class **`HammingDecode`** reads bytes of a source file (using **`decodeHammingBit`**). The first byte of the source file gives the message size for the particular Hamming code used in the file. The last byte of the file gives the number of bits used in the next-to-the-last byte, so it necessary to read three bytes ahead during processing. After the first byte, each bit of each byte is sent of another function (**`decodeBit`**), which accumulates them until there is a block the size of the desired codeword size. This block is transformed to a message block when an instance of **`Hamming`** removes check bits. Then the resulting block is sent one bit at a time to a function **`writeBit`** (different from the previous **`writeBit`**) which accumulates them until it has 8 to write as a byte.

Java class: HammingEncode

```
// HammingEncode: encode an input file with the Hamming code
import java.io.*;
public class HammingEncode {
   Hamming ham = new Hamming(); // Hamming code implementation
   InputStream in; // input file
   OutputStream out; // output file
   int currPos = 0, currPosRest = 0; // keep track of bit positions
   int bRest = 0; // last byte
   int messLen; // bit length of original message
   int[] mess, messRest, res; // message, remaining message, code result
   int[] bMask = {0x1, 0x2, 0x4, 0x8, 0x10, 0x20, 0x40, 0x80}; // bit masks
```

```
public HammingEncode(int messL, InputStream infile, OutputStream outfile) {
   messLen = messL;
   in =  infile;
   out = outfile;
   mess = new int[messLen];
}

// encodeHammingBit: read bytes, pass bits to encodeBit.
//   Bits of a byte numbered from 0 as least significant
public void encodeHammingBit() {
   writeByte(messLen); // write message length
   try {
      int b; // input byte
      int bit; // output bit
      while ((b = in.read()) != -1) {
         for (int dum = 0; dum < 8; dum++) {
            bit = b%2;
            encodeBit(bit);
            b = b >> 1;
         }
      } // end of while
   } catch (IOException e) {
      System.err.println("Error reading input file");
      System.exit(-1);
   } // end try
   encodeBit(-1);
}

// encodeBit: pass bit at a time to growing Hamming array
private void encodeBit(int bit) {
   if (bit == -1) { // no more bits; finish up
      if (currPos == 0) {
         writeBit(-1); // send finish up message to writeBit
         return;
      }
      messRest = new int[currPos];
      for (int i = 0; i < currPos; i++)
         messRest[i] = mess[i];
      res = ham.encodeMessage(messRest);
      // *** for debugging, insert random error into half of the blocks
      if (Math.random() < 0.5) { // insert half the time
         int errPos = (int)(Math.random()*res.length); // random positioon
         res[errPos] ^= 1; // insert the actual bit error
      }
      for (int i = 0; i < res.length; i++)
         writeBit(res[i]);
      writeBit(-1); // send finish up message to writeBit
      return;
   }
   mess[currPos++] = bit;
   if (currPos == messLen) {
      res = ham.encodeMessage(mess);
      // *** for debugging, insert random error into half of the blocks
```

```
            if (Math.random() < 0.5) { // insert half the time
               int errPos = (int)(Math.random()*res.length); // random position
               res[errPos] ^= 1;
            }
            for (int i = 0; i < res.length; i++)
               writeBit(res[i]);
            currPos = 0; // reset position for next block
         }
      }

      // writeBit: accumulate 8 bits, then write the byte
      private void writeBit(int bit) {
         if (bit == -1) { // received finish up message
            writeByte(bRest); // write last partial byte
            writeByte(currPosRest); // how many bits count in last byte?
            return;
         }
         if (bit == 1) bRest |= bMask[currPosRest]; // insert bit
         currPosRest++;
         if (currPosRest == 8) {
            writeByte(bRest);
            currPosRest = 0;
            bRest = 0;
         }
      }

      // writeByte: accumulate bits, then write byte
      private void writeByte(int b) {
         try {
            out.write(b);
         } catch (IOException e) {
            System.err.print("Error writing file");
            System.exit(-1);
         }
      }
   }
```

Java class: HammingDecode

```
// HammingDecode: decode an input file coded with the Hamming code
import java.io.*;
public class HammingDecode {
   Hamming ham = new Hamming(); // Hamming code implementation
   InputStream in; // input file
   OutputStream out; // output file
   int currPos = 0, currPosRest = 0; // bit positions in bytes
   int bRest = 0; // last byte
   int resLen, messLen; // bit length of: coded result, original message
   int[] mess, res, resRest; // message, coded result, remaining result
   int[] bMask = {0x1, 0x2, 0x4, 0x8, 0x10, 0x20, 0x40, 0x80}; // bit mask
   int[] codeLen; // codeLen[i] = j means if messLen == i, then resLen == j

   // HammingDecode: assign files and build codeLen table
   public HammingDecode(InputStream infile, OutputStream outfile) {
```

```
      codeLen = new int[ham.MAX_MESS_LEN + 1];
      codeLen[0] = 0;
      int next = 4, j = 3;
      for (int i = 1; i <= ham.MAX_MESS_LEN; i++) {
         codeLen[i] = next;
         if (j < ham.checkPos.length && next == ham.checkPos[j]) {
            j++; next++;
         }
         next++;
      }
      in =  infile;
      out = outfile;
   }

   // decodeHammingBit: read bytes, pass bits to instance of Hamming
   //   Bits of a byte numbered from 0 as least significant
   public void decodeHammingBit() {
      try {
         int b;     // input byte
         int bNext; // next byte (read ahead)
         int bEnd;  // next byte after bNext
         int bit;   // output bit
         messLen = in.read(); // initial byte holds message length
         resLen = codeLen[messLen]; // deduce coded result length
         res = new int[resLen];
         bNext = in.read(); // read ahead because last byte contains
         bEnd = in.read(); // the number of bits in next-to-last byte
         while (true) {
            b = bNext;
            bNext = bEnd;
            bEnd = in.read();
            if (bEnd == -1) { // end-of-file
               // bNext give # of bits of b to use
               for (int dum = 0; dum < bNext; dum++) {
                  bit = b%2;
                  decodeBit(bit);
                  b = b >> 1;
               }
               decodeBit(-1); // send end-of-file message
               return;
            }
            for (int dum = 0; dum < 8; dum++) {
               bit = b%2;
               decodeBit(bit);
               b = b >> 1;
            }
         } // end of while
      } catch (IOException e) {
         System.err.println("Error reading input file");
         System.exit(-1);
      } // end try
   }
```

```
// decodeBit: decode and send off bits
private void decodeBit(int bit) {
   if (bit == -1) {  // no more bits; finish up
      if (currPos == 0) { // ******** temp debug output ********
         System.out.println(ham.errCount + " errors detected");
         return; // no leftovers
      }
      resRest = new int[currPos];
      for (int i = 0; i < currPos; i++)
         resRest[i] = res[i];
      mess = ham.decodeMessage(resRest);
      if (mess == null) { // double error
         System.out.println("Double error detected");
         System.exit(-1);
      }
      for (int i = 0; i < mess.length; i++)
         writeBit(mess[i]);
      writeBit(-1); // end-of-file message
      System.out.println(ham.errCount + " errors detected"); // temp output
      return;
   }
   res[currPos++] = bit;
   if (currPos == resLen) {
      mess = ham.decodeMessage(res);
      if (mess == null) { // double error
         System.out.println("Double error detected");
         System.exit(-1);
      }
      for (int i = 0; i < mess.length; i++)
         writeBit(mess[i]);
      currPos = 0;
   }
}

// writeBit: accumulate bits until ready to write a byte
private void writeBit(int bit) {
   if (bit == -1) { // no more bits
      if (currPosRest == 0) return;
      writeByte(bRest);
      return;
   }
   if (bit == 1) bRest |= bMask[currPosRest]; // insert bit
   currPosRest++;
   if (currPosRest == 8) {
      writeByte(bRest);
      currPosRest = 0;
      bRest = 0;
   }
}

// writeByte: actually write the byte
private void writeByte(int b) {
   try {
```

```
            out.write(b);
         } catch (IOException e) {
            System.err.print("Error writing file");
            System.exit(-1);
         }
      }
   }
```

Java class: HammingFiles

```
// HammingFiles: encode or decode files with Hamming code
import java.io.*;
public class HammingFiles {
   static InputStream in; // input file args[1] or args[2]
   static OutputStream out; // output file args[2] or args[3]

   public static void openFiles(String infile, String outfile) {
      try {
         in =  new FileInputStream(infile);
         out = new FileOutputStream(outfile);
      } catch (IOException e) {
         System.err.println("Error opening files");
         System.exit(-1);
      }
   }

   public static void main(String[] args) {
      if (args[0].equals("-encode")) {
         int messLen = Integer.parseInt(args[1]);
         if (messLen > 120) {
            System.err.println("Error: Message length > 120");
            System.exit(-1);
         }
         openFiles(args[2], args[3]);
         HammingEncode hamEncode = new HammingEncode(messLen, in, out);
         hamEncode.encodeHammingBit();
      }
      else if (args[0].equals("-decode")) {
        openFiles(args[1], args[2]);
        HammingDecode hamDecode = new HammingDecode(in, out);
        hamDecode.decodeHammingBit();
      }
      else System.err.println("Usage: java HammingBit " +
         "(-encode messageLen | -decode) infile outfile");
   }
}
```

Finally, here is a sample debug run of this software with a 3116-byte input binary file (PDF format): **`utsa.pdf`**. Extra errors are inserted half the time.

```
% java HammingFiles -encode 53 utsa.pdf utsa53.code
% java HammingFiles -decode utsa53.code utsa2_53.pdf
212 errors detected
```

```
% wc utsa53.code    (coded file, using 53 bits per message)
      15      95    3531 utsa53.code
% wc utsa2_120.pdf  (recovered file using Hamming code)
      53     175    3116 utsa2_53.pdf
```

Here are different sizes of the coded file when different message lengths are used for the coding. These are not experimental results, but the sizes are strictly dictated by the properties of the Hamming code. Notice that message lengths 4 and 5 result in exactly the same sizes for the final encoded files.

Hamming-encoded File			
Message size	**Codeword size**	**File size**	**Coded file size**
1	4	3116	12467
2	6	3116	9351
3	7	3116	7273
4	8	3116	6235
5	10	3116	6235
10	15	3116	4677
20	26	3116	4054
30	37	3116	3846
40	47	3116	3664
50	57	3116	3555
100	108	3116	3368
120	128	3116	3327

Program II.7.a
U.S. Banking Scheme

Referred to from page 53.

Here is the simple scheme used by U.S. banks, involving successive weights of 3, 7 and 1, repeated.

Java class: ErrorDetection

```
// ErrorDetection.java: base class for single-digit error detection
public class ErrorDetection {

   public static void printArray(int[] a) {
      for (int i = 0; i < a.length; i++) {
        if (a[i] == 10) System.out.print("X ");
        else System.out.print(a[i]);
        if (i%5 == 0) System.out.print(" ");
      }
      System.out.println();
   }

   public static void printUnchecked(int[] a) {
      System.out.print("? ");
      for (int i = 1; i < a.length; i++) {
        System.out.print(a[i]);
        if (i%5 == 0) System.out.print(" ");
      }
      System.out.println();
   }
}
```

Java class: BanksErrorDetection

```
// BanksErrorDetection.java: Implement scheme used by US banks
public class BanksErrorDetection extends ErrorDetection {

   public static int insertCheck(int[] a) {
      int check = 0;
      for (int i = 1; i < a.length; i++)
         {\timesbf if (i%3 == 1)
            check = (check + 3*a[i])%10;
         else  if (i%3 == 2)
            check = (check + 7*a[i])%10;
         else
            check = (check + a[i])%10;}
      if (check == 0) a[0] = 0;
      else a[0] = -check + 10;
      return a[0];
   }

   public static boolean doCheck(int[] a) {
      int check = 0;
```

```
      for (int i = 0; i < a.length; i++)
         if (i%3 == 1) check = (check + 3*a[i])%10;
         else if (i%3 == 2) check = (check + 7*a[i])%10;
         else check = (check + a[i])%10;
      if (check != 0) return false;
      else return true;
   }

   // main function
   public static void main (String[] args) {
      int[] a = new int[9];
      boolean checkFlag = false;
      for (int i = 1; i < a.length; i++)
         a[i] = (int)(Math.random() * 10.0);
      printUnchecked(a);
      BanksErrorDetection.insertCheck(a);
      printArray(a);
      System.out.println(BanksErrorDetection.doCheck(a));
      a[4] = (a[4] + 1)%10;
      BanksErrorDetection.printArray(a);
      System.out.println(BanksErrorDetection.doCheck(a));

      // test all adjacent transpositions
      System.out.println("\nUS Banks, error detection scheme");
      System.out.println("\nTest all adjacent transpositions ...");
      for (int pos = 4; pos < 7; pos++)
         for (int p1 = 0; p1 < 10; p1++)
            for (int p2 = 0; p2 < 10; p2++) {
               if (p1 != p2) {
                  a[pos] = p1; a[pos+1] = p2;
                  BanksErrorDetection.insertCheck(a);
                  // interchange
                  a[pos] ^= a[pos+1];
                  a[pos+1] ^= a[pos];
                  a[pos] ^= a[pos+1];
                  if (BanksErrorDetection.doCheck(a)) {
                     System.out.println("Warning: Interchange of " +
                        p1 + " and " + p2 + " not detected");
                     checkFlag = true;
                  }
               }
            }
      if (checkFlag)
         System.out.println("At least one transposition undetected");
      else
         System.out.println("All transpositions detected");
   } // end of main
}
```

Here is the output, showing a simple test, and a test of all adjacent interchanges. Here if digits differing by 5 are interchanged, the error goes undetected. I have tested interchanges for each pair of the three weights, so the 10 missed transpositions are repeated 3 times below.

```
? 95967 315
1 95967 315
true
1 95977 315
false

US Banks, error detection scheme

Test all adjacent transpositions ...
Warning: Interchange of 0 and 5 not detected
Warning: Interchange of 1 and 6 not detected
Warning: Interchange of 2 and 7 not detected
Warning: Interchange of 3 and 8 not detected
Warning: Interchange of 4 and 9 not detected
Warning: Interchange of 5 and 0 not detected
Warning: Interchange of 6 and 1 not detected
Warning: Interchange of 7 and 2 not detected
Warning: Interchange of 8 and 3 not detected
Warning: Interchange of 9 and 4 not detected
Warning: Interchange of 0 and 5 not detected
Warning: Interchange of 1 and 6 not detected
Warning: Interchange of 2 and 7 not detected
Warning: Interchange of 3 and 8 not detected
Warning: Interchange of 4 and 9 not detected
Warning: Interchange of 5 and 0 not detected
Warning: Interchange of 6 and 1 not detected
Warning: Interchange of 7 and 2 not detected
Warning: Interchange of 8 and 3 not detected
Warning: Interchange of 9 and 4 not detected
Warning: Interchange of 0 and 5 not detected
Warning: Interchange of 1 and 6 not detected
Warning: Interchange of 2 and 7 not detected
Warning: Interchange of 3 and 8 not detected
Warning: Interchange of 4 and 9 not detected
Warning: Interchange of 5 and 0 not detected
Warning: Interchange of 6 and 1 not detected
Warning: Interchange of 7 and 2 not detected
Warning: Interchange of 8 and 3 not detected
Warning: Interchange of 9 and 4 not detected
At least one transposition undetected
```

Program II.7.b
IBM Scheme

Referred to from page 53.

Here is the ”IBM” scheme used for credit card numbers.

Java class: ErrorDetection

```
// ErrorDetection.java: base class for single-digit error detection
public class ErrorDetection {

   public static void printArray(int[] a) {
      for (int i = 0; i < a.length; i++) {
        if (a[i] == 10)
           System.out.print("X ");
        else
           System.out.print(a[i]);
        if (i%5 == 0) System.out.print(" ");
      }
      System.out.println();
   }

   public static void printUnchecked(int[] a) {
      System.out.print("? ");
      for (int i = 1; i < a.length; i++) {
        System.out.print(a[i]);
        if (i%5 == 0) System.out.print(" ");
      }
      System.out.println();
   }
}
```

Java class: IbmErrorDetection

```
// IbmErrorDetection.java: Implement "IBM" decimal error detection
public class IbmErrorDetection extends ErrorDetection{

   {\timesbf private static int sharp(int d) {
       return (2*d)/10 + (2*d)%10;
   }}

   public static int insertCheck(int[] a) {
      int check = 0;
      for (int i = 1; i < a.length; i++)
         {\timesbf if (i%2 == 0)
            check = (check + a[i])%10;
         else
            check = (check + sharp(a[i]))%10;}
      if (check == 0) a[0] = 0;
      else a[0] = -check + 10;
      return a[0];
   }
```

```
   public static boolean doCheck(int[] a) {
      int check = 0;
      for (int i = 0; i < a.length; i++)
         {\timesbf if (i%2 == 0)
            check = (check + a[i])%10;
         else
            check = (check + sharp(a[i]))%10;}
      if (check != 0) return false;
      else return true;
   }

   // main function
   public static void main (String[] args) {
      int[] a = new int[15];
      boolean checkFlag = false;
      for (int i = 1; i < a.length; i++)
         a[i] = (int)(Math.random() * 10.0);
      printUnchecked(a);
      IbmErrorDetection.insertCheck(a);
      printArray(a);
      System.out.println(IbmErrorDetection.doCheck(a));
      a[4] = (a[4] + 1)%10;

      IbmErrorDetection.printArray(a);
      System.out.println(IbmErrorDetection.doCheck(a));

      // test all adjacent transpositions
      System.out.println("\nTest all adjacent transpositions ...");
      for (int p1 = 0; p1 < 10; p1++)
         for (int p2 = 0; p2 < 10; p2++) {
            if (p1 != p2) {
               a[8] = p1; a[9] = p2;
               IbmErrorDetection.insertCheck(a);
               // interchange
               a[8] ^= a[9];
               a[9] ^= a[8];
               a[8] ^= a[9];
               if (IbmErrorDetection.doCheck(a)) {
                  System.out.println("Warning: Interchange of " +
                     p1 + " and " + p2 + " not detected");
                  checkFlag = true;
               }
            }
         }
      if (checkFlag)
         System.out.println("At least one transposition undetected");
      else
         System.out.println("All transpositions detected");

   } // end of main
}
```

Here is the output, showing a simple test, and a test of all adjacent interchanges. Interchange errors not caught are **09** and **90**.

```
? 31623 91033 1003
7 31623 91033 1003
true
7 31633 91033 1003
false

Test all adjacent transpositions ...
Warning: Interchange of 0 and 9 not detected
Warning: Interchange of 9 and 0 not detected
At least one transposition undetected
```

Program II.7.c
ISBN mod 11 Scheme

Referred to from page 53.

Here is the ISBN mod 11 scheme used for U.S. book publishing numbers.

Java class: ErrorDetection

```
// ErrorDetection.java: base class for single-digit error detection
public class ErrorDetection {

   public static void printArray(int[] a) {
      for (int i = 0; i < a.length; i++) {
        if (a[i] == 10)
           System.out.print("X ");
        else
           System.out.print(a[i]);
        if (i%5 == 0) System.out.print(" ");
      }
      System.out.println();
   }

   public static void printUnchecked(int[] a) {
      System.out.print("? ");
      for (int i = 1; i < a.length; i++) {
        System.out.print(a[i]);
        if (i%5 == 0) System.out.print(" ");
      }
      System.out.println();
   }
}
```

Java class: ISBNErrorDetection

```
// ISBNErrorDetection.java: Implement mod 11 check used by ISBN
public class ISBNErrorDetection extends ErrorDetection {

   public static int insertCheck(int[] a) {
      int check = 0;
      for (int i = 1; i < a.length; i++)
         check = (check + (i%10 + 1)*a[i])%11;
      if (check == 0) a[0] = 0;
      else a[0] = -check + 11;
      return a[0];
   }

   public static boolean doCheck(int[] a) {
      int check = 0;
      for (int i = 0; i < a.length; i++)
         check = (check + (i%10 + 1)*a[i])%11;
      if (check != 0) return false;
      else return true;
   }
```

```
   // main function
   public static void main (String[] args) {
      int[] a = new int[9];
      boolean checkFlag = false;
      for (int i = 1; i < a.length; i++)
         a[i] = (int)(Math.random() * 10.0);
      ISBNErrorDetection.printUnchecked(a);
      ISBNErrorDetection.insertCheck(a);
      ISBNErrorDetection.printArray(a);
      System.out.println(ISBNErrorDetection.doCheck(a));
      a[4] = (a[4] + 3)%10;

      ISBNErrorDetection.printArray(a);
      System.out.println(ISBNErrorDetection.doCheck(a));

      // test all adjacent transpositions
      System.out.println("\nISBN error detection scheme");
      System.out.println("\nTest all adjacent transpositions ...");
      for (int pos = 4; pos < 7; pos++)
         for (int p1 = 0; p1 < 10; p1++)
            for (int p2 = 0; p2 < 10; p2++) {
               if (p1 != p2) {
                  a[pos] = p1; a[pos+1] = p2;
                  ISBNErrorDetection.insertCheck(a);
                  // interchange
                  a[pos] ^= a[pos+1];
                  a[pos+1] ^= a[pos];
                  a[pos] ^= a[pos+1];
                  if (ISBNErrorDetection.doCheck(a)) {
                     System.out.println("Warning: Interchange of " +
                        p1 + " and " + p2 + " not detected");
                     checkFlag = true;
                  }
               }
            }
      if (checkFlag)
         System.out.println("At least one transposition undetected");
      else
         System.out.println("All transpositions detected");

   } // end of main
}
```

Here is the output, first showing a simple test. I tweaked the test until the check ”digit” was an ”X”. Next is a test of all adjacent interchanges. Here all interchanges are caught.

```
? 11696 554
X 11696 554
true
X 11626 554
false
```

```
ISBN error detection scheme

Test all adjacent transpositions ...
All transpositions detected
```

Program II.7.d
Mod 97 Scheme

Referred to from page 53.

Here is the mod **97** scheme used for extra error detection. The check below tests its performance for adjacent double error detection. One expects this check to catch apprximately **99%** of all random errors. However, it catches **99.94%** of all *adjacent double* errors (except for possible adjacent double errors involving one of the two decimals representing the check ”digit” and the first actual data digit).

Java class: ErrorDetection

```
// ErrorDetection.java: base class for single-digit error detection
public class ErrorDetection {

   public static void printArray(int[] a) {
      for (int i = 0; i < a.length; i++) {
        if (a[i] == 10)
           System.out.print("X ");
        else
           System.out.print(a[i]);
        if (i%5 == 0) System.out.print(" ");
      }
      System.out.println();
   }

   public static void printUnchecked(int[] a) {
      System.out.print("? ");
      for (int i = 1; i < a.length; i++) {
        System.out.print(a[i]);
        if (i%5 == 0) System.out.print(" ");
      }
      System.out.println();
   }
}
```

Java class: Mod97ErrorDetection

```
// Mod97ErrorDetection.java: Impl the mod 97 check for error detection
//   uses successive powers of 10 (modulo 97) for the weights
public class Mod97ErrorDetection extends ErrorDetection {

   public static int insertCheck(int[] a) {
      int check = 0;
      int weight = 10;
      for (int i = 1; i < a.length; i++) {
         check = (check + weight*a[i])%97;
         weight = (weight*10)%97;
      }
      if (check == 0) a[0] = 0;
      else a[0] = -check + 97;
      return a[0];
```

```
}

public static boolean doCheck(int[] a) {
   int check = 0;
   int weight = 1;
   for (int i = 0; i < a.length; i++) {
      check = (check + weight*a[i])%97;
      weight = (weight*10)%97;
   }
   if (check != 0) return false;
   else return true;
}

// main function
public static void main (String[] args) {
   int[] a = new int[100];
   boolean checkFlag = false;
   // no need for a random start
   for (int i = 1; i < a.length; i++)
      a[i] = (int)(Math.random() * 10.0);

   // try all adjacent double errors
   int errorCount = 0;
   int totalCount = 0;
   // try each successive position (all the same)
   for (int pos = 1; pos < 99; pos++)
      // try every pair of digits for the initial pair
      for (int p1 = 0; p1 < 10; p1++)
         for (int p2 = 0; p2 < 10; p2++) {
            // insert the initial pair
            a[pos] = p1; a[pos+1] = p2;
            // do the check and insert mod 97 check "digit"
            Mod97ErrorDetection.insertCheck(a);
            // try every pair of digits for the double error
            for (int n1 = 0; n1 < 10; n1++)
               for (int n2 = 0; n2 < 10; n2++)
                  // only try if an actual change
                  if (n1 != p1 || n2 != p2) {
                     totalCount++;
                     // insert new air as an error
                     a[pos] = n1; a[pos+1] = n2;
                     // check if the change is not detected
                     if (Mod97ErrorDetection.doCheck(a)) {
                        System.out.println("Error, old digits: " +
                           p1 + p2 + ", new digits: " + n1 + n2 +
                           ". Position: " + pos);
                        errorCount++;
                     }
                  }
         }
   System.out.println("Adjacent double errors undetected: " +
      errorCount + ", out of " + totalCount + ", or " +
      ((double)errorCount/totalCount*100) + "%");
```

```
   } // end of main
}
```

Here is the output, showing that there are only 6 kinds of adjacent double errors that remain undetected. For example, **"10"** changed to **"89"**. Here in the weight equation, **"10"** is an additional **1 + 0*10 = 1** (along with extra power of **10** weight), while **"89"** is an additional **8 + 9*10 = 98** (along with the same extra power of **10** weight). When the **98** is reduced modulo **97**, it also becomes **1**, so that the new equation has the same value as the old. The error rate is 0.060606...%, or the equation *catches* 99.94% of all adjacent double errors.

```
Error, old digits: 00, new digits: 79. Position: 1
Error, old digits: 10, new digits: 89. Position: 1
Error, old digits: 20, new digits: 99. Position: 1
Error, old digits: 79, new digits: 00. Position: 1
Error, old digits: 89, new digits: 10. Position: 1
Error, old digits: 99, new digits: 20. Position: 1

(similar entries for Position = 2, 3, ..., 98)

Adjacent double errors undetected: 588, out of 970200,
   or 0.06060606060606061%
```

Program II.7.e
Hamming mod 11 Scheme

Referred to from page 53.

Here is a test of the Hamming mod 11 Error Correcting code, using 3 check digits and 121 digits altogether. The test starts with a random initial word. It first inserts the proper check digits in positions 0, 1 and 11. The test then makes a random change in each position and corrects the change, so there are 121 changes (some are 0 added — no change). In repeated runs, the code has always corrected the proper position to the old value.

Java class: H11EC

```
// H11EC.java: Implement the mod 11 Hamming code
public class H11EC {

   public static int[] inv = {0, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1};

   // Using the sum1 check sum, if have error e and check1 result c,
   //    then pos[e][c] gives the position in error (modulo 11),
   //    using the first check equation.
   //   If the error is e and check11 result is c,
   //    then pos[e][c] gives the value position/11.
   public static int[][] pos = {
         {0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0},
         {0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10},
         {0,  6,  1,  7,  2,  8,  3,  9,  4, 10,  5},
         {0,  4,  8,  1,  5,  9,  2,  6, 10,  3,  7},
         {0,  3,  6,  9,  1,  4,  7, 10,  2,  5,  8},
         {0,  9,  7,  5,  3,  1, 10,  8,  6,  4,  2},
         {0,  2,  4,  6,  8, 10,  1,  3,  5,  7,  9},
         {0,  8,  5,  2, 10,  7,  4,  1,  9,  6,  3},
         {0,  7,  3, 10,  6,  2,  9,  5,  1,  8,  4},
         {0,  5, 10,  4,  9,  3,  8,  2,  7,  1,  6},
         {0, 10,  9,  8,  7,  6,  5,  4,  3,  2,  1}};

   public static void printArray(int[] a) {
      for (int i = 0; i < a.length; i++) {
        if (a[i] == 10)
           System.out.print("X ");
        else
           System.out.print(a[i]);
        if (i%5 == 0) System.out.print(" ");
      }
      System.out.println();
   }

   public static void printUnchecked(int[] a) {
      System.out.print("? "); // position 0
      System.out.print("?"); // position 1
      for (int i = 2; i < a.length; i++) {
```

```
      if (i == 11) System.out.print("?"); // position 11
      else System.out.print(a[i]);
     if (i%5 == 0) System.out.print(" ");
   }
   System.out.println();
}

public static void insertCheck(int[] a) {
   a[1] = inv[sum1NoCheck(a)];
   a[11] = inv[sum11NoCheck(a)];
   a[0] = inv[sum0NoCheck(a)];
   if (!doCheck(a))
      System.out.println("Failure in insertCheck");
}

public static boolean doCheck(int[] a) {
   int error = sum0(a); // amount of error
   int check1 = sum1(a);
   int check11 = sum11(a);
   if (error == 0 && check1 == 0 && check11 == 0) return true;
   if (error == 0) return false; // a double error
   int position = pos[error][check11]*11 + pos[error][check1];
   if (position >= a.length) {
     System.out.println("doCheck: position: " + position +
        ", error: " + error + ", check1: " + check1 +
        ", check11: " + check11);
     System.exit(0);
   }

   a[position] = (a[position] - error + 11)%11;
   System.out.println("Position " + position +
      " corrected to " +  a[position]);
   return true;
}

public static int sum0(int[] a) {
   int check = 0;
   for (int i = 0; i < a.length; i++)
      check = (check + a[i])%11;
   return check;
}

public static int sum0NoCheck(int[] a) {
   int check = 0;
   for (int i = 1; i < a.length; i++)
      check = (check + a[i])%11;
   return check;
}

public static int sum1(int[] a) {
   int check = 0;
   for (int i = 0; i < a.length; i++)
      check = (check + (i%11)*a[i])%11;
```

```
      return check;
   }

   public static int sum1NoCheck(int[] a) {
      int check = 0;
      for (int i = 2; i < a.length; i++)
         check = (check + (i%11)*a[i])%11;
      return check;
   }

   public static int sum11(int[] a) {
      int check = 0;
      for (int i = 0; i < a.length; i++)
         check = (check + ((i/11)%11)*a[i])%11;
      return check;
   }

   public static int sum11NoCheck(int[] a) {
      int check = 0;
      for (int i = 12; i < a.length; i++)
         check = (check + ((i/11)%11)*a[i])%11;
      return check;
   }

   // main function
   public static void main (String[] args) {
      int[] a = new int[121];
      boolean checkFlag = false;
      for (int i = 0; i < a.length; i++)
         if (i != 11) a[i] = (int)(Math.random() * 10.0);
      for (int i = 0; i < a.length; i++) {
         H11EC.insertCheck(a);
         int oldValue = a[i];
         a[i] = (a[i] + (int)(Math.random() * 10.0))%10;
         System.out.print("Position: " + i + " changed from " +
             oldValue + " to " + a[i] + "; ");
         if (oldValue == a[i]) System.out.println();
         H11EC.doCheck(a);
         if (a[i] != oldValue)
             System.out.println("*************************");
      }
   } // end of main
}
```

Here is the output from a run:

```
% java H11EC
Position: 0 changed from 10 to 3; Position 0 corrected to 10
Position: 1 changed from 10 to 3; Position 1 corrected to 10
Position: 2 changed from 1 to 1;
Position: 3 changed from 0 to 2; Position 3 corrected to 0
Position: 4 changed from 0 to 0;
```

```
  . . . (many lines omitted) . . .

Position: 117 changed from 4 to 1; Position 117 corrected to 4
Position: 118 changed from 5 to 6; Position 118 corrected to 5
Position: 119 changed from 4 to 9; Position 119 corrected to 4
Position: 120 changed from 7 to 7;
```

Program II.7.f
Hamming mod 11 Scheme, Double Errors

Referred to from page 53.

Here is a Java program that simulates random double errors. There are two input parameters on the command line: the number of simulation trials to run, and the number of digits to use. The program identifies various outcomes:

Outcome	% for 18 digits	% for 121 digits
Positions 1 and 11 give location of error, but position 0 gives amount of error as 0.	9.19%	10.00%
The check tries to correct an error with location out of range of the number.	75.01%	0.00%
Check tries to do single error correction with a ten (X) somewhere besides positions 0, 1, or 11	1.30%	8.51%
Misses error: miscorrects as if there were a single error	**14.44%**	**81.48%**

Thus, with 18 digits, only 14.4% of double errors get ”corrected” as if there were a single error, while with 121 digits 81.5% are miscorrected in this way.

Java class: H11ED

```
// H11ED.java: Implement the mod 11 Hamming code
public class H11ED {

   public static int[] inv = {0, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1};

   public static int totalTrials, misCorrected = 0,
      errorZero = 0, toTen = 0, subscriptRange = 0, allZero = 0;
   public static int arraySize;
   public static int[][] pos = {
         {0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0},
         {0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10},
         {0,  6,  1,  7,  2,  8,  3,  9,  4, 10,  5},
         {0,  4,  8,  1,  5,  9,  2,  6, 10,  3,  7},
         {0,  3,  6,  9,  1,  4,  7, 10,  2,  5,  8},
         {0,  9,  7,  5,  3,  1, 10,  8,  6,  4,  2},
         {0,  2,  4,  6,  8, 10,  1,  3,  5,  7,  9},
         {0,  8,  5,  2, 10,  7,  4,  1,  9,  6,  3},
         {0,  7,  3, 10,  6,  2,  9,  5,  1,  8,  4},
         {0,  5, 10,  4,  9,  3,  8,  2,  7,  1,  6},
         {0, 10,  9,  8,  7,  6,  5,  4,  3,  2,  1}};
```

```
public static void printArray(int[] a) {
   for (int i = 0; i < a.length; i++) {
      if (a[i] == 10)
         System.out.print("X ");
      else
         System.out.print(a[i]);
      if (i%5 == 0) System.out.print(" ");
   }
   System.out.println();
}

public static void printUnchecked(int[] a) {
   System.out.print("? "); // position 0
   System.out.print("?"); // position 1
   for (int i = 2; i < a.length; i++) {
      if (i == 11) System.out.print("?"); // position 11
      else System.out.print(a[i]);
      if (i%5 == 0) System.out.print(" ");
   }
   System.out.println();
}

public static void insertCheck(int[] a) {
   a[1] = inv[sum1NoCheck(a)];
   a[11] = inv[sum11NoCheck(a)];
   a[0] = inv[sum0NoCheck(a)];
}

public static boolean doCheck(int[] a) {
   int error = sum0(a); // amount of error
   int check1 = sum1(a);
   int check11 = sum11(a);
   if (error == 0 && check1 == 0 && check11 == 0) {
      allZero++;
      return true;
   }
   if (error == 0) {
       // System.out.println("Double error: check 0 is zero");
       errorZero++;
       return false; // a double error
   }
   int position = pos[error][check11]*11 + pos[error][check1];
   if (position >= a.length) {
      subscriptRange++;
      return false;
   }
   a[position] = (a[position] - error + 11)%11;
   if (a[position] == 10 && (position != 0 && position != 1 &&
         position != 11)) {
      toTen++;
      return false;
   }
```

```
      misCorrected++;
      return true;
   }

   public static int sum0(int[] a) {
      int check = 0;
      for (int i = 0; i < a.length; i++)
         check = (check + a[i])%11;
      return check;
   }

   public static int sum0NoCheck(int[] a) {
      int check = 0;
      for (int i = 1; i < a.length; i++)
         check = (check + a[i])%11;
      return check;
   }

   public static int sum1(int[] a) {
      int check = 0;
      for (int i = 0; i < a.length; i++)
         check = (check + (i%11)*a[i])%11;
      return check;
   }

   public static int sum1NoCheck(int[] a) {
      int check = 0;
      for (int i = 2; i < a.length; i++)
         check = (check + (i%11)*a[i])%11;
      return check;
   }

   public static int sum11(int[] a) {
      int check = 0;
      for (int i = 0; i < a.length; i++)
         check = (check + ((i/11)%11)*a[i])%11;
      return check;
   }

   public static int sum11NoCheck(int[] a) {
      int check = 0;
      for (int i = 12; i < a.length; i++)
         check = (check + ((i/11)%11)*a[i])%11;
      return check;
   }

   // main function
   public static void main (String[] args) {
      totalTrials = Integer.parseInt(args[0]); // total num of trials
      arraySize   = Integer.parseInt(args[1]); // size of array
      int[] a = new int[arraySize];
      int loc1, loc2;
      boolean checkFlag = false;
```

```
      // start with random word
      for (int i = 2; i < a.length; i++)
         if (i != 11) a[i] = (int)(Math.random() * 10.0);
      H11ED.insertCheck(a);
      for (int i = 0; i < totalTrials; i++) {
         // try random pair of errors, choose 2 distinct random ints
         loc1 = (int)(Math.random() * a.length);
         do {
            loc2 = (int)(Math.random() * a.length);
         } while (loc1 == loc2);
         a[loc1] = (a[loc1] + (int)(Math.random() * 9.0 + 1.0))%10;
         a[loc2] = (a[loc2] + (int)(Math.random() * 9.0 + 1.0))%10;
         H11ED.doCheck(a);
      }
      if (totalTrials != (misCorrected + errorZero + toTen +
            subscriptRange + allZero))
         System.out.println("Count Off");
      System.out.println("Total: " + totalTrials +
         ", errorZero: "    + errorZero +
         ", toTen: "        + toTen +
         ", subscript: "    + subscriptRange +
         ", misCorrected: " + misCorrected +
         ", allZero: "      + allZero);
   } // end of main
}
```

Here are the results of two runs, the first using 18 digits (15 data digits) and the second using the maximum of 121 digits (118 data digits). In each case a large number of double errors were deliberately introduced. In the first case, all but **14.9%** of these double errors were detected. In the second case only **18.5%** of double errors were detected.

```
% myjava H11ED 10000000 18
Total: 10000000, errorZero: 919248, toTen: 130462, subscript: 7501457,
       misCorrected: 1444074, allZero: 4759
% myjava H11ED 1000000 121
Total: 1000000, errorZero: 100002, toTen: 85080, subscript: 0,
       misCorrected: 814827, allZero: 91
```

Program II.8.a
Use of the Dihedral Group

Referred to from page 57.

Here is the a scheme using the dihedral group without any special permutations. Notice that **2/3** of all adjacent transpositions are detected (**60** out of **90**).

Java class: ErrorDetection

```
// ErrorDetection.java: base class for single-digit error detection
public class ErrorDetection {

   public static void printArray(int[] a) {
      for (int i = 0; i < a.length; i++) {
        if (a[i] == 10)
           System.out.print("X ");
        else
           System.out.print(a[i]);
        if (i%5 == 0) System.out.print(" ");
      }
      System.out.println();
   }

   public static void printUnchecked(int[] a) {
      System.out.print("? ");
      for (int i = 1; i < a.length; i++) {
        System.out.print(a[i]);
        if (i%5 == 0) System.out.print(" ");
      }
      System.out.println();
   }
}
```

Java class: DihedralErrorDetection

```
// DihedralErrorDetection.java: the dihedral group
//    for decimal error detection
public class DihedralErrorDetection extends ErrorDetection {

   private static int[][] op= {
      {0, 1, 2, 3, 4, 5, 6, 7, 8, 9},
      {1, 2, 3, 4, 0, 6, 7, 8, 9, 5},
      {2, 3, 4, 0, 1, 7, 8, 9, 5, 6},
      {3, 4, 0, 1, 2, 8, 9, 5, 6, 7},
      {4, 0, 1, 2, 3, 9, 5, 6, 7, 8},
      {5, 9, 8, 7, 6, 0, 4, 3, 2 ,1},
      {6, 5, 9, 8, 7, 1, 0, 4, 3, 2},
      {7, 6, 5, 9, 8, 2, 1, 0, 4, 3},
      {8, 7, 6, 5, 9, 3, 2, 1, 0, 4},
      {9, 8, 7, 6, 5, 4, 3, 2, 1, 0} };

   private static int[] inv = {0, 4, 3, 2, 1, 5, 6, 7, 8, 9};
```

```
public static int insertCheck(int[] a) {
   int check = 0;
   for (int i = 1; i < a.length; i++)
      check = op[check][ a[i] ];
   a[0] = inv[check];
   return a[0];
}

public static boolean doCheck(int[] a) {
   int check = 0;
   for (int i = 0; i < a.length; i++)
      check = op[check][ a[i] ];
   if (check != 0) return false;
   else return true;
}

// main function
public static void main (String[] args) {
   int[] a = new int[15];
   boolean checkFlag = false;
   for (int i = 1; i < a.length; i++)
      a[i] = (int)(Math.random() * 10.0);
   DihedralErrorDetection.printUnchecked(a);
   DihedralErrorDetection.insertCheck(a);
   DihedralErrorDetection.printArray(a);
   System.out.println(DihedralErrorDetection.doCheck(a));
   a[4] = (a[4] + 1)%10;

   printArray(a);
   System.out.println(DihedralErrorDetection.doCheck(a));

   // test all adjacent transpositions
   System.out.println("\nThe straight dihedral group");
   System.out.println("\nTest all adjacent transpositions");
   for (int p1 = 0; p1 < 10; p1++)
      for (int p2 = 0; p2 < 10; p2++) {
         if (p1 != p2) {
            a[8] = p1; a[9] = p2;
            DihedralErrorDetection.insertCheck(a);
            // interchange
            a[8] ^= a[9];
            a[9] ^= a[8];
            a[8] ^= a[9];
            if (DihedralErrorDetection.doCheck(a)) {
               System.out.println("Warning: Interchange of " +
                  p1 + " and " + p2 + " not detected");
               checkFlag = true;
            }
         }
      }
   if (checkFlag)
      System.out.println("At least one transposition undetected");
```

```
      else
         System.out.println("All transpositions detected");

   } // end of main
}
```

Here is the output, showing a simple test, and a test of all adjacent interchanges. Notice that 30 (out of 90) adjacent transpositions go undetected.

```
? 49588 58802 3606
8 49588 58802 3606
true
8 49598 58802 3606
false

The straight dihedral group

Test all adjacent transpositions
Warning: Interchange of 0 and 1 not detected
Warning: Interchange of 0 and 2 not detected
Warning: Interchange of 0 and 3 not detected
Warning: Interchange of 0 and 4 not detected
Warning: Interchange of 0 and 5 not detected
Warning: Interchange of 0 and 6 not detected
Warning: Interchange of 0 and 7 not detected
Warning: Interchange of 0 and 8 not detected
Warning: Interchange of 0 and 9 not detected
Warning: Interchange of 1 and 0 not detected
Warning: Interchange of 1 and 2 not detected
Warning: Interchange of 1 and 3 not detected
Warning: Interchange of 1 and 4 not detected
Warning: Interchange of 2 and 0 not detected
Warning: Interchange of 2 and 1 not detected
Warning: Interchange of 2 and 3 not detected
Warning: Interchange of 2 and 4 not detected
Warning: Interchange of 3 and 0 not detected
Warning: Interchange of 3 and 1 not detected
Warning: Interchange of 3 and 2 not detected
Warning: Interchange of 3 and 4 not detected
Warning: Interchange of 4 and 0 not detected
Warning: Interchange of 4 and 1 not detected
Warning: Interchange of 4 and 2 not detected
Warning: Interchange of 4 and 3 not detected
Warning: Interchange of 5 and 0 not detected
Warning: Interchange of 6 and 0 not detected
Warning: Interchange of 7 and 0 not detected
Warning: Interchange of 8 and 0 not detected
Warning: Interchange of 9 and 0 not detected
At least one transposition undetected
```

Program II.8.b
Verhoeff's Scheme

Referred to from page 57.

Here is Verhoeff's scheme using the dihedral group using special permutations. Notice that all adjacent transpositions are detected.

Java class: ErrorDetection

```
// ErrorDetection.java: base class for single-digit error detection
public class ErrorDetection {

   public static void printArray(int[] a) {
      for (int i = 0; i < a.length; i++) {
        if (a[i] == 10)
           System.out.print("X ");
        else
           System.out.print(a[i]);
        if (i%5 == 0) System.out.print(" ");
      }
      System.out.println();
   }

   public static void printUnchecked(int[] a) {
      System.out.print("? ");
      for (int i = 1; i < a.length; i++) {
        System.out.print(a[i]);
        if (i%5 == 0) System.out.print(" ");
      }
      System.out.println();
   }
}
```

Java class: VerhoeffErrorDetection

```
// VerhoeffErrorDetection.java: Verhoeff's decimal error detection
public class VerhoeffErrorDetection extends ErrorDetection {

   private static int[][] op= {
      {0, 1, 2, 3, 4, 5, 6, 7, 8, 9},
      {1, 2, 3, 4, 0, 6, 7, 8, 9, 5},
      {2, 3, 4, 0, 1, 7, 8, 9, 5, 6},
      {3, 4, 0, 1, 2, 8, 9, 5, 6, 7},
      {4, 0, 1, 2, 3, 9, 5, 6, 7, 8},
      {5, 9, 8, 7, 6, 0, 4, 3, 2 ,1},
      {6, 5, 9, 8, 7, 1, 0, 4, 3, 2},
      {7, 6, 5, 9, 8, 2, 1, 0, 4, 3},
      {8, 7, 6, 5, 9, 3, 2, 1, 0, 4},
      {9, 8, 7, 6, 5, 4, 3, 2, 1, 0} };

   private static int[] inv = {0, 4, 3, 2, 1, 5, 6, 7, 8, 9};
   private static int[][] F = new int[8][];
```

```
public VerhoeffErrorDetection() { // identity and "magic" perms
   F[0] = new int[]{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}; // identity
   F[1] = new int[]{1, 5, 7, 6, 2, 8, 3, 0, 9, 4}; // "magic"
   for (int i = 2; i < 8; i++) {
      F[i] = new int[10];
      for (int j = 0; j < 10; j++)
         F[i][j] = F[i-1] [F[1][j]];
   }
}

public static int insertCheck(int[] a) {
   int check = 0;
   for (int i = 1; i < a.length; i++)
      check = op[check][ F[i % 8][a[i]] ];
   a[0] = inv[check];
   return a[0];
}

public static boolean doCheck(int[] a) {
   int check = 0;
   for (int i = 0; i < a.length; i++)
      check = op[check][ F[i % 8][a[i]] ];
   if (check != 0) return false;
   else return true;
}

// main function
public static void main (String[] args) {
   VerhoeffErrorDetection v = new VerhoeffErrorDetection();
   int[] a = new int[15];
   boolean checkFlag = false;
   for (int i = 1; i < a.length; i++)
      a[i] = (int)(Math.random() * 10.0);
   VerhoeffErrorDetection.printUnchecked(a);
   VerhoeffErrorDetection.insertCheck(a);
   VerhoeffErrorDetection.printArray(a);
   System.out.println(VerhoeffErrorDetection.doCheck(a));
   a[4] = (a[4] + 1)%10;

   VerhoeffErrorDetection.printArray(a);
   System.out.println(VerhoeffErrorDetection.doCheck(a));

   // test all adjacent transpositions
   System.out.println("\nTest all adjacent transpositions");
   for (int p1 = 0; p1 < 10; p1++)
      for (int p2 = 0; p2 < 10; p2++) {
         if (p1 != p2) {
            a[8] = p1; a[9] = p2;
            VerhoeffErrorDetection.insertCheck(a);
            // interchange
            a[8] ^= a[9];
            a[9] ^= a[8];
```

```
                a[8] ^= a[9];
                if (VerhoeffErrorDetection.doCheck(a)) {
                   System.out.println("Warning: Interchange of " +
                      p1 + " and " + p2 + " not detected");
                   checkFlag = true;
                }
             }
          }
       if (checkFlag)
          System.out.println("At least one transposition undetected");
       else
          System.out.println("All transpositions detected");
    } // end of main
}
```

Here is the output, showing a simple test, and a test of all adjacent interchanges. All interchange errors are detected.

```
? 75787 12372 9429
1 75787 12372 9429
true
1 75797 12372 9429
false

Test all adjacent transpositions
All transpositions detected
```

Program III.9.a
Cryptogram Program

Referred to from page 62.

Here is a Java program to create standard cryptograms, as they are found in newspapers. The program will read the quotation to be scrambled into a cryptogram from the standard input. In Unix this file can just be directed into the program, as shown in the commands below. Each time it is executed, the program will create a new and unique translation table to create the cryptogram. The resulting table and cryptogram itself are output on the standard output file, which might be redirected into a named file.

Java class: Cryptogram

```
// Cryptogram: create a cryptogram as in a newspaper
import java.io.*;
public class Cryptogram {

   private char[] alf = new char[26]; // translation vector
   public Cryptogram() {
      for (int i = 0; i < alf.length; i++) alf[i] = (char)('A' + i);
      randomize();
   }
   private int rand(int r, int s) { // r <= rand <= s
      return (int)((s - r + 1)*Math.random() + r);
   }
   private void randomize() {
      for (int i = 0; i < alf.length - 1; i++) {
         // Note: for a random permutation, replace "i+1" by "i" below
         // However, we want no letter to remain in its original spot
         int ind = rand(i+1, alf.length - 1);
         char t = alf[i];
         alf[i] = alf[ind];
         alf[ind] = t;
      }
   }
   public void printArray() {
      System.out.print("Alphabet:      ");
      for (int i = 0; i < alf.length; i++)
         System.out.print((char)('A' + i));
      System.out.println();
      System.out.print("Translated to: ");
      for (int i = 0; i < alf.length; i++)
         System.out.print(alf[i]);
      System.out.println("\n");
   }
   // getNextChar: fetch next char.
   public char getNextChar() {
      char ch = ' '; // = ' ' to keep compiler happy
      try {
         ch = (char)System.in.read();
```

```
            } catch (IOException e) {
               System.out.println("Exception reading character");
            }
            return ch;
         }
         public void createCryptogram() {
            char ch;
            while ((byte)(ch = getNextChar()) != -1) {
               if (Character.isUpperCase(ch)) ch = alf[ch - 'A'];
               System.out.print(ch);
            }
         }
         // main: for cryptogram program
         public static void main(String[] args) {
            Cryptogram crypto = new Cryptogram();
            crypto.printArray();
            crypto.createCryptogram();
         }
      }
```

Here is a run of the program, first showing the quotation to be translated, and then the translated version, that is, the cryptogram:

```
% cat quote.text
AND WE ARE HERE AS ON A DARKLING PLAIN
SWEPT WITH CONFUSED ALARMS OF STRUGGLE AND FLIGHT,
WHERE IGNORANT ARMIES CLASH BY NIGHT.
DOVER BEACH, MATHEW ARNOLD
% java Cryptogram < quote.text
Alphabet:      ABCDEFGHIJKLMNOPQRSTUVWXYZ
Translated to: ZUWYMPILBDJRVFHQSGAXNCTKOE

ZFY TM ZGM LMGM ZA HF Z YZGJRBFI QRZBF
ATMQX TBXL WHFPNAMY ZRZGVA HP AXGNIIRM ZFY PRBILX,
TLMGM BIFHGZFX ZGVBMA WRZAL UO FBILX.
YHCMG UMZWL, VZXLMT ZGFHRY
```

Now suppose one wants to have nothing but the letters in the cryptogram (no spaces, newlines, or other punctuation). This is the same as the other program, except that the **`createCryptogram()`** has become:

```
      public void createCryptogram()
         char ch;
         while ((byte)(ch = getNextChar()) != -1)
            if (Character.isUpperCase(ch))
               ch = alf[ch - 'A'];
               System.out.print(ch);


         System.out.println();
```

Here is the output of this program:

```
% java Cryptogram2 < quote.text
```

```
Alphabet:      ABCDEFGHIJKLMNOPQRSTUVWXYZ
Translated to: OXKQFDZGACIVLHEJSMBWPNURTY

OHQUFOMFGFMFOBEHOQOMIVAHZJVOAHBUFJWUAWGKEHDPBFQOVOMLBEDBWMPZZVF
OHQDVAZGWUGFMFAZHEMOHWOMLAFBKVOBGXTHAZGWQENFMXFOKGLOWGFUOMHEVQ
```

Here is what the message looks like after decrypting:

```
ANDWEAREHEREASONADARKLINGPLAINSWEPTWITHCONFUSEDALARMSOFSTRUGGLE
ANDFLIGHTWHEREIGNORANTARMIESCLASHBYNIGHTDOVERBEACHMATHEWARNOLD
```

Program III.10.a
Caesar Cipher

Referred to from page 67.

Here is a Java implementation of the Caesar cipher. The program reads the input message from the standard input and outputs the ciphertext on the standard output. The key is an integer in the range from 0 to 25 inclusive, and one must use the same key for encryption and decryption. The program uses a function **`rotate`** to translate lowercase characters in a circle by a distance of **`key`**.

Java class: Caesar

```
// Caesar.java: implement the Caesar cipher
//   This carries out a simple rotation of lower-case letters, and
//   does nothing to all other characters, making the decryption
//   process even easier, because caps and punctuation marks survive
//   unchanged.
// Usage: java Caesar (-d | -e) key
//   Above, option "-d" is for decryption, "-e" is for encryption
import java.io.*;
public class Caesar {
   private Reader in; // standard input stream for message
   private int key; // (en|de)cryption key

   // Caesar: constructor, opens standard input, passes key
   public Caesar(int k) {
      // open file
      in = new InputStreamReader(System.in);
      key = k;
   }
   // (en|de)crypt: just feed in opposite parameters
   public void encrypt() { translate(key); }
   public void decrypt() { translate(-key); }

   // translate: input message, translate
   private void translate(int k) {
      char c;
      while ((byte)(c = getNextChar()) != -1) {
         if (Character.isLowerCase(c)) {
            c = rotate(c, k);
         }
         System.out.print(c);
      }
   }

   // getNextChar: fetches next char.
   public char getNextChar() {
      char ch = ' '; // = ' ' to keep compiler happy
      try {
         ch = (char)in.read();
```

```
      } catch (IOException e) {
         System.out.println("Exception reading character");
      }
      return ch;
   }
   // rotate: translate using rotation, version with table lookup
   public char rotate(char c, int key) { // c must be lowercase
      String s = "abcdefghijklmnopqrstuvwxyz";
      int i = 0;
      while (i < 26) {
         // extra +26 below because key might be negative
          if (c == s.charAt(i)) return s.charAt((i + key + 26)%26);
             i++;
      }
      return c;
   }

   // main: check command, (en|de)crypt, feed in key value
   public static void main(String[] args) {
      if (args.length != 2) {
         System.out.println("Usage: java Caesar (-d | -e) key");
         System.exit(1);
      }
      Caesar cipher = new Caesar(Integer.parseInt(args[1]));
      if (args[0].equals("-e")) cipher.encrypt();
      else if (args[0].equals("-d")) cipher.decrypt();
      else {
         System.out.println("Usage: java Caesar (-d | -e) key");
         System.exit(1);
      }
   }
}
```

Here is the result of an initial run of the program. First is the message file (a quotation for Ecclesiastes), followed by encryption by the key **3**, and then by encryption followed by decryption (both using the same key), showing that the original message results. An simple analysis of the ciphertext would show a distribution of letters that would immediately lead to breaking the code. Notice also that the plaintext and ciphertext both end in double letters (**ll** and **oo**).

```
% cat message.text
i returned, and saw under the sun, that the race is not to the swift,
nor the battle to the strong, neither yet bread to the wise, nor yet
riches to men of understanding, nor yet favour to men of skill; but
time and chance happeneth to them all.

% java Caesar -e 3 < message.text
l uhwxuqhg, dqg vdz xqghu wkh vxq, wkdw wkh udfh lv qrw wr wkh vzliw,
qru wkh edwwoh wr wkh vwurqj, qhlwkhu bhw euhdg wr wkh zlvh, qru bhw
ulfkhv wr phq ri xqghuvwdqglqj, qru bhw idyrxu wr phq ri vnloo; exw
wlph dqg fkdqfh kdsshqhwk wr wkhp doo.

% java Caesar -e 3 < message.text | java Caesar -d 3
```

```
i returned, and saw under the sun, that the race is not to the swift,
nor the battle to the strong, neither yet bread to the wise, nor yet
riches to men of understanding, nor yet favour to men of skill; but
time and chance happeneth to them all.
```

Notice that the ciphertext and decrypted plaintext both have all the original punctuation characters, making it even easier to break the system. A more reasonable system would drop all such punctuation characters from the ciphertext. The one must break the decrypted ciphertext into separate words — a task that is not hard for English. Here is a run of the program that has been altered to discard all characters except for lower-case letters. The final version of the message shows the words run together.

```
% java Caesar2 -e 3 < message.text
luhwxuqhgdqgvdzxqghuwkhvxqwkdwwkhudfhlvqrwwrwkhvzliwqruwkhedwwo
hwrwkhvwurqjqhlwkhubhweuhdgwrwkhzlvhqrubhwulfkhvwrphqrixqghuvwd
qglqjqrubhwidyrxuwrphqrivnlooexwwlphdqgfkdqfhkdsshqhwkwrwkhpdoo

% java Caesar2 -e 3 < message.text | java Caesar2 -d 3
ireturnedandsawunderthesunthattheraceisnottotheswiftnorthebattl
etothestrongneitheryetbreadtothewisenoryetrichestomenofundersta
ndingnoryetfavourtomenofskillbuttimeandchancehappenethtothemall
```

Program III.10.b
Beale Cipher

Referred to from page 67.

Here is a Java implementation of the Beale cipher. As with the Caesar cipher, the program reads the input message from the standard input and outputs the ciphertext on the standard output. However, this program also reads a file to use as the key: **`key1.text`** in the first run and **`key1.text`** in the second run. The first key file is very simple — just the letter **`d`** repeated over and over. This shows that the Beale cipher includes the Caesar cipher as a special case. The second key file is just another quotation (from B.F. Skinner). The program **`Beale.java`** only uses successive lowercase letters from a key file (and discards the other letters).

Java class: Beale

```
// Beale.java: implement the Beale cipher
// Usage: java Beale (-d | -e) keyFile
//  The program reads a separate file (keyFile) for the key or pad.
//  The message is just the standard input.
//  Caps and punctuation marks in the message remain unchanged.
//  Only lowercase letters are used in the key file/
import java.io.*;
public class Beale {
   private Reader messIn; // System.in for message
   private Reader keyIn;  // keyFile for key file

   // Beale: constructor -- just open files
   public Beale(String keyFile) {
      messIn = new InputStreamReader(System.in);
      try {
         keyIn = new FileReader(keyFile);
      } catch (IOException e) {
         System.out.println("Exception opening keyFile");
      }
   }

   // (en|de)crypt: just feed in opposite parameters
   public void encrypt() { translate(1); }
   public void decrypt() { translate(-1); }

   // translate: read keyFile and input, translate
   private void translate(int direction) {
      char c, key_c;
      while ((byte)(c = getNextChar(messIn)) != -1) {
         if (Character.isLowerCase(c)) {
            // fetch lowercase letter from key file
            while (!Character.isLowerCase(key_c = getNextChar(keyIn)))
               ;
            c = rotate(c, ((direction*(key_c - 'a')) + 26)%26);
         }
         System.out.print(c);
```

```
      }
   }

   // getNextChar: fetches next char.  Also opens input file
   public char getNextChar(Reader in) {
      char ch = ' '; // = ' ' to keep compiler happy
      try {
         ch = (char)in.read();
      } catch (IOException e) {
         System.out.println("Exception reading character");
      }
      return ch;
   }

   // rotate: translate using rotation -- simpler version
   //  This just uses arithmetic on char types,
   public char rotate(char c, int key) {
      int res = ((c - 'a') + key + 26)%26 + 'a';
      return (char)res;
   }

   // main: check command, (en|de)crypt, feed in keyFile
   public static void main(String[] args) {
      if (args.length != 2) {
         System.out.println("Usage: java Beale (-d | -e) keyFile");
         System.exit(1);
      }
      Beale cipher = new Beale(args[1]);
      if (args[0].equals("-e")) cipher.encrypt();
      else if (args[0].equals("-d")) cipher.decrypt();
      else {
         System.out.println("Usage: java Beale (-d | -e) keyFile");
         System.exit(1);
      }
   }
}
```

Here are the results of a run of the program where the key file consists of all letters "d", so that is does the same rotation by 3 as the previous example of the Caesar cipher:

```
% cat message.text
i returned, and saw under the sun, that the race is not to the swift,
nor the battle to the strong, neither yet bread to the wise, nor yet
riches to men of understanding, nor yet favour to men of skill; but
time and chance happeneth to them all.

% cat key1.text
ddddddddddddddddddddddddddddddddddddddddddddddddddd
ddddddddddddddddddddddddddddddddddddddddddddddddddd
ddddddddddddddddddddddddddddddddddddddddddddddddddd
ddddddddddddddddddddddddddddddddddddddddddddddddddd
ddddddddddddddddddddddddddddddddddddddddddddddddddd
ddddddddddddddddddddddddddddddddddddddddddddddddddd
```

```
% java Beale -e key1.text < message.text
l uhwxuqhg, dqg vdz xqghu wkh vxq, wkdw wkh udfh lv qrw wr wkh vzliw,
qru wkh edwwoh wr wkh vwurqj, qhlwkhu bhw euhdg wr wkh zlvh, qru bhw
ulfkhv wr phq ri xqghuvwdqglqj, qru bhw idyrxu wr phq ri vnloo; exw
wlph dqg fkdqfh kdsshqhwk wr wkhp doo.

% java Beale -e key1.text < message.text | java Beale -d key.text
i returned, and saw under the sun, that the race is not to the swift,
nor the battle to the strong, neither yet bread to the wise, nor yet
riches to men of understanding, nor yet favour to men of skill; but
time and chance happeneth to them all.
```

Notice that the ciphertext and decrpted plaintext has all the original punctuation characters, making it easier to break the system. Here is a run of a variation of the program that discards all characters except for lower-case letters:

```
% java Beale2 -e key1.text < message.text
luhwxuqhgdqgvdzxqghuwkhvxqwkdwwkhudfhlvqrwwrwkhvzliwqruwkhedwwo
hwrwkhvwurqjqhlwkhubhweuhdgwrwkhzlvhqrubhwulfkhvwrphqrixqghuvwd
qglqjqrubhwidyrxuwrphqrivnlooexwwlphdqgfkdqfhkdsshqhwkwrwkhpdoo

% java Beale2 -e key1.text < message.text | java Beale2 -d key1.text
ireturnedandsawunderthesunthattheraceisnottotheswiftnorthebattl
etothestrongneitheryetbreadtothewisenoryetrichestomenofundersta
ndingnoryetfavourtomenofskillbuttimeandchancehappenethtothemall
```

Finally, here is a real run with an actual non-trivial key file: **`key2.text`**. Recall that without knowing the text of the key file, this would be very difficult to cryptanalyze (break). For example, as before, the message ends with double letters: **`ll`**. However, this time the ciphertext ends with two different letters: **`gp`**.

```
% cat key2.text
A Golden Age, whether of art or music or science or peace or
plenty, is out of reach of our economic and governmental techniques.
something may be done by accident, as it has from time to time in the
past, but not by deliberate intent.  At this very moment enormous
numbers of intelligent men and women of good will are trying to build a
better world.  But problems are born faster than they can be solved.

% java Beale -e key2.text < message.text
w chxhxrak, egk wrk znuxf kty kcp, hysv blr teqv xw nqx hf isi fpgnl,
bik hmv favazj hi klg ggfavi, nrlzvzv prf fexao ms vor eymi, fgf kim
yqpnqs rp qhb bj vldgtaweawifo, gvr qjk tmowgv mc fmz sn fdppa; bmm
ucfr oge akeykf lrpiivrml gh malu sgp.

% java Beale -e key2.text < message.text | java Beale -d key2.text
i returned, and saw under the sun, that the race is not to the swift,
nor the battle to the strong, neither yet bread to the wise, nor yet
riches to men of understanding, nor yet favour to men of skill; but
time and chance happeneth to them all.
```

Here as before is a run that discards the punctuation characters:

```
% java Beale2 -e key2.text < message.text
wchxhxrakegkwrkznuxfktykcphysvblrteqvxwnqxhfisifpgnlbikhmvfavaz
```

```
jhiklgggfavinrlzvzvprffexaomsvoreymifgfkimyqpnqsrpqhbbjvldgtawe
awifogvrqjktmowgvmcfmzsnfdppabmmucfrogeakeykflrpiivrmlghmalusgp

% java Beale2 -e key2.text < message.text | java Beale2 -d key2.text
ireturnedandsawunderthesunthattheraceisnottotheswiftnorthebattl
etothestrongneitheryetbreadtothewisenoryetrichestomenofundersta
ndingnoryetfavourtomenofskillbuttimeandchancehappenethtothemall
```

Program III.10.c
Generating a One-Time Pad

Referred to from page 69.

Here is a Java program which, when executed, generates a unique Postscript file that will print two copies of 1000 random letters for a one-time pad. This program is just for simple demonstration purposes, and would not be suitable for applications because of weaknesses in the random number generator and its seed. A program for actual use would need a more secure generator with at least a 128-bit seed made up from various random inputs.

Java class: Pad

```
// Pad.java: generate Postscript code to print a one-time pad
import java.text.DecimalFormat;

public class Pad {
   static DecimalFormat twoDigits = new DecimalFormat("00");
   static char[] let =
      {'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'};
   static int xCoord = 0, yCoord = 0, lineCount = 0;

   public static void main (String[] args) {
      System.out.println("%!PS-Adobe-2.0");
      System.out.println("/Courier-Bold findfont 14 scalefont setfont");
      System.out.println("/onepad {");
      for (int i = 0; i < 20; i++) {
         System.out.println("0 " + yCoord + " moveto");
         System.out.print("(" + twoDigits.format(lineCount) + " ");
         for (int j = 0; j < 50; j++) {
            System.out.print(oneLet());
            if (j%5  == 4) System.out.print(" ");
            if (j%10 == 9) System.out.print(" ");
         }
         System.out.println(") show");
         yCoord -= 15;
         if (lineCount == 9) System.out.println();
         if (lineCount%5 == 4) yCoord -= 15;
         lineCount++;
      }
      System.out.println("}def");
      System.out.println("gsave 30 750 translate onepad grestore");
      System.out.println("gsave 30 360 translate onepad grestore");
      System.out.println("10 390 moveto");
      System.out.print  ("(============t=e=a=r===h=e=r=e=======");
      System.out.println("=======t=e=a=r===h=e=r=e==========) show");
      System.out.println("showpage");
   } // end of main

   private static char oneLet() { return let[(int)(Math.random()*26)];}
```

```
    }
```

Here is typical output of the program, a Postscript file. (It will have different random characters in it each time it is generated by executing the Java program.)

```
%!PS-Adobe-2.0
/Courier-Bold findfont 14 scalefont setfont
/onepad {
0 0 moveto
(00 XLCWT HZZTC  HUTXA GQAUN  FXCUI QFBVW  DKAPS SXKHK  XBLLP LTHFO  ) show
0 -15 moveto
(01 XUMVS SRMGB  SPSDI UAFYO  CQYHQ CYSHU  UCATL HLDKZ  XWFGR LRMOL  ) show
0 -30 moveto
(02 DXCBR FMARY  MOLUR CDJVT  MACWT IRFVP  DZBBY ZTFZG  HCUUL YYKPF  ) show
0 -45 moveto
(03 QSWIH HNWRC  FQQHM ZEKJP  URILH THNSY  TLMWA CWYJV  AWOWY IAELZ  ) show
0 -60 moveto
(04 KJKJE NGKDD  NYMYP NRNHK  KRIOX TIFSY  TGYRS HZLTQ  MCEBA YRBTU  ) show
0 -90 moveto
(05 CZLFW HYSTM  YGDBN GVAZE  RFCQY YFHLC  OGLHY OEBNB  QDQIS EHIBB  ) show
0 -105 moveto
(06 ZIWQC TYERS  UZGSS EGOVX  KGXJV AONYN  HXZCR RLUSN  SXXBR EJKQH  ) show
0 -120 moveto
(07 OIJMF LCZHA  CGMYK LPLMU  CIMHZ WBVHD  JXNZJ CJLSN  PJVFL HENCU  ) show
0 -135 moveto
(08 ZOHEW JRWPC  BHFRZ MEYBW  SHFPQ DYQFH  ARDQI UAHOC  OAQKR LZPBF  ) show
0 -150 moveto
(09 YAKMA YJVYY  HSAZB SMILL  WMHAJ KMYNO  YHTXY GIISP  CVYQC OMCIP  ) show

0 -180 moveto
(10 NPNVY VACCU  YWOFW GLHTO  TOUPS LRLOV  SZWBL WYNJO  NOHCZ SZSZS  ) show
0 -195 moveto
(11 UBSOT FSRSM  BLDEC IXQDZ  STPAJ YOMBO  CGIAD HUTHZ  JAURL MCRNY  ) show
0 -210 moveto
(12 VPVSF RSXSQ  UMOMQ KGJCY  BIJXE QQDFF  BHOGG RCTXO  KJRMC QINMZ  ) show
0 -225 moveto
(13 BEFIF UIEVB  UGWZJ ENOLD  CTNBD JWKCR  JTIKX DIRYK  WUZPH HGTAU  ) show
0 -240 moveto
(14 IPAOV IHDQA  VENRD RCFLR  ZRTOW NIHWY  TGWQX WWRHY  QSBMD IWSMQ  ) show
0 -270 moveto
(15 IYCMO WUPZM  OFPJW ZUUDB  QZIHP RNQTT  UMTDS ANATC  DANSQ YUYGT  ) show
0 -285 moveto
(16 XYCZU AKGWK  ZPEMS TAQLE  KKIYG UANUN  ZNTPP NZVTX  KPZCD BUYMT  ) show
0 -300 moveto
(17 DKZSJ SIMPP  EAFSS KNBOO  OATKY UKMYP  NZPGS ELIHP  JWWAQ ILRBV  ) show
0 -315 moveto
(18 RSBTM XROFG  MPGJT HIEIZ  XRSNN FIRYQ  AFGGR BYVQY  RXCFV CBBUO  ) show
0 -330 moveto
(19 ZTRPC HHVJO  BPXHL UGNLP  CLVUN DTGZI  NTCBM EDBRG  ZQUOY PFZHO  ) show
}def
gsave 30 750 translate onepad grestore
gsave 30 360 translate onepad grestore
```

```
10 390 moveto
(============t=e=a=r===h=e=r=e==============t=e=a=r===h=e=r=e==========) show
showpage
```

Here is the page that will be generated by the above Postscript program (shrunk to 60% of normal size).

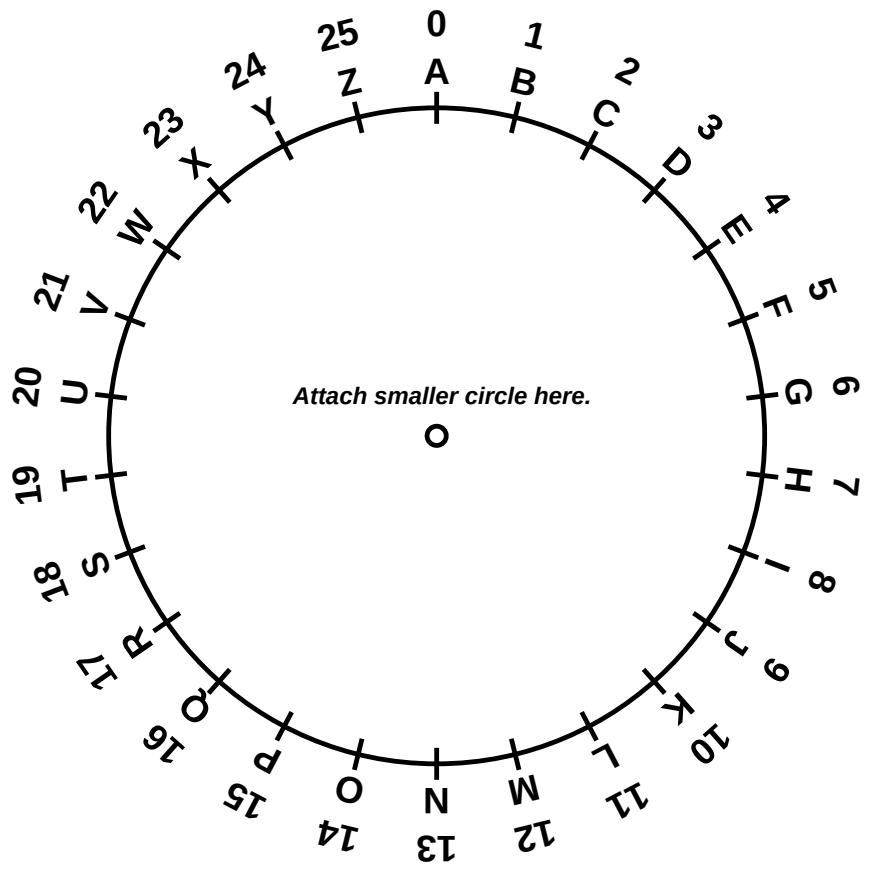
```
   00 XLCWT HZZTC  HUTXA GQAUN  FXCUI QFBVW  DKAPS SXKHK  XBLLP LTHFO
   01 XUMVS SRMGB  SPSDI UAFYO  CQYHQ CYSHU  UCATL HLDKZ  XWFGR LRMOL
   02 DXCBR FMARY  MOLUR CDJVT  MACWT IRFVP  DZBBY ZTFZG  HCUUL YYKPF
   03 QSWIH HNWRC  FQQHM ZEKJP  URILH THNSY  TLMWA CWYJV  AWOWY IAELZ
   04 KJKJE NGKDD  NYMYP NRNHK  KRIOX TIFSY  TGYRS HZLTQ  MCEBA YRBTU

   05 CZLFW HYSTM  YGDBN GVAZE  RFCQY YFHLC  OGLHY OEBNB  QDQIS EHIBB
   06 ZIWQC TYERS  UZGSS EGOVX  KGXJV AONYN  HXZCR RLUSN  SXXBR EJKQH
   07 OIJMF LCZHA  CGMYK LPLMU  CIMHZ WBVHD  JXNZJ CJLSN  PJVFL HENCU
   08 ZOHEW JRWPC  BHFRZ MEYBW  SHFPQ DYQFH  ARDQI UAHOC  OAQKR LZPBF
   09 YAKMA YJVYY  HSAZB SMILL  WMHAJ KMYNO  YHTXY GIISP  CVYQC OMCIP

   10 NPNVY VACCU  YWOFW GLHTO  TOUPS LRLOV  SZWBL WYNJO  NOHCZ SZSZS
   11 UBSOT FSRSM  BLDEC IXQDZ  STPAJ YOMBO  CGIAD HUTHZ  JAURL MCRNY
   12 VPVSF RSXSQ  UMOMQ KGJCY  BIJXE QQDFF  BHOGG RCTXO  KJRMC QINMZ
   13 BEFIF UIEVB  UGWZJ ENOLD  CTNBD JWKCR  JTIKX DIRYK  WUZPH HGTAU
   14 IPAOV IHDQA  VENRD RCFLR  ZRTOW NIHWY  TGWQX WWRHY  QSBMD IWSMQ

   15 IYCMO WUPZM  OFPJW ZUUDB  QZIHP RNQTT  UMTDS ANATC  DANSQ YUYGT
   16 XYCZU AKGWK  ZPEMS TAQLE  KKIYG UANUN  ZNTPP NZVTX  KPZCD BUYMT
   17 DKZSJ SIMPP  EAFSS KNBOO  OATKY UKMYP  NZPGS ELIHP  JWWAQ ILRBV
   18 RSBTM XROFG  MPGJT HIEIZ  XRSNN FIRYQ  AFGGR BYVQY  RXCFV CBBUO
   19 ZTRPC HHVJO  BPXHL UGNLP  CLVUN DTGZI  NTCBM EDBRG  ZQUOY PFZHO

============t=e=a=r===h=e=r=e==============t=e=a=r===h=e=r=e==========

   00 XLCWT HZZTC  HUTXA GQAUN  FXCUI QFBVW  DKAPS SXKHK  XBLLP LTHFO
   01 XUMVS SRMGB  SPSDI UAFYO  CQYHQ CYSHU  UCATL HLDKZ  XWFGR LRMOL
   02 DXCBR FMARY  MOLUR CDJVT  MACWT IRFVP  DZBBY ZTFZG  HCUUL YYKPF
   03 QSWIH HNWRC  FQQHM ZEKJP  URILH THNSY  TLMWA CWYJV  AWOWY IAELZ
   04 KJKJE NGKDD  NYMYP NRNHK  KRIOX TIFSY  TGYRS HZLTQ  MCEBA YRBTU

   05 CZLFW HYSTM  YGDBN GVAZE  RFCQY YFHLC  OGLHY OEBNB  QDQIS EHIBB
   06 ZIWQC TYERS  UZGSS EGOVX  KGXJV AONYN  HXZCR RLUSN  SXXBR EJKQH
   07 OIJMF LCZHA  CGMYK LPLMU  CIMHZ WBVHD  JXNZJ CJLSN  PJVFL HENCU
   08 ZOHEW JRWPC  BHFRZ MEYBW  SHFPQ DYQFH  ARDQI UAHOC  OAQKR LZPBF
   09 YAKMA YJVYY  HSAZB SMILL  WMHAJ KMYNO  YHTXY GIISP  CVYQC OMCIP

   10 NPNVY VACCU  YWOFW GLHTO  TOUPS LRLOV  SZWBL WYNJO  NOHCZ SZSZS
   11 UBSOT FSRSM  BLDEC IXQDZ  STPAJ YOMBO  CGIAD HUTHZ  JAURL MCRNY
   12 VPVSF RSXSQ  UMOMQ KGJCY  BIJXE QQDFF  BHOGG RCTXO  KJRMC QINMZ
   13 BEFIF UIEVB  UGWZJ ENOLD  CTNBD JWKCR  JTIKX DIRYK  WUZPH HGTAU
   14 IPAOV IHDQA  VENRD RCFLR  ZRTOW NIHWY  TGWQX WWRHY  QSBMD IWSMQ

   15 IYCMO WUPZM  OFPJW ZUUDB  QZIHP RNQTT  UMTDS ANATC  DANSQ YUYGT
   16 XYCZU AKGWK  ZPEMS TAQLE  KKIYG UANUN  ZNTPP NZVTX  KPZCD BUYMT
   17 DKZSJ SIMPP  EAFSS KNBOO  OATKY UKMYP  NZPGS ELIHP  JWWAQ ILRBV
   18 RSBTM XROFG  MPGJT HIEIZ  XRSNN FIRYQ  AFGGR BYVQY  RXCFV CBBUO
   19 ZTRPC HHVJO  BPXHL UGNLP  CLVUN DTGZI  NTCBM EDBRG  ZQUOY PFZHO
```

Program III.10.d
Circles for a One-Time Pad

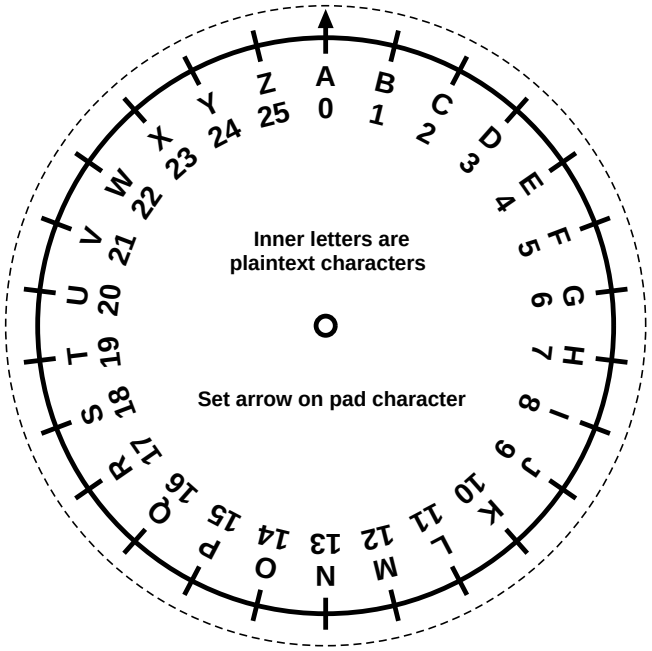
Referred to from page 69.

Here are two circles for use in creating a rotating tool for making use of one-time pad characters. Each image has been shrunk by 60%.

Outer letters belong to the ciphertext or to the pad.



Place arrow of inner circle on pad character.
To encrypt, follow inner circle plaintext letter to outer circle ciphertext letter.
To decrypt, follow outer circle ciphertext letter to inner circle plaintext letter.



(Cut along dashed line.)

Here is Postscript source to create the larger circle (at full size):

```
%!PS-Adobe-2.0
/r 360 26 div def
/inch { 72 mul} def
/Tempstr 2 string def
/radius 225 def
/circleofLetters {
 [(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)]
  /ary exch def % the array of letters

  0 1 25{ % from 0 to 25 in steps of 1
    /ind exch def % the for loop value
    /rind ind 0 eq {0} {26 ind sub} ifelse def
    gsave
      ind r mul neg rotate % rotate by (for loop value)*360/26
      /Helvetica-Bold findfont 23 scalefont setfont
      ary ind get stringwidth pop 2 div neg radius 5 sub moveto
      ary ind get show
      % convert ind to string, store in Tempstr, using cvs
      ind Tempstr cvs stringwidth pop 2 div neg radius 25 add moveto
      ind Tempstr cvs show
      3 setlinewidth
      0 radius 30 sub moveto 0 20 rlineto stroke
    grestore
  } for
}def
/circles {
   3 setlinewidth
   newpath 0 0 radius 20 sub 0 360 arc stroke
   newpath 0 0 6 0 360 arc stroke
} def
/Helvetica-Bold findfont 15 scalefont setfont
40 80 moveto (Place arrow of inner circle on pad character.) show
/Helvetica-Bold findfont 15 scalefont setfont
40 60 moveto (To encrypt, follow inner circle plaintext) show
( letter to outer circle ciphertext letter.) show
/Helvetica-BoldOblique findfont 15 scalefont setfont
40 40 moveto (To decrypt, follow outer circle ciphertext) show
( letterto inner circle plaintext letter.) show
/Helvetica-BoldOblique findfont 15 scalefont setfont
130 700 moveto (Outer letters belong to the ciphertext or to the pad.) show

8.5 inch 2 div 11 inch 2 div translate
-90 20 moveto (Attach smaller circle here.) show
circleofLetters  circles  showpage
```

Here is Postscript source to create the smaller circle (at full size):

```
%!PS-Adobe-2.0
/r 360 26 div def
/inch { 72 mul} def
/Tempstr 2 string def
/Helvetica-Bold findfont 18 scalefont setfont
/radius 120 def
/circleofLetters {
 [(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)]
  /ary exch def % the array of letters

  0 1 25{ % from 0 to 25 in steps of 1
    /ind exch def % the for loop value
    gsave
      ind r mul neg rotate % rotate by (for loop value)*360/26
      ary ind get stringwidth pop 2 div neg radius 30 add moveto
      ary ind get show
      % convert ind to string, store in Tempstr, using cvs
      ind Tempstr cvs stringwidth pop 2 div neg radius 10 add moveto
      ind Tempstr cvs show
      3 setlinewidth
      0 radius 50 add moveto 0 20 rlineto stroke
    grestore
  } for
}def
/circles {
   3 setlinewidth
   0 0 radius 60 add 0 360 arc stroke
   0 0 6 0 360 arc stroke
   1 setlinewidth
   0 0 radius 80 add 0 360 arc [5 3] 0 setdash stroke
} def
/arrowhead {
  newpath 0 radius 78 add moveto
  -5 -12 rlineto 10 0 rlineto closepath fill
} def

8.5 inch 2 div 11 inch 2 div translate
circleofLetters  circles  arrowhead
/Helvetica-Bold findfont 13 scalefont setfont
-45 50 moveto (Inner letters are) show
-60 35 moveto (plaintext characters) show
-80 -50 moveto (Set arrow on pad character) show
-75 radius 100 add neg moveto ((Cut along dashed line.))show
showpage
```

Program IV.14.a
RSA Implementation

Referred to from page 90.

This Java implementation of the basic RSA cryptosystem uses the Java **`BigInteger`** library class. This is just a "skeleton" implementation that creates keys from scratch and uses them, but does not save keys to a file for repeated use, or fetch such keys from the file. For further comments about the implementation, see the chapter on the RSA cryptosystem.
This code implements RSA using 3 Java classes:

- **`RSAPublicKey`**: The data and methods needed for RSA public keys, with the modulus n and exponent e, along with a username to keep the keys straight. The important methods are encryption and verification.
- **`RSAPrivateKey`**: This extends the previous class to add the primes p and q, and the decryption exponent e as data members. Important methods include decryption and signing, along with key generation.
- **`RSATest`**: A class to test out the system with realistic key sizes (1024 bits).

Java class: RSAPublicKey

```
// RSAPublicKey: RSA public key
import java.math.*; // for BigInteger
public class RSAPublicKey {
   public BigInteger n; // public modulus
   public BigInteger e = new BigInteger("3"); // encryption exponent
   public String userName; // attach name to each public/private key pair

   public RSAPublicKey(String name) {
      userName = name;
   }

   // setN: to give n a value in case only have public key
   public void setN(BigInteger newN) {
      n = newN;
   }

   // getN: provide n
   public BigInteger getN() {
      return n;
   }

   // RSAEncrypt: just raise m to power e (3) mod n
   public BigInteger RSAEncrypt(BigInteger m) {
      return m.modPow(e, n);
   }
```

```
   // RSAVerify: same as encryption, since RSA is symmetric
   public BigInteger RSAVerify(BigInteger s) {
      return s.modPow(e, n);
   }
}
```

Java class: RSAPrivateKey

```
// RSAPrivateKey: RSA private key
import java.math.*; // for BigInteger
import java.util.*; // for Random
public class RSAPrivateKey extends RSAPublicKey{
   private final BigInteger TWO = new BigInteger("2");
   private final BigInteger THREE = new BigInteger("3");

   private BigInteger p; // first prime
   private BigInteger q; // second prime
   private BigInteger d; // decryption exponent

   public RSAPrivateKey(int size, Random rnd, String name) {
      super(name); generateKeyPair(size, rnd);
   }

   public void generateKeyPair(int size, Random rnd) { // size = n (bits)
      // want sizes of primes close, not too close (10-20 bits).
      int size1 = size/2;
      int size2 = size1;
      int offset1 = (int)(5.0*(rnd.nextDouble()) + 5.0);
      int offset2 = -offset1;
      if (rnd.nextDouble() < 0.5) {
         offset1 = -offset1; offset2 = -offset2;
      }
      size1 += offset1; size2 += offset2;
      // generate two random primes, so that p*q = n has size bits
      BigInteger p1   = new BigInteger(size1, rnd); // random int
                 p    = nextPrime(p1);
      BigInteger pM1  = p.subtract(BigInteger.ONE);
      BigInteger q1   = new BigInteger(size2, rnd);
                 q    = nextPrime(q1);
      BigInteger qM1  = q.subtract(BigInteger.ONE);
                 n    = p.multiply(q);
      BigInteger phiN = pM1.multiply(qM1); // (p-1)*(q-1)
      BigInteger e = THREE;
                 d    = e.modInverse(phiN);
   }

   // nextPrime: next prime p after x, with p-1 and 3 relatively prime
   public BigInteger nextPrime(BigInteger x) {
      if ((x.remainder(TWO)).equals(BigInteger.ZERO))
         x = x.add(BigInteger.ONE);
      while(true) {
         BigInteger xM1 = x.subtract(BigInteger.ONE);
         if (!(xM1.remainder(THREE)).equals(BigInteger.ZERO))
            if (x.isProbablePrime(10)) break;
```

```
            x = x.add(TWO);
        }
        return x;
    }

    // RSADecrypt: decryption function
    public BigInteger RSADecrypt(BigInteger c) {
        return c.modPow(d, n);
    }

    // RSASign: same as decryption for RSA (since it is a symmetric PKC)
    public BigInteger RSASign(BigInteger m) {
        return m.modPow(d, n);
    }

    public BigInteger RSASignAndEncrypt(BigInteger m, RSAPublicKey other) {
        // two ways to go, depending on sizes of n and other.getN()
        if (n.compareTo(other.getN()) > 0)
            return RSASign(other.RSAEncrypt(m));
        else
            return other.RSAEncrypt(RSASign(m));
    }

    public BigInteger RSADecryptAndVerify(BigInteger c,
                                          RSAPrivateKey other) {
        // two ways to go, depending on sizes of n and other.getN()
        if (n.compareTo(other.getN()) > 0)
            return other.RSAVerify(RSADecrypt(c));
        else
            return RSADecrypt(other.RSAVerify(c));
    }
}
```

Java class: RSATest

```
// RSATest: Test RSA Implementation
import java.math.*; // for BigInteger
import java.util.*; // for Random
public class RSATest {

    public static void main(String[] args) {
        Random rnd = new Random();
        BigInteger m, m1, m2, m3, c, s, s1;
        RSAPrivateKey alice = new RSAPrivateKey(1024, rnd, "Alice");
        RSAPrivateKey bob   = new RSAPrivateKey(1024, rnd, "Bob  ");
        m = new BigInteger(
          "12345678909876543210123456789098765432l0" +
          "12345678909876543210123456789098765432l0" +
          "12345678909876543210123456789098765432l0" +
          "12345678909876543210123456789098765432l0" +
          "12345678909876543210123456789098765432l0" +
          "12345678909876543210123456789098765432l0");
        System.out.println("Message m:\n" + m + "\n");
        System.out.println("ALICE ENCRYPTS m FOR BOB; BOB DECRYPTS IT:");
```

```
        c = bob.RSAEncrypt(m); // Using Bob's public key
        System.out.println("Message encrypted with Bob's public key:\n" +
           c + "\n");
        m1 = bob.RSADecrypt(c); // Using Bob's private key
        System.out.println("Original message back, decrypted:\n" +
           m1 + "\n");

        System.out.println("ALICE SIGNS m FOR BOB; BOB VERIFIES SIGNATURE:");
        s = alice.RSASign(m); // Using Alice's private key
        System.out.println("Message signed with Alice's private key:\n" +
           c + "\n");
        m2 = alice.RSAVerify(s); // Using Alice's public key
        System.out.println("Original message back, verified:\n" +
           m2 + "\n");

        System.out.println("BOB SIGNS AND ENCRYPTS m FOR ALICE;" +
           "\n   ALICE VERIFIES SIGNATURE AND DECRYPTS:");
        c  = bob.RSASignAndEncrypt(m, alice);
        System.out.println("Message signed and encrypted," +
           "\n  using Bob's secret key and Alice's public key:\n" +
           c + "\n");
        m3 = alice.RSADecryptAndVerify(c, bob);
        System.out.println("Original message back, verified and decrypted," +
           "\n  using Alice's secret key and Bob's public key:\n" + m1);
    }
}
```

A Test Run.

Here is a run of the above test class, showing simple encryption, signing, and a combination of signing and encryption. Unix commands appear in **boldface**.

```
% javac RSAPublicKey.java
% javac RSAPrivateKey.java
% javac RSATest.java
% java RSATest
Message m:
123456789098765432101234567890987654321012345678909876543210
123456789098765432101234567890987654321012345678909876543210
123456789098765432101234567890987654321012345678909876543210
123456789098765432101234567890987654321012345678909876543210

ALICE ENCRYPTS m FOR BOB; BOB DECRYPTS IT:
Message encrypted with Bob's public key:
623387565362752740557713183298294394842904981992063743592594
444564441837460636112777656456876530809960075397677835706720
975503107091512394844823973429619412227989318053859609889705
833638590603829414072912488421444136560245226367742777088035
320797857638669726447023121838563030894198617138062884887534
61181488

Original message back, decrypted:
123456789098765432101234567890987654321012345678909876543210
```

```
12345678909876543210123456789098765432101234567890987654321
12345678909876543210123456789098765432101234567890987654321
12345678909876543210123456789098765432101234567890987654321

ALICE SIGNS m FOR BOB; BOB VERIFIES SIGNATURE:
Message signed with Alice's private key:
43937218657097546876935199771978159837318201212446313948230
48969021053734725233701441035596141299351045669267129491245
27301613351264122145743822642815234624613789843360005067184
82036781895678243991158862217920299328066514576707842515867
47748781553219047207889000050867990141337788684336511130898
31991525

Original message back, verified:
12345678909876543210123456789098765432101234567890987654321
12345678909876543210123456789098765432101234567890987654321
12345678909876543210123456789098765432101234567890987654321
12345678909876543210123456789098765432101234567890987654321

BOB SIGNS AND ENCRYPTS m FOR ALICE;
   ALICE VERIFIES SIGNATURE AND DECRYPTS:
Message signed and encrypted,
  using Bob's secret key and Alice's public key:
27334368604128703558213193949827019828348248292596875681512
46086839430318466863049866432840181599919878918036067906871
15859154381075648385363993421653018918793076693023041089609
62581152691427815441272294921259088510237350977263534672355
05368973730250834795504007563891922299697420556823064897186
74601320

Original message back, verified and decrypted,
  using Alice's secret key and Bob's public key:
12345678909876543210123456789098765432101234567890987654321
12345678909876543210123456789098765432101234567890987654321
12345678909876543210123456789098765432101234567890987654321
12345678909876543210123456789098765432101234567890987654321
```

Program IV.14.b
Faster RSA, Using Chinese Remainder Theorem

Referred to from page 92.

Here is an altered implementation of the RSA cryptosystem, using the the Chinese Remainder Theorem (CRT) to speed up decryption. Please refer first to the basic RSA as described in the main RSA section and then to the description of this version of RSA. Additions and changes related to the faster implementation are highlighted in **boldface**.

Java class: RSAPublicKey

```
// RSAPublicKey: RSA public key
//  See the listing in the previous section for this class.
```

Java class: RSAPrivateKeyFast

```
// RSAPrivateKeyFast: RSA private key, using fast CRT algorithm
import java.math.*; // for BigInteger
import java.util.*; // for Random
public class RSAPrivateKeyFast extends RSAPublicKey{
   private final BigInteger TWO = new BigInteger("2");
   private final BigInteger THREE = new BigInteger("3");

   private BigInteger p; // first prime
   private BigInteger q; // second prime
   private BigInteger d; // decryption exponent

   private BigInteger p1, pM1, q1, qM1, phiN; // for key generation
   private BigInteger dp, dq, c2; // for fast decryption

   public RSAPrivateKeyFast(int size, Random rnd, String name) {
      super(name); generateKeyPair(size, rnd);
   }

   public void generateKeyPair(int size, Random rnd) { // size = n
      // want sizes of primes close, not too close, 10-20 bits apart.
      int size1 = size/2;
      int size2 = size1;
      int offset1 = (int)(5.0*(rnd.nextDouble()) + 5.0);
      int offset2 = -offset1;
      if (rnd.nextDouble() < 0.5) {
         offset1 = -offset1; offset2 = -offset2;
      }
      size1 += offset1; size2 += offset2;
      // generate two random primes, so that p*q = n has size bits
      p1   = new BigInteger(size1, rnd); // random int
      p    = nextPrime(p1);
      pM1  = p.subtract(BigInteger.ONE);
      q1   = new BigInteger(size2, rnd);
```

```
      q    = nextPrime(q1);
      qM1  = q.subtract(BigInteger.ONE);
      n    = p.multiply(q);
      phiN = pM1.multiply(qM1); // (p-1)*(q-1)
      d    = e.modInverse(phiN);
      // remaining stuff needed for fast CRT decryption
      dp = d.remainder(pM1);
      dq = d.remainder(qM1);
      c2 = p.modInverse(q);
   }

   // nextPrime: next prime p after x, with p-1 and 3 rel prime
   public BigInteger nextPrime(BigInteger x) {
      if ((x.remainder(TWO)).equals(BigInteger.ZERO))
         x = x.add(BigInteger.ONE);
      while(true) {
         BigInteger xM1 = x.subtract(BigInteger.ONE);
         if (!(xM1.remainder(THREE)).equals(BigInteger.ZERO))
            if (x.isProbablePrime(10)) break;
         x = x.add(TWO);
      }
      return x;
   }

   // RSADecrypt: decryption function, <b>fast CRT version</b>
   public BigInteger RSADecrypt(BigInteger c) {
      // See 14.71 and 14.75 in Handbook of Applied Cryptography,
      //    by Menezes, van Oorschot and Vanstone
      BigInteger cDp = c.modPow(dp, p);
      BigInteger cDq = c.modPow(dq, q);
      BigInteger u = ((cDq.subtract(cDp)).multiply(c2)).remainder(q);
      if (u.compareTo(BigInteger.ZERO) < 0) u = u.add(q);
      return cDp.add(u.multiply(p));
   }

   // RSASign: same as decryption for RSA (since it is a symmetric PKC)
   public BigInteger RSASign(BigInteger m) {
      // return m.modPow(d, n);
     return RSADecrypt(m); // use fast CRT version
   }

   public BigInteger RSASignAndEncrypt(BigInteger m,
            RSAPublicKey other) {
      // two ways to go, depending on sizes of n and other.getN()
      if (n.compareTo(other.getN()) > 0)
          return RSASign(other.RSAEncrypt(m));
      else
          return other.RSAEncrypt(RSASign(m));
   }

   public BigInteger RSADecryptAndVerify(BigInteger c,
```

```
                                          RSAPrivateKeyFast other) {
      // two ways to go, depending on sizes of n and other.getN()
      if (n.compareTo(other.getN()) > 0)
          return other.RSAVerify(RSADecrypt(c));
      else
          return RSADecrypt(other.RSAVerify(c));
   }
}
```

Java class: RSATestFast

```
// RSATestFast: Test Fast RSA Implementation
import java.math.*; // for BigInteger
import java.util.*; // for Random
public class RSATestFast {
   public static void main(String[] args) {
      Random rnd = new Random();
      BigInteger m, m1, m2, m3, c, s, s1;
      RSAPrivateKeyFast alice = new RSAPrivateKeyFast(1024, rnd, "Alice");
      RSAPrivateKeyFast bob   = new RSAPrivateKeyFast(1024, rnd, "Bob  ");
      m = new BigInteger(
        "12345678909876543210123456789098765432l0" +
        "12345678909876543210123456789098765432l0" +
        "12345678909876543210123456789098765432l0" +
        "12345678909876543210123456789098765432l0" +
        "12345678909876543210123456789098765432l0" +
        "12345678909876543210123456789098765432l0");
      System.out.println("Message m:\n" + m + "\n");
      System.out.println("ALICE ENCRYPTS m FOR BOB; BOB DECRYPTS IT:");
      c = bob.RSAEncrypt(m); // Using Bob's public key
      System.out.println("Message encrypted with Bob's public key:\n" +
         c + "\n");
      m1 = bob.RSADecrypt(c); // Using Bob's private key
      System.out.println("Original message back, decrypted:\n" + m1 +"\n");

      System.out.println("ALICE SIGNS m FOR BOB; BOB VERIFIES SIGNATURE:");
      s = alice.RSASign(m); // Using Alice's private key
      System.out.println("Message signed with Alice's private key:\n" +
         s + "\n");
      m2 = alice.RSAVerify(s); // Using Alice's public key
      System.out.println("Original message back, verified:\n" + m2 + "\n");

      System.out.println("BOB SIGNS AND ENCRYPTS m FOR ALICE;" +
         "\n   ALICE VERIFIES SIGNATURE AND DECRYPTS:");
      c  = bob.RSASignAndEncrypt(m, alice);
      System.out.println("Message signed and encrypted," +
         "\n  using Bob's secret key and Alice's public key:\n" + c +"\n");
      m3 = alice.RSADecryptAndVerify(c, bob);
      System.out.println("Original message back, verified and decrypted," +
         "\n  using Alice's secret key and Bob's public key:\n" + m1);
   }
}
```

Here is a run of the above test class, showing simple encryption, signing, and a combination of

signing and encryption. Unix commands appear in **boldface**.

```
% javac RSAPublicKey.java
% javac RSAPrivateKey.java
% javac RSATest.java
% java RSATest
Message m:
123456789098765432101234567890987654321012345678909876543210
123456789098765432101234567890987654321012345678909876543210
123456789098765432101234567890987654321012345678909876543210
123456789098765432101234567890987654321012345678909876543210

ALICE ENCRYPTS m FOR BOB; BOB DECRYPTS IT:
Message encrypted with Bob's public key:
543405813676648078057012762872599683813667674133659925377335
760556755516424469233387398561035220096421942902314004442496
355392009986359056374479092883194576861821720618133177330634
484625941715294402963142587566926665244387837038418691448876
173245292324151150663861262596533907168126172311922973506760
70135287

Original message back, decrypted:
123456789098765432101234567890987654321012345678909876543210
123456789098765432101234567890987654321012345678909876543210
123456789098765432101234567890987654321012345678909876543210
123456789098765432101234567890987654321012345678909876543210

ALICE SIGNS m FOR BOB; BOB VERIFIES SIGNATURE:
Message signed with Alice's private key:
239990627092163586938360727219071875855725965597290038843626
784334056744376101809741282946428993573655987183640986372900
356678910437032277772334474986578993935720568974198358713462
782149869678768897151584050391219800123956436243445248715199
025995371266867400947136422789069497185692715034294109803570
5104040

Original message back, verified:
123456789098765432101234567890987654321012345678909876543210
123456789098765432101234567890987654321012345678909876543210
123456789098765432101234567890987654321012345678909876543210
123456789098765432101234567890987654321012345678909876543210

BOB SIGNS AND ENCRYPTS m FOR ALICE;
   ALICE VERIFIES SIGNATURE AND DECRYPTS:
Message signed and encrypted,
  using Bob's secret key and Alice's public key:
555680954489228451633956186412450975924427391258695282224282
350607993390891939181686306232760912706003539593775370490376
870445903174464182907612502285232696602221467528497111242219
800301035480234847470533403244513111604794010697819018320289
165817224833283798363570908599851775688615057167242160604046
11712970
```

```
Original message back, verified and decrypted,
  using Alice's secret key and Bob's public key:
12345678909876543210123456789098765432101234567890987654321
12345678909876543210123456789098765432101234567890987654321
12345678909876543210123456789098765432101234567890987654321
12345678909876543210123456789098765432101234567890987654321
```

Program IV.15.a
Square Roots mod n = p*q

Referred to from page 94.

Here is a program that prints a table of square roots modulo a product of two primes Rabin's cryptosystem makes use of square roots modulo **n = p*q**, where **p** and **q** are primes both equal to **3** modulo **4**. The program below, however, works with any two primes and produces a table of all square roots. Those square roots with a factor in common with either **p** or **q** are shown in the table in bold italic.

Java class: SquareTable

```
// SquareTable.java: create table of squares and square roots
public class SquareTable {

   // Link: to allow linked lists of square roots
   private class Link {
      int entry; // square root entry
      Link next; // next link
   }

   private Link[] table; // table entry for each possible square
   private int p, q, n; // primes and product

   // SquareTable: constructor, passes in the primes
   public SquareTable(int pIn, int qIn) {
      p = pIn; q = qIn; n = p*q;
      table = new Link[n];
   }

   // buildTable: construct the table: array of linked lists
   public void buildTable() {
      for (int i = 1; i < n; i++) {
         int iSqr = i*i % n;
         Link link = new Link();
         Link ptr;
         if (table[iSqr] == null) {
            table[iSqr] = link;
         }
         else {
            ptr = table[iSqr];
            while (ptr.next != null)
               ptr = ptr.next;
            ptr.next = link;
         }
         link.entry = i;
         link.next = null;
      }
   }
```

```
   // printTable: print the table in HTML form
   public void printTable() {
      boolean div; // for entries with either prime as divisor
      System.out.println("<table border>");
      System.out.println("<tr><th colspan=2>Numbers mod " + n +
         " = " + p + "*" + q + "</th><tr>");
      System.out.println("<tr><th>Square</th>");
      System.out.println("<th>Square Roots</th></tr>");
      System.out.println("<tr><td></td><td></td></tr>");

      for (int j = 1; j < n; j++) {
         div = false;
         if (j%p == 0 || j%q == 0) div = true;
         if (table[j] != null) {
            System.out.print("<tr><td>");
            if (div)
               System.out.print("<font color=FF0000><i>" +
                  j + "</i></font>");
            else
               System.out.print(j);
            System.out.print("</td><td>");
            if (div)
               System.out.print("<font color=FF0000><i>");
            Link loc = table[j];
            while (loc != null) {
               System.out.print(loc.entry);
               if (loc.next != null)
                  System.out.print(", ");
               loc = loc.next;
            }
            if (div)
               System.out.print("</i></font>");
            System.out.println("</td></tr>");
         }
      }
      System.out.println("</table>");
   }

   // main: feed in primes p and q from command line
   public static void main(String[] args) {
      SquareTable squareTable = new SquareTable(
          Integer.parseInt(args[0]), Integer.parseInt(args[1]));
      squareTable.buildTable();
      squareTable.printTable();
   }
}
```

Here is a table with **`p = 7`**, **`q = 11`**, and **`n = p*q = 77`**. Notice that the bold italic entries all have either **`7`** or **`11`** as a factor. Notice also the symmetry: if **`s`** is a square root, then so is **`n - s`**.

Numbers mod 77 = 7*11	
Square	Square Roots
1	1, 34, 43, 76
4	2, 9, 68, 75
9	3, 25, 52, 74
11	***33, 44***
14	***28, 49***
15	13, 20, 57, 64
16	4, 18, 59, 73
22	***22, 55***
23	10, 32, 45, 67
25	5, 16, 61, 72
36	6, 27, 50, 71
37	24, 31, 46, 53
42	***14, 63***
44	***11, 66***
49	***7, 70***
53	19, 30, 47, 58
56	***21, 56***
58	17, 38, 39, 60
60	26, 37, 40, 51
64	8, 36, 41, 69
67	12, 23, 54, 65
70	***35, 42***
71	15, 29, 48, 62

Program V.16.a
Linear Congruence Random Number Generators

Referred to from page 103.

This section contains an implementation of several simple linear congruence random number generators, using the Java **`BigInteger`** class. This makes the generators slow, but eliminates any overflow problems. In practice this method is fast enough for most applications.
The class **`Congruence`** below implements a specific generator with input multiplier, modulus, and seed. The the class **`Generators`** creates 8 instances of generators with different values for the multiplier and modulus. Notice that generator number 0 is the infamous "RANDU" which should not be used.

Java class: Congruence

```
// Congruence: linear congruence generators, all using BigInteger
import java.math.*; // for BigInteger
public class Congruence {
   public  BigInteger k; // multiplier
   public  BigInteger m; // modulus
   public  BigInteger s; // seed
   private BigInteger x; // next generator value
   private int rBits;
   private int twoToRBits; // 2^rBits
   private BigInteger bigTwoToRBits;

   // Congruence: constructor starts with multiplier, modulus and seed
   public Congruence(String ks, String ms, String ss) {
      k = new BigInteger(ks);
      m = new BigInteger(ms);
      s = new BigInteger(ss);
      x = s;
      // System.out.println("k: " + k + ", m: " + m + ", s: " + s);
   }

   // nextValue: cycle to the next BigInteger value and return it
   public BigInteger nextValue() {
      x = (k.multiply(x)).mod(m);
      return x;
   }

   // doubleValue: return x/m as a double
   public double doubleValue() {
      return (x.doubleValue()) / (m.doubleValue());
   }
}
```

Java class: Generators

```
// Generators.java: a variety of random number generators
```

```
import java.math.*; // for BigInteger

public class Generators {
   public Congruence cong[] = new Congruence[8]; // linear congruence RNGs

   // Generators: construct 8 linear congruence generators at once
   public Generators(String seed) {
      cong[0] = new Congruence("65539",
         "2147483648", seed); // m = 2^31
      cong[1] = new Congruence("16807",
         "2147483647", seed); // m = 2^31 - 1
      cong[2] = new Congruence("40692",
         "2147483399", seed); // m = 2^31 - 249
      cong[3] = new Congruence("48271",
         "2147483647", seed); // m = 2^31 - 1
      cong[4] = new Congruence("62089911",
         "2147483647", seed); // m = 2^31 - 1
      cong[5] = new Congruence("69069",
         "4294967296", seed); // m = 2^32
      cong[6] = new Congruence("31167285",
         "281474976710656", seed); // m = 2^48
      cong[7] = new Congruence("6364136223846793005",
        "18446744073709551616", seed); // m = 2^64
   }

   // nextValue: return next value of generator number i
   public BigInteger nextValue(int i) {
      return cong[i].nextValue();
   }

   // doubleValue: return double corresponding to value of gen i
   public double doubleValue(int i) {
      return cong[i].doubleValue();
   }

   // stuff below is just to demonstrate each generator
   public static void main(String[] args) {
      int iCong = Integer.parseInt(args[0]); // iCong = specific RNG
      String seed = args[1]; // seed for the RNG
      Generators gen = new Generators(seed);
      System.out.println("Generator: " + iCong +
                         ", k: " + gen.cong[iCong].k +
                         ", m: " + gen.cong[iCong].m +
                         ", s: " + gen.cong[iCong].s);
      for (int i = 0; i < 4; i++) {
         BigInteger x = gen.nextValue(iCong);
         System.out.println(x + " \t" + gen.doubleValue(iCong));
      }
   }
}
```

Here is brief output from each generator. More thorough testing appears in a later section.

```
% java Generators 0 11111
Generator: 0, k: 65539, m: 2147483648, s: 11111
728203829       0.33909633243456483
74155679        0.03453142894431949
333550557       0.15532158175483346
1333902231      0.6211466300301254
% java Generators 1 11111
Generator: 1, k: 16807, m: 2147483647, s: 11111
186742577       0.08695878884147797
1108883372      0.5163640587201175
1139744538      0.5307349090141873
132318926       0.06161580144502958
% java Generators 2 11111
Generator: 2, k: 40692, m: 2147483399, s: 11111
452128812       0.21053890903675387
535338671       0.24928652358816208
2077084275      0.9672178494917436
61700458        0.028731518031166862
% java Generators 3 11111
Generator: 3, k: 48271, m: 2147483647, s: 11111
536339081       0.24975234700820984
1708414366      0.7955424332970485
1350332739      0.6287976818293323
1487990525      0.6928995836958752
% java Generators 4 11111
Generator: 4, k: 62089911, m: 2147483647, s: 11111
538750434       0.2508752207508661
276008834       0.128526629008598
2053582492      0.9562738672626548
665642739       0.30996405487412776
% java Generators 5 11111
Generator: 5, k: 69069, m: 4294967296, s: 11111
767425659       0.17868021014146507
1131441535      0.26343426085077226
605430195       0.1409627019893378
656544599       0.15286370157264173
% java Generators 6 11111
Generator: 6, k: 31167285, m: 281474976710656, s: 11111
346299703635    0.001230303694068624
63576637476655  0.22586958961545278
245552537051579 0.8723778572473115
85713711193271  0.3045162742170895
% java Generators 7 11111
Generator: 7, k: 6364136223846793005, m: 18446744073709551616, s: 11111
5547548633005734427     0.3007332139936904
943960907862842303      0.05117222335231413
3205834541165224339     0.17378863870802033
1510653287195956183     0.08189267879251122
```

Program V.16.b
Exponential and Normal Distributions

Referred to from page 103.

This section demonstrates transformations from the uniform distribution to the exponential and the normal distributions. The version of uniform distribution implemented is Knuth's method with 2 seeds. An applet plots all three distributions, using 1000, 10000, 100000, and 1000000 points. Each plot uses 500 intervals over the range, although the exponential and normal distributions ignore the few points outside the displayed range. So the pictures at the end give 500 vertical lines that represent approximately 2, 20, 200, and 2000 trials. In each case the area covered by all the lines is approximately the same.

Java class: DistPlot

```
// DistPlot: plot 3 random distributions: exponential, uniform, normal
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
public class DistPlot extends Applet  implements ActionListener{
   int[] expC  = new int[1000]; // counter for nextExpDist
   int[] unifC = new int[500];  // counter for nextUniformDist
   int[] normC = new int[500];  // counter for nextNormalDist
   double scale; // scale factor for displaying distributions

   int xStart = 50, yStart = 150; int xSide = 500;
   boolean firstTime = true; // to paint axes the first time through
   Button next0, next1, next2, next3; // buttons
   int iter; // number of random points to plot

   public void init() {
      setBackground(Color.white);
      next0 = new Button("1000"); next1 = new Button("10000");
      next2 = new Button("100000"); next3 = new Button("1000000");
      next0.addActionListener(this); next1.addActionListener(this);
      next2.addActionListener(this); next3.addActionListener(this);
      add(next0); add(next1); add(next2); add(next3);
   }

   public void paint(Graphics g) {
      // for exponential distribution
      g.drawString("Graph showing exponential distribution " +
         "with average 1", xStart, 50);
      g.drawString("Iterations: " + iter, xStart + 350, 50);
      g.drawLine(xStart, yStart, xStart + xSide, yStart);
      for (int i = 0; i <= 5; i++)
         g.drawLine(xStart + i*xSide/5, yStart,
            xStart + i*xSide/5, yStart + 10);
      for (int i = 0; i <= 5; i++)
         g.drawString(i*1 + "", xStart + i*xSide/5 - 1, yStart + 23);
```

```
      // for uniform distribution
      g.drawString("Graph showing uniform distribution " +
         "from 0 to 1", xStart, 200);
      g.drawLine(xStart, yStart+150, xStart + xSide, yStart+150);
      for (int i = 0; i <= 10; i++)
         g.drawLine(xStart + i*xSide/10, yStart+150,
            xStart + i*xSide/10, yStart + 10 +150);
      for (int i = 0; i <= 9; i++)
         g.drawString("0." + i + "", xStart + i*xSide/10 - 1,
            yStart + 23 +150);
      g.drawString("1", xStart + 10*xSide/10 - 1,
         yStart + 23 +150);

      // for normal distribution
      g.drawString("Graph showing normal distribution " +
         "from 0 to 1", xStart, 350);
      g.drawLine(xStart, yStart+300, xStart + xSide, yStart+300);
      for (int i = 0; i <= 10; i++)
         g.drawLine(xStart + i*xSide/10, yStart+300,
                    xStart + i*xSide/10, yStart + 10 +300);
      for (int i = 0; i <= 10; i++)
         g.drawString(i-5 + "", xStart + i*xSide/10 - 1,
            yStart + 23 +300);
      firstTime = false;
      if (!firstTime) {
         for(int dummy = 0; dummy < iter; dummy++) {
            int i = (int)(100.0*nextExpDist());
            if (i < 1000) expC[i]++;
            else expC[999]++;
            unifC[(int)(500.0*nextUniformDist())]++;
            double r = nextNormalDist();
            int j =  (int)( 50.0*r) + 250;
            if (r < 0.0) j = (int)( 50.0*r) - 1 + 250;
            if (j < 0) normC[0]++;
            else if (j >= 500) normC[499]++;
            else normC[j]++;
         }
         // exponential distribution
         g.setColor(new Color(255, 0, 0)); // red
         for (int i = 0; i < 500; i++)
            g.drawLine(xStart + i, yStart - 1,
                       xStart + i, yStart - (int)(expC[i]/scale));
         // uniform distribution
         for (int i = 0; i < 500; i++)
            g.drawLine(xStart + i, yStart - 1 + 150,
                       xStart + i, yStart - (int)(unifC[i]/scale) + 150);
         // normal distribution
         for (int i = 0; i < 500; i++)
            g.drawLine(xStart + i, yStart - 1 + 300,
                       xStart + i, yStart - (int)(normC[i]/scale) + 300);
      }
   }
```

```
// class variables used by nextUniformDist
long seed1 = (int)(100000000.0*Math.random());
long seed2 = (int)(100000000.0*Math.random());
long seed3;
// nextUniform: uniformly dist on the interval from 0 to 1
private double nextUniformDist() {
   long m = 2147483647;
   long k1 = 2718283;
   long k2 = 314159269;
   seed3 = (k1*seed1 - k2*seed2)%m;
   if (seed3 < 0) seed3 += m;
   seed1 = seed2; seed2 = seed3;
   return (double)seed2 / (double)m;
}

// nextExpDist: exponentially dist with average interarrival time = 1
private double nextExpDist() {
   return 1.0*(-Math.log(nextUniformDist()));
}

// class variables used by nextNormalDist
boolean nextNormal = true;
double saveNormal;
// nextNormalDist: mean 0 and variance 1
//    x1 = sqrt(-2*log(u1))*cos(2*Pi*u2)
//    x2 = sqrt(-2*log(u1))*sin(2*Pi*u2)
private double nextNormalDist() {
   double u1, u2, x1, x2;
   if (nextNormal) {
      u1 = nextUniformDist(); u2 = nextUniformDist();
      double temp = Math.sqrt(-2.0*Math.log(u1));
      x1 = temp*Math.cos(2.0*Math.PI*u2);
      x2 = temp*Math.sin(2.0*Math.PI*u2);
      saveNormal = x2;
      nextNormal = false;
      return x1;
   }
   else {
      nextNormal = true;
      return saveNormal;
   }
}


public void actionPerformed(ActionEvent e) {
   for (int i = 0; i < 500; i++) {
      expC[i] = 0; expC[i + 500] = 0; unifC[i] = 0; normC[i] = 0;
   }
   if (e.getSource() == next0) {
      iter = 1000; scale = 0.1;
   }
   else if (e.getSource() == next1) {
      iter = 10000; scale = 1.0;
```

```
            }
            else if (e.getSource() == next2) {
               iter = 100000; scale = 10.0;
            }
            else if (e.getSource() == next3) {
               iter = 1000000; scale = 100.0;
            }
            repaint();
         }
      }
```

The output of the above program has not been included here because it takes up a lot of memory.

Program V.17.a
The logistic Lattice as a RNG

Referred to from page 110.

The program in this section uses the logistic lattice in two dimensions to return pseudo-random **`double`**s. Based on experiments described later in this section, this program sets the constants **`NU`** to 10^{-13} and **`NSTEP`** to 60. **`NSTEP`** is half the number of iterations, so the logistic equation will be iterated 120 times at each node. With this value of **`NU`**, the given number of iterations are sufficient to assure that each node will fill completely with noise and become independent of each other node by the time the code supplies more random numbers. The code below uses all 9 **`double`**s at each stage, thereby increasing efficiency by a factor of 9. (Each node will be independent of the other nodes.)

Java class: Chaotic2D

```
// Chaotic2D.java: a random number generator based on chaos theory
import java.util.*; // for Random
public class Chaotic2D {
   private final int NMAX = 3;          // size of lattice
   private final double BETA = 0.292893218813452476; // magic number in f
   private double  NU; // = 1.0e-13,    // viscosity constant in step
   private final double TWO_DIV_PI = 0.636619772367581343; // 2/Pi in S
   private int  NSTEP; // = 60,         // half # of steps to iterate
   private int flag;   // flag used in nextBlock
   private double[][] t;    // seed array, where the work occurs
   private double[][] tn;   // extra copy of seed array
   private double[][] tret; // array for returning values (these are
                            //   transformed to the uniform distribution)
   private Random random;   // extra RNG to initialize seed array t

   // mod: need instead of %, because % can yield negative result
   private int mod(int i, int j) {     // If i is negative, then i%j
      int k = i%j;                     // may be negative.  In general,
      if (k < 0)  k = k + j;           // result is machine dependent.
      return(k);                       // Check your own architecture.
   }

   // Chaotic2D: constructor -- allocate, use seed and auxiliary RNG
   //   to intialize array t, which serves as the real seed (9 doubles)
   public Chaotic2D(long seed) {
      t = new double[NMAX][NMAX];
      tn = new double[NMAX][NMAX];
      tret = new double[NMAX][NMAX];
      NU = 1.0e-13; NSTEP = 60; flag = -1;
      random = new Random(seed);
      for (int i = 0; i < NMAX; i++)
         for (int j = 0; j < NMAX; j++)
             t[i][j] = 2.0*random.nextDouble() - 1.0;
   }
```

```
private double f(double x) {        // Remapped logistic equation
   double temp = Math.abs(x);
   if (temp <= BETA) return(2.0*temp*(2.0-temp));
   else return(-2.0*(1.0-temp)*(1.0-temp));
}

private void step(double[][] t, double[][] tn) { // Coupled map lattice
   for (int i = 0; i < NMAX; i++)
      for (int j = 0; j < NMAX; j++)
         t[i][j] = f(t[i][j]);
   for (int i = 0; i < NMAX; i++)
      for (int j = 0; j < NMAX; j++)
         tn[i][j] = (1.0 - 4.0*NU)*t[i][j] +
    NU*(1.1*t[mod(i-1, NMAX)][j] +  // 1.1, 0.9, 1.2, 0.8
        0.9*t[mod(i+1, NMAX)][j] +  // added to prevent
1.2*t[i][mod(j-1, NMAX)] +  // falling into stable
0.8*t[i][mod(j+1, NMAX)]);  // configurations
}

private double S(double x) {        // Change distribution to uniform
   if (x >= 0) return(TWO_DIV_PI*Math.asin(Math.sqrt(x/2)));
   else return(TWO_DIV_PI*Math.asin(Math.sqrt(-x/2)) + 0.5);
}

private void chaoticUniform() {    // the generator itself
   for (int i = 0; i < NSTEP; i++) { // Iterate step 2*NSTEP times
      step(t, tn);
      step(tn, t);
   }
   for (int i = 0; i < NMAX; i++)
      for (int j = 0; j < NMAX; j++)
         tret[i][j] = S(t[i][j]);
}

public double nextRandom() { // call chaoticUniform once every 9 times
   double r = 0; // keep compiler happy
   if (flag == -1) {
      chaoticUniform(); // called only once every 9 times
      flag = 8;
   }
   int xf = flag/3, yf = flag%3;
   r = tret[xf][yf];
   flag--;
   return r;
}

public static void main(String[] args) {
   long seed = Long.parseLong(args[0]); // seed for RNG, up to 18 digits
   Chaotic2D chaotic2D = new Chaotic2D(seed);
   for (int i = 0; i < 10; i++)
      System.out.println(chaotic2D.nextRandom());
}
```

```
}
```

Below is simple output showing 10 random numbers coming from this generator. The next section shows a test of this generator using 256 000 random numbers. Notice that the seed is a Java **long** or up to 18 decimal digits. The real seed inside the generator is the array of 9 **double**s, and it would be possible to use this directly as the seed, rather than the method here of seeding using the Java **Random** class as an auxiliary generator.

```
% java Chaotic2D 999999999999999999
0.52692498814042
0.9980968834981016
0.5445878089596392
0.7249430391767984
0.07845814949526263
0.875011892397908
0.007317912434666028
0.497471911776451
0.8005000406206588
0.5236179345954219
```

Next is an experiment showing how the array of nodes evolves as the generator iterates. This experiment shows the results of three parallel runs, with the viscosity constant v equal to 0, 10^{-12}, and 10^{-13}. Simple changes were made to the code to allow these different values and to print values of all 9 nodes.

If all 9 initial numbers are the same, all node values will remain the same. In the run shown below, all nodes were set to 0.5 except for one which was set to 0.4999999999999. The resulting output is shown for three values of v in three columns below: 0.0, 10^{-12}, and 10^{-13}. The value $v = 0$ leaves all nodes independent of one another, that is, just 9 independent logistic equations. Iterations 2, 48, 74, 100, and 120 are shown.

Notice that after 120 iterations, all node values in the middle and right columns have drifted completely apart. This is the reason that 120 iterations are used in the generator itself.

Notice also that the initial value of 0.5 is transformed to itself by the remapped logistic equation. (This corresponds to 0.75 in the original equation.) This value is then transformed to $0.8333333\ldots$ by the transformation S that yields numbers uniformly distributed in the interval from 0 to 1.

The results of coupled values perturbing nearby nodes is shown in **bold** below:

```
v = 0                   v = 10^-12              v = 10^-13

Iteration number: 2
0.833333333333 1865     0.833333333333 1865     0.833333333333 1865
0.8333333333333334      0.8333333333333334      0.8333333333333334
0.8333333333333334      0.8333333333333334      0.8333333333333334
0.8333333333333334      0.8333333333333334      0.8333333333333334
0.8333333333333334      0.8333333333333334      0.8333333333333334
0.8333333333333334      0.8333333333333334      0.8333333333333334
0.8333333333333334      0.8333333333333334      0.8333333333333334
```

```
0.8333333333333334      0.8333333333333334      0.8333333333333334
0.8333333333333334      0.8333333333333334      0.8333333333333334

Iteration number: 48
0.5040834897791248      0.5036049050782517      0.5021691550116464
0.8333333333333334      0.8333333331818866      0.8333333332197 61
0.8333333333333334      0.8333333332366295      0.833333333325 15
0.8333333333333334      0.8333333332015176      0.8333333333229862
0.8333333333333334      0.8333333333333334      0.8333333333333334
0.8333333333333334      0.8333333333333334      0.8333333333333334
0.8333333333333334      0.8333333332189119      0.8333333333241839
0.8333333333333334      0.8333333333333334      0.8333333333333334
0.8333333333333334      0.8333333333333334      0.8333333333333334

Iteration number: 74
0.3602326768156076      0.5833674744417872      0.47153769384353167
0.8333333333333334      0.8231749738724833      0.8325716798549819
0.8333333333333334      0.826847020511059       0.8327845046032647
0.8333333333333334      0.8244919644092912      0.8326394144500628
0.8333333333333334      0.8333333333331181      0.8333333333333328
0.8333333333333334      0.8333333333331748      0.833333333333333
0.8333333333333334      0.8256584292117882      0.8327197098392694
0.8333333333333334      0.8333333333331479      0.8333333333333328
0.8333333333333334      0.8333333333332305      0.8333333333333331

Iteration number: 100
0.28322543483494855     0.9957436720669621      0.5329869100296993
0.8333333333333334      0.8698208441903008      0.3663600043966403
0.8333333333333334      0.24828271001134866     0.06072814757025873
0.8333333333333334      0.6086436328075263      0.7253693182396282
0.8333333333333334      0.8333152337340284      0.8333332244057435
0.8333333333333334      0.8333207796242934      0.8333332571545267
0.8333333333333334      0.7364311648949555      0.2422766762266617
0.8333333333333334      0.8333179900269688      0.8333332360776878
0.8333333333333334      0.833324925779414       0.8333332795639763

Iteration number: 120
0.8935574910647093      0.5833052795925358      0.7821714186572044
0.8333333333333334      0.7615175283592651      0.30797021553244913
0.8333333333333334      0.7909250549646865      0.07806647066456404
0.8333333333333334      0.6059141053682064      0.6417576242558025
0.8333333333333334      0.35452800150669156     0.719114512983271
0.8333333333333334      0.16981560934264034     0.7534540927550613
0.8333333333333334      0.045160386927817195    0.4919490670536787
0.8333333333333334      0.7447109018158442      0.7313534352471185
0.8333333333333334      0.9826254952427942      0.7769521043864014
```

Program V.18.a
Maurer's Universal Test

Referred to from page 112.

Java class: Maurer

```
// Maurer.java: implement Maurer's "Universal" Randomness Test
import java.util.*; // for Random

public class Maurer {
   private Generators gen;
   private double[] mu = {0, 0.7326495, 1.5374383, 2.4016068, 3.3112247,
                             4.2534266, 5.2177052, 6.1962507, 7.1836656,
                             8.1764248, 9.1723243, 10.170032, 11.168765,
                             12.168070, 13.167693, 14.167488, 15.167379};
   private double[] sigma2 = {0, 0.690, 1.338, 1.901, 2.358,
                                 2.705, 2.954, 3.125, 3.238,
                                 3.311, 3.356, 3.384, 3.401,
                                 3.410, 3.416, 3.419, 3.421};

   private int L; // number of bits in a block
   private int V; // 2^L, size of T
   private int Q; // number of initial runs to fill T
   private int K; // number of production runs
   private int[] T;
   // Maurer: constructor for this test
   //    pBits: number of bits in primes p and q.  n = p*q has 2*pBits bits
   //    rSize: number of bits to return each time
   public Maurer(int rBits) {
      L = rBits;
      V = (int)(Math.pow(2, L) + 0.000001);
      Q = 10*V;
      K = 1000*V;
      T = new int[V];
      Random rnd = new Random();
      gen = new Generators(rBits); // fetch rBits at a time
      double cLK = 0.7 - (0.8/(double)L) + (1.6 + (12.8/(double)L))*
         Math.pow((double)K, -4.0/(double)L);
      double sig2 = cLK*cLK*sigma2[L]/(double)K;
      double sig = Math.sqrt(sig2);
      System.out.println("\n*** Maurer's Universal Test ***");
      System.out.println("L: " + L + ", V: " + V +
                         ", Q: " + Q + ", K: " + K);
      System.out.println("Variance: " + sig2 + ", sigma: " + sig);
   }

   public double doTest() {
      for (int i = 0; i < 100; i++) {
        gen.nextBlock();
        // System.out.print(bbs.nextBlum() + " ");
```

```
         // if (i%20 == 19) System.out.println();
      }
      int bi;
      for (int j = 0; j < V; j++) T[j] = 0;
      for (int i = 1; i <= Q; i++) {
         bi = gen.nextBlock();
         T[bi] = i;
      }
      double sum = 0;
      for (int i = Q+1; i <= Q + K; i++) {
         bi = gen.nextBlock();
         sum += log2((double)(i - T[bi]));
         T[bi] = i;
         if (i % 1000 == 0) System.out.print("*");
      }
      System.out.println();
      double Xu = (double)sum/(double)K;
      return Xu;
   }

   private double log2(double x) {
      return Math.log(x)/Math.log(2.0);
   }

   public static void main(String[] args) {
      int rBits = Integer.parseInt(args[0]); // rBits = bits returned
      Maurer maurer = new Maurer(rBits);
      System.out.println(maurer.doTest());
   }
}
```

Program V.18.b
The Blum-Blum-Shub Perfect Generator

Referred to from page 112.

Program VI.20.a
Generate Multiplication Tables

Referred to from page 125.

The following Java program uses the slow $GF(2^8)$ multipy function to generate two tables needed for fast multiplication: a table of all powers of the generator **0x03**, and the inverse table. (The tables in the main section had a few extra frills inserted by hand.)

Java class: FFMultTables

```
// FFMultTables: create the arrays E and L, write html versions
public class FFMultTables {
   public byte[] E = new byte[256];
   public byte[] L = new byte[256];
   private String[] dig = {"0","1","2","3","4","5","6","7",
                           "8","9","a","b","c","d","e","f"};
   public byte FFMul(byte a, byte b) {
      byte aa = a, bb = b, r = 0, t;
      while (aa != 0) {
         if ((aa & 1) != 0)
            r = (byte)(r ^ bb);
         t = (byte)(bb & 0x80);
         bb = (byte)(bb << 1);
         if (t != 0)
            bb = (byte)(bb ^ 0x1b);
         aa = (byte)((aa & 0xff) >> 1);
      }
      return r;
   }

   public String hex(byte a) {
      return dig[(a & 0xff) >> 4] + dig[a & 0x0f];
   }

   public String hex(int a) {
      return dig[a];
   }

   public void loadE() {
      byte x = (byte)0x01;
      int index = 0;
      E[index++] = (byte)0x01;
      for (int i = 0; i < 255; i++) {
         byte y = FFMul(x, (byte)0x03);
         E[index++] = y;
         // System.out.print(hex(y) + " ");
         x = y;
      }
   }

   public void loadL() {
```

```
      int index;
      for (int i = 0; i < 255; i++) {
          L[E[i] & 0xff] = (byte)i;
      }
   }

   public void printE() {
      System.out.print("<table border><tr><td></td>");
      for (int i = 0; i < 16; i++)
         System.out.print("<th>" + hex(i) + "</th>");
      System.out.println("</tr>");
      for (int i = 0; i < 256; i++) {
         if (i%16 == 0)
            System.out.print("<tr><th> " + hex(i/16) +
               " </th>");
         System.out.print("<td> " + hex(E[i]) +
               " </td>");
         if (i%16 == 15) System.out.println("</tr>");
      }
      System.out.println("</table>");
   }

   public void printL() {
      System.out.print("<table border><tr><td></td>");
      for (int i = 0; i < 16; i++)
         System.out.print("<th>" + hex(i) + "</th>");
      System.out.println("</tr>");
      for (int i = 0; i < 256; i++) {
         if (i%16 == 0)
            System.out.print("<tr><th> " + hex(i/16) +
               " </th>");
         if (i == 0)
            System.out.print("<td>  </td>");
         else
            System.out.print("<td> " + hex(L[i]) +
               " </td>");
         if (i%16 == 15) System.out.println("</tr>");
      }
      System.out.println("</table>");
   }

   public static void main(String[] args) {
      FFMultTables ffm = new FFMultTables();
      ffm.loadE();
      ffm.loadL();
      ffm.printL();
      ffm.printE();
   }
}
```

Program VI.20.b
Compare Multiplication Results

Referred to from page 126.

There have been two algorithms for multiplying field elements, a slow one and a fast one. As a check, the following program compares the results of all 65536 possible products to see that the two methods agree (which they do):

Java class: FFMultTest

```
// FFMultTest: test two ways to multiply, all 65536 products
public class FFMultTest {
   public FFMultTest() {
      loadE();
      loadL();
   }

   public byte[] E = new byte[256]; // powers of 0x03
   public byte[] L = new byte[256]; // inverse of E
   private String[] dig = {"0","1","2","3","4","5","6","7",
                           "8","9","a","b","c","d","e","f"};

   {\timesbf // FFMulFast: fast multiply using table lookup
   public byte FFMulFast(byte a, byte b){
      int t = 0;;
      if (a == 0 || b == 0) return 0;
      t = (L[(a & 0xff)] & 0xff) + (L[(b & 0xff)] & 0xff);
      if (t > 255) t = t - 255;
      return E[(t & 0xff)];
   }

   // FFMul: slow multiply, using shifting
   public byte FFMul(byte a, byte b) {
      byte aa = a, bb = b, r = 0, t;
      while (aa != 0) {
         if ((aa & 1) != 0)
            r = (byte)(r ^ bb);
         t = (byte)(bb & 0x80);
         bb = (byte)(bb << 1);
         if (t != 0)
            bb = (byte)(bb ^ 0x1b);
         aa = (byte)((aa & 0xff) >> 1);
      }
      return r;
   }}

   // hex: print a byte as two hex digits
   public String hex(byte a) {
      return dig[(a & 0xff) >> 4] + dig[a & 0x0f];
   }
```

```
   // loadE: create and load the E table
   public void loadE() {
      byte x = (byte)0x01;
      int index = 0;
      E[index++] = (byte)0x01;
      for (int i = 0; i < 255; i++) {
         byte y = FFMul(x, (byte)0x03);
         E[index++] = y;
         // System.out.print(hex(y) + " ");
         x = y;
      }
   }

   // loadL: load the L table using the E table
   public void loadL() {
      int index;
      for (int i = 0; i < 255; i++) {
          L[E[i] & 0xff] = (byte)i;
      }
   }

   // testMul: go through all possible products of two bytes
   public void testMul() {
      byte a = 0;
      for(int i = 0; i < 256; i++) {
         byte b = 0;
         for(int j = 0; j < 256; j++) {
            byte x = FFMul(a, b);
            byte y = FFMulFast(a, b);
            if  (x != y) {
               System.out.println("a: " + hex(a) + ", b: " +
                  hex(b) + ", x: " + hex(x) + ", y: " + hex(y));
               System.exit(1);
            }
            b++;
         }
         a++;
      }
   }

   public static void main(String[] args) {
      FFMultTest ffmult = new FFMultTest();
      ffmult.testMul();
   }
}
```

Program VI.21.a
Generate AES Tables

Referred to from page 127.

Here is a Java program that will generate a number of 256-byte tables needed for the Advanced Encryption Standard:

Java class: Tables

```
// Tables: construct and print 256-byte tables needed for AES
public class Tables {
   public Tables() {
      loadE();
      loadL();
      loadInv();
      loadS();
      loadInvS();
      loadPowX();
   }

   public byte[] E = new byte[256]; // "exp" table (base 0x03)
   public byte[] L = new byte[256]; // "Log" table (base 0x03)
   public byte[] S = new byte[256]; // SubBytes table
   public byte[] invS = new byte[256]; // inv of SubBytes table
   public byte[] inv = new byte[256]; // multi inverse table
   public byte[] powX = new byte[15]; // powers of x = 0x02
   private String[] dig = {"0","1","2","3","4","5","6","7",
                           "8","9","a","b","c","d","e","f"};

   // FFMulFast: fast multiply using table lookup
   public byte FFMulFast(byte a, byte b){
      int t = 0;;
      if (a == 0 || b == 0) return 0;
      t = (L[(a & 0xff)] & 0xff) + (L[(b & 0xff)] & 0xff);
      if (t > 255) t = t - 255;
      return E[(t & 0xff)];
   }

   // FFMul: slow multiply, using shifting
   public byte FFMul(byte a, byte b) {
      byte aa = a, bb = b, r = 0, t;
      while (aa != 0) {
         if ((aa & 1) != 0)
            r = (byte)(r ^ bb);
         t = (byte)(bb & 0x80);
         bb = (byte)(bb << 1);
         if (t != 0)
            bb = (byte)(bb ^ 0x1b);
         aa = (byte)((aa & 0xff) >> 1);
      }
      return r;
```

```
}

// hex: print a byte as two hex digits
public String hex(byte a) {
   return dig[(a & 0xff) >> 4] + dig[a & 0x0f];
}

// hex: print a single digit (for tables)
public String hex(int a) {
   return dig[a];
}

// loadE: create and load the E table
public void loadE() {
   byte x = (byte)0x01;
   int index = 0;
   E[index++] = (byte)0x01;
   for (int i = 0; i < 255; i++) {
      byte y = FFMul(x, (byte)0x03);
      E[index++] = y;
      x = y;
   }
}

// loadL: load the L table using the E table
public void loadL() {
   int index;
   for (int i = 0; i < 255; i++) {
       L[E[i] & 0xff] = (byte)i;
   }
}

// loadS: load in the table S
public void loadS() {
   int index;
   for (int i = 0; i < 256; i++)
       S[i] = (byte)(subBytes((byte)(i & 0xff)) & 0xff);
}

// loadInv: load in the table inv
public void loadInv() {
   int index;
   for (int i = 0; i < 256; i++)
       inv[i] = (byte)(FFInv((byte)(i & 0xff)) & 0xff);
}

// loadInvS: load the invS table using the S table
public void loadInvS() {
   int index;
   for (int i = 0; i < 256; i++) {
       invS[S[i] & 0xff] = (byte)i;
   }
}
```

```
// loadPowX: load the powX table using multiplication
public void loadPowX() {
   int index;
   byte x = (byte)0x02;
   byte xp = x;
   powX[0] = 1; powX[1] = x;
   for (int i = 2; i < 15; i++) {
       xp = FFMulFast(xp, x);
       powX[i] = xp;
   }
}

// FFInv: the multiplicative inverse of a byte value
public byte FFInv(byte b) {
   byte e = L[b & 0xff];
   return E[0xff - (e & 0xff)];
}

// ithBIt: return the ith bit of a byte
public int ithBit(byte b, int i) {
   int m[] = {0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80};
   return  (b & m[i]) >> i;
}

// subBytes: the subBytes function
public int subBytes(byte b) {
   byte inB = b;
   int res = 0;
   if (b != 0) // if b == 0, leave it alone
      b = (byte)(FFInv(b) & 0xff);
   byte c = (byte)0x63;
   for (int i = 0; i < 8; i++) {
      int temp = 0;
      temp = ithBit(b, i) ^ ithBit(b, (i+4)%8) ^
        ithBit(b, (i+5)%8) ^ ithBit(b, (i+6)%8) ^
        ithBit(b, (i+7)%8) ^ ithBit(c, i);
      res = res | (temp << i);
   }
   return res;
}

// printTable: print a 256-byte table
public void printTable(byte[] S, String name) {
   System.out.print("<table border>");
   System.out.print("<tr><th colspan=2 rowspan=2>" +
      name + "(rs)</th>");
   System.out.print("<th colspan=16>s</th></tr><tr>");
   for (int i = 0; i < 16; i++)
      System.out.print("<th>" + hex(i) + "</th>");
   System.out.println("</tr><tr><th rowspan=17>r</th></tr>");
   for (int i = 0; i < 256; i++) {
      if (i%16 == 0)
```

```
            System.out.print("<tr><th> " + hex(i/16) +
               " </th>");
         System.out.print("<td> " + hex(S[i]) +
            " </td>");
         if (i%16 == 15) System.out.println("</tr>");
      }
      System.out.println("</table>");
   }

   // printL: print the L table
   public void printL() {
      printTable(L, "L");
   }

   // printE: print the E table
   public void printE() {
      printTable(E, "E");
   }

   // printS: print the S table
   public void printS() {
      printTable(S, "S");
   }

   // printInv: print the inv table
   public void printInv() {
      printTable(inv, "inv");
   }

   // printInvS: print the invS table
   public void printInvS() {
      printTable(invS, "iS");
   }

   // printpowX: print the powX table
   public void printPowX() {
      System.out.print("<table border><tr><th colspan=17>");
      System.out.print("Powers of x =
          0x02</th></tr><tr><th>i</th><th></th>");
      for (int i = 0; i < 15; i++)
         System.out.print("<th>" + i + "</th>");
      System.out.println("</tr><tr><th>x<sup>i</sup></th><th></th>");
      for (int i = 0; i < 15; i++)
         System.out.print("<td>" + hex(powX[i]) + "</td>");
      System.out.println("</tr></table>");
   }

   public static void main(String[] args) {
      Tables sB = new Tables();
      // sB.printL();
      // sB.printE();
      // sB.printS();
      // sB.printInvS();
```

```
      // sB.printInv();
      sB.printPowX();
   }
}
```

Program VI.23.a
AES Encryption

Referred to from page 135.

Encryption in the AES uses 6 classes: 2 principal ones, 3 utility ones, and a main driver. The results of testing both encryption and decryption appear after the next section on decryption.

- The class **`AESencrypt`** provides all the principle functions for the AES encryption algorithm.
- The class **`Tables`** gives access to computed tables and utility functions.
- The class **`GetBytes`** just reads bytes represented as Ascii hex characters (not in binary).
- The class **`Copy`** copies arrays back and forth for the AES.
- The class **`Print`** prints 1-and 2-dimensional arrays of bytes for debugging.
- The class **`AEStest`** is a driver for testing encryption.

Java class: AESencrypt

```
// AESencrypt: AES encryption
public class AESencrypt {
   private final int Nb = 4; // words in a block, always 4 for now
   private int Nk; // key length in words
   private int Nr; // number of rounds, = Nk + 6
   private int wCount; // position in w for RoundKey (= 0 each encrypt)
   private AEStables tab; // all the tables needed for AES
   private byte[] w; // the expanded key

   // AESencrypt: constructor for class.  Mainly expands key
   public AESencrypt(byte[] key, int NkIn) {
      Nk = NkIn; // words in a key, = 4, or 6, or 8
      Nr = Nk + 6; // corresponding number of rounds
      tab = new AEStables(); // class to give values of various functions
      w = new byte[4*Nb*(Nr+1)]; // room for expanded key
      KeyExpansion(key, w); // length of w depends on Nr
   }

   // Cipher: actual AES encrytion
   public void Cipher(byte[] in, byte[] out) {
      wCount = 0; // count bytes in expanded key throughout encryption
      byte[][] state = new byte[4][Nb]; // the state array
      Copy.copy(state, in); // actual component-wise copy
      AddRoundKey(state); // xor with expanded key
      for (int round = 1; round < Nr; round++) {
         Print.printArray("Start round  " + round + ":", state);
         SubBytes(state); // S-box substitution
```

```
      ShiftRows(state); // mix up rows
      MixColumns(state); // complicated mix of columns
      AddRoundKey(state); // xor with expanded key
   }
   Print.printArray("Start round " + Nr + ":", state);
   SubBytes(state); // S-box substitution
   ShiftRows(state); // mix up rows
   AddRoundKey(state); // xor with expanded key
   Copy.copy(out, state);
}

// KeyExpansion: expand key, byte-oriented code, but tracks words
private void KeyExpansion(byte[] key, byte[] w) {
   byte[] temp = new byte[4];
   // first just copy key to w
   int j = 0;
   while (j < 4*Nk) {
      w[j] = key[j++];
   }
   // here j == 4*Nk;
   int i;
   while(j < 4*Nb*(Nr+1)) {
      i = j/4; // j is always multiple of 4 here
      // handle everything word-at-a time, 4 bytes at a time
      for (int iTemp = 0; iTemp < 4; iTemp++)
         temp[iTemp] = w[j-4+iTemp];
      if (i % Nk == 0) {
         byte ttemp, tRcon;
         byte oldtemp0 = temp[0];
         for (int iTemp = 0; iTemp < 4; iTemp++) {
            if (iTemp == 3) ttemp = oldtemp0;
            else ttemp = temp[iTemp+1];
            if (iTemp == 0) tRcon = tab.Rcon(i/Nk);
            else tRcon = 0;
            temp[iTemp] = (byte)(tab.SBox(ttemp) ^ tRcon);
         }
      }
      else if (Nk > 6 && (i%Nk) == 4) {
         for (int iTemp = 0; iTemp < 4; iTemp++)
            temp[iTemp] = tab.SBox(temp[iTemp]);
      }
      for (int iTemp = 0; iTemp < 4; iTemp++)
         w[j+iTemp] = (byte)(w[j - 4*Nk + iTemp] ^ temp[iTemp]);
      j = j + 4;
   }
}

// SubBytes: apply Sbox substitution to each byte of state
private void SubBytes(byte[][] state) {
   for (int row = 0; row < 4; row++)
      for (int col = 0; col < Nb; col++)
         state[row][col] = tab.SBox(state[row][col]);
}
```

```
   // ShiftRows: simple circular shift of rows 1, 2, 3 by 1, 2, 3
   private void ShiftRows(byte[][] state) {
      byte[] t = new byte[4];
       for (int r = 1; r < 4; r++) {
         for (int c = 0; c < Nb; c++)
            t[c] = state[r][(c + r)%Nb];
         for (int c = 0; c < Nb; c++)
            state[r][c] = t[c];
      }
   }

   // MixColumns: complex and sophisticated mixing of columns
   private void MixColumns(byte[][] s) {
      int[] sp = new int[4];
      byte b02 = (byte)0x02, b03 = (byte)0x03;
      for (int c = 0; c < 4; c++) {
         sp[0] = tab.FFMul(b02, s[0][c]) ^ tab.FFMul(b03, s[1][c]) ^
                                s[2][c]  ^                 s[3][c];
         sp[1] =                s[0][c]  ^ tab.FFMul(b02, s[1][c]) ^
                 tab.FFMul(b03, s[2][c]) ^                 s[3][c];
         sp[2] =                s[0][c]  ^                 s[1][c]  ^
                 tab.FFMul(b02, s[2][c]) ^ tab.FFMul(b03, s[3][c]);
         sp[3] = tab.FFMul(b03, s[0][c]) ^                 s[1][c]  ^
                                s[2][c]  ^ tab.FFMul(b02, s[3][c]);
         for (int i = 0; i < 4; i++) s[i][c] = (byte)(sp[i]);
      }
   }

   // AddRoundKey: xor a portion of expanded key with state
   private void AddRoundKey(byte[][] state) {
      for (int c = 0; c < Nb; c++)
         for (int r = 0; r < 4; r++)
            state[r][c] = (byte)(state[r][c] ^ w[wCount++]);
   }
}
```

Java class: AEStables

```
// AEStables: construct various 256-byte tables needed for AES
public class AEStables {
   public AEStables() {
      loadE(); loadL(); loadInv();
      loadS(); loadInvS(); loadPowX();
   }

   private byte[] E = new byte[256]; // "exp" table (base 0x03)
   private byte[] L = new byte[256]; // "Log" table (base 0x03)
   private byte[] S = new byte[256]; // SubBytes table
   private byte[] invS = new byte[256]; // inverse of SubBytes table
   private byte[] inv = new byte[256]; // multiplicative inverse table
   private byte[] powX = new byte[15]; // powers of x = 0x02

   // Routines to access table entries
   public byte SBox(byte b) {
```

```
   return S[b & 0xff];
}

public byte invSBox(byte b) {
   return invS[b & 0xff];
}

public byte Rcon(int i) {
   return powX[i-1];
}

// FFMulFast: fast multiply using table lookup
public byte FFMulFast(byte a, byte b){
   int t = 0;;
   if (a == 0 || b == 0) return 0;
   t = (L[(a & 0xff)] & 0xff) + (L[(b & 0xff)] & 0xff);
   if (t > 255) t = t - 255;
   return E[(t & 0xff)];
}

// FFMul: slow multiply, using shifting
public byte FFMul(byte a, byte b) {
   byte aa = a, bb = b, r = 0, t;
   while (aa != 0) {
      if ((aa & 1) != 0)
         r = (byte)(r ^ bb);
      t = (byte)(bb & 0x80);
      bb = (byte)(bb << 1);
      if (t != 0)
         bb = (byte)(bb ^ 0x1b);
      aa = (byte)((aa & 0xff) >> 1);
   }
   return r;
}

// loadE: create and load the E table
private void loadE() {
   byte x = (byte)0x01;
   int index = 0;
   E[index++] = (byte)0x01;
   for (int i = 0; i < 255; i++) {
      byte y = FFMul(x, (byte)0x03);
      E[index++] = y;
      x = y;
   }
}

// loadL: load the L table using the E table
private void loadL() { // careful: had 254 below several places
   int index;
   for (int i = 0; i < 255; i++) {
       L[E[i] & 0xff] = (byte)i;
   }
```

```
}

// loadS: load in the table S
private void loadS() {
   int index;
   for (int i = 0; i < 256; i++)
       S[i] = (byte)(subBytes((byte)(i & 0xff)) & 0xff);
}

// loadInv: load in the table inv
private void loadInv() {
   int index;
   for (int i = 0; i < 256; i++)
       inv[i] = (byte)(FFInv((byte)(i & 0xff)) & 0xff);
}

// loadInvS: load the invS table using the S table
private void loadInvS() {
   int index;
   for (int i = 0; i < 256; i++) {
       invS[S[i] & 0xff] = (byte)i;
   }
}

// loadPowX: load the powX table using multiplication
private void loadPowX() {
   int index;
   byte x = (byte)0x02;
   byte xp = x;
   powX[0] = 1; powX[1] = x;
   for (int i = 2; i < 15; i++) {
       xp = FFMul(xp, x);
       powX[i] = xp;
   }
}

// FFInv: the multiplicative inverse of a byte value
public byte FFInv(byte b) {
   byte e = L[b & 0xff];
   return E[0xff - (e & 0xff)];
}

// ithBIt: return the ith bit of a byte
public int ithBit(byte b, int i) {
   int m[] = {0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80};
   return  (b & m[i]) >> i;
}

// subBytes: the subBytes function
public int subBytes(byte b) {
   byte inB = b;
   int res = 0;
   if (b != 0) // if b == 0, leave it alone
```

```
            b = (byte)(FFInv(b) & 0xff);
         byte c = (byte)0x63;
         for (int i = 0; i < 8; i++) {
            int temp = 0;
            temp = ithBit(b, i) ^ ithBit(b, (i+4)%8) ^ ithBit(b, (i+5)%8) ^
              ithBit(b, (i+6)%8) ^ ithBit(b, (i+7)%8) ^ ithBit(c, i);
            res = res | (temp << i);
         }
         return res;
      }
   }
```

Java class: GetBytes

```
// GetBytes: fetch array of bytes, represented in hex
import java.io.*;
public class GetBytes {
   private String fileName; // input filename
   private int arraySize; // number of bytes to read
   private Reader in;

   // GetBytes: constructor, opens input file
   public GetBytes(String file, int n) {
      fileName = file;
      arraySize = n;
      try {
         in = new FileReader(fileName);
      } catch (IOException e) {
         System.out.println("Exception opening " + fileName);
      }
   }

   // getNextChar: fetches next char
   private char getNextChar() {
      char ch = ' '; // = ' ' to keep compiler happy
      try {
         ch = (char)in.read();
      } catch (IOException e) {
         System.out.println("Exception reading character");
      }
      return ch;
   }

   // val: return int value of hex digit
   private int val(char ch) {
      if (ch >= '0' && ch <= '9') return ch - '0';
      if (ch >= 'a' && ch <= 'f') return ch - 'a' + 10;
      if (ch >= 'A' && ch <= 'F') return ch - 'A' + 10;
      return -1000000;
   }

   // getBytes: fetch array of bytes in hex
   public byte[] getBytes() {
      byte[] ret = new byte[arraySize];
```

```
      for (int i = 0; i < arraySize; i++) {
         char ch1 = getNextChar();
         char ch2 = getNextChar();
         ret[i] = (byte)(val(ch1)*16 + val(ch2));
      }
      return ret;
   }
}
```

Java class: Copy

```
// Copy: copy arrays of bytes
public class Copy {
   private static final int Nb = 4;
   // copy: copy in to state
   public static void copy(byte[][] state, byte[] in) {
      int inLoc = 0;
      for (int c = 0; c < Nb; c++)
         for (int r = 0; r < 4; r++)
            state[r][c] = in[inLoc++];
   }

   // copy: copy state to out
   public static void copy(byte[] out, byte[][] state) {
      int outLoc = 0;
      for (int c = 0; c < Nb; c++)
         for (int r = 0; r < 4; r++)
            out[outLoc++] = state[r][c];
   }
}
```

Java class: Print

```
// Print: print arrays of bytes
public class Print {
   private static final int Nb = 4;
   private static String[] dig = {"0","1","2","3","4","5","6","7",
                                  "8","9","a","b","c","d","e","f"};

   // hex: print a byte as two hex digits
   public static String hex(byte a) {
      return dig[(a & 0xff) >> 4] + dig[a & 0x0f];
   }

   public static void printArray(String name, byte[] a) {
      System.out.print(name + " ");
      for (int i = 0; i < a.length; i++)
         System.out.print(hex(a[i]) + " ");
      System.out.println();
   }

   public static void printArray(String name, byte[][] s) {
      System.out.print(name + " ");
      for (int c = 0; c < Nb; c++)
         for (int r = 0; r < 4; r++)
            System.out.print(hex(s[r][c]) + " ");
```

```
      System.out.println();
   }
}
```

Java class: AEStest

```
// AEStest: test AES encryption
public class AEStest {

   public static void main(String[] args) {
      // for 128-bit key, use 16, 16, and 4 below
      // for 192-bit key, use 16, 24 and 6 below
      // for 256-bit key, use 16, 32 and 8 below
      GetBytes getInput = new GetBytes("plaintext1.txt", 16);
      byte[] in = getInput.getBytes();
      GetBytes getKey = new GetBytes("key1.txt", 16);
      byte[] key = getKey.getBytes();
      AESencrypt aes = new AESencrypt(key, 4);
      Print.printArray("Plaintext:     ", in);
      Print.printArray("Key:           ", key);
      byte[] out = new byte[16];
      aes.Cipher(in, out);
      Print.printArray("Ciphertext:    ", out);
   }
}
```

Program VI.24.a
AES Decryption

Referred to from page 138.

Classes **`Tables`**, **`GetBytes`**, **`Copy`**, and **`Print`** are the same as for encryption as presented in the previous section.
The class **`AESdecrypt`** provides all the principle functions for the AES decryption algorithm, while **`AESinvTest`** is a driver for testing decryption.

Java class: AESdecrypt

```
// AESdecrypt: AES decryption
public class AESdecrypt {
   public final int Nb = 4; // words in a block, always 4 for now
   public int Nk; // key length in words
   public int Nr; // number of rounds, = Nk + 6
   private int wCount; // position in w (= 4*Nb*(Nr+1) each encrypt)
   AEStables tab; // all the tables needed for AES
   byte[] w; // the expanded key

   // AESdecrypt: constructor for class.  Mainly expands key
   public AESdecrypt(byte[] key, int NkIn) {
      Nk = NkIn; // words in a key, = 4, or 6, or 8
      Nr = Nk + 6; // corresponding number of rounds
      tab = new AEStables(); // class to give values of various functions
      w = new byte[4*Nb*(Nr+1)]; // room for expanded key
      KeyExpansion(key, w); // length of w depends on Nr
   }

   // InvCipher: actual AES decryption
   public void InvCipher(byte[] in, byte[] out) {
      wCount = 4*Nb*(Nr+1); // count bytes during decryption
      byte[][] state = new byte[4][Nb]; // the state array
      Copy.copy(state, in); // actual component-wise copy
      InvAddRoundKey(state); // xor with expanded key
      for (int round = Nr-1; round >= 1; round--) {
         Print.printArray("Start round  " + (Nr - round) + ":", state);
         InvShiftRows(state); // mix up rows
         InvSubBytes(state); // inverse S-box substitution
         InvAddRoundKey(state); // xor with expanded key
         InvMixColumns(state); // complicated mix of columns
      }
      Print.printArray("Start round  " + Nr + ":", state);
      InvShiftRows(state); // mix up rows
      InvSubBytes(state); // inverse S-box substitution
      InvAddRoundKey(state); // xor with expanded key
      Copy.copy(out, state);
   }

   // KeyExpansion: expand key, byte-oriented code, but tracks words
```

```
//  (the same as for encryption)
private void KeyExpansion(byte[] key, byte[] w) {
   byte[] temp = new byte[4];
   // first just copy key to w
   int j = 0;
   while (j < 4*Nk) {
      w[j] = key[j++];
   }
   // here j == 4*Nk;
   int i;
   while(j < 4*Nb*(Nr+1)) {
      i = j/4; // j is always multiple of 4 here
      // handle everything word-at-a time, 4 bytes at a time
      for (int iTemp = 0; iTemp < 4; iTemp++)
         temp[iTemp] = w[j-4+iTemp];
      if (i % Nk == 0) {
         byte ttemp, tRcon;
         byte oldtemp0 = temp[0];
         for (int iTemp = 0; iTemp < 4; iTemp++) {
            if (iTemp == 3) ttemp = oldtemp0;
            else ttemp = temp[iTemp+1];
            if (iTemp == 0) tRcon = tab.Rcon(i/Nk);
            else tRcon = 0;
            temp[iTemp] = (byte)(tab.SBox(ttemp) ^ tRcon);
         }
      }
      else if (Nk > 6 && (i%Nk) == 4) {
         for (int iTemp = 0; iTemp < 4; iTemp++)
            temp[iTemp] = tab.SBox(temp[iTemp]);
      }
      for (int iTemp = 0; iTemp < 4; iTemp++)
         w[j+iTemp] = (byte)(w[j - 4*Nk + iTemp] ^ temp[iTemp]);
      j = j + 4;
   }
}

// InvSubBytes: apply inverse Sbox substitution to each byte of state
private void InvSubBytes(byte[][] state) {
   for (int row = 0; row < 4; row++)
      for (int col = 0; col < Nb; col++)
         state[row][col] = tab.invSBox(state[row][col]);
}

// InvShiftRows: right circular shift of rows 1, 2, 3 by 1, 2, 3
private void InvShiftRows(byte[][] state) {
   byte[] t = new byte[4];
   for (int r = 1; r < 4; r++) {
      for (int c = 0; c < Nb; c++)
         t[(c + r)%Nb] = state[r][c];
      for (int c = 0; c < Nb; c++)
         state[r][c] = t[c];
   }
}
```

```
   // InvMixColumns: complex and sophisticated mixing of columns
   private void InvMixColumns(byte[][] s) {
      int[] sp = new int[4];
      byte b0b = (byte)0x0b; byte b0d = (byte)0x0d;
      byte b09 = (byte)0x09; byte b0e = (byte)0x0e;
      for (int c = 0; c < 4; c++) {
         sp[0] = tab.FFMul(b0e, s[0][c]) ^ tab.FFMul(b0b, s[1][c]) ^
                 tab.FFMul(b0d, s[2][c]) ^ tab.FFMul(b09, s[3][c]);
         sp[1] = tab.FFMul(b09, s[0][c]) ^ tab.FFMul(b0e, s[1][c]) ^
                 tab.FFMul(b0b, s[2][c]) ^ tab.FFMul(b0d, s[3][c]);
         sp[2] = tab.FFMul(b0d, s[0][c]) ^ tab.FFMul(b09, s[1][c]) ^
                 tab.FFMul(b0e, s[2][c]) ^ tab.FFMul(b0b, s[3][c]);
         sp[3] = tab.FFMul(b0b, s[0][c]) ^ tab.FFMul(b0d, s[1][c]) ^
                 tab.FFMul(b09, s[2][c]) ^ tab.FFMul(b0e, s[3][c]);
         for (int i = 0; i < 4; i++) s[i][c] = (byte)(sp[i]);
      }
   }

   // InvAddRoundKey: same as AddRoundKey, but backwards
   private void InvAddRoundKey(byte[][] state) {
   for (int c = Nb - 1; c >= 0; c--)
      for (int r = 3; r >= 0 ; r--)
         state[r][c] = (byte)(state[r][c] ^ w[--wCount]);
   }
}
```

Java class: AESinvTest

```
// AESinvTest: test AES decryption
public class AESinvTest {

   public static void main(String[] args) {
      // for 128-bit key, use 16, 16, and 4 below
      // for 192-bit key, use 16, 24 and 6 below
      // for 256-bit key, use 16, 32 and 8 below
      GetBytes getInput = new GetBytes("ciphertext1.txt", 16);
      byte[] in = getInput.getBytes();
      GetBytes getKey = new GetBytes("key1.txt", 16);
      byte[] key = getKey.getBytes();
      AESdecrypt aesDec = new AESdecrypt(key, 4);
      Print.printArray("Ciphertext:    ", in);
      Print.printArray("Key:           ", key);
      byte[] out = new byte[16];
      aesDec.InvCipher(in, out);
      Print.printArray("Plaintext:     ", out);
   }
}
```

Program VI.24.b
Test Runs of the AES Algorithm

Referred to from page 138.

Here are results of test runs with all the sample test data supplied in the AES Specification and in B. Gladman's writeup about the AES. The values in the **`state`** variable are shown at the start of each round. There are also test runs with plaintext and key all zeros and with a single **`1`** inserted. The AES Specification and Gladman also show step-by-step results of the key expansion for these cases, which was useful for my debugging, but I don't show that data here.

```
Gladman's Test Data, 128-bit key
Encrypting ...
Plaintext:      32 43 f6 a8 88 5a 30 8d 31 31 98 a2 e0 37 07 34
Key:            2b 7e 15 16 28 ae d2 a6 ab f7 15 88 09 cf 4f 3c
Start round  1: 19 3d e3 be a0 f4 e2 2b 9a c6 8d 2a e9 f8 48 08
Start round  2: a4 9c 7f f2 68 9f 35 2b 6b 5b ea 43 02 6a 50 49
Start round  3: aa 8f 5f 03 61 dd e3 ef 82 d2 4a d2 68 32 46 9a
Start round  4: 48 6c 4e ee 67 1d 9d 0d 4d e3 b1 38 d6 5f 58 e7
Start round  5: e0 92 7f e8 c8 63 63 c0 d9 b1 35 50 85 b8 be 01
Start round  6: f1 00 6f 55 c1 92 4c ef 7c c8 8b 32 5d b5 d5 0c
Start round  7: 26 0e 2e 17 3d 41 b7 7d e8 64 72 a9 fd d2 8b 25
Start round  8: 5a 41 42 b1 19 49 dc 1f a3 e0 19 65 7a 8c 04 0c
Start round  9: ea 83 5c f0 04 45 33 2d 65 5d 98 ad 85 96 b0 c5
Start round 10: eb 40 f2 1e 59 2e 38 84 8b a1 13 e7 1b c3 42 d2
Ciphertext:     39 25 84 1d 02 dc 09 fb dc 11 85 97 19 6a 0b 32
Decrypting ...
Ciphertext:     39 25 84 1d 02 dc 09 fb dc 11 85 97 19 6a 0b 32
Key:            2b 7e 15 16 28 ae d2 a6 ab f7 15 88 09 cf 4f 3c
Start round  1: e9 31 7d b5 cb 32 2c 72 3d 2e 89 5f af 09 07 94
Start round  2: 87 6e 46 a6 f2 4c e7 8c 4d 90 4a d8 97 ec c3 95
Start round  3: be 3b d4 fe d4 e1 f2 c8 0a 64 2c c0 da 83 86 4d
Start round  4: f7 83 40 3f 27 43 3d f0 9b b5 31 ff 54 ab a9 d3
Start round  5: a1 4f 3d fe 78 e8 03 fc 10 d5 a8 df 4c 63 29 23
Start round  6: e1 fb 96 7c e8 c8 ae 9b 35 6c d2 ba 97 4f fb 53
Start round  7: 52 a4 c8 94 85 11 6a 28 e3 cf 2f d7 f6 50 5e 07
Start round  8: ac c1 d6 b8 ef b5 5a 7b 13 23 cf df 45 73 11 b5
Start round  9: 49 db 87 3b 45 39 53 89 7f 02 d2 f1 77 de 96 1a
Start round 10: d4 bf 5d 30 e0 b4 52 ae b8 41 11 f1 1e 27 98 e5
Plaintext:      32 43 f6 a8 88 5a 30 8d 31 31 98 a2 e0 37 07 34
```

```
Gladman's Test Data, 192-bit key
Encrypting ...
Plaintext:      32 43 f6 a8 88 5a 30 8d 31 31 98 a2 e0 37 07 34
Key:            2b 7e 15 16 28 ae d2 a6 ab f7 15 88 09 cf 4f 3c
                76 2e 71 60 f3 8b 4d a5
Start round  1: 19 3d e3 be a0 f4 e2 2b 9a c6 8d 2a e9 f8 48 08
Start round  2: 72 48 f0 85 13 40 54 3f 5f 65 c0 61 17 35 e7 f1
Start round  3: 14 e2 0a 1f b3 dc 3a 62 36 27 2f d3 da 75 6f 70
```

```
Start round  4: cb 42 fd 92 33 3f 28 43 21 11 fe 84 3c bc a8 1a
Start round  5: 94 99 c6 ee b9 78 94 12 bb 04 09 b7 a7 97 c0 25
Start round  6: 8a 6c 1e 3e db 78 a6 4e f5 db 78 62 ea d6 a4 01
Start round  7: 43 5c e2 58 97 7c 16 d8 71 7c 0f f7 79 19 e5 19
Start round  8: 70 b8 37 b9 ae fc 8b bc 5c d2 ab a5 cc 56 d7 4e
Start round  9: 94 a2 c3 31 ed 28 bf de d7 d6 c5 83 4b a9 ed 1e
Start round 10: 52 2d 88 c5 ed ab 19 4e 25 ec 73 1c 11 fa 6b 08
Start round 11: ab 82 54 06 da 72 4d 0c 2b cc f6 c2 39 32 12 01
Start round 12: 43 88 b3 26 6a f7 68 e8 4f cc a4 2a 3a 4d 45 5f
Ciphertext:     f9 fb 29 ae fc 38 4a 25 03 40 d8 33 b8 7e bc 00
Decrypting ...
Ciphertext:     f9 fb 29 ae fc 38 4a 25 03 40 d8 33 b8 7e bc 00
Key:            2b 7e 15 16 28 ae d2 a6 ab f7 15 88 09 cf 4f 3c
                76 2e 71 60 f3 8b 4d a5
Start round  1: 1a 68 49 cf 02 4b 6e f7 84 e3 6d 9b 80 c4 45 e5
Start round  2: 62 40 42 7c 57 4b c9 6f f1 23 20 fe 12 13 e3 25
Start round  3: 00 62 8f 30 55 ce 7f a6 3f 2d c4 2f 82 d8 d4 9c
Start round  4: 22 34 a6 72 55 f6 55 c7 0e d3 2e 1d b3 3a 08 ec
Start round  5: 51 b0 62 2f e4 b5 0e 56 4a b1 9a 65 4b 6c 3d 06
Start round  6: 1a 10 76 d4 88 10 d9 6a a3 d4 98 61 b6 4a 47 68
Start round  7: 7e bc bc 7c b9 b9 49 b2 e6 f6 72 2f 87 50 24 aa
Start round  8: 22 bc 01 3f 56 f2 ba 28 ea 88 b4 c9 5c ee 22 a9
Start round  9: 1f 75 bb a2 c3 82 c2 4f fd 65 54 1a eb 2c 34 5f
Start round 10: fa 86 15 51 6d cc a8 c0 05 9d 67 aa 57 98 80 66
Start round 11: 40 09 ba a1 7d 4d 94 97 cf 96 8c 75 f0 52 20 ef
Start round 12: d4 bf 5d 30 e0 b4 52 ae b8 41 11 f1 1e 27 98 e5
Plaintext:      32 43 f6 a8 88 5a 30 8d 31 31 98 a2 e0 37 07 34
```

Gladman's Test Data, **256**-bit key

```
Encrypting ...
Plaintext:      32 43 f6 a8 88 5a 30 8d 31 31 98 a2 e0 37 07 34
Key:            2b 7e 15 16 28 ae d2 a6 ab f7 15 88 09 cf 4f 3c
                76 2e 71 60 f3 8b 4d a5 6a 78 4d 90 45 19 0c fe
Start round  1: 19 3d e3 be a0 f4 e2 2b 9a c6 8d 2a e9 f8 48 08
Start round  2: 72 48 f0 85 13 40 54 3f 22 80 9e ea 6d 1f 2a b2
Start round  3: 59 f8 a8 d4 12 71 bf 44 e2 2b a6 5e 5d 69 9a 49
Start round  4: 88 a8 eb d5 66 49 40 5e 9f ad 55 e9 33 0d 7f 84
Start round  5: 6d 0c 80 51 d5 bc 1d b5 c5 1f 45 0f 18 46 7f 34
Start round  6: cf f5 04 43 27 6f 76 55 a5 5a fd 7b b6 99 f4 5f
Start round  7: 63 93 d6 68 2d da 2e 4f 42 88 77 37 12 57 8a 11
Start round  8: 29 26 ae 58 f4 32 23 4b f0 70 ff 6e 56 9e 44 23
Start round  9: 3f da e4 32 0e ce 55 ce c9 32 d6 55 e4 3a cd 2b
Start round 10: d6 40 90 18 38 64 11 35 61 ef 7c 37 99 00 31 fd
Start round 11: 6b 3c e6 72 ea e1 1d 52 e2 8f 1d 54 96 e0 c0 d0
Start round 12: 84 74 88 72 49 e5 0a 9f 17 c0 5a 37 a1 a6 9f 41
Start round 13: bf 8a 29 14 80 f8 06 21 44 3e 2b 81 aa 2f 4c 16
Start round 14: d3 20 3d d1 1a c7 2d 8b 5e c4 72 24 95 3d fe 5b
Ciphertext:     1a 6e 6c 2c 66 2e 7d a6 50 1f fb 62 bc 9e 93 f3
Decrypting ...
Ciphertext:     1a 6e 6c 2c 66 2e 7d a6 50 1f fb 62 bc 9e 93 f3
Key:            2b 7e 15 16 28 ae d2 a6 ab f7 15 88 09 cf 4f 3c
                76 2e 71 60 f3 8b 4d a5 6a 78 4d 90 45 19 0c fe
Start round  1: 66 c6 40 39 a2 1c bb 3e 58 27 27 3d 2a b7 d8 36
```

```
Start round  2: 08 41 f1 47 cd b2 29 fa 1b 15 a5 fd ac 7e 6f 0c
Start round  3: 5f d9 be 83 3b ba db 40 f0 24 c4 db 32 92 67 9a
Start round  4: 7f f8 a4 70 87 73 ba 40 98 e1 8e 00 90 eb a4 20
Start round  5: f6 43 10 54 07 df c7 ad ef 63 60 96 ee 09 82 9a
Start round  6: 75 8b f6 f1 ab 23 bd 23 dd 80 69 8b 69 57 fc fc
Start round  7: a5 23 16 26 bf 51 1b 6a 8c 0b e4 b3 b1 f7 26 9f
Start round  8: fb 57 f5 82 d8 c4 7e 45 2c 5b f6 84 c9 dc 31 9a
Start round  9: 8a a8 54 cf cc be bf 1a 06 ee f2 fc 4e e6 38 21
Start round 10: 3c 65 6e 18 03 c0 d2 d1 a6 5a cd d5 ad fe a4 76
Start round 11: c4 3b fc 5f 33 95 d2 03 db d7 e9 58 c3 c2 09 1e
Start round 12: cb a3 24 3b c9 f1 b8 48 98 f9 c2 1b 4c 41 08 58
Start round 13: 40 09 0b 37 7d cd e5 97 93 c0 8c 75 3c 52 20 87
Start round 14: d4 bf 5d 30 e0 b4 52 ae b8 41 11 f1 1e 27 98 e5
Plaintext:      32 43 f6 a8 88 5a 30 8d 31 31 98 a2 e0 37 07 34
```

AES Specification Test Data, **128**-bit key

```
Encrypting ...
Plaintext:      00 11 22 33 44 55 66 77 88 99 aa bb cc dd ee ff
Key:            00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
Start round  1: 00 10 20 30 40 50 60 70 80 90 a0 b0 c0 d0 e0 f0
Start round  2: 89 d8 10 e8 85 5a ce 68 2d 18 43 d8 cb 12 8f e4
Start round  3: 49 15 59 8f 55 e5 d7 a0 da ca 94 fa 1f 0a 63 f7
Start round  4: fa 63 6a 28 25 b3 39 c9 40 66 8a 31 57 24 4d 17
Start round  5: 24 72 40 23 69 66 b3 fa 6e d2 75 32 88 42 5b 6c
Start round  6: c8 16 77 bc 9b 7a c9 3b 25 02 79 92 b0 26 19 96
Start round  7: c6 2f e1 09 f7 5e ed c3 cc 79 39 5d 84 f9 cf 5d
Start round  8: d1 87 6c 0f 79 c4 30 0a b4 55 94 ad d6 6f f4 1f
Start round  9: fd e3 ba d2 05 e5 d0 d7 35 47 96 4e f1 fe 37 f1
Start round 10: bd 6e 7c 3d f2 b5 77 9e 0b 61 21 6e 8b 10 b6 89
Ciphertext:     69 c4 e0 d8 6a 7b 04 30 d8 cd b7 80 70 b4 c5 5a
Decrypting ...
Ciphertext:     69 c4 e0 d8 6a 7b 04 30 d8 cd b7 80 70 b4 c5 5a
Key:            00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
Start round  1: 7a d5 fd a7 89 ef 4e 27 2b ca 10 0b 3d 9f f5 9f
Start round  2: 54 d9 90 a1 6b a0 9a b5 96 bb f4 0e a1 11 70 2f
Start round  3: 3e 1c 22 c0 b6 fc bf 76 8d a8 50 67 f6 17 04 95
Start round  4: b4 58 12 4c 68 b6 8a 01 4b 99 f8 2e 5f 15 55 4c
Start round  5: e8 da b6 90 14 77 d4 65 3f f7 f5 e2 e7 47 dd 4f
Start round  6: 36 33 9d 50 f9 b5 39 26 9f 2c 09 2d c4 40 6d 23
Start round  7: 2d 6d 7e f0 3f 33 e3 34 09 36 02 dd 5b fb 12 c7
Start round  8: 3b d9 22 68 fc 74 fb 73 57 67 cb e0 c0 59 0e 2d
Start round  9: a7 be 1a 69 97 ad 73 9b d8 c9 ca 45 1f 61 8b 61
Start round 10: 63 53 e0 8c 09 60 e1 04 cd 70 b7 51 ba ca d0 e7
Plaintext:      00 11 22 33 44 55 66 77 88 99 aa bb cc dd ee ff
```

AES Specification Test Data, **192**-bit key

```
Encrypting ...
Plaintext:      00 11 22 33 44 55 66 77 88 99 aa bb cc dd ee ff
Key:            00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
                10 11 12 13 14 15 16 17
Start round  1: 00 10 20 30 40 50 60 70 80 90 a0 b0 c0 d0 e0 f0
Start round  2: 4f 63 76 06 43 e0 aa 85 af f8 c9 d0 41 fa 0d e4
Start round  3: cb 02 81 8c 17 d2 af 9c 62 aa 64 42 8b b2 5f d7
```

```
Start round  4: f7 5c 77 78 a3 27 c8 ed 8c fe bf c1 a6 c3 7f 53
Start round  5: 22 ff c9 16 a8 14 74 41 64 96 f1 9c 64 ae 25 32
Start round  6: 80 12 1e 07 76 fd 1d 8a 8d 8c 31 bc 96 5d 1f ee
Start round  7: 67 1e f1 fd 4e 2a 1e 03 df dc b1 ef 3d 78 9b 30
Start round  8: 0c 03 70 d0 0c 01 e6 22 16 6b 8a cc d6 db 3a 2c
Start round  9: 72 55 da d3 0f b8 03 10 e0 0d 6c 6b 40 d0 52 7c
Start round 10: a9 06 b2 54 96 8a f4 e9 b4 bd b2 d2 f0 c4 43 36
Start round 11: 88 ec 93 0e f5 e7 e4 b6 cc 32 f4 c9 06 d2 94 14
Start round 12: af b7 3e eb 1c d1 b8 51 62 28 0f 27 fb 20 d5 85
Ciphertext:     dd a9 7c a4 86 4c df e0 6e af 70 a0 ec 0d 71 91
Decrypting ...
Ciphertext:     dd a9 7c a4 86 4c df e0 6e af 70 a0 ec 0d 71 91
Key:            00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
                10 11 12 13 14 15 16 17
Start round  1: 79 3e 76 97 9c 34 03 e9 aa b7 b2 d1 0f a9 6c cc
Start round  2: c4 94 bf fa e6 23 22 ab 4b b5 dc 4e 6f ce 69 dd
Start round  3: d3 7e 37 05 90 7a 1a 20 8d 1c 37 1e 8c 6f bf b5
Start round  4: 40 6c 50 10 76 d7 00 66 e1 70 57 ca 09 fc 7b 7f
Start round  5: fe 7c 7e 71 fe 7f 80 70 47 b9 51 93 f6 7b 8e 4b
Start round  6: 85 e5 c8 04 2f 86 14 54 9e bc a1 7b 27 72 72 df
Start round  7: cd 54 c7 28 38 64 c0 c5 5d 4c 72 7e 90 c9 a4 65
Start round  8: 93 fa a1 23 c2 90 3f 47 43 e4 dd 83 43 16 92 de
Start round  9: 68 cc 08 ed 0a bb d2 bc 64 2e f5 55 24 4a e8 78
Start round 10: 1f b5 43 0e f0 ac cf 64 aa 37 0c de 3d 77 79 2c
Start round 11: 84 e1 dd 69 1a 41 d7 6f 79 2d 38 97 83 fb ac 70
Start round 12: 63 53 e0 8c 09 60 e1 04 cd 70 b7 51 ba ca d0 e7
Plaintext:      00 11 22 33 44 55 66 77 88 99 aa bb cc dd ee ff
```

AES Specification Test Data, **256**-bit key

```
Encrypting ...
Plaintext:      00 11 22 33 44 55 66 77 88 99 aa bb cc dd ee ff
Key:            00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
                10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f
Start round  1: 00 10 20 30 40 50 60 70 80 90 a0 b0 c0 d0 e0 f0
Start round  2: 4f 63 76 06 43 e0 aa 85 ef a7 21 32 01 a4 e7 05
Start round  3: 18 59 fb c2 8a 1c 00 a0 78 ed 8a ad c4 2f 61 09
Start round  4: 97 5c 66 c1 cb 9f 3f a8 a9 3a 28 df 8e e1 0f 63
Start round  5: 1c 05 f2 71 a4 17 e0 4f f9 21 c5 c1 04 70 15 54
Start round  6: c3 57 aa e1 1b 45 b7 b0 a2 c7 bd 28 a8 dc 99 fa
Start round  7: 7f 07 41 43 cb 4e 24 3e c1 0c 81 5d 83 75 d5 4c
Start round  8: d6 53 a4 69 6c a0 bc 0f 5a ca ab 5d b9 6c 5e 7d
Start round  9: 5a a8 58 39 5f d2 8d 7d 05 e1 a3 88 68 f3 b9 c5
Start round 10: 4a 82 48 51 c5 7e 7e 47 64 3d e5 0c 2a f3 e8 c9
Start round 11: c1 49 07 f6 ca 3b 3a a0 70 e9 aa 31 3b 52 b5 ec
Start round 12: 5f 9c 6a bf ba c6 34 aa 50 40 9f a7 66 67 76 53
Start round 13: 51 66 04 95 43 53 95 03 14 fb 86 e4 01 92 25 21
Start round 14: 62 7b ce b9 99 9d 5a aa c9 45 ec f4 23 f5 6d a5
Ciphertext:     8e a2 b7 ca 51 67 45 bf ea fc 49 90 4b 49 60 89
Decrypting ...
Ciphertext:     8e a2 b7 ca 51 67 45 bf ea fc 49 90 4b 49 60 89
Key:            00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
                10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f
Start round  1: aa 5e ce 06 ee 6e 3c 56 dd e6 8b ac 26 21 be bf
```

```
Start round  2: d1 ed 44 fd 1a 0f 3f 2a fa 4f f2 7b 7c 33 2a 69
Start round  3: cf b4 db ed f4 09 38 08 53 85 02 ac 33 de 18 5c
Start round  4: 78 e2 ac ce 74 1e d5 42 51 00 c5 e0 e2 3b 80 c7
Start round  5: d6 f3 d9 dd a6 27 9b d1 43 0d 52 a0 e5 13 f3 fe
Start round  6: be b5 0a a6 cf f8 56 12 6b 0d 6a ff 45 c2 5d c4
Start round  7: f6 e0 62 ff 50 74 58 f9 be 50 49 76 56 ed 65 4c
Start round  8: d2 2f 0c 29 1f fe 03 1a 78 9d 83 b2 ec c5 36 4c
Start round  9: 2e 6e 7a 2d af c6 ee f8 3a 86 ac e7 c2 5b a9 34
Start round 10: 9c f0 a6 20 49 fd 59 a3 99 51 89 84 f2 6b e1 78
Start round 11: 88 db 34 fb 1f 80 76 78 d3 f8 33 c2 19 4a 75 9e
Start round 12: ad 9c 7e 01 7e 55 ef 25 bc 15 0f e0 1c cb 63 95
Start round 13: 84 e1 fd 6b 1a 5c 94 6f df 49 38 97 7c fb ac 23
Start round 14: 63 53 e0 8c 09 60 e1 04 cd 70 b7 51 ba ca d0 e7
Plaintext:      00 11 22 33 44 55 66 77 88 99 aa bb cc dd ee ff
```

Plaintest and Key all zeros, **128**-bit key

```
Encrypting ...
Plaintext:      00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Key:            00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Start round  1: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Start round  2: 01 00 00 00 01 00 00 00 01 00 00 00 01 00 00 00
Start round  3: c6 e4 e4 8b a4 87 87 e8 c6 e4 e4 8b a4 87 87 e8
Start round  4: 28 2d f3 c4 6a f3 86 25 4a 4e 90 a7 08 90 e5 46
Start round  5: ab d2 cd fe 37 5a b5 49 50 a0 af c0 75 9a 6a 5f
Start round  6: d4 6f 4f 6c 55 b8 96 33 7e 05 bb 3d 79 79 de 23
Start round  7: 04 f2 ca 97 07 78 28 45 e2 2f 01 96 49 c5 d7 10
Start round  8: b7 aa e4 c5 1d 25 2d 4f 6c 92 0f 81 94 e5 81 50
Start round  9: 23 e7 8c 3c 13 21 63 db aa c0 c6 57 2e 03 cb 95
Start round 10: 7f fe 0e 95 51 a5 66 35 0e 34 7c 47 29 29 ec cb
Ciphertext:     66 e9 4b d4 ef 8a 2c 3b 88 4c fa 59 ca 34 2b 2e
Decrypting ...
Ciphertext:     66 e9 4b d4 ef 8a 2c 3b 88 4c fa 59 ca 34 2b 2e
Key:            00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Start round  1: d2 06 10 1f d1 18 ce 2a ab a5 ab 96 a5 bb 33 a0
Start round  2: 26 fd b4 2a 7d ba 1f eb ac 7b 64 b9 31 94 fb 5b
Start round  3: a9 3f 76 53 a4 4f 0c a6 50 d9 69 84 22 ac d8 0c
Start round  4: f2 bc 7c ca c5 15 0e 88 98 a6 74 6e 3b 89 34 90
Start round  5: 48 6c ea 26 fc 6b 1d 50 f3 b6 84 c3 b6 a8 90 27
Start round  6: 62 be 79 cf 9a e0 02 bb 53 b8 bd 3b 9d b5 d5 ba
Start round  7: 34 0d 60 5a 02 2f d9 1c d6 60 0d 3f 30 d8 44 5c
Start round  8: b4 17 69 9b 49 69 17 3d b4 17 69 9b 49 69 17 3d
Start round  9: 7c 63 63 63 7c 63 63 63 7c 63 63 63 7c 63 63 63
Start round 10: 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63
Plaintext:      00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Plaintest all zeros, key a single 1, **128**-bit key

```
Encrypting ...
Plaintext:      00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Key:            00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 01
Start round  1: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 01
Start round  2: 1e 1f 3e 3e 01 00 1f 00 01 00 1f 00 01 00 1f 01
Start round  3: 66 e0 d8 04 d6 43 e3 f2 17 67 13 5a f9 cf 28 e9
Start round  4: 7e 41 45 5e 09 86 08 b5 a1 69 f0 da 70 61 e1 bf
```

```
Start round  5: 79 89 5e 0d d2 19 17 93 a2 96 41 74 c2 e1 0d 64
Start round  6: a0 f7 36 25 ca ae 92 22 a4 76 49 4b 04 d0 d6 8b
Start round  7: a8 1b dd b6 3b f7 72 10 81 6b 51 a9 70 27 04 e1
Start round  8: 89 51 a6 38 37 41 9e 27 9c 5d fe a7 d0 20 3c 26
Start round  9: 4c 39 fa 23 09 9f 8d 0b 53 f7 13 4f 0a 53 02 53
Start round 10: dd 9d a7 03 9c a1 f1 58 42 43 46 94 5d c8 68 7a
Ciphertext:     05 45 aa d5 6d a2 a9 7c 36 63 d1 43 2a 3d 1c 84
Decrypting ...
Ciphertext:     05 45 aa d5 6d a2 a9 7c 36 63 d1 43 2a 3d 1c 84
Key:            00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 01
Start round  1: c1 32 5a da de 1a 45 7b 2c e8 5c 6a 4c 5e a1 22
Start round  2: 29 db 7d ed 01 68 77 26 ed ed 2d 2b 67 12 5d 84
Start round  3: a7 83 bb f7 9a 4c eb 07 de b7 24 cc 70 d1 0b 5c
Start round  4: c2 68 d1 f8 e2 7f f2 4e 0c cc c1 ca 51 af 40 d3
Start round  5: e0 e4 3b 3d 74 38 f6 3f 49 70 05 93 f2 68 4f b3
Start round  6: b6 d4 83 43 b5 90 d7 d7 3a f8 58 dc 25 a7 f0 92
Start round  7: f3 44 8c 08 01 f9 f8 58 32 ef 6e d5 51 83 30 57
Start round  8: 33 1a 7d 1e f6 85 34 f2 f0 8a 61 89 99 e1 11 be
Start round  9: 72 63 c0 7c 7c 63 c0 b2 7c 63 b2 63 7c c0 c0 63
Start round 10: 63 63 63 7c 63 63 63 63 63 63 63 63 63 63 63 63
Plaintext:      00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

```
Plaintest a single 1, key all zeros, 128-bit key
Encrypting ...
Plaintext:      00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 01
Key:            00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Start round  1: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 01
Start round  2: 1e 1f 21 3e 01 00 00 00 01 00 00 00 01 00 00 00
Start round  3: da ea ea 99 75 56 ef 51 58 5d c3 15 5a da 24 4b
Start round  4: 6b 78 40 4b 44 dd 5b fb 89 73 17 e4 78 87 db aa
Start round  5: 14 47 70 88 31 17 26 e7 1f 03 f1 40 92 6c 27 a3
Start round  6: 30 dd 66 d8 e8 fe 56 7f 23 dd e9 d2 68 54 93 04
Start round  7: de d8 c5 d3 dc 39 2f e5 54 1e 04 f1 bc 15 a2 69
Start round  8: 26 b8 f7 a9 e8 1a 48 1e 78 15 35 0a 1b 22 26 19
Start round  9: 44 24 06 b6 d9 51 38 9d 40 4d 63 c4 95 fb 34 85
Start round 10: 83 f7 5d d0 88 f3 8d e2 2f 83 89 19 59 35 7f 36
Ciphertext:     58 e2 fc ce fa 7e 30 61 36 7f 1d 57 a4 e7 45 5a
Decrypting ...
Ciphertext:     58 e2 fc ce fa 7e 30 61 36 7f 1d 57 a4 e7 45 5a
Key:            00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Start round  1: ec 0d a7 05 c4 ec d2 70 15 96 4c 98 cb 68 5d d4
Start round  2: 1b d1 fb 97 35 e3 18 4e 09 0f 6f 5e 2a 36 07 1c
Start round  3: f7 a2 96 d4 9b 59 f7 d3 bc 93 68 72 af 6c 52 67
Start round  4: 1d 12 f2 f9 86 72 3a 66 20 59 a6 d9 65 61 15 a1
Start round  5: 04 bb 1e f2 9b c1 dc 61 26 20 33 d2 45 c1 b1 b5
Start round  6: fa f0 a1 0a c7 7b cc c4 c0 50 51 94 4f a0 f7 09
Start round  7: 7f c1 f0 ac 1b 8f b9 b3 a7 17 09 0f bc bc 39 69
Start round  8: 57 b1 2e b3 9d 4c 36 ee 6a 57 87 d1 be 87 df 59
Start round  9: 72 63 63 63 7c 63 63 b2 7c 63 fd 63 7c c0 63 63
Start round 10: 63 63 63 7c 63 63 63 63 63 63 63 63 63 63 63 63
Plaintext:      00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 01
```

Program X.28.a
Shamir's Threshold Scheme

Referred to from page 157.

The code below implements Shamir's threshold scheme using 6 Java classes:

- **`CreateThreshold`**: This class uses parameters fed into the constructor to create a new threshold scheme. Inputs are the secret **`s`**, the threshold value **`t`**, the number of users **`n`**, and the prime **`p`**. It first creates a random polynominal of degree **`t-1`** by choosing the coefficients at random. (The random number generator used is just Java's **`Math.random()`**, so an actual production system would need a better generator.) Then the class evaluates the polynomial at successive positive integers to create the shares.

- **`Secret`**: This class also uses parameters fed into the constructor to recover the secret value. The input parameters are the threshold value **`t`**, the prime **`p`**, and x and y coordinates for **`t`** shares. There is also a boolean parameter **`debug`** that will produce debug output if it is **`true`**. After calculating the secret, the method **`getSecret()`** will return it.

- **`NewThreshold`**: The main of this class uses command line parameters to create a new threshold scheme. On the command line are: the secret **`s`**, the threshold value **`t`**, the number of users **`n`**, and the prime **`p`**. The class supplies arrays into which the **`n`** shares are placed by the **`createThreshold`** class. The class finally writes the parameters (on the first line) and the shares (one to a line) to the standard output as **`4 + 2*n`** integers.

- **`RecoverSecret`**: The main of this class reads integers from the standard input, first the threshold value **`t`** and the prime **`p`**, and then **`t`** shares (x and y coordinates). The class finally writes the secret to the standard output.

- **`ThresholdTest`**: This class has a main that creates instances of **`CreateThreshold`** and **`Secret`** in debug mode. Using the same 4 command line parameters as **`NewThreshold`**, it creates a threshold scheme instance, providing debug output showing the **`n`** shares. Then **`t`** of these shares are chosen at random for input to the **`Secret`** class, which also provides debug output.

- **`GetNext`**: A class to read **`int`**s from the standard input. The integers are delimited by any non-digit characters, which are ignored.

Java class: CreateThreshold

```
// CreateThreshold: Shamir's threshold scheme: return shares
public class CreateThreshold {
   long s; // "secret"
   int  t; // threshold value, t <= n
   int  n; // number of users, n >= 2
```

```
long p; // prime for system, p > S, p > t
long[] X; // array of n shares, x coordinates
long[] Y; // array of n shares, y coordinates
long[] A; // array of t random coefficients for polynomial
boolean debug = false; // output debug information

// CreateThreshold: constructor, does most of the work
public CreateThreshold(long s1, int t1, int n1, long p1,
       long[] X1, long[] Y1, boolean d) {
   s = s1; t = t1; n = n1; p = p1; debug = d;
   if ( n < 2 || t > n || p <= s || p <= t) {
      System.out.println("Parameter out of range");
      System.exit(1);
   }
   X = X1;
   Y = Y1;
   A = new long[t];
   createF(); // puts random coefficients into A, to create poly
   if (debug) { // printout for debugging and demonstrations
      System.out.print("New (" + t + "," + n + ") threshold scheme,");
      System.out.println(" with p = " + p + " and s = " + s);
      System.out.print("Function f(x) = ");
      for (int i = 0; i < t; i++) {
         System.out.print(A[i] + "*x^" + i + " ");
         if (i != t-1) System.out.print("+ ");
      }
      System.out.println();
   }
   createShares(); // use poly to create shares
   if (debug) { // more debug printout
      System.out.print("All " + n + " Output Shares: ");
      for (int i = 0; i < n; i++)
         System.out.print("(" + X[i] + "," + Y[i] + ") ");
      System.out.println("\n");
   }
}

// evalF: evaluate the function f
private long evalF(long x) {
   long y = 0;
   for (int i = t - 1; i >= 0; i--) {
      y = y*x % p;
      y = (y + A[i]) % p;
   }
   return y;
}

// createF: create F with random coefficients
//   that is, load A with random coeffs
private void createF() {
   A[0] = s; // the secret
   for (int i = 1; i < t; i++)
      A[i] = randlong(0, p-1);
```

```
   }

   // randlong: return a random long x, a <= x <= b
   private long randlong(long a, long b) {
      return (long)( Math.random()*(b - a + 1) + a);
   }

   // createShares: load X and Y with (x,y) coords of each share
   public void createShares() {
      for(int i = 0; i < n; i++) {
         X[i] = i+1;
         Y[i] = evalF(i+1);
      }
   }
}
```

Java class: Secret

```
// Secret: Shamir's threshold scheme: return secret
public class Secret {
   long s; // "secret"
   int  t; // threshold value, t <= n
   long p; // prime for system, p > S, p > t
   long[] X; // array of t shares, x coordinates
   long[] Y; // array of t shares, y coordinates
   long[] C; // array of t coefficents
   boolean debug = false; // output debug information

   // Secret: constructor, does most of the work
   public Secret(int t1, long p1, long[] X1, long[] Y1, boolean d) {
      t = t1; p = p1; debug = d; X = X1; Y = Y1;
      if (p <= t) {
         System.out.println("Parameter out of range");
         System.exit(1);
      }
      if (debug) {
         System.out.print("Recover secret from t = " + t);
         System.out.println(" shares, with p = " + p);
         System.out.print("All " + t + " Input Shares: ");
         for (int i = 0; i < t; i++)
            System.out.print("(" + X[i] + "," + Y[i] + ") ");
         System.out.println();
      }
      C = new long[t];
      createC();
   }

   // getSecret: return the secret value
   public long getSecret() {
      return s;
   }

   // createC: do the calculation of the secret
   //  Note: interspresed debug output
```

```
    private void createC() {
       for (int i = 0; i < t; i++) { // calculate C[i]
          if (debug) System.out.print("C[" + i + "] = ");
          C[i] = 1;
          for (int j = 0; j < t; j++)
             if (i != j) {
                if (debug)
                    System.out.print(X[j] + "/(" + X[j] + "-" +
                                                   X[i] + ") ");
             long term = modP(X[j]*invModP(modP(X[j] - X[i])));
             if (debug)
                 System.out.print(" ( or " + term + ") ");
             C[i] = modP(C[i]*term );
          }
          if (debug) System.out.println("= " + C[i]);
       }
       s = 0;
       if (debug) System.out.print("Secret = ");
       for (int i = 0; i < t; i++) {
          if (debug)
             System.out.print(C[i] + "*" + Y[i] + " ");
          if (i != t-1) if (debug) System.out.print("+ ");
          s = modP(s + C[i]*Y[i]);
       }
       if (debug) System.out.println("= " + s);
    }

    // modP: does actual x mod p, even if x < 0
    private long modP(long x) {
       long y = x%p;
       if (y < 0) y = y + p;
       return y;
    }

    // invModP: calculate an inverse value mod p
    private long invModP(long x) {
       long[] res = new long[3];
       res = GCD(x, p);
       return modP(res[0]);
    }

    // GCD: extended GCD algorithm
    private static long[] GCD(long x, long y) {
       long[] u = {1, 0, x}, v = {0, 1, y}, t = new long[3];
       while (v[2] != 0) {
          long q = u[2]/v[2];
          for (int i = 0; i < 3; i++) {
             t[i] = u[i] -v[i]*q; u[i] = v[i]; v[i] = t[i];
          }
       }
       return u;
    }
}
```

Java class: NewThreshold

```
// NewThreshold: create instance of Shamir's threshold scheme
//  Input (on command line):
//    s (the secret), t (threshold value), n (# users), p (prime)
//  Output (in System.out):
//    s  t  n  p
//    n (x,y) threshold pairs, for the n users
public class NewThreshold {

   public static void main(String[] args) {
      long s =    Long.parseLong(args[0]); // "secret"
      int  t = Integer.parseInt (args[1]); // threshold, t <= n
      int  n = Integer.parseInt (args[2]); // number of users, n >= 2
      long p =    Long.parseLong(args[3]); // prime , p > S, p > t
      long[] X = new long[n];
      long[] Y = new long[n];
      CreateThreshold cT = new CreateThreshold(s, t, n, p, X, Y, false);
      System.out.println(s + " " + t + " " + n + " " + p);
      for (int i = 0; i < n; i++)
         System.out.println(X[i] + " " + Y[i]);
   }
}
```

Java class: RecoverSecret

```
// RecoverSecret: read t shares and recover the secret
//  Input (in System.in):
//    s  t
//    t (x,y) threshold pairs
//  Output (in System.out):
//    s (the secret)
public class RecoverSecret {

   public static void main(String[] args) {
      long s; // "secret"
      int  t; // threshold, t <= n
      long p; // prime , p > S, p > t
      GetNext getNext = new GetNext();
      t =       getNext.getNextInt();
      p = (long)getNext.getNextInt();
      long[] X = new long[t];
      long[] Y = new long[t];
      for (int i = 0; i < t; i++) {
         X[i] = (long)getNext.getNextInt();
         Y[i] = (long)getNext.getNextInt();
      }
      Secret secret = new Secret(t, p, X, Y, false);
      s = secret.getSecret();
      System.out.println(s);
   }
}
```

Java class: ThresholdTest

```
// ThresholdTest: Test Shamir's threshold scheme, produce debug output
public class ThresholdTest {
```

```
   public static void main(String[] args) {
      long s =    Long.parseLong(args[0]); // "secret"
      int  t = Integer.parseInt (args[1]); // threshold, t <= n
      int  n = Integer.parseInt (args[2]); // number of users, n >= 2
      long p =    Long.parseLong (args[3]); // prime, p > S, p > t
      long[] X = new long[n];
      long[] Y = new long[n];
      CreateThreshold cT = new CreateThreshold(s, t, n, p, X, Y, true);
      long[] Xs = new long[t];
      long[] Ys = new long[t];
      // choose t of the n shares at random
      int[] select = new int[n];
      for (int i = 0; i < n; i++) select[i] = i;  // indexes of shares
      for (int i = 0; i < t; i++) { // interchange first t at random
         int j =  (int)(Math.random()*((n-1) - i + 1) + i);
         int temp = select[i];
         select[i] = select[j];
         select[j] = temp;
      }
      for (int i = 0; i < t; i++) {
         Xs[i] = X[select[i]];
         Ys[i] = Y[select[i]];
      }
      Secret secret = new Secret(t, p, Xs, Ys, true);
      s = secret.getSecret();
      System.out.println("Secret: " + s);
   }
}
```

Java class: GetNext

```
// GetNext: fetch next char or unsigned integer from System.in
import java.io.*;
public class GetNext {
   private Reader in; // internal file name for input stream

   // GetNext: constructor
   public GetNext () {
      in = new InputStreamReader(System.in);
   }

   // getNextChar: fetches next char
   private char getNextChar() {
      char ch = ' '; // = ' ' to keep compiler happy
      try {
         ch = (char)in.read();
      } catch (IOException e) {
         System.out.println("Exception reading character");
      }
      return ch;
   }

   // getNextInt: fetch unsigned int
   public int getNextInt() {
```

```
      String s ;
      char ch;
      while (!Character.isDigit(ch = getNextChar()))
         ;
      s = "" + ch;
      while (Character.isDigit(ch = getNextChar()))
         s += ch;
      return Integer.parseInt(s);
   }
}
```

Here are two runs testing the threshold scheme implementation:

```
% java ThresholdTest 1111 6 9 1999
New (6,9) threshold scheme, with p = 1999 and s = 1111
Function f(x) = 1111*x^0 + 1981*x^1 + 196*x^2 + 961*x^3 + 288*x^4 +
   1696*x^5
All 9 Output Shares: (1,236) (2,461) (3,456) (4,1049) (5,850) (6,1870)
   (7,1147) (8,363) (9,1468)

Recover secret from t = 6 shares, with p = 1999
All 6 Input Shares: (6,1870) (8,363) (4,1049) (7,1147) (5,850) (2,461)
C[0] = 8/(8-6)  ( or 4) 4/(4-6)  ( or 1997) 7/(7-6)  ( or 7) 5/(5-6)
   ( or 1994) 2/(2-6)  ( or 999) = 1859
C[1] = 6/(6-8)  ( or 1996) 4/(4-8)  ( or 1998) 7/(7-8)  ( or 1992)
   5/(5-8)  ( or 1331) 2/(2-8)  ( or 666) = 1321
C[2] = 6/(6-4)  ( or 3) 8/(8-4)  ( or 2) 7/(7-4)  ( or 1335) 5/(5-4)
   ( or 5) 2/(2-4)  ( or 1998) = 1929
C[3] = 6/(6-7)  ( or 1993) 8/(8-7)  ( or 8) 4/(4-7)  ( or 665) 5/(5-7)
   ( or 997) 2/(2-7)  ( or 1199) = 64
C[4] = 6/(6-5)  ( or 6) 8/(8-5)  ( or 669) 4/(4-5)  ( or 1995) 7/(7-5)
   ( or 1003) 2/(2-5)  ( or 1332) = 1482
C[5] = 6/(6-2)  ( or 1001) 8/(8-2)  ( or 1334) 4/(4-2)  ( or 2) 7/(7-2)
   ( or 801) 5/(5-2)  ( or 668) = 1342
Secret = 1859*1870 + 1321*363 + 1929*1049 + 64*1147 + 1482*850 +
   1342*461 = 1111
Secret: 1111
% java ThresholdTest 444444444 4 6 536870909
New (4,6) threshold scheme, with p = 536870909 and s = 444444444
Function f(x) = 444444444*x^0 + 321956576*x^1 + 166564884*x^2 +
   237875836*x^3
All 6 Output Shares: (1,97099922) (2,436398366) (3,205240247)
   (4,294009672) (5,519348930) (6,161029401)

Recover secret from t = 4 shares, with p = 536870909
All 4 Input Shares: (3,205240247) (4,294009672) (6,161029401)
   (5,519348930)
C[0] = 4/(4-3)  ( or 4) 6/(6-3)  ( or 2) 5/(5-3)  ( or 268435457) = 20
C[1] = 3/(3-4)  ( or 536870906) 6/(6-4)  ( or 3) 5/(5-4)  ( or 5) =
   536870864
C[2] = 3/(3-6)  ( or 536870908) 4/(4-6)  ( or 536870907) 5/(5-6)
   ( or 536870904) = 536870899
```

```
C[3] = 3/(3-5)  ( or 268435453) 4/(4-5)  ( or 536870905) 6/(6-5)
   ( or 6) = 36
Secret = 20*205240247 + 536870864*294009672 + 536870899*161029401 +
   36*519348930 = 444444444
Secret: 444444444
```

The following code illustrates the acutal threshold schemes, without the debug information. First using standard input and output:

```
% java NewThreshold 2222222 5 8 10316017
2222222 5 8 10316017
1 9512402
2 8010272
3 7372056
4 8834487
5 5214807
6 1542801
7 4744780
8 3011547
% java RecoverSecret
5 10316017
3 7372056
4 8834487
7 4744780
8 3011547
2 8010272
2222222
```

Next using redirected files in Unix:

```
% java NewThreshold 2222222 5 8 10316017 > thresh.txt
% cat thresh.txt
2222222 5 8 10316017
1 1769097
2 766836
3 4599213
4 4208181
5 7041923
6 6106801
7 599390
8 6222495
% cat secret.txt
5 10316017
3 4599213
6 6106801
2 766836
7 599390
1 1769097
% java RecoverSecret < secret.txt
2222222
```

Appendices

A
The Laws of Cryptography
Using Printed Log Tables

This is a lesson from prehistoric times. It brings back nostalgic memories. Before calculators, one used *printed tables* to carry out calculations. The example in the main section was to calculate `23.427 * 23.427 * 3.1416`. To do this, one first needed the logarithms (base 10) of the two numbers. In colored bold italic below are the actual table entries (using a book of tables dating from 1957) — everything else you had to do mentally or on paper:

Number	Log	Explanation
2342	***36959***	using interpolation entry:
23427	369716	***7***th entry under ***18*** is ***12.6***
2343	***36977***	take `369590 + 126` to get `369716`

This means that `log(2.3427) = 0.369716` approximately. Then `log(23.427) = log(2.3427 * 10) = log(2.3427) + log(10) = 0.369716 + 1 = 1.369716`

Similarly, look up `3.1416`:

Number	Log	Explanation
3141	***49707***	using interpolation entry:
31416	497154	***6***th entry under ***14*** is ***8.4***
3142	***49721***	take `497070 + 84` to get `497154`

This means that `log(3.1416) = 0.497154` approximately.

Form the sum: `1.369716 + 1.369716 + 0.497154 = 3.236586` (this must be done by hand, with pencil and paper).

Now finally, one has to look up the "anti-log" in the same table:

Number	Log	Explanation
1724	***23654***	using interpolation entry:
17242	23659	***2***nd entry under ***25*** is ***5.0***
1725	***23679***	take `17240 + 2` to get `17242`

This means that `log(1.7242) = 0.23659` approximately, so `3.23659 = 3 + 0.23659 = log(1000) + log(1.7242) = log(1000 * 1.7242) = log(1724.2)`, or (finally), the answer is `1724.2` approximately. So the area of a circle of radius `23.427` is approximately `1724.2`.

All this pain just to multiply 3 numbers together, to get 4 or 5 digits of accuracy in the answer. The next two lessons from our primitive ancestors: how to use tables of the *logarithms* of trig functions (to save one lookup), and how to use a slide rule. (Just kidding.)

B
The Laws of Cryptography
Unsigned bytes in Java

The Advanced Encryption Standard (AES) makes extensive use of the **`unsigned byte`** type. This creates awkward code in Java, because Java supports only the *signed* **`byte`** type. In addition, some of the Java operators do not work as in the documentation, creating further problems. For example:

```
public class TestRightShift0 {
   public static void main(String[] args) {
      byte b = (byte)0x80;
      int c = b >>> 4; // <---------------------------------
      System.out.println("b: " + b + ", c: " + c);
      c = 0x0ffffff8;
      System.out.println("c: " + c);
   }
}
/* Output:
b: -128, c: 268435448
c: 268435448
*/
```

The output shows that b has the value **`0x80`** or **`1000 0000`** in binary, as one would expect. According to *The Java Programming Language*, *Third Edition*, page 164, the **`>>>`** operator should fill new high-order bits with zeros. In fact, though, Java is converting **`b`** to **`int`** type with sign-extended value **`0xffffff80`**, right shifting *this* and putting just four zeros at the right, to give **`0x0ffffff8`**. To get the desired value, one can use either of the following:

```
public class TestRightShift1 {
   public static void main(String[] args) {
      byte b = (byte)0x80;
      int  c = (b & 0xf0) >> 4;} // <---------------------------------
      System.out.println("b: " + b + ", c: " + c);
           c = (b >> 4) & 0xf;}  // <---------------------------------
      System.out.println("b: " + b + ", c: " + c);
   }
}
/* Output:
b: -128, c: 8
b: -128, c: 8
*/
```

Similarly, a right shift of **`3`** could use either of the lines:

```
public class TestRightShift2 {
   public static void main(String[] args) {
      byte b = (byte)0x80;
      int  c = (b & 0xf1) >> 3; // <---------------------------------
```

```
         System.out.println("b: " + b + ", c: " + c);
              c = (b >> 3) & 0x1f; // <---------------------------------
         System.out.println("b: " + b + ", c: " + c);
      }
   }
   /* Output:
   b: -128, c: 16
   b: -128, c: 16
   */
```

This seems to require different constants for different shifts, but actually, the first method works with a fixed constant:

```
public class TestRightShift3 {
   public static void main(String[] args) {
      byte b = (byte)0x80;
      for (int i = 0; i < 9; i++) {
         int c = (b & 0xff) >> i; // <---------------------------------
         System.out.println("b: " + b + ", shift by: " + i + ", c: " + c);
      }
   }
}
/* Output:
b: -128, shift by: 0, c: 128
b: -128, shift by: 1, c: 64
b: -128, shift by: 2, c: 32
b: -128, shift by: 3, c: 16
b: -128, shift by: 4, c: 8
b: -128, shift by: 5, c: 4
b: -128, shift by: 6, c: 2
b: -128, shift by: 7, c: 1
b: -128, shift by: 8, c: 0
*/
```

Law JAVA-BYTES-1:

In the Java language, to right shift an integer amount *shiftAmount*, use the code

int shiftedValue = (byteValue & 0xff) >> shiftAmount;

where *byteValue* is of type *byte* and *shiftAmount* is an *int* in the range from *0* to *8*. A *0* for *shiftAmount* is the same as not doing the shift, but just to store an unsigned *byte* into an *int* type requires

int shiftedValue = byteValue & 0xff;

Left shifts work as they ought to, but the result is an **`int`**, so it needs to be cast to a **`byte`** if that is needed.

```
public class TestLeftShift {
   public static void main(String[] args) {
```

```
      byte b = (byte)0x01;
      for (int i = 0; i < 9; i++) {
         int   c = (b << i);}        // <----------------------------------
         System.out.println("b: " + b + ", shift by: " + i + ", c: " + c);
         byte bb = (byte)(b << i);}  // <----------------------------------
         System.out.println("b: " + b + ", shift by: " + i + ", bb: " + bb);
      }
   }
}
/* Output:
b: 1, shift by: 0, c: 1
b: 1, shift by: 0, bb: 1
b: 1, shift by: 1, c: 2
b: 1, shift by: 1, bb: 2
b: 1, shift by: 2, c: 4
b: 1, shift by: 2, bb: 4
b: 1, shift by: 3, c: 8
b: 1, shift by: 3, bb: 8
b: 1, shift by: 4, c: 16
b: 1, shift by: 4, bb: 16
b: 1, shift by: 5, c: 32
b: 1, shift by: 5, bb: 32
b: 1, shift by: 6, c: 64
b: 1, shift by: 6, bb: 64
b: 1, shift by: 7, c: 128
b: 1, shift by: 7, bb: -128
b: 1, shift by: 8, c: 256
b: 1, shift by: 8, bb: 0
*/
```

Law JAVA-BYTES-2:

In the Java language, logical and shifting operators work as follows:

- **All operators return an *int*, so they must be cast to a *byte* if a *byte* is needed. This includes: *&*, >>, >>>, and <<.**
- **Hex constants such as *0xff* actually define an integer, so this is the same as *0x000000ff*. For values bigger than *Ox7f* a cast to *byte* is needed.**
- **Arithmetic (except for / and %) with Java's signed *byte*s works just as if the *byte*s were unsigned, since there is no overflow.**

C
The Laws of Cryptography
Projects

Project 1: Designing a Stream Cipher.

For this project you are to start with a very simple cryptosystem based on the exclusive-or of a pseudo-random number sequence with plain text to produce ciphertext, followed by the same exclusive-or to transform ciphertext back into plaintext. This is the simplest possible stream cipher.

As a stream cipher, an unending stream of bits is generated from a key (or seed to the random number generator) and is xored with successive message bits to form the ciphertext. Everything can be thought of occurring one bit at a time.

The code below uses the Java **`Random`** class for the pseudo-random number generator. For convenience of the Java input function, the encryption is done byte at a time, using 8 bits from the generator at each step. Despite the byte orientation, it is still essentially a stream cipher, with encryption occurring one bit at a time.

Since the generator is of the multiplicative linear congruence type, one needs to extract the *high order* 8 bits, rather than low order bits. (Knuth notes that with these generators, the high-order bits are much more random than low-order ones.)

Here is the program. I deliberately wrote this using as few lines of Java as possible (well, except for a few extra variables).

Java class: Crypto

```
// Cipher.java: simple cipher
import java.io.*;
import java.util.Random;
class Cipher {
   public static void main(String[] args) throws IOException {
      long seed = Long.parseLong(args[0]);
      Random rand = new Random(seed);
      int ch;
      while ((ch = System.in.read()) != -1) {
         double x = rand.nextDouble();
         int m = (int)(256.0*x);
         ch = (ch ^ m);
         System.out.write(ch);
      }
      System.out.close();
   }
```

```
}
```

Here is a sample run, starting with a binary PDF file **utsa.pdf**. Using Unix redirection, the encrypted output goes to a binary file **utsa.binary** of the same size. Finally, another run using the same key recovers the original file, this time named **utsa2.pdf**. Notice that the key is the **long** with value **98765432123456789**, nearly a full 64-bit integer.

```
% pandora% javac Cipher.java
% java Cipher 98765432123456789 < utsa.pdf > utsa.binary
% java Cipher 98765432123456789 < utsa.binary > utsa2.pdf
% ls -l
total 18
-rw-r--r--    1 wagner   faculty       769 May 26 22:33 Cipher.class
-rw-r--r--    1 wagner   faculty       462 May 26 20:34 Cipher.java
-rw-r--r--    1 wagner   faculty      3116 May 26 22:34 utsa2.pdf
-rw-r--r--    1 wagner   faculty      3116 May 26 22:33 utsa.binary
-rw-r--r--    1 wagner   faculty      3116 May 26 16:32 utsa.pdf
```

Analysis of the initial system: The above system uses the **Random** class from Java as the random number generator (RNG). This class is provided for simulation and for other similar uses and is not intended to be a *cryptographically secure* RNG, where this means a RNG whose future outputs cannot be efficiently calculated from earlier outputs. In the case of the class **Random**, the multiplier and modulus are known, so any output immediately allows calculation of later outputs. Even if the multiplier and modulus are not known, given a sequence of integers produced by such a RNG, there are efficient (though difficult) algorithms to calculate them.

In the case above, one is far from knowing any of the integer outputs, however. One knows only a succession of 8-bit inital values from the floating point output of the generator. A known plaintext or chosen plaintext attack will produce such a sequence of these values immediately. I have no idea if there are efficient algorithms to deduce the portion of the RNG in use just from a succession of these 8-bit values. It most likely would be harder to break if one only had a succession of 1-bit values output by the RNG, the leading bits of the floating point outputs. (This step doesn't help against the brute-force attack mentioned in the next paragraph below.)

> **Project 1.1** Modify the code above so that at each stage 8 calls to the RNG produce 8 most significant bits that are assembled into an 8-bit byte for use in xor as in the original code.

The RNG **Random** is described as a "48-bit random number linear congruence generator." In this case, by investigating the details of the generator, one should be able to mount a brute-force attack requiring $2^{48} \doteq 2.81475 \times 10^{14}$ or about 300 trillion steps. One starts with a known sequence of outputs, either the first 8 bits or the first bit in each case. At each step, the generator needs to be exercised until that initial value is eliminated or until the search succeeds. This calculation easily allows parallelization. If one could complete a step each giga-second, it would still take on the average 150 000 seconds or just over 40 hours. In parallel 10 000 such processors would take 15 seconds. So it all depends on how much hardware you have!

Systems with a modified generator: One could try modifying the generator. For example, one could use *two* seeds and *two* separate copies of the **`Random`** RNG. The input seeds would each be 64 bit long, but the RNGs are actually 48 bits, for 96 bits total in two generators. How can one use two generators? Consider some possibilities.

- Alternate the use of the two generators, back and forth, or according to some scheme publicly known. (Argue that this is essentially no help at all.)
- In order to get 8 bits for each byte of source, take 4 bits from one generator and 4 bits from the other. (Argue that this is essentially no help at all.)
- How about just averaging two floating point outputs from the two generators? (What is the matter with this? Can it be made to work?)
- Here is a scheme of Knuth's to use two generators: Choose a value n for the address size of a buffer, say, $n = 10$ for a buffer of size $2^n = 1024$. Fill the buffer with 8-bit values from the first generator. Then get 10-bit values from the second generator, using this number to fetch the 8-bit value from the first generator at that address in the buffer, and then replacing the value in the buffer using the first generator again.

One could also use one of the two double-seeded RNGs described at the end of Section 16.2. In each case these are two 32-bit seeds for a total of 64 bits. Thus the brute-force attack takes $2^{64} \doteq 1.8447 \times 10^{19}$ or 20 billion billion steps. Finally one could use a more sophisticated RNG such as the one based on chaos theory in Chapter 17, or the perfect RNGs of Chapter (?19? in Section V). I have no proof of the difficulty of breaking any scheme of the type shown here, but even the very first piece of Java code seems like it would be quite hard to break, if it could be broken at all except by a brute-force search (an attack which always succeeds, though perhaps taking an unacceptable amount of time). Each refinement may make the system stronger; it's hard to imagine any way to break a system based on the RNG in Chapter 17, since no one has ever solved such 2-dimensional chaotic equations analytically.

> **Project 1.2** Modify the code above so it uses a better RNG with a longer total number of seed bits, or uses two or more RNGs.

The key is somewhat awkwardly handled up to now. It would be better to have an arbitrary character string as the input key. This string should be hashed into whatever actual inputs are needed for the random number generator. The final project could look somewhat like the Unix **`crypt`** utility.

> **Project 1.3** Look up the **`man`** page on **`crypt`** and modify the code above so it handles an arbitrary character string as the input key. It should be indistinguishable from **`crypt`**.

Project 2: Designing a Block Cipher.

This project asks you to work on a simple, symmetric, conventional, block-oriented cryptosystem. I am giving you a skeleton Java program below that implements the Caesar cipher on each byte, using a rotation from 0 to 255.
One could regard this project as a generalization of Project 1, but I mean for you to handle it as a block cryptosystem.

- You are to produce a cipher that uses *the same short secret key* for encryption and decryption. It would be fairly easy to modify the code below so that it implements the Beale cipher, but you are not supposed to do that, since the Beale cipher uses a key the same size as the plaintext.
- Your system should be a block cipher, so each ciphertext block should depend only on the key and the plaintext block. Two separate but identical plaintext blocks should encrypt to the same ciphertext block. If you wish to give your cipher some of the characteristics of a stream cipher, with state and without the property of the previous sentence, then you should just use the cipher in the *cipher block chaining (CBC) mode*.
- The idea is to try to implement something that might not be easy to break. For the strongest possible system, you could assume that opponents do not know your secret key, but
  - have access to your actual Java code, so that they know exactly what your algorithms are and how they work, and
  - have access to a "black" encryption/decryption box (without access to the secret key), so that they can carry out a *chosen plaintext* attack, that is, can choose arbitrary plaintexts and get the corresponding ciphertexts.
- The code below is strictly to get students started who might otherwise not know where to start. You are also encouraged to write your own Java skeleton, perhaps entirely different from the one below. In particular you do not have to get file names as command line parameters, but you could even have the file names hard-wired into your code.
- Specifically, the minimum you should do is to modify the methods **`encode`** and **`decode`** of the class **`Code`** so that they will reverse one another, and so they will make a cryptosystem harder to break than the Caesar cipher (hopefully *much* harder to break). These methods appear in **boldface** in the listing below.

A few hints:

You are welcome to ignore these hints if you want.

- Remember that a system can't possibly be strong unless there are at least 2^{64} or more keys. You can use anything you like for the key, but it must not be arbitrarily

long (as it is with the Beale cipher). However, making the key long is something you can also leave as a feature you might have included if you had had more time.

- One standard trick is to mess around with the key, and exclusive-or the result with a plaintext block. Then to reverse this, mess around with the key the same way and exclusive-or again. Notice that in this case you do not need to be able to reverse what you do to the key.
- Block ciphers often use combinations of three basic elements, repeated in *rounds* (to decrypt, the rounds are carried out in the opposite order):
  - Use the trick above: xor with something to encrypt and xor with the same thing during decryption.
  - Use a *permutation* of elements of the plaintext or of the key. Here the word permutation means a *rearrangement* of elements of the plaintext or of the key. Of course one uses the reverse of the rearrangement to decrypt.
  - Use a *substitution* of new items for elements of the plaintext or key. Here something completely different is inserted in place of each element. This operation also needs to be reversed for decryption.
- See another appendix at the end of this book for material about bit operations in Java. In particular, the result of `b1^b2` (both bytes) in Java is an **`int`** with the proper bits in the 8 least significant bits. You then need to cast to a byte (using **`(byte)`** to get a byte value for assignment or other uses.
- If your block size in only 8 (as in this case), this is also a weakness (why?). How might you attack any block cipher with block size of 8? Obviously you could increase the block size, but what else might you do to eliminate this weakness?
- You could increase the block size using the skeleton below by just handling more than one byte at a time, say 4, 8, or 16 bytes at a time. In this case you may want to pad the file at the end so that its length is a multiple of the block size.

Java Code for a Skeleton System:

The first class below (**`Crypto`**) just accesses command line arguments (which say whether to encrypt or decrypt, give the key, give the name of the input file, and give the name of the output file), reads the key, opens the files, creates an instance of the other class (**`Code`**) that does the work, and invokes a method in that class (**`transform`**).

Java class: Crypto

```
// Crypto: simple encode and decode of a file, byte-at-a-time
import java.io.*;
public class Crypto {
   static InputStream in; // input file args[2]
   static OutputStream out; // output file args[3]

   public static void openFiles(String infile, String outfile) {
      try {
```

```
            in =  new FileInputStream(infile);
            out = new FileOutputStream(outfile);
         } catch (IOException e) {
            System.err.println("Error opening files");
            System.exit(-1);
         }
      }

      public static void main(String[] args) {
         boolean encode = true; // encode or decode
         int key = 0;
         String usage =
            "Usage: java Crypto (-encode | -decode) key infile outfile";
         if (args.length != 4) {
            System.err.println(usage);
            System.exit(-1);
         }
         try {
            key = Integer.parseInt(args[1]);
         } catch (NumberFormatException e) {
            System.err.println("Error converting key \"" +
               args[1] + "\" to int");
            System.exit(-1);
         }
         if (args[0].equals("-encode")) encode = true;
         else if (args[0].equals("-decode")) encode = false;
         else {
            System.err.println(usage);
            System.exit(-1);
         }
         openFiles(args[2], args[3]);
         Code code = new Code(encode, key, in, out);
         code.transform();
      }
   }
```

The second class below (**Code**) actually reads and writes the files, byte-at-a-time. Between reading and writing a byte, it either encodes or decodes the byte, depending on the value of a boolean switch **encode**. As mentioned before, your main work (in a simple version of the assignment) could be to find more complicated functions to use for the methods **encodeByte** and **decodeByte**, but remember that they must be inverses of one another.

Java class: Code

```
// Code: encode or decode a file
import java.io.*;
public class Code {
   int key; // input key
   InputStream in; // input file
   OutputStream out; // output file
   boolean encode = false; // encode or decode

   // code: constructor, pass parameters
```

```
   public Code(boolean mode, int keyP,
         InputStream inP, OutputStream outP) {
      encode = mode;
      key = keyP;
      in =  inP;
      out = outP;
   }

   // transform: read bytes, encode or decode each byte, write
   public void transform() {
      try {
         int inB; // input byte
         int outB; // output byte
         // read input file, byte-at-a-time
         while ((inB = in.read()) != -1) { // till end-of-file
            // make a simple change
            if (encode) outB = encodeByte(key, inB);
            else        outB = decodeByte(key, inB);
            writeByte(outB);
         } // end of while
      } catch (IOException e) {
         System.err.println("Error reading file");
         System.exit(-1);
      } // end try
   }

   // encodeByte: encode a byte
   private int encodeByte(int key, int inB) {
      return (inB + key)%256; // encode
   }

   // decodeByte: decode a byte
   private int decodeByte(int key, int inB) {
      return (inB - key)%256; // decode
   }

   // writeByte: then write byte
   private void writeByte(int outB) {
      try {
         out.write(outB);
      } catch (IOException e) {
         System.err.print("Error writing file");
         System.exit(-1);
      }
   }
}
```

Here are the results of a simple run on a Unix box, using the JDK directly. User input is in **boldface**. Notice that after encoding and decoding, the original file is recovered.

```
% cat mess.text
Now is the time for all good men to come to the aid of their party
% java Crypto -encode 13 mess.text cipher2.text
```

```
% java Crypto -decode 13 cipher2.text mess2.text
% vi cipher2.text
[|x84-vx80-x81ur-x81vzr-s|^?-nyy-t||q-zr-x81|-p|zr-x81|
-x81ur-nvq-|s-x8
1urv^?-n^?x81x86W  (binary garbage)
% cat mess2.text
Now is the time for all good men to come to the aid of their party
```